Kombu Documentation

Release 4.6.7

Ask Solem
contributors

Dec 07, 2019

CONTENTS

Contents:

CHAPTER

ONE

GETTING STARTED

Version 4.6.7

Web https://kombu.readthedocs.io/

Download https://pypi.org/project/kombu/

Source https://github.com/celery/kombu/

Keywords messaging, amqp, rabbitmq, redis, mongodb, python, queue

1.1 About

Kombu is a messaging library for Python.

The aim of *Kombu* is to make messaging in Python as easy as possible by providing an idiomatic high-level interface for the AMQ protocol, and also provide proven and tested solutions to common messaging problems.

AMQP is the Advanced Message Queuing Protocol, an open standard protocol for message orientation, queuing, routing, reliability and security, for which the RabbitMQ messaging server is the most popular implementation.

1.2 Features

- Allows application authors to support several message server solutions by using pluggable transports.
  - AMQP transport using the py-amqp, librabbitmq, or qpid-python libraries.
  - High performance AMQP transport written in C - when using librabbitmq

    This is automatically enabled if librabbitmq is installed:

    ```
    $ pip install librabbitmq
    ```

  - Virtual transports makes it really easy to add support for non-AMQP transports. There is already built-in support for Redis, Amazon SQS, Azure Storage Queues, Azure Service Bus, ZooKeeper, SoftLayer MQ and Pyro.
  - In-memory transport for unit testing.
- Supports automatic encoding, serialization and compression of message payloads.
- Consistent exception handling across transports.
- The ability to ensure that an operation is performed by gracefully handling connection and channel errors.

- Several annoyances with amqplib has been fixed, like supporting timeouts and the ability to wait for events on more than one channel.
- Projects already using carrot can easily be ported by using a compatibility layer.

For an introduction to AMQP you should read the article Rabbits and warrens, and the Wikipedia article about AMQP.

1.3 Transport Comparison

Client	Type	Direct	Topic	Fanout	Priority	TTL
amqp	Native	Yes	Yes	Yes	Yes[3]	Yes[4]
qpid	Native	Yes	Yes	Yes	No	No
redis	Virtual	Yes	Yes	Yes (PUB/SUB)	Yes	No
mongodb	Virtual	Yes	Yes	Yes	Yes	Yes
SQS	Virtual	Yes	Yes[1]	Yes[2]	No	No
zookeeper	Virtual	Yes	Yes[1]	No	Yes	No
in-memory	Virtual	Yes	Yes[1]	No	No	No
SLMQ	Virtual	Yes	Yes[1]	No	No	No
Pyro	Virtual	Yes	Yes[1]	No	No	No

1.3.1 Documentation

Kombu is using Sphinx, and the latest documentation can be found here:

https://kombu.readthedocs.io/

1.3.2 Quick overview

```
from kombu import Connection, Exchange, Queue

media_exchange = Exchange('media', 'direct', durable=True)
video_queue = Queue('video', exchange=media_exchange, routing_key='video')

def process_media(body, message):
    print body
    message.ack()

# connections
with Connection('amqp://guest:guest@localhost//') as conn:

    # produce
    producer = conn.Producer(serializer='json')
    producer.publish({'name': '/tmp/lolcat1.avi', 'size': 1301013},
                     exchange=media_exchange, routing_key='video',
                     declare=[video_queue])
```

(continues on next page)

[3] AMQP Message priority support depends on broker implementation.
[4] AMQP Message/Queue TTL support depends on broker implementation.
[1] Declarations only kept in memory, so exchanges/queues must be declared by all clients that needs them.
[2] Fanout supported via storing routing tables in SimpleDB. Disabled by default, but can be enabled by using the `supports_fanout` transport option.

(continued from previous page)

```
    # the declare above, makes sure the video queue is declared
    # so that the messages can be delivered.
    # It's a best practice in Kombu to have both publishers and
    # consumers declare the queue. You can also declare the
    # queue manually using:
    #     video_queue(conn).declare()

    # consume
    with conn.Consumer(video_queue, callbacks=[process_media]) as consumer:
        # Process messages and handle events on all channels
        while True:
            conn.drain_events()

# Consume from several queues on the same channel:
video_queue = Queue('video', exchange=media_exchange, key='video')
image_queue = Queue('image', exchange=media_exchange, key='image')

with connection.Consumer([video_queue, image_queue],
                         callbacks=[process_media]) as consumer:
    while True:
        connection.drain_events()
```

Or handle channels manually:

```
with connection.channel() as channel:
    producer = Producer(channel, ...)
    consumer = Producer(channel)
```

All objects can be used outside of with statements too, just remember to close the objects after use:

```
from kombu import Connection, Consumer, Producer

connection = Connection()
    # ...
connection.release()

consumer = Consumer(channel_or_connection, ...)
consumer.register_callback(my_callback)
consumer.consume()
    # ....
consumer.cancel()
```

Exchange and *Queue* are simply declarations that can be pickled and used in configuration files etc.

They also support operations, but to do so they need to be bound to a channel.

Binding exchanges and queues to a connection will make it use that connections default channel.

```
>>> exchange = Exchange('tasks', 'direct')

>>> connection = Connection()
>>> bound_exchange = exchange(connection)
>>> bound_exchange.delete()

# the original exchange is not affected, and stays unbound.
>>> exchange.delete()
raise NotBoundError: Can't call delete on Exchange not bound to
    a channel.
```

1.4 Terminology

There are some concepts you should be familiar with before starting:

- Producers

  Producers sends messages to an exchange.

- Exchanges

  Messages are sent to exchanges. Exchanges are named and can be configured to use one of several routing algorithms. The exchange routes the messages to consumers by matching the routing key in the message with the routing key the consumer provides when binding to the exchange.

- Consumers

  Consumers declares a queue, binds it to a exchange and receives messages from it.

- Queues

  Queues receive messages sent to exchanges. The queues are declared by consumers.

- Routing keys

  Every message has a routing key. The interpretation of the routing key depends on the exchange type. There are four default exchange types defined by the AMQP standard, and vendors can define custom types (so see your vendors manual for details).

  These are the default exchange types defined by AMQP/0.8:

  - Direct exchange

    Matches if the routing key property of the message and the *routing_key* attribute of the consumer are identical.

  - Fan-out exchange

    Always matches, even if the binding does not have a routing key.

  - Topic exchange

    Matches the routing key property of the message by a primitive pattern matching scheme. The message routing key then consists of words separated by dots (*"."*, like domain names), and two special characters are available; star (*"*"*) and hash (*"#"*). The star matches any word, and the hash matches zero or more words. For example *"*.stock.#"* matches the routing keys *"usd.stock"* and *"eur.stock.db"* but not *"stock.nasdaq"*.

1.5 Installation

You can install *Kombu* either via the Python Package Index (PyPI) or from source.

To install using *pip*,:

```
$ pip install kombu
```

To install using *easy_install*,:

```
$ easy_install kombu
```

If you have downloaded a source tarball you can install it by doing the following,:

```
$ python setup.py build
# python setup.py install # as root
```

1.6 Getting Help

1.6.1 Mailing list

Join the carrot-users mailing list.

1.7 Bug tracker

If you have any suggestions, bug reports or annoyances please report them to our issue tracker at http://github.com/celery/kombu/issues/

1.8 Contributing

Development of *Kombu* happens at Github: http://github.com/celery/kombu

You are highly encouraged to participate in the development. If you don't like Github (for some reason) you're welcome to send regular patches.

1.9 License

This software is licensed under the *New BSD License*. See the *LICENSE* file in the top distribution directory for the full license text.

CHAPTER

TWO

USER GUIDE

Release 4.6

Date Dec 07, 2019

2.1 Introduction

2.1.1 What is messaging?

In times long ago people didn't have email. They had the postal service, which with great courage would deliver mail from hand to hand all over the globe. Soldiers deployed at wars far away could only communicate with their families through the postal service, and posting a letter would mean that the recipient wouldn't actually receive the letter until weeks or months, sometimes years later.

It's hard to imagine this today when people are expected to be available for phone calls every minute of the day.

So humans need to communicate with each other, this shouldn't be news to anyone, but why would applications?

One example is banks. When you transfer money from one bank to another, your bank sends a message to a central clearinghouse. The clearinghouse then records and coordinates the transaction. Banks need to send and receive millions and millions of messages every day, and losing a single message would mean either losing your money (bad) or the banks money (very bad)

Another example is the stock exchanges, which also have a need for very high message throughputs and have strict reliability requirements.

Email is a great way for people to communicate. It is much faster than using the postal service, but still using email as a means for programs to communicate would be like the soldier above, waiting for signs of life from his girlfriend back home.

2.1.2 Messaging Scenarios

- Request/Reply

  The request/reply pattern works like the postal service example. A message is addressed to a single recipient, with a return address printed on the back. The recipient may or may not reply to the message by sending it back to the original sender.

  Request-Reply is achieved using *direct* exchanges.

- Broadcast

  In a broadcast scenario a message is sent to all parties. This could be none, one or many recipients.

  Broadcast is achieved using *fanout* exchanges.

- Publish/Subscribe

  In a publish/subscribe scenario producers publish messages to topics, and consumers subscribe to the topics they are interested in.

  If no consumers subscribe to the topic, then the message will not be delivered to anyone. If several consumers subscribe to the topic, then the message will be delivered to all of them.

  Pub-sub is achieved using *topic* exchanges.

2.1.3 Reliability

For some applications reliability is very important. Losing a message is a critical situation that must never happen. For other applications losing a message is fine, it can maybe recover in other ways, or the message is resent anyway as periodic updates.

AMQP defines two built-in delivery modes:

- persistent

  Messages are written to disk and survives a broker restart.

- transient

  Messages may or may not be written to disk, as the broker sees fit to optimize memory contents. The messages won't survive a broker restart.

Transient messaging is by far the fastest way to send and receive messages, so having persistent messages comes with a price, but for some applications this is a necessary cost.

2.2 Connections and transports

2.2.1 Basics

To send and receive messages you need a transport and a connection. There are several transports to choose from (amqp, librabbitmq, redis, qpid, in-memory, etc.), and you can even create your own. The default transport is amqp.

Create a connection using the default transport:

```
>>> from kombu import Connection
>>> connection = Connection('amqp://guest:guest@localhost:5672//')
```

The connection will not be established yet, as the connection is established when needed. If you want to explicitly establish the connection you have to call the *connect()* method:

```
>>> connection.connect()
```

You can also check whether the connection is connected:

```
>>> connection.connected
True
```

Connections must always be closed after use:

```
>>> connection.close()
```

But best practice is to release the connection instead, this will release the resource if the connection is associated with a connection pool, or close the connection if not, and makes it easier to do the transition to connection pools later:

```
>>> connection.release()
```

See also:

Connection and Producer Pools

Of course, the connection can be used as a context, and you are encouraged to do so as it makes it harder to forget releasing open resources:

```
with Connection() as connection:
    # work with connection
```

2.2.2 URLs

Connection parameters can be provided as a URL in the format:

```
transport://userid:password@hostname:port/virtual_host
```

All of these are valid URLs:

```
# Specifies using the amqp transport only, default values
# are taken from the keyword arguments.
amqp://

# Using Redis
redis://localhost:6379/

# Using Redis over a Unix socket
redis+socket:///tmp/redis.sock

# Using Qpid
qpid://localhost/

# Using virtual host '/foo'
amqp://localhost//foo

# Using virtual host 'foo'
amqp://localhost/foo

# Using Pyro with name server running on 'localhost'
pyro://localhost/kombu.broker
```

The query part of the URL can also be used to set options, e.g.:

```
amqp://localhost/myvhost?ssl=1
```

See *Keyword arguments* for a list of supported options.

A connection without options will use the default connection settings, which is using the localhost host, default port, user name *guest*, password *guest* and virtual host "/". A connection without arguments is the same as:

```
>>> Connection('amqp://guest:guest@localhost:5672//')
```

The default port is transport specific, for AMQP this is 5672.

Other fields may also have different meaning depending on the transport used. For example, the Redis transport uses the *virtual_host* argument as the redis database number.

2.2.3 Keyword arguments

The *Connection* class supports additional keyword arguments, these are:

hostname Default host name if not provided in the URL.

userid Default user name if not provided in the URL.

password Default password if not provided in the URL.

virtual_host Default virtual host if not provided in the URL.

port Default port if not provided in the URL.

transport Default transport if not provided in the URL. Can be a string specifying the path to the class. (e.g. `kombu.transport.pyamqp:Transport`), or one of the aliases: `pyamqp`, `librabbitmq`, `redis`, `qpid`, `memory`, and so on.

ssl Use SSL to connect to the server. Default is `False`. Only supported by the amqp and qpid transports.

insist Insist on connecting to a server. *No longer supported, relic from AMQP 0.8*

connect_timeout Timeout in seconds for connecting to the server. May not be supported by the specified transport.

transport_options A dict of additional connection arguments to pass to alternate kombu channel implementations. Consult the transport documentation for available options.

2.2.4 AMQP Transports

There are 4 transports available for AMQP use.

1. `pyamqp` uses the pure Python library `amqp`, automatically installed with Kombu.
2. `librabbitmq` uses the high performance transport written in C. This requires the `librabbitmq` Python package to be installed, which automatically compiles the C library.
3. `amqp` tries to use `librabbitmq` but falls back to `pyamqp`.
4. `qpid` uses the pure Python library `qpid.messaging`, automatically installed with Kombu. The Qpid library uses AMQP, but uses custom extensions specifically supported by the Apache Qpid Broker.

For the highest performance, you should install the `librabbitmq` package. To ensure librabbitmq is used, you can explicitly specify it in the transport URL, or use `amqp` to have the fallback.

2.2.5 Transport Comparison

Client	Type	Direct	Topic	Fanout	Priority
amqp	Native	Yes	Yes	Yes	Yes[3]
qpid	Native	Yes	Yes	Yes	No
redis	Virtual	Yes	Yes	Yes (PUB/SUB)	Yes
SQS	Virtual	Yes	Yes[1]	Yes[2]	No
zookeeper	Virtual	Yes	Yes[1]	No	Yes
in-memory	Virtual	Yes	Yes[1]	No	No
SLMQ	Virtual	Yes	Yes[1]	No	No

2.3 Producers

2.3.1 Basics

You can create a producer using a *Connection*:

```
>>> producer = connection.Producer()
```

You can also instantiate *Producer* directly, it takes a channel or a connection as an argument:

```
>>> with Connection('amqp://') as conn:
...     with conn.channel() as channel:
...         producer = Producer(channel)
```

Having a producer instance you can publish messages:

Mostly you will be getting a connection from a connection pool, and this connection can be stale, or you could lose the connection in the middle of sending the message. Using retries is a good way to handle these intermittent failures:

```
>>> producer.publish({'hello': 'world', ..., retry=True})
```

In addition a retry policy can be specified, which is a dictionary of parameters supported by the `retry_over_time()` function

```
>>> producer.publish(
...     {'hello': 'world'}, ...,
...     retry=True,
...     retry_policy={
...         'interval_start': 0,  # First retry immediately,
...         'interval_step': 2,   # then increase by 2s for every retry.
...         'interval_max': 30,   # but don't exceed 30s between retries.
...         'max_retries': 30,    # give up after 30 tries.
...     },
... )
```

The `declare` argument lets you pass a list of entities that must be declared before sending the message. This is especially important when using the `retry` flag, since the broker may actually restart during a retry in which case non-durable entities are removed.

Say you are writing a task queue, and the workers may have not started yet so the queues aren't declared. In this case you need to define both the exchange, and the declare the queue so that the message is delivered to the queue while the workers are offline:

```
>>> from kombu import Exchange, Queue
>>> task_queue = Queue('tasks', Exchange('tasks'), routing_key='tasks')

>>> producer.publish(
...     {'hello': 'world'}, ...,
...     retry=True,
...     exchange=task_queue.exchange,
...     routing_key=task_queue.routing_key,
...     declare=[task_queue],  # declares exchange, queue and binds.
... )
```

[3] AMQP Message priority support depends on broker implementation.

[1] Declarations only kept in memory, so exchanges/queues must be declared by all clients that needs them.

[2] Fanout supported via storing routing tables in SimpleDB. Disabled by default, but can be enabled by using the `supports_fanout` transport option.

Bypassing routing by using the anon-exchange

You may deliver to a queue directly, bypassing the brokers routing mechanisms, by using the "anon-exchange": set the exchange parameter to the empty string, and set the routing key to be the name of the queue:

```
>>> producer.publish(
...     {'hello': 'world'},
...     exchange='',
...     routing_key=task_queue.name,
... )
```

2.3.2 Serialization

Json is the default serializer when a non-string object is passed to publish, but you can also specify a different serializer:

```
>>> producer.publish({'hello': 'world'}, serializer='pickle')
```

See *Serialization* for more information.

2.3.3 Reference

class kombu.**Producer**(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)

Message Producer.

Parameters

- **channel** (*kombu.Connection, ChannelT*) – Connection or channel.
- **exchange** (*kombu.entity.Exchange, str*) – Optional default exchange.
- **routing_key** (*str*) – Optional default routing key.
- **serializer** (*str*) – Default serializer. Default is *"json"*.
- **compression** (*str*) – Default compression method. Default is no compression.
- **auto_declare** (*bool*) – Automatically declare the default exchange at instantiation. Default is `True`.
- **on_return** (`Callable`) – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to *publish()* is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

auto_declare = True

By default, if a defualt exchange is set, that exchange will be declare when publishing a message.

compression = None

Default compression method. Disabled by default.

declare()

Declare the exchange.

> **Note:** This happens automatically at instantiation when the *auto_declare* flag is enabled.

exchange = None
Default exchange

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare exchange if not already declared during this session.

on_return = None
Basic return callback.

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=None*, *expiration=None*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** (*Any*) – Message body.
- **routing_key** (*str*) – Message routing key.
- **delivery_mode** (*enum*) – See `delivery_mode`.
- **mandatory** (*bool*) – Currently not supported.
- **immediate** (*bool*) – Currently not supported.
- **priority** (*int*) – Message priority. A number between 0 and 9.
- **content_type** (*str*) – Content type. Default is auto-detect.
- **content_encoding** (*str*) – Content encoding. Default is auto-detect.
- **serializer** (*str*) – Serializer to use. Default is auto-detect.
- **compression** (*str*) – Compression method to use. Default is none.
- **headers** (*Dict*) – Mapping of arbitrary headers to pass along with the message body.
- **exchange** (*kombu.entity.Exchange, str*) – Override the exchange. Note that this exchange must have been declared.
- **declare** (*Sequence[EntityT]*) – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using *maybe_declare()*.
- **retry** (*bool*) – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** (*Dict*) – Retry configuration, this is the keywords supported by *ensure()*.
- **expiration** (*float*) – A TTL in seconds can be specified per message. Default is no expiration.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

revive(*channel*)
Revive the producer after connection loss.

routing_key = ''
Default routing key.

serializer = None
Default serializer to use. Default is JSON.

2.4 Consumers

2.4.1 Basics

The `Consumer` takes a connection (or channel) and a list of queues to consume from. Several consumers can be mixed to consume from different channels, as they all bind to the same connection, and `drain_events` will drain events from all channels on that connection.

Note: Kombu since 3.0 will only accept json/binary or text messages by default, to allow deserialization of other formats you have to specify them in the `accept` argument (in addition to setting the right content type for your messages):

```
Consumer(conn, accept=['json', 'pickle', 'msgpack', 'yaml'])
```

Draining events from a single consumer:

```
with Consumer(connection, queues, accept=['json']):
    connection.drain_events(timeout=1)
```

Draining events from several consumers:

```
from kombu.utils.compat import nested

with connection.channel(), connection.channel() as (channel1, channel2):
    with nested(Consumer(channel1, queues1, accept=['json']),
                Consumer(channel2, queues2, accept=['json'])):
        connection.drain_events(timeout=1)
```

Or using *ConsumerMixin*:

```
from kombu.mixins import ConsumerMixin

class C(ConsumerMixin):

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, Consumer, channel):
        return [
            Consumer(queues, callbacks=[self.on_message], accept=['json']),
        ]

    def on_message(self, body, message):
        print('RECEIVED MESSAGE: {0!r}'.format(body))
        message.ack()

C(connection).run()
```

and with multiple channels again:

```
from kombu import Consumer
from kombu.mixins import ConsumerMixin

class C(ConsumerMixin):
```

(continues on next page)

(continued from previous page)

```
    channel2 = None

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, _, default_channel):
        self.channel2 = default_channel.connection.channel()
        return [Consumer(default_channel, queues1,
                         callbacks=[self.on_message],
                         accept=['json']),
                Consumer(self.channel2, queues2,
                         callbacks=[self.on_special_message],
                         accept=['json'])]

    def on_consumer_end(self, connection, default_channel):
        if self.channel2:
            self.channel2.close()

C(connection).run()
```

There's also a *ConsumerProducerMixin* for consumers that need to also publish messages on a separate connection (e.g. sending rpc replies, streaming results):

```
from kombu import Producer, Queue
from kombu.mixins import ConsumerProducerMixin

rpc_queue = Queue('rpc_queue')

class Worker(ConsumerProducerMixin):

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, Consumer, channel):
        return [Consumer(
            queues=[rpc_queue],
            on_message=self.on_request,
            accept={'application/json'},
            prefetch_count=1,
        )]

    def on_request(self, message):
        n = message.payload['n']
        print(' [.] fib({0})'.format(n))
        result = fib(n)

        self.producer.publish(
            {'result': result},
            exchange='', routing_key=message.properties['reply_to'],
            correlation_id=message.properties['correlation_id'],
            serializer='json',
            retry=True,
        )
        message.ack()
```

See also:

`examples/rpc-tut6/` in the Github repository.

2.4.2 Advanced Topics

RabbitMQ

Consumer Priorities

RabbitMQ defines a consumer priority extension to the amqp protocol, that can be enabled by setting the `x-priority` argument to `basic.consume`.

In kombu you can specify this argument on the *Queue*, like this:

```
queue = Queue('name', Exchange('exchange_name', type='direct'),
              consumer_arguments={'x-priority': 10})
```

Read more about consumer priorities here: https://www.rabbitmq.com/consumer-priority.html

2.4.3 Reference

class kombu.**Consumer**(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*, *on_message=None*, *accept=None*, *prefetch_count=None*, *tag_prefix=None*)

Message consumer.

Parameters

- **channel** (*kombu.Connection, ChannelT*) – see *channel*.
- **queues** (*Sequence[kombu.Queue]*) – see *queues*.
- **no_ack** (*bool*) – see *no_ack*.
- **auto_declare** (*bool*) – see *auto_declare*
- **callbacks** (*Sequence[Callable]*) – see *callbacks*.
- **on_message** (*Callable*) – See *on_message*
- **on_decode_error** (*Callable*) – see *on_decode_error*.
- **prefetch_count** (*int*) – see `prefetch_count`.

exception ContentDisallowed

Consumer does not allow this content-type.

accept = None

List of accepted content-types.

An exception will be raised if the consumer receives a message with an untrusted content type. By default all content-types are accepted, but not if `kombu.disable_untrusted_serializers()` was called, in which case only json is allowed.

add_queue(*queue*)

Add a queue to the list of queues to consume from.

Note: This will not start consuming from the queue, for that you will have to call *consume()* after.

auto_declare = True

By default all entities will be declared at instantiation, if you want to handle this manually you can set this to `False`.

callbacks = None
List of callbacks called in order when a message is received.

The signature of the callbacks must take two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

cancel()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

cancel_by_queue(*queue*)
Cancel consumer by queue name.

channel = None
The connection/channel to use for this consumer.

close()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

consume(*no_ack=None*)
Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use *cancel_by_queue()*).

Parameters **no_ack** (*bool*) – See *no_ack*.

consuming_from(*queue*)
Return `True` if currently consuming from queue'.

declare()
Declare queues, exchanges and bindings.

Note: This is done automatically at instantiation when *auto_declare* is set.

---

flow(*active*)
Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

no_ack = None
Flag for automatic message acknowledgment. If enabled the messages are automatically acknowledged by the broker. This can increase performance but means that you have no control of when the message is removed.

Disabled by default.

on_decode_error = None
Callback called when a message can't be decoded.

The signature of the callback must take two arguments: *(message, exc)*, which is the message that can't be decoded and the exception that occurred while trying to decode it.

on_message = None
Optional function called whenever a message is received.

When defined this function will be called instead of the *receive()* method, and *callbacks* will be disabled.

So this can be used as an alternative to *callbacks* when you don't want the body to be automatically decoded. Note that the message will still be decompressed if the message has the `compression` header set.

The signature of the callback must take a single argument, which is the `Message` object.

Also note that the `message.body` attribute, which is the raw contents of the message body, may in some cases be a read-only `buffer` object.

prefetch_count = None
Initial prefetch count

If set, the consumer will set the prefetch_count QoS value at startup. Can also be changed using *qos()*.

purge()
Purge messages from all queues.

> **Warning:** This will *delete all ready messages*, there is no undo operation.

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the *no_ack* option is set.

Parameters

- **prefetch_size** (*int*) – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
- **prefetch_count** (*int*) – Specify the prefetch window in terms of whole messages.
- **apply_global** (*bool*) – Apply new settings globally on all channels.

property queues
A single *Queue*, or a list of queues to consume from.

receive(*body*, *message*)
Method called when a message is received.

This dispatches to the registered *callbacks*.

Parameters

- **body** (*Any*) – The decoded message body.

- **message** (*Message*) – The message instance.

Raises **NotImplementedError** – If no consumer callbacks have been registered.

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

Asks the broker to redeliver all unacknowledged messages on the specified channel.

Parameters **requeue** (*bool*) – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

register_callback(*callback*)
Register a new callback to be called when a message is received.

Note: The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

---

revive(*channel*)
Revive consumer after connection loss.

2.5 Examples

2.5.1 Hello World Example

Below example uses *Simple Interface* to send helloworld message through message broker (rabbitmq) and print received message

`hello_publisher.py`:

```
from __future__ import absolute_import, unicode_literals

import datetime

from kombu import Connection


with Connection('amqp://guest:guest@localhost:5672//') as conn:
    simple_queue = conn.SimpleQueue('simple_queue')
    message = 'helloworld, sent at {0}'.format(datetime.datetime.today())
    simple_queue.put(message)
    print('Sent: {0}'.format(message))
    simple_queue.close()
```

`hello_consumer.py`:

```
from __future__ import absolute_import, unicode_literals, print_function

from kombu import Connection  # noqa


with Connection('amqp://guest:guest@localhost:5672//') as conn:
    simple_queue = conn.SimpleQueue('simple_queue')
    message = simple_queue.get(block=True, timeout=1)
```

(continues on next page)

(continued from previous page)

```
    print('Received: {0}'.format(message.payload))
    message.ack()
    simple_queue.close()
```

2.5.2 Task Queue Example

Very simple task queue using pickle, with primitive support for priorities using different queues.

queues.py:

```
from __future__ import absolute_import, unicode_literals

from kombu import Exchange, Queue

task_exchange = Exchange('tasks', type='direct')
task_queues = [Queue('hipri', task_exchange, routing_key='hipri'),
               Queue('midpri', task_exchange, routing_key='midpri'),
               Queue('lopri', task_exchange, routing_key='lopri')]
```

worker.py:

```
from __future__ import absolute_import, unicode_literals

from kombu.mixins import ConsumerMixin
from kombu.log import get_logger
from kombu.utils.functional import reprcall

from .queues import task_queues

logger = get_logger(__name__)


class Worker(ConsumerMixin):

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, Consumer, channel):
        return [Consumer(queues=task_queues,
                         accept=['pickle', 'json'],
                         callbacks=[self.process_task])]

    def process_task(self, body, message):
        fun = body['fun']
        args = body['args']
        kwargs = body['kwargs']
        logger.info('Got task: %s', reprcall(fun.__name__, args, kwargs))
        try:
            fun(*args, **kwargs)
        except Exception as exc:
            logger.error('task raised exception: %r', exc)
        message.ack()

if __name__ == '__main__':
    from kombu import Connection
```

(continues on next page)

(continued from previous page)

```
    from kombu.utils.debug import setup_logging
    # setup root logger
    setup_logging(loglevel='INFO', loggers=[''])

    with Connection('amqp://guest:guest@localhost:5672//') as conn:
        try:
            worker = Worker(conn)
            worker.run()
        except KeyboardInterrupt:
            print('bye bye')
```

tasks.py:

```
from __future__ import absolute_import, unicode_literals


def hello_task(who='world'):
    print('Hello {0}'.format(who))
```

client.py:

```
from __future__ import absolute_import, unicode_literals

from kombu.pools import producers

from .queues import task_exchange

priority_to_routing_key = {
    'high': 'hipri',
    'mid': 'midpri',
    'low': 'lopri',
}


def send_as_task(connection, fun, args=(), kwargs={}, priority='mid'):
    payload = {'fun': fun, 'args': args, 'kwargs': kwargs}
    routing_key = priority_to_routing_key[priority]

    with producers[connection].acquire(block=True) as producer:
        producer.publish(payload,
                         serializer='pickle',
                         compression='bzip2',
                         exchange=task_exchange,
                         declare=[task_exchange],
                         routing_key=routing_key)

if __name__ == '__main__':
    from kombu import Connection
    from .tasks import hello_task

    connection = Connection('amqp://guest:guest@localhost:5672//')
    send_as_task(connection, fun=hello_task, args=('Kombu',), kwargs={},
                 priority='high')
```

2.6 Simple Interface

- *Sending and receiving messages*

kombu.simple is a simple interface to AMQP queueing. It is only slightly different from the `Queue` class in the Python Standard Library, which makes it excellent for users with basic messaging needs.

Instead of defining exchanges and queues, the simple classes only requires two arguments, a connection channel and a name. The name is used as the queue, exchange and routing key. If the need arises, you can specify a *Queue* as the name argument instead.

In addition, the *Connection* comes with shortcuts to create simple queues using the current connection:

```
>>> queue = connection.SimpleQueue('myqueue')
>>> # ... do something with queue
>>> queue.close()
```

This is equivalent to:

```
>>> from kombu.simple import SimpleQueue, SimpleBuffer

>>> channel = connection.channel()
>>> queue = SimpleBuffer(channel)
>>> # ... do something with queue
>>> channel.close()
>>> queue.close()
```

2.6.1 Sending and receiving messages

The simple interface defines two classes; *SimpleQueue*, and *SimpleBuffer*. The former is used for persistent messages, and the latter is used for transient, buffer-like queues. They both have the same interface, so you can use them interchangeably.

Here is an example using the *SimpleQueue* class to produce and consume logging messages:

```
import socket
import datetime
from time import time
from kombu import Connection


class Logger(object):

    def __init__(self, connection, queue_name='log_queue',
            serializer='json', compression=None):
        self.queue = connection.SimpleQueue(queue_name)
        self.serializer = serializer
        self.compression = compression

    def log(self, message, level='INFO', context={}):
        self.queue.put({'message': message,
                        'level': level,
                        'context': context,
                        'hostname': socket.gethostname(),
```

(continues on next page)

(continued from previous page)

```
                        'timestamp': time()},
                        serializer=self.serializer,
                        compression=self.compression)

    def process(self, callback, n=1, timeout=1):
        for i in xrange(n):
            log_message = self.queue.get(block=True, timeout=1)
            entry = log_message.payload # deserialized data.
            callback(entry)
            log_message.ack() # remove message from queue

    def close(self):
        self.queue.close()


if __name__ == '__main__':
    from contextlib import closing

    with Connection('amqp://guest:guest@localhost:5672//') as conn:
        with closing(Logger(conn)) as logger:

            # Send message
            logger.log('Error happened while encoding video',
                       level='ERROR',
                       context={'filename': 'cutekitten.mpg'})

            # Consume and process message

            # This is the callback called when a log message is
            # received.
            def dump_entry(entry):
                date = datetime.datetime.fromtimestamp(entry['timestamp'])
                print('[%s %s %s] %s %r' % (date,
                                            entry['hostname'],
                                            entry['level'],
                                            entry['message'],
                                            entry['context']))

            # Process a single message using the callback above.
            logger.process(dump_entry, n=1)
```

2.7 Connection and Producer Pools

2.7.1 Default Pools

Kombu ships with two global pools: one connection pool, and one producer pool.

These are convenient and the fact that they are global may not be an issue as connections should often be limited at the process level, rather than per thread/application and so on, but if you need custom pools per thread see *Custom Pool Groups*.

The connection pool group

The connection pools are available as `kombu.pools.connections`. This is a pool group, which means you give it a connection instance, and you get a pool instance back. We have one pool per connection instance to support multiple connections in the same app. All connection instances with the same connection parameters will get the same pool:

```
>>> from kombu import Connection
>>> from kombu.pools import connections

>>> connections[Connection('redis://localhost:6379')]
<kombu.connection.ConnectionPool object at 0x101805650>
>>> connections[Connection('redis://localhost:6379')]
<kombu.connection.ConnectionPool object at 0x101805650>
```

Let's acquire and release a connection:

```
from kombu import Connection
from kombu.pools import connections

connection = Connection('redis://localhost:6379')

with connections[connection].acquire(block=True) as conn:
    print('Got connection: {0!r}'.format(connection.as_uri()))
```

Note: The `block=True` here means that the acquire call will block until a connection is available in the pool. Note that this will block forever in case there is a deadlock in your code where a connection is not released. There is a `timeout` argument you can use to safeguard against this (see `kombu.connection.Resource.acquire()`).

If blocking is disabled and there aren't any connections left in the pool an *kombu.exceptions.ConnectionLimitExceeded* exception will be raised.

That's about it. If you need to connect to multiple brokers at once you can do that too:

```
from kombu import Connection
from kombu.pools import connections

c1 = Connection('amqp://')
c2 = Connection('redis://')

with connections[c1].acquire(block=True) as conn1:
    with connections[c2].acquire(block=True) as conn2:
        # ....
```

2.7.2 The producer pool group

This is a pool group just like the connections, except that it manages *Producer* instances used to publish messages.

Here is an example using the producer pool to publish a message to the `news` exchange:

```
from kombu import Connection, Exchange
from kombu.pools import producers

# The exchange we send our news articles to.
```

(continues on next page)

(continued from previous page)

```
news_exchange = Exchange('news')

# The article we want to send
article = {'title': 'No cellular coverage on the tube for 2012',
           'ingress': 'yadda yadda yadda'}

# The broker where our exchange is.
connection = Connection('amqp://guest:guest@localhost:5672//')

with producers[connection].acquire(block=True) as producer:
    producer.publish(
        article,
        exchange=news_exchange,
        routing_key='domestic',
        declare=[news_exchange],
        serializer='json',
        compression='zlib')
```

Setting pool limits

By default every connection instance has a limit of 200 connections. You can change this limit using *kombu.pools.set_limit()*. You are able to grow the pool at runtime, but you can't shrink it, so it is best to set the limit as early as possible after your application starts:

```
>>> from kombu import pools
>>> pools.set_limit()
```

Resetting all pools

You can close all active connections and reset all pool groups by using the *kombu.pools.reset()* function. Note that this will not respect anything currently using these connections, so will just drag the connections away from under their feet: you should be very careful before you use this.

Kombu will reset the pools if the process is forked, so that forked processes start with clean pool groups.

2.7.3 Custom Pool Groups

To maintain your own pool groups you should create your own `Connections` and `kombu.pools.Producers` instances:

```
from kombu import pools
from kombu import Connection

connections = pools.Connections(limit=100)
producers = pools.Producers(limit=connections.limit)

connection = Connection('amqp://guest:guest@localhost:5672//')

with connections[connection].acquire(block=True):
    # ...
```

If you want to use the global limit that can be set with *set_limit()* you can use a special value as the `limit` argument:

```
from kombu import pools

connections = pools.Connections(limit=pools.use_default_limit)
```

2.8 Serialization

2.8.1 Serializers

By default every message is encoded using JSON, so sending Python data structures like dictionaries and lists works. YAML, msgpack and Python's built-in *pickle* module is also supported, and if needed you can register any custom serialization scheme you want to use.

By default Kombu will only load JSON messages, so if you want to use other serialization format you must explicitly enable them in your consumer by using the `accept` argument:

```
Consumer(conn, [queue], accept=['json', 'pickle', 'msgpack'])
```

The accept argument can also include MIME-types.

Each option has its advantages and disadvantages.

***json* – JSON is supported in many programming languages, is now** a standard part of Python (since 2.6), and is fairly fast to decode using the modern Python libraries such as *cjson* or *simplejson*.

The primary disadvantage to *JSON* is that it limits you to the following data types: strings, Unicode, floats, boolean, dictionaries, and lists. Decimals and dates are notably missing.

Also, binary data will be transferred using Base64 encoding, which will cause the transferred data to be around 34% larger than an encoding which supports native binary types.

However, if your data fits inside the above constraints and you need cross-language support, the default setting of *JSON* is probably your best choice.

***pickle* – If you have no desire to support any language other than** Python, then using the *pickle* encoding will gain you the support of all built-in Python data types (except class instances), smaller messages when sending binary files, and a slight speedup over *JSON* processing.

Pickle and Security

The pickle format is very convenient as it can serialize and deserialize almost any object, but this is also a concern for security.

Carefully crafted pickle payloads can do almost anything a regular Python program can do, so if you let your consumer automatically decode pickled objects you must make sure to limit access to the broker so that untrusted parties do not have the ability to send messages!

By default Kombu uses pickle protocol 2, but this can be changed using the `PICKLE_PROTOCOL` environment variable or by changing the global `kombu.serialization.pickle_protocol` flag.

***yaml* – YAML has many of the same characteristics as *json*,** except that it natively supports more data types (including dates, recursive references, etc.)

However, the Python libraries for YAML are a good bit slower than the libraries for JSON.

If you need a more expressive set of data types and need to maintain cross-language compatibility, then *YAML* may be a better fit than the above.

To instruct *Kombu* to use an alternate serialization method, use one of the following options.

1. Set the serialization option on a per-producer basis:

```
>>> producer = Producer(channel,
...                     exchange=exchange,
...                     serializer='yaml')
```

2. Set the serialization option per message:

```
>>> producer.publish(message, routing_key=rkey,
...                  serializer='pickle')
```

Note that a *Consumer* do not need the serialization method specified. They can auto-detect the serialization method as the content-type is sent as a message header.

2.8.2 Sending raw data without Serialization

In some cases, you don't need your message data to be serialized. If you pass in a plain string or Unicode object as your message and a custom *content_type*, then *Kombu* will not waste cycles serializing/deserializing the data.

You can optionally specify a *content_encoding* for the raw data:

```
>>> with open('~/my_picture.jpg', 'rb') as fh:
...     producer.publish(fh.read(),
                         content_type='image/jpeg',
                         content_encoding='binary',
                         routing_key=rkey)
```

The *Message* object returned by the *Consumer* class will have a *content_type* and *content_encoding* attribute.

2.8.3 Creating extensions using Setuptools entry-points

A package can also register new serializers using Setuptools entry-points.

The entry-point must provide the name of the serializer along with the path to a tuple providing the rest of the args: `encoder_function, decoder_function, content_type, content_encoding`.

An example entrypoint could be:

```
from setuptools import setup

setup(
    entry_points={
        'kombu.serializers': [
            'my_serializer = my_module.serializer:register_args'
        ]
    }
)
```

Then the module `my_module.serializer` would look like:

```
register_args = (my_encoder, my_decoder, 'application/x-mimetype', 'utf-8')
```

When this package is installed the new 'my_serializer' serializer will be supported by Kombu.

Buffer Objects

The decoder function of custom serializer must support both strings and Python's old-style buffer objects.

Python pickle and json modules usually don't do this via its `loads` function, but you can easily add support by making a wrapper around the `load` function that takes file objects instead of strings.

Here's an example wrapping `pickle.loads()` in such a way:

```
import pickle
from io import BytesIO
from kombu import serialization


def loads(s):
    return pickle.load(BytesIO(s))

serialization.register(
    'my_pickle', pickle.dumps, loads,
    content_type='application/x-pickle2',
    content_encoding='binary',
)
```

CHAPTER

THREE

FREQUENTLY ASKED QUESTIONS

3.1 Questions

3.1.1 Q: Message.reject doesn't work?

Answer: Earlier versions of RabbitMQ did not implement `basic.reject`, so make sure your version is recent enough to support it.

3.1.2 Q: Message.requeue doesn't work?

Answer: See Message.reject doesn't work?

CHAPTER

FOUR

API REFERENCE

Release 4.6

Date Dec 07, 2019

4.1 Kombu - `kombu`

Messaging library for Python.

`kombu.`**`enable_insecure_serializers`**(*choices=<object object>*)
Enable serializers that are considered to be unsafe.

Note: Will enable `pickle`, `yaml` and `msgpack` by default, but you can also specify a list of serializers (by name or content type) to enable.

`kombu.`**`disable_insecure_serializers`**(*allowed=<object object>*)
Disable untrusted serializers.

Will disable all serializers except `json` or you can specify a list of deserializers to allow.

Note: Producers will still be able to serialize data in these formats, but consumers will not accept incoming data using the untrusted content types.

4.1.1 Connection

class kombu.**Connection**(*hostname='localhost'*, *userid=None*, *password=None*, *virtual_host=None*, *port=None*, *insist=False*, *ssl=False*, *transport=None*, *connect_timeout=5*, *transport_options=None*, *login_method=None*, *uri_prefix=None*, *heartbeat=0*, *failover_strategy='round-robin'*, *alternates=None*, ***kwargs*)

A connection to the broker.

Example

```
>>> Connection('amqp://guest:guest@localhost:5672//')
>>> Connection('amqp://foo;amqp://bar',
...            failover_strategy='round-robin')
>>> Connection('redis://', transport_options={
...     'visibility_timeout': 3000,
... })
```

```
>>> import ssl
>>> Connection('amqp://', login_method='EXTERNAL', ssl={
...     'ca_certs': '/etc/pki/tls/certs/something.crt',
...     'keyfile': '/etc/something/system.key',
...     'certfile': '/etc/something/system.cert',
...     'cert_reqs': ssl.CERT_REQUIRED,
... })
```

Note: SSL currently only works with the py-amqp, and qpid transports. For other transports you can use stunnel.

Parameters **URL** (*str*, *Sequence*) – Broker URL, or a list of URLs.

Keyword Arguments

- **ssl** (*bool*) – Use SSL to connect to the server. Default is `False`. May not be supported by the specified transport.
- **transport** (`Transport`) – Default transport if not specified in the URL.
- **connect_timeout** (*float*) – Timeout in seconds for connecting to the server. May not be supported by the specified transport.
- **transport_options** (*Dict*) – A dict of additional connection arguments to pass to alternate kombu channel implementations. Consult the transport documentation for available options.
- **heartbeat** (*float*) – Heartbeat interval in int/float seconds. Note that if heartbeats are enabled then the *heartbeat_check()* method must be called regularly, around once per second.

Note: The connection is established lazily when needed. If you need the connection to be established, then force it by calling *connect()*:

```
>>> conn = Connection('amqp://')
>>> conn.connect()
```

and always remember to close the connection:

```
>>> conn.release()
```

These options have been replaced by the URL argument, but are still supported for backwards compatibility:

Keyword Arguments

- **hostname** – Host name/address. NOTE: You cannot specify both the URL argument and use the hostname keyword argument at the same time.
- **userid** – Default user name if not provided in the URL.
- **password** – Default password if not provided in the URL.
- **virtual_host** – Default virtual host if not provided in the URL.
- **port** – Default port if not provided in the URL.

Attributes

hostname = None

port = None

userid = None

password = None

virtual_host = '/'

ssl = None

login_method = None

failover_strategy = 'round-robin'
Strategy used to select new hosts when reconnecting after connection failure. One of "round-robin", "shuffle" or any custom iterator constantly yielding new URLs to try.

connect_timeout = 5

heartbeat = None
Heartbeat value, currently only supported by the py-amqp transport.

default_channel
Default channel.

Created upon access and closed when the connection is closed.

Note: Can be used for automatic channel handling when you only need one channel, and also it is the channel implicitly used if a connection is passed instead of a channel, to functions that require a channel.

---

connected
Return true if the connection has been established.

recoverable_connection_errors
Recoverable connection errors.

List of connection related exceptions that can be recovered from, but where the connection must be closed and re-established first.

recoverable_channel_errors
Recoverable channel errors.

List of channel related exceptions that can be automatically recovered from without re-establishing the connection.

connection_errors
List of exceptions that may be raised by the connection.

channel_errors
List of exceptions that may be raised by the channel.

transport

connection
The underlying connection object.

> **Warning:** This instance is transport specific, so do not depend on the interface of this object.

uri_prefix = None

declared_entities = None
The cache of declared entities is per connection, in case the server loses data.

cycle = None
Iterator returning the next broker URL to try in the event of connection failure (initialized by *failover_strategy*).

host
The host as a host name/port pair separated by colon.

manager
AMQP Management API.

Experimental manager that can be used to manage/monitor the broker instance.

Not available for all transports.

supports_heartbeats

is_evented

Methods

as_uri(*include_password=False*, *mask='**'*, *getfields=operator.itemgetter('port'*, *'userid'*, *'password'*, *'virtual_host'*, *'transport')*)
Convert connection parameters to URL form.

connect()
Establish connection to server immediately.

channel()
Create and return a new channel.

drain_events(***kwargs*)
Wait for a single event from the server.

Parameters **timeout** (*float*) – Timeout in seconds before we give up.

Raises **socket.timeout** – if the timeout is exceeded.

release()
Close the connection (if open).

autoretry(*fun*, *channel=None*, ***ensure_options*)
Decorator for functions supporting a `channel` keyword argument.

The resulting callable will retry calling the function if it raises connection or channel related errors. The return value will be a tuple of `(retval, last_created_channel)`.

If a `channel` is not provided, then one will be automatically acquired (remember to close it afterwards).

See also:

ensure() for the full list of supported keyword arguments.

Example

```
>>> channel = connection.channel()
>>> try:
...     ret, channel = connection.autoretry(
...         publish_messages, channel)
... finally:
...     channel.close()
```

ensure_connection(*errback=None*, *max_retries=None*, *interval_start=2*, *interval_step=2*, *interval_max=30*, *callback=None*, *reraise_as_library_errors=True*, *timeout=None*)
Ensure we have a connection to the server.

If not retry establishing the connection with the settings specified.

Parameters

- **errback** (Callable) – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept `(exc, interval)`.
- **max_retries** (*int*) – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval_start** (*float*) – The number of seconds we start sleeping for.
- **interval_step** (*float*) – How many seconds added to the interval for each retry.
- **interval_max** (*float*) – Maximum number of seconds to sleep between each retry.
- **callback** (Callable) – Optional callback that is called for every internal iteration (1 s).
- **timeout** (*int*) – Maximum amount of time in seconds to spend waiting for connection

ensure(*obj*, *fun*, *errback=None*, *max_retries=None*, *interval_start=1*, *interval_step=1*, *interval_max=1*, *on_revive=None*)
Ensure operation completes.

Regardless of any channel/connection errors occurring.

Retries by establishing the connection, and reapplying the function.

Parameters

- **obj** – The object to ensure an action on.
- **fun** (Callable) – Method to apply.

- **errback** (Callable) – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept (exc, interval).
- **max_retries** (*int*) – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval_start** (*float*) – The number of seconds we start sleeping for.
- **interval_step** (*float*) – How many seconds added to the interval for each retry.
- **interval_max** (*float*) – Maximum number of seconds to sleep between each retry.
- **on_revive** (Callable) – Optional callback called whenever revival completes successfully

Examples

```
>>> from kombu import Connection, Producer
>>> conn = Connection('amqp://')
>>> producer = Producer(conn)
```

```
>>> def errback(exc, interval):
...     logger.error('Error: %r', exc, exc_info=1)
...     logger.info('Retry in %s seconds.', interval)
```

```
>>> publish = conn.ensure(producer, producer.publish,
...                       errback=errback, max_retries=3)
>>> publish({'hello': 'world'}, routing_key='dest')
```

revive(*new_channel*)
Revive connection after connection re-established.

create_transport()

get_transport_cls()
Get the currently used transport class.

clone(***kwargs*)
Create a copy of the connection with same settings.

info()
Get connection info.

switch(*conn_str*)
Switch connection parameters to use a new URL or hostname.

Note: Does not reconnect!

Parameters **conn_str** (*str*) – either a hostname or URL.

maybe_switch_next()
Switch to next URL given by the current failover strategy.

heartbeat_check(*rate=2*)
Check heartbeats.

Allow the transport to perform any periodic tasks required to make heartbeats work. This should be called approximately every second.

If the current transport does not support heartbeats then this is a noop operation.

Parameters **rate** (*int*) – Rate is how often the tick is called compared to the actual heartbeat value. E.g. if the heartbeat is set to 3 seconds, and the tick is called every 3 / 2 seconds, then the rate is 2. This value is currently unused by any transports.

maybe_close_channel(*channel*)
Close given channel, but ignore connection and channel errors.

register_with_event_loop(*loop*)

close()
Close the connection (if open).

_close()
Really close connection, even if part of a connection pool.

completes_cycle(*retries*)
Return true if the cycle is complete after number of *retries*.

get_manager(**args*, ***kwargs*)

Producer(*channel=None*, **args*, ***kwargs*)
Create new *kombu.Producer* instance.

Consumer(*queues=None*, *channel=None*, **args*, ***kwargs*)
Create new *kombu.Consumer* instance.

Pool(*limit=None*, ***kwargs*)
Pool of connections.

See also:

ConnectionPool.

Parameters **limit** (*int*) – Maximum number of active connections. Default is no limit.

Example

```
>>> connection = Connection('amqp://')
>>> pool = connection.Pool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
  raise ConnectionLimitExceeded(self.limit)
    kombu.exceptions.ConnectionLimitExceeded: 2
```

ChannelPool(*limit=None*, ***kwargs*)
Pool of channels.

See also:

ChannelPool.

Parameters **limit** (*int*) – Maximum number of active channels. Default is no limit.

Example

```
>>> connection = Connection('amqp://')
>>> pool = connection.ChannelPool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
  raise ChannelLimitExceeded(self.limit)
    kombu.connection.ChannelLimitExceeded: 2
```

SimpleQueue(*name*, *no_ack=None*, *queue_opts=None*, *queue_args=None*, *exchange_opts=None*, *channel=None*, ***kwargs*)

Simple persistent queue API.

Create new *SimpleQueue*, using a channel from this connection.

If `name` is a string, a queue and exchange will be automatically created using that name as the name of the queue and exchange, also it will be used as the default routing key.

Parameters

- **name** (*str, kombu.Queue*) – Name of the queue/or a queue.
- **no_ack** (*bool*) – Disable acknowledgments. Default is false.
- **queue_opts** (*Dict*) – Additional keyword arguments passed to the constructor of the automatically created *Queue*.
- **queue_args** (*Dict*) – Additional keyword arguments passed to the constructor of the automatically created *Queue* for setting implementation extensions (e.g., in RabbitMQ).
- **exchange_opts** (*Dict*) – Additional keyword arguments passed to the constructor of the automatically created *Exchange*.
- **channel** (*ChannelT*) – Custom channel to use. If not specified the connection default channel is used.

SimpleBuffer(*name*, *no_ack=None*, *queue_opts=None*, *queue_args=None*, *exchange_opts=None*, *channel=None*, ***kwargs*)

Simple ephemeral queue API.

Create new *SimpleQueue* using a channel from this connection.

See also:

Same as *SimpleQueue()*, but configured with buffering semantics. The resulting queue and exchange will not be durable, also auto delete is enabled. Messages will be transient (not persistent), and acknowledgments are disabled (`no_ack`).

4.1.2 Exchange

Example creating an exchange declaration:

```
>>> news_exchange = Exchange('news', type='topic')
```

For now *news_exchange* is just a declaration, you can't perform actions on it. It just describes the name and options for the exchange.

The exchange can be bound or unbound. Bound means the exchange is associated with a channel and operations can be performed on it. To bind the exchange you call the exchange with the channel as argument:

```
>>> bound_exchange = news_exchange(channel)
```

Now you can perform operations like `declare()` or `delete()`:

```
>>> # Declare exchange manually
>>> bound_exchange.declare()

>>> # Publish raw string message using low-level exchange API
>>> bound_exchange.publish(
...     'Cure for cancer found!',
...     routing_key='news.science',
... )

>>> # Delete exchange.
>>> bound_exchange.delete()
```

class kombu.**Exchange**(*name=''*, *type=''*, *channel=None*, ***kwargs*)

An Exchange declaration.

Parameters

- **name** (*str*) – See *name*.
- **type** (*str*) – See *type*.
- **channel** (kombu.Connection, *ChannelT*) – See *channel*.
- **durable** (*bool*) – See *durable*.
- **auto_delete** (*bool*) – See *auto_delete*.
- **delivery_mode** (*enum*) – See *delivery_mode*.
- **arguments** (*Dict*) – See *arguments*.
- **no_declare** (*bool*) – See *no_declare*

name

Name of the exchange. Default is no name (the default exchange).

Type str

type

This description of AMQP exchange types was shamelessly stolen from the blog post 'AMQP in 10 minutes: Part 4'_ by Rajith Attapattu. Reading this article is recommended if you're new to amqp.

"AMQP defines four default exchange types (routing algorithms) that covers most of the common messaging use cases. An AMQP broker can also define additional exchange types, so see your broker manual for more information about available exchange types.

- *direct* (*default*)

  Direct match between the routing key in the message, and the routing criteria used when a queue is bound to this exchange.

- *topic*

  Wildcard match between the routing key and the routing pattern specified in the exchange/queue binding. The routing key is treated as zero or more words delimited by *"."* and supports special wildcard characters. *"*"* matches a single word and *"#"* matches zero or more words.

- *fanout*

  Queues are bound to this exchange with no arguments. Hence any message sent to this exchange will be forwarded to all queues bound to this exchange.

- *headers*

  Queues are bound to this exchange with a table of arguments containing headers and values (optional). A special argument named "x-match" determines the matching algorithm, where *"all"* implies an *AND* (all pairs must match) and *"any"* implies *OR* (at least one pair must match).

  `arguments` is used to specify the arguments.

Type str

channel
The channel the exchange is bound to (if bound).

Type ChannelT

durable
Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged when a server restarts. Default is `True`.

Type bool

auto_delete
If set, the exchange is deleted when all queues have finished using it. Default is `False`.

Type bool

delivery_mode
The default delivery mode used for messages. The value is an integer, or alias string.

- 1 or *"transient"*

  The message is transient. Which means it is stored in memory only, and is lost if the server dies or restarts.

- **2 or "persistent" (*default*)** The message is persistent. Which means the message is stored both in-memory, and on disk, and therefore preserved if the server dies or restarts.

The default value is 2 (persistent).

Type enum

arguments
Additional arguments to specify when the exchange is declared.

Type Dict

no_declare
Never declare this exchange (`declare()` does nothing).

Type bool

maybe_bind(*channel*)
Bind instance to channel if not already bound.

Message(*body*, *delivery_mode=None*, *properties=None*, ***kwargs*)
Create message instance to be sent with *publish()*.

Parameters

- **body** (*Any*) – Message body.
- **delivery_mode** (*bool*) – Set custom delivery mode. Defaults to *delivery_mode*.
- **priority** (*int*) – Message priority, 0 to broker configured max priority, where higher is better.
- **content_type** (*str*) – The messages content_type. If content_type is set, no serialization occurs as it is assumed this is either a binary object, or you've done your own serialization. Leave blank if using built-in serialization as our library properly sets content_type.
- **content_encoding** (*str*) – The character set in which this object is encoded. Use "binary" if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets content_encoding.
- **properties** (*Dict*) – Message properties.
- **headers** (*Dict*) – Message headers.

PERSISTENT_DELIVERY_MODE = 2

TRANSIENT_DELIVERY_MODE = 1

attrs = (('name', None), ('type', None), ('arguments', None), ('durable', <class 'bool

auto_delete = False

bind_to(*exchange=''*, *routing_key=''*, *arguments=None*, *nowait=False*, *channel=None*, ***kwargs*)
Bind the exchange to another exchange.

Parameters **nowait** (*bool*) – If set the server will not respond, and the call will not block waiting for a response. Default is `False`.

binding(*routing_key=''*, *arguments=None*, *unbind_arguments=None*)

property can_cache_declaration
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

declare(*nowait=False*, *passive=None*, *channel=None*)
Declare the exchange.

Creates the exchange on the broker, unless passive is set in which case it will only assert that the exchange exists.

Argument:

nowait (bool): If set the server will not respond, and a response will not be waited for. Default is `False`.

delete(*if_unused=False*, *nowait=False*)
Delete the exchange declaration on server.

Parameters

- **if_unused** (*bool*) – Delete only if the exchange has no bindings. Default is `False`.
- **nowait** (*bool*) – If set the server will not respond, and a response will not be waited for. Default is `False`.

delivery_mode = None

durable = True

name = ''

no_declare = False

passive = False

publish(*message*, *routing_key=None*, *mandatory=False*, *immediate=False*, *exchange=None*)
Publish message.

Parameters

- **message** (*Union[kombu.Message, str, bytes]*) – Message to publish.
- **routing_key** (*str*) – Message routing key.
- **mandatory** (*bool*) – Currently not supported.
- **immediate** (*bool*) – Currently not supported.

type = 'direct'

unbind_from(*source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *channel=None*)
Delete previously created exchange binding from the server.

4.1.3 Queue

Example creating a queue using our exchange in the *Exchange* example:

```
>>> science_news = Queue('science_news',
...                      exchange=news_exchange,
...                      routing_key='news.science')
```

For now *science_news* is just a declaration, you can't perform actions on it. It just describes the name and options for the queue.

The queue can be bound or unbound. Bound means the queue is associated with a channel and operations can be performed on it. To bind the queue you call the queue instance with the channel as an argument:

```
>>> bound_science_news = science_news(channel)
```

Now you can perform operations like `declare()` or `purge()`:

```
>>> bound_science_news.declare()
>>> bound_science_news.purge()
>>> bound_science_news.delete()
```

class kombu.**Queue**(*name=''*, *exchange=None*, *routing_key=''*, *channel=None*, *bindings=None*, *on_declared=None*, ***kwargs*)
A Queue declaration.

Parameters

- **name** (*str*) – See *name*.

- **exchange** (*Exchange, str*) – See *exchange*.
- **routing_key** (*str*) – See *routing_key*.
- **channel** (*kombu.Connection, ChannelT*) – See *channel*.
- **durable** (*bool*) – See *durable*.
- **exclusive** (*bool*) – See *exclusive*.
- **auto_delete** (*bool*) – See *auto_delete*.
- **queue_arguments** (*Dict*) – See *queue_arguments*.
- **binding_arguments** (*Dict*) – See *binding_arguments*.
- **consumer_arguments** (*Dict*) – See *consumer_arguments*.
- **no_declare** (*bool*) – See *no_declare*.
- **on_declared** (*Callable*) – See *on_declared*.
- **expires** (*float*) – See *expires*.
- **message_ttl** (*float*) – See *message_ttl*.
- **max_length** (*int*) – See *max_length*.
- **max_length_bytes** (*int*) – See *max_length_bytes*.
- **max_priority** (*int*) – See *max_priority*.

name
Name of the queue. Default is no name (default queue destination).

Type str

exchange
The *Exchange* the queue binds to.

Type *Exchange*

routing_key
The routing key (if any), also called *binding key*.

The interpretation of the routing key depends on the *Exchange.type*.

- direct exchange

  Matches if the routing key property of the message and the *routing_key* attribute are identical.

- fanout exchange

  Always matches, even if the binding does not have a key.

- topic exchange

  Matches the routing key property of the message by a primitive pattern matching scheme. The message routing key then consists of words separated by dots (*"."*, like domain names), and two special characters are available; star (*"*"*) and hash (*"#"*). The star matches any word, and the hash matches zero or more words. For example *"*.stock.#"* matches the routing keys *"usd.stock"* and *"eur.stock.db"* but not *"stock.nasdaq"*.

Type str

channel
The channel the Queue is bound to (if bound).

Type ChannelT

durable
Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.

Default is `True`.

Type bool

exclusive
Exclusive queues may only be consumed from by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'.

Default is `False`.

Type bool

auto_delete
If set, the queue is deleted when all consumers have finished using it. Last consumer can be canceled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted.

Type bool

expires
Set the expiry time (in seconds) for when this queue should expire.

The expiry time decides how long the queue can stay unused before it's automatically deleted. *Unused* means the queue has no consumers, the queue has not been redeclared, and `Queue.get` has not been invoked for a duration of at least the expiration period.

See https://www.rabbitmq.com/ttl.html#queue-ttl

RabbitMQ extension: Only available when using RabbitMQ.

Type float

message_ttl
Message time to live in seconds.

This setting controls how long messages can stay in the queue unconsumed. If the expiry time passes before a message consumer has received the message, the message is deleted and no consumer will see the message.

See https://www.rabbitmq.com/ttl.html#per-queue-message-ttl

RabbitMQ extension: Only available when using RabbitMQ.

Type float

max_length
Set the maximum number of messages that the queue can hold.

If the number of messages in the queue size exceeds this limit, new messages will be dropped (or dead-lettered if a dead letter exchange is active).

See https://www.rabbitmq.com/maxlength.html

RabbitMQ extension: Only available when using RabbitMQ.

Type int

max_length_bytes
Set the max size (in bytes) for the total of messages in the queue.

If the total size of all the messages in the queue exceeds this limit, new messages will be dropped (or dead-lettered if a dead letter exchange is active).

RabbitMQ extension: Only available when using RabbitMQ.

Type int

max_priority
Set the highest priority number for this queue.

For example if the value is 10, then messages can delivered to this queue can have a `priority` value between 0 and 10, where 10 is the highest priority.

RabbitMQ queues without a max priority set will ignore the priority field in the message, so if you want priorities you need to set the max priority field to declare the queue as a priority queue.

RabbitMQ extension: Only available when using RabbitMQ.

Type int

queue_arguments
Additional arguments used when declaring the queue. Can be used to to set the arguments value for RabbitMQ/AMQP's `queue.declare`.

Type Dict

binding_arguments
Additional arguments used when binding the queue. Can be used to to set the arguments value for RabbitMQ/AMQP's `queue.declare`.

Type Dict

consumer_arguments
Additional arguments used when consuming from this queue. Can be used to to set the arguments value for RabbitMQ/AMQP's `basic.consume`.

Type Dict

alias
Unused in Kombu, but applications can take advantage of this, for example to give alternate names to queues with automatically generated queue names.

Type str

on_declared
Optional callback to be applied when the queue has been declared (the `queue_declare` operation is complete). This must be a function with a signature that accepts at least 3 positional arguments: `(name, messages, consumers)`.

Type *Callable*

no_declare
Never declare this queue, nor related entities (*`declare()`* does nothing).

Type bool

maybe_bind(*channel*)
Bind instance to channel if not already bound.

exception ContentDisallowed
Consumer does not allow this content-type.

as_dict(*recurse=False*)

attrs = (('name', None), ('exchange', None), ('routing_key', None), ('queue_arguments'

auto_delete = False

bind(*channel*)
Create copy of the instance that is bound to a channel.

bind_to(*exchange=''*, *routing_key=''*, *arguments=None*, *nowait=False*, *channel=None*)

property can_cache_declaration
bool(x) -> bool

Returns True when the argument x is true, False otherwise. The builtins True and False are the only two instances of the class bool. The class bool is a subclass of the class int, and cannot be subclassed.

cancel(*consumer_tag*)
Cancel a consumer by consumer tag.

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

Consumers last as long as the channel they were created on, or until the client cancels them.

Parameters

- **consumer_tag** (*str*) – Unique identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.
- **no_ack** (*bool*) – If enabled the broker will automatically ack messages.
- **nowait** (*bool*) – Do not wait for a reply.
- **callback** (*Callable*) – callback called for each delivered message.

declare(*nowait=False*, *channel=None*)
Declare queue and exchange then binds queue to exchange.

delete(*if_unused=False*, *if_empty=False*, *nowait=False*)
Delete the queue.

Parameters

- **if_unused** (*bool*) – If set, the server will only delete the queue if it has no consumers. A channel error will be raised if the queue has consumers.
- **if_empty** (*bool*) – If set, the server will only delete the queue if it is empty. If it is not empty a channel error will be raised.
- **nowait** (*bool*) – Do not wait for a reply.

durable = True

exchange = <unbound Exchange ''(direct)>

exclusive = False

classmethod from_dict(*queue*, ***options*)

get(*no_ack=None*, *accept=None*)
Poll the server for a new message.

This method provides direct access to the messages in a queue using a synchronous dialogue, designed for specific types of applications where synchronous functionality is more important than performance.

Returns

if a message was available, or `None` otherwise.

Return type Message

Parameters

- **no_ack** (*bool*) – If enabled the broker will automatically ack messages.
- **accept** (*Set[str]*) – Custom list of accepted content types.

name = ''

no_ack = False

purge(*nowait=False*)
Remove all ready messages from the queue.

queue_bind(*nowait=False*, *channel=None*)
Create the queue binding on the server.

queue_declare(*nowait=False*, *passive=False*, *channel=None*)
Declare queue on the server.

Parameters

- **nowait** (*bool*) – Do not wait for a reply.
- **passive** (*bool*) – If set, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state.

queue_unbind(*arguments=None*, *nowait=False*, *channel=None*)

routing_key = ''

unbind_from(*exchange=''*, *routing_key=''*, *arguments=None*, *nowait=False*, *channel=None*)
Unbind queue by deleting the binding from the server.

when_bound()
Callback called when the class is bound.

4.1.4 Message Producer

class kombu.**Producer**(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)
Message Producer.

Parameters

- **channel** (*kombu.Connection, ChannelT*) – Connection or channel.
- **exchange** (*kombu.entity.Exchange, str*) – Optional default exchange.
- **routing_key** (*str*) – Optional default routing key.
- **serializer** (*str*) – Default serializer. Default is *"json"*.
- **compression** (*str*) – Default compression method. Default is no compression.
- **auto_declare** (*bool*) – Automatically declare the default exchange at instantiation. Default is `True`.

- **on_return** (Callable) – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to *publish()* is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

channel

exchange = None
Default exchange

routing_key = ''
Default routing key.

serializer = None
Default serializer to use. Default is JSON.

compression = None
Default compression method. Disabled by default.

auto_declare = True
By default, if a defualt exchange is set, that exchange will be declare when publishing a message.

on_return = None
Basic return callback.

connection

declare()
Declare the exchange.

Note: This happens automatically at instantiation when the *auto_declare* flag is enabled.

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare exchange if not already declared during this session.

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=None*, *expiration=None*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** (*Any*) – Message body.
- **routing_key** (*str*) – Message routing key.
- **delivery_mode** (*enum*) – See `delivery_mode`.
- **mandatory** (*bool*) – Currently not supported.
- **immediate** (*bool*) – Currently not supported.
- **priority** (*int*) – Message priority. A number between 0 and 9.
- **content_type** (*str*) – Content type. Default is auto-detect.
- **content_encoding** (*str*) – Content encoding. Default is auto-detect.
- **serializer** (*str*) – Serializer to use. Default is auto-detect.
- **compression** (*str*) – Compression method to use. Default is none.
- **headers** (*Dict*) – Mapping of arbitrary headers to pass along with the message body.

- **exchange** (*kombu.entity.Exchange, str*) – Override the exchange. Note that this exchange must have been declared.
- **declare** (*Sequence[EntityT]*) – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using *maybe_declare()*.
- **retry** (*bool*) – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** (*Dict*) – Retry configuration, this is the keywords supported by *ensure()*.
- **expiration** (*float*) – A TTL in seconds can be specified per message. Default is no expiration.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

revive(*channel*)
Revive the producer after connection loss.

4.1.5 Message Consumer

class kombu.**Consumer**(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*, *on_message=None*, *accept=None*, *prefetch_count=None*, *tag_prefix=None*)
Message consumer.

Parameters

- **channel** (kombu.Connection, *ChannelT*) – see *channel*.
- **queues** (*Sequence[kombu.Queue]*) – see *queues*.
- **no_ack** (*bool*) – see *no_ack*.
- **auto_declare** (*bool*) – see *auto_declare*
- **callbacks** (*Sequence[Callable]*) – see *callbacks*.
- **on_message** (Callable) – See *on_message*
- **on_decode_error** (Callable) – see *on_decode_error*.
- **prefetch_count** (*int*) – see prefetch_count.

channel = None
The connection/channel to use for this consumer.

queues
A single *Queue*, or a list of queues to consume from.

no_ack = None
Flag for automatic message acknowledgment. If enabled the messages are automatically acknowledged by the broker. This can increase performance but means that you have no control of when the message is removed.

Disabled by default.

auto_declare = True
By default all entities will be declared at instantiation, if you want to handle this manually you can set this to False.

callbacks = None
List of callbacks called in order when a message is received.

The signature of the callbacks must take two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

on_message = None
Optional function called whenever a message is received.

When defined this function will be called instead of the *receive()* method, and *callbacks* will be disabled.

So this can be used as an alternative to *callbacks* when you don't want the body to be automatically decoded. Note that the message will still be decompressed if the message has the `compression` header set.

The signature of the callback must take a single argument, which is the `Message` object.

Also note that the `message.body` attribute, which is the raw contents of the message body, may in some cases be a read-only `buffer` object.

on_decode_error = None
Callback called when a message can't be decoded.

The signature of the callback must take two arguments: *(message, exc)*, which is the message that can't be decoded and the exception that occurred while trying to decode it.

connection

declare()
Declare queues, exchanges and bindings.

Note: This is done automatically at instantiation when *auto_declare* is set.

---

register_callback(*callback*)
Register a new callback to be called when a message is received.

Note: The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

---

add_queue(*queue*)
Add a queue to the list of queues to consume from.

Note: This will not start consuming from the queue, for that you will have to call *consume()* after.

---

consume(*no_ack=None*)
Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use *cancel_by_queue()*).

Parameters **no_ack** (*bool*) – See *no_ack*.

cancel()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

cancel_by_queue(*queue*)
Cancel consumer by queue name.

consuming_from(*queue*)
Return `True` if currently consuming from queue'.

purge()
Purge messages from all queues.

Warning: This will *delete all ready messages*, there is no undo operation.

flow(*active*)
Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the *no_ack* option is set.

Parameters

- **prefetch_size** (*int*) – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
- **prefetch_count** (*int*) – Specify the prefetch window in terms of whole messages.
- **apply_global** (*bool*) – Apply new settings globally on all channels.

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

Asks the broker to redeliver all unacknowledged messages on the specified channel.

Parameters **requeue** (*bool*) – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

receive(*body*, *message*)
Method called when a message is received.

This dispatches to the registered *callbacks*.

Parameters

- **body** (*Any*) – The decoded message body.

- **message** (*Message*) – The message instance.

Raises **NotImplementedError** – If no consumer callbacks have been registered.

revive(*channel*)
Revive consumer after connection loss.

4.2 Common Utilities - kombu.common

Common Utilities.

class kombu.common.**Broadcast**(*name=None*, *queue=None*, *unique=False*, *auto_delete=True*, *exchange=None*, *alias=None*, ***kwargs*)
Broadcast queue.

Convenience class used to define broadcast queues.

Every queue instance will have a unique name, and both the queue and exchange is configured with auto deletion.

Parameters

- **name** (*str*) – This is used as the name of the exchange.
- **queue** (*str*) – By default a unique id is used for the queue name for every consumer. You can specify a custom queue name here.
- **unique** (*bool*) – Always create a unique queue even if a queue name is supplied.
- ****kwargs** (*Any*) – See *Queue* for a list of additional keyword arguments supported.

attrs = (('name', None), ('exchange', None), ('routing_key', None), ('queue_arguments'

kombu.common.**maybe_declare**(*entity*, *channel=None*, *retry=False*, ***retry_policy*)
Declare entity (cached).

kombu.common.**uuid**(*_uuid=<function uuid4>*)
Generate unique id in UUID4 format.

See also:

For now this is provided by uuid.uuid4().

kombu.common.**itermessages**(*conn*, *channel*, *queue*, *limit=1*, *timeout=None*, *callbacks=None*, ***kwargs*)
Iterator over messages.

kombu.common.**send_reply**(*exchange*, *req*, *msg*, *producer=None*, *retry=False*, *retry_policy=None*, ***props*)
Send reply for request.

Parameters

- **exchange** (kombu.Exchange, *str*) – Reply exchange
- **req** (*Message*) – Original request, a message with a reply_to property.
- **producer** (kombu.Producer) – Producer instance
- **retry** (*bool*) – If true must retry according to the reply_policy argument.
- **retry_policy** (*Dict*) – Retry settings.
- ****props** (*Any*) – Extra properties.

kombu.common.**collect_replies**(*conn*, *channel*, *queue*, **args*, ***kwargs*)
Generator collecting replies from `queue`.

kombu.common.**insured**(*pool*, *fun*, *args*, *kwargs*, *errback=None*, *on_revive=None*, ***opts*)
Function wrapper to handle connection errors.

Ensures function performing broker commands completes despite intermittent connection failures.

kombu.common.**drain_consumer**(*consumer*, *limit=1*, *timeout=None*, *callbacks=None*)
Drain messages from consumer instance.

kombu.common.**eventloop**(*conn*, *limit=None*, *timeout=None*, *ignore_timeouts=False*)
Best practice generator wrapper around `Connection.drain_events`.

Able to drain events forever, with a limit, and optionally ignoring timeout errors (a timeout of 1 is often used in environments where the socket can get "stuck", and is a best practice for Kombu consumers).

`eventloop` is a generator.

Examples

```
>>> from kombu.common import eventloop
```

```
>>> def run(conn):
...     it = eventloop(conn, timeout=1, ignore_timeouts=True)
...     next(it)   # one event consumed, or timed out.
...
...     for _ in eventloop(conn, timeout=1, ignore_timeouts=True):
...         pass  # loop forever.
```

It also takes an optional limit parameter, and timeout errors are propagated by default:

```
for _ in eventloop(connection, limit=1, timeout=1):
    pass
```

See also:

itermessages(), which is an event loop bound to one or more consumers, that yields any messages received.

4.3 Pattern matching registry - kombu.matcher

Pattern matching registry.

exception kombu.matcher.**MatcherNotInstalled**
Matcher not installed/found.

class kombu.matcher.**MatcherRegistry**
Pattern matching function registry.

exception MatcherNotInstalled
Matcher not installed/found.

match(*data*, *pattern*, *matcher=None*, *matcher_kwargs=None*)
Call the matcher.

matcher_pattern_first = ['pcre']

register(*name*, *matcher*)
Add matcher by name to the registry.

unregister(*name*)
Remove matcher by name from the registry.

kombu.matcher.**match**(*data*, *pattern*, *matcher=None*, *matcher_kwargs=None*)

register(name, matcher):
Register a new matching method.

Parameters

- **name** – A convience name for the mathing method.
- **matcher** – A method that will be passed data and pattern.

kombu.matcher.**register**(*name*, *matcher*)

unregister(name):
Unregister registered matching method.

Parameters **name** – Registered matching method name.

kombu.matcher.**register_glob**()
Register glob into default registry.

kombu.matcher.**register_pcre**()
Register pcre into default registry.

kombu.matcher.**registry = <kombu.matcher.MatcherRegistry object>**

match(data, pattern, matcher=default_matcher,
matcher_kwargs=None):

Match *data* by *pattern* using *matcher*.

Parameters

- **data** – The data that should be matched. Must be string.
- **pattern** – The pattern that should be applied. Must be string.

Keyword Arguments

- **matcher** – An optional string representing the mathcing method (for example, *glob* or *pcre*).

  If `None` (default), then *glob* will be used.
- **matcher_kwargs** – Additional keyword arguments that will be passed to the specified *matcher*.

Returns `True` if *data* matches pattern, `False` otherwise.

Raises ***MatcherNotInstalled*** – If the matching method requested is not available.

kombu.matcher.**unregister**(*name*)
Remove matcher by name from the registry.

4.4 Mixin Classes - kombu.mixins

Mixins.

class kombu.mixins.**ConsumerMixin**

Convenience mixin for implementing consumer programs.

It can be used outside of threads, with threads, or greenthreads (eventlet/gevent) too.

The basic class would need a `connection` attribute which must be a *`Connection`* instance, and define a *`get_consumers()`* method that returns a list of *`kombu.Consumer`* instances to use. Supporting multiple consumers is important so that multiple channels can be used for different QoS requirements.

Example

```
class Worker(ConsumerMixin):
    task_queue = Queue('tasks', Exchange('tasks'), 'tasks')

    def __init__(self, connection):
        self.connection = None

    def get_consumers(self, Consumer, channel):
        return [Consumer(queues=[self.task_queue],
                         callbacks=[self.on_task])]

    def on_task(self, body, message):
        print('Got task: {0!r}'.format(body))
        message.ack()
```

* **:meth:`extra_context`**

  Optional extra context manager that will be entered after the connection and consumers have been set up.

  Takes arguments `(connection, channel)`.

* **:meth:`on_connection_error`**

  Handler called if the connection is lost/ or is unavailable.

  Takes arguments `(exc, interval)`, where interval is the time in seconds when the connection will be retried.

  The default handler will log the exception.

* **:meth:`on_connection_revived`**

  Handler called as soon as the connection is re-established after connection failure.

  Takes no arguments.

* **:meth:`on_consume_ready`**

  Handler called when the consumer is ready to accept messages.

  Takes arguments `(connection, channel, consumers)`. Also keyword arguments to `consume` are forwarded to this handler.

* **:meth:`on_consume_end`**

  Handler called after the consumers are canceled. Takes arguments `(connection, channel)`.

* **:meth:`on_iteration`**

  Handler called for every iteration while draining events.

  Takes no arguments.

* :meth:`on_decode_error`
  Handler called if a consumer was unable to decode the body of a message.

  Takes arguments `(message, exc)` where message is the original message object.

  The default handler will log the error and acknowledge the message, so if you override make sure to call super, or perform these steps yourself.

Consumer()

channel_errors

connect_max_retries = None
maximum number of retries trying to re-establish the connection, if the connection is lost/unavailable.

connection_errors

consume(*limit=None*, *timeout=None*, *safety_interval=1*, ***kwargs*)

consumer_context(***kwargs*)

create_connection()

establish_connection()

extra_context(*connection*, *channel*)

get_consumers(*Consumer*, *channel*)

maybe_conn_error(*fun*)
Use `kombu.common.ignore_errors()` instead.

on_connection_error(*exc*, *interval*)

on_connection_revived()

on_consume_end(*connection*, *channel*)

on_consume_ready(*connection*, *channel*, *consumers*, ***kwargs*)

on_decode_error(*message*, *exc*)

on_iteration()

restart_limit

run(*_tokens=1*, ***kwargs*)

should_stop = False
When this is set to true the consumer should stop consuming and return, so that it can be joined if it is the implementation of a thread.

class kombu.mixins.**ConsumerProducerMixin**
Consumer and Producer mixin.

Version of ConsumerMixin having separate connection for also publishing messages.

Example

```
class Worker(ConsumerProducerMixin):

    def __init__(self, connection):
        self.connection = connection

    def get_consumers(self, Consumer, channel):
```

(continues on next page)

(continued from previous page)

```
        return [Consumer(queues=Queue('foo'),
                         on_message=self.handle_message,
                         accept='application/json',
                         prefetch_count=10)]

    def handle_message(self, message):
        self.producer.publish(
            {'message': 'hello to you'},
            exchange='',
            routing_key=message.properties['reply_to'],
            correlation_id=message.properties['correlation_id'],
            retry=True,
        )
```

on_consume_end(*connection*, *channel*)

property producer

property producer_connection

4.5 Simple Messaging API - kombu.simple

Simple messaging interface.

- *Persistent*
- *Buffer*

4.5.1 Persistent

class kombu.simple.**SimpleQueue**(*channel*, *name*, *no_ack=None*, *queue_opts=None*, *queue_args=None*, *exchange_opts=None*, *serializer=None*, *compression=None*, ***kwargs*)

Simple API for persistent queues.

channel
Current channel

producer
Producer used to publish messages.

consumer
Consumer used to receive messages.

no_ack
flag to enable/disable acknowledgments.

queue
Queue to consume from (if consuming).

queue_opts

Additional options for the queue declaration.

exchange_opts
Additional options for the exchange declaration.

get(*block=True*, *timeout=None*)

get_nowait()

put(*message*, *serializer=None*, *headers=None*, *compression=None*, *routing_key=None*, ***kwargs*)

clear()

__len__()
len(self) -> self.qsize().

qsize()

close()

4.5.2 Buffer

class kombu.simple.**SimpleBuffer**(*channel*, *name*, *no_ack=None*, *queue_opts=None*, *queue_args=None*, *exchange_opts=None*, *serializer=None*, *compression=None*, ***kwargs*)
Simple API for ephemeral queues.

channel
Current channel

producer
Producer used to publish messages.

consumer
Consumer used to receive messages.

no_ack
flag to enable/disable acknowledgments.

queue
Queue to consume from (if consuming).

queue_opts

Additional options for the queue declaration.

exchange_opts
Additional options for the exchange declaration.

get(*block=True*, *timeout=None*)

get_nowait()

put(*message*, *serializer=None*, *headers=None*, *compression=None*, *routing_key=None*, ***kwargs*)

clear()

__len__()
len(self) -> self.qsize().

qsize()

close()

4.6 Logical Clocks and Synchronization - kombu.clocks

Logical Clocks and Synchronization.

class kombu.clocks.**LamportClock**(*initial_value=0*, *Lock=<built-in function allocate_lock>*)

Lamport's logical clock.

From Wikipedia:

A Lamport logical clock is a monotonically incrementing software counter maintained in each process. It follows some simple rules:

- A process increments its counter before each event in that process;
- When a process sends a message, it includes its counter value with the message;
- On receiving a message, the receiver process sets its counter to be greater than the maximum of its own value and the received value before it considers the message received.

Conceptually, this logical clock can be thought of as a clock that only has meaning in relation to messages moving between processes. When a process receives a message, it resynchronizes its logical clock with the sender.

See also:

- Lamport timestamps
- Lamports distributed mutex

Usage

When sending a message use *forward()* to increment the clock, when receiving a message use *adjust()* to sync with the time stamp of the incoming message.

adjust(*other*)

forward()

sort_heap(*h*)

Sort heap of events.

List of tuples containing at least two elements, representing an event, where the first element is the event's scalar clock value, and the second element is the id of the process (usually `"hostname:pid"`): `sh([(clock, processid, ...?), (...)])`

The list must already be sorted, which is why we refer to it as a heap.

The tuple will not be unpacked, so more than two elements can be present.

Will return the latest event.

value = 0

The clocks current value.

class kombu.clocks.**timetuple**

Tuple of event clock information.

Can be used as part of a heap to keep events ordered.

Parameters

- **clock** (*int*) – Event clock value.
- **timestamp** (*float*) – Event UNIX timestamp value.

- **id** (*str*) – Event host id (e.g. `hostname:pid`).
- **obj** (*Any*) – Optional obj to associate with this event.

property clock
itemgetter(item, ...) –> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After f = itemgetter(2), the call f(r) returns r[2]. After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])

property id
itemgetter(item, ...) –> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After f = itemgetter(2), the call f(r) returns r[2]. After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])

property obj
itemgetter(item, ...) –> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After f = itemgetter(2), the call f(r) returns r[2]. After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])

property timestamp
itemgetter(item, ...) –> itemgetter object

Return a callable object that fetches the given item(s) from its operand. After f = itemgetter(2), the call f(r) returns r[2]. After g = itemgetter(2, 5, 3), the call g(r) returns (r[2], r[5], r[3])

4.7 Carrot Compatibility - `kombu.compat`

Carrot compatibility interface.

See https://pypi.org/project/carrot/ for documentation.

4.7.1 Publisher

Replace with *kombu.Producer*.

class kombu.compat.**Publisher**(*connection*, *exchange=None*, *routing_key=None*, *exchange_type=None*, *durable=None*, *auto_delete=None*, *channel=None*, ***kwargs*)

Carrot compatible producer.

auto_declare = True

auto_delete = False

property backend

property channel

close()

compression = None

property connection

declare()
Declare the exchange.

> **Note:** This happens automatically at instantiation when the *auto_declare* flag is enabled.

durable = True

exchange = ''

exchange_type = 'direct'

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare exchange if not already declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=None*, *expiration=None*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** (*Any*) – Message body.
- **routing_key** (*str*) – Message routing key.
- **delivery_mode** (*enum*) – See `delivery_mode`.
- **mandatory** (*bool*) – Currently not supported.
- **immediate** (*bool*) – Currently not supported.
- **priority** (*int*) – Message priority. A number between 0 and 9.
- **content_type** (*str*) – Content type. Default is auto-detect.
- **content_encoding** (*str*) – Content encoding. Default is auto-detect.
- **serializer** (*str*) – Serializer to use. Default is auto-detect.
- **compression** (*str*) – Compression method to use. Default is none.
- **headers** (*Dict*) – Mapping of arbitrary headers to pass along with the message body.
- **exchange** (*kombu.entity.Exchange, str*) – Override the exchange. Note that this exchange must have been declared.
- **declare** (*Sequence[EntityT]*) – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using *maybe_declare()*.
- **retry** (*bool*) – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** (*Dict*) – Retry configuration, this is the keywords supported by *ensure()*.
- **expiration** (*float*) – A TTL in seconds can be specified per message. Default is no expiration.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

release()

revive(*channel*)
Revive the producer after connection loss.

routing_key = ''

send(**args*, ***kwargs*)

serializer = None

4.7.2 Consumer

Replace with *kombu.Consumer*.

class kombu.compat.**Consumer**(*connection*, *queue=None*, *exchange=None*, *routing_key=None*, *exchange_type=None*, *durable=None*, *exclusive=None*, *auto_delete=None*, ***kwargs*)
Carrot compatible consumer.

exception ContentDisallowed
Consumer does not allow this content-type.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

accept = None

add_queue(*queue*)
Add a queue to the list of queues to consume from.

Note: This will not start consuming from the queue, for that you will have to call *consume()* after.

---

auto_declare = True

auto_delete = False

callbacks = None

cancel()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

cancel_by_queue(*queue*)
Cancel consumer by queue name.

channel = None

close()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

property connection

consume(*no_ack=None*)
Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use *cancel_by_queue()*).

Parameters **no_ack** (*bool*) – See *no_ack*.

consuming_from(*queue*)
Return `True` if currently consuming from queue'.

declare()
Declare queues, exchanges and bindings.

Note: This is done automatically at instantiation when *auto_declare* is set.

discard_all(*filterfunc=None*)

durable = True

exchange = ''

exchange_type = 'direct'

exclusive = False

fetch(*no_ack=None*, *enable_callbacks=False*)

flow(*active*)
Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

iterconsume(*limit=None*, *no_ack=None*)

iterqueue(*limit=None*, *infinite=False*)

no_ack = None

on_decode_error = None

on_message = None

prefetch_count = None

process_next()

purge()
Purge messages from all queues.

Warning: This will *delete all ready messages*, there is no undo operation.

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the *no_ack* option is set.

Parameters

- **prefetch_size** (*int*) – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
- **prefetch_count** (*int*) – Specify the prefetch window in terms of whole messages.
- **apply_global** (*bool*) – Apply new settings globally on all channels.

queue = ''

property queues

receive(*body*, *message*)
Method called when a message is received.

This dispatches to the registered *callbacks*.

Parameters

- **body** (*Any*) – The decoded message body.
- **message** (*Message*) – The message instance.

Raises **NotImplementedError** – If no consumer callbacks have been registered.

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

Asks the broker to redeliver all unacknowledged messages on the specified channel.

Parameters **requeue** (*bool*) – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

register_callback(*callback*)
Register a new callback to be called when a message is received.

Note: The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

---

revive(*channel*)
Revive consumer after connection loss.

routing_key = ''

wait(*limit=None*)

4.7.3 ConsumerSet

Replace with *kombu.Consumer*.

class kombu.compat.**ConsumerSet**(*connection*, *from_dict=None*, *consumers=None*, *channel=None*, ***kwargs*)

exception ContentDisallowed
Consumer does not allow this content-type.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

accept = None

add_consumer(*consumer*)

add_consumer_from_dict(*queue*, ***options*)

add_queue(*queue*)
Add a queue to the list of queues to consume from.

Note: This will not start consuming from the queue, for that you will have to call *consume()* after.

---

auto_declare = True

callbacks = None

cancel()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

cancel_by_queue(*queue*)
Cancel consumer by queue name.

channel = None

close()
End all active queue consumers.

Note: This does not affect already delivered messages, but it does mean the server will not send any more messages for this consumer.

---

property connection

consume(*no_ack=None*)
Start consuming messages.

Can be called multiple times, but note that while it will consume from new queues added since the last call, it will not cancel consuming from removed queues (use *cancel_by_queue()*).

Parameters **no_ack** (*bool*) – See *no_ack*.

consuming_from(*queue*)
Return `True` if currently consuming from queue'.

declare()
Declare queues, exchanges and bindings.

Note: This is done automatically at instantiation when *auto_declare* is set.

---

discard_all()

flow(*active*)
Enable/disable flow from peer.

This is a simple flow-control mechanism that a peer can use to avoid overflowing its queues or otherwise finding itself receiving more messages than it can process.

The peer that receives a request to stop sending content will finish sending the current content (if any), and then wait until flow is reactivated.

iterconsume(*limit=None*, *no_ack=False*)

no_ack = None

on_decode_error = None

on_message = None

prefetch_count = None

purge()
Purge messages from all queues.

> **Warning:** This will *delete all ready messages*, there is no undo operation.

qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Specify quality of service.

The client can request that messages should be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement.

The prefetch window is Ignored if the *no_ack* option is set.

Parameters

- **prefetch_size** (*int*) – Specify the prefetch window in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls within other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply.
- **prefetch_count** (*int*) – Specify the prefetch window in terms of whole messages.
- **apply_global** (*bool*) – Apply new settings globally on all channels.

property queues

receive(*body*, *message*)
Method called when a message is received.

This dispatches to the registered *callbacks*.

Parameters

- **body** (*Any*) – The decoded message body.
- **message** (*Message*) – The message instance.

Raises **NotImplementedError** – If no consumer callbacks have been registered.

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

Asks the broker to redeliver all unacknowledged messages on the specified channel.

Parameters **requeue** (*bool*) – By default the messages will be redelivered to the original recipient. With *requeue* set to true, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

register_callback(*callback*)
Register a new callback to be called when a message is received.

Note: The signature of the callback needs to accept two arguments: *(body, message)*, which is the decoded message body and the `Message` instance.

revive(*channel*)
Revive consumer after connection loss.

4.8 Pidbox - kombu.pidbox

Generic process mailbox.

4.8.1 Introduction

Creating the applications Mailbox

```
>>> mailbox = pidbox.Mailbox('celerybeat', type='direct')

>>> @mailbox.handler
>>> def reload_schedule(state, **kwargs):
...     state['beat'].reload_schedule()

>>> @mailbox.handler
>>> def connection_info(state, **kwargs):
...     return {'connection': state['connection'].info()}
```

Example Node

```
>>> connection = kombu.Connection()
>>> state = {'beat': beat,
             'connection': connection}
>>> consumer = mailbox(connection).Node(hostname).listen()
>>> try:
```

(continues on next page)

(continued from previous page)

```
...     while True:
...         connection.drain_events(timeout=1)
... finally:
...     consumer.cancel()
```

Example Client

```
>>> mailbox.cast('reload_schedule')    # cast is async.
>>> info = celerybeat.call('connection_info', timeout=1)
```

4.8.2 Mailbox

class kombu.pidbox.**Mailbox**(*namespace*, *type='direct'*, *connection=None*, *clock=None*, *accept=None*, *serializer=None*, *producer_pool=None*, *queue_ttl=None*, *queue_expires=None*, *reply_queue_ttl=None*, *reply_queue_expires=10.0*)

Process Mailbox.

namespace = None
Name of application.

connection = None
Connection (if bound).

type = 'direct'
Exchange type (usually direct, or fanout for broadcast).

exchange = None
mailbox exchange (init by constructor).

reply_exchange = None
exchange to send replies to.

Node(*hostname=None*, *state=None*, *channel=None*, *handlers=None*)

call(*destination*, *command*, *kwargs=None*, *timeout=None*, *callback=None*, *channel=None*)

cast(*destination*, *command*, *kwargs=None*)

abcast(*command*, *kwargs=None*)

multi_call(*command*, *kwargs=None*, *timeout=1*, *limit=None*, *callback=None*, *channel=None*)

get_reply_queue()

get_queue(*hostname*)

4.8.3 Node

class kombu.pidbox.**Node**(*hostname*, *state=None*, *channel=None*, *handlers=None*, *mailbox=None*)

Mailbox node.

hostname = None
hostname of the node.

mailbox = None
the *Mailbox* this is a node for.

handlers = None
map of method name/handlers.

state = None
current context (passed on to handlers)

channel = None
current channel.

Consumer(*channel=None*, *no_ack=True*, *accept=None*, ***options*)

handler(*fun*)

listen(*channel=None*, *callback=None*)

dispatch(*method*, *arguments=None*, *reply_to=None*, *ticket=None*, ***kwargs*)

dispatch_from_message(*body*, *message=None*)

handle_call(*method*, *arguments*)

handle_cast(*method*, *arguments*)

handle(*method*, *arguments=None*)

handle_message(*body*, *message=None*)

reply(*data*, *exchange*, *routing_key*, *ticket*, ***kwargs*)

4.9 Exceptions - kombu.exceptions

Exceptions.

exception kombu.exceptions.**NotBoundError**
Trying to call channel dependent method on unbound entity.

exception kombu.exceptions.**MessageStateError**
The message has already been acknowledged.

kombu.exceptions.**TimeoutError**
alias of socket.timeout

exception kombu.exceptions.**LimitExceeded**
Limit exceeded.

exception kombu.exceptions.**ConnectionLimitExceeded**
Maximum number of simultaneous connections exceeded.

exception kombu.exceptions.**ChannelLimitExceeded**
Maximum number of simultaneous channels exceeded.

4.10 Logging - kombu.log

Logging Utilities.

class kombu.log.**LogMixin**
Mixin that adds severity methods to any class.

annotate(*text*)

critical(**args*, ***kwargs*)

debug(**args*, ***kwargs*)

error(**args*, ***kwargs*)

get_logger()

get_loglevel(*level*)

info(**args*, ***kwargs*)

is_enabled_for(*level*)

log(*severity*, **args*, ***kwargs*)

logger

property logger_name

warn(**args*, ***kwargs*)

kombu.log.**get_loglevel**(*level*)
Get loglevel by name.

kombu.log.**setup_logging**(*loglevel=None*, *logfile=None*)
Setup logging.

4.11 Connection - kombu.connection

Client (Connection).

4.11.1 Connection

class kombu.connection.**Connection**(*hostname='localhost'*, *userid=None*, *password=None*, *virtual_host=None*, *port=None*, *insist=False*, *ssl=False*, *transport=None*, *connect_timeout=5*, *transport_options=None*, *login_method=None*, *uri_prefix=None*, *heartbeat=0*, *failover_strategy='round-robin'*, *alternates=None*, ***kwargs*)
A connection to the broker.

Example

```
>>> Connection('amqp://guest:guest@localhost:5672//')
>>> Connection('amqp://foo;amqp://bar',
...            failover_strategy='round-robin')
>>> Connection('redis://', transport_options={
...     'visibility_timeout': 3000,
... })
```

```
>>> import ssl
>>> Connection('amqp://', login_method='EXTERNAL', ssl={
...     'ca_certs': '/etc/pki/tls/certs/something.crt',
...     'keyfile': '/etc/something/system.key',
...     'certfile': '/etc/something/system.cert',
...     'cert_reqs': ssl.CERT_REQUIRED,
... })
```

Note: SSL currently only works with the py-amqp, and qpid transports. For other transports you can use stunnel.

Parameters **URL** (*str, Sequence*) – Broker URL, or a list of URLs.

Keyword Arguments

- **ssl** (*bool*) – Use SSL to connect to the server. Default is `False`. May not be supported by the specified transport.
- **transport** (`Transport`) – Default transport if not specified in the URL.
- **connect_timeout** (*float*) – Timeout in seconds for connecting to the server. May not be supported by the specified transport.
- **transport_options** (*Dict*) – A dict of additional connection arguments to pass to alternate kombu channel implementations. Consult the transport documentation for available options.
- **heartbeat** (*float*) – Heartbeat interval in int/float seconds. Note that if heartbeats are enabled then the *heartbeat_check()* method must be called regularly, around once per second.

Note: The connection is established lazily when needed. If you need the connection to be established, then force it by calling *connect()*:

```
>>> conn = Connection('amqp://')
>>> conn.connect()
```

and always remember to close the connection:

```
>>> conn.release()
```

These options have been replaced by the URL argument, but are still supported for backwards compatibility:

Keyword Arguments

- **hostname** – Host name/address. NOTE: You cannot specify both the URL argument and use the hostname keyword argument at the same time.
- **userid** – Default user name if not provided in the URL.
- **password** – Default password if not provided in the URL.
- **virtual_host** – Default virtual host if not provided in the URL.
- **port** – Default port if not provided in the URL.

ChannelPool(*limit=None*, ***kwargs*)
Pool of channels.

See also:

ChannelPool.

Parameters **limit** (*int*) – Maximum number of active channels. Default is no limit.

Example

```
>>> connection = Connection('amqp://')
>>> pool = connection.ChannelPool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
  raise ChannelLimitExceeded(self.limit)
    kombu.connection.ChannelLimitExceeded: 2
```

Consumer(*queues=None*, *channel=None*, **args*, ***kwargs*)
Create new *kombu.Consumer* instance.

Pool(*limit=None*, ***kwargs*)
Pool of connections.

See also:

ConnectionPool.

Parameters **limit** (*int*) – Maximum number of active connections. Default is no limit.

Example

```
>>> connection = Connection('amqp://')
>>> pool = connection.Pool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "kombu/connection.py", line 354, in acquire
  raise ConnectionLimitExceeded(self.limit)
    kombu.exceptions.ConnectionLimitExceeded: 2
```

Producer(*channel=None*, **args*, ***kwargs*)
Create new *kombu.Producer* instance.

SimpleBuffer(*name*, *no_ack=None*, *queue_opts=None*, *queue_args=None*, *exchange_opts=None*, *channel=None*, ***kwargs*)
Simple ephemeral queue API.

Create new *SimpleQueue* using a channel from this connection.

See also:

Same as *SimpleQueue()*, but configured with buffering semantics. The resulting queue and exchange will not be durable, also auto delete is enabled. Messages will be transient (not persistent), and acknowledgments are disabled (no_ack).

SimpleQueue(*name*, *no_ack=None*, *queue_opts=None*, *queue_args=None*, *exchange_opts=None*, *channel=None*, ***kwargs*)
Simple persistent queue API.

Create new *SimpleQueue*, using a channel from this connection.

If name is a string, a queue and exchange will be automatically created using that name as the name of the queue and exchange, also it will be used as the default routing key.

Parameters

- **name** (*str, kombu.Queue*) – Name of the queue/or a queue.
- **no_ack** (*bool*) – Disable acknowledgments. Default is false.
- **queue_opts** (*Dict*) – Additional keyword arguments passed to the constructor of the automatically created *Queue*.
- **queue_args** (*Dict*) – Additional keyword arguments passed to the constructor of the automatically created *Queue* for setting implementation extensions (e.g., in RabbitMQ).
- **exchange_opts** (*Dict*) – Additional keyword arguments passed to the constructor of the automatically created *Exchange*.
- **channel** (*ChannelT*) – Custom channel to use. If not specified the connection default channel is used.

as_uri(*include_password=False*, *mask='**'*, *getfields=operator.itemgetter('port', 'userid', 'password', 'virtual_host', 'transport')*)
Convert connection parameters to URL form.

autoretry(*fun*, *channel=None*, ***ensure_options*)
Decorator for functions supporting a channel keyword argument.

The resulting callable will retry calling the function if it raises connection or channel related errors. The return value will be a tuple of (retval, last_created_channel).

If a channel is not provided, then one will be automatically acquired (remember to close it afterwards).

See also:

ensure() for the full list of supported keyword arguments.

Example

```
>>> channel = connection.channel()
>>> try:
...     ret, channel = connection.autoretry(
...         publish_messages, channel)
... finally:
...     channel.close()
```

channel()
Create and return a new channel.

channel_errors
List of exceptions that may be raised by the channel.

clone(***kwargs*)
Create a copy of the connection with same settings.

close()
Close the connection (if open).

collect(*socket_timeout=None*)

completes_cycle(*retries*)
Return true if the cycle is complete after number of *retries*.

connect()
Establish connection to server immediately.

connect_timeout = 5

property connected
Return true if the connection has been established.

property connection
The underlying connection object.

> **Warning:** This instance is transport specific, so do not depend on the interface of this object.

connection_errors
List of exceptions that may be raised by the connection.

create_transport()

cycle = None
Iterator returning the next broker URL to try in the event of connection failure (initialized by *failover_strategy*).

declared_entities = None
The cache of declared entities is per connection, in case the server loses data.

property default_channel
Default channel.

Created upon access and closed when the connection is closed.

Note: Can be used for automatic channel handling when you only need one channel, and also it is the channel implicitly used if a connection is passed instead of a channel, to functions that require a channel.

---

drain_events(***kwargs*)
Wait for a single event from the server.

Parameters **timeout** (*float*) – Timeout in seconds before we give up.

Raises **socket.timeout** – if the timeout is exceeded.

ensure(*obj*, *fun*, *errback=None*, *max_retries=None*, *interval_start=1*, *interval_step=1*, *interval_max=1*, *on_revive=None*)
Ensure operation completes.

Regardless of any channel/connection errors occurring.

Retries by establishing the connection, and reapplying the function.

Parameters

- **obj** – The object to ensure an action on.
- **fun** (Callable) – Method to apply.
- **errback** (Callable) – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept `(exc, interval)`.
- **max_retries** (*int*) – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval_start** (*float*) – The number of seconds we start sleeping for.
- **interval_step** (*float*) – How many seconds added to the interval for each retry.
- **interval_max** (*float*) – Maximum number of seconds to sleep between each retry.
- **on_revive** (Callable) – Optional callback called whenever revival completes successfully

Examples

```
>>> from kombu import Connection, Producer
>>> conn = Connection('amqp://')
>>> producer = Producer(conn)
```

```
>>> def errback(exc, interval):
...     logger.error('Error: %r', exc, exc_info=1)
...     logger.info('Retry in %s seconds.', interval)
```

```
>>> publish = conn.ensure(producer, producer.publish,
...                       errback=errback, max_retries=3)
>>> publish({'hello': 'world'}, routing_key='dest')
```

ensure_connection(*errback=None*, *max_retries=None*, *interval_start=2*, *interval_step=2*, *interval_max=30*, *callback=None*, *reraise_as_library_errors=True*, *timeout=None*)

Ensure we have a connection to the server.

If not retry establishing the connection with the settings specified.

Parameters

- **errback** (Callable) – Optional callback called each time the connection can't be established. Arguments provided are the exception raised and the interval that will be slept `(exc, interval)`.
- **max_retries** (*int*) – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval_start** (*float*) – The number of seconds we start sleeping for.
- **interval_step** (*float*) – How many seconds added to the interval for each retry.
- **interval_max** (*float*) – Maximum number of seconds to sleep between each retry.
- **callback** (Callable) – Optional callback that is called for every internal iteration (1 s).
- **timeout** (*int*) – Maximum amount of time in seconds to spend waiting for connection

failover_strategies = {'round-robin': <class 'itertools.cycle'>, 'shuffle': <functio

failover_strategy = 'round-robin'
Strategy used to select new hosts when reconnecting after connection failure. One of "round-robin", "shuffle" or any custom iterator constantly yielding new URLs to try.

get_heartbeat_interval()

get_manager(**args*, ***kwargs*)

get_transport_cls()
Get the currently used transport class.

heartbeat = None
Heartbeat value, currently only supported by the py-amqp transport.

heartbeat_check(*rate=2*)
Check heartbeats.

Allow the transport to perform any periodic tasks required to make heartbeats work. This should be called approximately every second.

If the current transport does not support heartbeats then this is a noop operation.

Parameters **rate** (*int*) – Rate is how often the tick is called compared to the actual heartbeat value. E.g. if the heartbeat is set to 3 seconds, and the tick is called every 3 / 2 seconds, then the rate is 2. This value is currently unused by any transports.

property host
The host as a host name/port pair separated by colon.

hostname = None

info()
Get connection info.

property is_evented

login_method = None

manager
AMQP Management API.

Experimental manager that can be used to manage/monitor the broker instance.

Not available for all transports.

maybe_close_channel(*channel*)
Close given channel, but ignore connection and channel errors.

maybe_switch_next()
Switch to next URL given by the current failover strategy.

password = None

port = None

property qos_semantics_matches_spec

recoverable_channel_errors
Recoverable channel errors.

List of channel related exceptions that can be automatically recovered from without re-establishing the connection.

recoverable_connection_errors
Recoverable connection errors.

List of connection related exceptions that can be recovered from, but where the connection must be closed and re-established first.

register_with_event_loop(*loop*)

release()
Close the connection (if open).

resolve_aliases = {'librabbitmq': 'amqp', 'pyamqp': 'amqp'}

revive(*new_channel*)
Revive connection after connection re-established.

ssl = None

supports_exchange_type(*exchange_type*)

property supports_heartbeats

switch(*conn_str*)
Switch connection parameters to use a new URL or hostname.

Note: Does not reconnect!

Parameters **conn_str** (*str*) – either a hostname or URL.

property transport

transport_options = None
Additional transport specific options, passed on to the transport instance.

uri_prefix = None

userid = None

virtual_host = '/'

4.11.2 Pools

See also:

The shortcut methods *Connection.Pool()* and *Connection.ChannelPool()* is the recommended way to instantiate these classes.

class kombu.connection.**ConnectionPool**(*connection*, *limit=None*, ***kwargs*)
Pool of connections.

LimitExceeded = <class 'kombu.exceptions.ConnectionLimitExceeded'>

acquire(*block=False*, *timeout=None*)
Acquire resource.

Parameters

- **block** (*bool*) – If the limit is exceeded, then block until there is an available item.
- **timeout** (*float*) – Timeout to wait if `block` is true. Default is `None` (forever).

Raises ***LimitExceeded*** – if block is false and the limit has been exceeded.

release(*resource*)

force_close_all()
Close and remove all resources in the pool (also those in use).

Used to close resources from parent processes after fork (e.g. sockets/connections).

class kombu.connection.**ChannelPool**(*connection*, *limit=None*, ***kwargs*)
Pool of channels.

LimitExceeded = <class 'kombu.exceptions.ChannelLimitExceeded'>

acquire(*block=False*, *timeout=None*)
Acquire resource.

Parameters

- **block** (*bool*) – If the limit is exceeded, then block until there is an available item.
- **timeout** (*float*) – Timeout to wait if `block` is true. Default is `None` (forever).

Raises ***LimitExceeded*** – if block is false and the limit has been exceeded.

release(*resource*)

force_close_all()
Close and remove all resources in the pool (also those in use).

Used to close resources from parent processes after fork (e.g. sockets/connections).

4.12 Message Objects - kombu.message

Message class.

class kombu.message.**Message**(*body=None*, *delivery_tag=None*, *content_type=None*, *content_encoding=None*, *delivery_info=None*, *properties=None*, *headers=None*, *postencode=None*, *accept=None*, *channel=None*, ***kwargs*)
Base class for received messages.

Keyword Arguments

- **channel** (*ChannelT*) – If message was received, this should be the channel that the message was received on.
- **body** (*str*) – Message body.
- **delivery_mode** (*bool*) – Set custom delivery mode. Defaults to `delivery_mode`.
- **priority** (*int*) – Message priority, 0 to broker configured max priority, where higher is better.
- **content_type** (*str*) – The messages content_type. If content_type is set, no serialization occurs as it is assumed this is either a binary object, or you've done your own serialization. Leave blank if using built-in serialization as our library properly sets content_type.
- **content_encoding** (*str*) – The character set in which this object is encoded. Use "binary" if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets content_encoding.
- **properties** (*Dict*) – Message properties.
- **headers** (*Dict*) – Message headers.

exception MessageStateError
The message has already been acknowledged.

accept

ack(*multiple=False*)
Acknowledge this message as being processed.

This will remove the message from the queue.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

ack_log_error(*logger*, *errors*, *multiple=False*)

property acknowledged
Set to true if the message has been acknowledged.

body

channel

content_encoding

content_type

decode()
Deserialize the message body.

Returning the original python structure sent by the publisher.

Note: The return value is memoized, use *_decode* to force re-evaluation.

delivery_info

delivery_tag

errors = None

headers

property payload
The decoded message body.

properties

reject(*requeue=False*)
Reject this message.

The message will be discarded by the server.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

reject_log_error(*logger*, *errors*, *requeue=False*)

requeue()
Reject this message and put it back on the queue.

Warning: You must not use this method as a means of selecting messages to process.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

4.13 Message Compression - kombu.compression

Compression utilities.

4.13.1 Encoding/decoding

kombu.compression.**compress**(*body*, *content_type*)
Compress text.

Parameters

- **body** (*AnyStr*) – The text to compress.
- **content_type** (*str*) – mime-type of compression method to use.

kombu.compression.**decompress**(*body*, *content_type*)
Decompress compressed text.

Parameters

- **body** (*AnyStr*) – Previously compressed text to uncompress.
- **content_type** (*str*) – mime-type of compression method used.

4.13.2 Registry

kombu.compression.**encoders**()
Return a list of available compression methods.

kombu.compression.**get_encoder**(*t*)
Get encoder by alias name.

kombu.compression.**get_decoder**(*t*)
Get decoder by alias name.

kombu.compression.**register**(*encoder*, *decoder*, *content_type*, *aliases=None*)
Register new compression method.

Parameters

- **encoder** (*Callable*) – Function used to compress text.
- **decoder** (*Callable*) – Function used to decompress previously compressed text.
- **content_type** (*str*) – The mime type this compression method identifies as.
- **aliases** (*Sequence[str]*) – A list of names to associate with this compression method.

4.14 Connection/Producer Pools - kombu.pools

Public resource pools.

class kombu.pools.**ProducerPool**(*connections*, **args*, ***kwargs*)
Pool of *kombu.Producer* instances.

class Producer(*channel*, *exchange=None*, *routing_key=None*, *serializer=None*, *auto_declare=None*, *compression=None*, *on_return=None*)
Message Producer.

Parameters

- **channel** (*kombu.Connection, ChannelT*) – Connection or channel.
- **exchange** (*kombu.entity.Exchange, str*) – Optional default exchange.
- **routing_key** (*str*) – Optional default routing key.
- **serializer** (*str*) – Default serializer. Default is *"json"*.
- **compression** (*str*) – Default compression method. Default is no compression.
- **auto_declare** (*bool*) – Automatically declare the default exchange at instantiation. Default is `True`.
- **on_return** (*Callable*) – Callback to call for undeliverable messages, when the *mandatory* or *immediate* arguments to *publish()* is used. This callback needs the following signature: *(exception, exchange, routing_key, message)*. Note that the producer needs to drain events to use this feature.

auto_declare = True

property channel

close()

compression = None

property connection

declare()
Declare the exchange.

> **Note:** This happens automatically at instantiation when the *auto_declare* flag is enabled.

exchange = None

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare exchange if not already declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=None*, *expiration=None*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** (*Any*) – Message body.
- **routing_key** (*str*) – Message routing key.
- **delivery_mode** (*enum*) – See `delivery_mode`.

- **mandatory** (*bool*) – Currently not supported.
- **immediate** (*bool*) – Currently not supported.
- **priority** (*int*) – Message priority. A number between 0 and 9.
- **content_type** (*str*) – Content type. Default is auto-detect.
- **content_encoding** (*str*) – Content encoding. Default is auto-detect.
- **serializer** (*str*) – Serializer to use. Default is auto-detect.
- **compression** (*str*) – Compression method to use. Default is none.
- **headers** (*Dict*) – Mapping of arbitrary headers to pass along with the message body.
- **exchange** (*kombu.entity.Exchange, str*) – Override the exchange. Note that this exchange must have been declared.
- **declare** (*Sequence[EntityT]*) – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using *maybe_declare()*.
- **retry** (*bool*) – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** (*Dict*) – Retry configuration, this is the keywords supported by *ensure()*.
- **expiration** (*float*) – A TTL in seconds can be specified per message. Default is no expiration.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

release()

revive(*channel*)
Revive the producer after connection loss.

routing_key = ''

serializer = None

close_after_fork = True

close_resource(*resource*)

create_producer()

new()

prepare(*p*)

release(*resource*)

setup()

class kombu.pools.**PoolGroup**(*limit=None*, *close_after_fork=True*)
Collection of resource pools.

create(*resource*, *limit*)

kombu.pools.**register_group**(*group*)
Register group (can be used as decorator).

kombu.pools.**get_limit**()
Get current connection pool limit.

kombu.pools.**set_limit**(*limit*, *force=False*, *reset_after=False*, *ignore_errors=False*)
Set new connection pool limit.

kombu.pools.**reset**(**args*, ***kwargs*)
Reset all pools by closing open resources.

4.15 Abstract Classes - kombu.abstract

Object utilities.

class kombu.abstract.**MaybeChannelBound**(**args*, ***kwargs*)
Mixin for classes that can be bound to an AMQP channel.

bind(*channel*)
Create copy of the instance that is bound to a channel.

can_cache_declaration = False
Defines whether maybe_declare can skip declaring this entity twice.

property channel
Current channel if the object is bound.

property is_bound
Flag set if the channel is bound.

maybe_bind(*channel*)
Bind instance to channel if not already bound.

revive(*channel*)
Revive channel after the connection has been re-established.

Used by *ensure()*.

when_bound()
Callback called when the class is bound.

4.16 Resource Management - kombu.resource

Generic resource pool implementation.

class kombu.resource.**LifoQueue**(*maxsize=0*)
Last in first out version of Queue.

class kombu.resource.**Resource**(*limit=None*, *preload=None*, *close_after_fork=None*)
Pool of resources.

exception LimitExceeded
Limit exceeded.

acquire(*block=False*, *timeout=None*)
Acquire resource.

Parameters

- **block** (*bool*) – If the limit is exceeded, then block until there is an available item.
- **timeout** (*float*) – Timeout to wait if `block` is true. Default is `None` (forever).

Raises ***LimitExceeded*** – if block is false and the limit has been exceeded.

close_after_fork = False

close_resource(*resource*)

collect_resource(*resource*)

force_close_all()
Close and remove all resources in the pool (also those in use).

Used to close resources from parent processes after fork (e.g. sockets/connections).

property limit

prepare(*resource*)

release(*resource*)

release_resource(*resource*)

replace(*resource*)
Replace existing resource with a new instance.

This can be used in case of defective resources.

resize(*limit*, *force=False*, *ignore_errors=False*, *reset=False*)

setup()

4.17 Event Loop - kombu.asynchronous

Event loop.

class kombu.asynchronous.**Hub**(*timer=None*)
Event loop object.

Parameters **timer** (*kombu.asynchronous.Timer*) – Specify custom timer instance.

ERR = 24
Flag set on error, and the fd should be read from asap.

READ = 1
Flag set if reading from an fd will not block.

WRITE = 4
Flag set if writing to an fd will not block.

add(*fd*, *callback*, *flags*, *args=()*, *consolidate=False*)

add_reader(*fds*, *callback*, **args*)

add_writer(*fds*, *callback*, **args*)

call_at(*when*, *callback*, **args*)

call_later(*delay*, *callback*, **args*)

call_repeatedly(*delay*, *callback*, **args*)

call_soon(*callback*, **args*)

close(**args*)

create_loop(*generator=<class 'generator'>*, *sleep=<built-in function sleep>*, *min=<built-in function min>*, *next=<built-in function next>*, *Empty=<class '_queue.Empty'>*, *StopIteration=<class 'StopIteration'>*, *KeyError=<class 'KeyError'>*, *READ=1*, *WRITE=4*, *ERR=24*)

fire_timers(*min_delay=1*, *max_delay=10*, *max_timers=10*, *propagate=()*)

property loop

on_callback_error(*callback*, *exc*)

on_close = None
List of callbacks to be called when the loop is exiting, applied with the hub instance as sole argument.

property poller

remove(*fd*)

remove_reader(*fd*)

remove_writer(*fd*)

repr_active()

repr_events(*events*)

reset()

run_forever()

run_once()

scheduler

stop()

kombu.asynchronous.**get_event_loop**()
Get current event loop object.

kombu.asynchronous.**set_event_loop**(*loop*)
Set the current event loop object.

4.18 Event Loop Implementation - kombu.asynchronous.hub

Event loop implementation.

class kombu.asynchronous.hub.**Hub**(*timer=None*)
Event loop object.

Parameters **timer** (*kombu.asynchronous.Timer*) – Specify custom timer instance.

ERR = 24
Flag set on error, and the fd should be read from asap.

READ = 1
Flag set if reading from an fd will not block.

WRITE = 4
Flag set if writing to an fd will not block.

add(*fd*, *callback*, *flags*, *args=()*, *consolidate=False*)

add_reader(*fds*, *callback*, **args*)

add_writer(*fds*, *callback*, **args*)

call_at(*when*, *callback*, **args*)

call_later(*delay*, *callback*, **args*)

call_repeatedly(*delay*, *callback*, **args*)

call_soon(*callback*, **args*)

close(**args*)

create_loop(*generator=<class 'generator'>*, *sleep=<built-in function sleep>*, *min=<built-in function min>*, *next=<built-in function next>*, *Empty=<class '_queue.Empty'>*, *StopIteration=<class 'StopIteration'>*, *KeyError=<class 'KeyError'>*, *READ=1*, *WRITE=4*, *ERR=24*)

fire_timers(*min_delay=1*, *max_delay=10*, *max_timers=10*, *propagate=()*)

property loop

on_callback_error(*callback*, *exc*)

on_close = None
List of callbacks to be called when the loop is exiting, applied with the hub instance as sole argument.

property poller

remove(*fd*)

remove_reader(*fd*)

remove_writer(*fd*)

repr_active()

repr_events(*events*)

reset()

run_forever()

run_once()

scheduler

stop()

kombu.asynchronous.hub.**get_event_loop**()
Get current event loop object.

kombu.asynchronous.hub.**set_event_loop**(*loop*)
Set the current event loop object.

4.19 Semaphores - kombu.asynchronous.semaphore

Semaphores and concurrency primitives.

class kombu.asynchronous.semaphore.**DummyLock**
Pretending to be a lock.

class kombu.asynchronous.semaphore.**LaxBoundedSemaphore**(*value*)
Asynchronous Bounded Semaphore.

Lax means that the value will stay within the specified range even if released more times than it was acquired.

Example

```
>>> from future import print_statement as printf
# ^ ignore: just fooling stupid pyflakes
```

```
>>> x = LaxBoundedSemaphore(2)
```

```
>>> x.acquire(printf, 'HELLO 1')
HELLO 1
```

```
>>> x.acquire(printf, 'HELLO 2')
HELLO 2
```

```
>>> x.acquire(printf, 'HELLO 3')
>>> x._waiters   # private, do not access directly
[print, ('HELLO 3',)]
```

```
>>> x.release()
HELLO 3
```

acquire(*callback*, **partial_args*, ***partial_kwargs*)
Acquire semaphore.

This will immediately apply `callback` if the resource is available, otherwise the callback is suspended until the semaphore is released.

Parameters

- **callback** (*Callable*) – The callback to apply.
- ***partial_args** (*Any*) – partial arguments to callback.

clear()
Reset the semaphore, which also wipes out any waiting callbacks.

grow(*n=1*)
Change the size of the semaphore to accept more users.

release()
Release semaphore.

Note: If there are any waiters this will apply the first waiter that is waiting for the resource (FIFO order).

---

shrink(*n=1*)
Change the size of the semaphore to accept less users.

4.20 Timer - kombu.asynchronous.timer

Timer scheduling Python callbacks.

class kombu.asynchronous.timer.**Entry**(*fun*, *args=None*, *kwargs=None*)
Schedule Entry.

args

cancel()

canceled

property cancelled

fun

kwargs

tref

class kombu.asynchronous.timer.**Timer**(*max_interval=None*, *on_error=None*, ***kwargs*)
Async timer implementation.

class Entry(*fun*, *args=None*, *kwargs=None*)
Schedule Entry.

args

cancel()

canceled

property cancelled

fun

kwargs

tref

apply_entry(*entry*)

call_after(*secs*, *fun*, *args=()*, *kwargs=None*, *priority=0*)

call_at(*eta*, *fun*, *args=()*, *kwargs=None*, *priority=0*)

call_repeatedly(*secs*, *fun*, *args=()*, *kwargs=None*, *priority=0*)

cancel(*tref*)

clear()

enter_after(*secs*, *entry*, *priority=0*, *time=<built-in function monotonic>*)

enter_at(*entry*, *eta=None*, *priority=0*, *time=<built-in function monotonic>*)
Enter function into the scheduler.

Parameters

- **entry** (Entry) – Item to enter.
- **eta** (*datetime.datetime*) – Scheduled time.
- **priority** (*int*) – Unused.

handle_error(*exc_info*)

on_error = None

property queue
Snapshot of underlying datastructure.

property schedule

stop()

kombu.asynchronous.timer.**to_timestamp**(*d*, *default_timezone=<UTC>*, *time=<built-in function monotonic>*)
Convert datetime to timestamp.

If d' is already a timestamp, then that will be used.

4.21 Event Loop Debugging Utils - kombu.asynchronous.debug

Event-loop debugging tools.

kombu.asynchronous.debug.**callback_for**(*h*, *fd*, *flag*, **default*)
Return the callback used for hub+fd+flag.

kombu.asynchronous.debug.**repr_active**(*h*)
Return description of active readers and writers.

kombu.asynchronous.debug.**repr_events**(*h*, *events*)
Return description of events returned by poll.

kombu.asynchronous.debug.**repr_flag**(*flag*)
Return description of event loop flag.

kombu.asynchronous.debug.**repr_readers**(*h*)
Return description of pending readers.

kombu.asynchronous.debug.**repr_writers**(*h*)
Return description of pending writers.

4.22 Async HTTP Client - kombu.asynchronous.http

kombu.asynchronous.http.**Client**(*hub=None*, ***kwargs*)
Create new HTTP client.

class kombu.asynchronous.http.**Headers**
Represents a mapping of HTTP headers.

complete = False
Set when all of the headers have been read.

class kombu.asynchronous.http.**Response**(*request*, *code*, *headers=None*, *buffer=None*, *effective_url=None*, *error=None*, *status=None*)
HTTP Response.

Parameters

- **request** (Request) – See *request*.
- **code** (*int*) – See *code*.
- **headers** (Headers) – See *headers*.
- **buffer** (*bytes*) – See *buffer*
- **effective_url** (*str*) – See *effective_url*.
- **status** (*str*) – See *status*.

request
object used to get this response.

Type *Request*

code
HTTP response code (e.g. 200, 404, or 500).

Type int

headers
HTTP headers for this response.

Type *Headers*

buffer
Socket read buffer.

Type bytes

effective_url
The destination url for this request after following redirects.

Type str

error
Error instance if the request resulted in a HTTP error code.

Type Exception

status
Human equivalent of `code`, e.g. `OK`, *Not found*, or 'Internal Server Error'.

Type str

property body
The full contents of the response body.

Note: Accessing this propery will evaluate the buffer and subsequent accesses will be cached.

buffer

code

property content

effective_url

error

headers

raise_for_error()
Raise if the request resulted in an HTTP error code.

:raises `HttpError`:

request

status

property status_code

class kombu.asynchronous.http.**Request**(*url*, *method='GET'*, *on_ready=None*, *on_timeout=None*, *on_stream=None*, *on_prepare=None*, *on_header=None*, *headers=None*, ***kwargs*)
A HTTP Request.

Parameters

- **url** (*str*) – The URL to request.
- **method** (*str*) – The HTTP method to use (defaults to `GET`).

Keyword Arguments

- **headers** (*Dict, Headers*) – Optional headers for this request
- **body** (*str*) – Optional body for this request.
- **connect_timeout** (*float*) – Connection timeout in float seconds Default is 30.0.
- **timeout** (*float*) – Time in float seconds before the request times out Default is 30.0.
- **follow_redirects** (*bool*) – Specify if the client should follow redirects Enabled by default.
- **max_redirects** (*int*) – Maximum number of redirects (default 6).
- **use_gzip** (*bool*) – Allow the server to use gzip compression. Enabled by default.
- **validate_cert** (*bool*) – Set to true if the server certificate should be verified when performing `https://` requests. Enabled by default.
- **auth_username** (*str*) – Username for HTTP authentication.
- **auth_password** (*str*) – Password for HTTP authentication.
- **auth_mode** (*str*) – Type of HTTP authentication (`basic` or `digest`).
- **user_agent** (*str*) – Custom user agent for this request.
- **network_interace** (*str*) – Network interface to use for this request.
- **on_ready** (`Callable`) – Callback to be called when the response has been received. Must accept single `response` argument.
- **on_stream** (`Callable`) – Optional callback to be called every time body content has been read from the socket. If specified then the response body and buffer attributes will not be available.
- **on_timeout** (*callable*) – Optional callback to be called if the request times out.
- **on_header** (`Callable`) – Optional callback to be called for every header line received from the server. The signature is `(headers, line)` and note that if you want `response.headers` to be populated then your callback needs to also call `client.on_header(headers, line)`.
- **on_prepare** (`Callable`) – Optional callback that is implementation specific (e.g. curl client will pass the `curl` instance to this callback).
- **proxy_host** (*str*) – Optional proxy host. Note that a `proxy_port` must also be provided or a `ValueError` will be raised.
- **proxy_username** (*str*) – Optional username to use when logging in to the proxy.
- **proxy_password** (*str*) – Optional password to use when authenticating with the proxy server.
- **ca_certs** (*str*) – Custom CA certificates file to use.
- **client_key** (*str*) – Optional filename for client SSL key.
- **client_cert** (*str*) – Optional filename for client SSL certificate.

auth_mode = None

auth_password = None

auth_username = None

body = None

ca_certs = None

client_cert = None

client_key = None

connect_timeout = 30.0

follow_redirects = True

headers

max_redirects = 6

method

network_interface = None

on_header

on_prepare

on_ready

on_stream

on_timeout

proxy_host = None

proxy_password = None

proxy_port = None

proxy_username = None

request_timeout = 30.0

then(*callback*, *errback=None*)

url

use_gzip = True

user_agent = None

validate_cert = True

4.23 Async HTTP Client Interface - kombu.asynchronous.http.base

Base async HTTP client implementation.

class kombu.asynchronous.http.base.**Headers**

Represents a mapping of HTTP headers.

complete = False

Set when all of the headers have been read.

class kombu.asynchronous.http.base.**Response**(*request*, *code*, *headers=None*, *buffer=None*, *effective_url=None*, *error=None*, *status=None*)

HTTP Response.

Parameters

- **request** (Request) – See *request*.

- **code** (*int*) – See *code*.
- **headers** (Headers) – See *headers*.
- **buffer** (*bytes*) – See *buffer*
- **effective_url** (*str*) – See *effective_url*.
- **status** (*str*) – See *status*.

request
object used to get this response.

Type *Request*

code
HTTP response code (e.g. 200, 404, or 500).

Type int

headers
HTTP headers for this response.

Type *Headers*

buffer
Socket read buffer.

Type bytes

effective_url
The destination url for this request after following redirects.

Type str

error
Error instance if the request resulted in a HTTP error code.

Type Exception

status
Human equivalent of *code*, e.g. OK, *Not found*, or 'Internal Server Error'.

Type str

property body
The full contents of the response body.

Note: Accessing this propery will evaluate the buffer and subsequent accesses will be cached.

buffer

code

property content

effective_url

error

headers

raise_for_error()
Raise if the request resulted in an HTTP error code.

:raises HttpError:

request

status

property status_code

class kombu.asynchronous.http.base.**Request**(*url*, *method='GET'*, *on_ready=None*, *on_timeout=None*, *on_stream=None*, *on_prepare=None*, *on_header=None*, *headers=None*, ***kwargs*)

A HTTP Request.

Parameters

- **url** (*str*) – The URL to request.
- **method** (*str*) – The HTTP method to use (defaults to `GET`).

Keyword Arguments

- **headers** (*Dict, Headers*) – Optional headers for this request
- **body** (*str*) – Optional body for this request.
- **connect_timeout** (*float*) – Connection timeout in float seconds Default is 30.0.
- **timeout** (*float*) – Time in float seconds before the request times out Default is 30.0.
- **follow_redirects** (*bool*) – Specify if the client should follow redirects Enabled by default.
- **max_redirects** (*int*) – Maximum number of redirects (default 6).
- **use_gzip** (*bool*) – Allow the server to use gzip compression. Enabled by default.
- **validate_cert** (*bool*) – Set to true if the server certificate should be verified when performing `https://` requests. Enabled by default.
- **auth_username** (*str*) – Username for HTTP authentication.
- **auth_password** (*str*) – Password for HTTP authentication.
- **auth_mode** (*str*) – Type of HTTP authentication (`basic` or `digest`).
- **user_agent** (*str*) – Custom user agent for this request.
- **network_interace** (*str*) – Network interface to use for this request.
- **on_ready** (*Callable*) – Callback to be called when the response has been received. Must accept single `response` argument.
- **on_stream** (*Callable*) – Optional callback to be called every time body content has been read from the socket. If specified then the response body and buffer attributes will not be available.
- **on_timeout** (*callable*) – Optional callback to be called if the request times out.
- **on_header** (*Callable*) – Optional callback to be called for every header line received from the server. The signature is `(headers, line)` and note that if you want `response.headers` to be populated then your callback needs to also call `client.on_header(headers, line)`.
- **on_prepare** (*Callable*) – Optional callback that is implementation specific (e.g. curl client will pass the `curl` instance to this callback).
- **proxy_host** (*str*) – Optional proxy host. Note that a `proxy_port` must also be provided or a `ValueError` will be raised.

- **proxy_username** (*str*) – Optional username to use when logging in to the proxy.
- **proxy_password** (*str*) – Optional password to use when authenticating with the proxy server.
- **ca_certs** (*str*) – Custom CA certificates file to use.
- **client_key** (*str*) – Optional filename for client SSL key.
- **client_cert** (*str*) – Optional filename for client SSL certificate.

auth_mode = None

auth_password = None

auth_username = None

body = None

ca_certs = None

client_cert = None

client_key = None

connect_timeout = 30.0

follow_redirects = True

headers

max_redirects = 6

method

network_interface = None

on_header

on_prepare

on_ready

on_stream

on_timeout

proxy_host = None

proxy_password = None

proxy_port = None

proxy_username = None

request_timeout = 30.0

then(*callback*, *errback=None*)

url

use_gzip = True

user_agent = None

validate_cert = True

4.24 Async pyCurl HTTP Client - kombu.asynchronous.http.curl

HTTP Client using pyCurl.

class kombu.asynchronous.http.curl.**CurlClient**(*hub=None*, *max_clients=10*)
Curl HTTP Client.

Curl = None

add_request(*request*)

close()

on_readable(*fd*, *_pycurl=None*)

on_writable(*fd*, *_pycurl=None*)

4.25 Async Amazon AWS Client - kombu.asynchronous.aws

kombu.asynchronous.aws.**connect_sqs**(*aws_access_key_id=None*, *aws_secret_access_key=None*, ***kwargs*)
Return async connection to Amazon SQS.

4.26 Amazon AWS Connection - kombu.asynchronous.aws.connection

Amazon AWS Connection.

class kombu.asynchronous.aws.connection.**AsyncHTTPSConnection**(*strict=None*, *timeout=20.0*, *http_client=None*)
Async HTTP Connection.

class Request(*url*, *method='GET'*, *on_ready=None*, *on_timeout=None*, *on_stream=None*, *on_prepare=None*, *on_header=None*, *headers=None*, ***kwargs*)
A HTTP Request.

Parameters

- **url** (*str*) – The URL to request.
- **method** (*str*) – The HTTP method to use (defaults to GET).

Keyword Arguments

- **headers** (*Dict, Headers*) – Optional headers for this request
- **body** (*str*) – Optional body for this request.
- **connect_timeout** (*float*) – Connection timeout in float seconds Default is 30.0.
- **timeout** (*float*) – Time in float seconds before the request times out Default is 30.0.
- **follow_redirects** (*bool*) – Specify if the client should follow redirects Enabled by default.
- **max_redirects** (*int*) – Maximum number of redirects (default 6).
- **use_gzip** (*bool*) – Allow the server to use gzip compression. Enabled by default.

- **validate_cert** (*bool*) – Set to true if the server certificate should be verified when performing `https://` requests. Enabled by default.
- **auth_username** (*str*) – Username for HTTP authentication.
- **auth_password** (*str*) – Password for HTTP authentication.
- **auth_mode** (*str*) – Type of HTTP authentication (`basic` or `digest`).
- **user_agent** (*str*) – Custom user agent for this request.
- **network_interace** (*str*) – Network interface to use for this request.
- **on_ready** (`Callable`) – Callback to be called when the response has been received. Must accept single `response` argument.
- **on_stream** (`Callable`) – Optional callback to be called every time body content has been read from the socket. If specified then the response body and buffer attributes will not be available.
- **on_timeout** (*callable*) – Optional callback to be called if the request times out.
- **on_header** (`Callable`) – Optional callback to be called for every header line received from the server. The signature is `(headers, line)` and note that if you want `response.headers` to be populated then your callback needs to also call `client.on_header(headers, line)`.
- **on_prepare** (`Callable`) – Optional callback that is implementation specific (e.g. curl client will pass the `curl` instance to this callback).
- **proxy_host** (*str*) – Optional proxy host. Note that a `proxy_port` must also be provided or a `ValueError` will be raised.
- **proxy_username** (*str*) – Optional username to use when logging in to the proxy.
- **proxy_password** (*str*) – Optional password to use when authenticating with the proxy server.
- **ca_certs** (*str*) – Custom CA certificates file to use.
- **client_key** (*str*) – Optional filename for client SSL key.
- **client_cert** (*str*) – Optional filename for client SSL certificate.

auth_mode = None

auth_password = None

auth_username = None

body = None

ca_certs = None

client_cert = None

client_key = None

connect_timeout = 30.0

follow_redirects = True

headers

max_redirects = 6

method

network_interface = None

on_header

on_prepare

on_ready

on_stream

on_timeout

proxy_host = None

proxy_password = None

proxy_port = None

proxy_username = None

request_timeout = 30.0

then(*callback*, *errback=None*)

url

use_gzip = True

user_agent = None

validate_cert = True

Response
alias of AsyncHTTPResponse

body = None

close()

connect()

default_ports = {'http': 80, 'https': 443}

endheaders()

getrequest()

getresponse(*callback=None*)

method = 'GET'

path = '/'

putheader(*header*, *value*)

putrequest(*method*, *path*)

request(*method*, *path*, *body=None*, *headers=None*)

send(*data*)

set_debuglevel(*level*)

class kombu.asynchronous.aws.connection.**AsyncConnection**(*sqs_connection*, *http_client=None*, ***kwargs*)

Async AWS Connection.

get_http_connection()

4.27 Async Amazon SQS Client - kombu.asynchronous.aws.sqs

4.28 SQS Connection - kombu.asynchronous.aws.sqs.connection

Amazon SQS Connection.

class kombu.asynchronous.aws.sqs.connection.**AsyncSQSConnection**(*sqs_connection*, *debug=0*, *region=None*, ***kwargs*)

Async SQS Connection.

add_permission(*queue*, *label*, *aws_account_id*, *action_name*, *callback=None*)

change_message_visibility(*queue*, *receipt_handle*, *visibility_timeout*, *callback=None*)

change_message_visibility_batch(*queue*, *messages*, *callback=None*)

create_queue(*queue_name*, *visibility_timeout=None*, *callback=None*)

delete_message(*queue*, *receipt_handle*, *callback=None*)

delete_message_batch(*queue*, *messages*, *callback=None*)

delete_message_from_handle(*queue*, *receipt_handle*, *callback=None*)

delete_queue(*queue*, *force_deletion=False*, *callback=None*)

get_all_queues(*prefix=''*, *callback=None*)

get_dead_letter_source_queues(*queue*, *callback=None*)

get_queue(*queue_name*, *callback=None*)

get_queue_attributes(*queue*, *attribute='All'*, *callback=None*)

get_queue_url(*queue*)

lookup(*queue_name*, *callback=None*)

receive_message(*queue*, *number_messages=1*, *visibility_timeout=None*, *attributes=None*, *wait_time_seconds=None*, *callback=None*)

remove_permission(*queue*, *label*, *callback=None*)

send_message(*queue*, *message_content*, *delay_seconds=None*, *callback=None*)

send_message_batch(*queue*, *messages*, *callback=None*)

set_queue_attribute(*queue*, *attribute*, *value*, *callback=None*)

4.29 SQS Messages - kombu.asynchronous.aws.sqs.message

Amazon SQS message implementation.

class kombu.asynchronous.aws.sqs.message.**AsyncMessage**(*body=None*, *delivery_tag=None*, *content_type=None*, *content_encoding=None*, *delivery_info=None*, *properties=None*, *headers=None*, *postencode=None*, *accept=None*, *channel=None*, ***kwargs*)

Serialized message.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

encode(*value*)

Encode/decode the value using Base64 encoding.

headers

properties

class kombu.asynchronous.aws.sqs.message.**AsyncRawMessage**(*body=None*, *delivery_tag=None*, *content_type=None*, *content_encoding=None*, *delivery_info=None*, *properties=None*, *headers=None*, *postencode=None*, *accept=None*, *channel=None*, ***kwargs*)

Raw Message.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

class kombu.asynchronous.aws.sqs.message.**BaseAsyncMessage**(*body=None*, *delivery_tag=None*, *content_type=None*, *content_encoding=None*, *delivery_info=None*, *properties=None*, *headers=None*, *postencode=None*, *accept=None*, *channel=None*, ***kwargs*)

Base class for messages received on async client.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

4.30 SQS Queues - kombu.asynchronous.aws.sqs.queue

Amazon SQS queue implementation.

class kombu.asynchronous.aws.sqs.queue.**AsyncQueue**(*connection=None*, *url=None*, *message_class=<class 'kombu.asynchronous.aws.sqs.message.AsyncMessage'>*)

Async SQS Queue.

add_permission(*label*, *aws_account_id*, *action_name*, *callback=None*)

change_message_visibility_batch(*messages*, *callback=None*)

clear(**args*, ***kwargs*)

count(*page_size=10*, *vtimeout=10*, *callback=None*, *_attr='ApproximateNumberOfMessages'*)

count_slow(**args*, ***kwargs*)

delete(*callback=None*)

delete_message(*message*, *callback=None*)

delete_message_batch(*messages*, *callback=None*)

dump(**args*, ***kwargs*)

get_attributes(*attributes='All'*, *callback=None*)

get_messages(*num_messages=1*, *visibility_timeout=None*, *attributes=None*, *wait_time_seconds=None*, *callback=None*)

get_timeout(*callback=None*, *_attr='VisibilityTimeout'*)

load(**args*, ***kwargs*)

load_from_file(**args*, ***kwargs*)

load_from_filename(**args*, ***kwargs*)

load_from_s3(**args*, ***kwargs*)

read(*visibility_timeout=None*, *wait_time_seconds=None*, *callback=None*)

remove_permission(*label*, *callback=None*)

save(**args*, ***kwargs*)

save_to_file(**args*, ***kwargs*)

save_to_filename(**args*, ***kwargs*)

save_to_s3(**args*, ***kwargs*)

set_attribute(*attribute*, *value*, *callback=None*)

set_timeout(*visibility_timeout*, *callback=None*)

write(*message*, *delay_seconds=None*, *callback=None*)

write_batch(*messages*, *callback=None*)

kombu.asynchronous.aws.sqs.queue.**list_first**(*rs*)
Get the first item in a list, or None if list empty.

4.31 Built-in Transports - kombu.transport

Built-in transports.

4.31.1 Data

kombu.transport.**DEFAULT_TRANSPORT**
Default transport used when no transport specified.

kombu.transport.**TRANSPORT_ALIASES**
Mapping of transport aliases/class names.

4.31.2 Functions

kombu.transport.**get_transport_cls**(*transport=None*)
Get transport class by name.

The transport string is the full path to a transport class, e.g.:

```
"kombu.transport.pyamqp:Transport"
```

If the name does not include "." (is not fully qualified), the alias table will be consulted.

kombu.transport.**resolve_transport**(*transport=None*)
Get transport by name.

Parameters **transport** (*Union[str, type]*) – This can be either an actual transport class, or the fully qualified path to a transport class, or the alias of a transport.

4.32 Azure Storage Queues Transport - kombu.transport.azurestoragequeues

Azure Storage Queues transport.

The transport can be enabled by setting the CELERY_BROKER_URL to:

```
` azurestoragequeues://:{Storage Account Access Key}@{Storage Account Name} `
```

Note that if the access key for the storage account contains a slash, it will have to be regenerated before it can be used in the connection URL.

More information about Azure Storage Queues: https://azure.microsoft.com/en-us/services/storage/queues/

4.32.1 Transport

class kombu.transport.azurestoragequeues.**Transport**(*client*, ***kwargs*)
Azure Storage Queues transport.

class Channel(**args*, ***kwargs*)
Azure Storage Queues channel.

basic_consume(*queue*, *no_ack*, **args*, ***kwargs*)
Consume from *queue*.

property conninfo

domain_format = 'kombu%(vhost)s'

entity_name(*name*, *table={33: 45, 34: 45, 35: 45, 36: 45, 37: 45, 38: 45, 39: 45, 40: 45, 41: 45, 42: 45, 43: 45, 44: 45, 45: 45, 46: 45, 47: 45, 58: 45, 59: 45, 60: 45, 61: 45, 62: 45, 63: 45, 64: 45, 91: 45, 92: 45, 93: 45, 94: 45, 95: 45, 96: 45, 123: 45, 124: 45, 125: 45, 126: 45}*)
Format AMQP queue name into a valid Azure Storage Queue name.

no_ack = True

queue_name_prefix

property queue_service

property transport_options

default_port = None

polling_interval = 1

4.32.2 Channel

class kombu.transport.azurestoragequeues.**Channel**(**args*, ***kwargs*)
Azure Storage Queues channel.

basic_consume(*queue*, *no_ack*, **args*, ***kwargs*)
Consume from *queue*.

property conninfo

domain_format = 'kombu%(vhost)s'

entity_name(*name*, *table={33: 45, 34: 45, 35: 45, 36: 45, 37: 45, 38: 45, 39: 45, 40: 45, 41: 45, 42: 45, 43: 45, 44: 45, 45: 45, 46: 45, 47: 45, 58: 45, 59: 45, 60: 45, 61: 45, 62: 45, 63: 45, 64: 45, 91: 45, 92: 45, 93: 45, 94: 45, 95: 45, 96: 45, 123: 45, 124: 45, 125: 45, 126: 45}*)
Format AMQP queue name into a valid Azure Storage Queue name.

no_ack = True

queue_name_prefix

property queue_service

property transport_options

4.33 Azure Service Bus Transport - kombu.transport.azureservicebus

Azure Service Bus Message Queue transport.

The transport can be enabled by setting the CELERY_BROKER_URL to:

` azureservicebus://{SAS policy name}:{SAS key}@{Service Bus Namespace} `

Note that the Shared Access Policy used to connect to Azure Service Bus requires Manage, Send and Listen claims since the broker will create new queues and delete old queues as required.

Note that if the SAS key for the Service Bus account contains a slash, it will have to be regenerated before it can be used in the connection URL.

More information about Azure Service Bus: https://azure.microsoft.com/en-us/services/service-bus/

4.33.1 Transport

class kombu.transport.azureservicebus.**Transport**(*client*, ***kwargs*)
Azure Service Bus transport.

class Channel(**args*, ***kwargs*)
Azure Service Bus channel.

property conninfo

default_peek_lock = False

default_visibility_timeout = 1800

default_wait_time_seconds = 5

domain_format = 'kombu%(vhost)s'

entity_name(*name*, *table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)

Format AMQP queue name into a valid ServiceBus queue name.

peek_lock

queue_name_prefix

property queue_service

property transport_options

visibility_timeout

wait_time_seconds

default_port = None

polling_interval = 1

4.33.2 Channel

class kombu.transport.azureservicebus.**Channel**(**args*, ***kwargs*)

Azure Service Bus channel.

property conninfo

default_peek_lock = False

default_visibility_timeout = 1800

default_wait_time_seconds = 5

domain_format = 'kombu%(vhost)s'

entity_name(*name*, *table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)

Format AMQP queue name into a valid ServiceBus queue name.

peek_lock

queue_name_prefix

property queue_service

property transport_options

visibility_timeout

wait_time_seconds

4.34 Pure-python AMQP Transport - kombu.transport.pyamqp

Pure-Python amqp transport.

4.34.1 Transport

class kombu.transport.pyamqp.**Transport**(*client*, *default_port=None*, *default_ssl_port=None*, ***kwargs*)

AMQP Transport.

class Connection(*host='localhost:5672'*, *userid='guest'*, *password='guest'*, *login_method=None*, *login_response=None*, *authentication=()*, *virtual_host='/'*, *locale='en_US'*, *client_properties=None*, *ssl=False*, *connect_timeout=None*, *channel_max=None*, *frame_max=None*, *heartbeat=0*, *on_open=None*, *on_blocked=None*, *on_unblocked=None*, *confirm_publish=False*, *on_tune_ok=None*, *read_timeout=None*, *write_timeout=None*, *socket_settings=None*, *frame_handler=<function frame_handler>*, *frame_writer=<function frame_writer>*, ***kwargs*)

AMQP Connection.

class Channel(*connection*, *channel_id=None*, *auto_decode=True*, *on_open=None*)

AMQP Channel.

class Message(*msg*, *channel=None*, ***kwargs*)

AMQP Message.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

message_to_python(*raw_message*)

Convert encoded message body back to a Python value.

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*, *_Message=<class 'amqp.basic_message.Message'>*)

Prepare message so that it can be sent using this transport.

prepare_queue_arguments(*arguments*, ***kwargs*)

channel_errors = (<class 'amqp.exceptions.ChannelError'>,)

close_connection(*connection*)
Close the AMQP broker connection.

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'OSError'>, <cl

create_channel(*connection*)

property default_connection_params

default_port = 5672

default_ssl_port = 5671

drain_events(*connection*, ***kwargs*)

driver_name = 'py-amqp'

driver_type = 'amqp'

driver_version()

establish_connection()
Establish connection to the AMQP broker.

get_heartbeat_interval(*connection*)

get_manager(**args*, ***kwargs*)

heartbeat_check(*connection*, *rate=2*)

implements = {'asynchronous': True, 'exchange_type': frozenset({'direct', 'fanout',

qos_semantics_matches_spec(*connection*)

recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)

recoverable_connection_errors = (<class 'amqp.exceptions.RecoverableConnectionError'>,

register_with_event_loop(*connection*, *loop*)

verify_connection(*connection*)

4.34.2 Connection

class kombu.transport.pyamqp.**Connection**(*host='localhost:5672'*, *userid='guest'*, *password='guest'*, *login_method=None*, *login_response=None*, *authentication=()*, *virtual_host='/'*, *locale='en_US'*, *client_properties=None*, *ssl=False*, *connect_timeout=None*, *channel_max=None*, *frame_max=None*, *heartbeat=0*, *on_open=None*, *on_blocked=None*, *on_unblocked=None*, *confirm_publish=False*, *on_tune_ok=None*, *read_timeout=None*, *write_timeout=None*, *socket_settings=None*, *frame_handler=<function frame_handler>*, *frame_writer=<function frame_writer>*, ***kwargs*)

AMQP Connection.

class Channel(*connection*, *channel_id=None*, *auto_decode=True*, *on_open=None*)
AMQP Channel.

Consumer(**args*, ***kwargs*)

class Message(*msg*, *channel=None*, ***kwargs*)
AMQP Message.

exception MessageStateError
The message has already been acknowledged.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

accept

ack(*multiple=False*)
Acknowledge this message as being processed.

This will remove the message from the queue.
Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

ack_log_error(*logger*, *errors*, *multiple=False*)

property acknowledged
Set to true if the message has been acknowledged.

body

channel

content_encoding

content_type

decode()
Deserialize the message body.

Returning the original python structure sent by the publisher.

Note: The return value is memoized, use *_decode* to force re-evaluation.

---

delivery_info

delivery_tag

errors = None

headers

property payload
The decoded message body.

properties

reject(*requeue=False*)
Reject this message.

The message will be discarded by the server.
Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

reject_log_error(*logger*, *errors*, *requeue=False*)

requeue()
Reject this message and put it back on the queue.

> **Warning:** You must not use this method as a means of selecting messages to process.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

Producer(**args*, ***kwargs*)

after_reply_message_received(*queue*)
Callback called after RPC reply received.

Notes

Reply queue semantics: can be used to delete the queue after transient reply message received.

basic_ack(*delivery_tag*, *multiple=False*, *argsig='Lb'*)
Acknowledge one or more messages.

This method acknowledges one or more messages delivered via the Deliver or Get-Ok methods. The client can ask to confirm a single message or a set of messages up to and including a specific message.

Parameters

- **delivery_tag** – longlong

  server-assigned delivery tag

  The server-assigned and channel-specific delivery tag

  RULE:
  The delivery tag is valid only within the channel from which the message was received. I.e. a client MUST NOT receive a message on one channel and then acknowledge it on another.

  RULE:
  The server MUST NOT use a zero value for delivery tags. Zero is reserved for client use, meaning "all messages so far received".

- **multiple** – boolean

  acknowledge multiple messages

  If set to True, the delivery tag is treated as "up to and including", so that the client can acknowledge multiple messages with a single method. If set to False, the delivery tag refers to a single message. If the multiple field is True, and the delivery tag is zero, tells the server to acknowledge all outstanding mesages.

  RULE:
  The server MUST validate that a non-zero delivery- tag refers to an delivered message, and raise a channel exception if this is not the case.

basic_cancel(*consumer_tag*, *nowait=False*, *argsig='sb'*)
End a queue consumer.

This method cancels a consumer. This does not affect already delivered messages, but it does mean the server will not send any more messages for that consumer. The client may receive an abitrary number of messages in between sending the cancel method and receiving the cancel-ok reply.

RULE:

If the queue no longer exists when the client sends a cancel command, or the consumer has been cancelled for other reasons, this command has no effect.

Parameters

- **consumer_tag** – shortstr

  consumer tag

  Identifier for the consumer, valid within the current connection.

  RULE:

  The consumer tag is valid only within the channel from which the consumer was created. I.e. a client MUST NOT create a consumer in one channel and then use it in another.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

basic_consume(*queue=''*, *consumer_tag=''*, *no_local=False*, *no_ack=False*, *exclusive=False*, *nowait=False*, *callback=None*, *arguments=None*, *on_cancel=None*, *argsig='BssbbbbF'*)

Start a queue consumer.

This method asks the server to start a "consumer", which is a transient request for messages from a specific queue. Consumers last as long as the channel they were created on, or until the client cancels them.

RULE:

The server SHOULD support at least 16 consumers per queue, unless the queue was declared as private, and ideally, impose no limit except as defined by available resources.

Parameters

- **queue** – shortstr

  Specifies the name of the queue to consume from. If the queue name is null, refers to the current queue for the channel, which is the last declared queue.

  RULE:

  If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

- **consumer_tag** – shortstr

  Specifies the identifier for the consumer. The consumer tag is local to a connection, so two clients can use the same consumer tags. If this field is empty the server will generate a unique tag.

  RULE:

  The tag MUST NOT refer to an existing consumer. If the client attempts to create two consumers with the same non-empty tag the server MUST raise a connection exception with reply code 530 (not allowed).

- **no_local** – boolean

  do not deliver own messages

  If the no-local field is set the server will not send messages to the client that published them.

- **no_ack** – boolean

  no acknowledgment needed

  If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.

- **exclusive** – boolean

  request exclusive access

  Request exclusive consumer access, meaning only this consumer can access the queue.

  RULE:

  If the server cannot grant exclusive access to the queue when asked, - because there are other consumers active - it MUST raise a channel exception with return code 403 (access refused).

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

- **callback** – Python callable

  function/method called with each delivered message

  For each message delivered by the broker, the callable will be called with a Message object as the single argument. If no callable is specified, messages are quietly discarded, no_ack should probably be set to True in that case.

basic_get(*queue=''*, *no_ack=False*, *argsig='Bsb'*)

Direct access to a queue.

This method provides a direct access to the messages in a queue using a synchronous dialogue that is designed for specific types of application where synchronous functionality is more important than performance.

Parameters

- **queue** – shortstr

  Specifies the name of the queue to consume from. If the queue name is null, refers to the current queue for the channel, which is the last declared queue.

  RULE:

  If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

- **no_ack** – boolean

  no acknowledgment needed

  If this field is set the server does not expect acknowledgments for messages. That is, when a message is delivered to the client the server automatically and silently acknowledges it on behalf of the client. This functionality increases performance but at the cost of reliability. Messages can get lost if a client dies before it can deliver them to the application.

Non-blocking, returns a amqp.basic_message.Message object, or None if queue is empty.

basic_publish(*msg*, *exchange=''*, *routing_key=''*, *mandatory=False*, *immediate=False*, *timeout=None*, *argsig='Bssbb'*)

Publish a message.

This method publishes a message to a specific exchange. The message will be routed to queues as defined by the exchange configuration and distributed to any active consumers when the transaction, if any, is committed.

When channel is in confirm mode (when Connection parameter confirm_publish is set to True), each message is confirmed. When broker rejects published message (e.g. due internal broker constrains), MessageNacked exception is raised.

Parameters

- **exchange** – shortstr

  Specifies the name of the exchange to publish to. The exchange name can be empty, meaning the default exchange. If the exchange name is specified, and that exchange does not exist, the server will raise a channel exception.

  RULE:

  The server MUST accept a blank exchange name to mean the default exchange.

  RULE:

  The exchange MAY refuse basic content in which case it MUST raise a channel exception with reply code 540 (not implemented).

- **routing_key** – shortstr

  Message routing key

  Specifies the routing key for the message. The routing key is used for routing messages depending on the exchange configuration.

- **mandatory** – boolean

  indicate mandatory routing

  This flag tells the server how to react if the message cannot be routed to a queue. If this flag is True, the server will return an unroutable message with a Return method. If this flag is False, the server silently drops the message.

  RULE:

  The server SHOULD implement the mandatory flag.

- **immediate** – boolean

  request immediate delivery

  This flag tells the server how to react if the message cannot be routed to a queue consumer immediately. If this flag is set, the server will return an undeliverable

message with a Return method. If this flag is zero, the server will queue the message, but with no guarantee that it will ever be consumed.

RULE:

The server SHOULD implement the immediate flag.

basic_publish_confirm(**args*, ***kwargs*)

basic_qos(*prefetch_size*, *prefetch_count*, *a_global*, *argsig='lBb'*)

Specify quality of service.

This method requests a specific quality of service. The QoS can be specified for the current channel or for all channels on the connection. The particular properties and semantics of a qos method always depend on the content class semantics. Though the qos method could in principle apply to both peers, it is currently meaningful only for the server.

Parameters

- **prefetch_size** – long

  prefetch window in octets

  The client can request that messages be sent in advance so that when the client finishes processing a message, the following message is already held locally, rather than needing to be sent down the channel. Prefetching gives a performance improvement. This field specifies the prefetch window size in octets. The server will send a message in advance if it is equal to or smaller in size than the available prefetch size (and also falls into other prefetch limits). May be set to zero, meaning "no specific limit", although other prefetch limits may still apply. The prefetch-size is ignored if the no-ack option is set.

  RULE:

  The server MUST ignore this setting when the client is not processing any messages - i.e. the prefetch size does not limit the transfer of single messages to a client, only the sending in advance of more messages while the client still has one or more unacknowledged messages.

- **prefetch_count** – short

  prefetch window in messages

  Specifies a prefetch window in terms of whole messages. This field may be used in combination with the prefetch-size field; a message will only be sent in advance if both prefetch windows (and those at the channel and connection level) allow it. The prefetch- count is ignored if the no-ack option is set.

  RULE:

  The server MAY send less data in advance than allowed by the client's specified prefetch windows but it MUST NOT send more.

- **a_global** – boolean

  apply to entire connection

  By default the QoS settings apply to the current channel only. If this field is set, they are applied to the entire connection.

basic_recover(*requeue=False*)

Redeliver unacknowledged messages.

This method asks the broker to redeliver all unacknowledged messages on a specified channel. Zero or more messages may be redelivered. This method is only allowed on non-transacted channels.

RULE:

The server MUST set the redelivered flag on all messages that are resent.

RULE:

The server MUST raise a channel exception if this is called on a transacted channel.

Parameters **`requeue`** – boolean

requeue the message

If this field is False, the message will be redelivered to the original recipient. If this field is True, the server will attempt to requeue the message, potentially then delivering it to an alternative subscriber.

`basic_recover_async`(*requeue=False*)

`basic_reject`(*delivery_tag*, *requeue*, *argsig='Lb'*)

Reject an incoming message.

This method allows a client to reject a message. It can be used to interrupt and cancel large incoming messages, or return untreatable messages to their original queue.

RULE:

The server SHOULD be capable of accepting and process the Reject method while sending message content with a Deliver or Get-Ok method. I.e. the server should read and process incoming methods while sending output frames. To cancel a partially-send content, the server sends a content body frame of size 1 (i.e. with no data except the frame-end octet).

RULE:

The server SHOULD interpret this method as meaning that the client is unable to process the message at this time.

RULE:

A client MUST NOT use this method as a means of selecting messages to process. A rejected message MAY be discarded or dead-lettered, not necessarily passed to another client.

Parameters

- **`delivery_tag`** – longlong

  server-assigned delivery tag

  The server-assigned and channel-specific delivery tag

  RULE:

  The delivery tag is valid only within the channel from which the message was received. I.e. a client MUST NOT receive a message on one channel and then acknowledge it on another.

  RULE:

  The server MUST NOT use a zero value for delivery tags. Zero is reserved for client use, meaning "all messages so far received".

- **requeue** – boolean

  requeue the message

  If this field is False, the message will be discarded. If this field is True, the server will attempt to requeue the message.

  RULE:

  The server MUST NOT deliver the message to the same client within the context of the current channel. The recommended strategy is to attempt to deliver the message to an alternative consumer, and if that is not possible, to move the message to a dead-letter queue. The server MAY use more sophisticated tracking to hold the message on the queue and redeliver it to the same client at a later stage.

close(*reply_code=0*, *reply_text=''*, *method_sig=(0, 0)*, *argsig='BsBB'*)

Request a channel close.

This method indicates that the sender wants to close the channel. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class and method id of the method which caused the exception.

RULE:

After sending this method any received method except Channel.Close-OK MUST be discarded.

RULE:

The peer sending this method MAY use a counter or timeout to detect failure of the other peer to respond correctly with Channel.Close-OK..

Parameters

- **reply_code** – short

  The reply code. The AMQ reply codes are defined in AMQ RFC 011.

- **reply_text** – shortstr

  The localised reply text. This text can be logged as an aid to resolving issues.

- **class_id** – short

  failing method class

  When the close is provoked by a method exception, this is the class of the method.

- **method_id** – short

  failing method ID

  When the close is provoked by a method exception, this is the ID of the method.

collect()

Tear down this object.

Best called after we've agreed to close with the server.

confirm_select(*nowait=False*)

Enable publisher confirms for this channel.

Note: This is an RabbitMQ extension.

Can now be used if the channel is in transactional mode.

Parameters **nowait** – If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

dispatch_method(*method_sig*, *payload*, *content*)

exchange_bind(*destination*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)

Bind an exchange to an exchange.

RULE:

A server MUST allow and ignore duplicate bindings - that is, two or more bind methods for a specific exchanges, with identical arguments - without treating these as an error.

RULE:

A server MUST allow cycles of exchange bindings to be created including allowing an exchange to be bound to itself.

RULE:

A server MUST not deliver the same message more than once to a destination exchange, even if the topology of exchanges and bindings results in multiple (even infinite) routes to that exchange.

Parameters

- **reserved-1** – short
- **destination** – shortstr

  Specifies the name of the destination exchange to bind.

  RULE:

  A client MUST NOT be allowed to bind a non- existent destination exchange.

  RULE:

  The server MUST accept a blank exchange name to mean the default exchange.

- **source** – shortstr

  Specifies the name of the source exchange to bind.

  RULE:

  A client MUST NOT be allowed to bind a non- existent source exchange.

  RULE:

  The server MUST accept a blank exchange name to mean the default exchange.

- **routing-key** – shortstr

  Specifies the routing key for the binding. The routing key is used for routing messages depending on the exchange configuration. Not all exchanges use a routing key - refer to the specific exchange documentation.

- **no-wait** – bit
- **arguments** – table

  A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

exchange_declare(*exchange*, *type*, *passive=False*, *durable=False*, *auto_delete=True*, *nowait=False*, *arguments=None*, *argsig='BssbbbbbF'*)

Declare exchange, create if needed.

This method creates an exchange if it does not already exist, and if the exchange exists, verifies that it is of the correct and expected class.

RULE:

The server SHOULD support a minimum of 16 exchanges per virtual host and ideally, impose no limit except as defined by available resources.

Parameters

- **exchange** – shortstr

  RULE:

  Exchange names starting with "amq." are reserved for predeclared and standardised exchanges. If the client attempts to create an exchange starting with "amq.", the server MUST raise a channel exception with reply code 403 (access refused).

- **type** – shortstr

  exchange type

  Each exchange belongs to one of a set of exchange types implemented by the server. The exchange types define the functionality of the exchange - i.e. how messages are routed through it. It is not valid or meaningful to attempt to change the type of an existing exchange.

  RULE:

  If the exchange already exists with a different type, the server MUST raise a connection exception with a reply code 507 (not allowed).

  RULE:

  If the server does not support the requested exchange type it MUST raise a connection exception with a reply code 503 (command invalid).

- **passive** – boolean

  do not create exchange

  If set, the server will not create the exchange. The client can use this to check whether an exchange exists without modifying the server state.

  RULE:

  If set, and the exchange does not already exist, the server MUST raise a channel exception with reply code 404 (not found).

- **durable** – boolean

  request a durable exchange

  If set when creating a new exchange, the exchange will be marked as durable. Durable exchanges remain active when a server restarts. Non-durable exchanges (transient exchanges) are purged if/when a server restarts.

  RULE:

  The server MUST support both durable and transient exchanges.

RULE:

The server MUST ignore the durable field if the exchange already exists.

- **auto_delete** – boolean

  auto-delete when unused

  If set, the exchange is deleted when all queues have finished using it.

  RULE:

  The server SHOULD allow for a reasonable delay between the point when it determines that an exchange is not being used (or no longer used), and the point when it deletes the exchange. At the least it must allow a client to create an exchange and then bind a queue to it, with a small but non-zero delay between these two actions.

  RULE:

  The server MUST ignore the auto-delete field if the exchange already exists.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

- **arguments** – table

  arguments for declaration

  A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation. This field is ignored if passive is True.

exchange_delete(*exchange*, *if_unused=False*, *nowait=False*, *argsig='Bsbb'*)

Delete an exchange.

This method deletes an exchange. When an exchange is deleted all queue bindings on the exchange are cancelled.

Parameters

- **exchange** – shortstr

  RULE:

  The exchange MUST exist. Attempting to delete a non-existing exchange causes a channel exception.

- **if_unused** – boolean

  delete only if unused

  If set, the server will only delete the exchange if it has no queue bindings. If the exchange has queue bindings the server does not delete it but raises a channel exception instead.

  RULE:

  If set, the server SHOULD delete the exchange but only if it has no queue bindings.

  RULE:

If set, the server SHOULD raise a channel exception if the exchange is in use.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

exchange_unbind(*destination*, *source=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)

Unbind an exchange from an exchange.

RULE:

If a unbind fails, the server MUST raise a connection exception.

Parameters

- **reserved-1** – short
- **destination** – shortstr

  Specifies the name of the destination exchange to unbind.

  RULE:

  The client MUST NOT attempt to unbind an exchange that does not exist from an exchange.

  RULE:

  The server MUST accept a blank exchange name to mean the default exchange.

- **source** – shortstr

  Specifies the name of the source exchange to unbind.

  RULE:

  The client MUST NOT attempt to unbind an exchange from an exchange that does not exist.

  RULE:

  The server MUST accept a blank exchange name to mean the default exchange.

- **routing-key** – shortstr

  Specifies the routing key of the binding to unbind.

- **no-wait** – bit
- **arguments** – table

  Specifies the arguments of the binding to unbind.

flow(*active*)

Enable/disable flow from peer.

This method asks the peer to pause or restart the flow of content data. This is a simple flow-control mechanism that a peer can use to avoid oveflowing its queues or otherwise finding itself receiving more messages than it can process. Note that this method is not intended for window control. The peer that receives a request to stop sending content should finish sending the current content, if any, and then wait until it receives a Flow restart method.

RULE:

When a new channel is opened, it is active. Some applications assume that channels are inactive until started. To emulate this behaviour a client MAY open the channel, then pause it.

RULE:

When sending content data in multiple frames, a peer SHOULD monitor the channel for incoming methods and respond to a Channel.Flow as rapidly as possible.

RULE:

A peer MAY use the Channel.Flow method to throttle incoming content data for internal reasons, for example, when exchangeing data over a slower connection.

RULE:

The peer that requests a Channel.Flow method MAY disconnect and/or ban a peer that does not respect the request.

Parameters **active** – boolean

start/stop content frames

If True, the peer starts sending content frames. If False, the peer stops sending content frames.

get_bindings()

message_to_python(*raw_message*)
Convert encoded message body back to a Python value.

no_ack_consumers = None

open()
Open a channel for use.

This method opens a virtual connection (a channel).

RULE:

This method MUST NOT be called when the channel is already open.

Parameters **out_of_band** – shortstr (DEPRECATED)

out-of-band settings

Configures out-of-band transfers on this channel. The syntax and meaning of this field will be formally defined at a later date.

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*, *_Message=<class 'amqp.basic_message.Message'>*)
Prepare message so that it can be sent using this transport.

prepare_queue_arguments(*arguments*, ***kwargs*)

queue_bind(*queue*, *exchange=''*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssbF'*)
Bind queue to an exchange.

This method binds a queue to an exchange. Until a queue is bound it will not receive any messages. In a classic messaging model, store-and-forward queues are bound to a dest exchange and subscription queues are bound to a dest_wild exchange.

RULE:

A server MUST allow ignore duplicate bindings - that is, two or more bind methods for a specific queue, with identical arguments - without treating these as an error.

RULE:

If a bind fails, the server MUST raise a connection exception.

RULE:

The server MUST NOT allow a durable queue to bind to a transient exchange. If the client attempts this the server MUST raise a channel exception.

RULE:

Bindings for durable queues are automatically durable and the server SHOULD restore such bindings after a server restart.

RULE:

The server SHOULD support at least 4 bindings per queue, and ideally, impose no limit except as defined by available resources.

Parameters

- **queue** – shortstr

  Specifies the name of the queue to bind. If the queue name is empty, refers to the current queue for the channel, which is the last declared queue.

  RULE:

  If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

  RULE:

  If the queue does not exist the server MUST raise a channel exception with reply code 404 (not found).

- **exchange** – shortstr

  The name of the exchange to bind to.

  RULE:

  If the exchange does not exist the server MUST raise a channel exception with reply code 404 (not found).

- **routing_key** – shortstr

  message routing key

  Specifies the routing key for the binding. The routing key is used for routing messages depending on the exchange configuration. Not all exchanges use a routing key - refer to the specific exchange documentation. If the routing key is empty and the queue name is empty, the routing key will be the current queue for the channel, which is the last declared queue.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

- **arguments** – table

  arguments for binding

  A set of arguments for the binding. The syntax and semantics of these arguments depends on the exchange class.

queue_declare(*queue=''*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*, *argsig='BsbbbbbF'*)

Declare queue, create if needed.

This method creates or checks a queue. When creating a new queue the client can specify various properties that control the durability of the queue and its contents, and the level of sharing for the queue.

RULE:

The server MUST create a default binding for a newly- created queue to the default exchange, which is an exchange of type 'direct'.

RULE:

The server SHOULD support a minimum of 256 queues per virtual host and ideally, impose no limit except as defined by available resources.

Parameters

- **queue** – shortstr

  RULE:

  The queue name MAY be empty, in which case the server MUST create a new queue with a unique generated name and return this to the client in the Declare-Ok method.

  RULE:

  Queue names starting with "amq." are reserved for predeclared and standardised server queues. If the queue name starts with "amq." and the passive option is False, the server MUST raise a connection exception with reply code 403 (access refused).

- **passive** – boolean

  do not create queue

  If set, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state.

  RULE:

  If set, and the queue does not already exist, the server MUST respond with a reply code 404 (not found) and raise a channel exception.

- **durable** – boolean

  request a durable queue

If set when creating a new queue, the queue will be marked as durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue.

RULE:

The server MUST recreate the durable queue after a restart.

RULE:

The server MUST support both durable and transient queues.

RULE:

The server MUST ignore the durable field if the queue already exists.

- **exclusive** – boolean

  request an exclusive queue

  Exclusive queues may only be consumed from by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'.

  RULE:

  The server MUST support both exclusive (private) and non-exclusive (shared) queues.

  RULE:

  The server MUST raise a channel exception if 'exclusive' is specified and the queue already exists and is owned by a different connection.

- **auto_delete** – boolean

  auto-delete queue when unused

  If set, the queue is deleted when all consumers have finished using it. Last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted.

  RULE:

  The server SHOULD allow for a reasonable delay between the point when it determines that a queue is not being used (or no longer used), and the point when it deletes the queue. At the least it must allow a client to create a queue and then create a consumer to read from it, with a small but non-zero delay between these two actions. The server should equally allow for clients that may be disconnected prematurely, and wish to re- consume from the same queue without losing messages. We would recommend a configurable timeout, with a suitable default value being one minute.

  RULE:

  The server MUST ignore the auto-delete field if the queue already exists.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

- **arguments** – table

  arguments for declaration

  A set of arguments for the declaration. The syntax and semantics of these arguments depends on the server implementation. This field is ignored if passive is True.

Returns a tuple containing 3 items: the name of the queue (essential for automatically-named queues), message count and consumer count

queue_delete(*queue=''*, *if_unused=False*, *if_empty=False*, *nowait=False*, *argsig='Bsbbb'*)
Delete a queue.

This method deletes a queue. When a queue is deleted any pending messages are sent to a dead-letter queue if this is defined in the server configuration, and all consumers on the queue are cancelled.

RULE:

The server SHOULD use a dead-letter queue to hold messages that were pending on a deleted queue, and MAY provide facilities for a system administrator to move these messages back to an active queue.

Parameters

- **queue** – shortstr

  Specifies the name of the queue to delete. If the queue name is empty, refers to the current queue for the channel, which is the last declared queue.

  RULE:

  If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

  RULE:

  The queue must exist. Attempting to delete a non- existing queue causes a channel exception.

- **if_unused** – boolean

  delete only if unused

  If set, the server will only delete the queue if it has no consumers. If the queue has consumers the server does does not delete it but raises a channel exception instead.

  RULE:

  The server MUST respect the if-unused flag when deleting a queue.

- **if_empty** – boolean

  delete only if empty

  If set, the server will only delete the queue if it has no messages. If the queue is not empty the server raises a channel exception.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

If nowait is False, returns the number of deleted messages.

queue_purge(*queue=''*, *nowait=False*, *argsig='Bsb'*)
Purge a queue.

This method removes all messages from a queue. It does not cancel consumers. Purged messages are deleted without any formal "undo" mechanism.

RULE:

A call to purge MUST result in an empty queue.

RULE:

On transacted channels the server MUST not purge messages that have already been sent to a client but not yet acknowledged.

RULE:

The server MAY implement a purge queue or log that allows system administrators to recover accidentally-purged messages. The server SHOULD NOT keep purged messages in the same storage spaces as the live messages since the volumes of purged messages may get very large.

Parameters

- **queue** – shortstr

  Specifies the name of the queue to purge. If the queue name is empty, refers to the current queue for the channel, which is the last declared queue.

  RULE:

  If the client did not previously declare a queue, and the queue name in this method is empty, the server MUST raise a connection exception with reply code 530 (not allowed).

  RULE:

  The queue must exist. Attempting to purge a non- existing queue causes a channel exception.

- **nowait** – boolean

  do not send a reply method

  If set, the server will not respond to the method. The client should not wait for a reply method. If the server could not complete the method it will raise a channel or connection exception.

If nowait is False, returns a number of purged messages.

queue_unbind(*queue*, *exchange*, *routing_key=''*, *nowait=False*, *arguments=None*, *argsig='BsssF'*)
Unbind a queue from an exchange.

This method unbinds a queue from an exchange.

RULE:

If a unbind fails, the server MUST raise a connection exception.

Parameters

- **queue** – shortstr

  Specifies the name of the queue to unbind.

  RULE:

  The client MUST either specify a queue name or have previously declared a queue on the same channel

  RULE:

  The client MUST NOT attempt to unbind a queue that does not exist.

- **exchange** – shortstr

  The name of the exchange to unbind from.

  RULE:

  The client MUST NOT attempt to unbind a queue from an exchange that does not exist.

  RULE:

  The server MUST accept a blank exchange name to mean the default exchange.

- **routing_key** – shortstr

  routing key of binding

  Specifies the routing key of the binding to unbind.

- **arguments** – table

  arguments of binding

  Specifies the arguments of the binding to unbind.

send_method(*sig*, *format=None*, *args=None*, *content=None*, *wait=None*, *callback=None*, *returns_tuple=False*)

then(*on_success*, *on_error=None*)

tx_commit()

Commit the current transaction.

This method commits all messages published and acknowledged in the current transaction. A new transaction starts immediately after a commit.

tx_rollback()

Abandon the current transaction.

This method abandons all messages published and acknowledged in the current transaction. A new transaction starts immediately after a rollback.

tx_select()

Select standard transaction mode.

This method sets the channel to use standard transactions. The client must use this method at least once on a channel before using the Commit or Rollback methods.

wait(*method*, *callback=None*, *timeout=None*, *returns_tuple=False*)

Transport(*host*, *connect_timeout*, *ssl=False*, *read_timeout=None*, *write_timeout=None*, *socket_settings=None*, ***kwargs*)

blocking_read(*timeout=None*)

bytes_recv = 0

bytes_sent = 0

channel(*channel_id=None*, *callback=None*)
Create new channel.

Fetch a Channel object identified by the numeric channel_id, or create that object if it doesn't already exist.

channel_errors = (<class 'amqp.exceptions.ChannelError'>,)

client_heartbeat = None

close(*reply_code=0*, *reply_text=''*, *method_sig=(0, 0)*, *argsig='BsBB'*)
Request a connection close.

This method indicates that the sender wants to close the connection. This may be due to internal conditions (e.g. a forced shut-down) or due to an error handling a specific method, i.e. an exception. When a close is due to an exception, the sender provides the class and method id of the method which caused the exception.

RULE:

After sending this method any received method except the Close-OK method MUST be discarded.

RULE:

The peer sending this method MAY use a counter or timeout to detect failure of the other peer to respond correctly with the Close-OK method.

RULE:

When a server receives the Close method from a client it MUST delete all server-side resources associated with the client's context. A client CANNOT reconnect to a context after sending or receiving a Close method.

Parameters

- **reply_code** – short

  The reply code. The AMQ reply codes are defined in AMQ RFC 011.

- **reply_text** – shortstr

  The localised reply text. This text can be logged as an aid to resolving issues.

- **class_id** – short

  failing method class

  When the close is provoked by a method exception, this is the class of the method.

- **method_id** – short

  failing method ID

  When the close is provoked by a method exception, this is the ID of the method.

collect()

connect(*callback=None*)

property connected

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'OSError'>, <cl

dispatch_method(*method_sig*, *payload*, *content*)

drain_events(*timeout=None*)

property frame_writer

heartbeat = None

heartbeat_tick(*rate=2*)
Send heartbeat packets if necessary.

Raises **ConnectionForvced** – if none have been received recently.

Note: This should be called frequently, on the order of once per second.

Keyword Arguments **rate** (*int*) – Previously used, but ignored now.

is_alive()

last_heartbeat_received = 0

last_heartbeat_sent = 0

library_properties = {'product': 'py-amqp', 'product_version': '2.5.2'}

negotiate_capabilities = {'authentication_failure_close': True, 'connection.blocked':

property on_inbound_frame

on_inbound_method(*channel_id*, *method_sig*, *payload*, *content*)

prev_recv = None

prev_sent = None

recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)

recoverable_connection_errors = (<class 'amqp.exceptions.RecoverableConnectionError'>,

send_heartbeat()

send_method(*sig*, *format=None*, *args=None*, *content=None*, *wait=None*, *callback=None*, *returns_tuple=False*)

property server_capabilities

server_heartbeat = None

property sock

then(*on_success*, *on_error=None*)

property transport

wait(*method*, *callback=None*, *timeout=None*, *returns_tuple=False*)

4.34.3 Channel

class kombu.transport.pyamqp.**Channel**(*connection*, *channel_id=None*, *auto_decode=True*, *on_open=None*)
AMQP Channel.

class Message(*msg*, *channel=None*, ***kwargs*)
AMQP Message.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

message_to_python(*raw_message*)
Convert encoded message body back to a Python value.

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*, *_Message=<class 'amqp.basic_message.Message'>*)
Prepare message so that it can be sent using this transport.

prepare_queue_arguments(*arguments*, ***kwargs*)

4.34.4 Message

class kombu.transport.pyamqp.**Message**(*msg*, *channel=None*, ***kwargs*)
AMQP Message.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

4.35 librabbitmq AMQP transport - kombu.transport.librabbitmq

4.36 Apache QPid Transport - kombu.transport.qpid

Qpid Transport.

Qpid transport using qpid-python as the client and qpid-tools for broker management.

The use this transport you must install the necessary dependencies. These dependencies are available via PyPI and can be installed using the pip command:

```
$ pip install kombu[qpid]
```

or to install the requirements manually:

```
$ pip install qpid-tools qpid-python
```

Python 3 and PyPy Limitations

The Qpid transport does not support Python 3 or PyPy environments due to underlying dependencies not being compatible. This version is tested and works with with Python 2.7.

4.36.1 Authentication

This transport supports SASL authentication with the Qpid broker. Normally, SASL mechanisms are negotiated from a client list and a server list of possible mechanisms, but in practice, different SASL client libraries give different behaviors. These different behaviors cause the expected SASL mechanism to not be selected in many cases. As such, this transport restricts the mechanism types based on Kombu's configuration according to the following table.

Broker String	SASL Mechanism
qpid://hostname/	ANONYMOUS
qpid://username:password@hostname/	PLAIN
see instructions below	EXTERNAL

The user can override the above SASL selection behaviors and specify the SASL string using the *login_method* argument to the *Connection* object. The string can be a single SASL mechanism or a space separated list of SASL mechanisms. If you are using Celery with Kombu, this can be accomplished by setting the *BROKER_LOGIN_METHOD* Celery option.

Note: While using SSL, Qpid users may want to override the SASL mechanism to use *EXTERNAL*. In that case, Qpid requires a username to be presented that matches the *CN* of the SSL client certificate. Ensure that the broker string contains the corresponding username. For example, if the client certificate has *CN=asdf* and the client connects to *example.com* on port 5671, the broker string should be:

qpid://asdf@example.com:5671/

4.36.2 Transport Options

The `transport_options` argument to the *Connection* object are passed directly to the `qpid.messaging.endpoints.Connection` as keyword arguments. These options override and replace any other default or specified values. If using Celery, this can be accomplished by setting the *BROKER_TRANSPORT_OPTIONS* Celery option.

- *Transport*
- *Connection*
- *Channel*

- *Message*

Transport

class kombu.transport.qpid.**Transport**(**args*, ***kwargs*)

Kombu native transport for a Qpid broker.

Provide a native transport for Kombu that allows consumers and producers to read and write messages to/from a broker. This Transport is capable of supporting both synchronous and asynchronous reading. All writes are synchronous through the *Channel* objects that support this Transport.

Asynchronous reads are done using a call to *drain_events()*, which synchronously reads messages that were fetched asynchronously, and then handles them through calls to the callback handlers maintained on the *Connection* object.

The Transport also provides methods to establish and close a connection to the broker. This Transport establishes a factory-like pattern that allows for singleton pattern to consolidate all Connections into a single one.

The Transport can create *Channel* objects to communicate with the broker with using the *create_channel()* method.

The Transport identifies recoverable connection errors and recoverable channel errors according to the Kombu 3.0 interface. These exception are listed as tuples and store in the Transport class attribute *recoverable_connection_errors* and *recoverable_channel_errors* respectively. Any exception raised that is not a member of one of these tuples is considered non-recoverable. This allows Kombu support for automatic retry of certain operations to function correctly.

For backwards compatibility to the pre Kombu 3.0 exception interface, the recoverable errors are also listed as *connection_errors* and *channel_errors*.

class Connection(***connection_options*)

Qpid Connection.

Encapsulate a connection object for the *Transport*.

Parameters

- **host** – The host that connections should connect to.
- **port** – The port that connection should connect to.
- **username** – The username that connections should connect with. Optional.
- **password** – The password that connections should connect with. Optional but requires a username.
- **transport** – The transport type that connections should use. Either 'tcp', or 'ssl' are expected as values.
- **timeout** – the timeout used when a Connection connects to the broker.
- **sasl_mechanisms** – The sasl authentication mechanism type to use. refer to SASL documentation for an explanation of valid values.

Note: qpid.messaging has an AuthenticationFailure exception type, but instead raises a ConnectionError with a message that indicates an authentication failure occurred in those situations. ConnectionError is listed as a recoverable error type, so kombu will attempt to retry if a ConnectionError is raised. Retrying the operation without adjusting the credentials is not correct, so this method specifically checks for a ConnectionError that indicates an Authentication Failure occurred. In those situations, the error type is

mutated while preserving the original message and raised so kombu will allow the exception to not be considered recoverable.

A connection object is created by a *Transport* during a call to *establish_connection()*. The *Transport* passes in connection options as keywords that should be used for any connections created. Each *Transport* creates exactly one Connection.

A Connection object maintains a reference to a Connection which can be accessed through a bound getter method named *get_qpid_connection()* method. Each Channel uses a the Connection for each BrokerAgent, and the Transport maintains a session for all senders and receivers.

The Connection object is also responsible for maintaining the dictionary of references to callbacks that should be called when messages are received. These callbacks are saved in _callbacks, and keyed on the queue name associated with the received message. The _callbacks are setup in *Channel.basic_consume()*, removed in *Channel.basic_cancel()*, and called in *Transport.drain_events()*.

The following keys are expected to be passed in as keyword arguments at a minimum:

All keyword arguments are collected into the connection_options dict and passed directly through to qpid.messaging.endpoints.Connection.establish().

class Channel(*connection*, *transport*)

Supports broker configuration and messaging send and receive.

Parameters

- **connection** (kombu.transport.qpid.Connection) – A Connection object that this Channel can reference. Currently only used to access callbacks.
- **transport** (kombu.transport.qpid.Transport) – The Transport this Channel is associated with.

A channel object is designed to have method-parity with a Channel as defined in AMQP 0-10 and earlier, which allows for the following broker actions:

- exchange declare and delete
- queue declare and delete
- queue bind and unbind operations
- queue length and purge operations
- sending/receiving/rejecting messages
- structuring, encoding, and decoding messages
- supports synchronous and asynchronous reads
- reading state about the exchange, queues, and bindings

Channels are designed to all share a single TCP connection with a broker, but provide a level of isolated communication with the broker while benefiting from a shared TCP connection. The Channel is given its *Connection* object by the *Transport* that instantiates the channel.

This channel inherits from StdChannel, which makes this a 'native' channel versus a 'virtual' channel which would inherit from kombu.transports.virtual.

Messages sent using this channel are assigned a delivery_tag. The delivery_tag is generated for a message as they are prepared for sending by *basic_publish()*. The delivery_tag is unique per channel instance. The delivery_tag has no meaningful context in other objects, and is only maintained in the memory of this object, and the underlying *QoS* object that provides support.

Each channel object instantiates exactly one *QoS* object for prefetch limiting, and asynchronous ACKing. The *QoS* object is lazily instantiated through a property method *qos()*. The *QoS* object is a supporting object that should not be accessed directly except by the channel itself.

Synchronous reads on a queue are done using a call to *basic_get()* which uses `_get()` to perform the reading. These methods read immediately and do not accept any form of timeout. *basic_get()* reads synchronously and ACKs messages before returning them. ACKing is done in all cases, because an application that reads messages using qpid.messaging, but does not ACK them will experience a memory leak. The no_ack argument to *basic_get()* does not affect ACKing functionality.

Asynchronous reads on a queue are done by starting a consumer using *basic_consume()*. Each call to *basic_consume()* will cause a `Receiver` to be created on the `Session` started by the :class: *Transport*. The receiver will asynchronously read using qpid.messaging, and prefetch messages before the call to `Transport.basic_drain()` occurs. The prefetch_count value of the *QoS* object is the capacity value of the new receiver. The new receiver capacity must always be at least 1, otherwise none of the receivers will appear to be ready for reading, and will never be read from.

Each call to *basic_consume()* creates a consumer, which is given a consumer tag that is identified by the caller of *basic_consume()*. Already started consumers can be cancelled using by their consumer_tag using *basic_cancel()*. Cancellation of a consumer causes the `Receiver` object to be closed.

Asynchronous message ACKing is supported through *basic_ack()*, and is referenced by delivery_tag. The Channel object uses its *QoS* object to perform the message ACKing.

class Message(*payload*, *channel=None*, ***kwargs*)

Message object.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

serializable()

class QoS(*session*, *prefetch_count=1*)

A helper object for message prefetch and ACKing purposes.

Keyword Arguments **prefetch_count** – Initial prefetch count, hard set to 1.

NOTE: prefetch_count is currently hard set to 1, and needs to be improved

This object is instantiated 1-for-1 with a *Channel* instance. QoS allows `prefetch_count` to be set to the number of outstanding messages the corresponding *Channel* should be allowed to prefetch. Setting `prefetch_count` to 0 disables prefetch limits, and the object can hold an arbitrary number of messages.

Messages are added using *append()*, which are held until they are ACKed asynchronously through a call to *ack()*. Messages that are received, but not ACKed will not be delivered by the broker to another consumer until an ACK is received, or the session is closed. Messages

are referred to using delivery_tag, which are unique per *Channel*. Delivery tags are managed outside of this object and are passed in with a message to *append()*. Un-ACKed messages can be looked up from QoS using *get()* and can be rejected and forgotten using *reject()*.

ack(*delivery_tag*)
Acknowledge a message by delivery_tag.

Called asynchronously once the message has been handled and can be forgotten by the broker.

Parameters **delivery_tag** (*uuid.UUID*) – the delivery tag associated with the message to be acknowledged.

append(*message*, *delivery_tag*)
Append message to the list of un-ACKed messages.

Add a message, referenced by the delivery_tag, for ACKing, rejecting, or getting later. Messages are saved into an `collections.OrderedDict` by delivery_tag.

Parameters

- **message** (*qpid.messaging.Message*) – A received message that has not yet been ACKed.
- **delivery_tag** (*uuid.UUID*) – A UUID to refer to this message by upon receipt.

can_consume()
Return True if the *Channel* can consume more messages.

Used to ensure the client adheres to currently active prefetch limits.

Returns True, if this QoS object can accept more messages without violating the prefetch_count. If prefetch_count is 0, can_consume will always return True.

Return type bool

can_consume_max_estimate()
Return the remaining message capacity.

Returns an estimated number of outstanding messages that a *kombu.transport.qpid.Channel* can accept without exceeding `prefetch_count`. If `prefetch_count` is 0, then this method returns 1.

Returns The number of estimated messages that can be fetched without violating the prefetch_count.

Return type int

get(*delivery_tag*)
Get an un-ACKed message by delivery_tag.

If called with an invalid delivery_tag a `KeyError` is raised.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be returned.

Returns An un-ACKed message that is looked up by delivery_tag.

Return type qpid.messaging.Message

reject(*delivery_tag*, *requeue=False*)
Reject a message by delivery_tag.

Explicitly notify the broker that the channel associated with this QoS object is rejecting the message that was previously delivered.

If requeue is False, then the message is not requeued for delivery to another consumer. If requeue is True, then the message is requeued for delivery to another consumer.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

Keyword Arguments **requeue** – If True, the broker will be notified to requeue the message. If False, the broker will be told to drop the message entirely. In both cases, the message will be removed from this object.

basic_ack(*delivery_tag*, *multiple=False*)

Acknowledge a message by delivery_tag.

Acknowledges a message referenced by delivery_tag. Messages can only be ACKed using *basic_ack()* if they were acquired using *basic_consume()*. This is the ACKing portion of the asynchronous read behavior.

Internally, this method uses the *QoS* object, which stores messages and is responsible for the ACKing.

Parameters

- **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be acknowledged.
- **multiple** (*bool*) – not implemented. If set to True an AssertionError is raised.

basic_cancel(*consumer_tag*)

Cancel consumer by consumer tag.

Request the consumer stops reading messages from its queue. The consumer is a `Receiver`, and it is closed using `close()`.

This method also cleans up all lingering references of the consumer.

Parameters **consumer_tag** (*an immutable object*) – The tag which refers to the consumer to be cancelled. Originally specified when the consumer was created as a parameter to *basic_consume()*.

basic_consume(*queue*, *no_ack*, *callback*, *consumer_tag*, ***kwargs*)

Start an asynchronous consumer that reads from a queue.

This method starts a consumer of type `Receiver` using the `Session` created and referenced by the *Transport* that reads messages from a queue specified by name until stopped by a call to *basic_cancel()*.

Messages are available later through a synchronous call to *Transport.drain_events()*, which will drain from the consumer started by this method. *Transport.drain_events()* is synchronous, but the receiving of messages over the network occurs asynchronously, so it should still perform well. *Transport.drain_events()* calls the callback provided here with the Message of type self.Message.

Each consumer is referenced by a consumer_tag, which is provided by the caller of this method.

This method sets up the callback onto the self.connection object in a dict keyed by queue name. *drain_events()* is responsible for calling that callback upon message receipt.

All messages that are received are added to the QoS object to be saved for asynchronous ACKing later after the message has been handled by the caller of *drain_events()*. Messages can be ACKed after being received through a call to *basic_ack()*.

If no_ack is True, The no_ack flag indicates that the receiver of the message will not call *basic_ack()* later. Since the message will not be ACKed later, it is ACKed immediately.

basic_consume() transforms the message object type prior to calling the callback. Initially the message comes in as a `qpid.messaging.Message`. This method unpacks the payload of the `qpid.messaging.Message` and creates a new object of type self.Message.

This method wraps the user delivered callback in a runtime-built function which provides the type transformation from `qpid.messaging.Message` to *Message*, and adds the message to the associated *QoS* object for asynchronous ACKing if necessary.

Parameters

- **queue** (*str*) – The name of the queue to consume messages from
- **no_ack** (*bool*) – If True, then messages will not be saved for ACKing later, but will be ACKed immediately. If False, then messages will be saved for ACKing later with a call to *basic_ack()*.
- **callback** (*a callable object*) – a callable that will be called when messages arrive on the queue.
- **consumer_tag** (*an immutable object*) – a tag to reference the created consumer by. This consumer_tag is needed to cancel the consumer.

basic_get(*queue*, *no_ack=False*, ***kwargs*)

Non-blocking single message get and ACK from a queue by name.

Internally this method uses `_get()` to fetch the message. If an `Empty` exception is raised by `_get()`, this method silences it and returns None. If `_get()` does return a message, that message is ACKed. The no_ack parameter has no effect on ACKing behavior, and all messages are ACKed in all cases. This method never adds fetched Messages to the internal QoS object for asynchronous ACKing.

This method converts the object type of the method as it passes through. Fetching from the broker, `_get()` returns a `qpid.messaging.Message`, but this method takes the payload of the `qpid.messaging.Message` and instantiates a *Message* object with the payload based on the class setting of self.Message.

Parameters **queue** (*str*) – The queue name to fetch a message from.

Keyword Arguments **no_ack** – The no_ack parameter has no effect on the ACK behavior of this method. Un-ACKed messages create a memory leak in qpid.messaging, and need to be ACKed in all cases.

Returns The received message.

Return type *Message*

basic_publish(*message*, *exchange*, *routing_key*, ***kwargs*)

Publish message onto an exchange using a routing key.

Publish a message onto an exchange specified by name using a routing key specified by routing_key. Prepares the message in the following ways before sending:

- encodes the body using *encode_body()*
- **wraps the body as a buffer object, so that** `qpid.messaging.endpoints.Sender` uses a content type that can support arbitrarily large messages.
- sets delivery_tag to a random uuid.UUID
- sets the exchange and routing_key info as delivery_info

Internally uses `_put()` to send the message synchronously. This message is typically called by `kombu.messaging.Producer._publish` as the final step in message publication.

Parameters

- **message** (*dict*) – A dict containing key value pairs with the message data. A valid message dict can be generated using the *prepare_message()* method.
- **exchange** (*str*) – The name of the exchange to submit this message onto.
- **routing_key** (*str*) – The routing key to be used as the message is submitted onto the exchange.

basic_qos(*prefetch_count*, **args*)
Change *QoS* settings for this Channel.

Set the number of un-acknowledged messages this Channel can fetch and hold. The prefetch_value is also used as the capacity for any new `Receiver` objects.

Currently, this value is hard coded to 1.

Parameters **prefetch_count** (*int*) – Not used. This method is hard-coded to 1.

basic_reject(*delivery_tag*, *requeue=False*)
Reject a message by delivery_tag.

Rejects a message that has been received by the Channel, but not yet acknowledged. Messages are referenced by their delivery_tag.

If requeue is False, the rejected message will be dropped by the broker and not delivered to any other consumers. If requeue is True, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

Keyword Arguments **requeue** – If False, the rejected message will be dropped by the broker and not delivered to any other consumers. If True, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

body_encoding = 'base64'

close()
Cancel all associated messages and close the Channel.

This cancels all consumers by calling *basic_cancel()* for each known consumer_tag. It also closes the self._broker sessions. Closing the sessions implicitly causes all outstanding, un-ACKed messages to be considered undelivered by the broker.

codecs = {'base64': <kombu.transport.virtual.base.Base64 object>}

decode_body(*body*, *encoding=None*)
Decode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in self.codecs. self.codecs uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through encode and decode methods.

Parameters **body** (*str*) – The body to be encoded.

Keyword Arguments **encoding** – The encoding type to be used. Must be a supported codec listed in self.codecs.

Returns If encoding is specified, the decoded body is returned. If encoding is not specified, the body is returned unchanged.

Return type str

encode_body(*body*, *encoding=None*)
Encode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in self.codecs. self.codecs uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through encode and decode methods.

Parameters **body** (*str*) – The body to be encoded.

Keyword Arguments **encoding** – The encoding type to be used. Must be a supported codec listed in self.codecs.

Returns If encoding is specified, return a tuple with the first position being the encoded body, and the second position the encoding used. If encoding is not specified, the body is passed through unchanged.

Return type tuple

exchange_declare(*exchange=''*, *type='direct'*, *durable=False*, ***kwargs*)
Create a new exchange.

Create an exchange of a specific type, and optionally have the exchange be durable. If an exchange of the requested name already exists, no action is taken and no exceptions are raised. Durable exchanges will survive a broker restart, non-durable exchanges will not.

Exchanges provide behaviors based on their type. The expected behaviors are those defined in the AMQP 0-10 and prior specifications including 'direct', 'topic', and 'fanout' functionality.

Keyword Arguments

- **type** – The exchange type. Valid values include 'direct', 'topic', and 'fanout'.
- **exchange** – The name of the exchange to be created. If no exchange is specified, then a blank string will be used as the name.
- **durable** – True if the exchange should be durable, or False otherwise.

exchange_delete(*exchange_name*, ***kwargs*)
Delete an exchange specified by name.

Parameters **exchange_name** (*str*) – The name of the exchange to be deleted.

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*)
Prepare message data for sending.

This message is typically called by `kombu.messaging.Producer._publish()` as a preparation step in message publication.

Parameters **body** (*str*) – The body of the message

Keyword Arguments

- **priority** – A number between 0 and 9 that sets the priority of the message.
- **content_type** – The content_type the message body should be treated as. If this is unset, the `qpid.messaging.endpoints.Sender` object tries to autodetect the content_type from the body.
- **content_encoding** – The content_encoding the message body is encoded as.

- **headers** – Additional Message headers that should be set. Passed in as a key-value pair.
- **properties** – Message properties to be set on the message.

Returns Returns a dict object that encapsulates message attributes. See parameters for more details on attributes that can be set.

Return type dict

property qos

QoS manager for this channel.

Lazily instantiates an object of type *QoS* upon access to the self.qos attribute.

Returns An already existing, or newly created QoS object

Return type *QoS*

queue_bind(*queue*, *exchange*, *routing_key*, ***kwargs*)

Bind a queue to an exchange with a bind key.

Bind a queue specified by name, to an exchange specified by name, with a specific bind key. The queue and exchange must already exist on the broker for the bind to complete successfully. Queues may be bound to exchanges multiple times with different keys.

Parameters

- **queue** (*str*) – The name of the queue to be bound.
- **exchange** (*str*) – The name of the exchange that the queue should be bound to.
- **routing_key** (*str*) – The bind key that the specified queue should bind to the specified exchange with.

queue_declare(*queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*)

Create a new queue specified by name.

If the queue already exists, no change is made to the queue, and the return value returns information about the existing queue.

The queue name is required and specified as the first argument.

If passive is True, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state. Default is False.

If durable is True, the queue will be durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. Default is False.

If exclusive is True, the queue will be exclusive. Exclusive queues may only be consumed by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'. Default is False.

If auto_delete is True, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Default is True.

The nowait parameter is unused. It was part of the 0-9-1 protocol, but this AMQP client implements 0-10 which removed the nowait option.

The arguments parameter is a set of arguments for the declaration of the queue. Arguments are passed as a dict or None. This field is ignored if passive is True. Default is None.

This method returns a `namedtuple` with the name 'queue_declare_ok_t' and the queue name as 'queue', message count on the queue as 'message_count', and the number of active consumers as 'consumer_count'. The named tuple values are ordered as queue, message_count, and consumer_count respectively.

Due to Celery's non-ACKing of events, a ring policy is set on any queue that starts with the string 'celeryev' or ends with the string 'pidbox'. These are celery event queues, and Celery does not ack them, causing the messages to build-up. Eventually Qpid stops serving messages unless the 'ring' policy is set, at which point the buffer backing the queue becomes circular.

Parameters

- **queue** (*str*) – The name of the queue to be created.
- **passive** (*bool*) – If True, the sever will not create the queue.
- **durable** (*bool*) – If True, the queue will be durable.
- **exclusive** (*bool*) – If True, the queue will be exclusive.
- **auto_delete** (*bool*) – If True, the queue is deleted when all consumers have finished using it.
- **nowait** (*bool*) – This parameter is unused since the 0-10 specification does not include it.
- **arguments** (*dict or None*) – A set of arguments for the declaration of the queue.

Returns A named tuple representing the declared queue as a named tuple. The tuple values are ordered as queue, message count, and the active consumer count.

Return type `namedtuple`

queue_delete(*queue*, *if_unused=False*, *if_empty=False*, ***kwargs*)

Delete a queue by name.

Delete a queue specified by name. Using the if_unused keyword argument, the delete can only occur if there are 0 consumers bound to it. Using the if_empty keyword argument, the delete can only occur if there are 0 messages in the queue.

Parameters **queue** (*str*) – The name of the queue to be deleted.

Keyword Arguments

- **if_unused** – If True, delete only if the queue has 0 consumers. If False, delete a queue even with consumers bound to it.
- **if_empty** – If True, only delete the queue if it is empty. If False, delete the queue if it is empty or not.

queue_purge(*queue*, ***kwargs*)

Remove all undelivered messages from queue.

Purge all undelivered messages from a queue specified by name. If the queue does not exist an exception is raised. The queue message depth is first checked, and then the broker is asked to purge that number of messages. The integer number of messages requested to be purged is returned. The actual number of messages purged may be different than the requested number of messages to purge.

Sometimes delivered messages are asked to be purged, but are not. This case fails silently, which is the correct behavior when a message that has been delivered to a different consumer, who has not ACKed the message, and still has an active session with the broker. Messages in that case are not safe for purging and will be retained by the broker. The client is unable to change this delivery behavior.

Internally, this method relies on `_purge()`.

Parameters **queue** (*str*) – The name of the queue which should have all messages removed.

Returns The number of messages requested to be purged.

Return type int

Raises `qpid.messaging.exceptions.NotFound` if the queue being purged cannot be found.

queue_unbind(*queue*, *exchange*, *routing_key*, ***kwargs*)
Unbind a queue from an exchange with a given bind key.

Unbind a queue specified by name, from an exchange specified by name, that is already bound with a bind key. The queue and exchange must already exist on the broker, and bound with the bind key for the operation to complete successfully. Queues may be bound to exchanges multiple times with different keys, thus the bind key is a required field to unbind in an explicit way.

Parameters

- **queue** (*str*) – The name of the queue to be unbound.
- **exchange** (*str*) – The name of the exchange that the queue should be unbound from.
- **routing_key** (*str*) – The existing bind key between the specified queue and a specified exchange that should be unbound.

typeof(*exchange*, *default='direct'*)
Get the exchange type.

Lookup and return the exchange type for an exchange specified by name. Exchange types are expected to be 'direct', 'topic', and 'fanout', which correspond with exchange functionality as specified in AMQP 0-10 and earlier. If the exchange cannot be found, the default exchange type is returned.

Parameters **exchange** (*str*) – The exchange to have its type lookup up.

Keyword Arguments **default** – The type of exchange to assume if the exchange does not exist.

Returns The exchange type either 'direct', 'topic', or 'fanout'.

Return type str

close()
Close the connection.

Closing the connection will close all associated session, senders, or receivers used by the Connection.

close_channel(*channel*)
Close a Channel.

Close a channel specified by a reference to the *Channel* object.

Parameters **channel** (*Channel*.) – Channel that should be closed.

get_qpid_connection()
Return the existing connection (singleton).

Returns The existing qpid.messaging.Connection

Return type qpid.messaging.endpoints.Connection

channel_errors = (None,)

close_connection(*connection*)
Close the *Connection* object.

Parameters **connection** (*kombu.transport.qpid.Connection*) – The Connection that should be closed.

connection_errors = (None, <class 'OSError'>)

create_channel(*connection*)
Create and return a *Channel*.

Creates a new channel, and appends the channel to the list of channels known by the Connection. Once the new channel is created, it is returned.

Parameters **connection** (kombu.transport.qpid.Connection) – The connection that should support the new *Channel*.

Returns The new Channel that is made.

Return type *kombu.transport.qpid.Channel*.

property default_connection_params
Return a dict with default connection parameters.

These connection parameters will be used whenever the creator of Transport does not specify a required parameter.

Returns A dict containing the default parameters.

Return type dict

drain_events(*connection*, *timeout=0*, ***kwargs*)
Handle and call callbacks for all ready Transport messages.

Drains all events that are ready from all Receiver that are asynchronously fetching messages.

For each drained message, the message is called to the appropriate callback. Callbacks are organized by queue name.

Parameters **connection** (kombu.transport.qpid.Connection) – The *Connection* that contains the callbacks, indexed by queue name, which will be called by this method.

Keyword Arguments **timeout** – The timeout that limits how long this method will run for. The timeout could interrupt a blocking read that is waiting for a new message, or cause this method to return before all messages are drained. Defaults to 0.

driver_name = 'qpid'

driver_type = 'qpid'

establish_connection()
Establish a Connection object.

Determines the correct options to use when creating any connections needed by this Transport, and create a *Connection* object which saves those values for connections generated as they are needed. The options are a mixture of what is passed in through the creator of the Transport, and the defaults provided by *default_connection_params()*. Options cover broker network settings, timeout behaviors, authentication, and identity verification settings.

This method also creates and stores a `Session` using the `Connection` created by this method. The Session is stored on self.

Returns The created *Connection* object is returned.

Return type *Connection*

implements = {'asynchronous': True, 'exchange_type': frozenset({'direct', 'fanout',

on_readable(*connection*, *loop*)

Handle any messages associated with this Transport.

This method clears a single message from the externally monitored file descriptor by issuing a read call to the self.r file descriptor which removes a single '0' character that was placed into the pipe by the Qpid session message callback handler. Once a '0' is read, all available events are drained through a call to *drain_events()*.

The file descriptor self.r is modified to be non-blocking, ensuring that an accidental call to this method when no more messages will not cause indefinite blocking.

Nothing is expected to be returned from *drain_events()* because *drain_events()* handles messages by calling callbacks that are maintained on the *Connection* object. When *drain_events()* returns, all associated messages have been handled.

This method calls drain_events() which reads as many messages as are available for this Transport, and then returns. It blocks in the sense that reading and handling a large number of messages may take time, but it does not block waiting for a new message to arrive. When *drain_events()* is called a timeout is not specified, which causes this behavior.

One interesting behavior of note is where multiple messages are ready, and this method removes a single '0' character from self.r, but *drain_events()* may handle an arbitrary amount of messages. In that case, extra '0' characters may be left on self.r to be read, where messages corresponding with those '0' characters have already been handled. The external epoll loop will incorrectly think additional data is ready for reading, and will call on_readable unnecessarily, once for each '0' to be read. Additional calls to *on_readable()* produce no negative side effects, and will eventually clear out the symbols from the self.r file descriptor. If new messages show up during this draining period, they will also be properly handled.

Parameters

- **connection** (`kombu.transport.qpid.Connection`) – The connection associated with the readable events, which contains the callbacks that need to be called for the readable objects.
- **loop** (`kombu.asynchronous.Hub`) – The asynchronous loop object that contains epoll like functionality.

polling_interval = None

recoverable_channel_errors = (None,)

recoverable_connection_errors = (None, <class 'OSError'>)

register_with_event_loop(*connection*, *loop*)

Register a file descriptor and callback with the loop.

Register the callback self.on_readable to be called when an external epoll loop sees that the file descriptor registered is ready for reading. The file descriptor is created by this Transport, and is written to when a message is available.

Because supports_ev == True, Celery expects to call this method to give the Transport an opportunity to register a read file descriptor for external monitoring by celery using an Event I/O notification mechanism such as epoll. A callback is also registered that is to be called once the external epoll loop is ready to handle the epoll event associated with messages that are ready to be handled for this Transport.

The registration call is made exactly once per Transport after the Transport is instantiated.

Parameters

- **connection** (kombu.transport.qpid.Connection) – A reference to the connection associated with this Transport.
- **loop** (kombu.asynchronous.hub.Hub) – A reference to the external loop.

verify_runtime_environment()

Verify that the runtime environment is acceptable.

This method is called as part of __init__ and raises a RuntimeError in Python3 or PyPI environments. This module is not compatible with Python3 or PyPI. The RuntimeError identifies this to the user up front along with suggesting Python 2.6+ be used instead.

This method also checks that the dependencies qpidtoollibs and qpid.messaging are installed. If either one is not installed a RuntimeError is raised.

Raises RuntimeError if the runtime environment is not acceptable.

Connection

class kombu.transport.qpid.**Connection**(***connection_options*)

Qpid Connection.

Encapsulate a connection object for the *Transport*.

Parameters

- **host** – The host that connections should connect to.
- **port** – The port that connection should connect to.
- **username** – The username that connections should connect with. Optional.
- **password** – The password that connections should connect with. Optional but requires a username.
- **transport** – The transport type that connections should use. Either 'tcp', or 'ssl' are expected as values.
- **timeout** – the timeout used when a Connection connects to the broker.
- **sasl_mechanisms** – The sasl authentication mechanism type to use. refer to SASL documentation for an explanation of valid values.

Note: qpid.messaging has an AuthenticationFailure exception type, but instead raises a ConnectionError with a message that indicates an authentication failure occurred in those situations. ConnectionError is listed as a recoverable error type, so kombu will attempt to retry if a ConnectionError is raised. Retrying the operation without adjusting the credentials is not correct, so this method specifically checks for a ConnectionError that indicates an Authentication Failure occurred. In those situations, the error type is mutated while preserving the original message and raised so kombu will allow the exception to not be considered recoverable.

A connection object is created by a *Transport* during a call to *establish_connection()*. The *Transport* passes in connection options as keywords that should be used for any connections created. Each *Transport* creates exactly one Connection.

A Connection object maintains a reference to a Connection which can be accessed through a bound getter method named *get_qpid_connection()* method. Each Channel uses a the Connection for each BrokerAgent, and the Transport maintains a session for all senders and receivers.

The Connection object is also responsible for maintaining the dictionary of references to callbacks that should be called when messages are received. These callbacks are saved in _callbacks, and keyed on the queue name associated with the received message. The _callbacks are setup in *Channel.basic_consume()*, removed in *Channel.basic_cancel()*, and called in *Transport.drain_events()*.

The following keys are expected to be passed in as keyword arguments at a minimum:

All keyword arguments are collected into the connection_options dict and passed directly through to qpid.messaging.endpoints.Connection.establish().

class Channel(*connection*, *transport*)

Supports broker configuration and messaging send and receive.

Parameters

- **connection** (kombu.transport.qpid.Connection) – A Connection object that this Channel can reference. Currently only used to access callbacks.
- **transport** (kombu.transport.qpid.Transport) – The Transport this Channel is associated with.

A channel object is designed to have method-parity with a Channel as defined in AMQP 0-10 and earlier, which allows for the following broker actions:

- exchange declare and delete
- queue declare and delete
- queue bind and unbind operations
- queue length and purge operations
- sending/receiving/rejecting messages
- structuring, encoding, and decoding messages
- supports synchronous and asynchronous reads
- reading state about the exchange, queues, and bindings

Channels are designed to all share a single TCP connection with a broker, but provide a level of isolated communication with the broker while benefiting from a shared TCP connection. The Channel is given its *Connection* object by the *Transport* that instantiates the channel.

This channel inherits from StdChannel, which makes this a 'native' channel versus a 'virtual' channel which would inherit from kombu.transports.virtual.

Messages sent using this channel are assigned a delivery_tag. The delivery_tag is generated for a message as they are prepared for sending by *basic_publish()*. The delivery_tag is unique per channel instance. The delivery_tag has no meaningful context in other objects, and is only maintained in the memory of this object, and the underlying *QoS* object that provides support.

Each channel object instantiates exactly one *QoS* object for prefetch limiting, and asynchronous ACKing. The *QoS* object is lazily instantiated through a property method *qos()*. The *QoS* object is a supporting object that should not be accessed directly except by the channel itself.

Synchronous reads on a queue are done using a call to *basic_get()* which uses _get() to perform the reading. These methods read immediately and do not accept any form of timeout. *basic_get()* reads synchronously and ACKs messages before returning them. ACKing is done in all cases, because an application that reads messages using qpid.messaging, but does not ACK them will experience a memory leak. The no_ack argument to *basic_get()* does not affect ACKing functionality.

Asynchronous reads on a queue are done by starting a consumer using *basic_consume()*. Each call to *basic_consume()* will cause a Receiver to be created on the Session started by the :class: *Transport*. The receiver will asynchronously read using qpid.messaging, and prefetch messages before the call to Transport.basic_drain() occurs. The prefetch_count value of the *QoS* object is the capacity value of the new receiver. The new receiver capacity must always be at least 1, otherwise none of the receivers will appear to be ready for reading, and will never be read from.

Each call to *basic_consume()* creates a consumer, which is given a consumer tag that is identified by the caller of *basic_consume()*. Already started consumers can be cancelled using by their consumer_tag using *basic_cancel()*. Cancellation of a consumer causes the Receiver object to be closed.

Asynchronous message ACKing is supported through *basic_ack()*, and is referenced by delivery_tag. The Channel object uses its *QoS* object to perform the message ACKing.

class Message(*payload*, *channel=None*, ***kwargs*)

Message object.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

serializable()

class QoS(*session*, *prefetch_count=1*)

A helper object for message prefetch and ACKing purposes.

Keyword Arguments **prefetch_count** – Initial prefetch count, hard set to 1.

NOTE: prefetch_count is currently hard set to 1, and needs to be improved

This object is instantiated 1-for-1 with a *Channel* instance. QoS allows prefetch_count to be set to the number of outstanding messages the corresponding *Channel* should be allowed to prefetch. Setting prefetch_count to 0 disables prefetch limits, and the object can hold an arbitrary number of messages.

Messages are added using *append()*, which are held until they are ACKed asynchronously through a call to *ack()*. Messages that are received, but not ACKed will not be delivered by the broker to another consumer until an ACK is received, or the session is closed. Messages are referred to using delivery_tag, which are unique per *Channel*. Delivery tags are managed outside of this object and are passed in with a message to *append()*. Un-ACKed messages can be looked up from QoS using *get()* and can be rejected and forgotten using *reject()*.

ack(*delivery_tag*)
Acknowledge a message by delivery_tag.

Called asynchronously once the message has been handled and can be forgotten by the broker.

Parameters **delivery_tag** (*uuid.UUID*) – the delivery tag associated with the message to be acknowledged.

append(*message*, *delivery_tag*)
Append message to the list of un-ACKed messages.

Add a message, referenced by the delivery_tag, for ACKing, rejecting, or getting later. Messages are saved into an `collections.OrderedDict` by delivery_tag.

Parameters

- **message** (*qpid.messaging.Message*) – A received message that has not yet been ACKed.
- **delivery_tag** (*uuid.UUID*) – A UUID to refer to this message by upon receipt.

can_consume()
Return True if the *Channel* can consume more messages.

Used to ensure the client adheres to currently active prefetch limits.

Returns True, if this QoS object can accept more messages without violating the prefetch_count. If prefetch_count is 0, can_consume will always return True.

Return type bool

can_consume_max_estimate()
Return the remaining message capacity.

Returns an estimated number of outstanding messages that a *kombu.transport.qpid.Channel* can accept without exceeding `prefetch_count`. If `prefetch_count` is 0, then this method returns 1.

Returns The number of estimated messages that can be fetched without violating the prefetch_count.

Return type int

get(*delivery_tag*)
Get an un-ACKed message by delivery_tag.

If called with an invalid delivery_tag a `KeyError` is raised.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be returned.

Returns An un-ACKed message that is looked up by delivery_tag.

Return type qpid.messaging.Message

reject(*delivery_tag*, *requeue=False*)
Reject a message by delivery_tag.

Explicitly notify the broker that the channel associated with this QoS object is rejecting the message that was previously delivered.

If requeue is False, then the message is not requeued for delivery to another consumer. If requeue is True, then the message is requeued for delivery to another consumer.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

Keyword Arguments **requeue** – If True, the broker will be notified to requeue the message. If False, the broker will be told to drop the message entirely. In both cases, the message will be removed from this object.

basic_ack(*delivery_tag*, *multiple=False*)
Acknowledge a message by delivery_tag.

Acknowledges a message referenced by delivery_tag. Messages can only be ACKed using *basic_ack()* if they were acquired using *basic_consume()*. This is the ACKing portion of the asynchronous read behavior.

Internally, this method uses the *QoS* object, which stores messages and is responsible for the ACKing.

Parameters

- **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be acknowledged.
- **multiple** (*bool*) – not implemented. If set to True an AssertionError is raised.

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

Request the consumer stops reading messages from its queue. The consumer is a `Receiver`, and it is closed using `close()`.

This method also cleans up all lingering references of the consumer.

Parameters **consumer_tag** (*an immutable object*) – The tag which refers to the consumer to be cancelled. Originally specified when the consumer was created as a parameter to *basic_consume()*.

basic_consume(*queue*, *no_ack*, *callback*, *consumer_tag*, ***kwargs*)
Start an asynchronous consumer that reads from a queue.

This method starts a consumer of type `Receiver` using the `Session` created and referenced by the *Transport* that reads messages from a queue specified by name until stopped by a call to *basic_cancel()*.

Messages are available later through a synchronous call to *Transport.drain_events()*, which will drain from the consumer started by this method. *Transport.drain_events()* is synchronous, but the receiving of messages over the network occurs asynchronously, so it should still perform well. *Transport.drain_events()* calls the callback provided here with the Message of type self.Message.

Each consumer is referenced by a consumer_tag, which is provided by the caller of this method.

This method sets up the callback onto the self.connection object in a dict keyed by queue name. *drain_events()* is responsible for calling that callback upon message receipt.

All messages that are received are added to the QoS object to be saved for asynchronous ACKing later after the message has been handled by the caller of *drain_events()*. Messages can be ACKed after being received through a call to *basic_ack()*.

If no_ack is True, The no_ack flag indicates that the receiver of the message will not call *basic_ack()* later. Since the message will not be ACKed later, it is ACKed immediately.

basic_consume() transforms the message object type prior to calling the callback. Initially the message comes in as a `qpid.messaging.Message`. This method unpacks the payload of the `qpid.messaging.Message` and creates a new object of type self.Message.

This method wraps the user delivered callback in a runtime-built function which provides the type transformation from `qpid.messaging.Message` to *Message*, and adds the message to the associated *QoS* object for asynchronous ACKing if necessary.

Parameters

- **queue** (*str*) – The name of the queue to consume messages from
- **no_ack** (*bool*) – If True, then messages will not be saved for ACKing later, but will be ACKed immediately. If False, then messages will be saved for ACKing later with a call to *basic_ack()*.
- **callback** (*a callable object*) – a callable that will be called when messages arrive on the queue.
- **consumer_tag** (*an immutable object*) – a tag to reference the created consumer by. This consumer_tag is needed to cancel the consumer.

basic_get(*queue*, *no_ack=False*, ***kwargs*)

Non-blocking single message get and ACK from a queue by name.

Internally this method uses `_get()` to fetch the message. If an `Empty` exception is raised by `_get()`, this method silences it and returns None. If `_get()` does return a message, that message is ACKed. The no_ack parameter has no effect on ACKing behavior, and all messages are ACKed in all cases. This method never adds fetched Messages to the internal QoS object for asynchronous ACKing.

This method converts the object type of the method as it passes through. Fetching from the broker, `_get()` returns a `qpid.messaging.Message`, but this method takes the payload of the `qpid.messaging.Message` and instantiates a *Message* object with the payload based on the class setting of self.Message.

Parameters **queue** (*str*) – The queue name to fetch a message from.

Keyword Arguments **no_ack** – The no_ack parameter has no effect on the ACK behavior of this method. Un-ACKed messages create a memory leak in qpid.messaging, and need to be ACKed in all cases.

Returns The received message.

Return type *Message*

basic_publish(*message*, *exchange*, *routing_key*, ***kwargs*)

Publish message onto an exchange using a routing key.

Publish a message onto an exchange specified by name using a routing key specified by routing_key. Prepares the message in the following ways before sending:

- encodes the body using *encode_body()*
- **wraps the body as a buffer object, so that** `qpid.messaging.endpoints.Sender` uses a content type that can support arbitrarily large messages.
- sets delivery_tag to a random uuid.UUID
- sets the exchange and routing_key info as delivery_info

Internally uses `_put()` to send the message synchronously. This message is typically called by `kombu.messaging.Producer._publish` as the final step in message publication.

Parameters

- **message** (*dict*) – A dict containing key value pairs with the message data. A valid message dict can be generated using the *prepare_message()* method.
- **exchange** (*str*) – The name of the exchange to submit this message onto.
- **routing_key** (*str*) – The routing key to be used as the message is submitted onto the exchange.

basic_qos(*prefetch_count*, **args*)
Change *QoS* settings for this Channel.

Set the number of un-acknowledged messages this Channel can fetch and hold. The prefetch_value is also used as the capacity for any new `Receiver` objects.

Currently, this value is hard coded to 1.

Parameters **prefetch_count** (*int*) – Not used. This method is hard-coded to 1.

basic_reject(*delivery_tag*, *requeue=False*)
Reject a message by delivery_tag.

Rejects a message that has been received by the Channel, but not yet acknowledged. Messages are referenced by their delivery_tag.

If requeue is False, the rejected message will be dropped by the broker and not delivered to any other consumers. If requeue is True, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

Keyword Arguments **requeue** – If False, the rejected message will be dropped by the broker and not delivered to any other consumers. If True, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

body_encoding = 'base64'

close()
Cancel all associated messages and close the Channel.

This cancels all consumers by calling *basic_cancel()* for each known consumer_tag. It also closes the self._broker sessions. Closing the sessions implicitly causes all outstanding, un-ACKed messages to be considered undelivered by the broker.

codecs = {'base64': <kombu.transport.virtual.base.Base64 object>}

decode_body(*body*, *encoding=None*)
Decode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in self.codecs. self.codecs uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through encode and decode methods.

Parameters **body** (*str*) – The body to be encoded.

Keyword Arguments **encoding** – The encoding type to be used. Must be a supported codec listed in self.codecs.

Returns If encoding is specified, the decoded body is returned. If encoding is not specified, the body is returned unchanged.

Return type str

encode_body(*body*, *encoding=None*)
Encode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in self.codecs. self.codecs uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through encode and decode methods.

Parameters **body** (*str*) – The body to be encoded.

Keyword Arguments **encoding** – The encoding type to be used. Must be a supported codec listed in self.codecs.

Returns If encoding is specified, return a tuple with the first position being the encoded body, and the second position the encoding used. If encoding is not specified, the body is passed through unchanged.

Return type tuple

exchange_declare(*exchange=''*, *type='direct'*, *durable=False*, ***kwargs*)
Create a new exchange.

Create an exchange of a specific type, and optionally have the exchange be durable. If an exchange of the requested name already exists, no action is taken and no exceptions are raised. Durable exchanges will survive a broker restart, non-durable exchanges will not.

Exchanges provide behaviors based on their type. The expected behaviors are those defined in the AMQP 0-10 and prior specifications including 'direct', 'topic', and 'fanout' functionality.

Keyword Arguments

- **type** – The exchange type. Valid values include 'direct', 'topic', and 'fanout'.
- **exchange** – The name of the exchange to be created. If no exchange is specified, then a blank string will be used as the name.
- **durable** – True if the exchange should be durable, or False otherwise.

exchange_delete(*exchange_name*, ***kwargs*)
Delete an exchange specified by name.

Parameters **exchange_name** (*str*) – The name of the exchange to be deleted.

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*)
Prepare message data for sending.

This message is typically called by `kombu.messaging.Producer._publish()` as a preparation step in message publication.

Parameters **body** (*str*) – The body of the message

Keyword Arguments

- **priority** – A number between 0 and 9 that sets the priority of the message.
- **content_type** – The content_type the message body should be treated as. If this is unset, the `qpid.messaging.endpoints.Sender` object tries to autodetect the content_type from the body.
- **content_encoding** – The content_encoding the message body is encoded as.
- **headers** – Additional Message headers that should be set. Passed in as a key-value pair.

- **properties** – Message properties to be set on the message.

Returns Returns a dict object that encapsulates message attributes. See parameters for more details on attributes that can be set.

Return type dict

property qos

QoS manager for this channel.

Lazily instantiates an object of type *QoS* upon access to the self.qos attribute.

Returns An already existing, or newly created QoS object

Return type *QoS*

queue_bind(*queue*, *exchange*, *routing_key*, ***kwargs*)

Bind a queue to an exchange with a bind key.

Bind a queue specified by name, to an exchange specified by name, with a specific bind key. The queue and exchange must already exist on the broker for the bind to complete successfully. Queues may be bound to exchanges multiple times with different keys.

Parameters

- **queue** (*str*) – The name of the queue to be bound.
- **exchange** (*str*) – The name of the exchange that the queue should be bound to.
- **routing_key** (*str*) – The bind key that the specified queue should bind to the specified exchange with.

queue_declare(*queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*)

Create a new queue specified by name.

If the queue already exists, no change is made to the queue, and the return value returns information about the existing queue.

The queue name is required and specified as the first argument.

If passive is True, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state. Default is False.

If durable is True, the queue will be durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. Default is False.

If exclusive is True, the queue will be exclusive. Exclusive queues may only be consumed by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'. Default is False.

If auto_delete is True, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Default is True.

The nowait parameter is unused. It was part of the 0-9-1 protocol, but this AMQP client implements 0-10 which removed the nowait option.

The arguments parameter is a set of arguments for the declaration of the queue. Arguments are passed as a dict or None. This field is ignored if passive is True. Default is None.

This method returns a `namedtuple` with the name 'queue_declare_ok_t' and the queue name as 'queue', message count on the queue as 'message_count', and the number of active consumers

as 'consumer_count'. The named tuple values are ordered as queue, message_count, and consumer_count respectively.

Due to Celery's non-ACKing of events, a ring policy is set on any queue that starts with the string 'celeryev' or ends with the string 'pidbox'. These are celery event queues, and Celery does not ack them, causing the messages to build-up. Eventually Qpid stops serving messages unless the 'ring' policy is set, at which point the buffer backing the queue becomes circular.

Parameters

- **queue** (*str*) – The name of the queue to be created.
- **passive** (*bool*) – If True, the sever will not create the queue.
- **durable** (*bool*) – If True, the queue will be durable.
- **exclusive** (*bool*) – If True, the queue will be exclusive.
- **auto_delete** (*bool*) – If True, the queue is deleted when all consumers have finished using it.
- **nowait** (*bool*) – This parameter is unused since the 0-10 specification does not include it.
- **arguments** (*dict or None*) – A set of arguments for the declaration of the queue.

Returns A named tuple representing the declared queue as a named tuple. The tuple values are ordered as queue, message count, and the active consumer count.

Return type namedtuple

queue_delete(*queue*, *if_unused=False*, *if_empty=False*, ***kwargs*)

Delete a queue by name.

Delete a queue specified by name. Using the if_unused keyword argument, the delete can only occur if there are 0 consumers bound to it. Using the if_empty keyword argument, the delete can only occur if there are 0 messages in the queue.

Parameters **queue** (*str*) – The name of the queue to be deleted.

Keyword Arguments

- **if_unused** – If True, delete only if the queue has 0 consumers. If False, delete a queue even with consumers bound to it.
- **if_empty** – If True, only delete the queue if it is empty. If False, delete the queue if it is empty or not.

queue_purge(*queue*, ***kwargs*)

Remove all undelivered messages from queue.

Purge all undelivered messages from a queue specified by name. If the queue does not exist an exception is raised. The queue message depth is first checked, and then the broker is asked to purge that number of messages. The integer number of messages requested to be purged is returned. The actual number of messages purged may be different than the requested number of messages to purge.

Sometimes delivered messages are asked to be purged, but are not. This case fails silently, which is the correct behavior when a message that has been delivered to a different consumer, who has not ACKed the message, and still has an active session with the broker. Messages in that case are not safe for purging and will be retained by the broker. The client is unable to change this delivery behavior.

Internally, this method relies on `_purge()`.

Parameters **queue** (*str*) – The name of the queue which should have all messages removed.

Returns The number of messages requested to be purged.

Return type int

Raises `qpid.messaging.exceptions.NotFound` if the queue being purged cannot be found.

queue_unbind(*queue*, *exchange*, *routing_key*, ***kwargs*)
Unbind a queue from an exchange with a given bind key.

Unbind a queue specified by name, from an exchange specified by name, that is already bound with a bind key. The queue and exchange must already exist on the broker, and bound with the bind key for the operation to complete successfully. Queues may be bound to exchanges multiple times with different keys, thus the bind key is a required field to unbind in an explicit way.

Parameters

- **queue** (*str*) – The name of the queue to be unbound.
- **exchange** (*str*) – The name of the exchange that the queue should be unbound from.
- **routing_key** (*str*) – The existing bind key between the specified queue and a specified exchange that should be unbound.

typeof(*exchange*, *default='direct'*)
Get the exchange type.

Lookup and return the exchange type for an exchange specified by name. Exchange types are expected to be 'direct', 'topic', and 'fanout', which correspond with exchange functionality as specified in AMQP 0-10 and earlier. If the exchange cannot be found, the default exchange type is returned.

Parameters **exchange** (*str*) – The exchange to have its type lookup up.

Keyword Arguments **default** – The type of exchange to assume if the exchange does not exist.

Returns The exchange type either 'direct', 'topic', or 'fanout'.

Return type str

close()
Close the connection.

Closing the connection will close all associated session, senders, or receivers used by the Connection.

close_channel(*channel*)
Close a Channel.

Close a channel specified by a reference to the *Channel* object.

Parameters **channel** (*Channel*.) – Channel that should be closed.

get_qpid_connection()
Return the existing connection (singleton).

Returns The existing qpid.messaging.Connection

Return type `qpid.messaging.endpoints.Connection`

Channel

class kombu.transport.qpid.**Channel**(*connection*, *transport*)

Supports broker configuration and messaging send and receive.

Parameters

- **connection** (kombu.transport.qpid.Connection) – A Connection object that this Channel can reference. Currently only used to access callbacks.
- **transport** (kombu.transport.qpid.Transport) – The Transport this Channel is associated with.

A channel object is designed to have method-parity with a Channel as defined in AMQP 0-10 and earlier, which allows for the following broker actions:

- exchange declare and delete
- queue declare and delete
- queue bind and unbind operations
- queue length and purge operations
- sending/receiving/rejecting messages
- structuring, encoding, and decoding messages
- supports synchronous and asynchronous reads
- reading state about the exchange, queues, and bindings

Channels are designed to all share a single TCP connection with a broker, but provide a level of isolated communication with the broker while benefiting from a shared TCP connection. The Channel is given its *Connection* object by the *Transport* that instantiates the channel.

This channel inherits from StdChannel, which makes this a 'native' channel versus a 'virtual' channel which would inherit from kombu.transports.virtual.

Messages sent using this channel are assigned a delivery_tag. The delivery_tag is generated for a message as they are prepared for sending by *basic_publish()*. The delivery_tag is unique per channel instance. The delivery_tag has no meaningful context in other objects, and is only maintained in the memory of this object, and the underlying *QoS* object that provides support.

Each channel object instantiates exactly one *QoS* object for prefetch limiting, and asynchronous ACKing. The *QoS* object is lazily instantiated through a property method *qos()*. The *QoS* object is a supporting object that should not be accessed directly except by the channel itself.

Synchronous reads on a queue are done using a call to *basic_get()* which uses _get() to perform the reading. These methods read immediately and do not accept any form of timeout. *basic_get()* reads synchronously and ACKs messages before returning them. ACKing is done in all cases, because an application that reads messages using qpid.messaging, but does not ACK them will experience a memory leak. The no_ack argument to *basic_get()* does not affect ACKing functionality.

Asynchronous reads on a queue are done by starting a consumer using *basic_consume()*. Each call to *basic_consume()* will cause a Receiver to be created on the Session started by the :class: *Transport*. The receiver will asynchronously read using qpid.messaging, and prefetch messages before the call to Transport.basic_drain() occurs. The prefetch_count value of the *QoS* object is the capacity value of the new receiver. The new receiver capacity must always be at least 1, otherwise none of the receivers will appear to be ready for reading, and will never be read from.

Each call to *basic_consume()* creates a consumer, which is given a consumer tag that is identified by the caller of *basic_consume()*. Already started consumers can be cancelled using by their consumer_tag using *basic_cancel()*. Cancellation of a consumer causes the Receiver object to be closed.

Asynchronous message ACKing is supported through *basic_ack()*, and is referenced by delivery_tag. The Channel object uses its *QoS* object to perform the message ACKing.

class Message(*payload*, *channel=None*, ***kwargs*)

message class used.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

serializable()

class QoS(*session*, *prefetch_count=1*)

A class reference that will be instantiated using the qos property.

ack(*delivery_tag*)

Acknowledge a message by delivery_tag.

Called asynchronously once the message has been handled and can be forgotten by the broker.

Parameters **delivery_tag** (*uuid.UUID*) – the delivery tag associated with the message to be acknowledged.

append(*message*, *delivery_tag*)

Append message to the list of un-ACKed messages.

Add a message, referenced by the delivery_tag, for ACKing, rejecting, or getting later. Messages are saved into an `collections.OrderedDict` by delivery_tag.

Parameters

- **message** (*qpid.messaging.Message*) – A received message that has not yet been ACKed.
- **delivery_tag** (*uuid.UUID*) – A UUID to refer to this message by upon receipt.

can_consume()

Return True if the *Channel* can consume more messages.

Used to ensure the client adheres to currently active prefetch limits.

Returns True, if this QoS object can accept more messages without violating the prefetch_count. If prefetch_count is 0, can_consume will always return True.

Return type bool

can_consume_max_estimate()

Return the remaining message capacity.

Returns an estimated number of outstanding messages that a *kombu.transport.qpid.Channel* can accept without exceeding `prefetch_count`. If `prefetch_count` is 0, then this method returns 1.

Returns The number of estimated messages that can be fetched without violating the prefetch_count.

Return type int

get(*delivery_tag*)
Get an un-ACKed message by delivery_tag.

If called with an invalid delivery_tag a `KeyError` is raised.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be returned.

Returns An un-ACKed message that is looked up by delivery_tag.

Return type qpid.messaging.Message

reject(*delivery_tag*, *requeue=False*)
Reject a message by delivery_tag.

Explicitly notify the broker that the channel associated with this QoS object is rejecting the message that was previously delivered.

If requeue is False, then the message is not requeued for delivery to another consumer. If requeue is True, then the message is requeued for delivery to another consumer.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

Keyword Arguments **requeue** – If True, the broker will be notified to requeue the message. If False, the broker will be told to drop the message entirely. In both cases, the message will be removed from this object.

basic_ack(*delivery_tag*, *multiple=False*)
Acknowledge a message by delivery_tag.

Acknowledges a message referenced by delivery_tag. Messages can only be ACKed using *basic_ack()* if they were acquired using *basic_consume()*. This is the ACKing portion of the asynchronous read behavior.

Internally, this method uses the *QoS* object, which stores messages and is responsible for the ACKing.

Parameters

- **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be acknowledged.
- **multiple** (*bool*) – not implemented. If set to True an AssertionError is raised.

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

Request the consumer stops reading messages from its queue. The consumer is a `Receiver`, and it is closed using `close()`.

This method also cleans up all lingering references of the consumer.

Parameters **consumer_tag** (*an immutable object*) – The tag which refers to the consumer to be cancelled. Originally specified when the consumer was created as a parameter to *basic_consume()*.

basic_consume(*queue*, *no_ack*, *callback*, *consumer_tag*, ***kwargs*)
Start an asynchronous consumer that reads from a queue.

This method starts a consumer of type `Receiver` using the `Session` created and referenced by the *Transport* that reads messages from a queue specified by name until stopped by a call to *basic_cancel()*.

Messages are available later through a synchronous call to *Transport.drain_events()*, which will drain from the consumer started by this method. *Transport.drain_events()* is synchronous,

but the receiving of messages over the network occurs asynchronously, so it should still perform well. *Transport.drain_events()* calls the callback provided here with the Message of type self.Message.

Each consumer is referenced by a consumer_tag, which is provided by the caller of this method.

This method sets up the callback onto the self.connection object in a dict keyed by queue name. *drain_events()* is responsible for calling that callback upon message receipt.

All messages that are received are added to the QoS object to be saved for asynchronous ACKing later after the message has been handled by the caller of *drain_events()*. Messages can be ACKed after being received through a call to *basic_ack()*.

If no_ack is True, The no_ack flag indicates that the receiver of the message will not call *basic_ack()* later. Since the message will not be ACKed later, it is ACKed immediately.

basic_consume() transforms the message object type prior to calling the callback. Initially the message comes in as a `qpid.messaging.Message`. This method unpacks the payload of the `qpid.messaging.Message` and creates a new object of type self.Message.

This method wraps the user delivered callback in a runtime-built function which provides the type transformation from `qpid.messaging.Message` to *Message*, and adds the message to the associated *QoS* object for asynchronous ACKing if necessary.

Parameters

- **queue** (*str*) – The name of the queue to consume messages from
- **no_ack** (*bool*) – If True, then messages will not be saved for ACKing later, but will be ACKed immediately. If False, then messages will be saved for ACKing later with a call to *basic_ack()*.
- **callback** (*a callable object*) – a callable that will be called when messages arrive on the queue.
- **consumer_tag** (*an immutable object*) – a tag to reference the created consumer by. This consumer_tag is needed to cancel the consumer.

basic_get(*queue*, *no_ack=False*, ***kwargs*)

Non-blocking single message get and ACK from a queue by name.

Internally this method uses `_get()` to fetch the message. If an `Empty` exception is raised by `_get()`, this method silences it and returns None. If `_get()` does return a message, that message is ACKed. The no_ack parameter has no effect on ACKing behavior, and all messages are ACKed in all cases. This method never adds fetched Messages to the internal QoS object for asynchronous ACKing.

This method converts the object type of the method as it passes through. Fetching from the broker, `_get()` returns a `qpid.messaging.Message`, but this method takes the payload of the `qpid.messaging.Message` and instantiates a *Message* object with the payload based on the class setting of self.Message.

Parameters **queue** (*str*) – The queue name to fetch a message from.

Keyword Arguments **no_ack** – The no_ack parameter has no effect on the ACK behavior of this method. Un-ACKed messages create a memory leak in qpid.messaging, and need to be ACKed in all cases.

Returns The received message.

Return type *Message*

basic_publish(*message*, *exchange*, *routing_key*, ***kwargs*)

Publish message onto an exchange using a routing key.

Publish a message onto an exchange specified by name using a routing key specified by routing_key. Prepares the message in the following ways before sending:

- encodes the body using *encode_body()*
- **wraps the body as a buffer object, so that** `qpid.messaging.endpoints.Sender` uses a content type that can support arbitrarily large messages.
- sets delivery_tag to a random uuid.UUID
- sets the exchange and routing_key info as delivery_info

Internally uses `_put()` to send the message synchronously. This message is typically called by `kombu.messaging.Producer._publish` as the final step in message publication.

Parameters

- **message** (*dict*) – A dict containing key value pairs with the message data. A valid message dict can be generated using the *prepare_message()* method.
- **exchange** (*str*) – The name of the exchange to submit this message onto.
- **routing_key** (*str*) – The routing key to be used as the message is submitted onto the exchange.

basic_qos(*prefetch_count*, **args*)

Change *QoS* settings for this Channel.

Set the number of un-acknowledged messages this Channel can fetch and hold. The prefetch_value is also used as the capacity for any new `Receiver` objects.

Currently, this value is hard coded to 1.

Parameters **prefetch_count** (*int*) – Not used. This method is hard-coded to 1.

basic_reject(*delivery_tag*, *requeue=False*)

Reject a message by delivery_tag.

Rejects a message that has been received by the Channel, but not yet acknowledged. Messages are referenced by their delivery_tag.

If requeue is False, the rejected message will be dropped by the broker and not delivered to any other consumers. If requeue is True, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

Parameters **delivery_tag** (*uuid.UUID*) – The delivery tag associated with the message to be rejected.

Keyword Arguments **requeue** – If False, the rejected message will be dropped by the broker and not delivered to any other consumers. If True, then the rejected message will be requeued for delivery to another consumer, potentially to the same consumer who rejected the message previously.

body_encoding = 'base64'

Default body encoding. NOTE: `transport_options['body_encoding']` will override this value.

close()

Cancel all associated messages and close the Channel.

This cancels all consumers by calling *basic_cancel()* for each known consumer_tag. It also closes the self._broker sessions. Closing the sessions implicitly causes all outstanding, un-ACKed messages to be considered undelivered by the broker.

codecs = {'base64': <kombu.transport.virtual.base.Base64 object>}
Binary <-> ASCII codecs.

decode_body(*body*, *encoding=None*)
Decode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in self.codecs. self.codecs uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through encode and decode methods.

Parameters **body** (*str*) – The body to be encoded.

Keyword Arguments **encoding** – The encoding type to be used. Must be a supported codec listed in self.codecs.

Returns If encoding is specified, the decoded body is returned. If encoding is not specified, the body is returned unchanged.

Return type str

encode_body(*body*, *encoding=None*)
Encode a body using an optionally specified encoding.

The encoding can be specified by name, and is looked up in self.codecs. self.codecs uses strings as its keys which specify the name of the encoding, and then the value is an instantiated object that can provide encoding/decoding of that type through encode and decode methods.

Parameters **body** (*str*) – The body to be encoded.

Keyword Arguments **encoding** – The encoding type to be used. Must be a supported codec listed in self.codecs.

Returns If encoding is specified, return a tuple with the first position being the encoded body, and the second position the encoding used. If encoding is not specified, the body is passed through unchanged.

Return type tuple

exchange_declare(*exchange=''*, *type='direct'*, *durable=False*, ***kwargs*)
Create a new exchange.

Create an exchange of a specific type, and optionally have the exchange be durable. If an exchange of the requested name already exists, no action is taken and no exceptions are raised. Durable exchanges will survive a broker restart, non-durable exchanges will not.

Exchanges provide behaviors based on their type. The expected behaviors are those defined in the AMQP 0-10 and prior specifications including 'direct', 'topic', and 'fanout' functionality.

Keyword Arguments

- **type** – The exchange type. Valid values include 'direct', 'topic', and 'fanout'.
- **exchange** – The name of the exchange to be created. If no exchange is specified, then a blank string will be used as the name.
- **durable** – True if the exchange should be durable, or False otherwise.

exchange_delete(*exchange_name*, ***kwargs*)
Delete an exchange specified by name.

Parameters **exchange_name** (*str*) – The name of the exchange to be deleted.

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*)
Prepare message data for sending.

This message is typically called by `kombu.messaging.Producer._publish()` as a preparation step in message publication.

Parameters **body** (*str*) – The body of the message

Keyword Arguments

- **priority** – A number between 0 and 9 that sets the priority of the message.
- **content_type** – The content_type the message body should be treated as. If this is unset, the `qpid.messaging.endpoints.Sender` object tries to autodetect the content_type from the body.
- **content_encoding** – The content_encoding the message body is encoded as.
- **headers** – Additional Message headers that should be set. Passed in as a key-value pair.
- **properties** – Message properties to be set on the message.

Returns Returns a dict object that encapsulates message attributes. See parameters for more details on attributes that can be set.

Return type dict

property qos

QoS manager for this channel.

Lazily instantiates an object of type *QoS* upon access to the self.qos attribute.

Returns An already existing, or newly created QoS object

Return type *QoS*

queue_bind(*queue*, *exchange*, *routing_key*, ***kwargs*)

Bind a queue to an exchange with a bind key.

Bind a queue specified by name, to an exchange specified by name, with a specific bind key. The queue and exchange must already exist on the broker for the bind to complete successfully. Queues may be bound to exchanges multiple times with different keys.

Parameters

- **queue** (*str*) – The name of the queue to be bound.
- **exchange** (*str*) – The name of the exchange that the queue should be bound to.
- **routing_key** (*str*) – The bind key that the specified queue should bind to the specified exchange with.

queue_declare(*queue*, *passive=False*, *durable=False*, *exclusive=False*, *auto_delete=True*, *nowait=False*, *arguments=None*)

Create a new queue specified by name.

If the queue already exists, no change is made to the queue, and the return value returns information about the existing queue.

The queue name is required and specified as the first argument.

If passive is True, the server will not create the queue. The client can use this to check whether a queue exists without modifying the server state. Default is False.

If durable is True, the queue will be durable. Durable queues remain active when a server restarts. Non-durable queues (transient queues) are purged if/when a server restarts. Note that durable queues do not necessarily hold persistent messages, although it does not make sense to send persistent messages to a transient queue. Default is False.

If exclusive is True, the queue will be exclusive. Exclusive queues may only be consumed by the current connection. Setting the 'exclusive' flag always implies 'auto-delete'. Default is False.

If auto_delete is True, the queue is deleted when all consumers have finished using it. The last consumer can be cancelled either explicitly or because its channel is closed. If there was no consumer ever on the queue, it won't be deleted. Default is True.

The nowait parameter is unused. It was part of the 0-9-1 protocol, but this AMQP client implements 0-10 which removed the nowait option.

The arguments parameter is a set of arguments for the declaration of the queue. Arguments are passed as a dict or None. This field is ignored if passive is True. Default is None.

This method returns a `namedtuple` with the name 'queue_declare_ok_t' and the queue name as 'queue', message count on the queue as 'message_count', and the number of active consumers as 'consumer_count'. The named tuple values are ordered as queue, message_count, and consumer_count respectively.

Due to Celery's non-ACKing of events, a ring policy is set on any queue that starts with the string 'celeryev' or ends with the string 'pidbox'. These are celery event queues, and Celery does not ack them, causing the messages to build-up. Eventually Qpid stops serving messages unless the 'ring' policy is set, at which point the buffer backing the queue becomes circular.

Parameters

- **queue** (*str*) – The name of the queue to be created.
- **passive** (*bool*) – If True, the sever will not create the queue.
- **durable** (*bool*) – If True, the queue will be durable.
- **exclusive** (*bool*) – If True, the queue will be exclusive.
- **auto_delete** (*bool*) – If True, the queue is deleted when all consumers have finished using it.
- **nowait** (*bool*) – This parameter is unused since the 0-10 specification does not include it.
- **arguments** (*dict or None*) – A set of arguments for the declaration of the queue.

Returns A named tuple representing the declared queue as a named tuple. The tuple values are ordered as queue, message count, and the active consumer count.

Return type `namedtuple`

queue_delete(*queue*, *if_unused=False*, *if_empty=False*, ***kwargs*)

Delete a queue by name.

Delete a queue specified by name. Using the if_unused keyword argument, the delete can only occur if there are 0 consumers bound to it. Using the if_empty keyword argument, the delete can only occur if there are 0 messages in the queue.

Parameters **queue** (*str*) – The name of the queue to be deleted.

Keyword Arguments

- **if_unused** – If True, delete only if the queue has 0 consumers. If False, delete a queue even with consumers bound to it.
- **if_empty** – If True, only delete the queue if it is empty. If False, delete the queue if it is empty or not.

queue_purge(*queue*, ***kwargs*)
Remove all undelivered messages from queue.

Purge all undelivered messages from a queue specified by name. If the queue does not exist an exception is raised. The queue message depth is first checked, and then the broker is asked to purge that number of messages. The integer number of messages requested to be purged is returned. The actual number of messages purged may be different than the requested number of messages to purge.

Sometimes delivered messages are asked to be purged, but are not. This case fails silently, which is the correct behavior when a message that has been delivered to a different consumer, who has not ACKed the message, and still has an active session with the broker. Messages in that case are not safe for purging and will be retained by the broker. The client is unable to change this delivery behavior.

Internally, this method relies on `_purge()`.

Parameters **queue** (*str*) – The name of the queue which should have all messages removed.

Returns The number of messages requested to be purged.

Return type int

Raises `qpid.messaging.exceptions.NotFound` if the queue being purged cannot be found.

queue_unbind(*queue*, *exchange*, *routing_key*, ***kwargs*)
Unbind a queue from an exchange with a given bind key.

Unbind a queue specified by name, from an exchange specified by name, that is already bound with a bind key. The queue and exchange must already exist on the broker, and bound with the bind key for the operation to complete successfully. Queues may be bound to exchanges multiple times with different keys, thus the bind key is a required field to unbind in an explicit way.

Parameters

- **queue** (*str*) – The name of the queue to be unbound.
- **exchange** (*str*) – The name of the exchange that the queue should be unbound from.
- **routing_key** (*str*) – The existing bind key between the specified queue and a specified exchange that should be unbound.

typeof(*exchange*, *default='direct'*)
Get the exchange type.

Lookup and return the exchange type for an exchange specified by name. Exchange types are expected to be 'direct', 'topic', and 'fanout', which correspond with exchange functionality as specified in AMQP 0-10 and earlier. If the exchange cannot be found, the default exchange type is returned.

Parameters **exchange** (*str*) – The exchange to have its type lookup up.

Keyword Arguments **default** – The type of exchange to assume if the exchange does not exist.

Returns The exchange type either 'direct', 'topic', or 'fanout'.

Return type str

Message

class kombu.transport.qpid.**Message**(*payload*, *channel=None*, ***kwargs*)
Message object.

accept

body

channel

content_encoding

content_type

delivery_info

delivery_tag

headers

properties

serializable()

4.37 In-memory Transport - kombu.transport.memory

In-memory transport.

4.37.1 Transport

class kombu.transport.memory.**Transport**(*client*, ***kwargs*)

In-memory Transport.

class Channel(*connection*, ***kwargs*)

In-memory Channel.

after_reply_message_received(*queue*)

Callback called after RPC reply received.

Notes

Reply queue semantics: can be used to delete the queue after transient reply message received.

close()

Close channel.

Cancel all consumers, and requeue unacked messages.

do_restore = False

events = {}

queues = {}

supports_fanout = True

driver_name = 'memory'

driver_type = 'memory'

driver_version()

implements = {'asynchronous': False, 'exchange_type': frozenset({'direct', 'fanout',

state = <kombu.transport.virtual.base.BrokerState object>
memory backend state is global.

4.37.2 Channel

class kombu.transport.memory.**Channel**(*connection*, ***kwargs*)
In-memory Channel.

after_reply_message_received(*queue*)
Callback called after RPC reply received.

Notes

Reply queue semantics: can be used to delete the queue after transient reply message received.

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

do_restore = False

events = {}

queues = {}

supports_fanout = True

4.38 Redis Transport - kombu.transport.redis

Redis transport.

4.38.1 Transport

class kombu.transport.redis.**Transport**(**args*, ***kwargs*)
Redis Transport.

class Channel(**args*, ***kwargs*)
Redis Channel.

class QoS(**args*, ***kwargs*)
Redis Ack Emulation.

ack(*delivery_tag*)
Acknowledge message and remove from transactional state.

append(*message*, *delivery_tag*)
Append message to transactional state.

pipe_or_acquire(*pipe=None*, *client=None*)

reject(*delivery_tag*, *requeue=False*)
Remove from transactional state and requeue message.

restore_at_shutdown = True

restore_by_tag(*tag*, *client=None*, *leftmost=False*)

restore_unacked(*client=None*)
Restore all unacknowledged messages.

restore_visible(*start=0*, *num=10*, *interval=10*)
Restore any pending unackwnowledged messages.

To be filled in for visibility_timeout style implementations.

Note: This is implementation optional, and currently only used by the Redis transport.

unacked_index_key

unacked_key

unacked_mutex_expire

unacked_mutex_key

visibility_timeout

ack_emulation = True

property active_queues
Set of queues being consumed from (excluding fanout queues).

property async_pool

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

basic_consume(*queue*, **args*, ***kwargs*)
Consume from *queue*.

client
Client used to publish messages, BRPOP etc.

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

conn_or_acquire(*client=None*)

connection_class = None

fanout_patterns = True

fanout_prefix = True

from_transport_options = ('body_encoding', 'deadletter_queue', 'sep', 'ack_emulati

get_table(*exchange*)
Get table of bindings for *exchange*.

keyprefix_fanout = '/{db}.'

keyprefix_queue = '_kombu.binding.%s'

max_connections = 10

property pool

priority(*n*)

priority_steps = [0, 3, 6, 9]

queue_order_strategy = 'round_robin'

sep = '\x06\x16'

socket_connect_timeout = None

socket_keepalive = None

socket_keepalive_options = None

socket_timeout = None

subclient
Pub/Sub connection used to consume fanout queues.

supports_fanout = True

unacked_index_key = 'unacked_index'

unacked_key = 'unacked'

unacked_mutex_expire = 300

unacked_mutex_key = 'unacked_mutex'

unacked_restore_limit = None

visibility_timeout = 3600

default_port = 6379

driver_name = 'redis'

driver_type = 'redis'

driver_version()

implements = {'asynchronous': True, 'exchange_type': frozenset({'direct', 'fanout',

on_readable(*fileno*)
Handle AIO event for one of our file descriptors.

polling_interval = None

register_with_event_loop(*connection*, *loop*)

4.38.2 Channel

class kombu.transport.redis.**Channel**(**args*, ***kwargs*)
Redis Channel.

class QoS(**args*, ***kwargs*)
Redis Ack Emulation.

ack(*delivery_tag*)
Acknowledge message and remove from transactional state.

append(*message*, *delivery_tag*)
Append message to transactional state.

pipe_or_acquire(*pipe=None*, *client=None*)

reject(*delivery_tag*, *requeue=False*)
Remove from transactional state and requeue message.

restore_at_shutdown = True

restore_by_tag(*tag*, *client=None*, *leftmost=False*)

restore_unacked(*client=None*)
Restore all unacknowledged messages.

restore_visible(*start=0*, *num=10*, *interval=10*)
Restore any pending unackwnowledged messages.

To be filled in for visibility_timeout style implementations.

Note: This is implementation optional, and currently only used by the Redis transport.

unacked_index_key

unacked_key

unacked_mutex_expire

unacked_mutex_key

visibility_timeout

ack_emulation = True

property active_queues
Set of queues being consumed from (excluding fanout queues).

property async_pool

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

basic_consume(*queue*, **args*, ***kwargs*)
Consume from *queue*.

client
Client used to publish messages, BRPOP etc.

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

conn_or_acquire(*client=None*)

connection_class = None

fanout_patterns = True
If enabled the fanout exchange will support patterns in routing and binding keys (like a topic exchange but using PUB/SUB).

Enabled by default since Kombu 4.x. Disable for backwards compatibility with Kombu 3.x.

fanout_prefix = True
Transport option to disable fanout keyprefix. Can also be string, in which case it changes the default prefix ('/{db}.') into to something else. The prefix must include a leading slash and a trailing dot.

Enabled by default since Kombu 4.x. Disable for backwards compatibility with Kombu 3.x.

from_transport_options = ('body_encoding', 'deadletter_queue', 'sep', 'ack_emulation',

get_table(*exchange*)
Get table of bindings for *exchange*.

keyprefix_fanout = '/{db}.'

keyprefix_queue = '_kombu.binding.%s'

max_connections = 10

property pool

priority(*n*)

priority_steps = [0, 3, 6, 9]

queue_order_strategy = 'round_robin'
Order in which we consume from queues.

Can be either string alias, or a cycle strategy class

- `round_robin` (*round_robin_cycle*).

  Make sure each queue has an equal opportunity to be consumed from.

- `sorted` (*sorted_cycle*).

  Consume from queues in alphabetical order. If the first queue in the sorted list always contains messages, then the rest of the queues will never be consumed from.

- `priority` (*priority_cycle*).

  Consume from queues in original order, so that if the first queue always contains messages, the rest of the queues in the list will never be consumed from.

The default is to consume from queues in round robin.

sep = '\x06\x16'

socket_connect_timeout = None

socket_keepalive = None

socket_keepalive_options = None

socket_timeout = None

subclient
Pub/Sub connection used to consume fanout queues.

supports_fanout = True

unacked_index_key = 'unacked_index'

unacked_key = 'unacked'

unacked_mutex_expire = 300

unacked_mutex_key = 'unacked_mutex'

unacked_restore_limit = None

visibility_timeout = 3600

4.39 MongoDB Transport - kombu.transport.mongodb

MongoDB transport.

copyright

(c) 2010 - 2013 by Flavio Percoco Premoli.

license BSD, see LICENSE for more details.

4.39.1 Transport

class kombu.transport.mongodb.**Transport**(*client*, ***kwargs*)

MongoDB Transport.

class Channel(**vargs*, ***kwargs*)

MongoDB Channel.

broadcast

broadcast_collection = 'messages.broadcast'

calc_queue_size = True

capped_queue_size = 100000

client

connect_timeout = None

default_database = 'kombu_default'

default_hostname = '127.0.0.1'

default_port = 27017

from_transport_options = ('body_encoding', 'deadletter_queue', 'connect_timeout',

get_now()

Return current time in UTC.

get_table(*exchange*)

Get table of bindings for *exchange*.

messages

messages_collection = 'messages'

prepare_queue_arguments(*arguments*, ***kwargs*)

queue_delete(*queue*, ***kwargs*)

Delete queue.

queues

```
        queues_collection = 'messages.queues'
        routing
        routing_collection = 'messages.routing'
        ssl = False
        supports_fanout = True
        ttl = False
    can_parse_url = True
    channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'pymongo.errors.Conne
    connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'pymongo.errors
    default_port = 27017
    driver_name = 'pymongo'
    driver_type = 'mongodb'
    driver_version()
    implements = {'asynchronous':  False, 'exchange_type':  frozenset({'direct', 'fanout',
    polling_interval = 1
```

4.39.2 Channel

class kombu.transport.mongodb.**Channel**(**vargs*, ***kwargs*)

MongoDB Channel.

broadcast

broadcast_collection = 'messages.broadcast'

calc_queue_size = True

capped_queue_size = 100000

client

connect_timeout = None

default_database = 'kombu_default'

default_hostname = '127.0.0.1'

default_port = 27017

from_transport_options = ('body_encoding', 'deadletter_queue', 'connect_timeout', 'ssl

get_now()

Return current time in UTC.

get_table(*exchange*)

Get table of bindings for *exchange*.

messages

messages_collection = 'messages'

prepare_queue_arguments(*arguments*, ***kwargs*)

queue_delete(*queue*, ***kwargs*)
Delete queue.

queues

queues_collection = 'messages.queues'

routing

routing_collection = 'messages.routing'

ssl = False

supports_fanout = True

ttl = False

4.40 Consul Transport - kombu.transport.consul

Consul Transport.

It uses Consul.io's Key/Value store to transport messages in Queues

It uses python-consul for talking to Consul's HTTP API

4.40.1 Transport

class kombu.transport.consul.**Transport**(**args*, ***kwargs*)
Consul K/V storage Transport for Kombu.

class Channel(**args*, ***kwargs*)
Consul Channel class which talks to the Consul Key/Value store.

index = None

lock_name

prefix = 'kombu'

session_ttl = 30

timeout = '10s'

default_port = 8500

driver_name = 'consul'

driver_type = 'consul'

driver_version()

verify_connection(*connection*)

4.40.2 Channel

class kombu.transport.consul.**Channel**(**args*, ***kwargs*)
Consul Channel class which talks to the Consul Key/Value store.

index = None

lock_name

prefix = 'kombu'

session_ttl = 30

timeout = '10s'

4.41 Etcd Transport - kombu.transport.etcd

Etcd Transport.

It uses Etcd as a store to transport messages in Queues

It uses python-etcd for talking to Etcd's HTTP API

4.41.1 Transport

class kombu.transport.etcd.**Transport**(**args*, ***kwargs*)
Etcd storage Transport for Kombu.

class Channel(**args*, ***kwargs*)
Etcd Channel class which talks to the Etcd.

index = None

lock_ttl = 10

lock_value

prefix = 'kombu'

session_ttl = 30

timeout = 10

default_port = 2379

driver_name = 'python-etcd'

driver_type = 'etcd'

driver_version()
Return the version of the etcd library.

> **Note:** python-etcd has no __version__. This is a workaround.

implements = {'asynchronous': False, 'exchange_type': frozenset({'direct'}), 'heartb

polling_interval = 3

verify_connection(*connection*)
Verify the connection works.

4.41.2 Channel

class kombu.transport.etcd.**Channel**(**args*, ***kwargs*)
Etcd Channel class which talks to the Etcd.

index = None

lock_ttl = 10

lock_value

prefix = 'kombu'

session_ttl = 30

timeout = 10

4.42 Zookeeper Transport - kombu.transport.zookeeper

Zookeeper transport.

copyright

(c) 2010 - 2013 by Mahendra M.

license BSD, see LICENSE for more details.

Synopsis

Connects to a zookeeper node as <server>:<port>/<vhost> The <vhost> becomes the base for all the other znodes. So we can use it like a vhost.

This uses the built-in kazoo recipe for queues

References

- https://zookeeper.apache.org/doc/trunk/recipes.html#sc_recipes_Queues
- https://kazoo.readthedocs.io/en/latest/api/recipe/queue.html

Limitations This queue does not offer reliable consumption. An entry is removed from the queue prior to being processed. So if an error occurs, the consumer has to re-queue the item or it will be lost.

4.42.1 Transport

class kombu.transport.zookeeper.**Transport**(**args*, ***kwargs*)
Zookeeper Transport.

class Channel(*connection*, ***kwargs*)
Zookeeper Channel.

property client

channel_errors = (<class 'amqp.exceptions.ChannelError'>,)

connection_errors = (<class 'amqp.exceptions.ConnectionError'>,)

default_port = 2181

driver_name = 'kazoo'

driver_type = 'zookeeper'

driver_version()

polling_interval = 1

4.42.2 Channel

class kombu.transport.zookeeper.**Channel**(*connection*, ***kwargs*)
Zookeeper Channel.

property client

4.43 File-system Transport - kombu.transport.filesystem

File-system Transport.

Transport using the file-system as the message store.

4.43.1 Transport

class kombu.transport.filesystem.**Transport**(*client*, ***kwargs*)
Filesystem Transport.

class Channel(*connection*, ***kwargs*)
Filesystem Channel.

data_folder_in

data_folder_out

processed_folder

store_processed

property transport_options

default_port = 0

driver_name = 'filesystem'

driver_type = 'filesystem'

driver_version()

4.43.2 Channel

class kombu.transport.filesystem.**Channel**(*connection*, ***kwargs*)
Filesystem Channel.

data_folder_in

data_folder_out

processed_folder

store_processed

property transport_options

4.44 SQLAlchemy Transport Model - kombu.transport.sqlalchemy

4.45 SQLAlchemy Transport Model - kombu.transport.sqlalchemy.models

4.46 Amazon SQS Transport - kombu.transport.SQS

Amazon SQS Transport.

Amazon SQS transport module for Kombu. This package implements an AMQP-like interface on top of Amazons SQS service, with the goal of being optimized for high performance and reliability.

The default settings for this module are focused now on high performance in task queue situations where tasks are small, idempotent and run very fast.

SQS Features supported by this transport:

Long Polling: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-long-polling.html

Long polling is enabled by setting the *wait_time_seconds* transport option to a number > 1. Amazon supports up to 20 seconds. This is enabled with 10 seconds by default.

Batch API Actions: https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/sqs-batch-api.html

The default behavior of the SQS Channel.drain_events() method is to request up to the 'prefetch_count' messages on every request to SQS. These messages are stored locally in a deque object and passed back to the Transport until the deque is empty, before triggering a new API call to Amazon.

This behavior dramatically speeds up the rate that you can pull tasks from SQS when you have short-running tasks (or a large number of workers).

When a Celery worker has multiple queues to monitor, it will pull down up to 'prefetch_count' messages from queueA and work on them all before moving on to queueB. If queueB is empty, it will wait up until 'polling_interval' expires before moving back and checking on queueA.

4.46.1 Transport

class kombu.transport.SQS.**Transport**(*client*, ***kwargs*)

SQS Transport.

Additional queue attributes can be supplied to SQS during queue creation by passing an `sqs-creation-attributes` key in transport_options. `sqs-creation-attributes` must be a dict whose key-value pairs correspond with Attributes in the CreateQueue SQS API.

For example, to have SQS queues created with server-side encryption enabled using the default Amazon Managed Customer Master Key, you can set `KmsMasterKeyId` Attribute. When the queue is initially created by Kombu, encryption will be enabled.

```
from kombu.transport.SQS import Transport

transport = Transport(
    ...,
    transport_options={
        'sqs-creation-attributes': {
            'KmsMasterKeyId': 'alias/aws/sqs',
        },
    }
)
```

class Channel(**args*, ***kwargs*)

SQS Channel.

property asynsqs

basic_ack(*delivery_tag*, *multiple=False*)

Acknowledge message.

basic_cancel(*consumer_tag*)

Cancel consumer by consumer tag.

basic_consume(*queue*, *no_ack*, **args*, ***kwargs*)

Consume from *queue*.

canonical_queue_name(*queue_name*)

close()

Close channel.

Cancel all consumers, and requeue unacked messages.

property conninfo

default_region = 'us-east-1'

default_visibility_timeout = 1800

default_wait_time_seconds = 10

domain_format = 'kombu%(vhost)s'

drain_events(*timeout=None*, *callback=None*, ***kwargs*)
Return a single payload message from one of our queues.

Raises **Queue.Empty** – if no messages available.

endpoint_url

entity_name(*name*, *table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 46: 45, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)
Format AMQP queue name into a legal SQS queue name.

is_secure

port

queue_name_prefix

region

regioninfo

property sqs

supports_fanout

property transport_options

visibility_timeout

wait_time_seconds

channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'kombu.asynchronous.a

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'kombu.asynchro

property default_connection_params

default_port = None

driver_name = 'sqs'

driver_type = 'sqs'

implements = {'asynchronous': True, 'exchange_type': frozenset({'direct'}), 'heartbe

polling_interval = 1

wait_time_seconds = 0

4.46.2 Channel

class kombu.transport.SQS.**Channel**(**args*, ***kwargs*)
SQS Channel.

property asynsqs

basic_ack(*delivery_tag*, *multiple=False*)
Acknowledge message.

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

basic_consume(*queue*, *no_ack*, **args*, ***kwargs*)
Consume from *queue*.

canonical_queue_name(*queue_name*)

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

property conninfo

default_region = 'us-east-1'

default_visibility_timeout = 1800

default_wait_time_seconds = 10

domain_format = 'kombu%(vhost)s'

drain_events(*timeout=None*, *callback=None*, ***kwargs*)
Return a single payload message from one of our queues.

Raises **Queue.Empty** – if no messages available.

endpoint_url

entity_name(*name*, *table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 46: 45, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)
Format AMQP queue name into a legal SQS queue name.

is_secure

port

queue_name_prefix

region

regioninfo

property sqs

supports_fanout

property transport_options

visibility_timeout

wait_time_seconds

4.47 SLMQ Transport - kombu.transport.SLMQ

SoftLayer Message Queue transport.

- *Transport*
- *Channel*

4.47.1 Transport

class kombu.transport.SLMQ.**Transport**(*client*, ***kwargs*)
SLMQ Transport.

class Channel(**args*, ***kwargs*)
SLMQ Channel.

basic_ack(*delivery_tag*)
Acknowledge message.

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

basic_consume(*queue*, *no_ack*, **args*, ***kwargs*)
Consume from *queue*.

property conninfo

default_visibility_timeout = 1800

delete_message(*queue*, *message_id*)

domain_format = 'kombu%(vhost)s'

entity_name(*name*, *table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)
Format AMQP queue name into a valid SLQS queue name.

queue_name_prefix

property slmq

property transport_options

visibility_timeout

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, None, <class 'OSError'

default_port = None

polling_interval = 1

4.47.2 Channel

class kombu.transport.SLMQ.**Channel**(**args*, ***kwargs*)
SLMQ Channel.

basic_ack(*delivery_tag*)
Acknowledge message.

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

basic_consume(*queue*, *no_ack*, **args*, ***kwargs*)
Consume from *queue*.

property conninfo

default_visibility_timeout = 1800

delete_message(*queue*, *message_id*)

domain_format = 'kombu%(vhost)s'

entity_name(*name*, *table={33: 95, 34: 95, 35: 95, 36: 95, 37: 95, 38: 95, 39: 95, 40: 95, 41: 95, 42: 95, 43: 95, 44: 95, 45: 95, 46: 95, 47: 95, 58: 95, 59: 95, 60: 95, 61: 95, 62: 95, 63: 95, 64: 95, 91: 95, 92: 95, 93: 95, 94: 95, 96: 95, 123: 95, 124: 95, 125: 95, 126: 95}*)
Format AMQP queue name into a valid SLQS queue name.

queue_name_prefix

property slmq

property transport_options

visibility_timeout

4.48 Pyro Transport - kombu.transport.pyro

Pyro transport, and Kombu Broker daemon.

Requires the `Pyro4` library to be installed.

To use the Pyro transport with Kombu, use an url of the form: `pyro://localhost/kombu.broker`

The hostname is where the transport will be looking for a Pyro name server, which is used in turn to locate the kombu.broker Pyro service. This broker can be launched by simply executing this transport module directly, with the command: `python -m kombu.transport.pyro`

- *Transport*
- *Channel*
- *KombuBroker*

4.48.1 Transport

class kombu.transport.pyro.**Transport**(*client*, ***kwargs*)
Pyro Transport.

class Channel(*connection*, ***kwargs*)
Pyro Channel.

after_reply_message_received(*queue*)
Callback called after RPC reply received.

Notes

Reply queue semantics: can be used to delete the queue after transient reply message received.

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

queues()

shared_queues

default_port = 9090

driver_name = 'pyro'

driver_type = 'pyro'

driver_version()

shared_queues

state = <kombu.transport.virtual.base.BrokerState object>
memory backend state is global.

4.48.2 Channel

class kombu.transport.pyro.**Channel**(*connection*, ***kwargs*)
Pyro Channel.

after_reply_message_received(*queue*)
Callback called after RPC reply received.

Notes

Reply queue semantics: can be used to delete the queue after transient reply message received.

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

queues()

shared_queues

4.48.3 KombuBroker

4.49 Transport Base Class - kombu.transport.base

Base transport interface.

- *Message*
- *Transport*

4.49.1 Message

class kombu.transport.base.**Message**(*body=None*, *delivery_tag=None*, *content_type=None*, *content_encoding=None*, *delivery_info=None*, *properties=None*, *headers=None*, *postencode=None*, *accept=None*, *channel=None*, ***kwargs*)
Base class for received messages.

Keyword Arguments

- **channel** (*ChannelT*) – If message was received, this should be the channel that the message was received on.

- **body** (*str*) – Message body.
- **delivery_mode** (*bool*) – Set custom delivery mode. Defaults to `delivery_mode`.
- **priority** (*int*) – Message priority, 0 to broker configured max priority, where higher is better.
- **content_type** (*str*) – The messages content_type. If content_type is set, no serialization occurs as it is assumed this is either a binary object, or you've done your own serialization. Leave blank if using built-in serialization as our library properly sets content_type.
- **content_encoding** (*str*) – The character set in which this object is encoded. Use "binary" if sending in raw binary objects. Leave blank if using built-in serialization as our library properly sets content_encoding.
- **properties** (*Dict*) – Message properties.
- **headers** (*Dict*) – Message headers.

payload
The decoded message body.

channel

delivery_tag

content_type

content_encoding

delivery_info

headers

properties

body

acknowledged
Set to true if the message has been acknowledged.

ack(*multiple=False*)
Acknowledge this message as being processed.

This will remove the message from the queue.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

reject(*requeue=False*)
Reject this message.

The message will be discarded by the server.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

requeue()
Reject this message and put it back on the queue.

> **Warning:** You must not use this method as a means of selecting messages to process.

Raises *MessageStateError* – If the message has already been acknowledged/requeued/rejected.

decode()
Deserialize the message body.

Returning the original python structure sent by the publisher.

Note: The return value is memoized, use *_decode* to force re-evaluation.

4.49.2 Transport

class kombu.transport.base.**Transport**(*client*, ***kwargs*)
Base class for transports.

client = None
The *Connection* owning this instance.

default_port = None
Default port used when no port has been specified.

recoverable_connection_errors
Optional list of connection related exceptions that can be recovered from, but where the connection must be closed and re-established first.

If not defined then all *connection_errors* and *channel_errors* will be regarded as recoverable, but needing to close the connection first.

recoverable_channel_errors
Optional list of channel related exceptions that can be automatically recovered from without re-establishing the connection.

connection_errors = (<class 'amqp.exceptions.ConnectionError'>,)
Tuple of errors that can happen due to connection failure.

channel_errors = (<class 'amqp.exceptions.ChannelError'>,)
Tuple of errors that can happen due to channel/method failure.

establish_connection()

close_connection(*connection*)

create_channel(*connection*)

close_channel(*connection*)

drain_events(*connection*, ***kwargs*)

4.50 Virtual Transport Base Class - kombu.transport.virtual

- *Quality Of Service*
- *In-memory State*

4.50.1 Transports

class kombu.transport.virtual.**Transport**(*client*, ***kwargs*)

Virtual transport.

Parameters **client** (kombu.Connection) – The client this is a transport for.

Channel = <class 'kombu.transport.virtual.base.Channel'>

Cycle = <class 'kombu.utils.scheduling.FairCycle'>

polling_interval = 1.0

Time to sleep between unsuccessful polls.

default_port = None

port number used when no port is specified.

state = <kombu.transport.virtual.base.BrokerState object>

Global *BrokerState* containing declared exchanges and bindings.

cycle = None

FairCycle instance used to fairly drain events from channels (set by constructor).

establish_connection()

close_connection(*connection*)

create_channel(*connection*)

close_channel(*channel*)

drain_events(*connection*, *timeout=None*)

4.50.2 Channel

class kombu.transport.virtual.**AbstractChannel**

Abstract channel interface.

This is an abstract class defining the channel methods you'd usually want to implement in a virtual channel.

Note: Do not subclass directly, but rather inherit from *Channel*.

class kombu.transport.virtual.**Channel**(*connection*, ***kwargs*)

Virtual channel.

Parameters **connection** (*ConnectionT*) – The transport instance this channel is part of.

Message = <class 'kombu.transport.virtual.base.Message'>

message class used.

state

Broker state containing exchanges and bindings.

qos

QoS manager for this channel.

do_restore = True

flag to restore unacked messages when channel goes out of scope.

exchange_types = {'direct': <class 'kombu.transport.virtual.exchange.DirectExchange'>
mapping of exchange types and corresponding classes.

exchange_declare(*exchange=None*, *type='direct'*, *durable=False*, *auto_delete=False*, *arguments=None*, *nowait=False*, *passive=False*)
Declare exchange.

exchange_delete(*exchange*, *if_unused=False*, *nowait=False*)
Delete *exchange* and all its bindings.

queue_declare(*queue=None*, *passive=False*, ***kwargs*)
Declare queue.

queue_delete(*queue*, *if_unused=False*, *if_empty=False*, ***kwargs*)
Delete queue.

queue_bind(*queue*, *exchange=None*, *routing_key=''*, *arguments=None*, ***kwargs*)
Bind *queue* to *exchange* with *routing key*.

queue_purge(*queue*, ***kwargs*)
Remove all ready messages from queue.

basic_publish(*message*, *exchange*, *routing_key*, ***kwargs*)
Publish message.

basic_consume(*queue*, *no_ack*, *callback*, *consumer_tag*, ***kwargs*)
Consume from *queue*.

basic_cancel(*consumer_tag*)
Cancel consumer by consumer tag.

basic_get(*queue*, *no_ack=False*, ***kwargs*)
Get message by direct access (synchronous).

basic_ack(*delivery_tag*, *multiple=False*)
Acknowledge message.

basic_recover(*requeue=False*)
Recover unacked messages.

basic_reject(*delivery_tag*, *requeue=False*)
Reject message.

basic_qos(*prefetch_size=0*, *prefetch_count=0*, *apply_global=False*)
Change QoS settings for this channel.

Note: Only *prefetch_count* is supported.

---

get_table(*exchange*)
Get table of bindings for *exchange*.

typeof(*exchange*, *default='direct'*)
Get the exchange type instance for *exchange*.

drain_events(*timeout=None*, *callback=None*)

prepare_message(*body*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*)
Prepare message data.

message_to_python(*raw_message*)
Convert raw message to `Message` instance.

flow(*active=True*)
Enable/disable message flow.

Raises **NotImplementedError** – as flow is not implemented by the base virtual implementation.

close()
Close channel.

Cancel all consumers, and requeue unacked messages.

4.50.3 Message

class kombu.transport.virtual.**Message**(*payload*, *channel=None*, ***kwargs*)
Message object.

exception MessageStateError
The message has already been acknowledged.

args

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

accept

ack(*multiple=False*)
Acknowledge this message as being processed.

This will remove the message from the queue.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

ack_log_error(*logger*, *errors*, *multiple=False*)

property acknowledged
Set to true if the message has been acknowledged.

body

channel

content_encoding

content_type

decode()
Deserialize the message body.

Returning the original python structure sent by the publisher.

Note: The return value is memoized, use *_decode* to force re-evaluation.

delivery_info

delivery_tag

errors = None

headers

property payload
The decoded message body.

properties

reject(*requeue=False*)
Reject this message.

The message will be discarded by the server.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

reject_log_error(*logger*, *errors*, *requeue=False*)

requeue()
Reject this message and put it back on the queue.

> **Warning:** You must not use this method as a means of selecting messages to process.

Raises ***MessageStateError*** – If the message has already been acknowledged/requeued/rejected.

serializable()

4.50.4 Quality Of Service

class kombu.transport.virtual.**QoS**(*channel*, *prefetch_count=0*)
Quality of Service guarantees.

Only supports *prefetch_count* at this point.

Parameters

- **channel** (*ChannelT*) – Connection channel.
- **prefetch_count** (*int*) – Initial prefetch count (defaults to 0).

ack(*delivery_tag*)
Acknowledge message and remove from transactional state.

append(*message*, *delivery_tag*)
Append message to transactional state.

can_consume()
Return true if the channel can be consumed from.

Used to ensure the client adhers to currently active prefetch limits.

can_consume_max_estimate()
Return the maximum number of messages allowed to be returned.

Returns an estimated number of messages that a consumer may be allowed to consume at once from the broker. This is used for services where bulk 'get message' calls are preferred to many individual 'get message' calls - like SQS.

Returns greater than zero.

Return type int

get(*delivery_tag*)

prefetch_count = 0
current prefetch count value

reject(*delivery_tag*, *requeue=False*)
Remove from transactional state and requeue message.

restore_at_shutdown = True
If disabled, unacked messages won't be restored at shutdown.

restore_unacked()
Restore all unacknowledged messages.

restore_unacked_once(*stderr=None*)
Restore all unacknowledged messages at shutdown/gc collect.

Note: Can only be called once for each instance, subsequent calls will be ignored.

---

restore_visible(**args*, ***kwargs*)
Restore any pending unackwnowledged messages.

To be filled in for visibility_timeout style implementations.

Note: This is implementation optional, and currently only used by the Redis transport.

4.50.5 In-memory State

class kombu.transport.virtual.**BrokerState**(*exchanges=None*)
Broker state holds exchanges, queues and bindings.

binding_declare(*queue*, *exchange*, *routing_key*, *arguments*)

binding_delete(*queue*, *exchange*, *routing_key*)

bindings = None
This is the actual bindings registry, used to store bindings and to test 'in' relationships in constant time. It has the following structure:

```
{
    (queue, exchange, routing_key): arguments,
    # ...,
}
```

clear()

exchanges = None
Mapping of exchange name to *kombu.transport.virtual.exchange.ExchangeType*

has_binding(*queue*, *exchange*, *routing_key*)

queue_bindings(*queue*)

queue_bindings_delete(*queue*)

queue_index = None
The queue index is used to access directly (constant time) all the bindings of a certain queue. It has the following structure:

```
{
    queue: {
        (queue, exchange, routing_key),
        # ...,
    },
    # ...,
}
```

4.51 Virtual AMQ Exchange Implementation - kombu.transport.virtual.exchange

Virtual AMQ Exchange.

Implementations of the standard exchanges defined by the AMQ protocol (excluding the *headers* exchange).

- *Direct*
- *Topic*
- *Fanout*
- *Interface*

4.51.1 Direct

class kombu.transport.virtual.exchange.**DirectExchange**(*channel*)

Direct exchange.

The *direct* exchange routes based on exact routing keys.

deliver(*message*, *exchange*, *routing_key*, ***kwargs*)

lookup(*table*, *exchange*, *routing_key*, *default*)

Lookup all queues matching *routing_key* in *exchange*.

Returns queue name, or 'default' if no queues matched.

Return type str

type = 'direct'

4.51.2 Topic

class kombu.transport.virtual.exchange.**TopicExchange**(*channel*)

Topic exchange.

The *topic* exchange routes messages based on words separated by dots, using wildcard characters * (any single word), and # (one or more words).

deliver(*message*, *exchange*, *routing_key*, ***kwargs*)

key_to_pattern(*rkey*)

Get the corresponding regex for any routing key.

lookup(*table*, *exchange*, *routing_key*, *default*)
Lookup all queues matching *routing_key* in *exchange*.

Returns queue name, or 'default' if no queues matched.

Return type str

prepare_bind(*queue*, *exchange*, *routing_key*, *arguments*)
Prepare queue-binding.

Returns

of *(routing_key, regex, queue)* to be stored for bindings to this exchange.

Return type Tuple[str, Pattern, str]

type = 'topic'

wildcards = {'#': '.*?', '*': '.*?[^\\.]'}
map of wildcard to regex conversions

4.51.3 Fanout

class kombu.transport.virtual.exchange.**FanoutExchange**(*channel*)
Fanout exchange.

The *fanout* exchange implements broadcast messaging by delivering copies of all messages to all queues bound to the exchange.

To support fanout the virtual channel needs to store the table as shared state. This requires that the *Channel.supports_fanout* attribute is set to true, and the *Channel._queue_bind* and *Channel.get_table* methods are implemented.

See also:

the redis backend for an example implementation of these methods.

deliver(*message*, *exchange*, *routing_key*, ***kwargs*)

lookup(*table*, *exchange*, *routing_key*, *default*)
Lookup all queues matching *routing_key* in *exchange*.

Returns queue name, or 'default' if no queues matched.

Return type str

type = 'fanout'

4.51.4 Interface

class kombu.transport.virtual.exchange.**ExchangeType**(*channel*)
Base class for exchanges.

Implements the specifics for an exchange type.
Parameters **channel** (*ChannelT*) – AMQ Channel.
equivalent(*prev*, *exchange*, *type*, *durable*, *auto_delete*, *arguments*)
Return true if *prev* and *exchange* is equivalent.

lookup(*table*, *exchange*, *routing_key*, *default*)
Lookup all queues matching *routing_key* in *exchange*.

Returns queue name, or 'default' if no queues matched.

Return type str

prepare_bind(*queue*, *exchange*, *routing_key*, *arguments*)
Prepare queue-binding.

Returns

of *(routing_key, regex, queue)* to be stored for bindings to this exchange.

Return type Tuple[str, Pattern, str]

type = None

4.52 Message Serialization - kombu

Serialization utilities.

4.52.1 Overview

Centralized support for encoding/decoding of data structures. Contains json, pickle, msgpack, and yaml serializers.

Optionally installs support for YAML if the PyYAML package is installed.

Optionally installs support for msgpack if the msgpack-python package is installed.

4.52.2 Exceptions

exception kombu.serialization.**SerializerNotInstalled**
Support for the requested serialization type is not installed.

4.52.3 Serialization

kombu.serialization.**dumps**(*data*, *serializer=None*)
Encode data.

Serialize a data structure into a string suitable for sending as an AMQP message body.

Parameters

- **data** (*List, Dict, str*) – The message data to send.
- **serializer** (*str*) – An optional string representing the serialization method you want the data marshalled into. (For example, *json*, *raw*, or *pickle*).

  If None (default), then json will be used, unless *data* is a str or unicode object. In this latter case, no serialization occurs as it would be unnecessary.

Note that if *serializer* is specified, then that serialization method will be used even if a `str` or `unicode` object is passed in.

Returns A three-item tuple containing the content type (e.g., *application/json*), content encoding, (e.g., *utf-8*) and a string containing the serialized data.

Return type Tuple[str, str, str]

Raises ***SerializerNotInstalled*** – If the serialization method requested is not available.

`kombu.serialization.`**`loads`**(*data*, *content_type*, *content_encoding*, *accept=None*, *force=False*, *_trusted_content=frozenset({'application/data', 'application/text'})*)

Decode serialized data.

Deserialize a data stream as serialized using *dumps* based on *content_type*.

Parameters

- **`data`** (*bytes, buffer, str*) – The message data to deserialize.
- **`content_type`** (*str*) – The content-type of the data. (e.g., *application/json*).
- **`content_encoding`** (*str*) – The content-encoding of the data. (e.g., *utf-8*, *binary*, or *us-ascii*).
- **`accept`** (*Set*) – List of content-types to accept.

Raises ***ContentDisallowed*** – If the content-type is not accepted.

Returns The unserialized data.

Return type Any

`kombu.serialization.`**`raw_encode`**(*data*)

Special case serializer.

4.52.4 Registry

`kombu.serialization.`**`register`**(*name*, *encoder*, *decoder*, *content_type*, *content_encoding='utf-8'*)

Register a new encoder/decoder.

Parameters

- **`name`** (*str*) – A convenience name for the serialization method.
- **`encoder`** (*callable*) – A method that will be passed a python data structure and should return a string representing the serialized data. If `None`, then only a decoder will be registered. Encoding will not be possible.
- **`decoder`** (`Callable`) – A method that will be passed a string representing serialized data and should return a python data structure. If `None`, then only an encoder will be registered. Decoding will not be possible.
- **`content_type`** (*str*) – The mime-type describing the serialized structure.
- **`content_encoding`** (*str*) – The content encoding (character set) that the *decoder* method will be returning. Will usually be *utf-8*, *us-ascii*, or *binary*.

`kombu.serialization.`**`unregister`**(*name*)

Unregister registered encoder/decoder.

Parameters **`name`** (*str*) – Registered serialization method name.

Raises ***SerializerNotInstalled*** – If a serializer by that name cannot be found.

`kombu.serialization.`**`registry = <kombu.serialization.SerializerRegistry object>`**

Global registry of serializers/deserializers.

4.53 Generic RabbitMQ manager - kombu.utils.amq_manager

AMQP Management API utilities.

kombu.utils.amq_manager.**get_manager**(*client*, *hostname=None*, *port=None*, *userid=None*, *password=None*)

Get pyrabbit manager.

4.54 Custom Collections - kombu.utils.collections

Custom maps, sequences, etc.

class kombu.utils.collections.**EqualityDict**

Dict using the eq operator for keying.

class kombu.utils.collections.**HashedSeq**(**seq*)

Hashed Sequence.

Type used for hash() to make sure the hash is not generated multiple times.

a

e

h

hashvalue

l

s

u

v

kombu.utils.collections.**eqhash**(*o*)

Call `obj.__eqhash__`.

4.55 Python Compatibility - kombu.utils.compat

Python Compatibility Utilities.

kombu.utils.compat.**coro**(*gen*)

Decorator to mark generator as co-routine.

kombu.utils.compat.**detect_environment**()

Detect the current environment: default, eventlet, or gevent.

kombu.utils.compat.**entrypoints**(*namespace*)

Return setuptools entrypoints for namespace.

kombu.utils.compat.**fileno**(*f*)

Get fileno from file-like object.

kombu.utils.compat.**maybe_fileno**(*f*)

Get object fileno, or `None` if not defined.

kombu.utils.compat.**nested**(**managers*)

Nest context managers.

4.56 Debugging Utilities - kombu.utils.debug

Debugging support.

kombu.utils.debug.**setup_logging**(*loglevel=10*, *loggers=None*)
Setup logging to stdout.

class kombu.utils.debug.**Logwrapped**(*instance*, *logger=None*, *ident=None*)
Wrap all object methods, to log on call.

4.57 Div Utilities - kombu.utils.div

Div. Utilities.

kombu.utils.div.**emergency_dump_state**(*state*, *open_file=<built-in function open>*, *dump=None*, *stderr=None*)
Dump message state to stdout or file.

4.58 String Encoding Utilities - kombu.utils.encoding

Text encoding utilities.

Utilities to encode text, and to safely emit text from running applications without crashing from the infamous UnicodeDecodeError exception.

kombu.utils.encoding.**bytes_to_str**(*s*)
Convert bytes to str.

kombu.utils.encoding.**default_encode**(*obj*)
Encode using default encoding.

kombu.utils.encoding.**default_encoding**(*file=None*)
Get default encoding.

kombu.utils.encoding.**default_encoding_file = None**
safe_str takes encoding from this file by default. *set_default_encoding_file()* can used to set the default output file.

kombu.utils.encoding.**ensure_bytes**(*s*)
Ensure s is bytes, not str.

kombu.utils.encoding.**from_utf8**(*s*, **args*, ***kwargs*)
Get str from utf-8 encoding.

kombu.utils.encoding.**get_default_encoding_file**()
Get file used to get codec information.

kombu.utils.encoding.**safe_repr**(*o*, *errors='replace'*)
Safe form of repr, void of Unicode errors.

kombu.utils.encoding.**safe_str**(*s*, *errors='replace'*)
Safe form of str(), void of unicode errors.

kombu.utils.encoding.**set_default_encoding_file**(*file*)
Set file used to get codec information.

kombu.utils.encoding.**str_to_bytes**(*s*)
Convert str to bytes.

4.59 Async I/O Selectors - kombu.utils.eventio

Selector Utilities.

kombu.utils.eventio.**poll**(**args*, ***kwargs*)
Create new poller instance.

4.60 Functional-style Utilities - kombu.utils.functional

Functional Utilities.

class kombu.utils.functional.**LRUCache**(*limit=None*)
LRU Cache implementation using a doubly linked list to track access.
Parameters **limit** (*int*) – The maximum number of keys to keep in the cache. When a new key is inserted and the limit has been exceeded, the *Least Recently Used* key will be discarded from the cache.

incr(*key*, *delta=1*)

items()

iteritems()

iterkeys()

itervalues()

keys()

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values()

kombu.utils.functional.**memoize**(*maxsize=None*, *keyfun=None*, *Cache=<class 'kombu.utils.functional.LRUCache'>*)
Decorator to cache function return value.

class kombu.utils.functional.**lazy**(*fun*, **args*, ***kwargs*)
Holds lazy evaluation.

Evaluated when called or if the *evaluate()* method is called. The function is re-evaluated on every call.
Overloaded operations that will evaluate the promise: __str__(), __repr__(), __cmp__().

evaluate()

kombu.utils.functional.**maybe_evaluate**(*value*)
Evaluate value only if value is a *lazy* instance.

kombu.utils.functional.**is_list**(*l*, *scalars=(<class 'collections.abc.Mapping'>, <class 'str'>)*, *iters=(<class 'collections.abc.Iterable'>,)*)
Return true if the object is iterable.

Note: Returns false if object is a mapping or string.

---

kombu.utils.functional.**maybe_list**(*l*, *scalars=(<class 'collections.abc.Mapping'>, <class 'str'>)*)
Return list of one element if `l` is a scalar.

kombu.utils.functional.**dictfilter**(*d=None*, ***kw*)
Remove all keys from dict `d` whose value is `None`.

4.61 Module Importing Utilities - kombu.utils.imports

Import related utilities.

kombu.utils.imports.**symbol_by_name**(*name*, *aliases=None*, *imp=None*, *package=None*, *sep='.'*, *default=None*, ***kwargs*)
Get symbol by qualified name.

The name should be the full dot-separated path to the class:

```
modulename.ClassName
```

Example:

```
celery.concurrency.processes.TaskPool
                            ^- class name
```

or using ':' to separate module and symbol:

```
celery.concurrency.processes:TaskPool
```

If *aliases* is provided, a dict containing short name/long name mappings, the name is looked up in the aliases first.

Examples

```
>>> symbol_by_name('celery.concurrency.processes.TaskPool')
<class 'celery.concurrency.processes.TaskPool'>
```

```
>>> symbol_by_name('default', {
...     'default': 'celery.concurrency.processes.TaskPool'})
<class 'celery.concurrency.processes.TaskPool'>
```

Does not try to look up non-string names. >>> from celery.concurrency.processes import TaskPool >>> symbol_by_name(TaskPool) is TaskPool True

4.62 JSON Utilities - kombu.utils.json

JSON Serialization Utilities.

class kombu.utils.json.**DjangoPromise**
Dummy object.

class kombu.utils.json.**JSONEncoder**(***, *skipkeys=False*, *ensure_ascii=True*, *check_circular=True*, *allow_nan=True*, *sort_keys=False*, *indent=None*, *separators=None*, *default=None*)
Kombu custom json encoder.

default(*o*, *dates=(<class 'datetime.datetime'>, <class 'datetime.date'>)*, *times=(<class 'datetime.time'>,)*, *textual=(<class 'decimal.Decimal'>, <class 'uuid.UUID'>, <class 'kombu.utils.json.DjangoPromise'>)*, *isinstance=<built-in function isinstance>*, *datetime=<class 'datetime.datetime'>*, *text_t=<class 'str'>*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

kombu.utils.json.**dumps**(*s*, *_dumps=<function dumps>*, *cls=None*, *default_kwargs=None*, ***kwargs*)

Serialize object to json string.

kombu.utils.json.**loads**(*s*, *_loads=<function loads>*, *decode_bytes=True*)

Deserialize json from string.

4.63 Rate limiting - kombu.utils.limits

Token bucket implementation for rate limiting.

class kombu.utils.limits.**TokenBucket**(*fill_rate*, *capacity=1*)

Token Bucket Algorithm.

See also:

https://en.wikipedia.org/wiki/Token_Bucket

Most of this code was stolen from an entry in the ASPN Python Cookbook: https://code.activestate.com/recipes/511490/

> **Warning:** Thread Safety: This implementation is not thread safe. Access to a *TokenBucket* instance should occur within the critical section of any multithreaded code.

add(*item*)

can_consume(*tokens=1*)

Check if one or more tokens can be consumed.

Returns

true if the number of tokens can be consumed from the bucket. If they can be consumed, a call will also consume the requested number of tokens from the bucket. Calls will only consume *tokens* (the number requested) or zero tokens – it will never consume a partial number of tokens.

Return type bool

capacity = 1
Maximum number of tokens in the bucket.

clear_pending()

expected_time(*tokens=1*)
Return estimated time of token availability.

Returns the time in seconds.

Return type float

fill_rate = None
The rate in tokens/second that the bucket will be refilled.

pop()

timestamp = None
Timestamp of the last time a token was taken out of the bucket.

4.64 Object/Property Utilities - kombu.utils.objects

Object Utilities.

class kombu.utils.objects.**cached_property**(*fget=None*, *fset=None*, *fdel=None*, *doc=None*)
Cached property descriptor.

Caches the return value of the get method on first call.

Examples

```
@cached_property
def connection(self):
    return Connection()

@connection.setter  # Prepares stored value
def connection(self, value):
    if value is None:
        raise TypeError('Connection must be a connection')
    return value

@connection.deleter
def connection(self, value):
    # Additional action to do at del(self.attr)
    if value is not None:
        print('Connection {0!r} deleted'.format(value)
```

deleter(*fdel*)

setter(*fset*)

4.65 Consumer Scheduling - kombu.utils.scheduling

Scheduling Utilities.

class kombu.utils.scheduling.**FairCycle**(*fun*, *resources*, *predicate=<class 'Exception'>*)
Cycle between resources.

Consume from a set of resources, where each resource gets an equal chance to be consumed from.

Parameters

- **fun** (Callable) – Callback to call.
- **resources** (*Sequence[Any]*) – List of resources.
- **predicate** (*type*) – Exception predicate.

close()
Close cycle.

get(*callback*, ***kwargs*)
Get from next resource.

class kombu.utils.scheduling.**priority_cycle**(*it=None*)
Cycle that repeats items in order.

rotate(*last_used*)
Unused in this implementation.

class kombu.utils.scheduling.**round_robin_cycle**(*it=None*)
Iterator that cycles between items in round-robin.

consume(*n*)
Consume n items.

rotate(*last_used*)
Move most recently used item to end of list.

update(*it*)
Update items from iterable.

class kombu.utils.scheduling.**sorted_cycle**(*it=None*)
Cycle in sorted order.

consume(*n*)
Consume n items.

4.66 Text utilitites - kombu.utils.text

Text Utilities.

kombu.utils.text.**escape_regex**(*p*, *white=''*)
Escape string for use within a regular expression.

kombu.utils.text.**fmatch_best**(*needle*, *haystack*, *min_ratio=0.6*)
Fuzzy match - Find best match (scalar).

kombu.utils.text.**fmatch_iter**(*needle*, *haystack*, *min_ratio=0.6*)
Fuzzy match: iteratively.

Yields *Tuple* – of ratio and key.

kombu.utils.text.**version_string_as_tuple**(*s*)
Convert version string to version info tuple.

4.67 Time Utilities - kombu.utils.time

Time Utilities.

kombu.utils.time.**maybe_s_to_ms**(*v*)
Convert seconds to milliseconds, but return None for None.

4.68 URL Utilities - kombu.utils.url

URL Utilities.

kombu.utils.url.**as_url**(*scheme*, *host=None*, *port=None*, *user=None*, *password=None*, *path=None*, *query=None*, *sanitize=False*, *mask='**'*)
Generate URL from component parts.

kombu.utils.url.**maybe_sanitize_url**(*url*, *mask='**'*)
Sanitize url, or do nothing if url undefined.

kombu.utils.url.**parse_url**(*url*)
Parse URL into mapping of components.

kombu.utils.url.**safequote**(*string*, ***, *safe=''*, *encoding=None*, *errors=None*)
quote('abc def') -> 'abc%20def'

Each part of a URL, e.g. the path info, the query, etc., has a different set of reserved characters that must be quoted.

RFC 3986 Uniform Resource Identifiers (URI): Generic Syntax lists the following reserved characters.
reserved = ";" | "/" | "?" | ":" | "@" | "&" | "=" | "+" | "$" | "," | "~"
Each of these characters is reserved in some component of a URL, but not necessarily in all of them.

Python 3.7 updates from using RFC 2396 to RFC 3986 to quote URL strings. Now, "~" is included in the set of reserved characters.

By default, the quote function is intended for quoting the path section of a URL. Thus, it will not encode '/'. This character is reserved, but in typical usage the quote function is being called on a path where the existing slash characters are used as reserved characters.

string and safe may be either str or bytes objects. encoding and errors must not be specified if string is a bytes object.

The optional encoding and errors parameters specify how to deal with non-ASCII characters, as accepted by the str.encode method. By default, encoding='utf-8' (characters are encoded with UTF-8), and errors='strict' (unsupported characters raise a UnicodeEncodeError).

kombu.utils.url.**sanitize_url**(*url*, *mask='**'*)
Return copy of URL with password removed.

kombu.utils.url.**url_to_parts**(*url*)
Parse URL into *urlparts* tuple of components.

class kombu.utils.url.**urlparts**(*scheme*, *hostname*, *port*, *username*, *password*, *path*, *query*)

property hostname
Alias for field number 1

property password
Alias for field number 4

property path
Alias for field number 5

property port
Alias for field number 2

property query
Alias for field number 6

property scheme
Alias for field number 0

property username
Alias for field number 3

4.69 UUID Utilities - kombu.utils.uuid

UUID utilities.

kombu.utils.uuid.**uuid**(*_uuid=<function uuid4>*)
Generate unique id in UUID4 format.

See also:

For now this is provided by uuid.uuid4().

4.70 Python 2 to Python 3 utilities - kombu.five

Python 2/3 compatibility.

Compatibility implementations of features only available in newer Python versions.

class kombu.five.**Counter**(***kwds*)
Dict subclass for counting hashable items. Sometimes called a bag or multiset. Elements are stored as dictionary keys and their counts are stored as dictionary values.

```
>>> c = Counter('abcdeabcdabcaba')  # count elements from a string
```

```
>>> c.most_common(3)                # three most common elements
[('a', 5), ('b', 4), ('c', 3)]
>>> sorted(c)                       # list all unique elements
['a', 'b', 'c', 'd', 'e']
>>> ''.join(sorted(c.elements()))   # list elements with repetitions
'aaaaabbbbcccdde'
>>> sum(c.values())                 # total of all counts
15
```

```
>>> c['a']                          # count of letter 'a'
5
>>> for elem in 'shazam':           # update counts from an iterable
...     c[elem] += 1                # by adding 1 to each element's count
>>> c['a']                          # now there are seven 'a'
7
>>> del c['b']                      # remove all 'b'
>>> c['b']                          # now there are zero 'b'
0
```

```
>>> d = Counter('simsalabim')       # make another counter
>>> c.update(d)                     # add in the second counter
>>> c['a']                          # now there are nine 'a'
9
```

```
>>> c.clear()                       # empty the counter
>>> c
Counter()
```

Note: If a count is set to zero or reduced to zero, it will remain in the counter until the entry is deleted or the counter is cleared:

```
>>> c = Counter('aaabbc')
>>> c['b'] -= 2                     # reduce the count of 'b' by two
>>> c.most_common()                 # 'b' is still in, but its count is zero
[('a', 3), ('c', 1), ('b', 0)]
```

copy()
Return a shallow copy.

elements()
Iterator over elements repeating each as many times as its count.

```
>>> c = Counter('ABCABC')
>>> sorted(c.elements())
['A', 'A', 'B', 'B', 'C', 'C']
```

Knuth's example for prime factors of 1836: 2**2 * 3**3 * 17**1 >>> prime_factors = Counter({2: 2, 3: 3, 17: 1}) >>> product = 1 >>> for factor in prime_factors.elements(): # loop over factors ... product *= factor # and multiply them >>> product 1836

Note, if an element's count has been set to zero or is a negative number, elements() will ignore it.

classmethod fromkeys(*iterable*, *v=None*)
Create a new dictionary with keys from iterable and values set to value.

most_common(*n=None*)
List the n most common elements and their counts from the most common to the least. If n is None, then list all element counts.

```
>>> Counter('abcdeabcdabcaba').most_common(3)
[('a', 5), ('b', 4), ('c', 3)]
```

subtract(***kwds*)
Like dict.update() but subtracts counts instead of replacing them. Counts can be reduced below zero. Both the inputs and outputs are allowed to contain zero and negative counts.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.subtract('witch')             # subtract elements from another iterable
>>> c.subtract(Counter('watch'))    # subtract elements from another counter
>>> c['h']                          # 2 in which, minus 1 in witch, minus 1␣
↪in watch
0
>>> c['w']                          # 1 in which, minus 1 in witch, minus 1␣
↪in watch
-1
```

update(***kwds*)
Like dict.update() but add counts instead of replacing them.

Source can be an iterable, a dictionary, or another Counter instance.

```
>>> c = Counter('which')
>>> c.update('witch')           # add elements from another iterable
>>> d = Counter('watch')
>>> c.update(d)                 # add elements from another counter
>>> c['h']                      # four 'h' in which, witch, and watch
4
```

kombu.five.**reload**(*module*)
Reload the module and return it.

The module must have been successfully imported before.

class kombu.five.**UserList**(*initlist=None*)
A more or less complete user-defined wrapper around list objects.

append(*item*)
S.append(value) – append value to the end of the sequence

clear() → None – remove all items from S

copy()

count(*value*) → integer – return number of occurrences of value

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

index(*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

Supporting start and stop arguments is optional, but recommended.

insert(*i*, *item*)
S.insert(index, value) – insert value before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

sort(**args*, ***kwds*)

class kombu.five.**UserDict**(***kwargs*)

copy()

classmethod fromkeys(*iterable*, *value=None*)

class kombu.five.**Callable**

class kombu.five.**Iterable**

class kombu.five.**Mapping**

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

items() → a set-like object providing a view on D's items

keys() → a set-like object providing a view on D's keys

values() → an object providing a view on D's values

class kombu.five.**Queue**(*maxsize=0*)
Create a queue object with a given maximum size.

If maxsize is <= 0, the queue size is infinite.

empty()
Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

full()
Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

get(*block=True*, *timeout=None*)
Remove and return an item from the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until an item is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Empty exception if no item was available within that time. Otherwise ('block' is false), return an item if one is immediately available, else raise the Empty exception ('timeout' is ignored in that case).

get_nowait()
Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()
Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)
Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()
Return the approximate size of the queue (not reliable!).

task_done()
Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

exception kombu.five.**Empty**
Exception raised by Queue.get(block=0)/get_nowait().

exception kombu.five.**Full**
Exception raised by Queue.put(block=0)/put_nowait().

class kombu.five.**LifoQueue**(*maxsize=0*)
Variant of Queue that retrieves most recently added entries first.

class kombu.five.**array**(*typecode*[, *initializer*]) → array
Return a new array whose items are restricted by typecode, and initialized from the optional initializer value, which must be a list, string or iterable over elements of the appropriate type.

Arrays represent basic values and behave very much like lists, except the type of objects stored in them is constrained. The type is specified at object creation time by using a type code, which is a single character. The following type codes are defined:

Type code C Type Minimum size in bytes ‘b’ signed integer 1 ‘B’ unsigned integer 1 ‘u’ Unicode character 2 (see note) ‘h’ signed integer 2 ‘H’ unsigned integer 2 ‘i’ signed integer 2 ‘I’ unsigned integer 2 ‘l’ signed integer 4 ‘L’ unsigned integer 4 ‘q’ signed integer 8 (see note) ‘Q’ unsigned integer 8 (see note) ‘f’ floating point 4 ‘d’ floating point 8

NOTE: The ‘u’ typecode corresponds to Python’s unicode character. On narrow builds this is 2-bytes on wide builds this is 4-bytes.

NOTE: The ‘q’ and ‘Q’ type codes are only available if the platform C compiler used to build Python supports ‘long long’, or, on Windows, ‘__int64’.

Methods:

append() – append a new item to the end of the array buffer_info() – return information giving the current memory info byteswap() – byteswap all the items of the array count() – return number of occurrences of an object extend() – extend array by appending multiple elements from an iterable fromfile() – read items from a file object fromlist() – append items from the list frombytes() – append items from the string index() – return index of first occurrence of an object insert() – insert a new item into the array at a provided position pop() – remove and return item (default last) remove() – remove first occurrence of an object reverse() – reverse the order of the items in the array tofile() – write all items to a file object tolist() – return the array converted to an ordinary list tobytes() – return the array converted to a string

Attributes:

typecode – the typecode character used to create the array itemsize – the length in bytes of one array item

append()
Append new value v to the end of the array.

buffer_info()
Return a tuple (address, length) giving the current memory address and the length in items of the buffer used to hold array’s contents.

The length should be multiplied by the itemsize attribute to calculate the buffer length in bytes.

byteswap()
Byteswap all items of the array.

If the items in the array are not 1, 2, 4, or 8 bytes in size, RuntimeError is raised.

count()
Return number of occurrences of v in the array.

extend()
Append items to the end of the array.

frombytes()
Appends items from the string, interpreting it as an array of machine values, as if it had been read from a file using the fromfile() method).

fromfile()
Read n objects from the file object f and append them to the end of the array.

fromlist()
Append items to array from list.

fromstring()
Appends items from the string, interpreting it as an array of machine values, as if it had been read from a file using the fromfile() method).

This method is deprecated. Use frombytes instead.

fromunicode()
Extends this array with data from the unicode string ustr.

The array must be a unicode type array; otherwise a ValueError is raised. Use array.frombytes(ustr.encode(...)) to append Unicode data to an array of some other type.

index()
Return index of first occurrence of v in the array.

insert()
Insert a new item v into the array before position i.

itemsize
the size, in bytes, of one array item

pop()
Return the i-th element and delete it from the array.

i defaults to -1.

remove()
Remove the first occurrence of v in the array.

reverse()
Reverse the order of the items in the array.

tobytes()
Convert the array to an array of machine values and return the bytes representation.

tofile()
Write all items (as machine values) to the file object f.

tolist()
Convert array to an ordinary list with the same items.

tostring()
Convert the array to an array of machine values and return the bytes representation.

This method is deprecated. Use tobytes instead.

tounicode()
Extends this array with data from the unicode string ustr.

Convert the array to a unicode string. The array must be a unicode type array; otherwise a ValueError is raised. Use array.tobytes().decode() to obtain a unicode string from an array of some other type.

typecode
the typecode character used to create the array

class kombu.five.**zip_longest**
zip_longest(iter1 [,iter2 [...]], [fillvalue=None]) –> zip_longest object

Return a zip_longest object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the longest iterable in the argument sequence is exhausted and then it raises StopIteration. When the shorter iterables are exhausted, the fillvalue is substituted in their place. The fillvalue defaults to None or can be specified by a keyword argument.

class kombu.five.**map**
map(func, *iterables) –> map object

Make an iterator that computes the function using arguments from each of the iterables. Stops when the shortest iterable is exhausted.

class kombu.five.**zip**
zip(iter1 [,iter2 [...]]) –> zip object

Return a zip object whose .__next__() method returns a tuple where the i-th element comes from the i-th iterable argument. The .__next__() method continues until the shortest iterable in the argument sequence is exhausted and then it raises StopIteration.

kombu.five.**string**
alias of builtins.str

kombu.five.**string_t**
alias of builtins.str

kombu.five.**bytes_t**
alias of builtins.bytes

kombu.five.**bytes_if_py2**(*s*)
Convert str to bytes if running under Python 2.

kombu.five.**long_t**
alias of builtins.int

kombu.five.**text_t**
alias of builtins.str

kombu.five.**module_name_t**
alias of builtins.str

class kombu.five.**range**(*stop*) → range object
range(start, stop[, step]) -> range object

Return an object that produces a sequence of integers from start (inclusive) to stop (exclusive) by step. range(i, j) produces i, i+1, i+2, ..., j-1. start defaults to 0, and stop is omitted! range(4) produces 0, 1, 2, 3. These are exactly the valid indices for a list of 4 elements. When step is given, it specifies the increment (or decrement).

count(*value*) → integer – return number of occurrences of value

index(*value*[, *start*[, *stop*]]) → integer – return index of value.
Raise ValueError if the value is not present.

start

step

stop

kombu.five.**items**(*d*)
Get dict items iterator.

kombu.five.**keys**(*d*)
Get dict keys iterator.

kombu.five.**values**(*d*)
Get dict values iterator.

kombu.five.**nextfun**(*it*)
Get iterator next method.

kombu.five.**reraise**(*tp*, *value*, *tb=None*)
Reraise exception.

class kombu.five.**WhateverIO**(*v=None*, **a*, ***kw*)
StringIO that takes bytes or str.

write(*data*)
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

kombu.five.**with_metaclass**(*Type*, *skip_attrs=None*)
Class decorator to set metaclass.

Works with both Python 2 and Python 3 and it does not add an extra class in the lookup order like six.with_metaclass does (that is – it copies the original class instead of using inheritance).

class kombu.five.**StringIO**
Text I/O implementation using an in-memory buffer.

The initial_value argument sets the value of object. The newline argument is like the one of TextIOWrapper's constructor.

close()
Close the IO object.

Attempting any further operation after the object is closed will raise a ValueError.

This method has no effect if the file is already closed.

closed

getvalue()
Retrieve the entire contents of the object.

line_buffering

newlines
Line endings translated so far.

Only line endings translated during reading are considered.

Subclasses should override.

read()
Read at most size characters, returned as a string.

If the argument is negative or omitted, read until EOF is reached. Return an empty string at EOF.

readable()
Returns True if the IO object can be read.

readline()
Read until newline or EOF.

Returns an empty string if EOF is hit immediately.

seek()
Change stream position.

Seek to character offset pos relative to position indicated by whence: 0 Start of stream (the default). pos should be >= 0; 1 Current position - pos must be 0; 2 End of stream - pos must be 0.

Returns the new absolute position.

seekable()
Returns True if the IO object can be seeked.

tell()
Tell the current file position.

truncate()
Truncate size to pos.

The pos argument defaults to the current file position, as returned by tell(). The current file position is unchanged. Returns the new absolute position.

writable()
Returns True if the IO object can be written.

write()
Write string to file.

Returns the number of characters written, which is always equal to the length of the string.

kombu.five.**getfullargspec**(*func*)
Get the names and default values of a callable object's parameters.

A tuple of seven things is returned: (args, varargs, varkw, defaults, kwonlyargs, kwonlydefaults, annotations). 'args' is a list of the parameter names. 'varargs' and 'varkw' are the names of the * and ** parameters or None. 'defaults' is an n-tuple of the default values of the last n parameters. 'kwonlyargs' is a list of keyword-only parameter names. 'kwonlydefaults' is a dictionary mapping names from kwonlyargs to defaults. 'annotations' is a dictionary mapping parameter names to annotations.
Notable differences from inspect.signature():

- the "self" parameter is always reported, even for bound methods
- wrapper chains defined by __wrapped__ *not* unwrapped automatically

kombu.five.**format_d**(*i*)
Format number.

kombu.five.**monotonic**() → float
Monotonic clock, cannot go backward.

class kombu.five.**buffer_t**
Python 3 does not have a buffer type.

kombu.five.**python_2_unicode_compatible**(*cls*)
Class decorator to ensure class is compatible with Python 2.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z