ammoo Documentation

Release 1.0.0b1

Mathias Frojdman

Jul 21, 2017

Contents

CHAPTER 1

Examples

Connect to AMQP server

Use the *connect()* function to connect to an AMQP server, which returns a *Connection* object. The connection to the server is opened when entering the *async for* block, and likewise closed when it's exited.

Most of the library's functionality is in the *Channel* class. Call *Connection.channel()* to open one. It's a context manager too: The channel is closed when *async for* exits:

```
import ammoo

async with await ammoo.connect('amqp://localhost/') as connection:
    async with connection.channel() as channel:
        pass  # channel is open
```

A connection may have several channels open at the same time:

```
async with await ammoo.connect('amqp://localhost/') as connection:
    async with connection.channel() as channel_1, connection.channel() as channel_2:
        print('two channels opened')
        async with connection.channel() as channel_3:
            print('yet another one opened - 3 in total')
        print('channel_3 is closed now')
```

Next: *Declare a queue and exchange*

Declare a queue and exchange

Declare a queue

Declare a queue with *Channel.declare_queue()*:

```
await channel.declare_queue('my_queue')  # declares queue with explicit name
```

declare_queue returns some parameters, which might come handy if allowing the server to generate a queue name:

```
declaration = await channel.declare_queue('', exclusive=True)
print(declaration.queue_name)
# something like amq.gen-3Wb1ZY42ejtq31P5LmKVkw on RabbitMQ
```

Declare an exchange

Declare an exchange with *Channel.declare_exchange()*:

```
await channel.declare_exchange('my_exchange', 'fanout')
```

Bind a queue to an exchange

Use *Channel.bind_queue()*:

```
await channel.bind_queue('my_queue', 'my_exchange', 'my_routing_key')
```

Next: *Publish*

Publish

Publish bytes, str and json bodies

Publishing messages happens with *Channel.publish()*, which has a rather huge amount of arguments. Only three are mandatory though: *exchange_name*, *route* and *body/json*. Publish some bytes:

```
await channel.publish('my_exchange', 'my_routing_key', b'message body')
```

A `str` can be used for body. It will be encoded to bytes with the *Channel*'s encoding (utf-8 by default), or with the *encoding* argument if used:

```
await channel.publish('my_exchange', 'my_routing_key', 'text body')
await channel.publish('my_exchange', 'my_routing_key', 'text body', encoding='iso-
↪8859-1')
```

The body argument can be replaced with the json keyword argument:

```
await channel.publish('my_exchange', 'my_routing_key', json=['a', 'list', 'of', {
↪'json: 123}])
```

Publish to a headers exchange

While the other exchange types use routing keys (direct and fanout) or patterns for routing keys (topic) that are strings, the headers exchange type works with keys and values. When publishing a message to a headers exchange, pass a *dict* as route:

```
await channel.publish('my_exchange', {'key': 'value', 'another_key': 123}, b'message
↪body')
```

Next: *Get messages*

Get messages

Direct access to a queue

Messages can be retrieved from a queue synchronously with the *Channel.get()* method:

```
message = await channel.get('my_queue')
```

If the queue is empty, an `EmptyQueue` exception is raised. Otherwise *get* returns a *GetMessage* object. It has a body attribute, which is a *bytearray*:

```
message.body
# bytearray(b'message body')
```

To get a *str* instead, call *message.decode*:

```
message.decode()
# 'message body'
```

While using message.body.decode() is possible, message.decode is shorter and uses the message's content-encoding property if set.

If the message body is JSON, use the *json* method to decode it:

```
message.json()
# ['body', 'is', {'json': True}]
```

Finally, acknowledge the message (unless `no_ack=True` was passed to *get*):

```
await message.ack()
```

or reject it:

```
await message.reject(requeue=False)
```

Subscribe to messages from queue with a consumer

Creating a consumer on queue will make the server send messages to the client when they arrive in the queue, without the need to retrieve each one separately. Calling *Channel.consume()* and entering the returned *Consumer* context with *async for* subscribes to messages from a queue. To access the messages delivered to the consumer, use *async for*:

```
async with channel.consume('my_queue') as consumer:
    async for message in consumer:
        print(message.body)
        await message.ack()
```

CHAPTER 2

API documentation

API documentation

Ammoo leverages Python 3.5's new syntax to make working with AMQP easier:

```
import ammoo

async with await ammoo.connect('amqp://broker/') as connection:
    async with connection.channel() as channel:
        await channel.publish('my_exchange', 'routing_key', 'text body')

        async with channel.consume('my_queue') as consumer:
            async for message in consumer:
                print('Received message: {}'.format(message.body)
                await message.ack()
```

Get a message from queue, decode it's body as text and reply with some JSON:

```
message = await channel.get('my_queue', no_ack=True)
await message.reply(json={'original message': message.decode()})
```

Connect

coroutine ammoo.**connect**(*url=None*, ***, *host*, *port*, *virtualhost*)

Parameters

- **url** (*str*) – URL for the connection. Any component in the URL can be overridden with a keyword argument, or omitted enitrely.
- **host** (*str*) – Server hostname or IP address. Defaults to localhost.
- **port** (*int*) – Server port. Defaults to 5671 for unencrypted connections and 5672 for SSL.

- **virtualhost** (*str*) – AMQP virtualhost to open on connection initialization. Defaults to "/".
- **ssl** (*bool*) – Force encryption on/off. By default the connection is unencrypted. Using the amqps schema in the URL turns encryption on.
- **ssl_context** (*ssl.SSLContext*) – Explicit SSL context object. Use this argument for eg. client certificates or validate the server certificate against a particular certificate chain.
- **heartbeat_interval** (*int*) – Expected number of seconds between frames to consider the connection alive. Both the server and client will send a heartbeat frame with this interval if no other frames have been sent. Defaults to whatever the server suggests. 0 turns heartbeats off.
- **auth** – Authentication mechanism/chooser to use. Defaults to password authentication with the AMQPLAIN or PLAIN mechanism.
- **login** (*str*) – Username to use for default `auth`. Defaults to "guest".
- **password** (*str*) – Password to use for default `auth`. Defaults to "guest".
- **frame_max** (*int*) – Maximum AMQP frame size. Must be at least 4096 bytes. Defaults to 128kB, until the server demands a smaller one.
- **loop** (*asyncio.AbstractEventLoop*) – Event loop. Defaults to `asyncio.get_event_loop()`
- **connection_factory** – Class (or class returning function) to use instead of default *Connection*.

Connects to an AMQP server and returns a *Connection* instance:

```
await ammoo.connect('amqps://myserver/myvhost')
await ammoo.connect(host='myserver', virtualhost='myvhost', ssl=True)
```

Connecting to unencrypted AMQP on *localhost* to virtualhost */* is simply:

```
await ammoo.connect()
```

Connection

class `ammoo.`**Connection**

An AMQP connection. It's an asynchronous context manager, which does the server handshake while entering, and closes the connection in exit. Most of the class's methods are unusable until the connection has been entered, so use *connect()* to get one, and use it within *async for*:

```
async with await connect() as connection:
    # correct!

async with connect() as connection:
    # Fails: connect() is a coroutine that needs to be awaited

connection = await connect()  # works, but...
connection.channel()  # raises an exception because connection isn't entered
```

channel(**, prefetch_size=None, prefetch_count=None*)

Parameters

- **prefetch_size** (*int*) – Passed to *Channel.qos()* if used.
- **prefetch_count** (*int*) – Passed to *Channel.qos()* if used.

Return type *Channel*

Open a new channel. Should be used in a context manager:

```
async with connection.channel() as channel:
    ...
```

If prefetch_size or prefetch_count are given, *Channel.qos()* is called after opening the channel. The two async with blocks achieve the same:

```
async with connection.channel(prefetch_count=5) as channel:
    ...

async with connection.channel() as channel:
    await channel.qos(prefetch_count=5, prefetch_size=0)
    ...
```

Channel

class ammoo.**Channel**

An AMQP channel.

consume(*queue_name*, *, ...)

Start a new consumer on a queue.

Parameters

- **queue_name** (*str*) – Queue name
- **no_ack** (*bool*) – Optional: If True, server does not expect messages delivered to consumer to be acknowledged or rejected.
- **no_local** (*bool*) – Optional: If True, the server will not deliver messages to the connection that published them.
- **exclusive** (*bool*) – Optional: If True, only this consumer can access the queue.
- **priority** (*int*) – Optional: Set consumer priority. Lower priority consumers will receive messages only when higher priority ones are busy (Sets x-priority on the consumer).

Return type *Consumer*

Note: priority is a RabbitMQ extension

---

coroutine get(*queue_name*, *, ...)

Get a message from queue.

Parameters

- **queue_name** (*str*) – Queue name
- **no_ack** (*bool*) – Optional: If True, server does not expect message to be acknowledged or rejected.

Raises **EmptyQueue** – If there are no messages in queue, EmptyQueue is raised

Return type *GetMessage*

coroutine ack (*delivery_tag*)
Acknowledge a message.

Parameters **delivery_tag** (*int*) – Delivery tag of message to acknowledge (*ExpectedMessage.delivery_tag*)

See also:

ExpectedMessage.ack()

coroutine reject (*delivery_tag*, *requeue*)
Reject a message. Opposite of *ack()*.

Parameters

- **delivery_tag** (*int*) – Delivery tag of message to reject (*ExpectedMessage.delivery_tag*)
- **requeue** (*bool*) – If True, the server will try to requeue the message. False means the message is discarded or dead-lettered.

See also:

ExpectedMessage.reject()

coroutine qos (*prefetch_size*, *prefetch_count*, *global_*)
Limit how many unacknowledged messages (or message data) will be delivered to consumers. Without *qos*, the server will deliver all of the queue's messages to the consumer, possibly causing the consumer to run out of memory or starving other consumers of messages.

Parameters

- **prefetch_size** (*int*) – Maximum number of unacknowledged messages server will deliver to a consumer. Zero turns the limit off.
- **prefetch_count** (*int*) – Maximum combined size of unacknowledged messages server will deliver to a consumer. Zero turns the limit off.
- **global** (*bool*) – For standard AMQP: True applies the qos to the whole connection, and False to the channel only. For RabbitMQ: True applies the setting to both the channel's current consumers and future ones, while False only applies to the latter.

coroutine recover (*requeue*)
Ask server to redeliver unacknowledged messages

Parameters **requeue** (*bool*) – If False, redeliver messages to the original recipient. If True, the message may be delivered to another recipient.

coroutine publish (*exchange_name*, *route*[, *body*], *, ...)
Publish a message *body* to *exchange_name* with *route*. *body* and *json* are mutually exclusive, but one of them has to be used.

Publish a binary body with `bytes` or `bytearray`:

```
await channel.publish(exchange_name, routing_key, b'binary bytes')
await channel.publish(exchange_name, routing_key, bytearray(b'binary bytarray
↪'))
```

Publish a `str`:

```
# encoded to bytes with channel's default encoding
await channel.publish(exchange_name, routing_key, 'text string')
# use non-default encoding
await channel.publish(exchange_name, routing_key, 'text string', encoding=
↪'iso-8859-1')
```

Serialize JSON into body:

```
await channel.publish(exchange_name, routing_key, json={'key': 123})
```

Set the content-encoding property:

```
# body will also be encoded with iso-8859-1 instead of channel's default␣
↪encoding
await channel.publish(exchange_name, routing_key, 'some text', content_
↪encoding='iso-8859-1'})
# content-encoding property is set to channel's default encoding
await channel.publish(exchange_name, routing_key, 'some text', content_
↪encoding=True})
# can be used for json too
await channel.publish(exchange_name, routing_key, json={'key': 123}, content_
↪encoding='iso-8859-1'})
```

Set the content-type property:

```
await channel.publish(exchange_name, routing_key, b'binary data', content_
↪type=True})  # bytes body -> content-type is set to application/octet-stream
await channel.publish(exchange_name, routing_key, 'some text', content_
↪type=True})  # str body -> content-type is set to text/plain
await channel.publish(exchange_name, routing_key, json={'key': 123}, content_
↪type=True})  # json -> content-type is set to application/json
await channel.publish(exchange_name, routing_key, body, content_type=
↪'application/acme-2000'})  # or a regular str
```

Parameters

- **exchange_name** (*str*) – Exchange name. Use an empty string for the default exchange.
- **route** – A `str` is used as a literal routing key, while a `Mapping` is used for the headers exchange type.
- **body** – Message body. `str`, `bytes` or `bytearray`. If the *json* keyword argument is used, body may be omitted.
- **json** – Optional: Object to serialize as JSON into body. Cannot be used at the same time as the body argument.
- **mandatory** (*bool*) – Optional: When True, messages the cannot be routed to a queue are returned back to the client.
- **immediate** (*bool*) – Optional: When True, messages that are not routed to a consumer immediately are returned back to the client.
- **encoding** (*str*) – Optional: Encode `str` body to bytes with this encoding.
- **correlation_id** (*str*) – Optional: Correlation-id property.
- **reply_to** (*str*) – Optional: Reply-to property.
- **expiration** – Optional: Message expiration property, usually in milliseconds. Messages die if they are not consumed from queue within this TTL. `int` or `str`.
- **cc** – Optional: Additional routing keys to use when routing message to queues.
- **bcc** – Optional: Like cc, but bcc will be removed from message before delivery.

- **priority** (*int*) – Optional: priority property.
- **delivery_mode** (*int*) – Optional: delivery-mode property 1 for non-persistent or 2 for persistent.
- **timestamp** (*datetime*) – Optional: Message timestamp property. If time zone is not set, UTC is assumed.
- **content_encoding** – Optional: content-encoding property. A `str`, or if a `bool` True, the value of the *encoding* argument or the channel's default.
- **content_type** – Optional: content-type property. A `str`, or if a `bool` True, set *application/octet-stream* if body is `bytes` or `bytearray`, *text/plain* if it's `str`, and *application/json* if the *json* argument was used.
- **message_id** (*str*) – Optional: message-id property.
- **type** (*str*) – Optional: type property.
- **user_id** (*str*) – Optional: user-id property.
- **app_id** (*str*) – Optional: app-id property.

Note: cc and bcc are RabbitMQ extensions. Not supported by standard AMQP.

---

coroutine select_confirm()
Turns publisher confirms on for the channel.

Note: RabbitMQ extension. Not supported by standard AMQP.

---

coroutine declare_exchange(*exchange_name*, *exchange_type*, *, ...)
Declare exchange.

Parameters

- **exchange_name** (*str*) – Exchange name
- **exchange_type** (*str*) – Exchange type: direct, fanout, topic, or headers.
- **durable** (*bool*) – Optional: If True, exchange will survive server restart.
- **auto_delete** (*bool*) – Optional: If True, the exchange is deleted (after a delay) when the last queue is unbound from it.
- **alternate_exchange_name** – Optional: Alternate exchange name. Messages that can't be routed to any queue are instead published on the alternate exchange (Sets alternate-exchange on the exchange).

coroutine delete_exchange(*exchange_name*, *, ...)
Delete exchange *exchange_name*.

Parameters

- **exchange_name** (*str*) – Exchange name
- **if_unused** (*bool*) – Optional: Only delete exchange if it is unused.

coroutine assert_exchange_exists(*exchange_name*)
Asserts *exchange_name* exists*. Channel will be closed if it does not!

Parameters **exchange_name** – Exchange name

coroutine bind_exchange(*destination*, *source*, *routing_key*)
Bind *source* exchange to *destination* exchange for *routing_key*.

Parameters

- **destination** (*str*) – Destination exchange name
- **source** (*str*) – Source exchange name
- **routing_key** (*str*) – Routing key

Note: RabbitMQ extension. Not supported by standard AMQP.

---

coroutine declare_queue(*queue_name*, *, ...)
Declare a queue named *queue_name*.

Parameters

- **queue_name** (*str*) – Queue name
- **exclusive** (*bool*) – Optional: If True, the queue can be accessed only by this connection and is deleted when connection is closed.
- **durable** (*bool*) – Optional: If True, the queue will be marked as durable, surviving server restarts.
- **auto_delete** (*bool*) – Optional: If True, the queue is deleted when all consumers are finished using it.
- **ttl** (*int*) – Optional: Messages die after this number of milliseconds, if no one has consumed them first (Sets x-message-ttl on the queue).
- **expires** (*int*) – Optional: Milliseconds queue is unused before it is deleted (Sets x-expires on the queue).
- **max_length** (*int*) – Optional: Maximum number of messages in the queue before the oldest will die (Sets x-max-length on the queue).
- **max_length_bytes** (*int*) – Optional: Maximum number of message bytes in the queue before the oldest will die (Sets x-max-length on the queue).
- **dead_letter_exchange** (*str*) – Optional: Name of dead letter exchange. The queue's dead messages are routed here (Sets x-dead-letter-exchange on the queue).
- **dead_letter_routing_key** – Optional: Override dead messages' routing key when routing them to *dead_letter_exchange* (Sets x-dead-letter-routing-key on the queue).
- **max_priority** (*int*) – Optional: Queue's maximum priority (Sets x-max-priority).

Return type *QueueDeclareOkParameters*

coroutine delete_queue(*queue_name*)
Delete a queue named *queue_name*. If the queue does not exist, the method merely asserts it is not there.

Parameters

- **queue_name** – Queue name
- **if_unused** (*bool*) – Optional: Only delete queue if it has no consumers.
- **if_empty** (*bool*) – Optional: Only delete queue if it has no messages.

Returns Number of messages in queue before it was deleted

coroutine purge_queue(*queue_name*)
Purges a queue of messages, emptying it.

Parameters **queue_name** (*str*) – Queue name

Returns Number of messages in queue before it was purged

coroutine bind_queue(*queue_name*, *exchange_name*, *routing_key*)
Bind *queue_name* to *exchange_name* for *routing_key*.

Parameters

- **queue_name** (*str*) – Queue name
- **exchange_name** (*str*) – Exchange name
- **routing_key** – A `str` is used as a literal routing key, and a `Mapping` for the headers exchange type.

coroutine unbind_queue(*queue_name*, *exchange_name*, *routing_key*)
Unbind *queue_name* from *exchange_name* for *routing_key*. Undoes *bind_queue()*.

Parameters

- **queue_name** (*str*) – Queue name
- **exchange_name** (*str*) – Exchange name
- **routing_key** – Same as for *bind_queue()*.

coroutine assert_queue_exists(*queue_name*)
Asserts *queue_name* exists*. Channel is closed by the server if it does not!

Parameters **queue_name** – Queue name

Return type *QueueDeclareOkParameters*

Consumer

class ammoo.**Consumer**
An AMQP consumer. Asynchronous iterable that returns *DeliverMessage* instances. Must be used in an *async for* block:

```
async with channel.consume('my_queue') as consumer:
    async for message in consumer:
        ...
```

Message

class ammoo.**Message**
Base class for messages. Not instantiated directly:

```
Message: body, decode(), json(), exchange_name, routing_key, properties
-> ReturnMessage: reply_code, reply_text
-> ExpectedMessage: ack(), reject(), reply(), delivery_tag, redelivered
   -> DeliverMessage: consumer_tag
   -> GetMessage: message_count
```

body
Message's raw body as a `bytearray` instance

```
>>> message.body
bytearray(b'binary data')
```

exchange_name
Exchange message was published to

routing_key
Routing key message was published with

properties
Message's *BasicHeaderProperties*

decode(*encoding=None*)
Decode body into a `str`. When the encoding argument is not passed, the encoding defaults to the message's content-encoding property (if defined), or the channel's default encoding.

```
>>> message.decode()
'text body'
>>> message.decode('iso-8859-1')
'sí'
```

Parameters **encoding** (*str*) – Optional: Encoding to use to decode body instead of content-encoding property/channel's default encoding

Return type str

json(*encoding=None*)
Decode body as JSON.

Parameters **encoding** (*str*) – Optional: Encoding to use to decode body instead of content-encoding property/channel's default encoding

Return type str, bool, int, float, None, dict, list

class ammoo.**ExpectedMessage**
Base class for *DeliverMessage* and *GetMessage*; not instantiated directly. Subclass of *Message*.

delivery_tag
Message's delivery tag, used for acknowledging or rejecting message to server

---

Note: Using the *ack()* or *reject()* methods of this class instead of *Channel*'s avoids needing to pass the delivery tag explicitly.

---

coroutine ack()
Acknowledge message to server. Calls *Channel.ack()* with the message's delivery tag.

coroutine reject(*requeue*)
Reject message to server. Calls *Channel.reject()* with the message's delivery tag.

coroutine reply(*[body]*, *, ...)
Publish a reply to a message that has the reply-to property set. If the message has the correlation-id property, it's also set on the published message.

The method accepts the same keyword arguments as *Channel.publish()*.

Note: Direct reply to is a RabbitMQ extension

---

class ammoo.**DeliverMessage**
Message delivered to a *Consumer*. Subclass of *ExpectedMessage*.

consumer_tag
str consumer tag parameter of delivered message.

class ammoo.**GetMessage**
A message from queue returned by calling *Channel.get()*. Subclass of *ExpectedMessage*.

message_count
Number of messages still in queue after getting this message.

class ammoo.**ReturnMessage**
Message returned by server as a consequence of using the *mandatory* or *immediate* flags of *Channel.publish()*. Subclass of *Message*.

reply_code
int code for why message could not be routed to queue/consumed.

reply_text
str description of why message was returned.

Message properties

class ammoo.wire.frames.header.**BasicHeaderProperties**(*content_type*, *content_encoding*, *headers*, *delivery_mode*, *priority*, *correlation_id*, *reply_to*, *expiration*, *message_id*, *timestamp*, *type_*, *user_id*, *app_id*, *cluster_id*)

app_id
Alias for field number 12

cluster_id
Alias for field number 13

content_encoding
Alias for field number 1

content_type
Alias for field number 0

correlation_id
Alias for field number 5

delivery_mode
Alias for field number 3

expiration
Alias for field number 7

headers
Alias for field number 2

message_id
Alias for field number 8

priority
Alias for field number 4

reply_to
Alias for field number 6

timestamp
Alias for field number 9

type_
Alias for field number 10

user_id
Alias for field number 11

Parameters

class ammoo.wire.frames.method.queue.**QueueDeclareOkParameters**(*queue_name*, *message_count*, *consumer_count*)

consumer_count
Alias for field number 2

message_count
Alias for field number 1

queue_name
Alias for field number 0

Python Module Index

a

Index

Q

R

S

T

U