asynqp Documentation

Release 0.4

Benjamin Hodgson

June 30, 2015

Contents

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

CHAPTER 1

Example

```
import asyncio
import asynqp


@asyncio.coroutine
def hello_world():
    """
    Sends a 'hello world' message and then reads it from the queue.
    """
    # 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({'hello': 'world'})
    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 channel.close()
    yield from connection.close()


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

CHAPTER 2

Installation

asynqp has no dependencies outside of the standard library. To install the package:

```
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*, *sock=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.
- **loop** (*BaseEventLoop*) – An instance of `BaseEventLoop` to use. (Defaults to `asyncio.get_event_loop`)
- **sock** (*socket*) – A `socket` instance to use for the connection. This is passed on to `loop.create_connection()`. If `sock` is supplied then `host` and `port` will be ignored.

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

Returns the *Connection* object.

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

Connect to an AMQP server and open a channel on the connection. This function is a coroutine.

Parameters of this function are the same as *connect*.

Returns a tuple of `(connection, channel)`.

Equivalent to:

```
    connection = yield from connect(host, port, username, password, virtual_host, loop=loop, **kwarg
    channel = yield from connection.open_channel()
    return connection, channel
```

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

transport
The BaseTransport over which the connection is communicating with the server

protocol
The Protocol which is paired with the transport

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*, *arguments=None*)
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 unbound from it.
- **internal** (*bool*) – If true, the exchange cannot be published to directly; it can only be bound to other exchanges.
- **arguments** (*dict*) – Table of optional parameters for extensions to the AMQP protocol. See *Protocol extensions*.

Returns the new `Exchange` object.

declare_queue(*name=''*, ***, *durable=True*, *exclusive=False*, *auto_delete=False*, *arguments=None*)
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.
- **arguments** (*dict*) – Table of optional parameters for extensions to the AMQP protocol. See *Protocol extensions*.

Returns The new `Queue` object.

close()
Close the channel by handshaking with the server.

This method is a coroutine.

set_qos(*prefetch_size=0*, *prefetch_count=0*, *apply_globally=False*)
Specify quality of service by requesting that messages be pre-fetched from the server. Pre-fetching means that the server will deliver messages to the client while the client is still processing unacknowledged messages.

This method is a coroutine.

Parameters

- **prefetch_size** (*int*) – Specifies a prefetch window in bytes. Messages smaller than this will be sent from the server in advance. This value may be set to 0, which means "no specific limit".
- **prefetch_count** (*int*) – Specifies a prefetch window in terms of whole messages.
- **apply_globally** (*bool*) – If true, apply these QoS settings on a global level. The meaning of this is implementation-dependent. From the RabbitMQ documentation:

  > RabbitMQ has reinterpreted this field. The original specification said: "By default the QoS settings apply to the current channel only. If this field is set, they are applied to the entire connection." Instead, RabbitMQ takes global=false to mean that the QoS settings should apply per-consumer (for new consumers on the channel; existing ones being unaffected) and global=true to mean that the QoS settings should apply per-channel.

set_return_handler(*handler*)
Set `handler` as the callback function for undeliverable messages that were returned by the server.

By default, an exception is raised, which will be handled by the event loop's exception handler (see `BaseEventLoop.set_exception_handler`). If `handler` is None, this default behaviour is set.

Parameters **handler** (*callable*) – A function to be called when a message is returned. The callback will be passed the undelivered message.

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

arguments
A dictionary of the extra arguments that were used to declare the queue.

bind(*exchange*, *routing_key*, *, *arguments=None*)
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
- **arguments** (*dict*) – Table of optional parameters for extensions to the AMQP protocol. See *Protocol extensions*.

Returns The new *QueueBinding* object

consume(*callback*, *, *no_local=False*, *no_ack=False*, *exclusive=False*, *arguments=None*)
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.
- **arguments** (*dict*) – Table of optional parameters for extensions to the AMQP protocol. See *Protocol extensions*.

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 (*arguments=None*)
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.

This method is a coroutine.

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 (defaults to 'application/json' if body is a dict, or 'application/octet-stream' otherwise)
- **content_encoding** (*str*) – MIME encoding (defaults to 'utf-8')
- **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.**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.

3.1.5 Exceptions

exception asynqp.exceptions.**ConnectionClosedError**
Connection was closed normally by either the amqp server or the client.

exception asynqp.exceptions.**ConnectionLostError**
Connection was closed unexpectedly

exception asynqp.exceptions.**UndeliverableMessage**

exception asynqp.exceptions.**Deleted**

exception asynqp.exceptions.**AMQPError**

exception asynqp.exceptions.**PreconditionFailed**

exception asynqp.exceptions.**UnexpectedFrame**

exception asynqp.exceptions.**ResourceError**

exception asynqp.exceptions.**SyntaxError**

exception asynqp.exceptions.**InvalidPath**

exception asynqp.exceptions.**ConnectionForced**

exception asynqp.exceptions.**AccessRefused**

exception asynqp.exceptions.**FrameError**

exception asynqp.exceptions.**NoConsumers**

exception asynqp.exceptions.**NotAllowed**

exception asynqp.exceptions.**NotImplemented**

exception asynqp.exceptions.**CommandInvalid**

exception asynqp.exceptions.**ResourceLocked**

exception asynqp.exceptions.**ChannelError**

exception asynqp.exceptions.**InternalError**

exception asynqp.exceptions.**ContentTooLarge**

exception asynqp.exceptions.**NotFound**

3.2 Examples

3.2.1 Hello World

```
import asyncio
import asynqp


@asyncio.coroutine
def hello_world():
    """
    Sends a 'hello world' message and then reads it from the queue.
    """
    # 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({'hello': 'world'})
    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 channel.close()
    yield from connection.close()


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

3.2.2 Reconnecting

```
'''
Example async consumer and publisher that will reconnect
automatically when a connection to rabbitmq is broken and
restored.
Note that no attempt is made to re-send messages that are
generated while the connection is down.
'''
import asyncio
import asynqp
from asyncio.futures import InvalidStateError


# Global variables are ugly, but this is a simple example
CHANNELS = []
CONNECTION = None
CONSUMER = None
PRODUCER = None


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

```
                                            password='guest')
    return connection


@asyncio.coroutine
def setup_exchange_and_queue(connection):
    # 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')

    # Save a reference to each channel so we can close it later
    CHANNELS.append(channel)

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

    return exchange, queue


@asyncio.coroutine
def setup_consumer(connection):
    # callback will be called each time a message is received from the queue
    def callback(msg):
        print('Received: {}'.format(msg.body))
        msg.ack()

    _, queue = yield from setup_exchange_and_queue(connection)

    # connect the callback to the queue
    consumer = yield from queue.consume(callback)
    return consumer


@asyncio.coroutine
def setup_producer(connection):
    '''
    The producer will live as an asyncio.Task
    to stop it call Task.cancel()
    '''
    exchange, _ = yield from setup_exchange_and_queue(connection)

    count = 0
    while True:
        msg = asynqp.Message('Message #{}'.format(count))
        exchange.publish(msg, 'routing.key')
        yield from asyncio.sleep(1)
        count += 1


@asyncio.coroutine
def start(loop):
    '''
    Creates a connection, starts the consumer and producer.
    If it fails, it will attempt to reconnect after waiting
    1 second
```

```
    '''
    global CONNECTION
    global CONSUMER
    global PRODUCER
    try:
        CONNECTION = yield from setup_connection(loop)
        CONSUMER = yield from setup_consumer(CONNECTION)
        PRODUCER = loop.create_task(setup_producer(CONNECTION))
    # Multiple exceptions may be thrown, ConnectionError, OsError
    except Exception:
        print('failed to connect, trying again.')
        yield from asyncio.sleep(1)
        loop.create_task(start(loop))


@asyncio.coroutine
def stop():
    '''
    Cleans up connections, channels, consumers and producers
    when the connection is closed.
    '''
    global CHANNELS
    global CONNECTION
    global PRODUCER
    global CONSUMER

    yield from CONSUMER.cancel()  # this is a coroutine
    PRODUCER.cancel()  # this is not

    for channel in CHANNELS:
        yield from channel.close()
    CHANNELS = []

    if CONNECTION is not None:
        try:
            yield from CONNECTION.close()
        except InvalidStateError:
            pass  # could be automatically closed, so this is expected
        CONNECTION = None


def connection_lost_handler(loop, context):
    '''
    Here we setup a custom exception handler to listen for
    ConnectionErrors.

    The exceptions we can catch follow this inheritance scheme

        - ConnectionError - base
            |
            - asynqp.exceptions.ConnectionClosedError - connection closed properly
                |
                - asynqp.exceptions.ConnectionLostError - closed unexpectedly
    '''
    exception = context.get('exception')
    if isinstance(exception, asynqp.exceptions.ConnectionClosedError):
        print('Connection lost -- trying to reconnect')
        # close everything before recpnnecting
```

```
        close_task = loop.create_task(stop())
        asyncio.wait_for(close_task, None)
        # reconnect
        loop.create_task(start(loop))
    else:
        # default behaviour
        loop.default_exception_handler(context)


loop = asyncio.get_event_loop()
loop.set_exception_handler(connection_lost_handler)
loop.create_task(start(loop))
loop.run_forever()
```

3.3 AMQP Procotol Support

asynqp is under development. Here is a table documenting the parts of the AMQP protocol that are currently supported by asynqp.

Note: This library is alpha software. Even the methods marked as 'full support' may still have bugs. Please report any bugs to the Github tracker.

Class	Method	Support	API	Notes
connection		partial	*asynqp.Connection*	
	start/start-ok	full	*asynqp.connect*	
	secure/secure-ok	none		Not required for default auth
	tune/tune-ok	partial		Not presently user-customisab
	open/open-ok	full	*asynqp.connect*	
	close/close-ok	full	*asynqp.Connection.close*	
channel		partial	*asynqp.Channel*	
	open/open-ok	full	*asynqp.Connection.open_channel*	
	flow/flow-ok	none		
	close/close-ok	full	*asynqp.Channel.close*	
exchange		partial	*asynqp.Exchange*	
	declare/declare-ok	partial	*asynqp.Channel.declare_exchange*	Not all parameters presently s
	delete/delete-ok	full	*asynqp.Exchange.delete*	
	bind/bind-ok	none		RabbitMQ extension
	unbind/unbind-ok	none		RabbitMQ extension
queue		partial	*asynqp.Queue*	
	declare/declare-ok	partial	*asynqp.Channel.declare_queue*	Not all parameters presently s
	bind/bind-ok	partial	*asynqp.Queue.bind*	Not all parameters presently s
	unbind/unbind-ok	full	*asynqp.QueueBinding.unbind*	
	purge/purge-ok	partial	*asynqp.Queue.purge*	no-wait not presently supp
	delete/delete-ok	partial	*asynqp.Queue.delete*	no-wait not presently supp
basic		partial		
	qos/qos-ok	full	*asynqp.Channel.set_qos*	
	consume/consume-ok	partial	*asynqp.Queue.consume*	Not all parameters presently s
	cancel/cancel-ok	partial	*asynqp.Consumer.cancel*	no-wait not presently supp
	publish	partial	*asynqp.Exchange.publish*	immediate not presently su
	return	full	*asynqp.Channel.set_return_handler*	

Continued on

Table 3.1 – continued from previous page

Class	Method	Support	API	Notes
	deliver	full		
	get/get-ok/get-empty	full	*asynqp.Queue.get*	
	ack	full	*asynqp.IncomingMessage.ack*	
	reject	full	*asynqp.IncomingMessage.reject*	
	recover/recover-ok	none		
	recover-async	none		
	nack	none		RabbitMQ extension
tx		none		
	select/select-ok	none		
	commit/commit-ok	none		
	rollback/rollback-ok	none		
confirm		none		
	select/select-ok	none		

3.4 Protocol extensions

RabbitMQ, and other brokers, support certain extensions to the AMQP protocol. *asynqp*'s support for such extensions currently includes *optional extra arguments* to certain methods such as *Channel.declare_queue()*.

The acceptable parameters for optional argument dictionaries is implementation-dependent. See'RabbitMQ's supported extensions <http://www.rabbitmq.com/extensions.html>'.

- genindex
- modindex
- search

a

Q

R

S

T

U