aioamqp Documentation

Release 0.4.0

Benoît Calvez

August 19, 2015

Contents

Aioamqp is a library to connect to an amqp broker. It uses asyncio under the hood

CHAPTER 1

Limitations

For the moment, aioamqp is tested against Rabbitmq.

Contents:

1.1 Introduction

This is the documentation for the aioamqp module.

1.1.1 Prerequisites

Aioamqp works only with python >= 3.3 using asyncio library. If your are using Python 3.3 you'll have to install asyncio from pypi, but asyncio is now included in python 3.4 standard library.

1.1.2 Installation

You can install the most recent aioamqp release from pypi using pip or easy_install:

```
pip install aioamqp
easy_install aioamqp
```

1.2 Changelog

1.2.1 Next version (not yet released)

- Call the error callback on all circumtstances.

1.2.2 Aioamqp 0.3.0

- The consume callback takes now 3 parameters: body, envelope, properties, closes #33.
- Channel ids are now recycled, closes #36.

1.2.3 Aioamqp 0.2.1

- connect returns a transport and protocol instance.

1.2.4 Aioamqp 0.2.0

- Use a callback to consume messages.

1.3 API

1.3.1 Basics

There are two principal objects when using aioamqp:

- The protocol object, used to begin a connection to aioamqp,
- The channel object, used when creating a new channel to effectively use an AMQP channel.

1.3.2 Starting a connection

Starting a connection to AMQP really mean instanciate a new asyncio Protocol subclass:

```
import asyncio
import aioamqp

@asyncio.coroutine
def connect():
    try:
        transport, protocol = yield from aioamqp.connect()  # use default parameters
    except aioamqp.AmqpClosedConnection:
        print("closed connections")
        return

    print("connected !")
    yield from asyncio.sleep(1)

    print("close connection")
    yield from protocol.close()
    transport.close()

asyncio.get_event_loop().run_until_complete(connect())
```

In this example, we just use the method “start_connection” to begin a communication with the server, which deals with credentials and connection tunning.

1.3.3 Handling errors

The connect() method has an extra ‘on_error’ kwarg option. This on_error is a callback or a coroutine function which is called with an exception as the argument:

```
import asyncio
import aioamqp

@asyncio.coroutine
def error_callback(exception):
    print(exception)

@asyncio.coroutine
def connect():
    try:
        transport, protocol = yield from aioamqp.connect(
            host='nonexistant.com',
            on_error=error_callback,
        )
    except aioamqp.AmqpClosedConnection:
        print("closed connections")
        return

asyncio.get_event_loop().run_until_complete(connect())
```

1.3.4 Publishing messages

A channel is the main object when you want to send message to an exchange, or to consume message from a queue:

```
channel = yield from protocol.channel()
```

When you want to produce some content, you declare a queue then publish message into it:

```
queue = yield from channel.queue_declare("my_queue")
yield from queue.publish("aioamqp hello", '', "my_queue")
```

Note: we're pushing message to "my_queue" queue, through the default amqp exchange.

1.3.5 Consuming messages

When consuming message, you connect to the same queue you previously created:

```
import asyncio
import aioamqp

@asyncio.coroutine
def callback(body, envelope, properties):
    print(body)

channel = yield from protocol.channel()
yield from channel.basic_consume("my_queue", callback=callback)
```

The `basic_consume` method tells the server to send us the messages, and will call `callback` with amqp response arguments.

The `consumer_tag` is the id of your consumer, and the `delivery_tag` is the tag used if you want to acknowledge the message.

In the callback:

- the first `body` parameter is the message

- the `envelope` is an instance of envelope.Envelope class which encapsulate a group of amqp parameter such as:

```
consumer_tag
delivery_tag
exchange_name
routing_key
is_redeliver
```

- the `properties` are message properties, an instance of properties.Properties with the following members:

```
content_type
content_encoding
headers
delivery_mode
priority
correlation_id
reply_to
expiration
message_id
timestamp
type
user_id
app_id
cluster_id
```

1.3.6 Using exchanges

You can bind an exchange to a queue:

```
channel = yield from protocol.channel()
exchange = yield from channel.exchange_declare(exchange_name="my_exchange", type_name='fanout')
yield from channel.queue_declare("my_queue")
yield from channel.queue_bind("my_queue", "my_exchange")
```

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

a

A