amqpstorm Documentation

Release 2.8.1

Erik Olof Gunnar Andersson

Aug 18, 2020

Usage

Thread-safe Python RabbitMQ Client & Management library.

CHAPTER 1

Installation

The latest version can be installed using pip and is available at pypi here

```
pip install amqpstorm
```

CHAPTER 2

Basic Example

```
with amqpstorm.Connection('rmq.amqpstorm.io', 'guest', 'guest') as␣
↪connection:
    with connection.channel() as channel:
        channel.queue.declare('fruits')
        message = amqpstorm.Message.create(
            channel, body='Hello RabbitMQ!', properties={
                'content_type': 'text/plain'
            })
        message.publish('fruits')
```

CHAPTER 3

Additional Examples

A wide verity of examples are available on Github at here

3.1 Connection

class amqpstorm.**Connection**(*hostname*, *username*, *password*, *port=5672*, ***kwargs*)

RabbitMQ Connection.

e.g.

```
import amqpstorm
connection = amqpstorm.Connection('localhost', 'guest', 'guest')
```

Using a SSL Context:

```
import ssl
import amqpstorm
ssl_options = {
    'context': ssl.create_default_context(cafile='cacert.pem'),
    'server_hostname': 'rmq.eandersson.net'
}
connection = amqpstorm.Connection(
    'rmq.eandersson.net', 'guest', 'guest', port=5671,
    ssl=True, ssl_options=ssl_options
)
```

Parameters

- **hostname** (*str*) – Hostname
- **username** (*str*) – Username
- **password** (*str*) – Password
- **port** (*int*) – Server port
- **virtual_host** (*str*) – Virtual host
- **heartbeat** (*int*) – RabbitMQ Heartbeat interval
- **timeout** (*int, float*) – Socket timeout
- **ssl** (*bool*) – Enable SSL
- **ssl_options** (*dict*) – SSL kwargs
- **client_properties** (*dict*) – None or dict of client properties
- **lazy** (*bool*) – Lazy initialize the connection

Raises ***AMQPConnectionError*** – Raises if the connection encountered an error.

channels
Returns a dictionary of the Channels currently available.

Return type dict

fileno
Returns the Socket File number.

Return type integer,None

is_blocked
Is the connection currently being blocked from publishing by the remote server.

Return type bool

max_allowed_channels
Returns the maximum allowed channels for the connection.

Return type int

max_frame_size
Returns the maximum allowed frame size for the connection.

Return type int

server_properties
Returns the RabbitMQ Server Properties.

Return type dict

socket
Returns an instance of the Socket used by the Connection.

Return type socket.socket

channel(*rpc_timeout=60*, *lazy=False*)
Open a Channel.

Parameters **rpc_timeout** (*int*) – Timeout before we give up waiting for an RPC response from the server.

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type *amqpstorm.Channel*

check_for_errors()
Check Connection for errors.

Raises ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

close()
Close the Connection.

Raises ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

open()
Open Connection.

Raises ***AMQPConnectionError*** – Raises if the connection encountered an error.

3.2 UriConnection

class amqpstorm.**UriConnection**(*uri*, *ssl_options=None*, *client_properties=None*, *lazy=False*)
RabbitMQ Connection that takes a Uri string.

e.g.

```
import amqpstorm
connection = amqpstorm.UriConnection(
    'amqp://guest:guest@localhost:5672/%2F?heartbeat=60'
)
```

Using a SSL Context:

```
import ssl
import amqpstorm
ssl_options = {
    'context': ssl.create_default_context(cafile='cacert.pem'),
    'server_hostname': 'rmq.eandersson.net'
}
connection = amqpstorm.UriConnection(
    'amqps://guest:guest@rmq.eandersson.net:5671/%2F?heartbeat=60
↪',
    ssl_options=ssl_options
)
```

Parameters

- **uri** (*str*) – AMQP Connection string
- **ssl_options** (*dict*) – SSL kwargs
- **client_properties** (*dict*) – None or dict of client properties
- **lazy** (*bool*) – Lazy initialize the connection

Raises

- **TypeError** – Raises on invalid uri.
- **ValueError** – Raises on invalid uri.
- **AttributeError** – Raises on invalid uri.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

3.3 Channel

class amqpstorm.**Channel**(*channel_id*, *connection*, *rpc_timeout*, *on_close_impl=None*)

RabbitMQ Channel.

e.g.

```
channel = connection.channel()
```

basic
RabbitMQ Basic Operations.

e.g.

```
message = channel.basic.get(queue='hello_world')
```

Return type *amqpstorm.basic.Basic*

exchange
RabbitMQ Exchange Operations.

e.g.

```
channel.exchange.declare(exchange='hello_world')
```

Return type *amqpstorm.exchange.Exchange*

queue
RabbitMQ Queue Operations.

e.g.

```
channel.queue.declare(queue='hello_world')
```

Return type *amqpstorm.queue.Queue*

tx
RabbitMQ Tx Operations.

e.g.

```
channel.tx.commit()
```

Return type *amqpstorm.tx.Tx*

build_inbound_messages(*break_on_empty=False*, *to_tuple=False*, *auto_decode=True*)
Build messages in the inbound queue.

Parameters

- **break_on_empty** (*bool*) – Should we break the loop when there are no more messages in our inbound queue.

This does not guarantee that the queue is emptied before the loop is broken, as messages may be consumed faster then they are being delivered by RabbitMQ, causing the loop to be broken prematurely.

- **to_tuple** (*bool*) – Should incoming messages be converted to a tuple before delivery.
- **auto_decode** (*bool*) – Auto-decode strings when possible.

Raises

- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type generator

close(*reply_code=200*, *reply_text=''*)
Close Channel.

Parameters

- **reply_code** (*int*) – Close reply code (e.g. 200)
- **reply_text** (*str*) – Close reply text

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

check_for_errors()
Check connection and channel for errors.

Raises

- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

check_for_exceptions()
Check channel for exceptions.

Raises ***AMQPChannelError*** – Raises if the channel encountered an error.

Returns

confirm_deliveries()
Set the channel to confirm that each message has been successfully delivered.

Raises

- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

process_data_events(*to_tuple=False*, *auto_decode=True*)
Consume inbound messages.

Parameters

- **to_tuple** (*bool*) – Should incoming messages be converted to a tuple before delivery.
- **auto_decode** (*bool*) – Auto-decode strings when possible.

Raises

- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

start_consuming(*to_tuple=False*, *auto_decode=True*)
Start consuming messages.

Parameters

- **to_tuple** (*bool*) – Should incoming messages be converted to a tuple before delivery.
- **auto_decode** (*bool*) – Auto-decode strings when possible.

Raises

- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

stop_consuming()
Stop consuming messages.

Raises

- ***AMQPChannelError*** – Raises if the channel encountered an error.

- **AMQPConnectionError** – Raises if the connection encountered an error.

Returns

3.4 Channel.Basic

class amqpstorm.basic.**Basic**(*channel*, *max_frame_size=None*)

RabbitMQ Basic Operations.

qos(*prefetch_count=0*, *prefetch_size=0*, *global_=False*)

Specify quality of service.

Parameters

- **prefetch_count** (*int*) – Prefetch window in messages
- **prefetch_size** (*int/long*) – Prefetch window in octets
- **global** (*bool*) – Apply to entire connection

Raises

- **AMQPInvalidArgument** – Invalid Parameters
- **AMQPChannelError** – Raises if the channel encountered an error.
- **AMQPConnectionError** – Raises if the connection encountered an error.

Return type dict

get(*queue=''*, *no_ack=False*, *to_dict=False*, *auto_decode=True*)

Fetch a single message.

Parameters

- **queue** (*str*) – Queue name
- **no_ack** (*bool*) – No acknowledgement needed
- **to_dict** (*bool*) – Should incoming messages be converted to a dictionary before delivery.
- **auto_decode** (*bool*) – Auto-decode strings when possible.

Raises

- **AMQPInvalidArgument** – Invalid Parameters
- **AMQPChannelError** – Raises if the channel encountered an error.

- **AMQPConnectionError** – Raises if the connection encountered an error.

Returns Returns a single message, as long as there is a message in the queue. If no message is available, returns None.

Return type *amqpstorm.Message*,dict,None

recover(*requeue=False*)
Redeliver unacknowledged messages.

Parameters **requeue** (*bool*) – Re-queue the messages

Raises

- **AMQPInvalidArgument** – Invalid Parameters
- **AMQPChannelError** – Raises if the channel encountered an error.
- **AMQPConnectionError** – Raises if the connection encountered an error.

Return type dict

consume(*callback=None*, *queue=''*, *consumer_tag=''*, *exclusive=False*, *no_ack=False*, *no_local=False*, *arguments=None*)
Start a queue consumer.

Parameters

- **callback** (*function*) – Message callback
- **queue** (*str*) – Queue name
- **consumer_tag** (*str*) – Consumer tag
- **no_local** (*bool*) – Do not deliver own messages
- **no_ack** (*bool*) – No acknowledgement needed
- **exclusive** (*bool*) – Request exclusive access
- **arguments** (*dict*) – Consume key/value arguments

Raises

- **AMQPInvalidArgument** – Invalid Parameters
- **AMQPChannelError** – Raises if the channel encountered an error.
- **AMQPConnectionError** – Raises if the connection encountered an error.

Returns Consumer tag

Return type str

cancel(*consumer_tag=''*)
Cancel a queue consumer.

Parameters **consumer_tag** (*str*) – Consumer tag

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

publish(*body*, *routing_key*, *exchange=''*, *properties=None*, *mandatory=False*, *immediate=False*)
Publish a Message.

Parameters

- **body** (*bytes,str,unicode*) – Message payload
- **routing_key** (*str*) – Message routing key
- **exchange** (*str*) – The exchange to publish the message to
- **properties** (*dict*) – Message properties
- **mandatory** (*bool*) – Requires the message is published
- **immediate** (*bool*) – Request immediate delivery

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type bool,None

ack(*delivery_tag=0*, *multiple=False*)
Acknowledge Message.

Parameters

- **delivery_tag** (*int/long*) – Server-assigned delivery tag
- **multiple** (*bool*) – Acknowledge multiple messages

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters

- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

nack(*delivery_tag=0*, *multiple=False*, *requeue=True*)
Negative Acknowledgement.

Parameters

- **delivery_tag** (*int/long*) – Server-assigned delivery tag
- **multiple** (*bool*) – Negative acknowledge multiple messages
- **requeue** (*bool*) – Re-queue the message

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

reject(*delivery_tag=0*, *requeue=True*)
Reject Message.

Parameters

- **delivery_tag** (*int/long*) – Server-assigned delivery tag
- **requeue** (*bool*) – Re-queue the message

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

3.5 Channel.Exchange

class amqpstorm.exchange.**Exchange**(*channel*)
RabbitMQ Exchange Operations.

declare(*exchange=''*, *exchange_type='direct'*, *passive=False*, *durable=False*, *auto_delete=False*, *arguments=None*)

Declare an Exchange.

Parameters

- **exchange** (*str*) – Exchange name
- **exchange_type** (*str*) – Exchange type
- **passive** (*bool*) – Do not create
- **durable** (*bool*) – Durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Exchange key/value arguments

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

delete(*exchange=''*, *if_unused=False*)

Delete an Exchange.

Parameters

- **exchange** (*str*) – Exchange name
- **if_unused** (*bool*) – Delete only if unused

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

bind(*destination=''*, *source=''*, *routing_key=''*, *arguments=None*)

Bind an Exchange.

Parameters

- **destination** (*str*) – Exchange name
- **source** (*str*) – Exchange to bind to

- **routing_key** (*str*) – The routing key to use
- **arguments** (*dict*) – Bind key/value arguments

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

unbind(*destination=''*, *source=''*, *routing_key=''*, *arguments=None*)

Unbind an Exchange.

Parameters

- **destination** (*str*) – Exchange name
- **source** (*str*) – Exchange to unbind from
- **routing_key** (*str*) – The routing key used
- **arguments** (*dict*) – Unbind key/value arguments

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

3.6 Channel.Queue

class amqpstorm.queue.**Queue**(*channel*)

RabbitMQ Queue Operations.

declare(*queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=False*, *arguments=None*)

Declare a Queue.

Parameters

- **queue** (*str*) – Queue name
- **passive** (*bool*) – Do not create

- **durable** (*bool*) – Durable queue
- **exclusive** (*bool*) – Request exclusive access
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict*) – Queue key/value arguments

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

delete(*queue=''*, *if_unused=False*, *if_empty=False*)

Delete a Queue.

Parameters

- **queue** (*str*) – Queue name
- **if_unused** (*bool*) – Delete only if unused
- **if_empty** (*bool*) – Delete only if empty

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

purge(*queue*)

Purge a Queue.

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

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

bind(*queue=''*, *exchange=''*, *routing_key=''*, *arguments=None*)
Bind a Queue.

Parameters

- **queue** (*str*) – Queue name
- **exchange** (*str*) – Exchange name
- **routing_key** (*str*) – The routing key to use
- **arguments** (*dict*) – Bind key/value arguments

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

unbind(*queue=''*, *exchange=''*, *routing_key=''*, *arguments=None*)
Unbind a Queue.

Parameters

- **queue** (*str*) – Queue name
- **exchange** (*str*) – Exchange name
- **routing_key** (*str*) – The routing key used
- **arguments** (*dict*) – Unbind key/value arguments

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type dict

3.7 Channel.Tx

class amqpstorm.tx.**Tx**(*channel*)
RabbitMQ Transactions.

Server local transactions, in which the server will buffer published messages until the client commits (or rollback) the messages.

select()
Enable standard transaction mode.

This will enable transaction mode on the channel. Meaning that messages will be kept in the remote server buffer until such a time that either commit or rollback is called.

Returns

commit()
Commit the current transaction.

Commit all messages published during the current transaction session to the remote server.

A new transaction session starts as soon as the command has been executed.

Returns

rollback()
Abandon the current transaction.

Rollback all messages published during the current transaction session to the remote server.

Note that all messages published during this transaction session will be lost, and will have to be published again.

A new transaction session starts as soon as the command has been executed.

Returns

3.8 Exceptions

class amqpstorm.**AMQPError**(**args*, ***kwargs*)
General AMQP Error.

Exceptions raised by AMQPStorm are mapped based to the AMQP 0.9.1 specifications (when applicable).

e.g.

```
except AMQPChannelError as why:
    if why.error_code == 312:
        self.channel.queue.declare(queue_name)
```

documentation
AMQP Documentation string.

error_code
AMQP Error Code - A 3-digit reply code.

error_type
AMQP Error Type e.g. NOT-FOUND.

class amqpstorm.**AMQPConnectionError**(**args*, ***kwargs*)
AMQP Connection Error.

class amqpstorm.**AMQPChannelError**(**args*, ***kwargs*)
AMQP Channel Error.

class amqpstorm.**AMQPMessageError**(**args*, ***kwargs*)
AMQP Message Error.

class amqpstorm.**AMQPInvalidArgument**(**args*, ***kwargs*)
AMQP Argument Error.

3.9 Message

class amqpstorm.**Message**(*channel*, *auto_decode=True*, ***message*)
RabbitMQ Message.

e.g.

```
# Message Properties.
properties = {
    'content_type': 'text/plain',
    'expiration': '3600',
    'headers': {'key': 'value'},
}
# Create a new message.
message = Message.create(channel, 'Hello RabbitMQ!', properties)
# Publish the message to a queue called, 'my_queue'.
message.publish('my_queue')
```

Parameters

- **channel** (Channel) – AMQPStorm Channel

- **auto_decode** (*bool*) – Auto-decode strings when possible. Does not apply to to_dict, or to_tuple.
- **body** (*bytes,str,unicode*) – Message payload
- **method** (*dict*) – Message method
- **properties** (*dict*) – Message properties

static create(*channel*, *body*, *properties=None*)

Create a new Message.

Parameters

- **channel** (Channel) – AMQPStorm Channel
- **body** (*bytes,str,unicode*) – Message payload
- **properties** (*dict*) – Message properties

Return type *Message*

body

Return the Message Body.

If auto_decode is enabled, the body will automatically be decoded using decode('utf-8') if possible.

Return type bytes,str,unicode

channel

Return the Channel used by this message.

Return type *Channel*

method

Return the Message Method.

If auto_decode is enabled, all strings will automatically be decoded using decode('utf-8') if possible.

Return type dict

properties

Returns the Message Properties.

If auto_decode is enabled, all strings will automatically be decoded using decode('utf-8') if possible.

Return type dict

ack()
Acknowledge Message.

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Returns

nack(*requeue=True*)
Negative Acknowledgement.

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Parameters **requeue** (*bool*) – Re-queue the message

reject(*requeue=True*)
Reject Message.

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Parameters **requeue** (*bool*) – Re-queue the message

publish(*routing_key*, *exchange=''*, *mandatory=False*, *immediate=False*)
Publish Message.

Parameters

- **routing_key** (*str*) – Message routing key
- **exchange** (*str*) – The exchange to publish the message to
- **mandatory** (*bool*) – Requires the message is published
- **immediate** (*bool*) – Request immediate delivery

Raises

- ***AMQPInvalidArgument*** – Invalid Parameters
- ***AMQPChannelError*** – Raises if the channel encountered an error.
- ***AMQPConnectionError*** – Raises if the connection encountered an error.

Return type bool,None

app_id
Get AMQP Message attribute: app_id.

Returns

message_id
Get AMQP Message attribute: message_id.

Returns

content_encoding
Get AMQP Message attribute: content_encoding.

Returns

content_type
Get AMQP Message attribute: content_type.

Returns

correlation_id
Get AMQP Message attribute: correlation_id.

Returns

delivery_mode
Get AMQP Message attribute: delivery_mode.

Returns

timestamp
Get AMQP Message attribute: timestamp.

Returns

priority
Get AMQP Message attribute: priority.

Returns

reply_to
Get AMQP Message attribute: reply_to.

Returns

redelivered
Indicates if this message may have been delivered before (but not acknowledged).

Return type bool,None

delivery_tag
Server-assigned delivery tag.

Return type int,None

json()
Deserialize the message body, if it is JSON.

Returns

3.10 Management Api

class amqpstorm.management.**ManagementApi**(*api_url*, *username*, *password*, *timeout=10*, *verify=None*, *cert=None*)

RabbitMQ Management Api

e.g.

```
from amqpstorm.management import ManagementApi
client = ManagementApi('http://localhost:15672', 'guest', 'guest')
client.user.create('my_user', 'password')
client.user.set_permission(
    'my_user',
    virtual_host='/',
    configure_regex='.*',
    write_regex='.*',
    read_regex='.*'
)
```

basic
RabbitMQ Basic Operations.

e.g.

```
client.basic.publish('Hello RabbitMQ', routing_key='my_
↪queue')
```

Return type *amqpstorm.management.basic.Basic*

channel
RabbitMQ Channel Operations.

e.g.

```
client.channel.list()
```

Return type *amqpstorm.management.channel.Channel*

connection
RabbitMQ Connection Operations.

e.g.

```
client.connection.list()
```

Return type *amqpstorm.management.connection.Connection*

exchange
RabbitMQ Exchange Operations.

e.g.

```
client.exchange.declare('my_exchange')
```

Return type *amqpstorm.management.exchange.Exchange*

queue
RabbitMQ Queue Operations.

e.g.

```
client.queue.declare('my_queue', virtual_host='/')
```

Return type *amqpstorm.management.queue.Queue*

user
RabbitMQ User Operations.

e.g.

```
client.user.create('my_user', 'password')
```

Return type *amqpstorm.management.user.User*

aliveness_test(*virtual_host='/'*)
Aliveness Test.

e.g.

```
from amqpstorm.management import ManagementApi
client = ManagementApi('http://localhost:15672', 'guest',
↪'guest')
result = client.aliveness_test('/')
if result['status'] == 'ok':
    print("RabbitMQ is alive!")
else:
    print("RabbitMQ is not alive! :(")
```

Parameters **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

overview()
Get Overview.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

nodes()
Get Nodes.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

top()
Top Processes.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type list

whoami()
Who am I?

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

class amqpstorm.management.basic.**Basic**(*http_client*)

publish(*body*, *routing_key*, *exchange='amq.default'*, *virtual_host='/'*, *properties=None*, *payload_encoding='string'*)
Publish a Message.

Parameters

- **body** (*bytes,str,unicode*) – Message payload
- **routing_key** (*str*) – Message routing key
- **exchange** (*str*) – The exchange to publish the message to
- **virtual_host** (*str*) – Virtual host name
- **properties** (*dict*) – Message properties
- **payload_encoding** (*str*) – Payload encoding.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

get(*queue*, *virtual_host='/'*, *requeue=False*, *to_dict=False*, *count=1*, *truncate=50000*, *encoding='auto'*)
Get Messages.

Parameters

- **queue** (*str*) – Queue name
- **virtual_host** (*str*) – Virtual host name
- **requeue** (*bool*) – Re-queue message
- **to_dict** (*bool*) – Should incoming messages be converted to a dictionary before delivery.
- **count** (*int*) – How many messages should we try to fetch.

- **truncate** (*int*) – The maximum length in bytes, beyond that the server will truncate the message.
- **encoding** (*str*) – Message encoding.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type list

class amqpstorm.management.channel.**Channel**(*http_client*)

get(*channel*)

Get Connection details.

Parameters **channel** – Channel name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

list()

List all Channels.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type list

class amqpstorm.management.connection.**Connection**(*http_client*)

get(*connection*)

Get Connection details.

Parameters **connection** (*str*) – Connection name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

list()
Get Connections.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type list

close(*connection*, *reason='Closed via management api'*)
Close Connection.

Parameters

- **connection** (*str*) – Connection name
- **reason** (*str*) – Reason for closing connection.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type None

class amqpstorm.management.exchange.**Exchange**(*http_client*)

get(*exchange*, *virtual_host='/'*)
Get Exchange details.

Parameters

- **exchange** (*str*) – Exchange name
- **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

list(*virtual_host='/'*, *show_all=False*)
List Exchanges.

Parameters

- **virtual_host** (*str*) – Virtual host name
- **show_all** (*bool*) – List all Exchanges

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type list

declare(*exchange=''*, *exchange_type='direct'*, *virtual_host='/'*, *passive=False*, *durable=False*, *auto_delete=False*, *internal=False*, *arguments=None*)

Declare an Exchange.

Parameters

- **exchange** (*str*) – Exchange name
- **exchange_type** (*str*) – Exchange type
- **virtual_host** (*str*) – Virtual host name
- **passive** (*bool*) – Do not create
- **durable** (*bool*) – Durable exchange
- **auto_delete** (*bool*) – Automatically delete when not in use
- **internal** (*bool*) – Is the exchange for use by the broker only.
- **arguments** (*dict, None*) – Exchange key/value arguments

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type None

delete(*exchange*, *virtual_host='/'*)

Delete an Exchange.

Parameters

- **exchange** (*str*) – Exchange name
- **virtual_host** (*str*) – Virtual host name

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type dict

bindings(*exchange*, *virtual_host='/'*)

Get Exchange bindings.

Parameters

- **exchange** (*str*) – Exchange name
- **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type list

bind(*destination=''*, *source=''*, *routing_key=''*, *virtual_host='/'*, *arguments=None*)
Bind an Exchange.

Parameters

- **source** (*str*) – Source Exchange name
- **destination** (*str*) – Destination Exchange name
- **routing_key** (*str*) – The routing key to use
- **virtual_host** (*str*) – Virtual host name
- **arguments** (*dict, None*) – Bind key/value arguments

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type None

unbind(*destination=''*, *source=''*, *routing_key=''*, *virtual_host='/'*, *properties_key=None*)
Unbind an Exchange.

Parameters

- **source** (*str*) – Source Exchange name
- **destination** (*str*) – Destination Exchange name
- **routing_key** (*str*) – The routing key to use
- **virtual_host** (*str*) – Virtual host name
- **properties_key** (*str*) –

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type None

class amqpstorm.management.queue.**Queue**(*http_client*)

get(*queue*, *virtual_host='/'*)
Get Queue details.

Parameters

- **queue** – Queue name
- **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

list(*virtual_host='/'*, *show_all=False*)
List Queues.

Parameters

- **virtual_host** (*str*) – Virtual host name
- **show_all** (*bool*) – List all Queues

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type list

declare(*queue=''*, *virtual_host='/'*, *passive=False*, *durable=False*, *auto_delete=False*, *arguments=None*)
Declare a Queue.

Parameters

- **queue** (*str*) – Queue name
- **virtual_host** (*str*) – Virtual host name
- **passive** (*bool*) – Do not create
- **durable** (*bool*) – Durable queue
- **auto_delete** (*bool*) – Automatically delete when not in use
- **arguments** (*dict, None*) – Queue key/value arguments

Raises

- ***ApiError*** – Raises if the remote server encountered an error.

- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type dict

delete(*queue*, *virtual_host='/'*)
Delete a Queue.

Parameters

- **queue** (*str*) – Queue name
- **virtual_host** (*str*) – Virtual host name

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type dict

purge(*queue*, *virtual_host='/'*)
Purge a Queue.

Parameters

- **queue** (*str*) – Queue name
- **virtual_host** (*str*) – Virtual host name

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type None

bindings(*queue*, *virtual_host='/'*)
Get Queue bindings.

Parameters

- **queue** (*str*) – Queue name
- **virtual_host** (*str*) – Virtual host name

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type list

bind(*queue=''*, *exchange=''*, *routing_key=''*, *virtual_host='/'*, *arguments=None*)
Bind a Queue.

Parameters

- **queue** (*str*) – Queue name
- **exchange** (*str*) – Exchange name
- **routing_key** (*str*) – The routing key to use
- **virtual_host** (*str*) – Virtual host name
- **arguments** (*dict, None*) – Bind key/value arguments

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type None

unbind(*queue=''*, *exchange=''*, *routing_key=''*, *virtual_host='/'*, *properties_key=None*)

Unbind a Queue.

Parameters

- **queue** (*str*) – Queue name
- **exchange** (*str*) – Exchange name
- **routing_key** (*str*) – The routing key to use
- **virtual_host** (*str*) – Virtual host name
- **properties_key** (*str*) –

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type None

class amqpstorm.management.user.**User**(*http_client*)

get(*username*)

Get User details.

Parameters **username** (*str*) – Username

Return type dict

list()

List all Users.

Return type list

create(*username*, *password*, *tags=''*)
Create User.

Parameters

- **username** (*str*) – Username
- **password** (*str*) – Password
- **tags** (*str*) – Comma-separate list of tags (e.g. monitoring)

Return type None

delete(*username*)
Delete User.

Parameters **username** (*str*) – Username

Return type dict

get_permission(*username*, *virtual_host*)
Get User permissions for the configured virtual host.

Parameters

- **username** (*str*) – Username
- **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

get_permissions(*username*)
Get all Users permissions.

Parameters **username** (*str*) – Username

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

set_permission(*username*, *virtual_host*, *configure_regex='.*'*, *write_regex='.*'*, *read_regex='.*'*)
Set User permissions for the configured virtual host.

Parameters

- **username** (*str*) – Username

- **virtual_host** (*str*) – Virtual host name
- **configure_regex** (*str*) – Permission pattern for configuration operations for this user.
- **write_regex** (*str*) – Permission pattern for write operations for this user.
- **read_regex** (*str*) – Permission pattern for read operations for this user.

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

delete_permission(*username*, *virtual_host*)

Delete User permissions for the configured virtual host.

Parameters

- **username** (*str*) – Username
- **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

class amqpstorm.management.virtual_host.**VirtualHost**(*http_client*)

get(*virtual_host*)

Get Virtual Host details.

Parameters **virtual_host** (*str*) – Virtual host name

Raises

- ***ApiError*** – Raises if the remote server encountered an error.
- ***ApiConnectionError*** – Raises if there was a connectivity issue.

Return type dict

list()

List all Virtual Hosts.

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type list

create(*virtual_host*)
Create a Virtual Host.

Parameters **virtual_host** (*str*) – Virtual host name

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type dict

delete(*virtual_host*)
Delete a Virtual Host.

Parameters **virtual_host** (*str*) – Virtual host name

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type dict

get_permissions(*virtual_host*)
Get all Virtual hosts permissions.

Raises

- **ApiError** – Raises if the remote server encountered an error.
- **ApiConnectionError** – Raises if there was a connectivity issue.

Return type dict

3.11 Management Api Exceptions

class amqpstorm.management.**ApiConnectionError**(**args*, ***kwargs*)
Management Api Connection Error

class amqpstorm.management.**ApiError**(*message=None*, **args*, ***kwargs*)
Management Api Error

3.12 Flask RPC Client

```
"""
Example of a Flask web application using RabbitMQ for RPC calls.
"""
import threading
from time import sleep

import amqpstorm
from amqpstorm import Message
from flask import Flask

APP = Flask(__name__)


class RpcClient(object):
    """Asynchronous Rpc client."""

    def __init__(self, host, username, password, rpc_queue):
        self.queue = {}
        self.host = host
        self.username = username
        self.password = password
        self.channel = None
        self.connection = None
        self.callback_queue = None
        self.rpc_queue = rpc_queue
        self.open()

    def open(self):
        """Open Connection."""
        self.connection = amqpstorm.Connection(self.host, self.
↪username,
                                               self.password)
        self.channel = self.connection.channel()
        self.channel.queue.declare(self.rpc_queue)
        result = self.channel.queue.declare(exclusive=True)
        self.callback_queue = result['queue']
        self.channel.basic.consume(self._on_response, no_ack=True,
                                   queue=self.callback_queue)
        self._create_process_thread()

    def _create_process_thread(self):
        """Create a thread responsible for consuming messages in␣
↪response
        RPC requests.
```

(continues on next page)

(continued from previous page)

```
        """
        thread = threading.Thread(target=self._process_data_events)
        thread.setDaemon(True)
        thread.start()

    def _process_data_events(self):
        """Process Data Events using the Process Thread."""
        self.channel.start_consuming()

    def _on_response(self, message):
        """On Response store the message with the correlation id in a␣
↪local
        dictionary.
        """
        self.queue[message.correlation_id] = message.body

    def send_request(self, payload):
        # Create the Message object.
        message = Message.create(self.channel, payload)
        message.reply_to = self.callback_queue

        # Create an entry in our local dictionary, using the␣
↪automatically
        # generated correlation_id as our key.
        self.queue[message.correlation_id] = None

        # Publish the RPC request.
        message.publish(routing_key=self.rpc_queue)

        # Return the Unique ID used to identify the request.
        return message.correlation_id


@APP.route('/rpc_call/<payload>')
def rpc_call(payload):
    """Simple Flask implementation for making asynchronous Rpc calls. "
↪""

    # Send the request and store the requests Unique ID.
    corr_id = RPC_CLIENT.send_request(payload)

    # Wait until we have received a response.
    # TODO: Add a timeout here and clean up if it fails!
    while RPC_CLIENT.queue[corr_id] is None:
        sleep(0.1)
```

(continues on next page)

(continued from previous page)

```
    # Return the response to the user.
    return RPC_CLIENT.queue.pop(corr_id)


if __name__ == '__main__':
    RPC_CLIENT = RpcClient('localhost', 'guest', 'guest', 'rpc_queue')
    APP.run()
```

3.13 Robust Consumer

```
"""
Robust Consumer that will automatically re-connect on failure.
"""
import logging
import time

import amqpstorm
from amqpstorm import Connection

logging.basicConfig(level=logging.INFO)
LOGGER = logging.getLogger()


class Consumer(object):
    def __init__(self, max_retries=None):
        self.max_retries = max_retries
        self.connection = None

    def create_connection(self):
        """Create a connection.

        :return:
        """
        attempts = 0
        while True:
            attempts += 1
            try:
                self.connection = Connection('localhost', 'guest',
↪'guest')
                break
            except amqpstorm.AMQPError as why:
                LOGGER.exception(why)
```

(continues on next page)

(continued from previous page)

```
                if self.max_retries and attempts > self.max_retries:
                    break
                time.sleep(min(attempts * 2, 30))
            except KeyboardInterrupt:
                break

    def start(self):
        """Start the Consumers.

        :return:
        """
        if not self.connection:
            self.create_connection()
        while True:
            try:
                channel = self.connection.channel()
                channel.queue.declare('simple_queue')
                channel.basic.consume(self, 'simple_queue', no_
↪ack=False)
                channel.start_consuming()
                if not channel.consumer_tags:
                    channel.close()
            except amqpstorm.AMQPError as why:
                LOGGER.exception(why)
                self.create_connection()
            except KeyboardInterrupt:
                self.connection.close()
                break

    def __call__(self, message):
        print("Message:", message.body)

        # Acknowledge that we handled the message without any issues.
        message.ack()

        # Reject the message.
        # message.reject()

        # Reject the message, and put it back in the queue.
        # message.reject(requeue=True)


if __name__ == '__main__':
    CONSUMER = Consumer()
    CONSUMER.start()
```

3.14 Simple Consumer

```
"""
A simple example consuming messages from RabbitMQ.
"""
import logging

from amqpstorm import Connection

logging.basicConfig(level=logging.INFO)


def on_message(message):
    """This function is called on message received.

    :param message:
    :return:
    """
    print("Message:", message.body)

    # Acknowledge that we handled the message without any issues.
    message.ack()

    # Reject the message.
    # message.reject()

    # Reject the message, and put it back in the queue.
    # message.reject(requeue=True)


with Connection('localhost', 'guest', 'guest') as connection:
    with connection.channel() as channel:
        # Declare the Queue, 'simple_queue'.
        channel.queue.declare('simple_queue')

        # Set QoS to 100.
        # This will limit the consumer to only prefetch a 100 messages.

        # This is a recommended setting, as it prevents the
        # consumer from keeping all of the messages in a queue to␣
↪itself.
        channel.basic.qos(100)

        # Start consuming the queue 'simple_queue' using the callback
        # 'on_message' and last require the message to be acknowledged.
        channel.basic.consume(on_message, 'simple_queue', no_ack=False)
```

(continues on next page)

(continued from previous page)

```
        try:
            # Start consuming messages.
            channel.start_consuming()
        except KeyboardInterrupt:
            channel.close()
```

3.15 Simple Publisher

```
"""
A simple example publishing a message to RabbitMQ.
"""
import logging

from amqpstorm import Connection
from amqpstorm import Message

logging.basicConfig(level=logging.INFO)

with Connection('localhost', 'guest', 'guest') as connection:
    with connection.channel() as channel:
        # Declare the Queue, 'simple_queue'.
        channel.queue.declare('simple_queue')

        # Message Properties.
        properties = {
            'content_type': 'text/plain',
            'headers': {'key': 'value'}
        }

        # Create the message.
        message = Message.create(channel, 'Hello World!', properties)

        # Publish the message to a queue called, 'simple_queue'.
        message.publish('simple_queue')
```

3.16 Simple RPC Client

```
"""
A simple RPC Client.
```

(continues on next page)

(continued from previous page)

```
"""
import amqpstorm

from amqpstorm import Message


class FibonacciRpcClient(object):
    def __init__(self, host, username, password):
        """
        :param host: RabbitMQ Server e.g. localhost
        :param username: RabbitMQ Username e.g. guest
        :param password: RabbitMQ Password e.g. guest
        :return:
        """
        self.host = host
        self.username = username
        self.password = password
        self.channel = None
        self.response = None
        self.connection = None
        self.callback_queue = None
        self.correlation_id = None
        self.open()

    def open(self):
        self.connection = amqpstorm.Connection(self.host,
                                               self.username,
                                               self.password)

        self.channel = self.connection.channel()

        result = self.channel.queue.declare(exclusive=True)
        self.callback_queue = result['queue']

        self.channel.basic.consume(self._on_response, no_ack=True,
                                   queue=self.callback_queue)

    def close(self):
        self.channel.stop_consuming()
        self.channel.close()
        self.connection.close()

    def call(self, number):
        self.response = None
        message = Message.create(self.channel, body=str(number))
```

(continues on next page)

(continued from previous page)

```
        message.reply_to = self.callback_queue
        self.correlation_id = message.correlation_id
        message.publish(routing_key='rpc_queue')

        while not self.response:
            self.channel.process_data_events()
        return int(self.response)

    def _on_response(self, message):
        if self.correlation_id != message.correlation_id:
            return
        self.response = message.body


if __name__ == '__main__':
    FIBONACCI_RPC = FibonacciRpcClient('localhost', 'guest', 'guest')

    print(" [x] Requesting fib(30)")
    RESPONSE = FIBONACCI_RPC.call(30)
    print(" [.] Got %r" % (RESPONSE,))
    FIBONACCI_RPC.close()
```

3.17 Simple RPC Server

```
"""
A simple RPC Server.
"""
import amqpstorm

from amqpstorm import Message


def fib(number):
    if number == 0:
        return 0
    elif number == 1:
        return 1
    else:
        return fib(number - 1) + fib(number - 2)


def on_request(message):
    number = int(message.body)
```

(continues on next page)

(continued from previous page)

```
    print(" [.] fib(%s)" % (number,))

    response = str(fib(number))

    properties = {
        'correlation_id': message.correlation_id
    }

    response = Message.create(message.channel, response, properties)
    response.publish(message.reply_to)

    message.ack()


if __name__ == '__main__':
    CONNECTION = amqpstorm.Connection('localhost', 'guest', 'guest')
    CHANNEL = CONNECTION.channel()

    CHANNEL.queue.declare(queue='rpc_queue')
    CHANNEL.basic.qos(prefetch_count=1)
    CHANNEL.basic.consume(on_request, queue='rpc_queue')

    print(" [x] Awaiting RPC requests")
    CHANNEL.start_consuming()
```

3.18 SSL connection

```
"""
Example of connecting to RabbitMQ using a SSL Certificate.
"""
import logging
import ssl

from amqpstorm import Connection

logging.basicConfig(level=logging.INFO)


def on_message(message):
    """This function is called on message received.

    :param message:
```

(continues on next page)

(continued from previous page)

```
    :return:
    """
    print("Message:", message.body)

    # Acknowledge that we handled the message without any issues.
    message.ack()

    # Reject the message.
    # message.reject()

    # Reject the message, and put it back in the queue.
    # message.reject(requeue=True)


SSL_OPTIONS = {
    'context': ssl.create_default_context(cafile='cacert.pem'),
    'server_hostname': 'rmq.eandersson.net'
}

with Connection('rmq.eandersson.net', 'guest', 'guest', port=5671,
                ssl=True, ssl_options=SSL_OPTIONS) as connection:
    with connection.channel() as channel:
        # Declare the Queue, 'simple_queue'.
        channel.queue.declare('simple_queue')

        # Set QoS to 100.
        # This will limit the consumer to only prefetch a 100 messages.

        # This is a recommended setting, as it prevents the
        # consumer from keeping all of the messages in a queue to␣
↪itself.
        channel.basic.qos(100)

        # Start consuming the queue 'simple_queue' using the callback
        # 'on_message' and last require the message to be acknowledged.
        channel.basic.consume(on_message, 'simple_queue', no_ack=False)

        try:
            # Start consuming messages.
            channel.start_consuming()
        except KeyboardInterrupt:
            channel.close()
```

CHAPTER 4

Issues

Please report any issues on Github here

CHAPTER 5

Source

AMQPStorm source code is available on Github here

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

Index

D

E

F

G

I

J

L

M

N

O

P