asynqp Documentation

Release 0.1

Benjamin Hodgson

July 16, 2014

Contents

An AMQP (aka RabbitMQ) client library for `asyncio`.

CHAPTER 1

Example

```python
import asyncio
import asynqp


@asyncio.coroutine
def send_and_receive():
    # connect to the RabbitMQ broker
    connection = yield from asynqp.connect('localhost', 5672, username='guest', password='guest')

    # Open a communications channel
    channel = yield from connection.open_channel()

    # Create a queue and an exchange on the broker
    exchange = yield from channel.declare_exchange('test.exchange', 'direct')
    queue = yield from channel.declare_queue('test.queue')

    # Bind the queue to the exchange, so the queue will get messages published to the exchange
    yield from queue.bind(exchange, 'routing.key')

    # If you pass in a dict it will be automatically converted to JSON
    msg = asynqp.Message({'test_body': 'content'})
    exchange.publish(msg, 'routing.key')

    # Synchronously get a message from the queue
    received_message = yield from queue.get()
    print(received_message.json())  # get JSON from incoming messages easily

    # Acknowledge a delivered message
    received_message.ack()

    yield from connection.close()


if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    loop.run_until_complete(send_and_receive())
```

CHAPTER 2

Installation

```
pip install asynqp
```

CHAPTER 3

Table of contents

3.1 Reference guide

3.1.1 Connecting to the AMQP broker

asynqp.**connect**(*host='localhost'*, *port=5672*, *username='guest'*, *password='guest'*, *virtual_host='/'*, ***, *loop=None*, ***kwargs*)

Connect to an AMQP server on the given host and port.

Log in to the given virtual host using the supplied credentials. This function is a *coroutine*.

Parameters

- **host** (*str*) – the host server to connect to.
- **port** (*int*) – the port which the AMQP server is listening on.
- **username** (*str*) – the username to authenticate with.
- **password** (*str*) – the password to authenticate with.
- **virtual_host** (*str*) – the AMQP virtual host to connect to.

Further keyword arguments are passed on to `create_connection()`.

Returns the `Connection` object.

3.1.2 Managing Connections and Channels

Connections

class asynqp.**Connection**

Manage connections to AMQP brokers.

A `Connection` is a long-lasting mode of communication with a remote server. Each connection occupies a single TCP connection, and may carry multiple `Channels`. A connection communicates with a single virtual host on the server; virtual hosts are sandboxed and may not communicate with one another.

Applications are advised to use one connection for each AMQP peer it needs to communicate with; if you need to perform multiple concurrent tasks you should open multiple channels.

Connections are created using `asynqp.connect()`.

closed

a `Future` which is done when the handshake to close the connection has finished

open_channel()

Open a new channel on this connection.

This method is a *coroutine*.

Returns The new `Channel` object.

close()

Close the connection by handshaking with the server.

This method is a *coroutine*.

Channels

class asynqp.**Channel**

Manage AMQP Channels.

A Channel is a 'virtual connection' over which messages are sent and received. Several independent channels can be multiplexed over the same `Connection`, so peers can perform several tasks concurrently while using a single socket.

Channels are created using `Connection.open_channel()`.

declare_exchange(*name*, *type*, ***, *durable=True*, *auto_delete=False*, *internal=False*)

Declare an `Exchange` on the broker. If the exchange does not exist, it will be created.

This method is a *coroutine*.

Parameters

- **name** (*str*) – the name of the exchange.
- **type** (*str*) – the type of the exchange (usually one of `'fanout'`, `'direct'`, `'topic'`, or `'headers'`)
- **durable** (*bool*) – If true, the exchange will be re-created when the server restarts.
- **auto_delete** (*bool*) – If true, the exchange will be deleted when the last queue is un-bound from it.
- **internal** (*bool*) – If true, the exchange cannot be published to directly; it can only be bound to other exchanges.

Returns the new `Exchange` object.

declare_queue(*name=''*, ***, *durable=True*, *exclusive=False*, *auto_delete=False*)

Declare a queue on the broker. If the queue does not exist, it will be created.

This method is a *coroutine*.

Parameters

- **name** (*str*) – the name of the queue. Supplying a name of '' will create a queue with a unique name of the server's choosing.
- **durable** (*bool*) – If true, the queue will be re-created when the server restarts.
- **exclusive** (*bool*) – If true, the queue can only be accessed by the current connection, and will be deleted when the connection is closed.
- **auto_delete** (*bool*) – If true, the queue will be deleted when the last consumer is cancelled. If there were never any conusmers, the queue won't be deleted.

Returns The new `Queue` object.

close()

Close the channel by handshaking with the server.

This method is a *coroutine*.

3.1.3 Sending and receiving messages with Queues and Exchanges

Queues

class asynqp.**Queue**

Manage AMQP Queues and consume messages.

A queue is a collection of messages, to which new messages can be delivered via an `Exchange`, and from which messages can be consumed by an application.

Queues are created using `Channel.declare_queue()`.

name

the name of the queue

durable

if True, the queue will be re-created when the broker restarts

exclusive

if True, the queue is only accessible over one channel

auto_delete

if True, the queue will be deleted when its last consumer is removed

bind(*exchange*, *routing_key*)

Bind a queue to an exchange, with the supplied routing key.

This action 'subscribes' the queue to the routing key; the precise meaning of this varies with the exchange type.

This method is a *coroutine*.

Parameters

- **exchange** (*asynqp.Exchange*) – the `Exchange` to bind to
- **routing_key** (*str*) – the routing key under which to bind

Returns The new `QueueBinding` object

consume(*callback*, *, *no_local=False*, *no_ack=False*, *exclusive=False*)

Start a consumer on the queue. Messages will be delivered asynchronously to the consumer. The callback function will be called whenever a new message arrives on the queue.

This method is a *coroutine*.

Parameters

- **callback** (*callable*) – a callback to be called when a message is delivered. The callback must accept a single argument (an instance of `IncomingMessage`).
- **no_local** (*bool*) – If true, the server will not deliver messages that were published by this connection.
- **no_ack** (*bool*) – If true, messages delivered to the consumer don't require acknowledgement.
- **exclusive** (*bool*) – If true, only this consumer can access the queue.

Returns The newly created `Consumer` object.

`get`(***, *no_ack=False*)

Synchronously get a message from the queue.

This method is a *coroutine*.

Parameters **no_ack** (*bool*) – if true, the broker does not require acknowledgement of receipt of the message.

Returns an `IncomingMessage`, or `None` if there were no messages on the queue.

`purge`()

Purge all undelivered messages from the queue.

This method is a *coroutine*.

`delete`(***, *if_unused=True*, *if_empty=True*)

Delete the queue.

This method is a *coroutine*.

Parameters

- **if_unused** (*bool*) – If true, the queue will only be deleted if it has no consumers.
- **if_empty** (*bool*) – If true, the queue will only be deleted if it has no unacknowledged messages.

Exchanges

class `asynqp.`**`Exchange`**

Manage AMQP Exchanges and publish messages.

An exchange is a ‘routing node’ to which messages can be published. When a message is published to an exchange, the exchange determines which `Queue` to deliver the message to by inspecting the message’s routing key and the exchange’s bindings. You can bind a queue to an exchange, to start receiving messages on the queue, using `Queue.bind`.

Exchanges are created using `Channel.declare_exchange()`.

`name`

the name of the exchange.

`type`

the type of the exchange (usually one of `'fanout'`, `'direct'`, `'topic'`, or `'headers'`).

`publish`(*message*, *routing_key*, ***, *mandatory=True*)

Publish a message on the exchange, to be asynchronously delivered to queues.

Parameters

- **message** (*asynqp.Message*) – the message to send
- **routing_key** (*str*) – the routing key with which to publish the message

`delete`(***, *if_unused=True*)

Delete the exchange.

This method is a *coroutine*.

Parameters **if_unused** (*bool*) – If true, the exchange will only be deleted if it has no queues bound to it.

Bindings

class `asynqp.`**`QueueBinding`**

Manage queue-exchange bindings.

Represents a binding between a `Queue` and an `Exchange`. Once a queue has been bound to an exchange, messages published to that exchange will be delivered to the queue. The delivery may be conditional, depending on the type of the exchange.

QueueBindings are created using `Queue.bind()`.

`queue`
the `Queue` which was bound

`exchange`
the `Exchange` to which the queue was bound

`routing_key`
the routing key used for the binding

`unbind``()`
Unbind the queue from the exchange. This method is a coroutine.

Consumers

class `asynqp.`**`Consumer`**

A consumer asynchronously recieves messages from a queue as they arrive.

Consumers are created using `Queue.consume()`.

`tag`
A string representing the *consumer tag* used by the server to identify this consumer.

`callback`
The callback function that is called when messages are delivered to the consumer. This is the function that was passed to `Queue.consume()`, and should accept a single `IncomingMessage` argument.

`cancelled`
Boolean. True if the consumer has been successfully cancelled.

`cancel``()`
Cancel the consumer and stop recieving messages.

3.1.4 Message objects

class `asynqp.`**`Message`**`(`*body*, ***, *headers=None*, *content_type=None*, *content_encoding=None*, *delivery_mode=None*, *priority=None*, *correlation_id=None*, *reply_to=None*, *expiration=None*, *message_id=None*, *timestamp=None*, *type=None*, *user_id=None*, *app_id=None*`)`

An AMQP Basic message.

Some of the constructor parameters are ignored by the AMQP broker and are provided just for the convenience of user applications. They are marked "for applications" in the list below.

Parameters

- **body** – `bytes` , `str` or `dict` representing the body of the message. Strings will be encoded according to the content_encoding parameter; dicts will be converted to a string using JSON.

- **headers** (*dict*) – a dictionary of message headers
- **content_type** (*str*) – MIME content type
- **content_encoding** (*str*) – MIME encoding
- **delivery_mode** (*int*) – 1 for non-persistent, 2 for persistent
- **priority** (*int*) – message priority - integer between 0 and 9
- **correlation_id** (*str*) – correlation id of the message *(for applications)*
- **reply_to** (*str*) – reply-to address *(for applications)*
- **expiration** (*str*) – expiration specification *(for applications)*
- **message_id** (*str*) – unique id of the message *(for applications)*
- **timestamp** (*datetime.datetime*) – `datetime` of when the message was sent (default: `datetime.now()`)
- **type** (*str*) – message type *(for applications)*
- **user_id** (*str*) – ID of the user sending the message *(for applications)*
- **app_id** (*str*) – ID of the application sending the message *(for applications)*

Attributes are the same as the constructor parameters.

`json``()`

Parse the message body as JSON.

Returns the parsed JSON.

class `asynqp.message.`**`IncomingMessage`**

A message that has been delivered to the client.

Subclass of `Message`.

`ack``()`

Acknowledge the message.

`reject``(*, requeue=True)`

Reject the message.

Parameters **redeliver** (*bool*) – if true, the broker will attempt to requeue the message and deliver it to an alternate consumer.

- *genindex*
- *modindex*
- *search*

Python Module Index