



Kombu Documentation

Release 5.2.01

**Ask Solem
contributors**

Nov 02, 2021

CONTENTS

1	Getting Started	3
1.1	About	3
1.2	Features	3
1.3	Transport Comparison	4
1.4	Terminology	6
1.5	Installation	7
1.6	Getting Help	7
1.7	Bug tracker	7
1.8	Contributing	7
1.9	License	7
2	User Guide	9
2.1	Introduction	9
2.2	Connections and transports	10
2.3	Producers	13
2.4	Consumers	16
2.5	Examples	23
2.6	Simple Interface	26
2.7	Connection and Producer Pools	28
2.8	Serialization	30
2.9	Automatic Failover	33
3	Frequently Asked Questions	37
3.1	Questions	37
4	API Reference	39
4.1	Kombu Core	39
4.2	Kombu Transports	96
4.3	Kombu Asynchronous	205
4.4	Kombu utils	222
5	Change history	231
5.1	5.2.0	231
5.2	5.2.0rc1	232
5.3	5.1.0	232
5.4	5.1.0b1	232
5.5	5.0.2	233
5.6	5.0.1	233
5.7	5.0.0	234
5.8	4.6.11	234

5.9	4.6.10	234
5.10	4.6.9	234
5.11	4.6.8	235
5.12	4.6.7	235
5.13	4.6.6	236
5.14	4.6.5	236
5.15	4.6.4	236
5.16	4.6.3	237
5.17	4.6.2	237
5.18	4.6.1	237
5.19	4.6.0	237
5.20	4.5.0	238
5.21	4.4.0	239
5.22	4.3.0	239
5.23	4.2.2-post1	242
5.24	4.2.2	242
5.25	4.2.1	243
5.26	4.2.0	243
5.27	4.1.0	244
5.28	4.0.2	246
5.29	4.0.1	246
5.30	4.0	246
5.31	3.0.37	251
5.32	3.0.36	251
5.33	3.0.35	252
5.34	3.0.34	252
5.35	3.0.33	252
5.36	3.0.32	253
5.37	3.0.31	253
5.38	3.0.30	253
5.39	3.0.29	253
5.40	3.0.28	254
5.41	3.0.27	254
5.42	3.0.26	255
5.43	3.0.25	255
5.44	3.0.24	256
5.45	3.0.23	256
5.46	3.0.22	257
5.47	3.0.21	257
5.48	3.0.20	257
5.49	3.0.19	258
5.50	3.0.18	258
5.51	3.0.17	258
5.52	3.0.16	258
5.53	3.0.15	259
5.54	3.0.14	260
5.55	3.0.13	260
5.56	3.0.12	261
5.57	3.0.11	261
5.58	3.0.10	261
5.59	3.0.9	262
5.60	3.0.8	262
5.61	3.0.7	263
5.62	3.0.6	263

5.63	3.0.5	263
5.64	3.0.4	264
5.65	3.0.3	264
5.66	3.0.2	264
5.67	3.0.1	264
5.68	3.0.0	265
5.69	2.5.16	267
5.70	2.5.15	267
5.71	2.5.14	268
5.72	2.5.13	268
5.73	2.5.12	268
5.74	2.5.11	268
5.75	2.5.10	269
5.76	2.5.9	270
5.77	2.5.8	271
5.78	2.5.7	271
5.79	2.5.6	271
5.80	2.5.5	272
5.81	2.5.4	272
5.82	2.5.3	272
5.83	2.5.2	273
5.84	2.5.2	273
5.85	2.5.1	273
5.86	2.5.0	273
5.87	2.4.10	275
5.88	2.4.9	275
5.89	2.4.8	275
5.90	2.4.7	276
5.91	2.4.6	276
5.92	2.4.5	277
5.93	2.4.4	277
5.94	2.4.3	277
5.95	2.4.2	277
5.96	2.4.1	277
5.97	2.4.0	278
5.98	2.3.2	278
5.99	2.3.1	278
5.100	2.3.0	279
5.101	2.2.6	280
5.102	2.2.5	280
5.103	2.2.4	280
5.104	2.2.3	280
5.105	2.2.2	281
5.106	2.2.1	281
5.107	2.2.0	282
5.108	2.1.8	285
5.109	2.1.7	286
5.110	2.1.6	286
5.111	2.1.5	286
5.112	2.1.4	286
5.113	2.1.3	287
5.114	2.1.2	287
5.115	2.1.1	287
5.116	2.1.0	288

5.117 2.0.0	288
5.118 1.5.1	290
5.119 1.5.0	290
5.120 1.4.3	291
5.121 1.4.2	291
5.122 1.4.1	292
5.123 1.4.0	292
5.124 1.3.5	293
5.125 1.3.4	293
5.126 1.3.3	293
5.127 1.3.2	293
5.128 1.3.1	293
5.129 1.3.0	294
5.130 1.2.1	295
5.131 1.2.0	296
5.132 1.1.6	296
5.133 1.1.5	297
5.134 1.1.4	297
5.135 1.1.3	297
5.136 1.1.2	298
5.137 1.1.1	298
5.138 1.1.0	298
5.139 1.0.7	299
5.140 1.0.6	300
5.141 1.0.5	300
5.142 1.0.4	301
5.143 1.0.3	301
5.144 1.0.2	301
5.145 1.0.1	301
5.146 1.0.0	301
5.147 0.1.0	302
6 Indices and tables	303
Python Module Index	305
Index	307

Contents:

GETTING STARTED

Version 5.2.01

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 [qpuid-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](#), [MongoDB](#) 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
<i>amqp</i>	Native	Yes	Yes	Yes	Yes ³	Yes ⁴
<i>qpid</i>	Native	Yes	Yes	Yes	No	No
<i>redis</i>	Virtual	Yes	Yes	Yes (PUB/SUB)	Yes	No
<i>mongodb</i>	Virtual	Yes	Yes	Yes	Yes	Yes
<i>SQS</i>	Virtual	Yes	Yes ¹	Yes ²	No	No
<i>zookeeper</i>	Virtual	Yes	Yes [?]	No	Yes	No
<i>in-memory</i>	Virtual	Yes	Yes [?]	No	No	No
<i>SLMQ</i>	Virtual	Yes	Yes [?]	No	No	No
<i>Pyro</i>	Virtual	Yes	Yes [?]	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',
```

(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)

```

        declare=[video_queue])

# 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 = Consumer(channel)

```

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

```

from kombu import Connection, Consumer

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)

```

(continues on next page)

(continued from previous page)

```
>>> 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

A consumer declares a queue, binds it to an 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.

USER GUIDE

Release 5.2

Date Nov 02, 2021

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 Redis sentinel
sentinel://sentinel1:26379;sentinel://sentinel2:26379

# 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
<i>amqp</i>	Native	Yes	Yes	Yes	Yes ³
<i>qpid</i>	Native	Yes	Yes	Yes	No
<i>redis</i>	Virtual	Yes	Yes	Yes (PUB/SUB)	Yes
<i>SQS</i>	Virtual	Yes	Yes ¹	Yes ²	No
<i>zookeeper</i>	Virtual	Yes	Yes [?]	No	Yes
<i>in-memory</i>	Virtual	Yes	Yes [?]	No	No
<i>SLMQ</i>	Virtual	Yes	Yes [?]	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:

```
>>> from kombu import Exchange

>>> exchange = Exchange('name', type='direct')

>>> producer.publish(
...     {'hello': 'world'}, # message to send
...     exchange=exchange, # destination exchange
...     routing_key='rk',   # destination routing key,
...     declare=[exchange], # make sure exchange is declared,
... )
```

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

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.

```
>>> 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.
... )
```

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 default 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*, *timeout=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.
- **timeout** (*float*) – Set timeout to wait maximum timeout second for message to publish.
- ****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'])
```

You can create a consumer using a *Connection*. This consumer is consuming from a single queue with name *'queue'*:

```
>>> queue = Queue('queue', routing_key='queue')
>>> consumer = connection.Consumer(queue)
```

You can also instantiate `Consumer` directly, it takes a channel or a connection as an argument. This consumer also consumes from single queue with name `'queue'`:

```
>>> queue = Queue('queue', routing_key='queue')
>>> with Connection('amqp://') as conn:
...     with conn.channel() as channel:
...         consumer = Consumer(channel, queue)
```

A consumer needs to specify a handler for received data. This handler is specified in the form of a callback. The callback function is called by kombu every time a new message is received. The callback is called with two parameters: `body`, containing deserialized data sent by a producer, and a `Message` instance `message`. The user is responsible for acknowledging messages when manual acknowledgement is set.

```
>>> def callback(body, message):
...     print(body)
...     message.ack()

>>> consumer.register_callback(callback)
```

Draining events from a single consumer

The method `drain_events` blocks indefinitely by default. This example sets the timeout to 1 second:

```
>>> with consumer:
...     connection.drain_events(timeout=1)
```

Draining events from several consumers

Each consumer has its own list of queues. Each consumer accepts data in `'json'` format:

```
>>> from kombu.utils.compat import nested

>>> queues1 = [Queue('queue11', routing_key='queue11'),
...             Queue('queue12', routing_key='queue12')]
>>> queues2 = [Queue('queue21', routing_key='queue21'),
...             Queue('queue22', routing_key='queue22')]
>>> 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)
```

The full example will look as follows:

```
from kombu import Connection, Consumer, Queue

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

queue1 = Queue('queue1', routing_key='queue1')
queue2 = Queue('queue2', routing_key='queue2')
```

(continues on next page)

(continued from previous page)

```
with Connection('amqp://') as conn:
    with conn.channel() as channel:
        consumer = Consumer(conn, [queue1, queue2], accept=['json'])
        consumer.register_callback(callback)
        with consumer:
            conn.drain_events(timeout=1)
```

2.4.2 Consumer mixin classes

Kombu provides predefined mixin classes in module *mixins*. It contains two classes: *ConsumerMixin* for creating consumers and *ConsumerProducerMixin* for creating consumers supporting also publishing messages. Consumers can be created just by subclassing mixin class and overriding some of the methods:

```
from kombu.mixins import ConsumerMixin

class C(ConsumerMixin):

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

    def get_consumers(self, Consumer, channel):
        return [
            Consumer(channel, 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):
    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'])]
```

(continues on next page)

(continued from previous page)

```

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

C(connection).run()

```

The main use of `ConsumerProducerMixin` is to create 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.

2.4.3 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>

2.4.4 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:

```
import datetime

from kombu import Connection

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

hello_consumer.py:

```
from kombu import Connection

with Connection('amqp://guest:guest@localhost:5672//') as conn:
    simple_queue = conn.SimpleQueue('simple_queue')
    message = simple_queue.get(block=True, timeout=1)
    print(f'Received: {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 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 kombu.log import get_logger
from kombu.mixins import ConsumerMixin
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
    from kombu.utils.debug import setup_logging

    # setup root logger
    setup_logging(loglevel='INFO', loggers=[''])

    with Connection('amqp://guest:guest@localhost:5672/') as conn:
```

(continues on next page)

(continued from previous page)

```

try:
    worker = Worker(conn)
    worker.run()
except KeyboardInterrupt:
    print('bye bye')

```

tasks.py:

```

def hello_task(who='world'):
    print(f'Hello {who}')

```

client.py:

```

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, 'mybuffer')
>>> # ... 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,
```

(continues on next page)

(continued from previous page)

```

        'context': context,
        'hostname': socket.gethostname(),
        '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' % (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.
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')
```

Pool limits

By default every connection instance has a limit of 10 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()
```

You can also get current limit using *kombu.pools.get_limit()*:

```
>>> from kombu import pools
>>> pools.get_limit()
10
>>> pools.set_limit(100)
100
>>> kombu.pools.get_limit()
100
```

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',
```

(continues on next page)

(continued from previous page)

```
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',
)
```

2.9 Automatic Failover

Automatic failover is functionality for connecting to clustered broker. Application using automatic failover should be able to automatically connect to healthy node and react to unexpected failure of node in cluster.

2.9.1 Connection failover

The `Connection` is accepting multiple URLs to several brokers. During connecting to broker, kombu is automatically picking the healthy node from the list. In the example below, kombu uses healthy.example.com broker:

```
>>> conn = Connection(
...     'amqp://guest:guest@broken.example.com;guest:guest@healthy.example.com'
... )
>>> conn.connect()
>>> conn
<Connection: amqp://guest:*@healthy.example.com at 0x6ffff751710>
```

`Connection` also accepts `failover_strategy` parameter which defines the strategy of trying the nodes:

```
>>> Connection(
...     'amqp://broker1.example.com;amqp://broker2.example.com',
...     failover_strategy='round-robin'
... )
```

The current list of available failover strategies is defined in `kombu.connection` module:

```
>>> import kombu
>>> kombu.connection.failover_strategies
{'round-robin': <class 'itertools.cycle'>, 'shuffle': <function shufflecycle at
↳ 0x6fffff8547a0>}
```

Failover during connection handle only failover during calling `connect()` method of `Connection`.

2.9.2 Operation failover

Failover of connection using multiple connection strings in `Connection` solves problem when broker is unavailable during creating new connection. But in real world these connections are long lived and hence it is possible that broker fails during lifetime of connection. For this scenario retrying of operation executed against broker is needed. Retrying ensures that failed operation triggers new connection to healthy broker and re-execution of failed operation.

Failover is implemented in `ensure()` method which tries to execute the function. When contacting broker fails, it reconnects the underlying connection and re-executes the function again. The following example is ensuring that `publish()` method is re-executed when errors occurred:

```
>>> 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')
```

Some methods are accepting channel as a parameter, e.g. `declare()`. Since channel is passed as parameter, it is not refreshed automatically during failover and hence retrying calling of method fails. In this scenarios `autoretry()` needs to be used which automatically passes channel and refresh it during failover:

```
>>> import kombu
>>> conn = kombu.Connection('amqp://broker1:5672;amqp://broker2:5672')
>>> conn.connect()
>>> q = kombu.Queue('test_queue')

>>> declare = conn.autoretry(q.declare)
>>> declare()
```

2.9.3 Producer

`publish()` can have automatic failover using `ensure()` as mentioned before. Moreover, it contains retry parameter as a shortcut for retrying. The following example is retrying publishing when error occurs:

```
>>> from kombu import *
>>> with Connection('amqp://broker1:5672;amqp://broker2:5672') as conn:
...     with conn.channel() as channel:
...         producer = conn.Producer()
...         producer = Producer(channel)
...         producer.publish(
...             {'hello': 'world'}, routing_key='queue', retry=True
...         )
```

2.9.4 Consumer

Consumer with failover functionality can be implemented using following function:

```
>>> def consume():
...     while True:
...         try:
...             conn.drain_events(timeout=1)
...         except socket.timeout:
...             pass
```

This function is draining events in infinite loop with timeout to avoid blocked connections of unavailable broker. Consumer with failover is implemented by wrapping consume function using `ensure()` method:

```
>>> consume = conn.ensure(conn, consume)
>>> consume()
```

The full example implementing consumer with failover is as follows:

```
>>> from kombu import *
>>> import socket

>>> def callback(body, message):
...     print(body)
...     message.ack()
```

(continues on next page)

(continued from previous page)

```

>>> queue = Queue('queue', routing_key='queue')
>>> with Connection('amqp://broker1:5672;amqp://broker2:5672') as conn:
...     def consume():
...         while True:
...             try:
...                 conn.drain_events(timeout=1)
...             except socket.timeout:
...                 pass
...         with conn.channel() as channel:
...             consumer = Consumer(channel, queue)
...             consumer.register_callback(callback)
...         with consumer:
...             while True:
...                 consume = conn.ensure(conn, consume)
...                 consume()

```

When implementing consumer as *ConsumerMixin*, the failover functionality is by wrapping consume method with *ensure()*:

```

>>> from kombu import *
>>> from kombu.mixins import ConsumerMixin

>>> class C(ConsumerMixin):
...     def __init__(self, connection):
...         self.connection = connection
...     def get_consumers(self, Consumer, channel):
...         return [
...             Consumer(
...                 [Queue('queue', routing_key='queue')],
...                 callbacks=[self.on_message], accept=['json']
...             ),
...         ]
...     def on_message(self, body, message):
...         print('RECEIVED MESSAGE: {0!r}'.format(body))
...         message.ack()
...     def consume(self, *args, **kwargs):
...         consume = conn.ensure(conn, super().consume)
...         return consume(*args, **kwargs)

>>> with Connection('amqp://broker1:5672;amqp://broker2:5672') as conn:
...     C(conn).run()

```


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?

API REFERENCE

Release 5.2

Date Nov 02, 2021

4.1 Kombu Core

4.1.1 Kombu - kombu

- *Connection*
- *Exchange*
- *Queue*
- *Message Producer*
- *Message Consumer*

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.

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/dict*) – 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')*) → *str*

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

Public interface of `_ensure_connection` for retro-compatibility.

Returns `kombu.Connection` instance.

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*).

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'>), ('passive', <class 'bool'>), ('auto_delete', <class 'bool'>),
('delivery_mode', <function Exchange.<lambda>>), ('no_declare', <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.

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', None), ('binding_arguments', None), ('consumer_arguments',
         None), ('durable', <class 'bool'>), ('exclusive', <class 'bool'>), ('auto_delete',
         <class 'bool'>), ('no_ack', None), ('alias', None), ('bindings', <class 'list'>),
         ('no_declare', <class 'bool'>), ('expires', <class 'float'>), ('message_ttl', <class
         'float'>), ('max_length', <class 'int'>), ('max_length_bytes', <class 'int'>),
         ('max_priority', <class 'int'>))
```

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.

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 default 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*, *timeout=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.
- **timeout** (*float*) – Set timeout to wait maximum timeout second for message to publish.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

revive(*channel*)

Revive the producer after connection loss.

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.

4.1.2 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', None), ('binding_arguments', None), ('consumer_arguments',
         None), ('durable', <class 'bool'>), ('exclusive', <class 'bool'>), ('auto_delete',
         <class 'bool'>), ('no_ack', None), ('alias', None), ('bindings', <class 'list'>),
         ('no_declare', <class 'bool'>), ('expires', <class 'float'>), ('message_ttl', <class
         'float'>), ('max_length', <class 'int'>), ('max_length_bytes', <class 'int'>),
         ('max_priority', <class 'int'>), ('queue', None))
```

`kombu.common.collect_replies`(*conn, channel, queue, *args, **kwargs*)

Generator collecting replies from queue.

`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.

`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.ittermessages(conn, channel, queue, limit=1, timeout=None, callbacks=None, **kwargs)`

Iterator over messages.

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

Declare entity (cached).

`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.uuid(_uuid=<function uuid4>)`

Generate unique id in UUID4 format.

See also:

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

4.1.3 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 convenience name for the matching 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 matching 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.

4.1.4 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):
        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

4.1.5 Simple Messaging API - kombu.simple

Simple messaging interface.

- *Persistent*
- *Buffer*

Persistent

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

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()

Buffer

```
class kombu.simple.SimpleBuffer(channel, name, no_ack=None, queue_opts=None, queue_args=None,
                                exchange_opts=None, serializer=None, compression=None,
                                accept=None)
```

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()

```

4.1.6 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](#)
- [Lampports 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**(*clock, timestamp, id, obj=None*)

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])`

4.1.7 Carrot Compatibility - kombu.compat

Carrot compatibility interface.

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

- *Publisher*
- *Consumer*
- *ConsumerSet*

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

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

auto_delete = False

property backend

property channel

close()

compression = None

Default compression method. Disabled by default.

property connection

declare()

Declare the exchange.

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

durable = True

exchange = ''

Default exchange

exchange_type = 'direct'

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, timeout=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.
- **timeout** (*float*) – Set timeout to wait maximum timeout second for message to publish.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

release()

revive(*channel*)

Revive the producer after connection loss.

routing_key = ''

Default routing key.

send(*args, **kwargs)

serializer = None

Default serializer to use. Default is JSON.

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

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.

auto_delete = 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.

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

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()`.

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*)

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

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_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

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.

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

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

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.

4.1.8 Pidbox - kombu.pidbox

Generic process mailbox.

- *Introduction*
 - *Creating the applications Mailbox*
 - *Example Node*
 - *Example Client*
- *Mailbox*
- *Node*

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:
...     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)
```

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)

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)
```

4.1.9 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.

4.1.10 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.

4.1.11 Connection - kombu.connection

Client (Connection).

- [Connection](#)
- [Pools](#)

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 qpidd transports. For other transports you can use stunnel.

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

Keyword Arguments

- **ssl** (*bool/dict*) – 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')*) → *str*

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(*args, **kwargs)

Public interface of `_ensure_connection` for retro-compatibility.

Returns kombu.Connection instance.

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

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 = `'/'`

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.ChannellLimitExceeded'>`

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).

4.1.12 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.

4.1.13 Message Compression - `kombu.compression`

Compression utilities.

- *Encoding/decoding*
 - *Registry*

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.

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.

4.1.14 Connection/Producer Pools - `kombu.pools`

Public resource pools.

class `kombu.pools.PoolGroup(limit=None, close_after_fork=True)`
Collection of resource pools.

create(*resource, limit*)

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

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

property channel**close()****compression = None**

Default compression method. Disabled by default.

property connection**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*, *timeout=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.
- **timeout** (*float*) – Set timeout to wait maximum timeout second for message to publish.
- ****properties** (*Any*) – Additional message properties, see AMQP spec.

release()

revive(*channel*)

Revive the producer after connection loss.

routing_key = ''

Default routing key.

serializer = `None`

Default serializer to use. Default is JSON.

close_after_fork = `True`

close_resource(*resource*)

create_producer()

new()

prepare(*p*)

release(*resource*)

setup()

`kombu.pools.get_limit()`

Get current connection pool limit.

`kombu.pools.register_group(group)`

Register group (can be used as decorator).

`kombu.pools.reset(*args, **kwargs)`

Reset all pools by closing open resources.

`kombu.pools.set_limit(limit, force=False, reset_after=False, ignore_errors=False)`

Set new connection pool limit.

4.1.15 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.

4.1.16 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()`

4.1.17 Message Serialization - `kombu.serialization`

Serialization utilities.

- [Overview](#)
- [Exceptions](#)
- [Serialization](#)
- [Registry](#)

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.

Exceptions

exception `kombu.serialization.SerializerNotInstalled`

Support for the requested serialization type is not installed.

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.

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.

4.2 Kombu Transports

4.2.1 Built-in Transports - `kombu.transport`

Built-in transports.

- *Data*
- *Functions*

Data

`kombu.transport.DEFAULT_TRANSPORT`

Default transport used when no transport specified.

`kombu.transport.TRANSPORT_ALIASES`

Mapping of transport aliases/class names.

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.

4.2.2 Transport Base Class - `kombu.transport.base`

Base transport interface.

- *Message*
- *Transport*

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.

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*)

4.2.3 Virtual Transport Base Class - `kombu.transport.virtual`

- *Transports*
- *Channel*
- *Message*
- *Quality Of Service*
- *In-memory State*

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

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*)

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'>, 'fanout': <class
'kombu.transport.virtual.exchange.FanoutExchange'>, 'topic': <class
'kombu.transport.virtual.exchange.TopicExchange'>}
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.

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()

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 adheres 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 unacknowledged messages.

To be filled in for visibility_timeout style implementations.

Note: This is implementation optional, and currently only used by the Redis transport.

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,
```

(continues on next page)

(continued from previous page)

```
# ... ,  
}
```

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),  
        # ... ,  
    },  
    # ... ,  
}
```

4.2.4 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*

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'

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

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'

Interface

class kombu.transport.virtual.exchange.**ExchangeType**(*channel*)

Base class for exchanges.

Implements the specifics for an exchange type.

Parameters **channel** (*ChannelIT*) – 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

4.2.5 Azure Storage Queues Transport - kombu.transport.azurestoragequeues

Azure Storage Queues transport module for kombu.

More information about Azure Storage Queues: <https://azure.microsoft.com/en-us/services/storage/queues/>

Features

- Type: Virtual
- Supports Direct: *Unreviewed*
- Supports Topic: *Unreviewed*
- Supports Fanout: *Unreviewed*
- Supports Priority: *Unreviewed*
- Supports TTL: *Unreviewed*

Connection String

Connection string has the following format:

`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.

Transport Options

- `queue_name_prefix`

- *Transport*
- *Channel*

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`

port number used when no port is specified.

polling_interval = `1`

Time to sleep between unsuccessful polls.

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
```

4.2.6 Azure Service Bus Transport - `kombu.transport.azure_servicebus`

Azure Service Bus Message Queue transport module for kombu.

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.

Notes when using with Celery if you are experiencing issues with programs not terminating properly. The Azure Service Bus SDK uses the Azure uAMQP library which in turn creates some threads. If the AzureServiceBus Channel is closed, said threads will be closed properly, but it seems there are times when Celery does not do this so these threads will be left running. As the uAMQP threads are not marked as Daemon threads, they will not be killed when the main thread exits. Setting the `uamqp_keep_alive_interval` transport option to 0 will prevent the `keep_alive` thread from starting

More information about Azure Service Bus: <https://azure.microsoft.com/en-us/services/service-bus/>

Features

- Type: Virtual
- Supports Direct: *Unreviewed*
- Supports Topic: *Unreviewed*
- Supports Fanout: *Unreviewed*
- Supports Priority: *Unreviewed*
- Supports TTL: *Unreviewed*

Connection String

Connection string has the following format:

`azureservicebus://SAS_POLICY_NAME:SAS_KEY@SERVICE_BUSNAMESPACE`

Transport Options

- `queue_name_prefix` - String prefix to prepend to queue names in a service bus namespace.
- `wait_time_seconds` - Number of seconds to wait to receive messages. Default 5
- `peek_lock_seconds` - Number of seconds the message is visible for before it is requeued and sent to another consumer. Default 60
- `uamqp_keep_alive_interval` - Interval in seconds the Azure uAMQP library should send keepalive messages. Default 30
- `retry_total` - Azure SDK retry total. Default 3
- `retry_backoff_factor` - Azure SDK exponential backoff factor. Default 0.8

- `retry_backoff_max` - Azure SDK retry total time. Default 120

- *Transport*
- *Channel*

Transport

class kombu.transport.azure_servicebus.**Transport**(*client*, ****kwargs**)

Azure Service Bus transport.

class **Channel**(**args*, ****kwargs**)

Azure Service Bus channel.

basic_ack(*delivery_tag: str*, *multiple: bool = False*) → *None*

Acknowledge message.

basic_cancel(*consumer_tag*)

Cancel consumer by consumer tag.

basic_consume(*queue*, *no_ack*, **args*, ****kwargs**)

Consume from *queue*.

close() → *None*

Close channel.

Cancel all consumers, and requeue unacked messages.

property conninfo

default_peek_lock_seconds = 60

default_retry_backoff_factor = 0.8

default_retry_backoff_max = 120

default_retry_total = 3

default_uamqp_keep_alive_interval = 30

default_wait_time_seconds = 5

domain_format = 'kombu%(vhost)s'

entity_name(*name: str*, *table: Optional[Dict[int, int]] = None*) → *str*

Format AMQP queue name into a valid ServiceBus queue name.

peek_lock_seconds

property queue_mgmt_service:

`azure.servicebus.management._management_client.ServiceBusAdministrationClient`

queue_name_prefix

property queue_service: `azure.servicebus._servicebus_client.ServiceBusClient`

retry_backoff_factor

retry_backoff_max

retry_total

property transport_options

```
    uamqp_keep_alive_interval
    wait_time_seconds

classmethod as_uri(uri: str, include_password=False, mask='**') → str
can_parse_url = True
    Set to True if Connection should pass the URL unmodified.
default_port = None
    port number used when no port is specified.
static parse_uri(uri: str) → Tuple[str, str, str]
polling_interval = 1
    Time to sleep between unsuccessful polls.
```

Channel

```
class kombu.transport.azure_servicebus.Channel(*args, **kwargs)
    Azure Service Bus channel.

    basic_ack(delivery_tag: str, multiple: bool = False) → None
        Acknowledge message.

    basic_cancel(consumer_tag)
        Cancel consumer by consumer tag.

    basic_consume(queue, no_ack, *args, **kwargs)
        Consume from queue.

    close() → None
        Close channel.

        Cancel all consumers, and requeue unacked messages.

    property conninfo
    default_peek_lock_seconds = 60
    default_retry_backoff_factor = 0.8
    default_retry_backoff_max = 120
    default_retry_total = 3
    default_uamqp_keep_alive_interval = 30
    default_wait_time_seconds = 5
    domain_format = 'kombu%(vhost)s'
    entity_name(name: str, table: Optional[Dict[int, int]] = None) → str
        Format AMQP queue name into a valid ServiceBus queue name.

    peek_lock_seconds

    property queue_mgmt_service:
        azure.servicebus.management._management_client.ServiceBusAdministrationClient

    queue_name_prefix

    property queue_service: azure.servicebus._servicebus_client.ServiceBusClient

    retry_backoff_factor
```

```
retry_backoff_max
retry_total
property transport_options
uamqp_keep_alive_interval
wait_time_seconds
```

4.2.7 Pure-python AMQP Transport - `kombu.transport.pyamqp`

pyamqp transport module for Kombu.

Pure-Python amqp transport using py-amqp library.

Features

- Type: Native
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: Yes
- Supports Priority: Yes
- Supports TTL: Yes

Connection String

Connection string can have the following formats:

```
amqp://[USER:PASSWORD@]BROKER_ADDRESS[:PORT][/VIRTUALHOST]
[USER:PASSWORD@]BROKER_ADDRESS[:PORT][/VIRTUALHOST]
amqp://
```

For TLS encryption use:

```
amqps://[USER:PASSWORD@]BROKER_ADDRESS[:PORT][/VIRTUALHOST]
```

Transport Options

Transport Options are passed to constructor of underlying py-amqp *Connection* class.

Using TLS

Transport over TLS can be enabled by `ssl` parameter of `Connection` class. By setting `ssl=True`, TLS transport is used:

```
conn = Connect('amqp://', ssl=True)
```

This is equivalent to `amqps://` transport URI:

```
conn = Connect('amqps://')
```

For adding additional parameters to underlying TLS, `ssl` parameter should be set with dict instead of `True`:

```
conn = Connect('amqp://broker.example.com', ssl={
    'keyfile': '/path/to/keyfile'
    'certfile': '/path/to/certfile',
    'ca_certs': '/path/to/ca_certfile'
})
```

All parameters are passed to `ssl` parameter of `amqp.connection.Connection` class.

SSL option `server_hostname` can be set to `None` which is causing using hostname from broker URL. This is useful when failover is used to fill `server_hostname` with currently used broker:

```
conn = Connect('amqp://broker1.example.com;broker2.example.com', ssl={
    'server_hostname': None
})
```

- *Transport*
- *Connection*
- *Channel*
- *Message*

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'>,)
    Tuple of errors that can happen due to channel/method failure.

close_connection(connection)
    Close the AMQP broker connection.

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'OSError'>,
<class 'OSError'>, <class 'OSError'>)
    Tuple of errors that can happen due to connection failure.

create_channel(connection)

property default_connection_params

default_port = 5672
    Default port used when no port has been specified.

default_ssl_port = 5671

drain_events(connection, **kwargs)

driver_name = 'py-amqp'
    Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'amqp'
    Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis
    (driver_type: 'redis'), etc...

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', 'headers', 'topic'})}, 'heartbeats': True}

qos_semantics_matches_spec(connection)

recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)

recoverable_connection_errors = (<class
'amqp.exceptions.RecoverableConnectionError'>, <class 'OSError'>, <class 'OSError'>,
<class 'OSError'>)>

register_with_event_loop(connection, loop)

verify_connection(connection)
```

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 messages.

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 arbitrary 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='BssbbbbbF')
```

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*, *confirm_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 constraints), `MessageNacked` exception is raised and set `confirm_timeout` to wait maximum `confirm_timeout` second for message to confirm.

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.

- **timeout** – short

timeout for publish

Set timeout to wait maximum `timeout` second for message to publish.

- **confirm_timeout** – short

`confirm_timeout` for publish in confirm mode

When the channel is in confirm mode set `confirm_timeout` to wait maximum `confirm_timeout` second for message to confirm.

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

Defines a scope of QoS. Semantics of this parameter differs between AMQP 0-9-1 standard and RabbitMQ broker:

MEANING IN AMQP 0-9-1: False: shared across all consumers on the channel
True: shared across all consumers on the connection

MEANING IN RABBITMQ:

False: applied separately to each new consumer on the channel

True: shared across all consumers on the channel

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='BssbbbbF'*)

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 overflowing 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 exchanging 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='BsbbbbF'*)

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 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

Number of successful reads from socket.

bytes_sent = 0

Number of successful writes to socket.

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

Original heartbeat interval value proposed by client.

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'>, <class 'OSError'>, <class 'OSError'>)

dispatch_method(*method_sig, payload, content*)

drain_events(*timeout=None*)

property frame_writer

heartbeat = None

Final heartbeat interval value (in float seconds) after negotiation

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

Time of last heartbeat received (in monotonic time, if available).

last_heartbeat_sent = 0

Time of last heartbeat sent (in monotonic time, if available).

library_properties = {'product': 'py-amqp', 'product_version': '5.0.6'}

These are sent to the server to announce what features we support, type of client etc.

negotiate_capabilities = {'authentication_failure_close': True, 'connection.blocked': True, 'consumer_cancel_notify': True}

Mapping of protocol extensions to enable. The server will report these in server_properties[capabilities], and if a key in this map is present the client will tell the server to either enable or disable the capability depending on the value set in this map. For example with:

```
negotiate_capabilities = { 'consumer_cancel_notify': True,
}
```

The client will enable this capability if the server reports support for it, but if the value is False the client will disable the capability.

property on_inbound_frame

on_inbound_method(*channel_id, method_sig, payload, content*)

prev_recv = None

Number of bytes received from socket at the last heartbeat check.

prev_sent = None

Number of bytes sent to socket at the last heartbeat check.

recoverable_channel_errors = (<class 'amqp.exceptions.RecoverableChannelError'>,)

recoverable_connection_errors = (<class 'amqp.exceptions.RecoverableConnectionError'>, <class 'OSError'>, <class 'OSError'>, <class 'OSError'>)

send_heartbeat()

send_method(*sig, format=None, args=None, content=None, wait=None, callback=None, returns_tuple=False*)

property server_capabilities

server_heartbeat = None

Original heartbeat interval proposed by server.

property sock

then(*on_success, on_error=None*)

property transport

wait(*method, callback=None, timeout=None, returns_tuple=False*)

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)
```

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

4.2.8 librabbitmq AMQP transport - `kombu.transport.librabbitmq`

4.2.9 Apache QPid Transport - `kombu.transport.qpid`

Qpid Transport module for kombu.

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.

Features

- Type: Native
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: Yes
- Supports Priority: Yes
- Supports TTL: Yes

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/
```

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'

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 con-

sumers 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,)

Tuple of errors that can happen due to channel/method failure.

close_connection(*connection*)

Close the [Connection](#) object.

Parameters **connection** (`kombu.transport.qpid.Connection`) – The Connection that should be closed.

connection_errors = (None, <class 'OSError'>)

Tuple of errors that can happen due to connection failure.

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'

Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'qpid'

Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis (driver_type: 'redis'), etc...

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', 'topic'}), 'heartbeats': False}
```

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 `AuthenticationFailure` 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 `qpuid.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 `qpuid.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'

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`

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 `qpido.messaging.Message`. This method unpacks the payload of the `qpido.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 `qpido.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 `qpido.messaging.Message`, but this method takes the payload of the `qpido.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 `qpido.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 `qpuid.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()
```

4.2.10 In-memory Transport - `kombu.transport.memory`

In-memory transport module for Kombu.

Simple transport using memory for storing messages. Messages can be passed only between threads.

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: No
- Supports Priority: No
- Supports TTL: Yes

Connection String

Connection string is in the following format:

```
memory://
```

- *Transport*
- *Channel*

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
```

flag to restore unacked messages when channel goes out of scope.

```
events = {}
```

```
queues = {}
```

```
supports_fanout = True
```

flag set if the channel supports fanout exchanges.

```
driver_name = 'memory'
```

Name of driver library (e.g. 'py-amqp', 'redis').

```
driver_type = 'memory'
```

Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis (driver_type: 'redis'), etc...

```
driver_version()
```

```
global_state = <kombu.transport.virtual.base.BrokerState object>
```

memory backend state is global.

```
implements = {'asynchronous': False, 'exchange_type': frozenset({'direct',  
'fanout', 'headers', 'topic'}), 'heartbeats': False}
```

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

flag to restore unacked messages when channel goes out of scope.

events = {}

queues = {}

supports_fanout = True

flag set if the channel supports fanout exchanges.

4.2.11 Redis Transport - `kombu.transport.redis`

Redis transport module for Kombu.

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: Yes
- Supports Priority: Yes
- Supports TTL: No

Connection String

Connection string has the following format:

```
redis://[USER:PASSWORD@]REDIS_ADDRESS[:PORT][/VIRTUALHOST]
rediss://[USER:PASSWORD@]REDIS_ADDRESS[:PORT][/VIRTUALHOST]
```

To use sentinel for dynamic Redis discovery, the connection string has following format:

```
sentinel://[USER:PASSWORD@]SENTINEL_ADDRESS[:PORT]
```

Transport Options

- `sep`
- `ack_emulation`: (bool) If set to True transport will simulate Acknowledge of AMQP protocol.
- `unacked_key`
- `unacked_index_key`
- `unacked_mutex_key`
- `unacked_mutex_expire`
- `visibility_timeout`
- `unacked_restore_limit`
- `fanout_prefix`
- `fanout_patterns`
- `global_keyprefix`: (str) The global key prefix to be prepended to all keys used by Kombu
- `socket_timeout`
- `socket_connect_timeout`
- `socket_keepalive`
- `socket_keepalive_options`
- `queue_order_strategy`
- `max_connections`
- `health_check_interval`
- `retry_on_timeout`
- `priority_steps`

- *Transport*
- *Channel*
- *SentinelChannel*

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**

If disabled, unacked messages won't be restored at shutdown.

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 unacknowledged 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

alias of `redis.connection.Connection`

connection_class_ssl

alias of `redis.connection.SSLConnection`

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', 'unacked_key', 'unacked_index_key', 'unacked_mutex_key',
                          'unacked_mutex_expire', 'visibility_timeout', 'unacked_restore_limit',
                          'fanout_prefix', 'fanout_patterns', 'global_keyprefix', 'socket_timeout',
                          'socket_connect_timeout', 'socket_keepalive', 'socket_keepalive_options',
                          'queue_order_strategy', 'max_connections', 'health_check_interval',
                          'retry_on_timeout', 'priority_steps')
```

get_table(exchange)

Get table of bindings for *exchange*.

global_keyprefix = ''

The global key prefix will be prepended to all keys used by Kombu, which can be useful when a redis database is shared by different users. By default, no prefix is prepended.

health_check_interval = 25

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.

retry_on_timeout = None

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
flag set if the channel supports fanout exchanges.

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
port number used when no port is specified.

driver_name = 'redis'
Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'redis'
Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis (driver_type: 'redis'), etc...

driver_version()

implements = {'asynchronous': True, 'exchange_type': frozenset({'direct', 'fanout', 'topic'}) , 'heartbeats': False}

on_readable(*fileno*)
Handle AIO event for one of our file descriptors.

polling_interval = None
Time to sleep between unsuccessful polls.

register_with_event_loop(*connection, loop*)

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
            If disabled, unacked messages won't be restored at shutdown.

        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 unacknowledged 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

alias of `redis.connection.Connection`

connection_class_ssl

alias of `redis.connection.SSLConnection`

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', 'unacked_key', 'unacked_index_key', 'unacked_mutex_key',
                           'unacked_mutex_expire', 'visibility_timeout', 'unacked_restore_limit',
                           'fanout_prefix', 'fanout_patterns', 'global_keyprefix', 'socket_timeout',
                           'socket_connect_timeout', 'socket_keepalive', 'socket_keepalive_options',
                           'queue_order_strategy', 'max_connections', 'health_check_interval',
                           'retry_on_timeout', 'priority_steps')
```

get_table(*exchange*)

Get table of bindings for *exchange*.

global_keyprefix = ''

The global key prefix will be prepended to all keys used by Kombu, which can be useful when a redis database is shared by different users. By default, no prefix is prepended.

health_check_interval = 25

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.

retry_on_timeout = None

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

flag set if the channel supports fanout exchanges.

```

unacked_index_key = 'unacked_index'
unacked_key = 'unacked'
unacked_mutex_expire = 300
unacked_mutex_key = 'unacked_mutex'
unacked_restore_limit = None
visibility_timeout = 3600

```

SentinelChannel

`class kombu.transport.redis.SentinelChannel(*args, **kwargs)`

Channel with explicit Redis Sentinel knowledge.

Broker url is supposed to look like:

```
sentinel://0.0.0.0:26379;sentinel://0.0.0.0:26380/...
```

where each sentinel is separated by a ;.

Other arguments for the sentinel should come from the transport options (see *transport_options* of [Connection](#)).

You must provide at least one option in Transport options:

- *master_name* - name of the redis group to poll

Example:

```

>>> import kombu
>>> c = kombu.Connection(
    'sentinel://sentinel1:26379;sentinel://sentinel2:26379',
    transport_options={'master_name': 'mymaster'})
>>> c.connect()

```

connection_class

alias of `redis.sentinel.SentinelManagedConnection`

connection_class_ssl

alias of `kombu.transport.redis.SentinelManagedSSLConnection`

```

from_transport_options = ('body_encoding', 'deadletter_queue', 'sep',
    'ack_emulation', 'unacked_key', 'unacked_index_key', 'unacked_mutex_key',
    'unacked_mutex_expire', 'visibility_timeout', 'unacked_restore_limit',
    'fanout_prefix', 'fanout_patterns', 'global_keyprefix', 'socket_timeout',
    'socket_connect_timeout', 'socket_keepalive', 'socket_keepalive_options',
    'queue_order_strategy', 'max_connections', 'health_check_interval',
    'retry_on_timeout', 'priority_steps', 'master_name', 'min_other_sentinels',
    'sentinel_kwargs')

```

4.2.12 MongoDB Transport - `kombu.transport.mongodb`

MongoDB transport module for kombu.

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: Yes
- Supports Priority: Yes
- Supports TTL: Yes

Connection String

Unreviewed

Transport Options

- `connect_timeout`,
- `ssl`,
- `ttl`,
- `capped_queue_size`,
- `default_hostname`,
- `default_port`,
- `default_database`,
- `messages_collection`,
- `routing_collection`,
- `broadcast_collection`,
- `queues_collection`,
- `calc_queue_size`,

- *Transport*
- *Channel*

Transport

```
class kombu.transport.mongodb.Transport(client, **kwargs)
    MongoDB Transport.

    class Channel(*args, **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', 'ttl', 'capped_queue_size', 'default_hostname',
            'default_port', 'default_database', 'messages_collection',
            'routing_collection', 'broadcast_collection', 'queues_collection',
            'calc_queue_size')
        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
            flag set if the channel supports fanout exchanges.
        ttl = False
        can_parse_url = True
            Set to True if Connection should pass the URL unmodified.
```

```
channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class
'pymongo.errors.ConnectionFailure'>, <class 'pymongo.errors.OperationFailure'>)
    Tuple of errors that can happen due to channel/method failure.

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class
'pymongo.errors.ConnectionFailure'>)
    Tuple of errors that can happen due to connection failure.

default_port = 27017
    port number used when no port is specified.

driver_name = 'pymongo'
    Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'mongodb'
    Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis
    (driver_type: 'redis'), etc...

driver_version()

implements = {'asynchronous': False, 'exchange_type': frozenset({'direct',
'fanout', 'topic'})}, 'heartbeats': False}

polling_interval = 1
    Time to sleep between unsuccessful polls.
```

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', 'ttl', 'capped_queue_size', 'default_hostname', 'default_port',
    'default_database', 'messages_collection', 'routing_collection',
    'broadcast_collection', 'queues_collection', 'calc_queue_size')

    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
    flag set if the channel supports fanout exchanges.
ttl = False

```

4.2.13 Consul Transport - kombu.transport.consul

Consul Transport module for Kombu.

Features

It uses Consul.io's Key/Value store to transport messages in Queues

It uses python-consul for talking to Consul's HTTP API

Features

- Type: Native
- Supports Direct: Yes
- Supports Topic: *Unreviewed*
- Supports Fanout: *Unreviewed*
- Supports Priority: *Unreviewed*
- Supports TTL: *Unreviewed*

Connection String

Connection string has the following format:

```
consul://CONSUL_ADDRESS[:PORT]
```

- *Transport*
- *Channel*

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
        port number used when no port is specified.

    driver_name = 'consul'
        Name of driver library (e.g. 'py-amqp', 'redis').

    driver_type = 'consul'
        Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis
        (driver_type: 'redis'), etc...

    driver_version()

    verify_connection(connection)
```

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'
```

4.2.14 Etcd Transport - kombu.transport.etcd

Etcd Transport module for Kombu.

It uses Etcd as a store to transport messages in Queues

It uses python-etcd for talking to Etcd's HTTP API

Features

- Type: Virtual
- Supports Direct: *Unreviewed*
- Supports Topic: *Unreviewed*
- Supports Fanout: *Unreviewed*
- Supports Priority: *Unreviewed*
- Supports TTL: *Unreviewed*

Connection String

Connection string has the following format:

```
'etcd'://SERVER:PORT
```

- *Transport*
- *Channel*

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

port number used when no port is specified.

driver_name = 'python-etcd'

Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'etcd'

Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis (driver_type: 'redis'), etc...

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'})  
, 'heartbeats': False}
```

```
polling_interval = 3
```

Time to sleep between unsuccessful polls.

```
verify_connection(connection)
```

Verify the connection works.

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
```

4.2.15 Zookeeper Transport - `kombu.transport.zookeeper`

Zookeeper transport module for kombu.

Zookeeper based transport. This transport uses the built-in kazoo Zookeeper based queue implementation.

References

- https://zookeeper.apache.org/doc/current/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.

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: No
- Supports Priority: Yes
- Supports TTL: No

Connection String

Connects to a zookeeper node as:

```
zookeeper://SERVER:PORT/VHOST
```

The <vhost> becomes the base for all the other znodes. So we can use it like a vhost.

Transport Options

- *Transport*
- *Channel*

Transport

```
class kombu.transport.zookeeper.Transport(*args, **kwargs)
    Zookeeper Transport.

    class Channel(connection, **kwargs)
        Zookeeper Channel.

        property client

        channel_errors = (<class 'amqp.exceptions.ChannelError'>,)
            Tuple of errors that can happen due to channel/method failure.

        connection_errors = (<class 'amqp.exceptions.ConnectionError'>,)
            Tuple of errors that can happen due to connection failure.

        default_port = 2181
            port number used when no port is specified.

        driver_name = 'kazoo'
            Name of driver library (e.g. 'py-amqp', 'redis').

        driver_type = 'zookeeper'
            Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis
            (driver_type: 'redis'), etc...

        driver_version()

        polling_interval = 1
            Time to sleep between unsuccessful polls.
```

Channel

```
class kombu.transport.zookeeper.Channel(connection, **kwargs)
    Zookeeper Channel.

    property client
```

4.2.16 File-system Transport - kombu.transport.filesystem

File-system Transport module for kombu.

Transport using the file-system as the message store. Messages written to the queue are stored in *data_folder_in* directory and messages read from the queue are read from *data_folder_out* directory. Both directories must be created manually. Simple example:

- Producer:

```
import kombu

conn = kombu.Connection(
    'filesystem://', transport_options={
        'data_folder_in': 'data_in', 'data_folder_out': 'data_out'
    }
)
conn.connect()

test_queue = kombu.Queue('test', routing_key='test')

with conn as conn:
    with conn.default_channel as channel:
        producer = kombu.Producer(channel)
        producer.publish(
            {'hello': 'world'},
            retry=True,
            exchange=test_queue.exchange,
            routing_key=test_queue.routing_key,
            declare=[test_queue],
            serializer='pickle'
        )
```

- Consumer:

```
import kombu

conn = kombu.Connection(
    'filesystem://', transport_options={
        'data_folder_in': 'data_out', 'data_folder_out': 'data_in'
    }
)
conn.connect()

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

test_queue = kombu.Queue('test', routing_key='test')

with conn as conn:
    with conn.default_channel as channel:
        consumer = kombu.Consumer(
            conn, [test_queue], accept=['pickle']
        )
```

(continues on next page)

(continued from previous page)

```

consumer.register_callback(callback)
with consumer:
    conn.drain_events(timeout=1)

```

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: No
- Supports Priority: No
- Supports TTL: No

Connection String

Connection string is in the following format:

```
filesystem://
```

Transport Options

- `data_folder_in` - directory where are messages stored when written to queue.
- `data_folder_out` - directory from which are messages read when read from queue.
- `store_processed` - if set to True, all processed messages are backed up to `processed_folder`.
- `processed_folder` - directory where are backed up processed files.

- *Transport*
- *Channel*

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
    port number used when no port is specified.

driver_name = 'filesystem'
    Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'filesystem'
    Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis
    (driver_type: 'redis'), etc...

driver_version()

global_state = <kombu.transport.virtual.base.BrokerState object>
```

Channel

```
class kombu.transport.filesystem.Channel(connection, **kwargs)
    Filesystem Channel.

    data_folder_in
    data_folder_out
    processed_folder
    store_processed
    property transport_options
```

4.2.17 SQLAlchemy Transport Model - kombu.transport.sqlalchemy

SQLAlchemy Transport module for kombu.

Kombu transport using SQL Database as the message store.

Features

- Type: Virtual
- Supports Direct: yes
- Supports Topic: yes
- Supports Fanout: no
- Supports Priority: no
- Supports TTL: no

Connection String

```
sqla+SQL_ALCHEMY_CONNECTION_STRING
sqlalchemy+SQL_ALCHEMY_CONNECTION_STRING
```

For details about `SQL_ALCHEMY_CONNECTION_STRING` see SQLAlchemy Engine Configuration documentation. Examples:

```
# PostgreSQL with default driver
sqla+postgresql://scott:tiger@localhost/mydatabase

# PostgreSQL with psycopg2 driver
sqla+postgresql+psycopg2://scott:tiger@localhost/mydatabase

# PostgreSQL with pg8000 driver
sqla+postgresql+pg8000://scott:tiger@localhost/mydatabase

# MySQL with default driver
sqla+mysql://scott:tiger@localhost/foo

# MySQL with mysqlclient driver (a maintained fork of MySQL-Python)
sqla+mysql+mysqldb://scott:tiger@localhost/foo

# MySQL with PyMySQL driver
sqla+mysql+pymysql://scott:tiger@localhost/foo
```

Transport Options

- `queue_tablename`: Name of table storing queues.
- `message_tablename`: Name of table storing messages.

Moreover parameters of `sqlalchemy.create_engine()` function can be passed as transport options.

- *Transport*
- *Channel*

Transport

```
class kombu.transport.sqlalchemy.Transport(client, **kwargs)
```

The transport class.

```
class Channel(connection, **kwargs)
```

The channel class.

```
message_cls
```

```
queue_cls
```

```
property session
```

```
can_parse_url = True
```

Set to True if *Connection* should pass the URL unmodified.

```
connection_errors = (<class 'sqlalchemy.exc.OperationalError'>,)
    Tuple of errors that can happen due to connection failure.

default_port = 0
    port number used when no port is specified.

driver_name = 'sqlalchemy'
    Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'sql'
    Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis
    (driver_type: 'redis'), etc...

driver_version()
```

Channel

```
class kombu.transport.sqlalchemy.Channel(connection, **kwargs)
    The channel class.

    message_cls
    queue_cls
    property session
```

SQLAlchemy Transport Model - kombu.transport.sqlalchemy.models

Kombu transport using SQLAlchemy as the message store.

- *Models*

Models

```
class kombu.transport.sqlalchemy.models.Queue(name)
    The queue class.

    id = Column(None, Integer(), table=None, primary_key=True, nullable=False,
    default=Sequence('queue_id_sequence'))

    name = Column(None, String(length=200), table=None)

class kombu.transport.sqlalchemy.models.Message(payload, queue)
    The message class.

    id = Column(None, Integer(), table=None, primary_key=True, nullable=False,
    default=Sequence('message_id_sequence'))

    visible = Column(None, Boolean(), table=None, default=ColumnDefault(True))

    sent_at = Column('timestamp', DateTime(), table=None,
    onupdate=ColumnDefault(<function datetime.now>))

    payload = Column(None, Text(), table=None, nullable=False)

    version = Column(None, SmallInteger(), table=None, nullable=False,
    default=ColumnDefault(1))
```


4.2.18 Amazon SQS Transport - `kombu.transport.SQS`

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.

Other Features supported by this transport

Predefined Queues

The default behavior of this transport is to use a single AWS credential pair in order to manage all SQS queues (e.g. listing queues, creating queues, polling queues, deleting messages).

If it is preferable for your environment to use multiple AWS credentials, you can use the `'predefined_queues'` setting inside the `'transport_options'` map. This setting allows you to specify the SQS queue URL and AWS credentials for each of your queues. For example, if you have two queues which both already exist in AWS) you can tell this transport about them as follows:

```
transport_options = {
    'predefined_queues': {
        'queue-1': {
            'url': 'https://sqs.us-east-1.amazonaws.com/xxx/aaa',
            'access_key_id': 'a',
            'secret_access_key': 'b',
            'backoff_policy': {1: 10, 2: 20, 3: 40, 4: 80, 5: 320, 6: 640}, # optional
            'backoff_tasks': ['svc.tasks.tasks.task1'] # optional
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    },
    'queue-2': {
        'url': 'https://sqs.us-east-1.amazonaws.com/xxx/bbb',
        'access_key_id': 'c',
        'secret_access_key': 'd',
        'backoff_policy': {1: 10, 2: 20, 3: 40, 4: 80, 5: 320, 6: 640}, # optional
        'backoff_tasks': ['svc.tasks.tasks.task2'] # optional
    },
}
'sts_role_arn': 'arn:aws:iam::<xxx>:role/STSTest', # optional
'sts_token_timeout': 900 # optional
}

```

backoff_policy & backoff_tasks are optional arguments. These arguments automatically change the message visibility timeout, in order to have different times between specific task retries. This would apply after task failure.

AWS STS authentication is supported, by using sts_role_arn, and sts_token_timeout. sts_role_arn is the assumed IAM role ARN we are trying to access with. sts_token_timeout is the token timeout, defaults (and minimum) to 900 seconds. After the mentioned period, a new token will be created.

If you authenticate using [Okta](#) (e.g. calling [gimme-aws-creds](#)), you can also specify a 'session_token' to connect to a queue. Note that those tokens have a limited lifetime and are therefore only suited for short-lived tests.

Client config

In some cases you may need to override the botocore config. You can do it as follows:

```

transport_option = {
    'client-config': {
        'connect_timeout': 5,
    },
}

```

For a complete list of settings you can adjust using this option see <https://botocore.amazonaws.com/v1/documentation/api/latest/reference/config.html>

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: Yes
- Supports Priority: No
- Supports TTL: No

- *Transport*
- *Channel*

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.

class QoS(*channel*, *prefetch_count=0*)

Quality of Service guarantees implementation for SQS.

apply_backoff_policy(*routing_key*, *delivery_tag*, *backoff_policy*, *backoff_tasks*)

static extract_task_name_and_number_of_retries(*message*)

reject(*delivery_tag*, *requeue=False*)

Remove from transactional state and requeue message.

asynsqs(*queue=None*)

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.

generate_sts_session_token(*role_arn, token_expiry_seconds*)

is_secure

new_sqs_client(*region, access_key_id, secret_access_key, session_token=None*)

port

predefined_queues
Map of queue_name to predefined queue settings.

queue_name_prefix

region

regioninfo

sqs(*queue=None*)

supports_fanout
flag set if the channel supports fanout exchanges.

property transport_options

visibility_timeout

wait_time_seconds

channel_errors = (<class 'amqp.exceptions.ChannelError'>, <class 'botocore.exceptions.BotoCoreError'>)
Tuple of errors that can happen due to channel/method failure.

connection_errors = (<class 'amqp.exceptions.ConnectionError'>, <class 'botocore.exceptions.BotoCoreError'>, <class 'OSError'>)
Tuple of errors that can happen due to connection failure.

property default_connection_params

default_port = None
port number used when no port is specified.

driver_name = 'sqs'
Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'sqs'
Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis (driver_type: 'redis'), etc...

implements = {'asynchronous': True, 'exchange_type': frozenset({'direct'})
'heartbeats': False}

```
polling_interval = 1
    Time to sleep between unsuccessful polls.

wait_time_seconds = 0
```

Channel

```
class kombu.transport.SQS.Channel(*args, **kwargs)
    SQS Channel.

    class QoS(channel, prefetch_count=0)
        Quality of Service guarantees implementation for SQS.

        apply_backoff_policy(routing_key, delivery_tag, backoff_policy, backoff_tasks)

        static extract_task_name_and_number_of_retries(message)

        reject(delivery_tag, requeue=False)
            Remove from transactional state and requeue message.

    asynsqs(queue=None)

    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.

    generate_sts_session_token(role_arn, token_expiry_seconds)

    is_secure

    new_sqs_client(region, access_key_id, secret_access_key, session_token=None)
```

port
predefined_queues
 Map of queue_name to predefined queue settings.
queue_name_prefix
region
regioninfo
sqs(queue=None)
supports_fanout
 flag set if the channel supports fanout exchanges.
property transport_options
visibility_timeout
wait_time_seconds

Back-off policy

Back-off policy is using SQS visibility timeout mechanism altering the time difference between task retries. The mechanism changes message specific `visibility_timeout` from queue `Default visibility_timeout` to policy configured timeout. The number of retries is managed by SQS (specifically by the `ApproximateReceiveCount` message attribute) and no further action is required by the user.

Configuring the queues and backoff policy:

```
broker_transport_options = {
    'predefined_queues': {
        'my-q': {
            'url': 'https://ap-southeast-2.queue.amazonaws.com/123456/my-q',
            'access_key_id': 'xxx',
            'secret_access_key': 'xxx',
            'backoff_policy': {1: 10, 2: 20, 3: 40, 4: 80, 5: 320, 6: 640},
            'backoff_tasks': ['svc.tasks.tasks.task1']
        }
    }
}
```

`backoff_policy` dictionary where key is number of retries, and value is delay seconds between retries (i.e SQS visibility timeout) `backoff_tasks` list of task names to apply the above policy

The above policy:

Attempt	Delay
2nd attempt	20 seconds
3rd attempt	40 seconds
4th attempt	80 seconds
5th attempt	320 seconds
6th attempt	640 seconds

4.2.19 SLMQ Transport - `kombu.transport.SLMQ`

SoftLayer Message Queue transport module for kombu.

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: No
- Supports Priority: No
- Supports TTL: No

Connection String

Unreviewed

Transport Options

Unreviewed

- *Transport*
- *Channel*

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'>)
    Tuple of errors that can happen due to connection failure.

default_port = None
    port number used when no port is specified.

polling_interval = 1
    Time to sleep between unsuccessful polls.
```

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
```


4.2.20 Pyro Transport - `kombu.transport.pyro`

Pyro transport module for kombu.

Pyro transport, and Kombu Broker daemon.

Requires the Pyro4 library to be installed.

Features

- Type: Virtual
- Supports Direct: Yes
- Supports Topic: Yes
- Supports Fanout: No
- Supports Priority: No
- Supports TTL: No

Connection String

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 Options

- *Transport*
- *Channel*
- *KombuBroker*

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

port number used when no port is specified.

driver_name = 'pyro'

Name of driver library (e.g. 'py-amqp', 'redis').

driver_type = 'pyro'

Type of driver, can be used to separate transports using the AMQP protocol (driver_type: 'amqp'), Redis (driver_type: 'redis'), etc...

driver_version()

global_state = <kombu.transport.virtual.base.BrokerState object>

shared_queues

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

KombuBroker

4.3 Kombu Asynchronous

4.3.1 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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.3.5 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.

4.3.6 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.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_interface (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`

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 property 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**

4.3.7 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.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_interface** (*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

```

```

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 property 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**

4.3.8 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*)

4.3.9 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.

4.3.10 Amazon AWS Connection - `kombu.asynchronous.aws.connection`

Amazon AWS Connection.

```
class kombu.asynchronous.aws.connection.AsyncConnection(sqs_connection, http_client=None,
                                                         **kwargs)
```

Async AWS Connection.

```
get_http_connection()
```

```
class kombu.asynchronous.aws.connection.AsyncHTTPConnection(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_interface** (*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 kombu.asynchronous.aws.connection.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)

```

4.3.11 Async Amazon SQS Client - `kombu.asynchronous.aws.sqs`

4.3.12 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, queue_url, number_messages=1, visibility_timeout=None,
                 attributes=('ApproximateReceiveCount',), 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)
```

4.3.13 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

4.3.14 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.
```

4.4 Kombu utils

4.4.1 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.
```

4.4.2 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.

```
hashvalue
```

`kombu.utils.collections.eqhash(o)`
 Call `obj.__eqhash__`.

4.4.3 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.

4.4.4 Debugging Utilities - `kombu.utils.debug`

Debugging support.

`class kombu.utils.debug.Logwrapped(instance, logger=None, ident=None)`
 Wrap all object methods, to log on call.

`kombu.utils.debug.setup_logging(loglevel=10, loggers=None)`
 Setup logging to stdout.

4.4.5 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.

4.4.6 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 be 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.

4.4.7 Async I/O Selectors - `kombu.utils.eventio`

Selector Utilities.

`kombu.utils.eventio.poll(*args, **kwargs)`

Create new poller instance.

4.4.8 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()` → a set-like object providing a view on D's items

`iteritems()`

`iterkeys()`

`itervalues()`

`keys()` → a set-like object providing a view on D's 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() → an object providing a view on D's values

`kombu.utils.functional.dictfilter(d=None, **kw)`

Remove all keys from dict d whose value is None.

`kombu.utils.functional.is_list(obj, 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.

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.maybe_list(obj, scalars=(<class 'collections.abc.Mapping'>, <class 'str'>))`

Return list of one element if l is a scalar.

`kombu.utils.functional.memoize(maxsize=None, keyfun=None, Cache=<class
'kombu.utils.functional.LRUCache'>)`

Decorator to cache function return value.

`kombu.utils.functional.retry_over_time(fun, catch, args=None, kwargs=None, errback=None,
max_retries=None, interval_start=2, interval_step=2,
interval_max=30, callback=None, timeout=None)`

Retry the function over and over until max retries is exceeded.

For each retry we sleep a for a while before we try again, this interval is increased for every retry until the max seconds is reached.

Parameters

- **fun** (*Callable*) – The function to try
- **catch** (*Tuple[BaseException]*) – Exceptions to catch, can be either tuple or a single exception class.

Keyword Arguments

- **args** (*Tuple*) – Positional arguments passed on to the function.
- **kwargs** (*Dict*) – Keyword arguments passed on to the function.
- **errback** (*Callable*) – Callback for when an exception in `catch` is raised. The callback must take three arguments: `exc`, `interval_range` and `retries`, where `exc` is the exception instance, `interval_range` is an iterator which return the time in seconds to sleep next, and `retries` is the number of previous retries.
- **max_retries** (*int*) – Maximum number of retries before we give up. If neither of this and timeout is set, we will retry forever. If one of this and timeout is reached, stop.
- **interval_start** (*float*) – How long (in seconds) we start sleeping between retries.

- **interval_step** (*float*) – By how much the interval is increased for each retry.
- **interval_max** (*float*) – Maximum number of seconds to sleep between retries.
- **timeout** (*int*) – Maximum seconds waiting before we give up.

4.4.9 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 >>> sym-
bol_by_name(TaskPool) is TaskPool True
```

4.4.10 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.

4.4.11 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.

4.4.12 Object/Property Utilities - kombu.utils.objects

Object Utilities.

class kombu.utils.objects.**cached_property**(*fget=None, fset=None, fdel=None*)

Implementation of Cached property.

deleter(*fdel*)

setter(*fset*)

4.4.13 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.

4.4.14 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.

4.4.15 Time Utilities - kombu.utils.time

Time Utilities.

kombu.utils.time.maybe_s_to_ms(*v*)

Convert seconds to milliseconds, but return None for None.

4.4.16 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_ssl_cert_reqs(*query_value*)

Given the query parameter for ssl_cert_reqs, return the SSL constant or None.

kombu.utils.url.parse_url(*url*)

Parse URL into mapping of components.

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: str, hostname: str, port: int, username: str, password: str, path: str, query: collections.abc.Mapping)`

Named tuple representing parts of the URL.

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

4.4.17 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()`.

CHANGE HISTORY

5.1 5.2.0

release-date soon

release-by Naomi Elstein

- v 1.4.x (#1338).
- stop mentioning librabbitmq (#1381).
- Merge branch 'master' of <https://github.com/celery/kombu>
- test new pytest version (#1383).
- drop python 3.6 from CI (#1382).
- Use ANY from unittest instead of case.mock.
- Fix missing dependency to redis in docs requirements.
- [pre-commit.ci] pre-commit autoupdate.
- Remove dependency to case (#1389).
- Fix: check redis response type.
- [pre-commit.ci] pre-commit autoupdate (#1393).
- py3.7+ on setup (#1392).
- Prevent caching of oid in pidbox (#1394).
- Added unittests for #1394 .
- fix flake8 in kombu/asynchronous/aws/connection.py (#1397).
- [pre-commit.ci] pre-commit autoupdate.
- Fix test_pidbox unittests to support non-linux platforms (#1398).
- [pre-commit.ci] pre-commit autoupdate.
- remove bdist.
- add python 3.10 to CI & fix other issues (#1402).
- try to fix CI (#1407).
- Dont failfast when pypy3 tests fail (#1408).
- Return empty list instead of InconsistencyError when exchange table is empty.
- [pre-commit.ci] pre-commit autoupdate.

5.2 5.2.0rc1

release-date 2021-09-07 7:00 P.M UTC+6:00

release-by Asif Saif Uddin

- Remove backward compatible code not used anymore (#1344).
- Add support for setting redis username (#1351).
- Add support for Python 3.9.
- Use hostname from URI when server_host is None.
- Use Python's built-in json module by default, instead of simplejson.
- SQS Channel.predefined_queues should be {} if not defined.
- Add global key prefix for keys set by Redis transporter (#1349).
- fix: raise BrokenPipeError (#1231).
- fix: add missing commands to prefix.
- Make BrokerState Transport specific.
- Tests & Docs cleanup.

5.3 5.1.0

release-date 2021-05-23 7:00 P.M UTC+3:00

release-by Omer Katz

- Fix queue names special characters replacement for Azure Service Bus. (#1324)
- Add support for SQLAlchemy 1.4. (#1328)
- Coerce seconds argument to a floating point number in Timer.enter_after. (#1330)
- Add accept parameter to SimpleQueue class. (#1140)
- prepare_accept_content() now raises SerializerNotInstalled instead of KeyError. (#1343)

5.4 5.1.0b1

release-date 2021-04-01 10:30 P.M UTC+6:00

release-by Asiff Saif Uddin

- Wheels are no longer universal.
- Revert “Added redis transport key_prefix from envvars”.
- Redis Transport: Small improvements of *SentinelChannel* (#1253).
- Fix pidbox not using default channels.
- Revert “on worker restart - restore visible regardless to time (#905)”.
- Add vine to dependencies.
- Pin urllib3<1.26 to fix failing unittests.

- Add timeout to producer publish (#1269).
- Remove python2 compatibility code (#1277).
- redis: Support Sentinel with SSL.
- Support for Azure Service Bus 7.0.0 (#1284).
- Allow specifying session token (#1283).
- kombu/asynchronous/http/curl: implement `_set_timeout`.
- Disable namedtuple to object feature in simplejson (#1297).
- Update to tox docker 2.0.
- SQS back-off policy (#1301).
- Fixed SQS unittests.
- Fix: non kombu json message decoding in SQS transport (#1306).
- Add Github Actions CI (#1309).
- Update default pickle protocol version to 4 (#1314).
- Update connection.py (#1311).
- Drop support for the lzma backport.
- Drop obsolete code importing pickle (#1315).
- Update default login method for librabbitmq and pyamqp (#936).
- SQS Broker - handle STS authentication with AWS (#1322).
- Min py-amqp version is v5.0.6 (#1325).
- Numerous docs & example fixes.
- Use a thread-safe implementation of `cached_property` (#1316).

5.5 5.0.2

release-date 2020-09-06 6:30 P.M UTC+3:00

release-by Omer Katz

- Bump required amqp version to 5.0.0.

5.6 5.0.1

release-date 2020-08-23 19:10 P.M UTC+3:00

release-by Omer Katz

- Removed kombu.five from the reference documentation since it no longer exists
- Adjusted the stable documentation's version in Sphinx's configuration since that was overlooked in the latest release

5.7 5.0.0

release-date 2020-08-05 16:00 P.M UTC+3:00

release-by Omer Katz

- **BREAKING CHANGE:** Dropped support for Python 2 (#1232)
- Add an SQS transport option for custom botocore config (#1219)

5.8 4.6.11

release-date 2020-06-24 1.15 P.M UTC+6:00

release-by Asif Saif Uddin

- Revert incompatible changes in #1193 and additional improvements (#1211)
- Default_channel should reconnect automatically (#1209)

5.9 4.6.10

release-date 2020-06-03 10.45 A.M UTC+6:00

release-by Asif Saif Uddin

- Doc improvement.
- set _connection in _ensure_connection (#1205)
- Fix for the issue #1172
- reuse connection [bug fix]

5.10 4.6.9

release-date 2020-06-01 14.00 P.M UTC+6:00

release-by Asif Saif Uddin

- Prevent failure if AWS creds are not explicitly defined on predefined.
- Raise RecoverableConnectionError in maybe_declare with retry on and.
- Fix for the issue #1172 .
- possible fix for #1174 .
- Fix: make SQLAlchemy Channel init thread-safe
- Added integration testing infrastructure for RabbitMQ
- Initial redis integration tests implementation
- SQLAlchemy transport: Use Query.with_for_update() instead of deprecated
- Fix Consumer Encoding
- Added Integration tests for direct, topic and fanout exchange types

- Added TTL integration tests
- Added integration tests for priority queues
- fix 100% cpu usage on linux while using sqs
- Modified Mutex to use redis LuaLock implementation
- Fix: eliminate remaining race conditions from SQLAlchemy Channel
- Fix connection imaybe_declare (#1196)
- Fix for issue #1198: Celery crashes in cases where there aren't enough
- Ensure connection when connecting to broker
- update pyamqp to 2.6 with optional cythonization

5.11 4.6.8

release-date 2020-03-29 20:45 A.M UTC+6:00

release-by Asif Saif Uddin

- Added support for health_check_interval option in broker_transport_options (#1145)
- Added retry_on_timeout parameter to Redis Channel (#1150)
- Added support for standard values for ssl_cert_reqs query parameter for Redis (#1139)
- Added predefined_queues option to SQS transport (#1156)
- Added ssl certificate verification against ca certificates when amqps is used for pyamqp transport (#1151)
- Fix issue (#701) where kombu.transport.redis.Mutex is broken in python 3 (#1141)
- Fix brop error in Redis Channel (#1144)

5.12 4.6.7

release-date 2019-12-07 20:45 A.M UTC+6:00

release-by Asif Saif Uddin

- Use importlib.metadata from the standard library on Python 3.8+ (#1086).
- Add peek lock settings to be changed using transport options (#1119).
- Fix redis health checks (#1122).
- Reset ready before execute callback (#1126).
- Add missing parameter queue_args in kombu.connection.SimpleBuffer (#1128)

5.13 4.6.6

release-date 2019-11-11 00:15 A.M UTC+6:00

release-by Asif Saif Uddin

- Revert `_lookup_direct` and related changes of redis.
- Python 3.8 support
- Fix `'NoneType'` object has no attribute `'can_read'` bug of redis transport
- Issue #1019 Fix redis transport socket timeout
- Add wait timeout settings to receive queue message (#1110)
- Bump py-amqp to 2.5.2

5.14 4.6.5

release-date 2019-09-30 19:30 P.M UTC+6:00

release-by Asif Saif Uddin

- Revert `_lookup_api` and correct redis implemetnation.
- Major overhaul of redis test cases by adding more full featured fakeredis module.
- Add more test cases to boost coverage of kombu redis transport.
- Refactor the producer consumer test cases to be based on original mocks and be passing
- Fix lingering line length issue in test.
- Sanitise url when `include_password` is false
- Pinned pycurl to 7.43.0.2 as it is the latest build with wheels provided
- Bump py-amqp to 2.5.2

5.15 4.6.4

release-date 2019-08-14 22:45 P.M UTC+6:00

release-by Asif Saif Uddin

- Use `importlib-metadata` instead of `pkg_resources` for better performance
- Allow users to switch URLs while omitting the resource identifier (#1032)
- Don't stop receiving tasks on 503 SQS error. (#1064)
- Fix maybe declare (#1066)
- Revert "Revert "Use `SIMEMBERS` instead of `SMEMBERS` to check for queue (Redis Broker)"
- Fix MongoDB backend to work properly with TTL (#1076)
- Make sure that `max_retries=0` is treated differently than `None` (#1080)
- Bump py-amqp to 2.5.1

5.16 4.6.3

release-date 2019-06-15 12:45 A.M UTC+6:00

release-by Asif Saif Uddin

- Revert FastUUID for kombu 4.6

5.17 4.6.2

release-date 2019-06-15 12:45 A.M UTC+6:00

release-by Asif Saif Uddin

- Fix sbugs and regressions

5.18 4.6.1

release-date 2019-06-06 10:30 A.M UTC+6:00

release-by Asif Saif Uddin

- Fix some newly introduced bug in kombu 4.6

5.19 4.6.0

release-date 2019-05-30 15:30 P.M UTC+6:00

release-by Asif Saif Uddin

- Dropped python 3.4
- Bump py-amqp to 2.5.0
- Use SIMMEMBERS instead of SMEMBERS to check for queue (redis broker)
 - Add `_lookup_direct` method to virtual channel. (#994)

Add possibility to optimize lookup for queue in direct exchange set.

- Add `_lookup_direct` method to redis virtual channel. (#994)

Use *SISMEMBER* instead of *SMEMBERS* command to check if queue exists in a set. Time complexity is increased from $O(N)$ to $O(1)$ where N is the set cardinality.

Contributed by **Stevan Milic** and **Asif Saif Uddin**

- Include priority in properties only if it's not None. Since we attempt to serialize the priority property if it exists in the dictionary it must be an integer.

Contributed by **Omer Katz**

- Removed dangerous default mutable arguments from function definitions where appropriate.

Contributed by **Todd Cook**

- Codebase improvements and fixes by:
 - **Omer Katz**

– Asif Saif Uddin

5.20 4.5.0

release-date 2019-03-3 18:30 P.M UTC+3:00

release-by Omer Katz

- The Redis transport now supports a custom separator for keys.

Previously when storing a key in Redis which represents a queue we used the hardcoded value `\x06\x16` separator to store different attributes of the queue in the queue's name.

The separator is now configurable using the `sep` transport option:

```
with Connection('redis://', transport_options={
    'sep': ':',
}):
    # ...
    pass
```

Contributed by **Joris Beckers**

- When the SQS server returns a timeout we ignore it and keep trying instead of raising an error.

This will prevent Celery from raising an error and hanging.

Contributed by **Erwin Rossen**

- Properly declare async support for the Qpid transport.

If you are using this transport we strongly urge you to upgrade.

Contributed by **Rohan McGovern**

- Revert `celery/kombu#906` and introduce unique broadcast queue names as an optional keyword argument.

If you want each broadcast queue to have a unique name specify `unique=True`:

```
>>> from kombu.common import Broadcast
>>> q = Broadcast(queue='foo', unique=True)
>>> q.name
'foo.7ee1ac20-cda3-4966-aaf8-e7a3bb548688'
>>> q = Broadcast(queue='foo')
>>> q.name
'foo'
```

- Codebase improvements and fixes by:

– **Omer Katz**

5.21 4.4.0

release-date 2019-03-3 9:00 P.M UTC+2:00

release-by Omer Katz

- Restore bz2 import checks in compression module.

The checks were removed in [celery/kombu#938](#) due to assumption that it only affected Jython. However, bz2 support can be missing in Pythons built without bz2 support.

Contributed by **Patrick Woods**

- Fix regression that occurred in 4.3.0 when parsing Redis Sentinel master URI containing password.

Contributed by **Peter Lithhammer**

- Handle the case when only one Redis Sentinel node is provided.

Contributed by **Peter Lithhammer**

- Support SSL URL parameters correctly for *rediss://* URIs.

Contributed by **Paul Bailey**

- Revert [celery/kombu#954](#). Instead bump the required redis-py dependency to 3.2.0 to include this fix [andymccurdy/redis-py@4e1e748](#).

Contributed by **Peter Lithhammer**

- Added support for broadcasting using a regular expression pattern or a glob pattern to multiple Pidboxes.

Contributed by **Jason Held**

5.22 4.3.0

release-date 2019-01-14 7:00 P.M UTC+2:00

release-by Omer Katz

- Added Python 3.7 support.

Contributed by **Omer Katz, Mads Jensen** and **Asif Saif Uddin**

- Avoid caching queues which are declared with a TTL.

Queues that are declared with a TTL are now also be excluded from the in-memory cache in case they expire between publishes on the same channel.

Contributed by **Matt Yule-Bennett**

- Added an index to the Message table for the SQLAlchemy transport.

The index allows to effectively sorting the table by the message's timestamp.

Note: We do not provide migrations for this model yet. You will need to add the index manually if you are already using the SQLAlchemy transport.

The syntax may vary between databases. Please refer to your database's documentation for instructions.

Contributed by **Mikhail Shcherbinin**

- Added a timeout that limits the amount of time we retry to reconnect to a transport.

Contributed by **:github_user:`tothegump`**

- `:class:celery.asynchronous.hub.Hub` is now reentrant.

This allows calling `celery.bin.celery.main()` to revive a worker in the same process after rescuing from shutdown (`:class:SystemExit`).

Contributed by **Alan Justino da Silva**

- Queues now accept string exchange names as arguments as documented.

Tests were added to avoid further regressions.

Contributed by **Antonio Gutierrez**

- Specifying names for broadcast queues now work as expected.

Previously, named broadcast queues did not create multiple queues per worker. They incorrectly declared the named queue which resulted in one queue per fanout exchange, thus missing the entire point of a fanout exchange. The behavior is now matched to unnamed broadcast queues.

Contributed by **Kuan Hsuan-Tso**

- When initializing the Redis transport in conjunction with `gevent` restore all unacknowledged messages to queue.

Contributed by **Gal Cohen**

- Allow `:class:kombu.simple.SimpleQueue` to pass `queue_arguments` to Queue object.

This allows `:class:kombu.simple.SimpleQueue` to connect to RabbitMQ queues with custom arguments like `'x-queue-mode'='lazy'`.

Contributed by **C Blue Neeh**

- Add support for `'rediss'` scheme for secure Redis connections.

The `rediss` scheme defaults to the least secure form, as there is no suitable default location for `ca_certs`. The recommendation would still be to follow the documentation and specify `broker_use_ssl` if coming from celery.

Contributed by **Daniel Blair**

- Added the Azure Storage Queues transport.

The transport is implemented on top of Azure Storage Queues. This offers a simple but scalable and low-cost PaaS transport for Celery users in Azure. The transport is intended to be used in conjunction with the Azure Block Blob Storage backend.

Contributed by **Clemens Wolff**, **:github_user:`@ankurokok`**, **Denis Kisselev**, **Evandro de Paula**, **Martin Peck** and **:github_user:`@michaelperel`**

- Added the Azure Service Bus transport.

The transport is implemented on top of Azure Service Bus and offers PaaS support for more demanding Celery workloads in Azure. The transport is intended to be used in conjunction with the Azure CosmosDB backend.

Contributed by **Clemens Wolff**, **:github_user:`@ankurokok`**, **Denis Kisselev**, **Evandro de Paula**, **Martin Peck** and **:github_user:`@michaelperel`**

- Drop remaining mentions of Jython support completely.

Contributed by **Asif Saif Uddin** and **Mads Jensen**

- When publishing messages to the `Pidbox`, retry if an error occurs.

Contributed by **Asif Saif Uddin**

- Fix infinite loop in `:method:kombu.asynchronous.hub.Hub.create_loop`.

Previous attempt to fix the problem (PR kombu/760) did not consider an edge case. It is now fixed.

Contributed by **Vsevolod Strukchinsky**

- Worker shutdown no longer duplicates messages when using the SQS broker.

Contributed by **Mintu Kumar Sah**

- When using the SQS broker, prefer boto's default region before our hardcoded default.

Contributed by **Victor Villas**

- Fixed closing of shared redis sockets which previously caused Celery to hang.

Contributed by **Alexey Popravka**

- the `Pyro` transport (`kombu.transport.pyro`) now works with recent Pyro versions. Also added a Pyro Kombu Broker that this transport needs for its queues.

Contributed by **Irmen de Jong**

- Handle non-base64-encoded SQS messages.

Fix contributed by **Tim Li, Asif Saif Uddin** and **Omer Katz**.

- Move the handling of Sentinel failures to the redis library itself.

Previously, Redis Sentinel worked only if the first node's sentinel service in the URI was up. A server outage would have caused downtime.

Contributed by **Brian Price**

- When using Celery and the pickle serializer with binary data as part of the payload, `UnicodeDecodeError` would be raised as the content was not utf-8. We now replace on errors.

Contributed by **Jian Dai**

- Allow setting `:method:boto3.sqs.create_queue` Attributes via `transport_options`.

Contributed by **Hunter Fernandes**

- Fixed infinite loop when `entity.channel` is replaced by `revive()` on connection drop.

Contributed by **Tzach Yarimi**

- Added optional support for Brotli compression.

Contributed by **Omer Katz**

- When using the SQS broker, FIFO queues with names that ended with the 'f' letter were incorrectly parsed. This is now fixed.

Contributed by **Alex Vishnya** and **Ilya Konstantinov**

- Added optional support for LZMA compression.

Contributed by **Omer Katz**

- Added optional support for ZStandard compression.

Contributed by **Omer Katz**

- Require py-amqp 2.4.0 as the minimum version.

Contributed by **Asif Saif Uddin**

- The value of `DISABLE_TRACEBACKS` environment variable is now respected on debug, info and warning logger level.

Contributed by **Ludovic Rivallain**

- As documented in kombu/#741 and eventlet/eventlet#415 there is a mismatch between the monkey-patched eventlet queue and the interface Kombu is expecting. This causes Celery to crash when the *broker_pool_limit* configuration option is set eventlet/eventlet#415 suggests that the mutex can be a noop. This is now the case.

Contributed by **Josh Morrow**

- Codebase improvements and fixes by:
 - **Omer Katz**
 - **Mads Jensen**
 - **Asif Saif Uddin**
 - **Lars Rinn**
- Documentation improvements by:
 - **Jon Dufresne**
 - **Fay Cheng**
 - **Asif Saif Uddin**
 - **Kyle Verhoog**
 - **Noah Hall**
 - `:github_user:`brabiega``

5.23 4.2.2-post1

release-date 2019-01-01 04:00 P.M IST

release-by Omer Katz

Note: The previous release contained code from master. It is now deleted from PyPi. Please use this release instead.

- No changes since previous release.

5.24 4.2.2

release-date 2018-12-06 04:30 P.M IST

release-by Omer Katz

- Support both Redis client version 2.x and version 3.x.

Contributed by **Ash Berlin-Taylor** and **Jeppe Fihl-Pearson**

5.25 4.2.1

release-date 2018-05-21 09:00 A.M IST

release-by Omer Katz

Note: The 4.2.0 release contained remains of the `async` module by accident. This is now fixed.

- Handle `librabbitmq` `fileno` raising a `ValueError` when socket is not connected.

Contributed by **Bryan Shelton**

5.26 4.2.0

release-date 2018-05-21 09:00 A.M IST

release-by Omer Katz

- Now passing `max_retries`, `interval_start`, `interval_step`, `interval_max` parameters from broker `transport_options` to `ensure_connection()` when returning `default_connection()` (Issue #765).

Contributed by **Anthony Lukach**.

- Qpid: messages are now durable by default

Contributed by **David Davis**

- Kombu now requires version 2.10.4 or greater of the `redis` library, in line with Celery

Contributed by **Colin Jeanne**

- Fixed `ImportError` in some environments with outdated `simplejson`

Contributed by **Aaron Morris**

- MongoDB: fixed failure on MongoDB versions with an “-rc” tag

Contributed by **dust8**

- Ensure periodic polling frequency does not exceed timeout in virtual transport

Contributed by **Arcadiy Ivanov**

- Fixed string handling when using `python-future` module

Contributed by **John Koehl**

- Replaced “`async`” with “`asynchronous`” in preparation for Python 3.7

Contributed by **Thomas Achtemichuk**

- Allow removing pool size limit when in use

Contributed by **Alex Hill**

- Codebase improvements and fixes by:

- **j2gg0s**
- **Jon Dufresne**
- **Jonas Lergell**
- **Mads Jensen**

- Nicolas Delaby
- Omer Katz
- Documentation improvements by:
 - Felix Yan
 - Harry Moreno
 - Mads Jensen
 - Omer Katz
 - Radha Krishna. S.
 - Wojciech Matyśkiewicz

5.27 4.1.0

release-date 2017-07-17 04:45 P.M MST

release-by Anthony Lukach

- SQS: Added support for long-polling on all supported queries. Fixed bug causing error on parsing responses with no retrieved messages from SQS.

Contributed by **Anthony Lukach**.

- Async hub: Fixed potential infinite loop while performing todo tasks (Issue celery/celery#3712).
- Qpid: Fixed bug where messages could have duplicate `delivery_tag` (Issue #563).

Contributed by **bmbouter**.

- MongoDB: Fixed problem with using `readPreference` option at pymongo 3.x.

Contributed by **Mikhail Elovskikh**.

- Re-added support for `:ypi:SQLAlchemy`

Contributed by **Amin Ghadersohi**.

- SQS: Fixed bug where hostname would default to `localhost` if not specified in settings.

Contributed by **Anthony Lukach**.

- Redis: Added support for reading password from transport URL (Issue #677).

Contributed by **George Psarakis**.

- RabbitMQ: Ensured safer encoding of queue arguments.

Contributed by **Robert Kopaczewski**.

- Added fallback to `:func:uuid.uuid5` in `:func:generate_oid` if `:func:uuid.uuid3` fails.

Contributed by **Bill Nottingham**.

- Fixed race condition and innacurate timeouts for `:class:kombu.simple.SimpleBase` (Issue #720).

Contributed by **c-nichols**.

- Zookeeper: Fixed last chroot character trimming

Contributed by **Dima Kurguzov**.

- RabbitMQ: Fixed bug causing an exception when attempting to close an already-closed connection (Issue #690).

Contributed by **eavictor**.

- Removed deprecated use of StopIteration in generators and invalid regex escape sequence.

Contributed by **Jon Dufresne**.

- Added Python 3.6 to CI testing.

Contributed by **Jon Dufresne**.

- SQS: Allowed endpoint URL to be specified in the boto3 connection.

Contributed by **georgepsarakis**.

- SQS: Added support for Python 3.4.

Contributed by **Anthony Lukach**.

- SQS: kombu[sqs] now depends on **boto3** (no longer using **boto**).
 - Adds support for Python 3.4+
 - Adds support for FIFO queues (Issue #678) and (Issue celery/celery#3690)
 - Avoids issues around a broken endpoints file (Issue celery/celery#3672)

Contributed by **Mischa Spiegelmock** and **Jerry Seutter**.

- Zookeeper: Added support for delaying task with Python 3.

Contributed by **Dima Kurguzov**.

- SQS: Fixed bug where kombu.transport.SQS.drain_events() did not support callback argument (Issue #694).

Contributed by **Michael Montgomery**.

- Fixed bug around modifying dictionary size while iterating over it (Issue #675).

Contributed by **Felix Yan**.

- etcd: Added handling for EtcdException exception rather than EtcdError.

Contributed by **Stephen Milner**.

- Documentation improvements by:

- **Mads Jensen**
- **Matias Insaurrealde**
- **Omer Katz**
- **Dmitry Dygalo**
- **Christopher Hoskin**

5.28 4.0.2

release-date 2016-12-15 03:31 P.M PST

release-by Ask Solem

- Now depends on `amqp` 2.1.4

This new version takes advantage of TCP Keepalive settings on Linux, making it better at detecting closed connections, also in failover conditions.

- Redis: Priority was reversed so, e.g. priority 0 became priority 9.

5.29 4.0.1

release-date 2016-12-07 06:00 P.M PST

release-by Ask Solem

- Now depends on `amqp` 2.1.3

This new version takes advantage of the new `TCP_USER_TIMEOUT` socket option on Linux.

- Producer: Fixed performance degradation when default exchange specified (Issue #651).
- QPid: Switch to using `getattr` in `qpido.Transport.__del__` (Issue #658)

Contributed by **Patrick Creech**.

- QPid: Now uses monotonic time for timeouts.
- MongoDB: Fixed compatibility with Python 3 (Issue #661).
- Consumer: `__exit__` now skips cancelling consumer if connection-related error raised (Issue #670).
- MongoDB: Removes use of natural sort (Issue #638).

Contributed by **Anton Chaporgin**.

- Fixed wrong keyword argument `channel` error (Issue #652).

Contributed by **Toomore Chiang**.

- Safe argument to `urllib.quote` must be bytes on Python 2.x (Issue #645).
- Documentation improvements by:
 - **Carlos Edo**
 - **Cemre Mengu**

5.30 4.0

release-date 2016-10-28 16:45 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 2.0.

The new `py-amqp` version have been refactored for better performance, using modern Python socket conventions, and API consistency.

- No longer depends on `anyjson`.

Kombu will now only choose between `simplejson` and the built-in `json`.

Using the latest version of `simplejson` is recommended:

```
$ pip install -U simplejson
```

- Removed transports that are no longer supported in this version:

- Django ORM transport
- SQLAlchemy ORM transport
- Beanstalk transport
- ZeroMQ transport
- amqpplib transport (use `pyamqp`).

- API Changes

- Signature of `kombu.Message` now takes `body` as first argument.

It used to be `Message(channel, body=body, **kw)`, but now it's `Message(body, channel=channel, **kw)`.

This is unlikely to affect you, as the Kombu API does not have users instantiate messages manually.

- New SQS transport

Donated by NextDoor, with additional contributions from mdk.

Note: `kombu[sqs]` now depends on `pycurl`.

- New Consul transport.

Contributed by **Wido den Hollander**.

- New etcd transport.

Contributed by **Stephen Milner**.

- New Qpid transport.

It was introduced as an experimental transport in Kombu 3.0, but is now mature enough to be fully supported.

Created and maintained by **Brian Bouterse**.

- Redis: Priority 0 is now lowest, 9 is highest. (**backward incompatible**)

This to match how priorities in AMQP works.

Fix contributed by **Alex Koshelev**.

- Redis: Support for Sentinel

You can point the connection to a list of sentinel URLs like:

```
sentinel://0.0.0.0:26379;sentinel://0.0.0.0:26380/...
```

where each sentinel is separated by a ;. Multiple sentinels are handled by *kombu.Connection* constructor, and placed in the alternative list of servers to connect to in case of connection failure.

Contributed by **Sergey Azovskov**, and **Lorenzo Mancini**

- RabbitMQ Queue Extensions

New arguments have been added to *kombu.Queue* that lets you directly and conveniently configure the RabbitMQ queue extensions.

- *Queue(expires=20.0)*
Set queue expiry time in float seconds.
See *kombu.Queue.expires*.
- *Queue(message_ttl=30.0)*
Set queue message time-to-live float seconds.
See *kombu.Queue.message_ttl*.
- *Queue(max_length=1000)*
Set queue max length (number of messages) as int.
See *kombu.Queue.max_length*.
- *Queue(max_length_bytes=1000)*
Set queue max length (message size total in bytes) as int.
See *kombu.Queue.max_length_bytes*.
- *Queue(max_priority=10)*
Declare queue to be a priority queue that routes messages based on the *priority* field of the message.
See *kombu.Queue.max_priority*.

- RabbitMQ: *Message.ack* now supports the *multiple* argument.

If *multiple* is set to *True*, then all messages received before the message being acked will also be acknowledged.

- *amqps://* can now be specified to require SSL (Issue #610).
- *Consumer.cancel_by_queue* is now constant time.
- *Connection.ensure** now raises *kombu.exceptions.OperationalError*.

Things that can be retried are now reraised as *kombu.exceptions.OperationalError*.

- Redis: Fixed SSL support.

Contributed by **Robert Kolba**.

- New *Queue.consumer_arguments* can be used for the ability to set consumer priority via *x-priority*.

See <https://www.rabbitmq.com/consumer-priority.html>

Example:

```
Queue(
    'qname',
    exchange=Exchange('exchange'),
    routing_key='qname',
    consumer_arguments={'x-priority': 3},
)
```

- Queue/Exchange: `no_declare` option added (also enabled for internal amq. exchanges) (Issue #565).
- JSON serializer now calls `obj.__json__` for unsupported types.

This means you can now define a `__json__` method for custom types that can be reduced down to a built-in json type.

Example:

```
class Person:
    first_name = None
    last_name = None
    address = None

    def __json__(self):
        return {
            'first_name': self.first_name,
            'last_name': self.last_name,
            'address': self.address,
        }
```

- JSON serializer now handles datetimes, Django promise, UUID and Decimal.
- Beanstalk: Priority 0 is now lowest, 9 is highest. (**backward incompatible**)

This to match how priorities in AMQP works.

Fix contributed by **Alex Koshelev**.

- Redis: now supports SSL using the `ssl` argument to [Connection](#).
- Redis: Fanout exchanges are no longer visible between vhosts, and fanout messages can be filtered by patterns. (**backward incompatible**)

It was possible to enable this mode previously using the `fanout_prefix`, and `fanout_patterns` transport options, but now these are enabled by default.

If you want to mix and match producers/consumers running different versions you need to configure your kombu 3.x clients to also enable these options:

```
>>> Connection(transport_options={
    'fanout_prefix': True,
    'fanout_patterns': True,
})
```

- Pidbox: Mailbox new arguments: TTL and expiry.

Mailbox now supports new arguments for controlling message TTLs and queue expiry, both for the mailbox queue and for reply queues.

- `queue_expires` (float/int seconds).
- `queue_ttl` (float/int seconds).

- `reply_queue_expires` (float/int seconds).
- `reply_queue_ttl` (float/int seconds).

All take seconds in int/float.

Contributed by **Alan Justino**.

- `Exchange.delivery_mode` now defaults to `None`, and the default is instead set by `Producer.publish`.
- `Consumer` now supports a new `prefetch_count` argument, which if provided will force the consumer to set an initial prefetch count just before starting.
- Virtual transports now stores `priority` as a property, not in `delivery_info`, to be compatible with AMQP.
- `reply_to` argument to `Producer.publish` can now be `Queue` instance.
- Connection: There's now a new method `Connection.supports_exchange_type(type)` that can be used to check if the current transport supports a specific exchange type.
- SQS: Consumers can now read json messages not sent by Kombu.

Contributed by **Juan Carlos Ferrer**.

- SQS: Will now log the access key used when authentication fails.

Contributed by **Hank John**.

- Added new `kombu.mixins.ConsumerProducerMixin` for consumers that will also publish messages on a separate connection.
- Messages: Now have a more descriptive `repr`.

Contributed by **Joshua Harlow**.

- Async: HTTP client based on curl.
- Async: Now uses `poll` instead of `select` where available.
- MongoDB: Now supports priorities

Contributed by **Alex Koshelev**.

- Virtual transports now supports multiple queue bindings.

Contributed by **Federico Ficarelli**.

- Virtual transports now supports the anon exchange.

If when publishing a message, the exchange argument is set to "" (empty string), the `routing_key` will be regarded as the destination queue.

This will bypass the routing table completely, and just deliver the message to the queue name specified in the routing key.

- Zookeeper: Transport now uses the built-in support in kazoo to handle failover when using a list of server names.

Contributed by **Joshua Harlow**.

- `ConsumerMixin.run` now passes keyword arguments to `.consume`.

5.30.1 Deprecations and removals

- The deprecated method `Consumer.add_queue_from_dict` has been removed.

Use instead:

```
consumer.add_queue(Queue.from_dict(queue_name, **options))
```

- The deprecated function `kombu.serialization.encode` has been removed.

Use `kombu.serialization.dumps()` instead.

- The deprecated function `kombu.serialization.decode` has been removed.

Use `kombu.serialization.loads()` instead.

- Removed module `kombu.syn`

`detect_environment` has been moved to `kombu.utils.compat`

5.31 3.0.37

release-date 2016-10-06 05:00 P.M PDT

release-by Ask Solem

- Connection: Return value of `.info()` was no longer JSON serializable, leading to “itertools.cycle object not JSON serializable” errors (Issue #635).

5.32 3.0.36

release-date 2016-09-30 03:06 P.M PDT

release-by Ask Solem

- Connection: Fixed bug when cloning connection with alternate urls.

Fix contributed by Emmanuel Cazenave.

- Redis: Fixed problem with unix socket connections.

<https://github.com/celery/celery/issues/2903>

Fix contributed by Raphael Michel.

- Redis: Fixed compatibility with older redis-py versions (Issue #576).
- Broadcast now retains queue name when being copied/pickled (Issue #578).

5.33 3.0.35

release-date 2016-03-22 11:22 P.M PST

release-by Ask Solem

- msgpack: msgpack support now requires msgpack-python > 0.4.7.
- Redis: TimeoutError was no longer handled as a recoverable error.
- Redis: Adds the ability to set more Redis connection options using `Connection(transport_options={...})`.
 - `socket_connect_timeout`
 - `socket_keepalive` (requires redis-py > 2.10)
 - `socket_keepalive_options` (requires redis-py > 2.10)
- msgpack: Fixes support for binary/unicode data

5.34 3.0.34

release-date 2016-03-03 05:30 P.M PST

release-by Ask Solem

- Qpid: Adds async error handling.
Contributed by Brian Bouterse.
- Qpid: Delivery tag is now a UUID4 (Issue #563).
Fix contributed by Brian Bouterse.
- Redis: `Connection.as_uri()` returned malformed URLs when the `redis+socket` scheme was used (Issue celery/celery#2995).
- msgpack: Use binary encoding instead of utf-8 (Issue #570).

5.35 3.0.33

release-date 2016-01-08 06:36 P.M PST

release-by Ask Solem

- Now depends on `amqp` 1.4.9.
- Redis: Fixed problem with auxilliary connections causing the main consumer connection to be closed (Issue #550).
- Qpid: No longer uses threads to operate, to ensure compatibility with all environments (Issue #531).

5.36 3.0.32

release-date 2015-12-16 02:29 P.M PST

release-by Ask Solem

- Redis: Fixed bug introduced in 3.0.31 where the redis transport always connects to localhost, regardless of host setting.

5.37 3.0.31

release-date 2015-12-16 12:00 P.M PST

release-by Ask Solem

- Redis: Fixed bug introduced in 3.0.30 where socket was prematurely disconnected.
- Hub: Removed debug logging message: “Deregistered fd...” (Issue #549).

5.38 3.0.30

release-date 2015-12-07 12:28 A.M PST

release-by Ask Solem

- Fixes compatibility with uuid in Python 2.7.11 and 3.5.1.
Fix contributed by Kai Groner.
- Redis transport: Attempt at fixing problem with hanging consumer after disconnected from server.
- **Event loop:** Attempt at fixing issue with 100% CPU when using the Redis transport,
- Database transport: Fixed oracle compatibility.
An “ORA-00907: missing right parenthesis” error could manifest when using an Oracle database with the database transport.
Fix contributed by Deepak N.
- Documentation fixes
Contributed by Tommaso Barbugli.

5.39 3.0.29

release-date 2015-10-26 11:10 A.M PDT

release-by Ask Solem

- Fixed serialization issue for `bindings.as_dict()` (Issue #453).
Fix contributed by Sergey Tikhonov.
- Json serializer wrongly treated bytes as `ascii`, not `utf-8` (Issue #532).
- MongoDB: Now supports pymongo 3.x.
Contributed by Len Buckens.

- SQS: Tests passing on Python 3.
Fix contributed by Felix Yan

5.40 3.0.28

release-date 2015-10-12 12:00 PM PDT

release-by Ask Solem

Django transport migrations.

If you're using Django 1.8 and have already created the `kombu_transport_django` tables, you have to run a fake initial migration:

```
$ python manage.py migrate kombu_transport_django --fake-initial
```

- No longer compatible with South by default.
To keep using `kombu.transport.django` with South migrations you now need to configure a new location for the kombu migrations:

```
SOUTH_MIGRATION_MODULES = {  
    'kombu_transport_django':  
        'kombu.transport.django.south_migrations',  
}
```

- Keep old South migrations in `kombu.transport.django.south_migrations`.
- Now works with Redis < 2.10 again.

5.41 3.0.27

release-date 2015-10-09 3:10 PM PDT

release-by Ask Solem

- Now depends on `amqp` 1.4.7.
- Fixed `libSystem` import error on some macOS 10.11 (El Capitan) installations.

Fix contributed by Eric Wang.

- Now compatible with Django 1.9.
- Django: Adds migrations for the database transport.
- Redis: Now depends on `py-redis` 2.10.0 or later (Issue #468).
- QPid: Can now connect as `localhost` (Issue #519).

Fix contributed by Brian Bouterse.

- QPid: Adds support for `login_method` (Issue #502, Issue #499).
Contributed by Brian Bouterse.
- QPid: Now reads SASL mechanism from broker string (Issue #498).

Fix contributed by Brian Bouterse.

- QPid: Monitor thread now properly terminated on session close (Issue #485).

Fix contributed by Brian Bouterse.

- QPid: Fixed file descriptor leak (Issue #476).

Fix contributed by Jeff Ortel

- Docs: Fixed wrong order for endpoint arguments (Issue #473).
- ConsumerMixin: Connection error logs now include traceback (Issue #480).
- BaseTransport now raises RecoverableConnectionError when disconnected (Issue #507).
- Consumer: Adds `tag_prefix` option to modify how consumer tags are generated (Issue #509).

5.42 3.0.26

release-date 2015-04-22 06:00 P.M UTC

release-by Ask Solem

- Fixed compatibility with py-redis versions before 2.10.3 (Issue #470).

5.43 3.0.25

release-date 2015-04-21 02:00 P.M UTC

release-by Ask Solem

- pyamqp/librabbitmq now uses 5671 as default port when SSL is enabled (Issue #459).
- Redis: Now supports passwords in `redis+socket://:pass@host:port` URLs (Issue #460).
- `Producer.publish` now defines the `expiration` property in support of the [RabbitMQ per-message TTL extension](#).

Contributed by Anastasis Andronidis.

- Connection transport attribute now set correctly for all transports.

Contributed by Alex Koshelev.

- qpid: Fixed bug where the connection was not being closed properly.

Contributed by Brian Bouterse.

- `bindings` is now JSON serializable (Issue #453).

Contributed by Sergey Tikhonov.

- Fixed typo in error when yamll is not installed (said msgpack).

Contributed by Joshua Harlow.

- Redis: Now properly handles `redis.exceptions.TimeoutError` raised by `redis`.

Contributed by markow.

- qpid: Adds additional string to check for when connecting to qpid.

When we connect to qpid, we need to ensure that we skip to the next SASL mechanism if the current mechanism fails. Otherwise, we will keep retrying the connection with a non-working mech.

Contributed by Chris Duryee.

- qpid: Handle `NotFound` exceptions.

Contributed by Brian Bouterse.

- `Queue.__repr__` now makes sure return value is not unicode (Issue #440).
- qpid: `Queue.purge` incorrectly raised `AttributeError` if the does not exist (Issue #439).

Contributed by Brian Bouterse.

- Linux: Now ignores permission errors on `epoll` unregister.

5.44 3.0.24

release-date 2014-11-17 11:00 P.M UTC

release-by Ask Solem

- The `Qpid` broker is supported for Python 2.x environments. The `Qpid` transport includes full SSL support within Kombu. See the `kombu.transport.qpid` docs for more info.

Contributed by Brian Bouterse and Chris Duryee through support from Red Hat.

- Dependencies: `extra[librabbitmq]` now requires `librabbitmq 1.6.0`
- Docstrings for `TokenBucket` did not match implementation.

Fix contributed by Jesse Dhillon.

- `oid_from()` accidentally called `uuid.getnode()` but did not use the return value.

Fix contributed by Alexander Todorov.

- Redis: Now ignores errors when closing the underlying connection.
- Redis: Restoring messages will now use a single connection.
- `kombu.five.monotonic`: Can now be imported even if `ctypes` is not available for some reason (e.g. App Engine)
- Documentation: Improved example to use the `declare` argument to `Producer` (Issue #423).
- Django: Fixed `app_label` for older Django versions (`< 1.7`). (Issue #414).

5.45 3.0.23

release-date 2014-09-14 10:45 P.M UTC

release-by Ask Solem

- Django: Fixed bug in the Django 1.7 compatibility improvements related to autocommit handling.

Contributed by Radek Czajka.

- Django: The Django transport models would not be created on syncdb after app label rename (Issue #406).

5.46 3.0.22

release-date 2014-09-04 03:00 P.M UTC

release-by Ask Solem

- kombu.async: Min. delay between waiting for timer was always increased to one second.
- Fixed bug in itermessages where message is received after the with statement exits the block.

Fixed by Romyana Neykova

- **Connection.autoretry: Now works with functions missing wrapped attributes** (`__module__`, `__name__`, `__doc__`). Fixes #392.

Contributed by johtso.

- Django: Now sets custom app label for `kombu.transport.django` to work with recent changes in Django 1.7.
- SimpleQueue removed messages from the wrong end of buffer (Issue #380).
- Tests: Now using `unittest.mock` if available (Issue #381).

5.47 3.0.21

release-date 2014-07-07 02:00 P.M UTC

release-by Ask Solem

- Fixed remaining bug in `maybe_declare` for `auto_delete` exchanges.

Fix contributed by Roger Hu.

- MongoDB: Creating a channel now properly evaluates a connection (Issue #363).

Fix contributed by Len Buckens.

5.48 3.0.20

release-date 2014-06-24 02:30 P.M UTC

release-by Ask Solem

- Reverts change in 3.0.17 where `maybe_declare` caches the declaration of `auto_delete` queues and exchanges.

Fix contributed by Roger Hu.

- Redis: Fixed race condition when using `gevent` and the channel is closed.

Fix contributed by Andrew Rodionoff.

5.49 3.0.19

release-date 2014-06-09 03:10 P.M UTC

release-by Ask Solem

- The wheel distribution did not support Python 2.6 by failing to list the extra dependencies required.
- Durable and `auto_delete` queues/exchanges can be be cached using `maybe_declare`.

5.50 3.0.18

release-date 2014-06-02 06:00 P.M UTC

release-by Ask Solem

- A typo introduced in 3.0.17 caused `kombu.async.hub` to crash (Issue #360).

5.51 3.0.17

release-date 2014-06-02 05:00 P.M UTC

release-by Ask Solem

- `kombu[librabbitmq]` now depends on `librabbitmq` 1.5.2.
- Async: Event loop now selectively removes file descriptors for the mode it failed in, and keeps others (e.g read vs write).

Fix contributed by Roger Hu.

- CouchDB: Now works without `userid` set.

Fix contributed by Latitia M. Haskins.

- SQLAlchemy: Now supports recovery from connection errors.

Contributed by Felix Schwarz.

- Redis: Restore at shutdown now works when `ack` emulation is disabled.
- `kombu.common.eventloop()` accidentally swallowed socket errors.
- Adds `kombu.utils.url.sanitize_url()`

5.52 3.0.16

release-date 2014-05-06 01:00 P.M UTC

release-by Ask Solem

- `kombu[librabbitmq]` now depends on `librabbitmq` 1.5.1.
- Redis: Fixes `TypeError` problem in `unregister` (Issue #342).

Fix contributed by Tobias Schottdorf.

- Tests: Some unit tests accidentally required the `redis-py` library.

Fix contributed by Randy Barlow.

- `librabbitmq`: Would crash when using an older version of `librabbitmq`, now emits warning instead.

5.53 3.0.15

release-date 2014-04-15 09:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.4.5.
- RabbitMQ 3.3 changes QoS semantics (Issue #339).

See the RabbitMQ release notes here: <http://www.rabbitmq.com/blog/2014/04/02/breaking-things-with-rabbitmq-3-3/>

A new connection property has been added that can be used to detect whether the remote server is using this new QoS behavior:

```
>>> Connection('amqp://').qos_behavior_matches_spec
False
```

so if your application depends on the old semantics you can use this to set the `apply_global` flag appropriately:

```
def update_prefetch_count(channel, new_value):
    channel.basic_qos(
        0, new_value,
        not channel.connection.client.qos_behavior_matches_spec,
    )
```

- Users of `librabbitmq` is encouraged to upgrade to `librabbitmq` 1.5.0.
The `kombu[librabbitmq]` extra has been updated to depend on this version.
- Pools: Now takes transport options into account when comparing connections (Issue #333).
- MongoDB: Fixes Python 3 compatibility.
- Async: select: Ignore socket errors when attempting to unregister handles from the loop.
- Pidbox: Can now be configured to use a serializer other than `json`, but specifying a serializer argument to [Mailbox](#).

Contributed by Dmitry Malinovsky.

- Message decompression now works with Python 3.

Fix contributed by Adam Gaca.

5.54 3.0.14

release-date 2014-03-19 07:00 P.M UTC

release-by Ask Solem

- **MongoDB**: Now endures a connection failover (Issue #123).
Fix contributed by Alex Koshelev.
- **MongoDB**: Fixed `KeyError` when a replica set member is removed.
Also fixes celery#971 and celery/#898.
Fix contributed by Alex Koshelev.
- **MongoDB**: Fixed MongoDB broadcast cursor re-initialization bug.
Fix contributed by Alex Koshelev.
- **Async**: Fixed bug in lax semaphore implementation where in some usage patterns the limit was not honored correctly.
Fix contributed by Ionel Cristian Mărieș.
- **Redis**: Fixed problem with fanout when using Python 3 (Issue #324).
- **Redis**: Fixed `AttributeError` from attempting to close a non-existing connection (Issue #320).

5.55 3.0.13

release-date 2014-03-03 04:00 P.M UTC

release-by Ask Solem

- **Redis**: Fixed serious race condition that could lead to data loss.

The delivery tags were accidentally set to be an incremental number local to the channel, but the delivery tags need to be globally unique so that a message can not overwrite an older message in the backup store.

This change is not backwards incompatible and you are encouraged to update all your system using a previous version as soon as possible.
- Now depends on `amqp` 1.4.4.
- **Pidbox**: Now makes sure message encoding errors are handled by default, so that a custom error handler does not need to be specified.
- **Redis**: The fanout exchange can now use AMQP patterns to route and filter messages.

This change is backwards incompatible and must be enabled with the `fanout_patterns` transport option:

```
>>> conn = kombu.Connection('redis://', transport_options={
...     'fanout_patterns': True,
... })
```


When enabled the exchange will work like an `amqp` topic exchange if the binding key is a pattern.
This is planned to be default behavior in the future.
- **Redis**: Fixed `cyclic no such attribute` error.

5.56 3.0.12

release-date 2014-02-09 03:50 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.4.3.
- Fixes Python 3.4 logging incompatibility (Issue #311).
- Redis: Now properly handles unknown pub/sub messages.

Fix contributed by Sam Stavinoha.

- `amqplib`: Fixed bug where more bytes were requested from the socket than necessary.

Fix contributed by Ionel Cristian Mărieș.

5.57 3.0.11

release-date 2014-02-03 05:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.4.2.
- Now always trusts messages of type *application/data* and *application/text* or which have an unspecified content type (Issue #306).
- Compression errors are now handled as decode errors and will trigger the `Consumer.on_decode_error` callback if specified.
- New `kombu.Connection.get_heartbeat_interval()` method that can be used to access the negotiated heartbeat value.
- ***kombu.common.oid_for* no longer uses the MAC address of the host, but** instead uses a process-wide UUID4 as a node id.

This avoids a call to `uuid.getnode()` at module scope.

- `Hub.add`: Now normalizes registered fileno.
- `SQS`: Fixed bug where the prefetch count limit was not respected.

Contributed by Ionel Cristian Mărieș.

5.58 3.0.10

release-date 2014-01-17 05:40 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.4.1.
- `maybe_declare` now raises a “recoverable connection error” if the channel is disconnected instead of a `ChannelError` so that the operation can be retried.
- Redis: `Consumer.cancel()` is now thread safe.

This fixes an issue when using `gevent/eventlet` and a message is handled after the consumer is canceled resulting in a “message for queue without consumers” error.

- Retry operations would not always respect the `interval_start` value when calculating the time to sleep for (Issue #303).

Fix contributed by Antoine Legrand.

- Timer: Fixed “unhashable type” error on Python 3.
- Hub: Do not attempt to unregister operations on an already closed poller instance.

5.59 3.0.9

release-date 2014-01-13 05:30 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.4.0.
- Redis: Basic cancel for fanout based queues now sends a corresponding `UNSUBSCRIBE` command to the server.

This fixes an issue with `pidbox` where reply messages could be received after the consumer was canceled, giving the “message to queue without consumers” error.

- MongoDB: Improved connection string and options handling (Issue #266 + Issue #120).

Contributed by Alex Koshelev.

- SQS: Limit the number of messages when receiving in batch to 10.

This is a hard limit enforced by Amazon so the `sqs` transport must not exceed this value.

Fix contributed by Eric Reynolds.

- ConsumerMixin: `consume` now checks heartbeat every time the socket times out.

Contributed by Dustin J. Mitchell.

- Retry Policy: A max retries of 0 did not retry forever.

Fix contributed by Antoine Legrand.

- Simple: If passing a Queue object the simple utils will now take default routing key from that queue.

Contributed by Fernando Jorge Mota.

- `repr(producer)` no longer evaluates the underlying channel.
- Redis: The map of Redis error classes are now exposed at the module level using the `kombu.transport.redis.get_redis_error_classes()` function.
- Async: `Hub.close` now sets `.poller` to `None`.

5.60 3.0.8

release-date 2013-12-16 05:00 P.M UTC

release-by Ask Solem

- Serializer: `loads` and `dumps` now wraps exceptions raised into `DecodeError` and `kombu.exceptions.EncodeError` respectively.

Contributed by Ionel Cristian Maries

- Redis: Would attempt to read from the wrong connection if a `select/epoll/kqueue` exception event happened.

Fix contributed by Michael Nelson.

- Redis: Disabling ack emulation now works properly.

Fix contributed by Michael Nelson.

- Redis: `IOError` and `OSError` are now treated as recoverable connection errors.
- SQS: Improved performance by reading messages in bulk.

Contributed by Matt Wise.

- Connection Pool: Attempting to acquire from a closed pool will now raise `RuntimeError`.

5.61 3.0.7

release-date 2013-12-02 04:00 P.M UTC

release-by Ask Solem

- Fixes Python 2.6 compatibility.
- Redis: Fixes 'bad file descriptor' issue.

5.62 3.0.6

release-date 2013-11-21 04:50 P.M UTC

release-by Ask Solem

- Timer: No longer attempts to hash keyword arguments (Issue #275).
- Async: Did not account for the long type for file descriptors.

Fix contributed by Fabrice Rabaute.

- PyPy: kqueue support was broken.
- Redis: Bad pub/sub payloads no longer crashes the consumer.
- Redis: Unix socket URLs can now specify a virtual host by including it as a query parameter.

Example URL specifying a virtual host using database number 3:

```
redis+socket:///tmp/redis.sock?virtual_host=3
```

- `kombu.VERSION` is now a named tuple.

5.63 3.0.5

release-date 2013-11-15 11:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.3.3.
- Redis: Fixed Python 3 compatibility problem (Issue #270).
- MongoDB: Fixed problem with URL parsing when authentication used.

Fix contributed by dongweiming.

- pyamqp: Fixed small issue when publishing the message and the property dictionary was set to None.

Fix contributed by Victor Garcia.

- Fixed problem in `repr(LaxBoundedSemaphore)`.

Fix contributed by Antoine Legrand.

- Tests now passing on Python 3.3.

5.64 3.0.4

release-date 2013-11-08 01:00 P.M UTC

release-by Ask Solem

- common.QoS: `decrement_eventually` now makes sure the value does not go below 1 if a prefetch count is enabled.

5.65 3.0.3

release-date 2013-11-04 03:00 P.M UTC

release-by Ask Solem

- SQS: Properly reverted patch that caused delays between messages.

Contributed by James Saryerwinnie

- select: Clear all registered fds on `poller.close`
- Eventloop: unregister if `EBADF` raised.

5.66 3.0.2

release-date 2013-10-29 02:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` version 1.3.2.
- select: Fixed problem where `unregister` did not properly remove the fd.

5.67 3.0.1

release-date 2013-10-24 04:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` version 1.3.1.
- Redis: New option `fanout_keyprefix`

This transport option is recommended for all users as it ensures that broadcast (fanout) messages sent is only seen by the current virtual host:

```
Connection('redis://', transport_options={'fanout_keyprefix': True})
```

However, enabling this means that you cannot send or receive messages from older Kombu versions so make sure all of your participants are upgraded and have the transport option enabled.

This will be the default behavior in Kombu 4.0.

- Distribution: Removed file `requirements/py25.txt`.
- MongoDB: Now disables `auto_start_request`.
- MongoDB: Enables `use_greenlets` if `eventlet/gevent` used.
- Pidbox: Fixes problem where expires header was `None`, which is a value not supported by the amq protocol.
- ConsumerMixin: New `consumer_context` method for starting the consumer without draining events.

5.68 3.0.0

release-date 2013-10-14 04:00 P.M BST

release-by Ask Solem

- Now depends on `amqp` version 1.3.
- No longer supports Python 2.5

The minimum Python version supported is now Python 2.6.0 for Python 2, and Python 3.3 for Python 3.

- Dual codebase supporting both Python 2 and 3.

No longer using `2to3`, making it easier to maintain support for both versions.

- `pickle`, `yaml` and `msgpack` deserialization is now disabled by default.

This means that Kombu will by default refuse to handle any content type other than `json`.

Pickle is known to be a security concern as it will happily load any object that is embedded in a pickle payload, and payloads can be crafted to do almost anything you want. The default serializer in Kombu is `json` but it also supports a number of other serialization formats that it will evaluate if received: including `pickle`.

It was always assumed that users were educated about the security implications of `pickle`, but in hindsight we don't think users should be expected to secure their services if we have the ability to be secure by default.

By disabling any content type that the user did not explicitly want enabled we ensure that the user must be conscious when they add `pickle` as a serialization format to support.

The other built-in serializers (`yaml` and `msgpack`) are also disabled even though they aren't considered insecure¹ at this point. Instead they're disabled so that if a security flaw is found in one of these libraries in the future, you will only be affected if you have explicitly enabled them.

To have your consumer accept formats other than `json` you have to explicitly add the wanted formats to a white-list of accepted content types:

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

¹ The PyYAML library has a `yaml.load()` function with some of the same security implications as `pickle`, but Kombu uses the `yaml.safe_load()` function which is not known to be affected.

or when using synchronous access:

```
>>> msg = queue.get(accept=['json', 'pickle', 'msgpack'])
```

The `accept` argument was first supported for consumers in version 2.5.10, and first supported by `Queue.get` in version 2.5.15 so to stay compatible with previous versions you can enable the previous behavior:

```
>>> from kombu import enable_insecure_serializers
>>> enable_insecure_serializers()
```

But note that this has global effect, so be very careful should you use it.

- `kombu.async`: Experimental event loop implementation.

This code was previously in Celery but was moved here to make it easier for async transport implementations.

The API is meant to match the Tulip API which will be included in Python 3.4 as the `asyncio` module. It's not a complete implementation obviously, but the goal is that it will be easy to change to it once that is possible.

- Utility function `kombu.common.ipublish` has been removed.

Use `Producer(..., retry=True)` instead.

- Utility function `kombu.common.isend_reply` has been removed

Use `send_reply(..., retry=True)` instead.

- `kombu.common.entry_to_queue` and `kombu.messaging.entry_to_queue` has been removed.

Use `Queue.from_dict(name, **options)` instead.

- Redis: Messages are now restored at the end of the list.

Contributed by Mark Lavin.

- **`StdConnectionError` and `StdChannelError` is removed** and `amqp.ConnectionError` and `amqp.ChannelError` is used instead.

- Message object implementation has moved to `kombu.message.Message`.

- Seralization: Renamed functions encode/decode to `dumps()` and `loads()`.

For backward compatibility the old names are still available as aliases.

- The `kombu.log.anon_logger` function has been removed.

Use `get_logger()` instead.

- `queue_declare` now returns namedtuple with `queue`, `message_count`, and `consumer_count` fields.

- `LamportClock`: Can now set lock class

- `kombu.utils.clock`: Utilities for ordering events added.

- `SimpleQueue` now allows you to override the exchange type used.

Contributed by Vince Gonzales.

- Zookeeper transport updated to support new changes in the kazoo library.

Contributed by Mahendra M.

- **`pyamqp/librabbitmq`: Transport options are now forwarded as keyword arguments** to the underlying connection (Issue #214).

- Transports may now distinguish between recoverable and irrecoverable connection and channel errors.
- `kombu.utils.Finalize` has been removed: Use `multiprocessing.util.Finalize` instead.
- Memory transport now supports the fanout exchange type.

Contributed by Davanum Srinivas.

- Experimental new [Pyro](#) transport (`kombu.transport.pyro`).

Contributed by Tommie McAfee.

- Experimental new [SoftLayer MQ](#) transport (`kombu.transport.SLMQ`).

Contributed by Kevin McDonald

- Eventio: Kqueue breaks in subtle ways so select is now used instead.
- SQLAlchemy transport: Can now specify table names using the `queue_tablename` and `message_tablename` transport options.

Contributed by Ryan Petrello.

Redis transport: Now supports using local UNIX sockets to communicate with the Redis server (Issue #1283)

To connect using a UNIX socket you have to use the `redis+socket` URL-prefix: `redis+socket://tmp/redis.sock`.

This functionality was merged from the [celery-redis-unixsocket](#) project. Contributed by Maxime Rouyrre.

ZeroMQ transport: `drain_events` now supports timeout.

Contributed by Jesper Thomschütz.

5.69 2.5.16

release-date 2013-10-04 03:30 P.M BST

release-by Ask Solem

- Python 3: Fixed problem with dependencies not being installed.

5.70 2.5.15

release-date 2013-10-04 03:30 P.M BST

release-by Ask Solem

- Declaration cache: Now only keeps hash of declaration so that it does not keep a reference to the channel.
- Declaration cache: Now respects `entity.can_cache_declaration` attribute.
- Fixes Python 2.5 compatibility.
- Fixes tests after python-msgpack changes.
- `Queue.get`: Now supports `accept` argument.

5.71 2.5.14

release-date 2013-08-23 05:00 P.M BST

release-by Ask Solem

- `safe_str` did not work properly resulting in `UnicodeDecodeError` (Issue #248).

5.72 2.5.13

release-date 2013-08-16 04:00 P.M BST

release-by Ask Solem

- Now depends on `amqp` 1.0.13
- Fixed typo in Django functional tests.
- `safe_str` now returns Unicode in Python 2.x
Fix contributed by Germán M. Bravo.
- `amqp`: Transport options are now merged with arguments supplied to the connection.
- Tests no longer depends on `distribute`, which was deprecated and merged back into `setuptools`.

Fix contributed by Sascha Peilicke.

- `ConsumerMixin` now also restarts on channel related errors.

Fix contributed by Corentin Ardeois.

5.73 2.5.12

release-date 2013-06-28 03:30 P.M BST

release-by Ask Solem

- Redis: Ignore errors about keys missing in the round-robin cycle.
- Fixed test suite errors on Python 3.
- Fixed `msgpack` test failures.

5.74 2.5.11

release-date 2013-06-25 02:30 P.M BST

release-by Ask Solem

- Now depends on `amqp` 1.0.12 (Py3 compatibility issues).
- MongoDB: Removed cause of a “database name in URI is being ignored” warning.

Fix by Flavio Percoco Premoli

- Adds `passive` option to `Exchange`.

Setting this flag means that the exchange will not be declared by kombu, but that it must exist already (or an exception will be raised).

Contributed by Rafal Malinowski

- `Connection.info()` now gives the current hostname and not the list of available hostnames.

Fix contributed by John Shuping.

- `pyamqp`: Transport options are now forwarded as kwargs to `amqp.Connection`.
- `librabbitmq`: Transport options are now forwarded as kwargs to `librabbitmq.Connection`.
- `librabbitmq`: Now raises `NotImplementedError` if SSL is enabled.

The `librabbitmq` library does not support ssl, but you can use stunnel or change to the `pyamqp://` transport instead.

Fix contributed by Dan LaMotte.

- `librabbitmq`: Fixed a cyclic reference at connection close.
- `eventio`: select implementation now removes bad file descriptors.
- `eventio`: Fixed Py3 compatibility problems.
- Functional tests added for `py-amqp` and `librabbitmq` transports.
- `Resource.force_close_all` no longer uses a mutex.
- `Pidbox`: Now ignores `InconsistencyError` when sending replies, as this error simply means that the client may no longer be alive.
- Adds new `Connection.collect` method, that can be used to clean up after connections without I/O.
- `queue_bind` is no longer called for queues bound to the “default exchange” (Issue #209).

Contributed by Jonathan Halcrow.

- The `max_retries` setting for retries was not respected correctly (off by one).

5.75 2.5.10

release-date 2013-04-11 06:10 P.M BST

release-by Ask Solem

5.75.1 Note about upcoming changes for Kombu 3.0

Kombu 3 consumers will no longer accept pickle/yaml or msgpack by default, and you will have to explicitly enable untrusted deserializers either globally using `kombu.enable_insecure_serializers()`, or using the `accept` argument to `Consumer`.

5.75.2 Changes

- New utility function to disable/enable untrusted serializers.
 - `kombu.disable_insecure_serializers()`
 - `kombu.enable_insecure_serializers()`.
- Consumer: `accept` can now be used to specify a whitelist of content types to accept.

If the `accept` whitelist is set and a message is received with a content type that is not in the whitelist then a `ContentDisallowed` exception is raised. Note that this error can be handled by the already existing `on_decode_error` callback

Examples:

```
Consumer(accept=['application/json'])
Consumer(accept=['pickle', 'json'])
```

- Now depends on `amqp 1.0.11`
- `pidbox`: `Mailbox` now supports the `accept` argument.
- `Redis`: More friendly error for when keys are missing.
- Connection URLs: The parser did not work well when there were multiple '+' tokens.

5.76 2.5.9

release-date 2013-04-08 05:07 P.M BST

release-by Ask Solem

- `Pidbox`: Now warns if there are multiple nodes consuming from the same `pidbox`.
- Adds `Queue.on_declared`

A callback to be called when the queue is declared, with signature `(name, messages, consumers)`.
- Now uses fuzzy matching to suggest alternatives to typos in transport names.
- `SQS`: Adds new transport option `queue_prefix`.

Contributed by `j0hnsmith`.
- `pyamqp`: No longer overrides `verify_connection`.
- `SQS`: Now specifies the `driver_type` and `driver_name` attributes.

Fix contributed by `Mher Movsisyan`.
- Fixed bug with `kombu.utils.retry_over_time` when no `errback` specified.

5.77 2.5.8

release-date 2013-03-21 04:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.0.10 which fixes a Python 3 compatibility error.
- Redis: Fixed a possible race condition (Issue #171).
- Redis: Ack emulation/`visibility_timeout` can now be disabled using a transport option.

Ack emulation adds quite a lot of overhead to ensure data is safe even in the event of an unclean shutdown. If data loss do not worry you there is now an `ack_emulation` transport option you can use to disable it:

```
Connection('redis://', transport_options={'ack_emulation': False})
```

- SQS: Fixed `boto` v2.7 compatibility (Issue #207).
- Exchange: Should not try to re-declare default exchange ("") (Issue #209).
- SQS: Long polling is now disabled by default as it was not implemented correctly, resulting in long delays between receiving messages (Issue #202).
- Fixed Python 2.6 incompatibility depending on `exc.errno` being available.

Fix contributed by Ephemera.

5.78 2.5.7

release-date 2013-03-08 01:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.0.9
- Redis: A regression in 2.5.6 caused the redis transport to ignore options set in `transport_options`.
- Redis: New `socket_timeout` transport option.
- Redis: `InconsistencyError` is now regarded as a recoverable error.
- Resource pools: Will no longer attempt to release resource that was never acquired.
- MongoDB: Now supports the `ssl` option.

Contributed by Sebastian Pawlus.

5.79 2.5.6

release-date 2013-02-08 01:00 P.M UTC

release-by Ask Solem

- Now depends on `amqp` 1.0.8 which works around a bug found on some Python 2.5 installations where `2**32` overflows to 0.

5.80 2.5.5

release-date 2013-02-07 05:00 P.M UTC

release-by Ask Solem

SQS: Now supports long polling (Issue #176).

The polling interval default has been changed to 0 and a new transport option (`wait_time_seconds`) has been added. This parameter specifies how long to wait for a message from SQS, and defaults to 20 seconds, which is the maximum value currently allowed by Amazon SQS.

Contributed by James Saryerwinnie.

- SQS: Now removes unpickleable fields before restoring messages.
- Consumer.__exit__ now ignores exceptions occurring while canceling the consumer.
- Virtual: Routing keys can now consist of characters also used in regular expressions (e.g. parens) (Issue #194).
- Virtual: Fixed compression header when restoring messages.

Fix contributed by Alex Koshelev.

- Virtual: ack/reject/requeue now works while using `basic_get`.
- Virtual: Message.reject is now supported by virtual transports (requeue depends on individual transport support).
- Fixed typo in hack used for static analyzers.

Fix contributed by Basil Mironenko.

5.81 2.5.4

release-date 2012-12-10 12:35 P.M UTC

release-by Ask Solem

- Fixed problem with connection clone and multiple URLs (Issue #182).

Fix contributed by Dane Guempel.

- zeromq: Now compatible with libzmq 3.2.x.

Fix contributed by Andrey Antukh.

- Fixed Python 3 installation problem (Issue #187).

5.82 2.5.3

release-date 2012-11-29 12:35 P.M UTC

release-by Ask Solem

- Pidbox: Fixed compatibility with Python 2.6

5.83 2.5.2

release-date 2012-11-29 12:35 P.M UTC

release-by Ask Solem

5.84 2.5.2

release-date 2012-11-29 12:35 P.M UTC

release-by Ask Solem

- [Redis] Fixed connection leak and added a new 'max_connections' transport option.

5.85 2.5.1

release-date 2012-11-28 12:45 P.M UTC

release-by Ask Solem

- Fixed bug where return value of `Queue.as_dict` could not be serialized with JSON (Issue #177).

5.86 2.5.0

release-date 2012-11-27 04:00 P.M UTC

release-by Ask Solem

- `py-amqp` is now the new default transport, replacing `amqplib`.

The new `py-amqp` library is a fork of `amqplib` started with the following goals:

- Uses AMQP 0.9.1 instead of 0.8
- Support for heartbeats (Issue #79 + Issue #131)
- Automatically revives channels on channel errors.
- **Support for all RabbitMQ extensions**
 - * Consumer Cancel Notifications (Issue #131)
 - * Publisher Confirms (Issue #131).
 - * Exchange-to-exchange bindings: `exchange_bind` / `exchange_unbind`.
- API compatible with `librabbitmq` so that it can be used as a pure-python replacement in environments where `rabbitmq-c` cannot be compiled. `librabbitmq` will be updated to support all the same features as `py-amqp`.
- Support for using multiple connection URL's for failover.

The first argument to `Connection` can now be a list of connection URLs:

```
Connection(['amqp://foo', 'amqp://bar'])
```

or it can be a single string argument with several URLs separated by semicolon:

```
Connection('amqp://foo;amqp://bar')
```

There is also a new keyword argument `failover_strategy` that defines how `ensure_connection()`/`ensure()/kombu.Connection.autoretry()` will reconnect in the event of connection failures.

The default reconnection strategy is `round-robin`, which will simply cycle through the list forever, and there's also a `shuffle` strategy that will select random hosts from the list. Custom strategies can also be used, in that case the argument must be a generator yielding the URL to connect to.

Example:

```
Connection('amqp://foo;amqp://bar')
```

- Now supports PyDev, PyCharm, pylint and other static code analysis tools.
- `Queue` now supports multiple bindings.

You can now have multiple bindings in the same queue by having the second argument be a list:

```
from kombu import binding, Queue

Queue('name', [
    binding(Exchange('E1'), routing_key='foo'),
    binding(Exchange('E1'), routing_key='bar'),
    binding(Exchange('E2'), routing_key='baz'),
])
```

To enable this, helper methods have been added:

- `bind_to()`
- `unbind_from()`

Contributed by Romyana Neykova.

- Custom serializers can now be registered using Setuptools entry-points.
See *Creating extensions using Setuptools entry-points*.
- New `kombu.common.QoS` class used as a thread-safe way to manage changes to a consumer or channels `prefetch_count`.

This was previously an internal class used in Celery now moved to the `kombu.common` module.

- Consumer now supports a `on_message` callback that can be used to process raw messages (not decoded).
Other callbacks specified using the `callbacks` argument, and the `receive` method will be not be called when a `on_message` callback is present.
- New utility `kombu.common.ignore_errors()` ignores connection and channel errors.

Must only be used for cleanup actions at shutdown or on connection loss.

- Support for exchange-to-exchange bindings.

The `Exchange` entity gained `bind_to` and `unbind_from` methods:

```
e1 = Exchange('A')(connection)
e2 = Exchange('B')(connection)
```

(continues on next page)

(continued from previous page)

```
e2.bind_to(e1, routing_key='rkey', arguments=None)
e2.unbind_from(e1, routing_key='rkey', arguments=None)
```

This is currently only supported by the pyamqp transport.

Contributed by Romyana Neykova.

5.87 2.4.10

release-date 2012-11-22 06:00 P.M UTC

release-by Ask Solem

- The previous versions connection pool changes broke Redis support so that it would always connect to localhost (default setting) no matter what connection parameters were provided (Issue #176).

5.88 2.4.9

release-date 2012-11-21 03:00 P.M UTC

release-by Ask Solem

- Redis: Fixed race condition that could occur while trying to restore messages (Issue #171).

Fix contributed by Ollie Walsh.

- Redis: Each channel is now using a specific connection pool instance, which is disconnected on connection failure.
- ProducerPool: Fixed possible dead-lock in the acquire method.
- ProducerPool: `force_close_all` no longer tries to call the non-existent `Producer._close`.
- librabbitmq: Now implements `transport.verify_connection` so that connection pools will not give back connections that are no longer working.
- New and better `repr()` for Queue and Exchange objects.
- Python 3: Fixed problem with running the unit test suite.
- Python 3: Fixed problem with JSON codec.

5.89 2.4.8

release-date 2012-11-02 05:00 P.M UTC

release-by Ask Solem

- Redis: Improved fair queue cycle implementation (Issue #166).

Contributed by Kevin McCarthy.

- Redis: Unacked message restore limit is now unlimited by default.

Also, the limit can now be configured using the `unacked_restore_limit` transport option:

```
Connection('redis://', transport_options={
    'unacked_restore_limit': 100,
})
```

A limit of 100 means that the consumer will restore at most 100 messages at each pass.

- Redis: Now uses a mutex to ensure only one consumer restores messages at a time.
The mutex expires after 5 minutes by default, but can be configured using the `unacked_mutex_expire` transport option.
- `LamportClock.adjust` now returns the new clock value.
- Heartbeats can now be specified in URLs.
Fix contributed by Mher Movsisyan.
- Kombu can now be used with PyDev, PyCharm and other static analysis tools.
- Fixes problem with msgpack on Python 3 (Issue #162).
Fix contributed by Jasper Bryant-Greene
- `amqpplib`: Fixed bug with timeouts when SSL is used in non-blocking mode.
Fix contributed by Mher Movsisyan

5.90 2.4.7

release-date 2012-09-18 03:00 P.M BST

release-by Ask Solem

- Virtual: Unknown exchanges now default to 'direct' when sending a message.
- MongoDB: Fixed memory leak when merging keys stored in the db (Issue #159)
Fix contributed by Michael Korbakov.
- MongoDB: Better index for MongoDB transport (Issue #158).

This improvement will create a new compound index for `queue` and `_id` in order to be able to use both indexed fields for getting a new message (using `queue` field) and sorting by `_id`. It'll be necessary to manually delete the old index from the collection.

Improvement contributed by rmihael

5.91 2.4.6

release-date 2012-09-12 03:00 P.M BST

release-by Ask Solem

- Adds additional compatibility dependencies:
 - Python <= 2.6:
 - * `importlib`

- * ordereddict
- Python <= 2.5
- * simplejson

5.92 2.4.5

release-date 2012-08-30 03:36 P.M BST

release-by Ask Solem

- Last version broke installtion on PyPy and Jython due to test requirements clean-up.

5.93 2.4.4

release-date 2012-08-29 04:00 P.M BST

release-by Ask Solem

- amqplib: Fixed a bug with asynchronously reading large messages.
- pyamqp: Now requires amqp 0.9.3
- Cleaned up test requirements.

5.94 2.4.3

release-date 2012-08-25 10:30 P.M BST

release-by Ask Solem

- Fixed problem with amqp transport alias (Issue #154).

5.95 2.4.2

release-date 2012-08-24 05:00 P.M BST

release-by Ask Solem

- Having an empty transport name broke in 2.4.1.

5.96 2.4.1

release-date 2012-08-24 04:00 P.M BST

release-by Ask Solem

- Redis: Fixed race condition that could cause the consumer to crash (Issue #151)
Often leading to the error message "could not convert string to float"
- Connection retry could cause an inifite loop (Issue #145).

- The `amqp` alias is now resolved at runtime, so that eventlet detection works even if patching was done later.

5.97 2.4.0

release-date 2012-08-17 08:00 P.M BST

release-by Ask Solem

- New experimental ZeroMQ `<kombu.transport.zmq` transport.
Contributed by John Watson.
- Redis: Ack timed-out messages were not restored when using the eventloop.
- Now uses pickle protocol 2 by default to be cross-compatible with Python 3.
The protocol can also now be changed using the `PICKLE_PROTOCOL` environment variable.
- Adds `Transport.supports_ev` attribute.
- Pika: Queue purge was not working properly.
Fix contributed by Steeve Morin.
- Pika backend was no longer working since Kombu 2.3
Fix contributed by Steeve Morin.

5.98 2.3.2

release-date 2012-08-01 06:00 P.M BST

release-by Ask Solem

- Fixes problem with deserialization in Python 3.

5.99 2.3.1

release-date 2012-08-01 04:00 P.M BST

release-by Ask Solem

- `librabbitmq`: Can now handle messages that does not have a `content_encoding/content_type` set (Issue #149).
Fix contributed by C Anthony Risinger.
- Beanstalk: Now uses localhost by default if the URL does not contain a host.

5.100 2.3.0

release-date 2012-07-24 03:50 P.M BST

release-by Ask Solem

- New `pyamqp://` transport!

The new `py-amqp` library is a fork of `amqplib` started with the following goals:

- Uses AMQP 0.9.1 instead of 0.8
- Should support all RabbitMQ extensions
- API compatible with `librabbitmq` so that it can be used as a pure-python replacement in environments where `rabbitmq-c` cannot be compiled.

If you start using `py-amqp` instead of `amqplib` you can enjoy many advantages including:

- Heartbeat support (Issue #79 + Issue #131)
- Consumer Cancel Notifications (Issue #131)
- Publisher Confirms

`amqplib` has not been updated in a long while, so maintaining our own fork ensures that we can quickly roll out new features and fixes without resorting to monkey patching.

To use the `py-amqp` transport you must install the `amqp` library:

```
$ pip install amqp
```

and change the connection URL to use the correct transport:

```
>>> conn = Connection('pyamqp://guest:guest@localhost//')
```

The `pyamqp://` transport will be the default fallback transport in Kombu version 3.0, when `librabbitmq` is not installed, and `librabbitmq` will also be updated to support the same features.

- Connection now supports `heartbeat` argument.

If enabled you must make sure to manually maintain heartbeats by calling the `Connection.heartbeat_check` at twice the rate of the specified heartbeat interval.

E.g. if you have `Connection(heartbeat=10)`, then you must call `Connection.heartbeat_check()` every 5 seconds.

if the server has not sent heartbeats at a suitable rate then the heartbeat check method must raise an error that is listed in `Connection.connection_errors`.

The attribute `Connection.supports_heartbeats` has been added for the ability to inspect if a transport supports heartbeats or not.

Calling `heartbeat_check` on a transport that does not support heartbeats results in a noop operation.

- SQS: Fixed bug with invalid characters in queue names.

Fix contributed by Zach Smith.

- `utils.reprcall`: Fixed typo where `kwargs` argument was an empty tuple by default, and not an empty dict.

5.101 2.2.6

release-date 2012-07-10 05:00 P.M BST

release-by Ask Solem

- Adds `kombu.messaging.entry_to_queue` for compat with previous versions.

5.102 2.2.5

release-date 2012-07-10 05:00 P.M BST

release-by Ask Solem

- Pidbox: Now sets queue expire at 10 seconds for reply queues.
- EventIO: Now ignores `ValueError` raised by `epoll.unregister`.
- MongoDB: Fixes Issue #142

Fix by Flavio Percoco Premoli

5.103 2.2.4

release-date 2012-07-05 04:00 P.M BST

release-by Ask Solem

- Support for `msgpack-python` 0.2.0 (Issue #143)
The latest `msgpack` version no longer supports Python 2.5, so if you're still using that you need to depend on an earlier `msgpack-python` version.
Fix contributed by Sebastian Insua
- `maybe_declare()` no longer caches entities with the `auto_delete` flag set.
- New experimental filesystem transport.
Contributed by Bobby Beever.
- Virtual Transports: Now support anonymous queues and exchanges.

5.104 2.2.3

release-date 2012-06-24 05:00 P.M BST

release-by Ask Solem

- `BrokerConnection` now renamed to `Connection`.
The name `Connection` has been an alias for a very long time, but now the rename is official in the documentation as well.
The `Connection` alias has been available since version 1.1.3, and `BrokerConnection` will still work and is not deprecated.
- `Connection.clone()` now works for the sqlalchemy transport.

- `kombu.common.eventloop()`, `kombu.utils.uuid()`, and `kombu.utils.url.parse_url()` can now be imported from the `kombu` module directly.
- Pidbox transport callback `after_reply_message_received` now happens in a finally block.
- Trying to use the `librabbitmq://` transport will now show the right name in the `ImportError` if `librabbitmq` is not installed.

The `librabbitmq` falls back to the older `pylibrabbitmq` name for compatibility reasons and would therefore show `No module named pylibrabbitmq` instead of `librabbitmq`.

5.105 2.2.2

release-date 2012-06-22 02:30 P.M BST

release-by Ask Solem

- Now depends on `anyjson` 0.3.3
- Json serializer: Now passes `buffer` objects directly, since this is supported in the latest `anyjson` version.
- Fixes blocking `epoll` call if timeout was set to 0.

Fix contributed by John Watson.

- `setup.py` now takes requirements from the `requirements/` directory.
- The distribution directory `contrib/` is now renamed to `extra/`

5.106 2.2.1

release-date 2012-06-21 01:00 P.M BST

release-by Ask Solem

- SQS: Default visibility timeout is now 30 minutes.

Since we have ack emulation the visibility timeout is only in effect if the consumer is abruptly terminated.

- `retry` argument to `Producer.publish` now works properly, when the `declare` argument is specified.
- Json serializer: didn't handle `buffer` objects (Issue #135).

Fix contributed by Jens Hoffrichter.

- Virtual: Now supports `passive` argument to `exchange_declare`.
- Exchange & Queue can now be bound to connections (which will use the default channel):

```
>>> exchange = Exchange('name')
>>> bound_exchange = exchange(connection)
>>> bound_exchange.declare()
```

- `SimpleQueue` & `SimpleBuffer` can now be bound to connections (which will use the default channel).
- `Connection.manager.get_bindings` now works for `librabbitmq` and `pika`.
- Adds new transport info attributes:
 - `Transport.driver_type`

Type of underlying driver, e.g. “amqp”, “redis”, “sql”.

– `Transport.driver_name`

Name of library used e.g. “amqpplib”, “redis”, “pymongo”.

– `Transport.driver_version()`

Version of underlying library.

5.107 2.2.0

release-date 2012-06-07 03:10 P.M BST

release-by Ask Solem

5.107.1 Important Notes

- The canonical source code repository has been moved to

<http://github.com/celery/kombu>

- Pidbox: Exchanges used by pidbox are no longer `auto_delete`.

Auto delete has been described as a misfeature, and therefore we have disabled it.

For RabbitMQ users old exchanges used by pidbox must be removed, these are named `mailbox_name.pidbox`, and `reply.mailbox_name.pidbox`.

The following command can be used to clean up these exchanges:

```
$ VHOST=/ URL=amqp:// python -c 'import sys,kombu;[kombu.Connection(
    sys.argv[-1]).channel().exchange_delete(x)
    for x in sys.argv[1:-1]]' \
$(sudo rabbitmqctl -q list_exchanges -p "$VHOST" \
| grep \.pidbox | awk '{print $1}') "$URL"
```

The `VHOST` variable must be set to the target RabbitMQ virtual host, and the `URL` must be the AMQP URL to the server.

- The `amqp` transport alias will now use `librabbitmq` if installed.

[py-librabbitmq](#) is a fast AMQP client for Python using the `librabbitmq` C library.

It can be installed by:

```
$ pip install librabbitmq
```

It will not be used if the process is monkey patched by `eventlet/gevent`.

5.107.2 News

- Redis: Ack emulation improvements.

Reducing the possibility of data loss.

Acks are now implemented by storing a copy of the message when the message is consumed. The copy is not removed until the consumer acknowledges or rejects it.

This means that unacknowledged messages will be redelivered either when the connection is closed, or when the visibility timeout is exceeded.

– Visibility timeout

This is a timeout for acks, so that if the consumer does not ack the message within this time limit, the message is redelivered to another consumer.

The timeout is set to one hour by default, but can be changed by configuring a transport option:

```
>>> Connection('redis://', transport_options={
...     'visibility_timeout': 1800, # 30 minutes
... })
```

NOTE: Messages that have not been acked will be redelivered if the visibility timeout is exceeded, for Celery users this means that ETA/countdown tasks that are scheduled to execute with a time that exceeds the visibility timeout will be executed twice (or more). If you plan on using long ETA/countdowns you should tweak the visibility timeout accordingly:

```
BROKER_TRANSPORT_OPTIONS = {'visibility_timeout': 18000} # 5 hours
```

Setting a long timeout means that it will take a long time for messages to be redelivered in the event of a power failure, but if so happens you could temporarily set the visibility timeout lower to flush out messages when you start up the systems again.

- Experimental [Apache ZooKeeper](#) transport

More information is in the module reference: [kombu.transport.zookeeper](#).

Contributed by Mahendra M.

- Redis: Priority support.

The message's `priority` field is now respected by the Redis transport by having multiple lists for each named queue. The queues are then consumed by in order of priority.

The priority field is a number in the range of 0 - 9, where 0 is the default and highest priority.

The priority range is collapsed into four steps by default, since it is unlikely that nine steps will yield more benefit than using four steps. The number of steps can be configured by setting the `priority_steps` transport option, which must be a list of numbers in **sorted order**:

```
>>> x = Connection('redis://', transport_options={
...     'priority_steps': [0, 2, 4, 6, 8, 9],
... })
```

Priorities implemented in this way is not as reliable as priorities on the server side, which is why nickname the feature “quasi-priorities”; **Using routing is still the suggested way of ensuring quality of service**, as client implemented priorities fall short in a number of ways, e.g. if the worker is busy with long running tasks, has prefetched many messages, or the queues are congested.

Still, it is possible that using priorities in combination with routing can be more beneficial than using routing or priorities alone. Experimentation and monitoring should be used to prove this.

Contributed by Germán M. Bravo.

- Redis: Now cycles queues so that consuming is fair.

This ensures that a very busy queue won't block messages from other queues, and ensures that all queues have an equal chance of being consumed from.

This used to be the case before, but the behavior was accidentally changed while switching to using blocking pop.
- Redis: Auto delete queues that are bound to fanout exchanges is now deleted at `channel.close`.
- `amqpplib`: Refactored the `drain_events` implementation.
- `Pidbox`: Now uses `connection.default_channel`.
- Pickle serialization: Can now decode buffer objects.
- Exchange/Queue declarations can now be cached even if the entity is non-durable.

This is possible because the list of cached declarations are now kept with the connection, so that the entities will be redeclared if the connection is lost.
- Kombu source code now only uses one-level of explicit relative imports.

5.107.3 Fixes

- `eventio`: Now ignores `ENOENT` raised by `epoll.register`, and `EEXIST` from `epoll.unregister`.
- `eventio`: `kqueue` now ignores `KeyError` on `unregister`.
- Redis: `Message.reject` now supports the `requeue` argument.
- Redis: Remove superfluous pipeline call.

Fix contributed by Thomas Johansson.

- Redis: Now sets redelivered header for redelivered messages.
- Now always makes sure references to `sys.exc_info()` is removed.
- Virtual: The compression header is now removed before restoring messages.
- More tests for the SQLAlchemy backend.

Contributed by Franck Cuny.

- Url parsing did not handle MongoDB URLs properly.

Fix contributed by Flavio Percoco Premoli.

- Beanstalk: Ignore default tube when reserving.

Fix contributed by Zhao Xiaohong.

5.107.4 Nonblocking consume support

librabbitmq, amqpplib and redis transports can now be used non-blocking.

The interface is very manual, and only consuming messages is non-blocking so far.

The API should not be regarded as stable or final in any way. It is used by Celery which has very limited needs at this point. Hopefully we can introduce a proper callback-based API later.

- `Transport.eventmap`
Is a map of `fd -> callback(fileno, event)` to register in an eventloop.
- `Transport.on_poll_start()`
Is called before every call to poll. The poller must support `register(fd, callback)` and `unregister(fd)` methods.
- `Transport.on_poll_start(poller)`
Called when the hub is initialized. The poller argument must support the same interface as `kombu.utils.eventio.poll`.
- `Connection.ensure_connection` now takes a callback argument which is called for every loop while the connection is down.
- Adds `connection.drain_nowait`
This is a non-blocking alternative to `drain_events`, but only supported by amqpplib/librabbitmq.
- `drain_events` now sets `connection.more_to_read` if there is more data to read.
This is to support eventloops where other things must be handled between draining events.

5.108 2.1.8

release-date 2012-05-06 03:06 P.M BST

release-by Ask Solem

- Bound Exchange/Queue's are now pickleable.
- Consumer/Producer can now be instantiated without a channel, and only later bound using `.revive(channel)`.
- `ProducerPool` now takes `Producer` argument.
- `fxrange()` now counts forever if the stop argument is set to `None`. (`fxrange` is like `xrange` but for decimals).
- Auto delete support for virtual transports were incomplete and could lead to problems so it was removed.
- Cached declarations (`maybe_declare()`) are now bound to the underlying connection, so that entities are redeclared if the connection is lost.

This also means that previously uncacheable entities (e.g. non-durable) can now be cached.

- `compat ConsumerSet`: can now specify channel.

5.109 2.1.7

release-date 2012-04-27 06:00 P.M BST

release-by Ask Solem

- compat consumerset now accepts optional channel argument.

5.110 2.1.6

release-date 2012-04-23 01:30 P.M BST

release-by Ask Solem

- SQLAlchemy transport was not working correctly after URL parser change.
- maybe_declare now stores cached declarations per underlying connection instead of globally, in the rare case that data disappears from the broker after connection loss.
- Django: Added South migrations.

Contributed by Joseph Crosland.

5.111 2.1.5

release-date 2012-04-13 03:30 P.M BST

release-by Ask Solem

- The url parser removed more than the first leading slash (Issue #121).
- SQLAlchemy: Can now specify url using + separator

Example:

```
Connection('sqla+mysql://localhost/db')
```

- Better support for anonymous queues (Issue #116).
- Connection.as_uri now quotes url parts (Issue #117).
- Beanstalk: Can now set message TTR as a message property.

Contributed by Andrii Kostenko

5.112 2.1.4

release-date 2012-04-03 04:00 P.M GMT

release-by Ask Solem

- MongoDB: URL parsing are now delegated to the pymongo library (Fixes Issue #103 and Issue #87).
Fix contributed by Flavio Percoco Premoli and James Sullivan
- SQS: A bug caused SimpleDB to be used even if sdb persistence was not enabled (Issue #108).

Fix contributed by Anand Kumria.

- Django: Transaction was committed in the wrong place, causing data cleanup to fail (Issue #115).

Fix contributed by Daisuke Fujiwara.

- MongoDB: Now supports replica set URLs.

Contributed by Flavio Percoco Premoli.

- Redis: Now raises a channel error if a queue key that is currently being consumed from disappears.

Fix contributed by Stephan Jaekel.

- All transport 'channel_errors' lists now includes `kombu.exception.StdChannelError`.
- All kombu exceptions now inherit from a common `KombuError`.

5.113 2.1.3

release-date 2012-03-20 03:00 P.M GMT

release-by Ask Solem

- Fixes Jython compatibility issues.
- Fixes Python 2.5 compatibility issues.

5.114 2.1.2

release-date 2012-03-01 01:00 P.M GMT

release-by Ask Solem

- amqpplib: Last version broke SSL support.

5.115 2.1.1

release-date 2012-02-24 02:00 P.M GMT

release-by Ask Solem

- Connection URLs now supports encoded characters.
- Fixed a case where connection pool could not recover from connection loss.

Fix contributed by Florian Munz.

- We now patch amqpplib's `__del__` method to skip trying to close the socket if it is not connected, as this resulted in an annoying warning.
- Compression can now be used with binary message payloads.

Fix contributed by Steeve Morin.

5.116 2.1.0

release-date 2012-02-04 10:38 P.M GMT

release-by Ask Solem

- MongoDB: Now supports fanout (broadcast) (Issue #98).

Contributed by Scott Lyons.

- amqpplib: Now detects broken connections by using MSG_PEEK.
- pylibrabbitmq: Now supports `basic_get` (Issue #97).
- gevent: Now always uses the `select` polling backend.
- pika transport: Now works with pika 0.9.5 and 0.9.6dev.

The old pika transport (supporting 0.5.x) is now available as alias `oldpika`.

(Note terribly latency has been experienced with the new pika versions, so this is still an experimental transport).

- Virtual transports: can now set polling interval via the transport options (Issue #96).

Example:

```
>>> Connection('sqs://', transport_options={
...     'polling_interval': 5.0})
```

The default interval is transport specific, but usually 1.0s (or 5.0s for the Django database transport, which can also be set using the `KOMBU_POLLING_INTERVAL` setting).

- Adds convenience function: `kombu.common.eventloop()`.

5.117 2.0.0

release-date 2012-01-15 06:34 P.M GMT

release-by Ask Solem

5.117.1 Important Notes

Python Compatibility

- No longer supports Python 2.4.

Users of Python 2.4 can still use the 1.x series.

The 1.x series has entered bugfix-only maintenance mode, and will stay that way as long as there is demand, and a willingness to maintain it.

New Transports

- `django-kombu` is now part of Kombu core.

The Django message transport uses the Django ORM to store messages.

It uses polling, with a default polling interval of 5 seconds. The polling interval can be increased or decreased by configuring the `KOMBU_POLLING_INTERVAL` Django setting, which is the polling interval in seconds as an int or a float. Note that shorter polling intervals can cause extreme strain on the database: if responsiveness is needed you shall consider switching to a non-polling transport.

To use it you must use transport alias "`django`", or as a URL:

```
django://
```

and then add `kombu.transport.django` to `INSTALLED_APPS`, and run `manage.py syncdb` to create the necessary database tables.

Upgrading

If you have previously used `django-kombu`, then the entry in `INSTALLED_APPS` must be changed from `dkombu` to `kombu.transport.django`:

```
INSTALLED_APPS = (
    # ...
    'kombu.transport.django',
)
```

If you have previously used `django-kombu`, then there is no need to recreate the tables, as the old tables will be fully compatible with the new version.

- `kombu-sqlalchemy` is now part of Kombu core.

This change requires no code changes given that the `sqlalchemy` transport alias is used.

5.117.2 News

- `kombu.mixins.ConsumerMixin` is a mixin class that lets you easily write consumer programs and threads. See [Examples](#) and [Consumers](#).
- SQS Transport: Added support for SQS queue prefixes (Issue #84).

The queue prefix can be set using the transport option `queue_name_prefix`:

```
BrokerTransport('SQS://', transport_options={
    'queue_name_prefix': 'myapp'})
```

Contributed by Nitzan Miron.

- `Producer.publish` now supports automatic retry.

Retry is enabled by the `reply` argument, and retry options set by the `retry_policy` argument:

```
exchange = Exchange('foo')
producer.publish(message, exchange=exchange, retry=True,
                 declare=[exchange], retry_policy={
                     'interval_start': 1.0})
```

See [ensure\(\)](#) for a list of supported retry policy options.

- `Producer.publish` now supports a `declare` keyword argument.

This is a list of entities (Exchange, or Queue) that should be declared before the message is published.

5.117.3 Fixes

- Redis transport: Timeout was multiplied by 1000 seconds when using `select` for event I/O (Issue #86).

5.118 1.5.1

release-date 2011-11-30 01:00 P.M GMT

release-by Ask Solem

- Fixes issue with `kombu.compat` introduced in 1.5.0 (Issue #83).
- Adds the ability to disable `content_types` in the serializer registry.

Any message with a content type that is disabled will be refused. One example would be to disable the Pickle serializer:

```
>>> from kombu.serialization import registry
# by name
>>> registry.disable('pickle')
# or by mime-type.
>>> registry.disable('application/x-python-serialize')
```

5.119 1.5.0

release-date 2011-11-27 06:00 P.M GMT

release-by Ask Solem

- `kombu.pools`: Fixed a bug resulting in resources not being properly released.

This was caused by the use of `__hash__` to distinguish them.

- Virtual transports: Dead-letter queue is now disabled by default.

The dead-letter queue was enabled by default to help application authors, but now that Kombu is stable it should be removed. There are after all many cases where messages should just be dropped when there are no queues to buffer them, and keeping them without supporting automatic cleanup is rather considered a resource leak than a feature.

If wanted the dead-letter queue can still be enabled, by using the `deadletter_queue` transport option:

```
>>> x = Connection('redis://',
...               transport_options={'deadletter_queue': 'ae.undeliver'})
```

In addition, an `UndeliverableWarning` is now emitted when the dead-letter queue is enabled and a message ends up there.

Contributed by Ionel Maries Cristian.

- MongoDB transport now supports Replicasets (Issue #81).
Contributed by Ivan Metzlar.
- The `Connection.ensure` methods now accepts a `max_retries` value of 0.
A value of 0 now means *do not retry*, which is distinct from `None` which means *retry indefinitely*.
Contributed by Dan McGee.
- SQS Transport: Now has a lowercase `sqs` alias, so that it can be used with broker URLs (Issue #82).
Fix contributed by Hong Minhee
- SQS Transport: Fixes `KeyError` on message acknowledgments (Issue #73).
The SQS transport now uses UUID's for delivery tags, rather than a counter.
Fix contributed by Brian Bernstein.
- SQS Transport: Unicode related fixes (Issue #82).
Fix contributed by Hong Minhee.
- Redis version check could crash because of improper handling of types (Issue #63).
- Fixed error with `Resource.force_close_all` when resources were not yet properly initialized (Issue #78).

5.120 1.4.3

release-date 2011-10-27 10:00 P.M BST

release-by Ask Solem

- Fixes bug in `ProducerPool` where too many resources would be acquired.

5.121 1.4.2

release-date 2011-10-26 05:00 P.M BST

release-by Ask Solem

- Eventio: Polling should ignore `errno.EINTR`
- SQS: `str.encode` did only start accepting kwargs after Py2.7.
- `simple_task_queue` example didn't run correctly (Issue #72).
Fix contributed by Stefan Eletzhofer.
- Empty messages would not raise an exception not able to be handled by `on_decode_error` (Issue #72)
Fix contributed by Christophe Chauvet.
- CouchDB: Properly authenticate if `user/password` set (Issue #70)
Fix contributed by Rafael Duran Castaneda
- `Connection.Consumer` had the wrong signature.
Fix contributed by Pavel Skvazh

5.122 1.4.1

release-date 2011-09-26 04:00 P.M BST

release-by Ask Solem

- 1.4.0 broke the producer pool, resulting in new connections being established for every acquire.

5.123 1.4.0

release-date 2011-09-22 05:00 P.M BST

release-by Ask Solem

- Adds module *kombu.mixins*.

This module contains a *ConsumerMixin* class that can be used to easily implement a message consumer thread that consumes messages from one or more *kombu.Consumer* instances.

- New example: *Task Queue Example*

Using the *ConsumerMixin*, default channels and the global connection pool to demonstrate new Kombu features.

- MongoDB transport did not work with MongoDB ≥ 2.0 (Issue #66)

Fix contributed by James Turk.

- Redis-py version check did not account for beta identifiers in version string.

Fix contributed by David Ziegler.

- Producer and Consumer now accepts a connection instance as the first argument.

The connections default channel will then be used.

In addition shortcut methods has been added to Connection:

```
>>> connection.Producer(exchange)
>>> connection.Consumer(queues=..., callbacks=...)
```

- Connection has acquired a `connected` attribute that can be used to check if the connection instance has established a connection.
- `ConnectionPool.acquire_channel` now returns the connections default channel rather than establishing a new channel that must be manually handled.
- Added `kombu.common.maybe_declare`
`maybe_declare(entity)` declares an entity if it has not previously been declared in the same process.
- `kombu.compat.entry_to_queue()` has been moved to *kombu.common*
- New module *kombu.clocks* now contains an implementation of Lamports logical clock.

5.124 1.3.5

release-date 2011-09-16 06:00 P.M BST

release-by Ask Solem

- Python 3: `AMQP_PROTOCOL_HEADER` must be bytes, not str.

5.125 1.3.4

release-date 2011-09-16 06:00 P.M BST

release-by Ask Solem

- Fixes syntax error in `pools.reset`

5.126 1.3.3

release-date 2011-09-15 02:00 P.M BST

release-by Ask Solem

- `pools.reset` did not support after forker arguments.

5.127 1.3.2

release-date 2011-09-10 01:00 P.M BST

release-by Mher Movsisyan

- Broke Python 2.5 compatibility by importing `parse_qs1` from `urlparse`
- `Connection.default_channel` is now closed when connection is revived after connection failures.
- Pika: Channel now supports the `connection.client` attribute as required by the simple interface.
- `pools.set_limit` now raises an exception if the limit is lower than the previous limit.
- `pools.set_limit` no longer resets the pools.

5.128 1.3.1

release-date 2011-10-07 03:00 P.M BST

release-by Ask Solem

- Last release broke after fork for pool reinitialization.
- Producer/Consumer now has a `connection` attribute, giving access to the `Connection` of the instance.
- Pika: Channels now have access to the underlying `Connection` instance using `channel.connection.client`.

This was previously required by the `Simple` classes and is now also required by `Consumer` and `Producer`.

- `Connection.default_channel` is now closed at object revival.

- Adds kombu.clocks.LamportClock.
- compat.entry_to_queue has been moved to new module *kombu.common*.

5.129 1.3.0

release-date 2011-10-05 01:00 P.M BST

release-by Ask Solem

- Broker connection info can be now be specified using URLs

The broker hostname can now be given as a URL instead, of the format:

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

for example the default broker is expressed as:

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

Transport defaults to amqp, and is not required. user, password, port and virtual_host is also not mandatory and will default to the corresponding transports default.

Note: Note that the path component (virtual_host) always starts with a forward-slash. This is necessary to distinguish between the virtual host '' (empty) and '/', which are both acceptable virtual host names.

A virtual host of '/' becomes:

```
.. code-block:: text
```

```
amqp://guest:guest@localhost:5672//
```

and a virtual host of '' (empty) becomes:

```
amqp://guest:guest@localhost:5672/
```

So the leading slash in the path component is **always required**.

- Now comes with default global connection and producer pools.

The acquire a connection using the connection parameters from a Connection:

```
>>> from kombu import Connection, connections
>>> connection = Connection('amqp://guest:guest@localhost//')
>>> with connections[connection].acquire(block=True):
...     # do something with connection
```

To acquire a producer using the connection parameters from a Connection:

```
>>> from kombu import Connection, producers
>>> connection = Connection('amqp://guest:guest@localhost//')
>>> with producers[connection].acquire(block=True):
...     producer.publish({'hello': 'world'}, exchange='hello')
```

Acquiring a producer will in turn also acquire a connection from the associated pool in connections, so you the number of producers is bound the same limit as number of connections.

The default limit of 100 connections per connection instance can be changed by doing:

```
>>> from kombu import pools
>>> pools.set_limit(10)
```

The pool can also be forcefully closed by doing:

```
>>> from kombu import pools
>>> pool.reset()
```

- SQS Transport: Persistence using SimpleDB is now disabled by default, after reports of unstable SimpleDB connections leading to errors.
- Producer can now be used as a context manager.
- Producer.__exit__ now properly calls release instead of close.

The previous behavior would lead to a memory leak when using the `kombu.pools.ProducerPool`

- Now silences all exceptions from `import ctypes` to match behaviour of the standard Python `uuid` module, and avoid passing on MemoryError exceptions on SELinux-enabled systems (Issue #52 + Issue #53)
- `amqp` is now an alias to the `amqplib` transport.
- `kombu.syn.detect_environment` now returns 'default', 'eventlet', or 'gevent' depending on what monkey patches have been installed.
- Serialization registry has new attribute `type_to_name` so it is possible to lookup serializater name by content type.
- Exchange argument to `Producer.publish` can now be an `Exchange` instance.
- `compat.Publisher` now supports the `channel` keyword argument.
- Acking a message on some transports could lead to `KeyError` being raised (Issue #57).
- Connection pool: Connections are no long instantiated when the pool is created, but instantiated as needed instead.
- Tests now pass on PyPy.
- `Connection.as_uri` now includes the password if the keyword argument `include_password` is set.
- Virtual transports now comes with a default `default_connection_params` attribute.

5.130 1.2.1

release-date 2011-07-29 12:52 P.M BST

release-by Ask Solem

- Now depends on `amqplib >= 1.0.0`.
- Redis: Now automatically deletes `auto_delete` queues at `basic_cancel`.
- `serialization.unregister` added so it is possible to remove unwanted seralizers.
- Fixes MemoryError while importing ctypes on SELinux (Issue #52).

- `Connection.autoretry` is a version of `ensure` that works with arbitrary functions (i.e. it does not need an associated object that implements the `revive` method).

Example usage:

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

- `ConnectionPool.acquire` no longer force establishes the connection.
The connection will be established as needed.
- `Connection.ensure` now supports an `on_revive` callback that is applied whenever the connection is re-established.
- `Consumer.consuming_from(queue)` returns `True` if the `Consumer` is consuming from queue.
- `Consumer.cancel_by_queue` did not remove the queue from queues.
- `compat.ConsumerSet.add_queue_from_dict` now automatically declared the queue if `auto_declare` set.

5.131 1.2.0

release-date 2011-07-15 12:00 P.M BST

release-by Ask Solem

- Virtual: Fixes cyclic reference in `Channel.close` (Issue #49).
- `Producer.publish`: Can now set additional properties using keyword arguments (Issue #48).
- Adds `Queue.no_ack` option to control the `no_ack` option for individual queues.
- Recent versions broke `pylibrabbitmq` support.
- `SimpleQueue` and `SimpleBuffer` can now be used as contexts.
- Test requirements specifies `PyYAML==3.09` as 3.10 dropped Python 2.4 support
- Now properly reports default values in `Connection.info/as_uri`

5.132 1.1.6

release-date 2011-06-13 04:00 P.M BST

release-by Ask Solem

- Redis: Fixes issue introduced in 1.1.4, where a redis connection failure could leave consumer hanging forever.
- SQS: Now supports fanout messaging by using `SimpleDB` to store routing tables.

This can be disabled by setting the `supports_fanout` transport option:

```
>>> Connection(transport='SQS',
...             transport_options={'supports_fanout': False})
```

- SQS: Now properly deletes a message when a message is acked.

- SQS: Can now set the Amazon AWS region, by using the `region` transport option.
- amqpplib: Now uses *localhost* as default hostname instead of raising an error.

5.133 1.1.5

release-date 2011-06-07 06:00 P.M BST

release-by Ask Solem

- Fixes compatibility with redis-py 2.4.4.

5.134 1.1.4

release-date 2011-06-07 04:00 P.M BST

release-by Ask Solem

- Redis transport: Now requires redis-py version 2.4.4 or later.
- New Amazon SQS transport added.

Usage:

```
>>> conn = Connection(transport='SQS',
...                   userid=aws_access_key_id,
...                   password=aws_secret_access_key)
```

The environment variables `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` are also supported.

- librabbitmq transport: Fixes default credentials support.
- amqpplib transport: Now supports *login_method* for SSL auth.

Connection now supports the *login_method* keyword argument.

Default *login_method* is `AMQPLAIN`.

5.135 1.1.3

release-date 2011-04-21 04:00 P.M CEST

release-by Ask Solem

- Redis: Consuming from multiple connections now works with Eventlet.
- Redis: Can now perform channel operations while the channel is in BRPOP/LISTEN mode (Issue #35).

Also the async BRPOP now times out after 1 second, this means that canceling consuming from a queue/starting consuming from additional queues has a latency of up to one second (BRPOP does not support subsecond timeouts).

- Virtual: Allow channel objects to be closed multiple times without error.
- amqpplib: `AttributeError` has been added to the list of known connection related errors (`Connection.connection_errors`).

- amqpplib: Now converts `SSLError` timeout errors to `socket.timeout` (<http://bugs.python.org/issue10272>)
- Ensures cyclic references are destroyed when the connection is closed.

5.136 1.1.2

release-date 2011-04-06 04:00 P.M CEST

release-by Ask Solem

- Redis: Fixes serious issue where messages could be lost.

The issue could happen if the message exceeded a certain number of kilobytes in size.

It is recommended that all users of the Redis transport should upgrade to this version, even if not currently experiencing any issues.

5.137 1.1.1

release-date 2011-04-05 03:51 P.M CEST

release-by Ask Solem

- 1.1.0 started using `Queue.LifoQueue` which is only available in Python 2.6+ (Issue #33). We now ship with our own `LifoQueue`.

5.138 1.1.0

release-date 2011-04-05 01:05 P.M CEST

release-by Ask Solem

5.138.1 Important Notes

- Virtual transports: Message body is now base64 encoded by default (Issue #27).

This should solve problems sending binary data with virtual transports.

Message compatibility is handled by adding a `body_encoding` property, so messages sent by older versions is compatible with this release. However – If you are accessing the messages directly not using Kombu, then you have to respect the `body_encoding` property.

If you need to disable base64 encoding then you can do so via the transport options:

```
Connection(transport='...',
            transport_options={'body_encoding': None})
```

For transport authors:

You don't have to change anything in your custom transports, as this is handled automatically by the base class.

If you want to use a different encoder you can do so by adding a key to `Channel.codecs`. Default encoding is specified by the `Channel.body_encoding` attribute.

A new codec must provide two methods: `encode(data)` and `decode(data)`.

- `ConnectionPool/ChannelPool/Resource`: Setting `limit=None` (or 0) now disables pool semantics, and will establish and close the resource whenever acquired or released.
- `ConnectionPool/ChannelPool/Resource`: Is now using a LIFO queue instead of the previous FIFO behavior.

This means that the last resource released will be the one acquired next. I.e. if only a single thread is using the pool this means only a single connection will ever be used.

- `Connection`: Cloned connections did not inherit `transport_options` (`__copy__`).
- `contrib/requirements` is now located in the top directory of the distribution.
- `MongoDB`: Now supports authentication using the `userid` and `password` arguments to `Connection` (Issue #30).
- `Connection`: Default authentication credentials are now delegated to the individual transports.

This means that the `userid` and `password` arguments to `Connection` is no longer *guest/guest* by default.

The `amqplib` and `pika` transports will still have the default credentials.

- `Consumer.__exit__()` did not have the correct signature (Issue #32).
- `Channel` objects now have a `channel_id` attribute.
- **MongoDB: Version sniffing broke with development versions of mongod** (Issue #29).
- **New environment variable `KOMBU_LOG_CONNECTION` will now emit debug** log messages for connection related actions.

`KOMBU_LOG_DEBUG` will also enable `KOMBU_LOG_CONNECTION`.

5.139 1.0.7

release-date 2011-03-28 05:45 P.M CEST

release-by Ask Solem

- Now depends on `anyjson` 0.3.1
 - `cjson` is no longer a recommended json implementation, and `anyjson` will now emit a deprecation warning if used.
- Please note that the `Pika` backend only works with version 0.5.2.
 - The latest version (0.9.x) drastically changed API, and it is not compatible yet.
- `on_decode_error` is now called for exceptions in `message_to_python` (Issue #24).
- `Redis`: did not respect QoS settings.
- `Redis`: Creating a connection now ensures the connection is established.

This means `Connection.ensure_connection` works properly with `Redis`.

- `consumer_tag` argument to `Queue.consume` can't be `None` (Issue #21).
 - A `None` value is now automatically converted to empty string. An empty string will make the server generate a unique tag.
- `Connection` now supports a `transport_options` argument.

This can be used to pass additional arguments to transports.

- Pika: `drain_events` raised `socket.timeout` even if no timeout set (Issue #8).

5.140 1.0.6

release-date 2011-03-22 04:00 P.M CET

release-by Ask Solem

- The `delivery_mode` aliases (persistent/transient) were not automatically converted to integer, and would cause a crash if using the `amqp`lib transport.
- Redis: The `redis-py` `InvalidData` exception suddenly changed name to `DataError`.
- The `KOMBU_LOG_DEBUG` environment variable can now be set to log all channel method calls.

Support for the following environment variables have been added:

- `KOMBU_LOG_CHANNEL` will wrap channels in an object that logs every method call.
- `KOMBU_LOG_DEBUG` both enables channel logging and configures the root logger to emit messages to standard error.

Example Usage:

```
$ KOMBU_LOG_DEBUG=1 python
>>> from kombu import Connection
>>> conn = Connection()
>>> channel = conn.channel()
Start from server, version: 8.0, properties:
    {u'product': 'RabbitMQ', ..... }
Open OK! known_hosts []
using channel_id: 1
Channel open
>>> channel.queue_declare('myq', passive=True)
[Kombu channel:1] queue_declare('myq', passive=True)
(u'myq', 0, 1)
```

5.141 1.0.5

release-date 2011-03-17 04:00 P.M CET

release-by Ask Solem

- Fixed memory leak when creating virtual channels. All virtual transports affected (redis, mongodb, memory, django, sqlalchemy, couchdb, beanstalk).
- Virtual Transports: Fixed potential race condition when acking messages.

If you have been affected by this, the error would show itself as an exception raised by the `OrderedDict` implementation. (object no longer exists).

- MongoDB transport requires the `findandmodify` command only available in MongoDB 1.3+, so now raises an exception if connected to an incompatible server version.
- Virtual Transports: `basic.cancel` should not try to remove unknown consumer tag.

5.142 1.0.4

release-date 2011-02-28 04:00 P.M CET

release-by Ask Solem

- Added `Transport.polling_interval`

Used by django-kombu to increase the time to sleep between SELECTs when there are no messages in the queue.

Users of django-kombu should upgrade to django-kombu v0.9.2.

5.143 1.0.3

release-date 2011-02-12 04:00 P.M CET

release-by Ask Solem

- `ConnectionPool`: Re-connect if amqp-lib connection closed
- Adds `Queue.as_dict` + `Exchange.as_dict`.
- Copyright headers updated to include 2011.

5.144 1.0.2

release-date 2011-01-31 10:45 P.M CET

release-by Ask Solem

- amqp-lib: Message properties were not set properly.
- Ghettoq backend names are now automatically translated to the new names.

5.145 1.0.1

release-date 2011-01-28 12:00 P.M CET

release-by Ask Solem

- Redis: Now works with Linux (epoll)

5.146 1.0.0

release-date 2011-01-27 12:00 P.M CET

release-by Ask Solem

- Initial release

5.147 0.1.0

release-date 2010-07-22 04:20 P.M CET

release-by Ask Solem

- Initial fork of carrot

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

k

- `kombu`, 39
- `kombu.abstract`, 92
- `kombu.asynchronous`, 205
- `kombu.asynchronous.aws`, 216
- `kombu.asynchronous.aws.connection`, 217
- `kombu.asynchronous.aws.sqs`, 219
- `kombu.asynchronous.aws.sqs.connection`, 219
- `kombu.asynchronous.aws.sqs.message`, 220
- `kombu.asynchronous.aws.sqs.queue`, 221
- `kombu.asynchronous.debug`, 209
- `kombu.asynchronous.http`, 210
- `kombu.asynchronous.http.base`, 213
- `kombu.asynchronous.http.curl`, 216
- `kombu.asynchronous.hub`, 206
- `kombu.asynchronous.semaphore`, 207
- `kombu.asynchronous.timer`, 208
- `kombu.clocks`, 67
- `kombu.common`, 60
- `kombu.compat`, 68
- `kombu.compression`, 89
- `kombu.connection`, 80
- `kombu.exceptions`, 79
- `kombu.log`, 79
- `kombu.matcher`, 62
- `kombu.message`, 88
- `kombu.mixins`, 63
- `kombu.pidbox`, 77
- `kombu.pools`, 90
- `kombu.resource`, 93
- `kombu.serialization`, 94
- `kombu.simple`, 65
- `kombu.transport`, 96
- `kombu.transport.azure_servicebus`, 108
- `kombu.transport.azurestoragequeues`, 106
- `kombu.transport.base`, 96
- `kombu.transport.consul`, 185
- `kombu.transport.etcd`, 186
- `kombu.transport.filesystem`, 190
- `kombu.transport.memory`, 172
- `kombu.transport.mongodb`, 182
- `kombu.transport.pyamqp`, 111
- `kombu.transport.pyro`, 203
- `kombu.transport.qpid`, 137
- `kombu.transport.redis`, 174
- `kombu.transport.SLMQ`, 201
- `kombu.transport.sqlalchemy`, 192
- `kombu.transport.sqlalchemy.models`, 194
- `kombu.transport.SQS`, 195
- `kombu.transport.virtual`, 99
- `kombu.transport.virtual.exchange`, 104
- `kombu.transport.zookeeper`, 188
- `kombu.utils.amq_manager`, 222
- `kombu.utils.collections`, 222
- `kombu.utils.compat`, 223
- `kombu.utils.debug`, 223
- `kombu.utils.div`, 223
- `kombu.utils.encoding`, 223
- `kombu.utils.eventio`, 224
- `kombu.utils.functional`, 224
- `kombu.utils.imports`, 226
- `kombu.utils.json`, 226
- `kombu.utils.limits`, 227
- `kombu.utils.objects`, 228
- `kombu.utils.scheduling`, 228
- `kombu.utils.text`, 229
- `kombu.utils.time`, 229
- `kombu.utils.url`, 229
- `kombu.utils.uuid`, 230

Symbols

`__len__()` (*kombu.simple.SimpleBuffer* method), 67
`__len__()` (*kombu.simple.SimpleQueue* method), 66
`_close()` (*kombu.Connection* method), 44

A

`abcast()` (*kombu.pidbox.Mailbox* method), 78
`AbstractChannel` (class in *kombu.transport.virtual*), 99
`accept` (*kombu.asynchronous.aws.sqs.message.AsyncMessage* attribute), 220
`accept` (*kombu.asynchronous.aws.sqs.message.AsyncRawMessage* attribute), 220
`accept` (*kombu.asynchronous.aws.sqs.message.BaseAsyncMessage* attribute), 221
`accept` (*kombu.compat.Consumer* attribute), 70
`accept` (*kombu.compat.ConsumerSet* attribute), 74
`accept` (*kombu.message.Message* attribute), 88
`accept` (*kombu.transport.pyamqp.Channel.Message* attribute), 136
`accept` (*kombu.transport.pyamqp.Connection.Channel.Message* attribute), 114
`accept` (*kombu.transport.pyamqp.Message* attribute), 137
`accept` (*kombu.transport.pyamqp.Transport.Connection.Channel.Message* attribute), 113
`accept` (*kombu.transport.qpid.Channel.Message* attribute), 163
`accept` (*kombu.transport.qpid.Connection.Channel.Message* attribute), 154
`accept` (*kombu.transport.qpid.Message* attribute), 172
`accept` (*kombu.transport.qpid.Transport.Connection.Channel.Message* attribute), 141
`accept` (*kombu.transport.virtual.Message* attribute), 101
`ack()` (*kombu.message.Message* method), 88
`ack()` (*kombu.transport.base.Message* method), 97
`ack()` (*kombu.transport.pyamqp.Connection.Channel.Message* method), 114
`ack()` (*kombu.transport.qpid.Channel.QoS* method), 164
`ack()` (*kombu.transport.qpid.Connection.Channel.QoS* method), 154
`ack()` (*kombu.transport.qpid.Transport.Connection.Channel.QoS* method), 141
`ack()` (*kombu.transport.redis.Channel.QoS* method), 178
`ack()` (*kombu.transport.redis.Transport.Channel.QoS* method), 175
`ack()` (*kombu.transport.virtual.Message* method), 101
`ack()` (*kombu.transport.virtual.QoS* method), 102
`ack_emulation` (*kombu.transport.redis.Channel* attribute), 179
`ack_emulation` (*kombu.transport.redis.Transport.Channel* attribute), 176
`ack_log_error()` (*kombu.message.Message* method), 88
`ack_log_error()` (*kombu.transport.pyamqp.Connection.Channel.Message* method), 114
`ack_log_error()` (*kombu.transport.virtual.Message* method), 101
`acknowledged` (*kombu.message.Message* property), 88
`acknowledged` (*kombu.transport.base.Message* attribute), 97
`acknowledged` (*kombu.transport.pyamqp.Connection.Channel.Message* property), 114
`acknowledged` (*kombu.transport.virtual.Message* property), 101
`acquire()` (*kombu.asynchronous.semaphore.LaxBoundedSemaphore* method), 207
`acquire()` (*kombu.connection.ChannelPool* method), 87
`acquire()` (*kombu.connection.ConnectionPool* method), 87
`acquire()` (*kombu.resource.Resource* method), 93
`active_queues` (*kombu.transport.redis.Channel* property), 179
`active_queues` (*kombu.transport.redis.Transport.Channel* property), 176
`add()` (*kombu.asynchronous.Hub* method), 205
`add()` (*kombu.asynchronous.hub.Hub* method), 206
`add()` (*kombu.utils.limits.TokenBucket* method), 227
`add_consumer()` (*kombu.compat.ConsumerSet* method), 74
`add_consumer_from_dict()` (*kombu.compat.ConsumerSet* method), 74
`add_permission()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQS* method), 219

[add_permission\(\)](#) (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* attribute), 70
 method), 221
[add_queue\(\)](#) (*kombu.compat.Consumer* method), 70
[add_queue\(\)](#) (*kombu.compat.ConsumerSet* method), 74
[add_queue\(\)](#) (*kombu.Consumer* method), 58
[add_reader\(\)](#) (*kombu.asynchronous.Hub* method), 205
[add_reader\(\)](#) (*kombu.asynchronous.hub.Hub* method), 206
[add_request\(\)](#) (*kombu.asynchronous.http.curl.CurlClient* method), 216
[add_writer\(\)](#) (*kombu.asynchronous.Hub* method), 205
[add_writer\(\)](#) (*kombu.asynchronous.hub.Hub* method), 206
[adjust\(\)](#) (*kombu.clocks.LamportClock* method), 67
[after_reply_message_received\(\)](#)
 (*kombu.transport.memory.Channel* method), 173
[after_reply_message_received\(\)](#)
 (*kombu.transport.memory.Transport.Channel* method), 173
[after_reply_message_received\(\)](#)
 (*kombu.transport.pyamqp.Connection.Channel* method), 115
[after_reply_message_received\(\)](#)
 (*kombu.transport.pyro.Channel* method), 204
[after_reply_message_received\(\)](#)
 (*kombu.transport.pyro.Transport.Channel* method), 203
[alias](#) (*kombu.Queue* attribute), 53
[annotate\(\)](#) (*kombu.log.LogMixin* method), 79
[append\(\)](#) (*kombu.transport.qpid.Channel.QoS* method), 164
[append\(\)](#) (*kombu.transport.qpid.Connection.Channel.QoS* method), 155
[append\(\)](#) (*kombu.transport.qpid.Transport.Connection.Channel.QoS* method), 142
[append\(\)](#) (*kombu.transport.redis.Channel.QoS* method), 178
[append\(\)](#) (*kombu.transport.redis.Transport.Channel.QoS* method), 175
[append\(\)](#) (*kombu.transport.virtual.QoS* method), 102
[apply_backoff_policy\(\)](#)
 (*kombu.transport.SQS.Channel.QoS* method), 199
[apply_backoff_policy\(\)](#)
 (*kombu.transport.SQS.Transport.Channel.QoS* method), 197
[apply_entry\(\)](#) (*kombu.asynchronous.timer.Timer* method), 208
[args](#) (*kombu.asynchronous.timer.Entry* attribute), 208
[args](#) (*kombu.asynchronous.timer.Timer.Entry* attribute), 208
[args](#) (*kombu.compat.Consumer.ContentDisallowed* attribute), 70
 (*kombu.compat.ConsumerSet.ContentDisallowed* attribute), 74
[args](#) (*kombu.transport.pyamqp.Connection.Channel.Message.MessageState* attribute), 114
[args](#) (*kombu.transport.virtual.Message.MessageStateError* attribute), 101
[arguments](#) (*kombu.Exchange* attribute), 48
[as_dict\(\)](#) (*kombu.Queue* method), 54
[as_uri\(\)](#) (*kombu.Connection* method), 42
[as_uri\(\)](#) (*kombu.connection.Connection* method), 83
[as_uri\(\)](#) (*kombu.transport.azure.servicebus.Transport* class method), 110
[as_url\(\)](#) (in module *kombu.utils.url*), 229
[async_pool](#) (*kombu.transport.redis.Channel* property), 179
[async_pool](#) (*kombu.transport.redis.Transport.Channel* property), 176
[AsyncConnection](#) (class in *kombu.asynchronous.aws.connection*), 217
[AsyncHTTPConnection](#) (class in *kombu.asynchronous.aws.connection*), 217
[AsyncHTTPConnection.Request](#) (class in *kombu.asynchronous.aws.connection*), 217
[AsyncMessage](#) (class in *kombu.asynchronous.aws.sqs.message*), 220
[AsyncQueue](#) (class in *kombu.asynchronous.aws.sqs.queue*), 221
[AsyncRawMessage](#) (class in *kombu.asynchronous.aws.sqs.message*), 220
[AsyncSQSConnection](#) (class in *kombu.asynchronous.aws.sqs.connection*), 219
[asynsqs\(\)](#) (*kombu.transport.SQS.Channel* method), 199
[asynsqs\(\)](#) (*kombu.transport.SQS.Transport.Channel* method), 197
[attrs](#) (*kombu.common.Broadcast* attribute), 60
[attrs](#) (*kombu.Exchange* attribute), 49
[attrs](#) (*kombu.Queue* attribute), 54
[auth_mode](#) (*kombu.asynchronous.aws.connection.AsyncHTTPConnection* attribute), 218
[auth_mode](#) (*kombu.asynchronous.http.base.Request* attribute), 214
[auth_mode](#) (*kombu.asynchronous.http.Request* attribute), 211
[auth_password](#) (*kombu.asynchronous.aws.connection.AsyncHTTPConnection* attribute), 218
[auth_password](#) (*kombu.asynchronous.http.base.Request* attribute), 214
[auth_password](#) (*kombu.asynchronous.http.Request* attribute), 211
[auth_username](#) (*kombu.asynchronous.aws.connection.AsyncHTTPConnection* attribute), 218
[auth_username](#) (*kombu.asynchronous.http.base.Request* attribute), 214

attribute), 214
 auth_username (*kombu.asynchronous.http.Request attribute*), 211
 auto_declare (*kombu.compat.Consumer attribute*), 71
 auto_declare (*kombu.compat.ConsumerSet attribute*), 74
 auto_declare (*kombu.compat.Publisher attribute*), 69
 auto_declare (*kombu.Consumer attribute*), 58
 auto_declare (*kombu.pools.ProducerPool.Producer attribute*), 91
 auto_declare (*kombu.Producer attribute*), 56
 auto_delete (*kombu.compat.Consumer attribute*), 71
 auto_delete (*kombu.compat.Publisher attribute*), 69
 auto_delete (*kombu.Exchange attribute*), 48, 49
 auto_delete (*kombu.Queue attribute*), 52, 54
 autoretry() (*kombu.Connection method*), 43
 autoretry() (*kombu.connection.Connection method*), 83
 AWS_ACCESS_KEY_ID, 297
 AWS_SECRET_ACCESS_KEY, 297

B

backend (*kombu.compat.Publisher property*), 69
 BaseAsyncMessage (class in *kombu.asynchronous.aws.sqs.message*), 221
 basic_ack() (*kombu.transport.azure_servicebus.Channel method*), 110
 basic_ack() (*kombu.transport.azure_servicebus.Transport.Channel method*), 107
 basic_ack() (*kombu.transport.pyamqp.Connection.Channel method*), 116
 basic_ack() (*kombu.transport.qpid.Channel method*), 165
 basic_ack() (*kombu.transport.qpid.Connection.Channel method*), 156
 basic_ack() (*kombu.transport.qpid.Transport.Connection.Channel method*), 142
 basic_ack() (*kombu.transport.SLMQ.Channel method*), 202
 basic_ack() (*kombu.transport.SLMQ.Transport.Channel method*), 201
 basic_ack() (*kombu.transport.SQS.Channel method*), 199
 basic_ack() (*kombu.transport.SQS.Transport.Channel method*), 197
 basic_ack() (*kombu.transport.virtual.Channel method*), 100
 basic_cancel() (*kombu.transport.azure_servicebus.Channel method*), 110
 basic_cancel() (*kombu.transport.azure_servicebus.Transport.Channel method*), 109
 basic_cancel() (*kombu.transport.pyamqp.Connection.Channel method*), 118
 basic_cancel() (*kombu.transport.qpid.Channel method*), 166
 basic_cancel() (*kombu.transport.qpid.Connection.Channel method*), 156
 basic_cancel() (*kombu.transport.qpid.Transport.Connection.Channel method*), 143
 basic_cancel() (*kombu.transport.redis.Channel method*), 179
 basic_cancel() (*kombu.transport.redis.Transport.Channel method*), 176
 basic_cancel() (*kombu.transport.SLMQ.Channel method*), 202
 basic_cancel() (*kombu.transport.SLMQ.Transport.Channel method*), 201
 basic_cancel() (*kombu.transport.SQS.Channel method*), 199
 basic_cancel() (*kombu.transport.SQS.Transport.Channel method*), 197
 basic_cancel() (*kombu.transport.virtual.Channel method*), 100
 basic_consume() (*kombu.transport.azure_servicebus.Channel method*), 110
 basic_consume() (*kombu.transport.azure_servicebus.Transport.Channel method*), 109
 basic_consume() (*kombu.transport.azurestoragequeues.Channel method*), 107
 basic_consume() (*kombu.transport.azurestoragequeues.Transport.Channel method*), 107
 basic_consume() (*kombu.transport.pyamqp.Connection.Channel method*), 117
 basic_consume() (*kombu.transport.qpid.Channel method*), 165
 basic_consume() (*kombu.transport.qpid.Connection.Channel method*), 156
 basic_consume() (*kombu.transport.qpid.Transport.Connection.Channel method*), 143
 basic_consume() (*kombu.transport.redis.Channel method*), 179
 basic_consume() (*kombu.transport.redis.Transport.Channel method*), 176
 basic_consume() (*kombu.transport.SLMQ.Channel method*), 202
 basic_consume() (*kombu.transport.SLMQ.Transport.Channel method*), 201
 basic_consume() (*kombu.transport.SQS.Channel method*), 199
 basic_consume() (*kombu.transport.SQS.Transport.Channel method*), 197
 basic_consume() (*kombu.transport.virtual.Channel method*), 100
 basic_get() (*kombu.transport.pyamqp.Connection.Channel method*), 118

`basic_get()` (*kombu.transport.qpid.Connection.Channel method*), 103
`basic_get()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 103
`basic_get()` (*kombu.transport.qpid.Connection.Channel method*), 144
`basic_get()` (*kombu.transport.virtual.Channel method*), 100
`basic_publish()` (*kombu.transport.pyamqp.Connection.Channel method*), 134
`basic_publish()` (*kombu.transport.pyamqp.Connection.Channel method*), 118
`basic_publish()` (*kombu.transport.qpid.Channel method*), 166
`basic_publish()` (*kombu.transport.qpid.Connection.Channel method*), 157
`basic_publish()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 144
`basic_publish()` (*kombu.transport.virtual.Channel method*), 100
`basic_publish_confirm()` (*kombu.transport.pyamqp.Connection.Channel method*), 120
`basic_qos()` (*kombu.transport.pyamqp.Connection.Channel method*), 120
`basic_qos()` (*kombu.transport.qpid.Channel method*), 167
`basic_qos()` (*kombu.transport.qpid.Connection.Channel method*), 158
`basic_qos()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 144
`basic_qos()` (*kombu.transport.virtual.Channel method*), 100
`basic_recover()` (*kombu.transport.pyamqp.Connection.Channel method*), 120
`basic_recover()` (*kombu.transport.virtual.Channel method*), 100
`basic_recover_async()` (*kombu.transport.pyamqp.Connection.Channel method*), 121
`basic_reject()` (*kombu.transport.pyamqp.Connection.Channel method*), 121
`basic_reject()` (*kombu.transport.qpid.Channel method*), 167
`basic_reject()` (*kombu.transport.qpid.Connection.Channel method*), 158
`basic_reject()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 145
`basic_reject()` (*kombu.transport.virtual.Channel method*), 100
`bind()` (*kombu.abstract.MaybeChannelBound method*), 92
`bind()` (*kombu.Queue method*), 54
`bind_to()` (*kombu.Exchange method*), 49
`bind_to()` (*kombu.Queue method*), 54
`binding()` (*kombu.Exchange method*), 49
`binding_arguments` (*kombu.Queue attribute*), 53
`binding_declare()` (*kombu.transport.virtual.BrokerState method*), 103
`binding_delete()` (*kombu.transport.virtual.BrokerState attribute*), 103
`bindings` (*kombu.transport.virtual.BrokerState attribute*), 103
`blocking_read()` (*kombu.transport.pyamqp.Connection attribute*), 219
`body` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection attribute*), 218
`body` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection.Request attribute*), 220
`body` (*kombu.asynchronous.aws.sqs.message.AsyncMessage attribute*), 220
`body` (*kombu.asynchronous.aws.sqs.message.AsyncRawMessage attribute*), 221
`body` (*kombu.asynchronous.aws.sqs.message.BaseAsyncMessage attribute*), 214
`body` (*kombu.asynchronous.http.base.Request attribute*), 215
`body` (*kombu.asynchronous.http.base.Response property*), 211
`body` (*kombu.asynchronous.http.Request attribute*), 212
`body` (*kombu.asynchronous.http.Response property*), 88
`body` (*kombu.message.Message attribute*), 97
`body` (*kombu.transport.base.Message attribute*), 136
`body` (*kombu.transport.pyamqp.Channel.Message attribute*), 114
`body` (*kombu.transport.pyamqp.Connection.Channel.Message attribute*), 137
`body` (*kombu.transport.pyamqp.Message attribute*), 113
`body` (*kombu.transport.pyamqp.Transport.Connection.Channel.Message attribute*), 164
`body` (*kombu.transport.qpid.Channel.Message attribute*), 154
`body` (*kombu.transport.qpid.Connection.Channel.Message attribute*), 172
`body` (*kombu.transport.qpid.Message attribute*), 141
`body` (*kombu.transport.qpid.Transport.Connection.Channel.Message attribute*), 101
`body_encoding` (*kombu.transport.qpid.Channel attribute*), 167
`body_encoding` (*kombu.transport.qpid.Connection.Channel attribute*), 158
`body_encoding` (*kombu.transport.qpid.Transport.Connection.Channel attribute*), 145
`Broadcast` (*class in kombu.common*), 60
`broadcast` (*kombu.transport.mongodb.Channel attribute*), 184
`broadcast` (*kombu.transport.mongodb.Transport.Channel attribute*), 183
`broadcast_collection` (*kombu.transport.mongodb.Channel attribute*), 184

- `broadcast_collection` (*kombu.transport.mongodb.Transport.Channel* attribute), 183
- `BrokerState` (class in *kombu.transport.virtual*), 103
- `buffer` (*kombu.asynchronous.http.base.Response* attribute), 215, 216
- `buffer` (*kombu.asynchronous.http.Response* attribute), 212
- `bytes_recv` (*kombu.transport.pyamqp.Connection* attribute), 134
- `bytes_sent` (*kombu.transport.pyamqp.Connection* attribute), 134
- `bytes_to_str()` (in module *kombu.utils.encoding*), 223
- ## C
- `ca_certs` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection* attribute), 218
- `ca_certs` (*kombu.asynchronous.http.base.Request* attribute), 214
- `ca_certs` (*kombu.asynchronous.http.Request* attribute), 211
- `cached_property` (class in *kombu.utils.objects*), 228
- `calc_queue_size` (*kombu.transport.mongodb.Channel* attribute), 184
- `calc_queue_size` (*kombu.transport.mongodb.Transport.Channel* attribute), 183
- `call()` (*kombu.pidbox.Mailbox* method), 78
- `call_after()` (*kombu.asynchronous.timer.Timer* method), 208
- `call_at()` (*kombu.asynchronous.Hub* method), 205
- `call_at()` (*kombu.asynchronous.hub.Hub* method), 206
- `call_at()` (*kombu.asynchronous.timer.Timer* method), 208
- `call_later()` (*kombu.asynchronous.Hub* method), 205
- `call_later()` (*kombu.asynchronous.hub.Hub* method), 206
- `call_repeatedly()` (*kombu.asynchronous.Hub* method), 205
- `call_repeatedly()` (*kombu.asynchronous.hub.Hub* method), 206
- `call_repeatedly()` (*kombu.asynchronous.timer.Timer* method), 208
- `call_soon()` (*kombu.asynchronous.Hub* method), 205
- `call_soon()` (*kombu.asynchronous.hub.Hub* method), 206
- `callback_for()` (in module *kombu.asynchronous.debug*), 209
- `callbacks` (*kombu.compat.Consumer* attribute), 71
- `callbacks` (*kombu.compat.ConsumerSet* attribute), 74
- `callbacks` (*kombu.Consumer* attribute), 58
- `can_cache_declaration` (*kombu.abstract.MaybeChannelBound* attribute), 92
- `can_cache_declaration` (*kombu.Exchange* property), 49
- `can_cache_declaration` (*kombu.Queue* property), 54
- `can_consume()` (*kombu.transport.qpid.Channel.QoS* method), 164
- `can_consume()` (*kombu.transport.qpid.Connection.Channel.QoS* method), 155
- `can_consume()` (*kombu.transport.qpid.Transport.Connection.Channel.QoS* method), 142
- `can_consume()` (*kombu.transport.virtual.QoS* method), 102
- `can_consume()` (*kombu.utils.limits.TokenBucket* method), 227
- `can_consume_max_estimate()` (*kombu.transport.qpid.Channel.QoS* method), 164
- `can_consume_max_estimate()` (*kombu.transport.qpid.Connection.Channel.QoS* method), 155
- `can_consume_max_estimate()` (*kombu.transport.qpid.Transport.Connection.Channel.QoS* method), 142
- `can_consume_max_estimate()` (*kombu.transport.virtual.QoS* method), 103
- `can_parse_url` (*kombu.transport.azure.servicebus.Transport* attribute), 110
- `can_parse_url` (*kombu.transport.mongodb.Transport* attribute), 183
- `can_parse_url` (*kombu.transport.sqlalchemy.Transport* attribute), 193
- `cancel()` (*kombu.asynchronous.timer.Entry* method), 208
- `cancel()` (*kombu.asynchronous.timer.Timer* method), 209
- `cancel()` (*kombu.asynchronous.timer.Timer.Entry* method), 208
- `cancel()` (*kombu.compat.Consumer* method), 71
- `cancel()` (*kombu.compat.ConsumerSet* method), 74
- `cancel()` (*kombu.Consumer* method), 59
- `cancel()` (*kombu.Queue* method), 54
- `cancel_by_queue()` (*kombu.compat.Consumer* method), 71
- `cancel_by_queue()` (*kombu.compat.ConsumerSet* method), 74
- `cancel_by_queue()` (*kombu.Consumer* method), 59
- `canceled` (*kombu.asynchronous.timer.Entry* attribute), 208
- `canceled` (*kombu.asynchronous.timer.Timer.Entry* attribute), 208
- `cancelled` (*kombu.asynchronous.timer.Entry* property), 208
- `cancelled` (*kombu.asynchronous.timer.Timer.Entry* property), 208
- `canonical_queue_name()`

- `(kombu.transport.SQS.Channel method)`, 199
- `canonical_queue_name()` (`kombu.transport.SQS.Transport.Channel method`), 197
- `capacity` (`kombu.utils.limits.TokenBucket attribute`), 227
- `capped_queue_size` (`kombu.transport.mongodb.Channel attribute`), 184
- `capped_queue_size` (`kombu.transport.mongodb.Transport.Channel attribute`), 183
- `cast()` (`kombu.pidbox.Mailbox method`), 78
- `change_message_visibility()` (`kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection attribute`), 113
- `change_message_visibility_batch()` (`kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method`), 219
- `change_message_visibility_batch()` (`kombu.asynchronous.aws.sqs.queue.AsyncQueueChannel method`), 221
- `Channel` (class in `kombu.transport.azurestoragequeues`), 110
- `Channel` (class in `kombu.transport.azurestoragequeues`), 107
- `Channel` (class in `kombu.transport.consul`), 186
- `Channel` (class in `kombu.transport.etcd`), 188
- `Channel` (class in `kombu.transport.filesystem`), 192
- `Channel` (class in `kombu.transport.memory`), 173
- `Channel` (class in `kombu.transport.mongodb`), 184
- `Channel` (class in `kombu.transport.pyamqp`), 136
- `Channel` (class in `kombu.transport.pyro`), 204
- `Channel` (class in `kombu.transport.qpid`), 163
- `Channel` (class in `kombu.transport.redis`), 178
- `Channel` (class in `kombu.transport.SLMQ`), 202
- `Channel` (class in `kombu.transport.sqlalchemy`), 194
- `Channel` (class in `kombu.transport.SQS`), 199
- `Channel` (class in `kombu.transport.virtual`), 99
- `Channel` (class in `kombu.transport.zookeeper`), 189
- `channel` (`kombu.abstract.MaybeChannelBound property`), 92
- `channel` (`kombu.asynchronous.aws.sqs.message.AsyncMessage attribute`), 220
- `channel` (`kombu.asynchronous.aws.sqs.message.AsyncRawMessage attribute`), 220
- `channel` (`kombu.asynchronous.aws.sqs.message.BaseAsyncMessage attribute`), 221
- `channel` (`kombu.compat.Consumer attribute`), 71
- `channel` (`kombu.compat.ConsumerSet attribute`), 74
- `channel` (`kombu.compat.Publisher property`), 69
- `channel` (`kombu.Consumer attribute`), 58
- `channel` (`kombu.Exchange attribute`), 48
- `channel` (`kombu.message.Message attribute`), 88
- `channel` (`kombu.pidbox.Node attribute`), 78
- `channel` (`kombu.pools.ProducerPool.Producer property`), 91
- `channel` (`kombu.Producer attribute`), 56
- `channel` (`kombu.Queue attribute`), 52
- `channel` (`kombu.simple.SimpleBuffer attribute`), 66
- `channel` (`kombu.simple.SimpleQueue attribute`), 65
- `channel` (`kombu.transport.base.Message attribute`), 97
- `channel` (`kombu.transport.pyamqp.Channel.Message attribute`), 136
- `channel` (`kombu.transport.pyamqp.Connection.Channel.Message attribute`), 137
- `channel` (`kombu.transport.pyamqp.Message attribute`), 137
- `channel` (`kombu.transport.pyamqp.Transport.Connection.Channel.Message attribute`), 113
- `channel` (`kombu.transport.qpid.Channel.Message attribute`), 164
- `channel` (`kombu.transport.qpid.Connection.Channel.Message attribute`), 154
- `channel` (`kombu.transport.qpid.Message attribute`), 172
- `channel` (`kombu.transport.qpid.Transport.Connection.Channel.Message attribute`), 141
- `channel` (`kombu.transport.virtual.Message attribute`), 101
- `Channel` (`kombu.transport.virtual.Transport attribute`), 99
- `channel()` (`kombu.Connection method`), 42
- `channel()` (`kombu.connection.Connection method`), 83
- `channel()` (`kombu.transport.pyamqp.Connection method`), 134
- `Channel.Message` (class in `kombu.transport.pyamqp`), 136
- `Channel.Message` (class in `kombu.transport.qpid`), 163
- `Channel.QoS` (class in `kombu.transport.qpid`), 164
- `Channel.QoS` (class in `kombu.transport.redis`), 178
- `Channel.QoS` (class in `kombu.transport.SQS`), 199
- `channel_errors` (`kombu.Connection attribute`), 42
- `channel_errors` (`kombu.connection.Connection attribute`), 83
- `channel_errors` (`kombu.mixins.ConsumerMixin attribute`), 64
- `channel_errors` (`kombu.transport.base.Transport attribute`), 98
- `channel_errors` (`kombu.transport.mongodb.Transport attribute`), 183
- `channel_errors` (`kombu.transport.pyamqp.Connection attribute`), 134
- `channel_errors` (`kombu.transport.pyamqp.Transport attribute`), 113
- `channel_errors` (`kombu.transport.qpid.Transport attribute`), 150
- `channel_errors` (`kombu.transport.SQS.Transport attribute`), 198
- `channel_errors` (`kombu.transport.zookeeper.Transport attribute`), 189
- `ChannelLimitExceeded`, 79

- ChannelPool (class in kombu.connection), 87
- ChannelPool() (kombu.Connection method), 45
- ChannelPool() (kombu.connection.Connection method), 81
- clear() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 221
- clear() (kombu.asynchronous.semaphore.LaxBoundedSemaphore method), 208
- clear() (kombu.asynchronous.timer.Timer method), 209
- clear() (kombu.simple.SimpleBuffer method), 67
- clear() (kombu.simple.SimpleQueue method), 66
- clear() (kombu.transport.virtual.BrokerState method), 104
- clear_pending() (kombu.utils.limits.TokenBucket method), 228
- client (kombu.transport.base.Transport attribute), 98
- client (kombu.transport.mongodb.Channel attribute), 184
- client (kombu.transport.mongodb.Transport.Channel attribute), 183
- client (kombu.transport.redis.Channel attribute), 179
- client (kombu.transport.redis.Transport.Channel attribute), 176
- client (kombu.transport.zookeeper.Channel property), 189
- client (kombu.transport.zookeeper.Transport.Channel property), 189
- Client() (in module kombu.asynchronous.http), 210
- client_cert (kombu.asynchronous.aws.connection.AsyncHTTPConnection attribute), 218
- client_cert (kombu.asynchronous.http.base.Request attribute), 214
- client_cert (kombu.asynchronous.http.Request attribute), 211
- client_heartbeat (kombu.transport.pyamqp.Connection attribute), 134
- client_key (kombu.asynchronous.aws.connection.AsyncHTTPConnection attribute), 218
- client_key (kombu.asynchronous.http.base.Request attribute), 214
- client_key (kombu.asynchronous.http.Request attribute), 211
- clock (kombu.clocks.timetuple property), 68
- clone() (kombu.Connection method), 44
- clone() (kombu.connection.Connection method), 83
- close() (kombu.asynchronous.aws.connection.AsyncHTTPConnection method), 219
- close() (kombu.asynchronous.http.curl.CurlClient method), 216
- close() (kombu.asynchronous.Hub method), 205
- close() (kombu.asynchronous.hub.Hub method), 206
- close() (kombu.compat.Consumer method), 71
- close() (kombu.compat.ConsumerSet method), 74
- close() (kombu.compat.Publisher method), 69
- close() (kombu.Connection method), 44
- close() (kombu.connection.Connection method), 83
- close() (kombu.pools.ProducerPool.Producer method), 91
- close() (kombu.simple.SimpleBuffer method), 67
- close() (kombu.simple.SimpleQueue method), 66
- close() (kombu.transport.azure.servicebus.Channel method), 110
- close() (kombu.transport.azure.servicebus.Transport.Channel method), 109
- close() (kombu.transport.memory.Channel method), 174
- close() (kombu.transport.memory.Transport.Channel method), 173
- close() (kombu.transport.pyamqp.Connection method), 134
- close() (kombu.transport.pyamqp.Connection.Channel method), 122
- close() (kombu.transport.pyro.Channel method), 204
- close() (kombu.transport.pyro.Transport.Channel method), 204
- close() (kombu.transport.qpid.Channel method), 167
- close() (kombu.transport.qpid.Connection method), 162
- close() (kombu.transport.qpid.Connection.Channel method), 158
- close() (kombu.transport.qpid.Transport.Connection method), 149
- close() (kombu.transport.qpid.Transport.Connection.Channel method), 145
- close() (kombu.transport.redis.Channel method), 179
- close() (kombu.transport.redis.Transport.Channel method), 176
- close() (kombu.transport.SQS.Channel method), 199
- close() (kombu.transport.SQS.Transport.Channel method), 197
- close() (kombu.transport.virtual.Channel method), 101
- close() (kombu.utils.scheduling.FairCycle method), 228
- close_after_fork (kombu.pools.ProducerPool attribute), 92
- close_after_fork (kombu.resource.Resource attribute), 93
- close_channel() (kombu.transport.base.Transport method), 98
- close_channel() (kombu.transport.qpid.Connection method), 162
- close_channel() (kombu.transport.qpid.Transport.Connection method), 149
- close_channel() (kombu.transport.virtual.Transport method), 99
- close_connection() (kombu.transport.base.Transport method), 98
- close_connection() (kombu.transport.pyamqp.Transport

[method](#), 113
[close_connection\(\)](#) ([kombu.transport.qpid.Transport](#) [method](#)), 150
[close_connection\(\)](#) ([kombu.transport.virtual.Transport](#) [method](#)), 99
[close_resource\(\)](#) ([kombu.pools.ProducerPool](#) [method](#)), 92
[close_resource\(\)](#) ([kombu.resource.Resource](#) [method](#)), 93
[code](#) ([kombu.asynchronous.http.base.Response](#) [attribute](#)), 215, 216
[code](#) ([kombu.asynchronous.http.Response](#) [attribute](#)), 212
[codecs](#) ([kombu.transport.qpid.Channel](#) [attribute](#)), 167
[codecs](#) ([kombu.transport.qpid.Connection.Channel](#) [attribute](#)), 158
[codecs](#) ([kombu.transport.qpid.Transport.Connection.Channel](#) [attribute](#)), 145
[collect\(\)](#) ([kombu.connection.Connection](#) [method](#)), 84
[collect\(\)](#) ([kombu.transport.pyamqp.Connection](#) [method](#)), 135
[collect\(\)](#) ([kombu.transport.pyamqp.Connection.Channel](#) [method](#)), 122
[collect_replies\(\)](#) (in module [kombu.common](#)), 60
[collect_resource\(\)](#) ([kombu.resource.Resource](#) [method](#)), 93
[complete](#) ([kombu.asynchronous.http.base.Headers](#) [attribute](#)), 213
[complete](#) ([kombu.asynchronous.http.Headers](#) [attribute](#)), 210
[completes_cycle\(\)](#) ([kombu.Connection](#) [method](#)), 45
[completes_cycle\(\)](#) ([kombu.connection.Connection](#) [method](#)), 84
[compress\(\)](#) (in module [kombu.compression](#)), 90
[compression](#) ([kombu.compat.Publisher](#) [attribute](#)), 69
[compression](#) ([kombu.pools.ProducerPool.Producer](#) [attribute](#)), 91
[compression](#) ([kombu.Producer](#) [attribute](#)), 56
[confirm_select\(\)](#) ([kombu.transport.pyamqp.Connection](#) [method](#)), 122
[conn_or_acquire\(\)](#) ([kombu.transport.redis.Channel](#) [method](#)), 179
[conn_or_acquire\(\)](#) ([kombu.transport.redis.Transport.Channel](#) [method](#)), 176
[connect\(\)](#) ([kombu.asynchronous.aws.connection.AsyncHTTPConnection](#) [method](#)), 219
[connect\(\)](#) ([kombu.Connection](#) [method](#)), 42
[connect\(\)](#) ([kombu.connection.Connection](#) [method](#)), 84
[connect\(\)](#) ([kombu.transport.pyamqp.Connection](#) [method](#)), 135
[connect_max_retries](#) ([kombu.mixins.ConsumerMixin](#) [attribute](#)), 64
[connect_sqs\(\)](#) (in module [kombu.asynchronous.aws](#)), 216
[connect_timeout](#) ([kombu.asynchronous.aws.connection.AsyncHTTPConnection](#) [attribute](#)), 218
[connect_timeout](#) ([kombu.asynchronous.http.base.Request](#) [attribute](#)), 214
[connect_timeout](#) ([kombu.asynchronous.http.Request](#) [attribute](#)), 211
[connect_timeout](#) ([kombu.Connection](#) [attribute](#)), 41
[connect_timeout](#) ([kombu.connection.Connection](#) [attribute](#)), 84
[connect_timeout](#) ([kombu.transport.mongodb.Channel](#) [attribute](#)), 184
[connect_timeout](#) ([kombu.transport.mongodb.Transport.Channel](#) [attribute](#)), 183
[connected](#) ([kombu.Connection](#) [attribute](#)), 41
[connected](#) ([kombu.connection.Connection](#) [property](#)), 84
[connected](#) ([kombu.transport.pyamqp.Connection](#) [property](#)), 135
[Connection](#) (class in [kombu](#)), 40
[Connection](#) (class in [kombu.connection](#)), 80
[Connection](#) (class in [kombu.transport.pyamqp](#)), 114
[Connection](#) (class in [kombu.transport.qpid](#)), 152
[connection](#) ([kombu.compat.Consumer](#) [property](#)), 71
[connection](#) ([kombu.compat.ConsumerSet](#) [property](#)), 75
[connection](#) ([kombu.compat.Publisher](#) [property](#)), 69
[connection](#) ([kombu.Connection](#) [attribute](#)), 42
[connection](#) ([kombu.connection.Connection](#) [property](#)), 84
[connection](#) ([kombu.Consumer](#) [attribute](#)), 58
[connection](#) ([kombu.pidbox.Mailbox](#) [attribute](#)), 78
[connection](#) ([kombu.pools.ProducerPool.Producer](#) [property](#)), 91
[connection](#) ([kombu.Producer](#) [attribute](#)), 56
[Connection.Channel](#) (class in [kombu.transport.pyamqp](#)), 114
[Connection.Channel](#) (class in [kombu.transport.qpid](#)), 153
[Connection.Channel.Message](#) (class in [kombu.transport.pyamqp](#)), 114
[Connection.Channel.Message](#) (class in [kombu.transport.qpid](#)), 154
[Connection.Channel.Message.MessageStateError](#), 114
[Connection.Channel.QoS](#) (class in [kombu.transport.qpid](#)), 154
[ConnectionError](#) (class in [kombu.transport.redis.Channel](#) [attribute](#)), 179
[connection_class](#) ([kombu.transport.redis.SentinelChannel](#) [attribute](#)), 181
[connection_class](#) ([kombu.transport.redis.Transport.Channel](#) [attribute](#)), 176
[connection_class_ssl](#) ([kombu.transport.redis.Channel](#) [attribute](#)), 179
[connection_class_ssl](#) ([kombu.transport.redis.SentinelChannel](#) [attribute](#)), 181

- tribute), 181
- connection_class_ssl (kombu.transport.redis.Transport.Channel attribute), 176
- connection_errors (kombu.Connection attribute), 42
- connection_errors (kombu.connection.Connection attribute), 84
- connection_errors (kombu.mixins.ConsumerMixin attribute), 64
- connection_errors (kombu.transport.base.Transport attribute), 98
- connection_errors (kombu.transport.mongodb.Transport attribute), 184
- connection_errors (kombu.transport.pyamqp.Connection attribute), 135
- connection_errors (kombu.transport.pyamqp.Transport attribute), 113
- connection_errors (kombu.transport.qpid.Transport attribute), 150
- connection_errors (kombu.transport.SLMQ.Transport attribute), 202
- connection_errors (kombu.transport.sqlalchemy.Transport attribute), 193
- connection_errors (kombu.transport.SQS.Transport attribute), 198
- connection_errors (kombu.transport.zookeeper.Transport attribute), 189
- ConnectionLimitExceeded, 79
- ConnectionPool (class in kombu.connection), 87
- conninfo (kombu.transport.azurestoragequeues.Channel property), 110
- conninfo (kombu.transport.azurestoragequeues.Transport.Channel property), 109
- conninfo (kombu.transport.azurestoragequeues.Channel property), 107
- conninfo (kombu.transport.azurestoragequeues.Transport.Channel property), 107
- conninfo (kombu.transport.SLMQ.Channel property), 202
- conninfo (kombu.transport.SLMQ.Transport.Channel property), 201
- conninfo (kombu.transport.SQS.Channel property), 199
- conninfo (kombu.transport.SQS.Transport.Channel property), 197
- consume() (kombu.compat.Consumer method), 71
- consume() (kombu.compat.ConsumerSet method), 75
- consume() (kombu.Consumer method), 59
- consume() (kombu.mixins.ConsumerMixin method), 64
- consume() (kombu.Queue method), 54
- consume() (kombu.utils.scheduling.round_robin_cycle method), 228
- consume() (kombu.utils.scheduling.sorted_cycle method), 229
- Consumer (class in kombu), 57
- Consumer (class in kombu.compat), 70
- consumer (kombu.simple.SimpleBuffer attribute), 66
- consumer (kombu.simple.SimpleQueue attribute), 65
- Consumer() (kombu.Connection method), 45
- Consumer() (kombu.connection.Connection method), 82
- Consumer() (kombu.mixins.ConsumerMixin method), 64
- Consumer() (kombu.pidbox.Node method), 78
- Consumer() (kombu.transport.pyamqp.Connection.Channel method), 114
- Consumer.ContentDisallowed, 70
- consumer_arguments (kombu.Queue attribute), 53
- consumer_context() (kombu.mixins.ConsumerMixin method), 64
- ConsumerMixin (class in kombu.mixins), 63
- ConsumerProducerMixin (class in kombu.mixins), 64
- ConsumerSet (class in kombu.compat), 74
- ConsumerSet.ContentDisallowed, 74
- consuming_from() (kombu.compat.Consumer method), 71
- consuming_from() (kombu.compat.ConsumerSet method), 75
- consuming_from() (kombu.Consumer method), 59
- content (kombu.asynchronous.http.base.Response property), 216
- content (kombu.asynchronous.http.Response property), 212
- content_encoding (kombu.asynchronous.aws.sqs.message.AsyncMessage attribute), 220
- content_encoding (kombu.asynchronous.aws.sqs.message.AsyncRawMessage attribute), 221
- content_encoding (kombu.asynchronous.aws.sqs.message.BaseAsyncMessage attribute), 221
- content_encoding (kombu.message.Message attribute), 88
- content_encoding (kombu.transport.base.Message attribute), 97
- content_encoding (kombu.transport.pyamqp.Channel.Message attribute), 136
- content_encoding (kombu.transport.pyamqp.Connection.Channel.Message attribute), 115
- content_encoding (kombu.transport.pyamqp.Message attribute), 137
- content_encoding (kombu.transport.pyamqp.Transport.Connection.Channel.Message attribute), 113
- content_encoding (kombu.transport.qpid.Channel.Message attribute), 164
- content_encoding (kombu.transport.qpid.Connection.Channel.Message attribute), 154
- content_encoding (kombu.transport.qpid.Message attribute), 172
- content_encoding (kombu.transport.qpid.Transport.Connection.Channel.Message attribute), 141
- content_encoding (kombu.transport.virtual.Message attribute), 101

- `content_type` (*kombu.asynchronous.aws.sqs.message.AsyncMessage* attribute), 220
 - `content_type` (*kombu.asynchronous.aws.sqs.message.AsyncRawMessage* attribute), 221
 - `content_type` (*kombu.asynchronous.aws.sqs.message.BaseAsyncMessage* attribute), 221
 - `content_type` (*kombu.message.Message* attribute), 88
 - `content_type` (*kombu.transport.base.Message* attribute), 97
 - `content_type` (*kombu.transport.pyamqp.Channel.Message* attribute), 136
 - `content_type` (*kombu.transport.pyamqp.Connection.Channel.Message* attribute), 115
 - `content_type` (*kombu.transport.pyamqp.Message* attribute), 137
 - `content_type` (*kombu.transport.pyamqp.Transport.Connection.Channel.Message* attribute), 113
 - `content_type` (*kombu.transport.qpid.Channel.Message* attribute), 164
 - `content_type` (*kombu.transport.qpid.Connection.Channel.Message* attribute), 154
 - `content_type` (*kombu.transport.qpid.Message* attribute), 172
 - `content_type` (*kombu.transport.qpid.Transport.Connection.Channel.Message* attribute), 141
 - `content_type` (*kombu.transport.virtual.Message* attribute), 101
 - `coro()` (in module *kombu.utils.compat*), 223
 - `count()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* method), 221
 - `count_slow()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* method), 221
 - `create()` (*kombu.pools.PoolGroup* method), 90
 - `create_channel()` (*kombu.transport.base.Transport* method), 98
 - `create_channel()` (*kombu.transport.pyamqp.Transport* method), 113
 - `create_channel()` (*kombu.transport.qpid.Transport* method), 150
 - `create_channel()` (*kombu.transport.virtual.Transport* method), 99
 - `create_connection()` (*kombu.mixins.ConsumerMixin* method), 64
 - `create_loop()` (*kombu.asynchronous.Hub* method), 205
 - `create_loop()` (*kombu.asynchronous.hub.Hub* method), 206
 - `create_producer()` (*kombu.pools.ProducerPool* method), 92
 - `create_queue()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection* method), 219
 - `create_transport()` (*kombu.Connection* method), 44
 - `create_transport()` (*kombu.connection.Connection* method), 84
 - `debug()` (*kombu.log.LogMixin* method), 79
 - `Curl` (*kombu.asynchronous.http.curl.CurlClient* attribute), 216
 - `CurlClient` (class in *kombu.asynchronous.http.curl*), 216
 - `cycle` (*kombu.Connection* attribute), 42
 - `cycle` (*kombu.connection.Connection* attribute), 84
 - `Cycle` (*kombu.transport.virtual.Transport* attribute), 99
 - `cycle` (*kombu.transport.virtual.Transport* attribute), 99
- ## D
- `data_folder_in` (*kombu.transport.filesystem.Channel* attribute), 192
 - `data_folder_in` (*kombu.transport.filesystem.Transport.Channel* attribute), 191
 - `data_folder_out` (*kombu.transport.filesystem.Channel* attribute), 192
 - `data_folder_out` (*kombu.transport.filesystem.Transport.Channel* attribute), 191
 - `debug()` (*kombu.log.LogMixin* method), 79
 - `declare()` (*kombu.compat.Consumer* method), 71
 - `declare()` (*kombu.compat.ConsumerSet* method), 75
 - `declare()` (*kombu.compat.Publisher* method), 69
 - `declare()` (*kombu.Consumer* method), 58
 - `declare()` (*kombu.Exchange* method), 49
 - `declare()` (*kombu.pools.ProducerPool.Producer* method), 91
 - `declare()` (*kombu.Producer* method), 56
 - `declare()` (*kombu.Queue* method), 54
 - `declared_entities` (*kombu.Connection* attribute), 42
 - `declared_entities` (*kombu.connection.Connection* attribute), 84
 - `decode()` (*kombu.message.Message* method), 88
 - `decode()` (*kombu.transport.base.Message* method), 98
 - `decode()` (*kombu.transport.pyamqp.Connection.Channel.Message* method), 115
 - `decode()` (*kombu.transport.virtual.Message* method), 101
 - `decode_body()` (*kombu.transport.qpid.Channel* method), 167
 - `decode_body()` (*kombu.transport.qpid.Connection.Channel* method), 158
 - `decode_body()` (*kombu.transport.qpid.Transport.Connection.Channel* method), 145
 - `decompress()` (in module *kombu.compression*), 90
 - `default()` (*kombu.utils.json.JSONEncoder* method), 226
 - `default_channel` (*kombu.Connection* attribute), 41
 - `default_channel` (*kombu.connection.Connection* property), 84
 - `default_connection_params` (*kombu.transport.pyamqp.Transport* property), 113

`default_connection_params` (*kombu.transport.qpid.Transport* property), 150
`default_connection_params` (*kombu.transport.SQS.Transport* property), 198
`default_database` (*kombu.transport.mongodb.Channel* attribute), 184
`default_database` (*kombu.transport.mongodb.Transport.Channel* attribute), 183
`default_encode()` (in module *kombu.utils.encoding*), 223
`default_encoding()` (in module *kombu.utils.encoding*), 223
`default_encoding_file` (in module *kombu.utils.encoding*), 224
`default_hostname` (*kombu.transport.mongodb.Channel* attribute), 184
`default_hostname` (*kombu.transport.mongodb.Transport.Channel* attribute), 183
`default_peek_lock_seconds` (*kombu.transport.azurestoragequeues.Channel* attribute), 110
`default_peek_lock_seconds` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 109
`default_port` (*kombu.transport.azurestoragequeues.Transport* attribute), 110
`default_port` (*kombu.transport.azurestoragequeues.Transport* attribute), 107
`default_port` (*kombu.transport.base.Transport* attribute), 98
`default_port` (*kombu.transport.consul.Transport* attribute), 186
`default_port` (*kombu.transport.etcd.Transport* attribute), 187
`default_port` (*kombu.transport.filesystem.Transport* attribute), 191
`default_port` (*kombu.transport.mongodb.Channel* attribute), 184
`default_port` (*kombu.transport.mongodb.Transport* attribute), 184
`default_port` (*kombu.transport.mongodb.Transport.Channel* attribute), 183
`default_port` (*kombu.transport.pyamqp.Transport* attribute), 113
`default_port` (*kombu.transport.pyro.Transport* attribute), 204
`default_port` (*kombu.transport.redis.Transport* attribute), 178
`default_port` (*kombu.transport.SLMQ.Transport* attribute), 202
`default_port` (*kombu.transport.sqlalchemy.Transport* attribute), 194
`default_port` (*kombu.transport.SQS.Transport* attribute), 198
`default_port` (*kombu.transport.virtual.Transport* attribute), 99
`default_port` (*kombu.transport.zookeeper.Transport* attribute), 189
`default_ports` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection* attribute), 219
`default_region` (*kombu.transport.SQS.Channel* attribute), 199
`default_region` (*kombu.transport.SQS.Transport.Channel* attribute), 197
`default_retry_backoff_factor` (*kombu.transport.azurestoragequeues.Channel* attribute), 110
`default_retry_backoff_factor` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 109
`default_retry_backoff_max` (*kombu.transport.azurestoragequeues.Channel* attribute), 110
`default_retry_backoff_max` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 109
`default_retry_total` (*kombu.transport.azurestoragequeues.Channel* attribute), 110
`default_retry_total` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 109
`default_ssl_port` (*kombu.transport.pyamqp.Transport* attribute), 113
`DEFAULT_TRANSPORT` (in module *kombu.transport*), 96
`default_uamqp_keep_alive_interval` (*kombu.transport.azurestoragequeues.Channel* attribute), 110
`default_uamqp_keep_alive_interval` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 109
`default_visibility_timeout` (*kombu.transport.SLMQ.Channel* attribute), 202
`default_visibility_timeout` (*kombu.transport.SLMQ.Transport.Channel* attribute), 201
`default_visibility_timeout` (*kombu.transport.SQS.Channel* attribute), 199
`default_visibility_timeout` (*kombu.transport.SQS.Transport.Channel* attribute), 197
`default_wait_time_seconds` (*kombu.transport.azurestoragequeues.Channel* attribute), 110

default_wait_time_seconds (kombu.transport.azure.servicebus.Transport.Channel attribute), 109

default_wait_time_seconds (kombu.transport.SQS.Channel attribute), 199

default_wait_time_seconds (kombu.transport.SQS.Transport.Channel attribute), 197

delete() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 221

delete() (kombu.Exchange method), 49

delete() (kombu.Queue method), 54

delete_message() (kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method), 219

delete_message() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 221

delete_message() (kombu.transport.SLMQ.Channel method), 202

delete_message() (kombu.transport.SLMQ.Transport.Channel method), 201

delete_message_batch() (kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method), 219

delete_message_batch() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 221

delete_message_from_handle() (kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method), 219

delete_queue() (kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method), 219

deleter() (kombu.utils.objects.cached_property method), 228

deliver() (kombu.transport.virtual.exchange.DirectExchange method), 104

deliver() (kombu.transport.virtual.exchange.FanoutExchange method), 105

deliver() (kombu.transport.virtual.exchange.TopicExchange method), 105

delivery_info (kombu.asynchronous.aws.sqs.message.AsyncMessage attribute), 220

delivery_info (kombu.asynchronous.aws.sqs.message.AsyncRawMessage attribute), 221

delivery_info (kombu.asynchronous.aws.sqs.message.BaseAsyncMessage attribute), 221

delivery_info (kombu.message.Message attribute), 89

delivery_info (kombu.transport.base.Message attribute), 97

delivery_info (kombu.transport.pyamqp.Channel.Message attribute), 136

delivery_info (kombu.transport.pyamqp.Connection.Channel.Message attribute), 115

delivery_info (kombu.transport.pyamqp.Message attribute), 137

delivery_info (kombu.transport.pyamqp.Transport.Connection.Channel.Message attribute), 113

delivery_info (kombu.transport.qpid.Channel.Message attribute), 164

delivery_info (kombu.transport.qpid.Connection.Channel.Message attribute), 154

delivery_info (kombu.transport.qpid.Message attribute), 172

delivery_info (kombu.transport.qpid.Transport.Connection.Channel.Message attribute), 141

delivery_info (kombu.transport.virtual.Message attribute), 102

delivery_info (kombu.transport.virtual.Exchange attribute), 48, 50

delivery_tag (kombu.asynchronous.aws.sqs.message.AsyncMessage attribute), 220

delivery_tag (kombu.asynchronous.aws.sqs.message.AsyncRawMessage attribute), 221

delivery_tag (kombu.asynchronous.aws.sqs.message.BaseAsyncMessage attribute), 221

delivery_tag (kombu.message.Message attribute), 89

delivery_tag (kombu.transport.base.Message attribute), 97

delivery_tag (kombu.transport.pyamqp.Channel.Message attribute), 136

delivery_tag (kombu.transport.pyamqp.Connection.Channel.Message attribute), 115

delivery_tag (kombu.transport.pyamqp.Message attribute), 137

delivery_tag (kombu.transport.pyamqp.Transport.Connection.Channel.Message attribute), 113

delivery_tag (kombu.transport.qpid.Channel.Message attribute), 164

delivery_tag (kombu.transport.qpid.Connection.Channel.Message attribute), 154

delivery_tag (kombu.transport.qpid.Message attribute), 172

delivery_tag (kombu.transport.qpid.Transport.Connection.Channel.Message attribute), 141

delivery_tag (kombu.transport.virtual.Message attribute), 102

detect_environment() (in module kombu.utils.compat), 223

dictfilter() (in module kombu.utils.functional), 225

DirectExchange (class in kombu.transport.virtual.exchange), 104

disable_insecure_serializers() (in module kombu), 39

discard_all() (kombu.compat.Consumer method), 71

discard_all() (kombu.compat.ConsumerSet method), 75

dispatch_message() (kombu.pidbox.Node method), 79

dispatch_from_message() (kombu.pidbox.Node method), 79

`dispatch_method()` (*kombu.transport.pyamqp.Connection* attribute), 173
`dispatch_method()` (*kombu.transport.pyamqp.Connection.Channel* attribute), 135
`dispatch_method()` (*kombu.transport.pyamqp.Connection.Channel* attribute), 184
`dispatch_method()` (*kombu.transport.pyamqp.Connection.Channel* attribute), 123
`DjangoPromise` (*class in kombu.utils.json*), 226
`do_restore` (*kombu.transport.memory.Channel* attribute), 174
`do_restore` (*kombu.transport.memory.Transport.Channel* attribute), 173
`do_restore` (*kombu.transport.virtual.Channel* attribute), 100
`domain_format` (*kombu.transport.azurestoragequeues.Channel* attribute), 110
`domain_format` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 109
`domain_format` (*kombu.transport.azurestoragequeues.Channel* attribute), 107
`domain_format` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 107
`domain_format` (*kombu.transport.SLMQ.Channel* attribute), 202
`domain_format` (*kombu.transport.SLMQ.Transport.Channel* attribute), 201
`domain_format` (*kombu.transport.SQS.Channel* attribute), 199
`domain_format` (*kombu.transport.SQS.Transport.Channel* attribute), 197
`drain_consumer()` (*in module kombu.common*), 60
`drain_events()` (*kombu.Connection* method), 42
`drain_events()` (*kombu.connection.Connection* method), 84
`drain_events()` (*kombu.transport.base.Transport* method), 98
`drain_events()` (*kombu.transport.pyamqp.Connection* method), 135
`drain_events()` (*kombu.transport.pyamqp.Transport* method), 113
`drain_events()` (*kombu.transport.qpid.Transport* method), 150
`drain_events()` (*kombu.transport.SQS.Channel* method), 199
`drain_events()` (*kombu.transport.SQS.Transport.Channel* method), 198
`drain_events()` (*kombu.transport.virtual.Channel* method), 101
`drain_events()` (*kombu.transport.virtual.Transport* method), 99
`driver_name` (*kombu.transport.consul.Transport* attribute), 186
`driver_name` (*kombu.transport.etcd.Transport* attribute), 187
`driver_name` (*kombu.transport.filesystem.Transport* attribute), 192
`driver_name` (*kombu.transport.memory.Transport* attribute), 173
`driver_name` (*kombu.transport.pyamqp.Connection* attribute), 173
`driver_name` (*kombu.transport.pyamqp.Connection.Channel* attribute), 184
`driver_name` (*kombu.transport.pyamqp.Transport* attribute), 113
`driver_name` (*kombu.transport.pyro.Transport* attribute), 204
`driver_name` (*kombu.transport.qpid.Transport* attribute), 150
`driver_name` (*kombu.transport.redis.Transport* attribute), 178
`driver_name` (*kombu.transport.sqlalchemy.Transport* attribute), 194
`driver_name` (*kombu.transport.SQS.Transport* attribute), 198
`driver_name` (*kombu.transport.zookeeper.Transport* attribute), 189
`driver_type` (*kombu.transport.consul.Transport* attribute), 186
`driver_type` (*kombu.transport.etcd.Transport* attribute), 187
`driver_type` (*kombu.transport.filesystem.Transport* attribute), 192
`driver_type` (*kombu.transport.memory.Transport* attribute), 173
`driver_type` (*kombu.transport.mongodb.Transport* attribute), 184
`driver_type` (*kombu.transport.pyamqp.Transport* attribute), 113
`driver_type` (*kombu.transport.pyro.Transport* attribute), 204
`driver_type` (*kombu.transport.qpid.Transport* attribute), 150
`driver_type` (*kombu.transport.redis.Transport* attribute), 178
`driver_type` (*kombu.transport.sqlalchemy.Transport* attribute), 194
`driver_type` (*kombu.transport.SQS.Transport* attribute), 198
`driver_type` (*kombu.transport.zookeeper.Transport* attribute), 189
`driver_version()` (*kombu.transport.consul.Transport* method), 186
`driver_version()` (*kombu.transport.etcd.Transport* method), 187
`driver_version()` (*kombu.transport.filesystem.Transport* method), 192
`driver_version()` (*kombu.transport.memory.Transport* method), 173
`driver_version()` (*kombu.transport.mongodb.Transport* method), 184
`driver_version()` (*kombu.transport.pyamqp.Transport* method), 113
`driver_version()` (*kombu.transport.pyro.Transport* method), 204

- method*), 204
- `driver_version()` (*kombu.transport.redis.Transport method*), 178
- `driver_version()` (*kombu.transport.sqlalchemy.Transport method*), 194
- `driver_version()` (*kombu.transport.zookeeper.Transport method*), 189
- `DummyLock` (*class in kombu.asynchronous.semaphore*), 207
- `dump()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue method*), 221
- `dumps()` (*in module kombu.serialization*), 94
- `dumps()` (*in module kombu.utils.json*), 227
- `durable` (*kombu.compat.Consumer attribute*), 71
- `durable` (*kombu.compat.Publisher attribute*), 69
- `durable` (*kombu.Exchange attribute*), 48, 50
- `durable` (*kombu.Queue attribute*), 52, 54
- E**
- `effective_url` (*kombu.asynchronous.http.base.Response attribute*), 215, 216
- `effective_url` (*kombu.asynchronous.http.Response attribute*), 212
- `emergency_dump_state()` (*in module kombu.utils.div*), 223
- `enable_insecure_serializers()` (*in module kombu*), 39
- `encode()` (*kombu.asynchronous.aws.sqs.message.AsyncMessage method*), 220
- `encode_body()` (*kombu.transport.qpid.Channel method*), 168
- `encode_body()` (*kombu.transport.qpid.Connection.Channel method*), 158
- `encode_body()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 145
- `encoders()` (*in module kombu.compression*), 90
- `endheaders()` (*kombu.asynchronous.aws.connection.AsyncConnection method*), 219
- `endpoint_url` (*kombu.transport.SQS.Channel attribute*), 199
- `endpoint_url` (*kombu.transport.SQS.Transport.Channel attribute*), 198
- `ensure()` (*kombu.Connection method*), 43
- `ensure()` (*kombu.connection.Connection method*), 84
- `ensure_bytes()` (*in module kombu.utils.encoding*), 224
- `ensure_connection()` (*kombu.Connection method*), 43
- `ensure_connection()` (*kombu.connection.Connection method*), 85
- `enter_after()` (*kombu.asynchronous.timer.Timer method*), 209
- `enter_at()` (*kombu.asynchronous.timer.Timer method*), 209
- `entity_name()` (*kombu.transport.azurestoragequeues.Channel method*), 110
- `entity_name()` (*kombu.transport.azurestoragequeues.Transport.Channel method*), 109
- `entity_name()` (*kombu.transport.azurestoragequeues.Channel method*), 107
- `entity_name()` (*kombu.transport.azurestoragequeues.Transport.Channel method*), 107
- `entity_name()` (*kombu.transport.SLMQ.Channel method*), 202
- `entity_name()` (*kombu.transport.SLMQ.Transport.Channel method*), 201
- `entity_name()` (*kombu.transport.SQS.Channel method*), 199
- `entity_name()` (*kombu.transport.SQS.Transport.Channel method*), 198
- `Entry` (*class in kombu.asynchronous.timer*), 208
- `entrypoints()` (*in module kombu.utils.compat*), 223
- environment variable
 - AWS_ACCESS_KEY_ID, 297
 - AWS_SECRET_ACCESS_KEY, 297
 - KOMBU_LOG_CHANNEL, 300
 - KOMBU_LOG_CONNECTION, 299
 - KOMBU_LOG_DEBUG, 299, 300
 - PICKLE_PROTOCOL, 31, 278
 - URL, 282
 - VHOST, 282
- `eqhash()` (*in module kombu.utils.collections*), 222
- `EqualityDict` (*class in kombu.utils.collections*), 222
- `equivalent()` (*kombu.transport.virtual.exchange.ExchangeType method*), 106
- `ERR` (*kombu.asynchronous.Hub attribute*), 205
- `ERR` (*kombu.asynchronous.hub.Hub attribute*), 206
- `error` (*kombu.asynchronous.http.base.Response attribute*), 215, 216
- `error` (*kombu.asynchronous.http.Response attribute*), 212
- `error()` (*kombu.log.LogMixin method*), 79
- `errors` (*kombu.message.Message attribute*), 89
- `errors` (*kombu.transport.pyamqp.Connection.Channel.Message attribute*), 115
- `errors` (*kombu.transport.virtual.Message attribute*), 102
- `escape_regex()` (*in module kombu.utils.text*), 229
- `establish_connection()`
 - (*kombu.mixins.ConsumerMixin method*), 64
- `establish_connection()`
 - (*kombu.transport.base.Transport method*), 98
- `establish_connection()`
 - (*kombu.transport.pyamqp.Transport method*), 113
- `establish_connection()`
 - (*kombu.transport.qpid.Transport method*), 150
- `establish_connection()`

- `(kombu.transport.virtual.Transport method)`, 99
 - `evaluate()` (*kombu.utils.functional.lazy method*), 225
 - `eventloop()` (*in module kombu.common*), 60
 - `events` (*kombu.transport.memory.Channel attribute*), 174
 - `events` (*kombu.transport.memory.Transport.Channel attribute*), 173
 - `Exchange` (*class in kombu*), 47
 - `exchange` (*kombu.compat.Consumer attribute*), 71
 - `exchange` (*kombu.compat.Publisher attribute*), 69
 - `exchange` (*kombu.pidbox.Mailbox attribute*), 78
 - `exchange` (*kombu.pools.ProducerPool.Producer attribute*), 91
 - `exchange` (*kombu.Producer attribute*), 56
 - `exchange` (*kombu.Queue attribute*), 51, 55
 - `exchange_bind()` (*kombu.transport.pyamqp.Connection.Channel method*), 123
 - `exchange_declare()` (*kombu.transport.pyamqp.Connection.Channel method*), 124
 - `exchange_declare()` (*kombu.transport.qpid.Channel method*), 168
 - `exchange_declare()` (*kombu.transport.qpid.Connection.Channel attribute*), 159
 - `exchange_declare()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 146
 - `exchange_declare()` (*kombu.transport.virtual.Channel method*), 100
 - `exchange_delete()` (*kombu.transport.pyamqp.Connection.Channel method*), 125
 - `exchange_delete()` (*kombu.transport.qpid.Channel method*), 168
 - `exchange_delete()` (*kombu.transport.qpid.Connection.Channel method*), 159
 - `exchange_delete()` (*kombu.transport.qpid.Transport.Connection.Channel method*), 146
 - `exchange_delete()` (*kombu.transport.virtual.Channel method*), 100
 - `exchange_opts` (*kombu.simple.SimpleBuffer attribute*), 66
 - `exchange_opts` (*kombu.simple.SimpleQueue attribute*), 66
 - `exchange_type` (*kombu.compat.Consumer attribute*), 72
 - `exchange_type` (*kombu.compat.Publisher attribute*), 69
 - `exchange_types` (*kombu.transport.virtual.Channel attribute*), 100
 - `exchange_unbind()` (*kombu.transport.pyamqp.Connection.Channel method*), 126
 - `exchanges` (*kombu.transport.virtual.BrokerState attribute*), 104
 - `ExchangeType` (*class in kombu.transport.virtual.exchange*), 106
 - `exclusive` (*kombu.compat.Consumer attribute*), 72
 - `exclusive` (*kombu.Queue attribute*), 52, 55
 - `expected_time()` (*kombu.utils.limits.TokenBucket method*), 228
 - `expires` (*kombu.Queue attribute*), 52
 - `extra_context()` (*kombu.mixins.ConsumerMixin method*), 64
 - `extract_task_name_and_number_of_retries()` (*kombu.transport.SQS.Channel.QoS static method*), 199
 - `extract_task_name_and_number_of_retries()` (*kombu.transport.SQS.Transport.Channel.QoS static method*), 197
- ## F
- `failover_strategies` (*kombu.connection.Connection attribute*), 85
 - `failover_strategy` (*kombu.Connection attribute*), 41
 - `failover_strategy` (*kombu.connection.Connection attribute*), 85
 - `FailureCycle` (*class in kombu.utils.scheduling*), 228
 - `fanout_patterns` (*kombu.transport.redis.Channel attribute*), 179
 - `fanout_patterns` (*kombu.transport.redis.Transport.Channel attribute*), 176
 - `fanout_prefix` (*kombu.transport.redis.Channel attribute*), 179
 - `fanout_prefix` (*kombu.transport.redis.Transport.Channel attribute*), 176
 - `FanoutExchange` (*class in kombu.transport.virtual.exchange*), 105
 - `fetch()` (*kombu.compat.Consumer method*), 72
 - `fileno()` (*in module kombu.utils.compat*), 223
 - `fill_rate` (*kombu.utils.limits.TokenBucket attribute*), 228
 - `fire_timers()` (*kombu.asynchronous.Hub method*), 206
 - `fire_timers()` (*kombu.asynchronous.hub.Hub method*), 206
 - `flow()` (*kombu.compat.Consumer method*), 72
 - `flow()` (*kombu.compat.ConsumerSet method*), 75
 - `flow()` (*kombu.Consumer method*), 59
 - `flow()` (*kombu.transport.pyamqp.Connection.Channel method*), 127
 - `flow()` (*kombu.transport.virtual.Channel method*), 101
 - `fmatch_best()` (*in module kombu.utils.text*), 229
 - `fmatch_iter()` (*in module kombu.utils.text*), 229
 - `follow_redirects` (*kombu.asynchronous.aws.connection.AsyncHTTPSC connection attribute*), 218
 - `follow_redirects` (*kombu.asynchronous.http.base.Request attribute*), 214
 - `follow_redirects` (*kombu.asynchronous.http.Request attribute*), 211
 - `force_close_all()` (*kombu.connection.ChannelPool method*), 87

- `force_close_all()` (*kombu.connection.ConnectionPool* method), 87
- `force_close_all()` (*kombu.resource.Resource* method), 93
- `forward()` (*kombu.clocks.LamportClock* method), 67
- `frame_writer` (*kombu.transport.pyamqp.Connection* property), 135
- `from_dict()` (*kombu.Queue* class method), 55
- `from_transport_options` (*kombu.transport.mongodb.Channel* attribute), 184
- `from_transport_options` (*kombu.transport.mongodb.Transport.Channel* attribute), 183
- `from_transport_options` (*kombu.transport.redis.Channel* attribute), 179
- `from_transport_options` (*kombu.transport.redis.SentinelChannel* attribute), 181
- `from_transport_options` (*kombu.transport.redis.Transport.Channel* attribute), 177
- `from_utf8()` (in module *kombu.utils.encoding*), 224
- `fun` (*kombu.asynchronous.timer.Entry* attribute), 208
- `fun` (*kombu.asynchronous.timer.Timer.Entry* attribute), 208
- ## G
- `generate_sts_session_token()` (*kombu.transport.SQS.Channel* method), 199
- `generate_sts_session_token()` (*kombu.transport.SQS.Transport.Channel* method), 198
- `get()` (*kombu.Queue* method), 55
- `get()` (*kombu.simple.SimpleBuffer* method), 66
- `get()` (*kombu.simple.SimpleQueue* method), 66
- `get()` (*kombu.transport.qpid.Channel.QoS* method), 164
- `get()` (*kombu.transport.qpid.Connection.Channel.QoS* method), 155
- `get()` (*kombu.transport.qpid.Transport.Connection.Channel.QoS* method), 142
- `get()` (*kombu.transport.virtual.QoS* method), 103
- `get()` (*kombu.utils.scheduling.FairCycle* method), 228
- `get_all_queues()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection* method), 219
- `get_attributes()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* method), 222
- `get_bindings()` (*kombu.transport.pyamqp.Connection.Channel* method), 127
- `get_consumers()` (*kombu.mixins.ConsumerMixin* method), 64
- `get_dead_letter_source_queues()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection* method), 219
- `get_decoder()` (in module *kombu.compression*), 90
- `get_default_encoding_file()` (in module *kombu.utils.encoding*), 224
- `get_encoder()` (in module *kombu.compression*), 90
- `get_event_loop()` (in module *kombu.asynchronous*), 206
- `get_event_loop()` (in module *kombu.asynchronous.hub*), 207
- `get_heartbeat_interval()` (*kombu.connection.Connection* method), 85
- `get_heartbeat_interval()` (*kombu.transport.pyamqp.Transport* method), 113
- `get_http_connection()` (*kombu.asynchronous.aws.connection.AsyncConnection* method), 217
- `get_limit()` (in module *kombu.pools*), 92
- `get_logger()` (*kombu.log.LogMixin* method), 79
- `get_loglevel()` (in module *kombu.log*), 80
- `get_loglevel()` (*kombu.log.LogMixin* method), 79
- `get_manager()` (in module *kombu.utils.amq_manager*), 222
- `get_manager()` (*kombu.Connection* method), 45
- `get_manager()` (*kombu.connection.Connection* method), 85
- `get_manager()` (*kombu.transport.pyamqp.Transport* method), 113
- `get_messages()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* method), 222
- `get_now()` (*kombu.transport.mongodb.Channel* method), 184
- `get_now()` (*kombu.transport.mongodb.Transport.Channel* method), 183
- `get_nowait()` (*kombu.simple.SimpleBuffer* method), 66
- `get_nowait()` (*kombu.simple.SimpleQueue* method), 66
- `get_qpid_connection()` (*kombu.transport.qpid.Connection* method), 162
- `get_qpid_connection()` (*kombu.transport.qpid.Transport.Connection* method), 149
- `get_queue()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection* method), 220
- `get_queue()` (*kombu.pidbox.Mailbox* method), 78
- `get_queue_attributes()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection* method), 220
- `get_queue_url()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection* method), 220
- `get_reply_queue()` (*kombu.pidbox.Mailbox* method), 78

- [get_table\(\)](#) (*kombu.transport.mongodb.Channel method*), 184
[get_table\(\)](#) (*kombu.transport.mongodb.Transport.Channel method*), 183
[get_table\(\)](#) (*kombu.transport.redis.Channel method*), 180
[get_table\(\)](#) (*kombu.transport.redis.Transport.Channel method*), 177
[get_table\(\)](#) (*kombu.transport.virtual.Channel method*), 101
[get_timeout\(\)](#) (*kombu.asynchronous.aws.sqs.queue.AsyncQueue method*), 222
[get_transport_cls\(\)](#) (*in module kombu.transport*), 96
[get_transport_cls\(\)](#) (*kombu.Connection method*), 44
[get_transport_cls\(\)](#) (*kombu.connection.Connection method*), 85
[getrequest\(\)](#) (*kombu.asynchronous.aws.connection.AsyncHTTPConnection method*), 219
[getresponse\(\)](#) (*kombu.asynchronous.aws.connection.AsyncHTTPConnection method*), 219
[global_keyprefix](#) (*kombu.transport.redis.Channel attribute*), 180
[global_keyprefix](#) (*kombu.transport.redis.Transport.Channel attribute*), 177
[global_state](#) (*kombu.transport.filesystem.Transport attribute*), 192
[global_state](#) (*kombu.transport.memory.Transport attribute*), 173
[global_state](#) (*kombu.transport.pyro.Transport attribute*), 204
[grow\(\)](#) (*kombu.asynchronous.semaphore.LaxBoundedSemaphore method*), 208
- ## H
- [handle\(\)](#) (*kombu.pidbox.Node method*), 79
[handle_call\(\)](#) (*kombu.pidbox.Node method*), 79
[handle_cast\(\)](#) (*kombu.pidbox.Node method*), 79
[handle_error\(\)](#) (*kombu.asynchronous.timer.Timer method*), 209
[handle_message\(\)](#) (*kombu.pidbox.Node method*), 79
[handler\(\)](#) (*kombu.pidbox.Node method*), 79
[handlers](#) (*kombu.pidbox.Node attribute*), 78
[has_binding\(\)](#) (*kombu.transport.virtual.BrokerState method*), 104
[HashedSeq](#) (*class in kombu.utils.collections*), 222
[hashvalue](#) (*kombu.utils.collections.HashedSeq attribute*), 222
[Headers](#) (*class in kombu.asynchronous.http*), 210
[Headers](#) (*class in kombu.asynchronous.http.base*), 213
[headers](#) (*kombu.asynchronous.aws.connection.AsyncHTTPConnection attribute*), 218
[headers](#) (*kombu.asynchronous.aws.sqs.message.AsyncRawMessage attribute*), 220
[headers](#) (*kombu.asynchronous.aws.sqs.message.BaseAsyncMessage attribute*), 221
[headers](#) (*kombu.asynchronous.http.base.Request attribute*), 214
[headers](#) (*kombu.asynchronous.http.base.Response attribute*), 215, 216
[headers](#) (*kombu.asynchronous.http.Request attribute*), 211
[headers](#) (*kombu.asynchronous.http.Response attribute*), 212, 213
[headers](#) (*kombu.message.Message attribute*), 89
[headers](#) (*kombu.transport.base.Message attribute*), 97
[headers](#) (*kombu.transport.pyamqp.Channel.Message attribute*), 136
[headers](#) (*kombu.transport.pyamqp.Connection.Channel.Message attribute*), 115
[headers](#) (*kombu.transport.pyamqp.Message attribute*), 137
[headers](#) (*kombu.transport.pyamqp.Transport.Connection.Channel.Message attribute*), 113
[headers](#) (*kombu.transport.qpid.Channel.Message attribute*), 164
[headers](#) (*kombu.transport.qpid.Connection.Channel.Message attribute*), 154
[headers](#) (*kombu.transport.qpid.Message attribute*), 172
[headers](#) (*kombu.transport.qpid.Transport.Connection.Channel.Message attribute*), 141
[headers](#) (*kombu.transport.virtual.Message attribute*), 102
[health_check_interval](#) (*kombu.transport.redis.Channel attribute*), 180
[health_check_interval](#) (*kombu.transport.redis.Transport.Channel attribute*), 177
[heartbeat](#) (*kombu.Connection attribute*), 41
[heartbeat](#) (*kombu.connection.Connection attribute*), 85
[heartbeat](#) (*kombu.transport.pyamqp.Connection attribute*), 135
[heartbeat_check\(\)](#) (*kombu.Connection method*), 44
[heartbeat_check\(\)](#) (*kombu.connection.Connection method*), 85
[heartbeat_check\(\)](#) (*kombu.transport.pyamqp.Transport method*), 113
[heartbeat_tick\(\)](#) (*kombu.transport.pyamqp.Connection method*), 135
[host](#) (*kombu.Connection attribute*), 42
[host](#) (*kombu.connection.Connection property*), 86
[hostname](#) (*kombu.Connection attribute*), 41
[hostname](#) (*kombu.connection.Connection attribute*), 86
[hostname](#) (*kombu.pidbox.Node attribute*), 78
[hostname](#) (*kombu.utils.url.urlparts property*), 230

Hub (class in kombu.asynchronous), 205
 Hub (class in kombu.asynchronous.hub), 206

I

id (kombu.clocks.timetuple property), 68
 id (kombu.transport.sqlalchemy.models.Message attribute), 194
 id (kombu.transport.sqlalchemy.models.Queue attribute), 194
 implements (kombu.transport.etcd.Transport attribute), 187
 implements (kombu.transport.memory.Transport attribute), 173
 implements (kombu.transport.mongodb.Transport attribute), 184
 implements (kombu.transport.pyamqp.Transport attribute), 114
 implements (kombu.transport.qpid.Transport attribute), 151
 implements (kombu.transport.redis.Transport attribute), 178
 implements (kombu.transport.SQS.Transport attribute), 198
 incr() (kombu.utils.functional.LRUCache method), 224
 index (kombu.transport.consul.Channel attribute), 186
 index (kombu.transport.consul.Transport.Channel attribute), 186
 index (kombu.transport.etcd.Channel attribute), 188
 index (kombu.transport.etcd.Transport.Channel attribute), 187
 info() (kombu.Connection method), 44
 info() (kombu.connection.Connection method), 86
 info() (kombu.log.LogMixin method), 79
 insured() (in module kombu.common), 61
 is_alive() (kombu.transport.pyamqp.Connection method), 135
 is_bound (kombu.abstract.MaybeChannelBound property), 93
 is_enabled_for() (kombu.log.LogMixin method), 79
 is_evented (kombu.Connection attribute), 42
 is_evented (kombu.connection.Connection property), 86
 is_list() (in module kombu.utils.functional), 225
 is_secure (kombu.transport.SQS.Channel attribute), 199
 is_secure (kombu.transport.SQS.Transport.Channel attribute), 198
 items() (kombu.utils.functional.LRUCache method), 224
 iterconsume() (kombu.compat.Consumer method), 72
 iterconsume() (kombu.compat.ConsumerSet method), 75
 iteritems() (kombu.utils.functional.LRUCache method), 224

iterkeys() (kombu.utils.functional.LRUCache method), 224
 itermessages() (in module kombu.common), 61
 iterqueue() (kombu.compat.Consumer method), 72
 itervalues() (kombu.utils.functional.LRUCache method), 224

J

JSONEncoder (class in kombu.utils.json), 226

K

key_to_pattern() (kombu.transport.virtual.exchange.TopicExchange method), 105
 keyprefix_fanout (kombu.transport.redis.Channel attribute), 180
 keyprefix_fanout (kombu.transport.redis.Transport.Channel attribute), 177
 keyprefix_queue (kombu.transport.redis.Channel attribute), 180
 keyprefix_queue (kombu.transport.redis.Transport.Channel attribute), 177
 keys() (kombu.utils.functional.LRUCache method), 224
 kombu
 module, 39
 kombu.abstract
 module, 92
 kombu.asynchronous
 module, 205
 kombu.asynchronous.aws
 module, 216
 kombu.asynchronous.aws.connection
 module, 217
 kombu.asynchronous.aws.sqs
 module, 219
 kombu.asynchronous.aws.sqs.connection
 module, 219
 kombu.asynchronous.aws.sqs.message
 module, 220
 kombu.asynchronous.aws.sqs.queue
 module, 221
 kombu.asynchronous.debug
 module, 209
 kombu.asynchronous.http
 module, 210
 kombu.asynchronous.http.base
 module, 213
 kombu.asynchronous.http.curl
 module, 216
 kombu.asynchronous.hub
 module, 206
 kombu.asynchronous.semaphore
 module, 207
 kombu.asynchronous.timer
 module, 208

- `kombu.clocks`
 - module, [67](#)
- `kombu.common`
 - module, [60](#)
- `kombu.compat`
 - module, [68](#)
- `kombu.compression`
 - module, [89](#)
- `kombu.connection`
 - module, [80](#)
- `kombu.exceptions`
 - module, [79](#)
- `kombu.log`
 - module, [79](#)
- `kombu.matcher`
 - module, [62](#)
- `kombu.message`
 - module, [88](#)
- `kombu.mixins`
 - module, [63](#)
- `kombu.pidbox`
 - module, [77](#)
- `kombu.pools`
 - module, [90](#)
- `kombu.resource`
 - module, [93](#)
- `kombu.serialization`
 - module, [94](#)
- `kombu.simple`
 - module, [65](#)
- `kombu.transport`
 - module, [96](#)
- `kombu.transport.azure_servicebus`
 - module, [108](#)
- `kombu.transport.azurestoragequeues`
 - module, [106](#)
- `kombu.transport.base`
 - module, [96](#)
- `kombu.transport.consul`
 - module, [185](#)
- `kombu.transport.etcd`
 - module, [186](#)
- `kombu.transport.filesystem`
 - module, [190](#)
- `kombu.transport.memory`
 - module, [172](#)
- `kombu.transport.mongodb`
 - module, [182](#)
- `kombu.transport.pyamqp`
 - module, [111](#)
- `kombu.transport.pyro`
 - module, [203](#)
- `kombu.transport.qpid`
 - module, [137](#)
- `kombu.transport.redis`
 - module, [174](#)
- `kombu.transport.SLMQ`
 - module, [201](#)
- `kombu.transport.sqlalchemy`
 - module, [192](#)
- `kombu.transport.sqlalchemy.models`
 - module, [194](#)
- `kombu.transport.SQS`
 - module, [195](#)
- `kombu.transport.virtual`
 - module, [99](#)
- `kombu.transport.virtual.exchange`
 - module, [104](#)
- `kombu.transport.zookeeper`
 - module, [188](#)
- `kombu.utils.amq_manager`
 - module, [222](#)
- `kombu.utils.collections`
 - module, [222](#)
- `kombu.utils.compat`
 - module, [223](#)
- `kombu.utils.debug`
 - module, [223](#)
- `kombu.utils.div`
 - module, [223](#)
- `kombu.utils.encoding`
 - module, [223](#)
- `kombu.utils.eventio`
 - module, [224](#)
- `kombu.utils.functional`
 - module, [224](#)
- `kombu.utils.imports`
 - module, [226](#)
- `kombu.utils.json`
 - module, [226](#)
- `kombu.utils.limits`
 - module, [227](#)
- `kombu.utils.objects`
 - module, [228](#)
- `kombu.utils.scheduling`
 - module, [228](#)
- `kombu.utils.text`
 - module, [229](#)
- `kombu.utils.time`
 - module, [229](#)
- `kombu.utils.url`
 - module, [229](#)
- `kombu.utils.uuid`
 - module, [230](#)
- `KOMBU_LOG_CHANNEL`, [300](#)
- `KOMBU_LOG_CONNECTION`, [299](#)
- `KOMBU_LOG_DEBUG`, [299](#), [300](#)
- `kwargs` (*kombu.asynchronous.timer.Entry* attribute), [208](#)

kwargs (*kombu.asynchronous.timer.Timer.Entry attribute*), 208

L

LamportClock (*class in kombu.clocks*), 67

last_heartbeat_received (*kombu.transport.pyamqp.Connection attribute*), 135

last_heartbeat_sent (*kombu.transport.pyamqp.Connection attribute*), 135

LaxBoundedSemaphore (*class in kombu.asynchronous.semaphore*), 207

lazy (*class in kombu.utils.functional*), 225

library_properties (*kombu.transport.pyamqp.Connection attribute*), 135

LifoQueue (*class in kombu.resource*), 93

limit (*kombu.resource.Resource property*), 93

LimitExceeded, 79

LimitExceeded (*kombu.connection.ChannelPool attribute*), 87

LimitExceeded (*kombu.connection.ConnectionPool attribute*), 87

list_first() (*in module kombu.asynchronous.aws.sqs.queue*), 222

listen() (*kombu.pidbox.Node method*), 79

load() (*kombu.asynchronous.aws.sqs.queue.AsyncQueue method*), 222

load_from_file() (*kombu.asynchronous.aws.sqs.queue.AsyncQueue method*), 222

load_from_filename() (*kombu.asynchronous.aws.sqs.queue.AsyncQueue method*), 222

load_from_s3() (*kombu.asynchronous.aws.sqs.queue.AsyncQueue method*), 222

loads() (*in module kombu.serialization*), 94

loads() (*in module kombu.utils.json*), 227

lock_name (*kombu.transport.consul.Channel attribute*), 186

lock_name (*kombu.transport.consul.Transport.Channel attribute*), 186

lock_ttl (*kombu.transport.etcd.Channel attribute*), 188

lock_ttl (*kombu.transport.etcd.Transport.Channel attribute*), 187

lock_value (*kombu.transport.etcd.Channel attribute*), 188

lock_value (*kombu.transport.etcd.Transport.Channel attribute*), 187

log() (*kombu.log.LogMixin method*), 79

logger (*kombu.log.LogMixin attribute*), 80

logger_name (*kombu.log.LogMixin property*), 80

login_method (*kombu.Connection attribute*), 41

login_method (*kombu.connection.Connection attribute*), 86

LogMixin (*class in kombu.log*), 79

Logwrapped (*class in kombu.utils.debug*), 223

lookup() (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method*), 220

lookup() (*kombu.transport.virtual.exchange.DirectExchange method*), 104

lookup() (*kombu.transport.virtual.exchange.ExchangeType method*), 106

lookup() (*kombu.transport.virtual.exchange.FanoutExchange method*), 105

lookup() (*kombu.transport.virtual.exchange.TopicExchange method*), 105

loop (*kombu.asynchronous.Hub property*), 205

loop (*kombu.asynchronous.hub.Hub property*), 206

LRUCache (*class in kombu.utils.functional*), 224

M

Mailbox (*class in kombu.pidbox*), 78

mailbox (*kombu.pidbox.Node attribute*), 78

manager (*kombu.Connection attribute*), 42

manager (*kombu.connection.Connection attribute*), 86

match() (*in module kombu.matcher*), 62

match() (*kombu.matcher.MatcherRegistry method*), 62

matcher_pattern_first (*kombu.matcher.MatcherRegistry attribute*), 62

MatcherNotInstalled, 62

MatcherRegistry (*class in kombu.matcher*), 62

MatcherRegistry.MatcherNotInstalled, 62

max_connections (*kombu.transport.redis.Channel attribute*), 180

max_connections (*kombu.transport.redis.Transport.Channel attribute*), 177

max_length (*kombu.Queue attribute*), 52

max_length_bytes (*kombu.Queue attribute*), 53

max_priority (*kombu.Queue attribute*), 53

max_redirects (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection attribute*), 218

max_redirects (*kombu.asynchronous.http.base.Request attribute*), 214

max_redirects (*kombu.asynchronous.http.Request attribute*), 211

maybe_bind() (*kombu.abstract.MaybeChannelBound method*), 93

maybe_bind() (*kombu.Exchange method*), 49

maybe_bind() (*kombu.Queue method*), 53

maybe_close_channel() (*kombu.Connection method*), 44

maybe_close_channel() (*kombu.connection.Connection method*), 86

maybe_conn_error() (*kombu.mixins.ConsumerMixin method*), 64

maybe_declare() (*in module kombu.common*), 61

- `maybe_declare()` (*kombu.compat.Publisher method*), 69
- `maybe_declare()` (*kombu.pools.ProducerPool.Producer method*), 91
- `maybe_declare()` (*kombu.Producer method*), 56
- `maybe_evaluate()` (*in module kombu.utils.functional*), 225
- `maybe_fileno()` (*in module kombu.utils.compat*), 223
- `maybe_list()` (*in module kombu.utils.functional*), 225
- `maybe_s_to_ms()` (*in module kombu.utils.time*), 229
- `maybe_sanitize_url()` (*in module kombu.utils.url*), 229
- `maybe_switch_next()` (*kombu.Connection method*), 44
- `maybe_switch_next()` (*kombu.connection.Connection method*), 86
- `MaybeChannelBound` (*class in kombu.abstract*), 92
- `memoize()` (*in module kombu.utils.functional*), 225
- `Message` (*class in kombu.message*), 88
- `Message` (*class in kombu.transport.base*), 97
- `Message` (*class in kombu.transport.pyamqp*), 137
- `Message` (*class in kombu.transport.qpid*), 172
- `Message` (*class in kombu.transport.sqlalchemy.models*), 194
- `Message` (*class in kombu.transport.virtual*), 101
- `Message` (*kombu.transport.virtual.Channel attribute*), 99
- `Message()` (*kombu.Exchange method*), 49
- `Message.MessageStateError`, 88, 101
- `message_cls` (*kombu.transport.sqlalchemy.Channel attribute*), 194
- `message_cls` (*kombu.transport.sqlalchemy.Transport.Channel attribute*), 193
- `message_to_python()` (*kombu.transport.pyamqp.Channel method*), 136
- `message_to_python()` (*kombu.transport.pyamqp.Connection.Channel method*), 127
- `message_to_python()` (*kombu.transport.pyamqp.Transport.Connection.Channel method*), 113
- `message_to_python()` (*kombu.transport.virtual.Channel method*), 101
- `message_ttl` (*kombu.Queue attribute*), 52
- `messages` (*kombu.transport.mongodb.Channel attribute*), 184
- `messages` (*kombu.transport.mongodb.Transport.Channel attribute*), 183
- `messages_collection` (*kombu.transport.mongodb.Channel attribute*), 184
- `messages_collection` (*kombu.transport.mongodb.Transport.Channel attribute*), 183
- `MessageStateError`, 79
- `method` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection attribute*), 219
- `method` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection.Request attribute*), 218
- `method` (*kombu.asynchronous.http.base.Request attribute*), 214
- `method` (*kombu.asynchronous.http.Request attribute*), 211
- `module`
 - `kombu`, 39
 - `kombu.abstract`, 92
 - `kombu.asynchronous`, 205
 - `kombu.asynchronous.aws`, 216
 - `kombu.asynchronous.aws.connection`, 217
 - `kombu.asynchronous.aws.sqs`, 219
 - `kombu.asynchronous.aws.sqs.connection`, 219
 - `kombu.asynchronous.aws.sqs.message`, 220
 - `kombu.asynchronous.aws.sqs.queue`, 221
 - `kombu.asynchronous.debug`, 209
 - `kombu.asynchronous.http`, 210
 - `kombu.asynchronous.http.base`, 213
 - `kombu.asynchronous.http.curl`, 216
 - `kombu.asynchronous.hub`, 206
 - `kombu.asynchronous.semaphore`, 207
 - `kombu.asynchronous.timer`, 208
 - `kombu.clocks`, 67
 - `kombu.common`, 60
 - `kombu.compat`, 68
 - `kombu.compression`, 89
 - `kombu.connection`, 80
 - `kombu.exceptions`, 79
 - `kombu.log`, 79
 - `kombu.matcher`, 62
 - `kombu.message`, 88
 - `kombu.mixins`, 63
 - `kombu.pidbox`, 77
 - `kombu.pools`, 90
 - `kombu.resource`, 93
 - `kombu.serialization`, 94
 - `kombu.simple`, 65
 - `kombu.transport`, 96
 - `kombu.transport.azure_servicebus`, 108
 - `kombu.transport.azurestoragequeues`, 106
 - `kombu.transport.base`, 96
 - `kombu.transport.consul`, 185
 - `kombu.transport.etcd`, 186
 - `kombu.transport.filesystem`, 190
 - `kombu.transport.memory`, 172
 - `kombu.transport.mongodb`, 182
 - `kombu.transport.pyamqp`, 111
 - `kombu.transport.pyro`, 203
 - `kombu.transport.qpid`, 137

- `kombu.transport.redis`, 174
- `kombu.transport.SLMQ`, 201
- `kombu.transport.sqlalchemy`, 192
- `kombu.transport.sqlalchemy.models`, 194
- `kombu.transport.SQS`, 195
- `kombu.transport.virtual`, 99
- `kombu.transport.virtual.exchange`, 104
- `kombu.transport.zookeeper`, 188
- `kombu.utils.amq_manager`, 222
- `kombu.utils.collections`, 222
- `kombu.utils.compat`, 223
- `kombu.utils.debug`, 223
- `kombu.utils.div`, 223
- `kombu.utils.encoding`, 223
- `kombu.utils.eventio`, 224
- `kombu.utils.functional`, 224
- `kombu.utils.imports`, 226
- `kombu.utils.json`, 226
- `kombu.utils.limits`, 227
- `kombu.utils.objects`, 228
- `kombu.utils.scheduling`, 228
- `kombu.utils.text`, 229
- `kombu.utils.time`, 229
- `kombu.utils.url`, 229
- `kombu.utils.uuid`, 230
- `multi_call()` (*kombu.pidbox.Mailbox* method), 78

N

- `name` (*kombu.Exchange* attribute), 47, 50
- `name` (*kombu.Queue* attribute), 51, 55
- `name` (*kombu.transport.sqlalchemy.models.Queue* attribute), 194
- `namespace` (*kombu.pidbox.Mailbox* attribute), 78
- `negotiate_capabilities`
 - (*kombu.transport.pyamqp.Connection* attribute), 135
- `nested()` (in module *kombu.utils.compat*), 223
- `network_interface` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection* attribute), 218
- `network_interface` (*kombu.asynchronous.http.base.Request* attribute), 214
- `network_interface` (*kombu.asynchronous.http.Request* attribute), 211
- `new()` (*kombu.pools.ProducerPool* method), 92
- `new_sqs_client()` (*kombu.transport.SQS.Channel* method), 199
- `new_sqs_client()` (*kombu.transport.SQS.Transport.Channel* method), 198
- `no_ack` (*kombu.compat.Consumer* attribute), 72
- `no_ack` (*kombu.compat.ConsumerSet* attribute), 75
- `no_ack` (*kombu.Consumer* attribute), 58
- `no_ack` (*kombu.Queue* attribute), 55
- `no_ack` (*kombu.simple.SimpleBuffer* attribute), 66
- `no_ack` (*kombu.simple.SimpleQueue* attribute), 66

- `no_ack` (*kombu.transport.azurestoragequeues.Channel* attribute), 108
- `no_ack` (*kombu.transport.azurestoragequeues.Transport.Channel* attribute), 107
- `no_ack_consumers` (*kombu.transport.pyamqp.Connection.Channel* attribute), 127
- `no_declare` (*kombu.Exchange* attribute), 48, 50
- `no_declare` (*kombu.Queue* attribute), 53
- `Node` (class in *kombu.pidbox*), 78
- `Node()` (*kombu.pidbox.Mailbox* method), 78
- `NotBoundError`, 79

O

- `obj` (*kombu.clocks.timetuple* property), 68
- `on_callback_error()` (*kombu.asynchronous.Hub* method), 205
- `on_callback_error()` (*kombu.asynchronous.hub.Hub* method), 206
- `on_close` (*kombu.asynchronous.Hub* attribute), 205
- `on_close` (*kombu.asynchronous.hub.Hub* attribute), 206
- `on_connection_error()`
 - (*kombu.mixins.ConsumerMixin* method), 64
- `on_connection_revived()`
 - (*kombu.mixins.ConsumerMixin* method), 64
- `on_consume_end()` (*kombu.mixins.ConsumerMixin* method), 64
- `on_consume_end()` (*kombu.mixins.ConsumerProducerMixin* method), 65
- `on_consume_ready()` (*kombu.mixins.ConsumerMixin* method), 64
- `on_declared` (*kombu.Queue* attribute), 53
- `on_decode_error` (*kombu.compat.Consumer* attribute), 72
- `on_decode_error` (*kombu.compat.ConsumerSet* attribute), 75
- `on_decode_error` (*kombu.transport.pyamqp.Connection.Consumer* attribute), 58
- `on_decode_error()` (*kombu.mixins.ConsumerMixin* method), 64
- `on_error` (*kombu.asynchronous.timer.Timer* attribute), 209
- `on_header` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection* attribute), 218
- `on_header` (*kombu.asynchronous.http.base.Request* attribute), 214
- `on_header` (*kombu.asynchronous.http.Request* attribute), 211
- `on_inbound_frame` (*kombu.transport.pyamqp.Connection* property), 136
- `on_inbound_method()`
 - (*kombu.transport.pyamqp.Connection* method), 136

- on_iteration() (*kombu.mixins.ConsumerMixin* method), 64
 on_message (*kombu.compat.Consumer* attribute), 72
 on_message (*kombu.compat.ConsumerSet* attribute), 75
 on_message (*kombu.Consumer* attribute), 58
 on_prepare (*kombu.asynchronous.aws.connection.AsyncHTTPSSConnection* attribute), 218
 on_prepare (*kombu.asynchronous.http.base.Request* attribute), 214
 on_prepare (*kombu.asynchronous.http.Request* attribute), 211
 on_readable() (*kombu.asynchronous.http.curl.CurlClient* method), 216
 on_readable() (*kombu.transport.qpid.Transport* method), 151
 on_readable() (*kombu.transport.redis.Transport* method), 178
 on_ready (*kombu.asynchronous.aws.connection.AsyncHTTPSSConnection* attribute), 218
 on_ready (*kombu.asynchronous.http.base.Request* attribute), 214
 on_ready (*kombu.asynchronous.http.Request* attribute), 211
 on_return (*kombu.compat.Publisher* attribute), 69
 on_return (*kombu.pools.ProducerPool.Producer* attribute), 91
 on_return (*kombu.Producer* attribute), 56
 on_stream (*kombu.asynchronous.aws.connection.AsyncHTTPSSConnection* attribute), 218
 on_stream (*kombu.asynchronous.http.base.Request* attribute), 214
 on_stream (*kombu.asynchronous.http.Request* attribute), 211
 on_timeout (*kombu.asynchronous.aws.connection.AsyncHTTPSSConnection* attribute), 218
 on_timeout (*kombu.asynchronous.http.base.Request* attribute), 214
 on_timeout (*kombu.asynchronous.http.Request* attribute), 211
 on_writable() (*kombu.asynchronous.http.curl.CurlClient* method), 216
 open() (*kombu.transport.pyamqp.Connection.Channel* method), 127
- ## P
- parse_ssl_cert_reqs() (in module *kombu.utils.url*), 229
 parse_uri() (*kombu.transport.azureservicebus.Transport* static method), 110
 parse_url() (in module *kombu.utils.url*), 229
 passive (*kombu.Exchange* attribute), 50
 password (*kombu.Connection* attribute), 41
 password (*kombu.connection.Connection* attribute), 86
 password (*kombu.utils.url.urlparts* property), 230
 path (*kombu.asynchronous.aws.connection.AsyncHTTPSSConnection* attribute), 219
 path (*kombu.utils.url.urlparts* property), 230
 payload (*kombu.message.Message* property), 89
 payload (*kombu.transport.base.Message* attribute), 97
 payload (*kombu.transport.pyamqp.Connection.Channel.Message* property), 115
 payload (*kombu.transport.sqlalchemy.models.Message* attribute), 194
 payload (*kombu.transport.virtual.Message* property), 102
 peek_lock_seconds (*kombu.transport.azureservicebus.Channel* attribute), 110
 peek_lock_seconds (*kombu.transport.azureservicebus.Transport.Channel* attribute), 109
 PERSISTENT_DELIVERY_MODE (*kombu.Exchange* attribute), 49
 PICKLE_PROTOCOL (*kombu.Transport* attribute), 31, 278
 pipe_or_acquire() (*kombu.transport.redis.Channel.QoS* method), 178
 pipe_or_acquire() (*kombu.transport.redis.Transport.Channel.QoS* method), 175
 poll() (in module *kombu.utils.eventio*), 224
 poller (*kombu.asynchronous.Hub* property), 205
 poller (*kombu.asynchronous.hub.Hub* property), 206
 polling_interval (*kombu.transport.azureservicebus.Transport* attribute), 110
 polling_interval (*kombu.transport.azurestoragequeues.Transport* attribute), 107
 polling_interval (*kombu.transport.etcd.Transport* attribute), 188
 polling_interval (*kombu.transport.mongodb.Transport* attribute), 184
 polling_interval (*kombu.transport.pyamqp.Connection.Channel* attribute), 151
 polling_interval (*kombu.transport.qpid.Transport* attribute), 151
 polling_interval (*kombu.transport.redis.Transport* attribute), 178
 polling_interval (*kombu.transport.SLMQ.Transport* attribute), 202
 polling_interval (*kombu.transport.SQS.Transport* attribute), 198
 polling_interval (*kombu.transport.virtual.Transport* attribute), 99
 polling_interval (*kombu.transport.zookeeper.Transport* attribute), 189
 pool (*kombu.transport.redis.Channel* property), 180
 pool (*kombu.transport.redis.Transport.Channel* property), 177
 Pool() (*kombu.Connection* method), 45
 Pool() (*kombu.connection.Connection* method), 82
 PoolGroup (class in *kombu.pools*), 90
 pop() (*kombu.utils.limits.TokenBucket* method), 228
 popitem() (*kombu.utils.functional.LRUCache* method), 224

- `port` (*kombu.Connection* attribute), 41
- `port` (*kombu.connection.Connection* attribute), 86
- `port` (*kombu.transport.SQS.Channel* attribute), 199
- `port` (*kombu.transport.SQS.Transport.Channel* attribute), 198
- `port` (*kombu.utils.url.urlparts* property), 230
- `predefined_queues` (*kombu.transport.SQS.Channel* attribute), 200
- `predefined_queues` (*kombu.transport.SQS.Transport.Channel* attribute), 198
- `prefetch_count` (*kombu.compat.Consumer* attribute), 72
- `prefetch_count` (*kombu.compat.ConsumerSet* attribute), 75
- `prefetch_count` (*kombu.transport.virtual.QoS* attribute), 103
- `prefix` (*kombu.transport.consul.Channel* attribute), 186
- `prefix` (*kombu.transport.consul.Transport.Channel* attribute), 186
- `prefix` (*kombu.transport.etcd.Channel* attribute), 188
- `prefix` (*kombu.transport.etcd.Transport.Channel* attribute), 187
- `prepare()` (*kombu.pools.ProducerPool* method), 92
- `prepare()` (*kombu.resource.Resource* method), 93
- `prepare_bind()` (*kombu.transport.virtual.exchange.Exchange* method), 106
- `prepare_bind()` (*kombu.transport.virtual.exchange.TopicExchange* method), 105
- `prepare_message()` (*kombu.transport.pyamqp.Channel* method), 136
- `prepare_message()` (*kombu.transport.pyamqp.Connection.Channel* method), 128
- `prepare_message()` (*kombu.transport.pyamqp.Transport.Connection.Channel* method), 113
- `prepare_message()` (*kombu.transport.qpid.Channel* method), 168
- `prepare_message()` (*kombu.transport.qpid.Connection.Channel* method), 159
- `prepare_message()` (*kombu.transport.qpid.Transport.Connection.Channel* method), 146
- `prepare_message()` (*kombu.transport.virtual.Channel* method), 101
- `prepare_queue_arguments()` (*kombu.transport.mongodb.Channel* method), 184
- `prepare_queue_arguments()` (*kombu.transport.mongodb.Transport.Channel* method), 183
- `prepare_queue_arguments()` (*kombu.transport.pyamqp.Channel* method), 137
- `prepare_queue_arguments()` (*kombu.transport.pyamqp.Connection.Channel* method), 128
- `prepare_queue_arguments()` (*kombu.transport.pyamqp.Transport.Connection.Channel* method), 113
- `prepare_queue_arguments()` (*kombu.transport.qpid.Channel.Message* attribute), 164
- `prepare_queue_arguments()` (*kombu.transport.pyamqp.Transport.Connection.Channel* method), 113
- `prev_recv` (*kombu.transport.pyamqp.Connection* attribute), 136
- `prev_sent` (*kombu.transport.pyamqp.Connection* attribute), 136
- `priority()` (*kombu.transport.redis.Channel* method), 180
- `priority()` (*kombu.transport.redis.Transport.Channel* method), 177
- `priority_cycle` (class in *kombu.utils.scheduling*), 228
- `priority_steps` (*kombu.transport.redis.Channel* attribute), 180
- `priority_steps` (*kombu.transport.redis.Transport.Channel* attribute), 177
- `process_next()` (*kombu.compat.Consumer* method), 72
- `processed_folder` (*kombu.transport.filesystem.Channel* attribute), 192
- `processed_folder` (*kombu.transport.filesystem.Transport.Channel* attribute), 191
- `Producer` (class in *kombu*), 56
- `producer` (*kombu.mixins.ConsumerProducerMixin* property), 65
- `producer` (*kombu.simple.SimpleBuffer* attribute), 66
- `producer` (*kombu.simple.SimpleQueue* attribute), 65
- `Producer()` (*kombu.Connection* method), 45
- `Producer()` (*kombu.connection.Connection* method), 82
- `Producer()` (*kombu.transport.pyamqp.Connection.Channel* method), 115
- `producer_connection` (*kombu.mixins.ConsumerProducerMixin* property), 65
- `ProducerPool` (class in *kombu.pools*), 90
- `ProducerPool.Producer` (class in *kombu.pools*), 90
- `properties` (*kombu.asynchronous.aws.sqs.message.AsyncMessage* attribute), 220
- `properties` (*kombu.asynchronous.aws.sqs.message.AsyncRawMessage* attribute), 221
- `properties` (*kombu.asynchronous.aws.sqs.message.BaseAsyncMessage* attribute), 221
- `properties` (*kombu.message.Message* attribute), 89
- `properties` (*kombu.transport.base.Message* attribute), 97
- `properties` (*kombu.transport.pyamqp.Channel.Message* attribute), 136
- `properties` (*kombu.transport.pyamqp.Connection.Channel.Message* attribute), 115
- `properties` (*kombu.transport.pyamqp.Message* attribute), 137
- `properties` (*kombu.transport.pyamqp.Transport.Connection.Channel.Message* attribute), 113

- properties (kombu.transport.qpid.Connection.Channel.Message attribute), 154
- properties (kombu.transport.qpid.Message attribute), 172
- properties (kombu.transport.qpid.Transport.Connection.Channel attribute), 141
- properties (kombu.transport.virtual.Message attribute), 102
- proxy_host (kombu.asynchronous.aws.connection.AsyncHTTPConnection.Request attribute), 218
- proxy_host (kombu.asynchronous.http.base.Request attribute), 214
- proxy_host (kombu.asynchronous.http.Request attribute), 211
- proxy_password (kombu.asynchronous.aws.connection.AsyncHTTPConnection.Request attribute), 218
- proxy_password (kombu.asynchronous.http.base.Request attribute), 214
- proxy_password (kombu.asynchronous.http.Request attribute), 211
- proxy_port (kombu.asynchronous.aws.connection.AsyncHTTPConnection.Request attribute), 218
- proxy_port (kombu.asynchronous.http.base.Request attribute), 215
- proxy_port (kombu.asynchronous.http.Request attribute), 211
- proxy_username (kombu.asynchronous.aws.connection.AsyncHTTPConnection.Request attribute), 218
- proxy_username (kombu.asynchronous.http.base.Request attribute), 215
- proxy_username (kombu.asynchronous.http.Request attribute), 211
- publish() (kombu.compat.Publisher method), 69
- publish() (kombu.Exchange method), 50
- publish() (kombu.pools.ProducerPool.Producer method), 91
- publish() (kombu.Producer method), 56
- Publisher (class in kombu.compat), 69
- purge() (kombu.compat.Consumer method), 72
- purge() (kombu.compat.ConsumerSet method), 76
- purge() (kombu.Consumer method), 59
- purge() (kombu.Queue method), 55
- put() (kombu.simple.SimpleBuffer method), 66
- put() (kombu.simple.SimpleQueue method), 66
- putheader() (kombu.asynchronous.aws.connection.AsyncHTTPConnection.Request method), 219
- putrequest() (kombu.asynchronous.aws.connection.AsyncHTTPConnection.Request method), 219
- QoS (class in kombu.transport.virtual), 102
- qos (kombu.transport.qpid.Channel property), 169
- qos (kombu.transport.qpid.Connection.Channel property), 160
- qos (kombu.transport.qpid.Transport.Connection.Channel property), 147
- qos (kombu.transport.virtual.Channel attribute), 100
- qos() (kombu.compat.Consumer method), 72
- qos() (kombu.compat.ConsumerSet method), 76
- qos() (kombu.Consumer method), 59
- qos_semantics_matches_spec (kombu.connection.Connection property), 86
- qos_semantics_matches_spec() (kombu.transport.pyamqp.Transport method), 114
- qsize() (kombu.simple.SimpleBuffer method), 67
- qsize() (kombu.simple.SimpleQueue method), 66
- Queue (class in kombu), 50
- Queue (class in kombu.transport.sqlalchemy.models), 194
- queue (kombu.asynchronous.timer.Timer property), 209
- queue (kombu.compat.Consumer attribute), 73
- queue (kombu.simple.SimpleBuffer attribute), 66
- queue (kombu.simple.SimpleQueue attribute), 66
- Queue.ContentDisallowed, 54
- queue_arguments (kombu.Queue attribute), 53
- queue_bind() (kombu.Queue method), 55
- queue_bind() (kombu.transport.pyamqp.Connection.Channel method), 128
- queue_bind() (kombu.transport.qpid.Connection.Channel method), 169
- queue_bind() (kombu.transport.qpid.Connection.Channel method), 160
- queue_bind() (kombu.transport.qpid.Transport.Connection.Channel method), 147
- queue_bind() (kombu.transport.virtual.Channel method), 100
- queue_bindings() (kombu.transport.virtual.BrokerState method), 104
- queue_bindings_delete() (kombu.transport.virtual.BrokerState method), 104
- queue_cls (kombu.transport.sqlalchemy.Channel attribute), 194
- queue_cls (kombu.transport.sqlalchemy.Transport.Channel attribute), 193
- queue_declare() (kombu.Queue method), 55
- queue_declare() (kombu.transport.pyamqp.Connection.Channel method), 129
- queue_declare() (kombu.transport.qpid.Channel method), 169
- queue_declare() (kombu.transport.qpid.Connection.Channel method), 160
- queue_declare() (kombu.transport.qpid.Transport.Connection.Channel method), 147
- queue_declare() (kombu.transport.virtual.Channel method), 100

Q

[queue_delete\(\) \(kombu.transport.mongodb.Channel method\), 185](#)
[queue_delete\(\) \(kombu.transport.mongodb.Transport.Channel method\), 183](#)
[queue_delete\(\) \(kombu.transport.pyamqp.Connection.Channel method\), 131](#)
[queue_delete\(\) \(kombu.transport.qpid.Channel method\), 170](#)
[queue_delete\(\) \(kombu.transport.qpid.Connection.Channel method\), 161](#)
[queue_delete\(\) \(kombu.transport.qpid.Transport.Connection.Channel method\), 148](#)
[queue_delete\(\) \(kombu.transport.virtual.Channel method\), 100](#)
[queue_index \(kombu.transport.virtual.BrokerState attribute\), 104](#)
[queue_mgmt_service \(kombu.transport.azureservicebus.Channel property\), 110](#)
[queue_mgmt_service \(kombu.transport.azureservicebus.Transport.Channel property\), 109](#)
[queue_name_prefix \(kombu.transport.azureservicebus.Channel attribute\), 110](#)
[queue_name_prefix \(kombu.transport.azureservicebus.Transport.Channel attribute\), 109](#)
[queue_name_prefix \(kombu.transport.azurestoragequeues.Channel attribute\), 108](#)
[queue_name_prefix \(kombu.transport.azurestoragequeues.Transport.Channel attribute\), 107](#)
[queue_name_prefix \(kombu.transport.SLMQ.Channel attribute\), 202](#)
[queue_name_prefix \(kombu.transport.SLMQ.Transport.Channel attribute\), 202](#)
[queue_name_prefix \(kombu.transport.SQS.Channel attribute\), 200](#)
[queue_name_prefix \(kombu.transport.SQS.Transport.Channel attribute\), 198](#)
[queue_opts \(kombu.simple.SimpleBuffer attribute\), 66](#)
[queue_opts \(kombu.simple.SimpleQueue attribute\), 66](#)
[queue_order_strategy \(kombu.transport.redis.Channel attribute\), 180](#)
[queue_order_strategy \(kombu.transport.redis.Transport.Channel attribute\), 177](#)
[queue_purge\(\) \(kombu.transport.pyamqp.Connection.Channel method\), 132](#)
[queue_purge\(\) \(kombu.transport.qpid.Channel method\), 170](#)
[queue_purge\(\) \(kombu.transport.qpid.Connection.Channel method\), 161](#)
[queue_purge\(\) \(kombu.transport.qpid.Transport.Connection.Channel method\), 148](#)
[queue_purge\(\) \(kombu.transport.virtual.Channel method\), 100](#)
[queue_service \(kombu.transport.azureservicebus.Channel property\), 110](#)
[queue_service \(kombu.transport.azureservicebus.Transport.Channel property\), 109](#)
[queue_service \(kombu.transport.azurestoragequeues.Channel property\), 108](#)
[queue_service \(kombu.transport.azurestoragequeues.Transport.Channel property\), 107](#)
[queue_unbind\(\) \(kombu.Queue method\), 55](#)
[queue_unbind\(\) \(kombu.transport.pyamqp.Connection.Channel method\), 133](#)
[queue_unbind\(\) \(kombu.transport.qpid.Channel method\), 171](#)
[queue_unbind\(\) \(kombu.transport.qpid.Connection.Channel method\), 162](#)
[queue_unbind\(\) \(kombu.transport.qpid.Transport.Connection.Channel method\), 149](#)
[queues \(kombu.compat.Consumer property\), 73](#)
[queues \(kombu.compat.ConsumerSet property\), 76](#)
[queues \(kombu.Consumer attribute\), 58](#)
[queues \(kombu.transport.memory.Channel attribute\), 174](#)
[queues \(kombu.transport.memory.Transport.Channel attribute\), 173](#)
[queues \(kombu.transport.mongodb.Channel attribute\), 185](#)
[queues \(kombu.transport.mongodb.Transport.Channel attribute\), 183](#)
[queues\(\) \(kombu.transport.pyro.Channel method\), 204](#)
[queues\(\) \(kombu.transport.pyro.Transport.Channel method\), 204](#)
[queues_collection \(kombu.transport.mongodb.Channel attribute\), 185](#)
[queues_collection \(kombu.transport.mongodb.Transport.Channel attribute\), 183](#)

R

[raise_for_error\(\) \(kombu.asynchronous.http.base.Response method\), 216](#)
[raise_for_error\(\) \(kombu.asynchronous.http.Response method\), 213](#)
[raw_encode\(\) \(in module kombu.serialization\), 95](#)
[READ \(kombu.asynchronous.Hub attribute\), 205](#)
[READ \(kombu.asynchronous.hub.Hub attribute\), 206](#)
[read\(\) \(kombu.asynchronous.aws.sqs.queue.AsyncQueue method\), 222](#)
[receive\(\) \(kombu.compat.Consumer method\), 73](#)
[receive\(\) \(kombu.compat.ConsumerSet method\), 76](#)
[receive\(\) \(kombu.Consumer method\), 60](#)
[receive_message\(\) \(kombu.asynchronous.aws.sqs.connection.AsyncSQS method\), 220](#)
[recover\(\) \(kombu.compat.Consumer method\), 73](#)
[recover\(\) \(kombu.compat.ConsumerSet method\), 76](#)
[recover\(\) \(kombu.Consumer method\), 59](#)

- `recoverable_channel_errors` (*kombu.Connection* attribute), 41
- `recoverable_channel_errors` (*kombu.connection.Connection* attribute), 86
- `recoverable_channel_errors` (*kombu.transport.base.Transport* attribute), 98
- `recoverable_channel_errors` (*kombu.transport.pyamqp.Connection* attribute), 136
- `recoverable_channel_errors` (*kombu.transport.pyamqp.Transport* attribute), 114
- `recoverable_channel_errors` (*kombu.transport.qpid.Transport* attribute), 151
- `recoverable_connection_errors` (*kombu.Connection* attribute), 41
- `recoverable_connection_errors` (*kombu.connection.Connection* attribute), 86
- `recoverable_connection_errors` (*kombu.transport.base.Transport* attribute), 98
- `recoverable_connection_errors` (*kombu.transport.pyamqp.Connection* attribute), 136
- `recoverable_connection_errors` (*kombu.transport.pyamqp.Transport* attribute), 114
- `recoverable_connection_errors` (*kombu.transport.qpid.Transport* attribute), 151
- `region` (*kombu.transport.SQS.Channel* attribute), 200
- `region` (*kombu.transport.SQS.Transport.Channel* attribute), 198
- `regioninfo` (*kombu.transport.SQS.Channel* attribute), 200
- `regioninfo` (*kombu.transport.SQS.Transport.Channel* attribute), 198
- `register()` (in module *kombu.compression*), 90
- `register()` (in module *kombu.matcher*), 62
- `register()` (in module *kombu.serialization*), 95
- `register()` (*kombu.matcher.MatcherRegistry* method), 62
- `register_callback()` (*kombu.compat.Consumer* method), 73
- `register_callback()` (*kombu.compat.ConsumerSet* method), 76
- `register_callback()` (*kombu.Consumer* method), 58
- `register_glob()` (in module *kombu.matcher*), 62
- `register_group()` (in module *kombu.pools*), 92
- `register_pcre()` (in module *kombu.matcher*), 62
- `register_with_event_loop()` (*kombu.Connection* method), 44
- `register_with_event_loop()` (*kombu.connection.Connection* method), 86
- `register_with_event_loop()` (*kombu.transport.pyamqp.Transport* method), 114
- `register_with_event_loop()` (*kombu.transport.qpid.Transport* method), 151
- `register_with_event_loop()` (*kombu.transport.redis.Transport* method), 178
- `registry` (in module *kombu.matcher*), 62
- `registry` (in module *kombu.serialization*), 95
- `reject()` (*kombu.message.Message* method), 89
- `reject()` (*kombu.transport.base.Message* method), 97
- `reject()` (*kombu.transport.pyamqp.Connection.Channel.Message* method), 115
- `reject()` (*kombu.transport.qpid.Channel.QoS* method), 165
- `reject()` (*kombu.transport.qpid.Connection.Channel.QoS* method), 155
- `reject()` (*kombu.transport.qpid.Transport.Connection.Channel.QoS* method), 142
- `reject()` (*kombu.transport.redis.Channel.QoS* method), 178
- `reject()` (*kombu.transport.redis.Transport.Channel.QoS* method), 176
- `reject()` (*kombu.transport.SQS.Channel.QoS* method), 199
- `reject()` (*kombu.transport.SQS.Transport.Channel.QoS* method), 197
- `reject()` (*kombu.transport.virtual.Message* method), 102
- `reject()` (*kombu.transport.virtual.QoS* method), 103
- `reject_log_error()` (*kombu.message.Message* method), 89
- `reject_log_error()` (*kombu.transport.pyamqp.Connection.Channel.Message* method), 115
- `reject_log_error()` (*kombu.transport.virtual.Message* method), 102
- `release()` (*kombu.asynchronous.semaphore.LaxBoundedSemaphore* method), 208
- `release()` (*kombu.compat.Publisher* method), 70
- `release()` (*kombu.Connection* method), 42
- `release()` (*kombu.connection.ChannelPool* method), 87
- `release()` (*kombu.connection.Connection* method), 86
- `release()` (*kombu.connection.ConnectionPool* method), 87
- `release()` (*kombu.pools.ProducerPool* method), 92
- `release()` (*kombu.pools.ProducerPool.Producer* method), 92
- `release()` (*kombu.resource.Resource* method), 93
- `release_resource()` (*kombu.resource.Resource* method), 93

- method*), 93
- `remove()` (*kombu.asynchronous.Hub method*), 205
- `remove()` (*kombu.asynchronous.hub.Hub method*), 206
- `remove_permission()`
 - (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection module kombu.pools*), 92
- `remove_permission()`
 - (*kombu.asynchronous.aws.sqs.queue.AsyncQueue module kombu.pools*), 92
- `remove_reader()` (*kombu.asynchronous.Hub method*), 205
- `remove_reader()` (*kombu.asynchronous.hub.Hub method*), 206
- `remove_writer()` (*kombu.asynchronous.Hub method*), 205
- `remove_writer()` (*kombu.asynchronous.hub.Hub method*), 206
- `replace()` (*kombu.resource.Resource method*), 93
- `reply()` (*kombu.pidbox.Node method*), 79
- `reply_exchange` (*kombu.pidbox.Mailbox attribute*), 78
- `repr_active()` (*in module kombu.asynchronous.debug*), 209
- `repr_active()` (*kombu.asynchronous.Hub method*), 205
- `repr_active()` (*kombu.asynchronous.hub.Hub method*), 206
- `repr_events()` (*in module kombu.asynchronous.debug*), 209
- `repr_events()` (*kombu.asynchronous.Hub method*), 205
- `repr_events()` (*kombu.asynchronous.hub.Hub method*), 207
- `repr_flag()` (*in module kombu.asynchronous.debug*), 209
- `repr_readers()` (*in module kombu.asynchronous.debug*), 209
- `repr_writers()` (*in module kombu.asynchronous.debug*), 209
- `Request` (*class in kombu.asynchronous.http*), 210
- `Request` (*class in kombu.asynchronous.http.base*), 213
- `request` (*kombu.asynchronous.http.base.Response attribute*), 215, 216
- `request` (*kombu.asynchronous.http.Response attribute*), 212, 213
- `request()` (*kombu.asynchronous.aws.connection.AsyncHTTPConnection module*), 219
- `request_timeout` (*kombu.asynchronous.aws.connection.AsyncHTTPConnection attribute*), 218
- `request_timeout` (*kombu.asynchronous.http.base.Request attribute*), 215
- `request_timeout` (*kombu.asynchronous.http.Request attribute*), 211
- `requeue()` (*kombu.message.Message method*), 89
- `requeue()` (*kombu.transport.base.Message method*), 98
- `requeue()` (*kombu.transport.pyamqp.Connection.Channel.Message method*), 115
- `requeue()` (*kombu.transport.virtual.Message method*), 102
- `reset()` (*in module kombu.pools*), 92
- `reset()` (*kombu.asynchronous.Hub method*), 205
- `reset()` (*kombu.asynchronous.hub.Hub method*), 207
- `resize()` (*kombu.resource.Resource method*), 93
- `resolve_aliases` (*kombu.connection.Connection attribute*), 86
- `resolve_transport()` (*in module kombu.transport*), 96
- `Resource` (*class in kombu.resource*), 93
- `Resource.LimitExceeded`, 93
- `Response` (*class in kombu.asynchronous.http*), 212
- `Response` (*class in kombu.asynchronous.http.base*), 215
- `Response` (*kombu.asynchronous.aws.connection.AsyncHTTPSConnection attribute*), 219
- `restart_limit` (*kombu.mixins.ConsumerMixin attribute*), 64
- `restore_at_shutdown`
 - (*kombu.transport.redis.Channel.QoS attribute*), 178
- `restore_at_shutdown`
 - (*kombu.transport.redis.Transport.Channel.QoS attribute*), 176
- `restore_at_shutdown` (*kombu.transport.virtual.QoS attribute*), 103
- `restore_by_tag()` (*kombu.transport.redis.Channel.QoS method*), 178
- `restore_by_tag()` (*kombu.transport.redis.Transport.Channel.QoS method*), 176
- `restore_unacked()` (*kombu.transport.redis.Channel.QoS method*), 179
- `restore_unacked()` (*kombu.transport.redis.Transport.Channel.QoS method*), 176
- `restore_unacked()` (*kombu.transport.virtual.QoS method*), 103
- `restore_unacked_once()`
 - (*kombu.transport.virtual.QoS method*), 103
- `restore_visible()` (*kombu.transport.redis.Channel.QoS method*), 179
- `restore_visible()` (*kombu.transport.redis.Transport.Channel.QoS method*), 176
- `restore_visible()` (*kombu.transport.virtual.QoS method*), 103
- `retry_backoff_factor`
 - (*kombu.transport.pyamqp.Connection.Channel attribute*), 110
- `retry_backoff_factor`
 - (*kombu.transport.azuremessagebus.Channel attribute*), 110
- `retry_backoff_factor`
 - (*kombu.transport.azuremessagebus.Transport.Channel attribute*), 109
- `retry_backoff_max` (*kombu.transport.azuremessagebus.Channel attribute*), 110
- `retry_backoff_max` (*kombu.transport.azuremessagebus.Transport.Channel attribute*), 109

- attribute), 109
- retry_on_timeout (kombu.transport.redis.Channel attribute), 180
- retry_on_timeout (kombu.transport.redis.Transport.Channel attribute), 177
- retry_over_time() (in module kombu.utils.functional), 225
- retry_total (kombu.transport.azureservicebus.Channel attribute), 111
- retry_total (kombu.transport.azureservicebus.Transport.Channel attribute), 109
- revive() (kombu.abstract.MaybeChannelBound method), 93
- revive() (kombu.compat.Consumer method), 73
- revive() (kombu.compat.ConsumerSet method), 76
- revive() (kombu.compat.Publisher method), 70
- revive() (kombu.Connection method), 44
- revive() (kombu.connection.Connection method), 86
- revive() (kombu.Consumer method), 60
- revive() (kombu.pools.ProducerPool.Producer method), 92
- revive() (kombu.Producer method), 57
- rotate() (kombu.utils.scheduling.priority_cycle method), 228
- rotate() (kombu.utils.scheduling.round_robin_cycle method), 229
- round_robin_cycle (class in kombu.utils.scheduling), 228
- routing (kombu.transport.mongodb.Channel attribute), 185
- routing (kombu.transport.mongodb.Transport.Channel attribute), 183
- routing_collection (kombu.transport.mongodb.Channel attribute), 185
- routing_collection (kombu.transport.mongodb.Transport.Channel attribute), 183
- routing_key (kombu.compat.Consumer attribute), 73
- routing_key (kombu.compat.Publisher attribute), 70
- routing_key (kombu.pools.ProducerPool.Producer attribute), 92
- routing_key (kombu.Producer attribute), 56
- routing_key (kombu.Queue attribute), 51, 55
- run() (kombu.mixins.ConsumerMixin method), 64
- run_forever() (kombu.asynchronous.Hub method), 205
- run_forever() (kombu.asynchronous.hub.Hub method), 207
- run_once() (kombu.asynchronous.Hub method), 205
- run_once() (kombu.asynchronous.hub.Hub method), 207
- S**
- safe_repr() (in module kombu.utils.encoding), 224
- safe_str() (in module kombu.utils.encoding), 224
- sanitize_url() (in module kombu.utils.url), 229
- save() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 222
- save_to_file() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 222
- save_to_filename() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 222
- save_to_s3() (kombu.asynchronous.aws.sqs.queue.AsyncQueue method), 222
- Schedule (kombu.asynchronous.timer.Timer property), 209
- scheduler (kombu.asynchronous.Hub attribute), 206
- scheduler (kombu.asynchronous.hub.Hub attribute), 207
- scheme (kombu.utils.url.urlparts property), 230
- send() (kombu.asynchronous.aws.connection.AsyncHTTPSConnection method), 219
- send() (kombu.compat.Publisher method), 70
- send_heartbeat() (kombu.transport.pyamqp.Connection method), 136
- send_message() (kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method), 220
- send_message_batch() (kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection method), 220
- send_method() (kombu.transport.pyamqp.Connection method), 136
- send_method() (kombu.transport.pyamqp.Connection.Channel method), 133
- send_reply() (in module kombu.common), 61
- sent_at (kombu.transport.sqlalchemy.models.Message attribute), 194
- SentinelChannel (class in kombu.transport.redis), 181
- sep (kombu.transport.redis.Channel attribute), 180
- sep (kombu.transport.redis.Transport.Channel attribute), 177
- serializable() (kombu.transport.qpid.Channel.Message method), 164
- serializable() (kombu.transport.qpid.Connection.Channel.Message method), 154
- serializable() (kombu.transport.qpid.Message method), 172
- serializable() (kombu.transport.qpid.Transport.Connection.Channel.Message method), 141
- serializable() (kombu.transport.virtual.Message method), 102
- serializer (kombu.compat.Publisher attribute), 70
- serializer (kombu.pools.ProducerPool.Producer attribute), 92
- serializer (kombu.Producer attribute), 56
- SerializerNotInstalled, 94
- server_capabilities (kombu.transport.pyamqp.Connection property), 136

- `server_heartbeat` (*kombu.transport.pyamqp.Connection attribute*), 136
- `session` (*kombu.transport.sqlalchemy.Channel property*), 194
- `session` (*kombu.transport.sqlalchemy.Transport.Channel property*), 193
- `session_ttl` (*kombu.transport.consul.Channel attribute*), 186
- `session_ttl` (*kombu.transport.consul.Transport.Channel attribute*), 186
- `session_ttl` (*kombu.transport.etcd.Channel attribute*), 188
- `session_ttl` (*kombu.transport.etcd.Transport.Channel attribute*), 187
- `set_attribute()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue attribute*), 177
- `set_debuglevel()` (*kombu.asynchronous.aws.connection.AsyncHTTPConnection attribute*), 219
- `set_default_encoding_file()` (*in module kombu.utils.encoding*), 224
- `set_event_loop()` (*in module kombu.asynchronous*), 206
- `set_event_loop()` (*in module kombu.asynchronous.hub*), 207
- `set_limit()` (*in module kombu.pools*), 92
- `set_queue_attribute()` (*kombu.asynchronous.aws.sqs.connection.AsyncSQSConnection attribute*), 220
- `set_timeout()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue attribute*), 222
- `setter()` (*kombu.utils.objects.cached_property method*), 228
- `setup()` (*kombu.pools.ProducerPool method*), 92
- `setup()` (*kombu.resource.Resource method*), 94
- `setup_logging()` (*in module kombu.log*), 80
- `setup_logging()` (*in module kombu.utils.debug*), 223
- `shared_queues` (*kombu.transport.pyro.Channel attribute*), 204
- `shared_queues` (*kombu.transport.pyro.Transport attribute*), 204
- `shared_queues` (*kombu.transport.pyro.Transport.Channel attribute*), 204
- `should_stop` (*kombu.mixins.ConsumerMixin attribute*), 64
- `shrink()` (*kombu.asynchronous.semaphore.LaxBoundedSemaphore method*), 208
- `SimpleBuffer` (*class in kombu.simple*), 66
- `SimpleBuffer()` (*kombu.Connection method*), 46
- `SimpleBuffer()` (*kombu.connection.Connection method*), 82
- `SimpleQueue` (*class in kombu.simple*), 65
- `SimpleQueue()` (*kombu.Connection method*), 46
- `SimpleQueue()` (*kombu.connection.Connection method*), 82
- `slmq` (*kombu.transport.SLMQ.Channel property*), 202
- `slmq` (*kombu.transport.SLMQ.Transport.Channel property*), 202
- `sock` (*kombu.transport.pyamqp.Connection property*), 136
- `socket_connect_timeout` (*kombu.transport.redis.Channel attribute*), 180
- `socket_connect_timeout` (*kombu.transport.redis.Transport.Channel attribute*), 177
- `socket_keepalive` (*kombu.transport.redis.Channel attribute*), 180
- `socket_keepalive` (*kombu.transport.redis.Transport.Channel attribute*), 177
- `socket_keepalive_options` (*kombu.transport.redis.Channel attribute*), 180
- `socket_keepalive_options` (*kombu.transport.redis.Transport.Channel attribute*), 177
- `socket_timeout` (*kombu.transport.redis.Channel attribute*), 180
- `socket_timeout` (*kombu.transport.redis.Transport.Channel attribute*), 177
- `sort_heap()` (*kombu.clocks.LamportClock method*), 67
- `SortedCycle` (*class in kombu.utils.scheduling*), 229
- `sqs()` (*kombu.transport.SQS.Channel method*), 200
- `sqs()` (*kombu.transport.SQS.Transport.Channel method*), 198
- `ssl` (*kombu.Connection attribute*), 41
- `ssl` (*kombu.connection.Connection attribute*), 86
- `ssl` (*kombu.transport.mongodb.Channel attribute*), 185
- `ssl` (*kombu.transport.mongodb.Transport.Channel attribute*), 183
- `state` (*kombu.pidbox.Node attribute*), 78
- `state` (*kombu.transport.virtual.Channel attribute*), 100
- `state` (*kombu.transport.virtual.Transport attribute*), 99
- `status` (*kombu.asynchronous.http.base.Response attribute*), 215, 216
- `status` (*kombu.asynchronous.http.Response attribute*), 212, 213
- `status_code` (*kombu.asynchronous.http.base.Response property*), 216
- `status_code` (*kombu.asynchronous.http.Response property*), 213
- `stop()` (*kombu.asynchronous.Hub method*), 206
- `stop()` (*kombu.asynchronous.hub.Hub method*), 207
- `stop()` (*kombu.asynchronous.timer.Timer method*), 209
- `store_processed` (*kombu.transport.filesystem.Channel attribute*), 192
- `store_processed` (*kombu.transport.filesystem.Transport.Channel attribute*), 191
- `str_to_bytes()` (*in module kombu.utils.encoding*), 224

- subclient (*kombu.transport.redis.Channel* attribute), 180
 - subclient (*kombu.transport.redis.Transport.Channel* attribute), 178
 - supports_exchange_type() (*kombu.connection.Connection* method), 86
 - supports_fanout (*kombu.transport.memory.Channel* attribute), 174
 - supports_fanout (*kombu.transport.memory.Transport.Channel* attribute), 173
 - supports_fanout (*kombu.transport.mongodb.Channel* attribute), 185
 - supports_fanout (*kombu.transport.mongodb.Transport.Channel* attribute), 183
 - supports_fanout (*kombu.transport.redis.Channel* attribute), 180
 - supports_fanout (*kombu.transport.redis.Transport.Channel* attribute), 178
 - supports_fanout (*kombu.transport.SQS.Channel* attribute), 200
 - supports_fanout (*kombu.transport.SQS.Transport.Channel* attribute), 198
 - supports_heartbeats (*kombu.Connection* attribute), 42
 - supports_heartbeats (*kombu.connection.Connection* property), 86
 - switch() (*kombu.Connection* method), 44
 - switch() (*kombu.connection.Connection* method), 86
 - symbol_by_name() (in module *kombu.utils.imports*), 226
- T**
- then() (*kombu.asynchronous.aws.connection.AsyncHTTPSTransport.Request* method), 218
 - then() (*kombu.asynchronous.http.base.Request* method), 215
 - then() (*kombu.asynchronous.http.Request* method), 211
 - then() (*kombu.transport.pyamqp.Connection* method), 136
 - then() (*kombu.transport.pyamqp.Connection.Channel* method), 133
 - timeout (*kombu.transport.consul.Channel* attribute), 186
 - timeout (*kombu.transport.consul.Transport.Channel* attribute), 186
 - timeout (*kombu.transport.etcd.Channel* attribute), 188
 - timeout (*kombu.transport.etcd.Transport.Channel* attribute), 187
 - TimeoutError (in module *kombu.exceptions*), 79
 - Timer (class in *kombu.asynchronous.timer*), 208
 - Timer.Entry (class in *kombu.asynchronous.timer*), 208
 - timestamp (*kombu.clocks.timetuple* property), 68
 - timestamp (*kombu.utils.limits.TokenBucket* attribute), 228
 - timetuple (class in *kombu.clocks*), 67
 - to_timestamp() (in module *kombu.asynchronous.timer*), 209
 - TokenBucket (class in *kombu.utils.limits*), 227
 - TopicExchange (class in *kombu.transport.virtual.exchange*), 105
 - TRANSIENT_DELIVERY_MODE (*kombu.Exchange* attribute), 49
 - Transport (class in *kombu.transport.azurestoragequeues*), 109
 - Transport (class in *kombu.transport.azurestoragequeues*), 107
 - Transport (class in *kombu.transport.base*), 98
 - Transport (class in *kombu.transport.consul*), 186
 - Transport (class in *kombu.transport.etcd*), 187
 - Transport (class in *kombu.transport.filesystem*), 191
 - Transport (class in *kombu.transport.memory*), 173
 - Transport (class in *kombu.transport.mongodb*), 183
 - Transport (class in *kombu.transport.pyamqp*), 112
 - Transport (class in *kombu.transport.pyro*), 203
 - Transport (class in *kombu.transport.qpid*), 139
 - Transport (class in *kombu.transport.redis*), 175
 - Transport (class in *kombu.transport.SLMQ*), 201
 - Transport (class in *kombu.transport.sqlalchemy*), 193
 - Transport (class in *kombu.transport.SQS*), 197
 - Transport (class in *kombu.transport.virtual*), 99
 - Transport (class in *kombu.transport.zookeeper*), 189
 - transport (*kombu.Connection* attribute), 42
 - transport (*kombu.connection.Connection* property), 87
 - transport (*kombu.transport.pyamqp.Connection* property), 136
 - Transport.Request (*kombu.transport.pyamqp.Connection* method), 134
 - Transport.Channel (class in *kombu.transport.azurestoragequeues*), 109
 - Transport.Channel (class in *kombu.transport.azurestoragequeues*), 107
 - Transport.Channel (class in *kombu.transport.consul*), 186
 - Transport.Channel (class in *kombu.transport.etcd*), 187
 - Transport.Channel (class in *kombu.transport.filesystem*), 191
 - Transport.Channel (class in *kombu.transport.memory*), 173
 - Transport.Channel (class in *kombu.transport.mongodb*), 183
 - Transport.Channel (class in *kombu.transport.pyro*), 203
 - Transport.Channel (class in *kombu.transport.redis*), 175
 - Transport.Channel (class in *kombu.transport.SLMQ*), 201

- 201
- `Transport.Channel` (class in `kombu.transport.sqlalchemy`), 193
- `Transport.Channel` (class in `kombu.transport.SQS`), 197
- `Transport.Channel` (class in `kombu.transport.zookeeper`), 189
- `Transport.Channel.QoS` (class in `kombu.transport.redis`), 175
- `Transport.Channel.QoS` (class in `kombu.transport.SQS`), 197
- `Transport.Connection` (class in `kombu.transport.pyamqp`), 112
- `Transport.Connection` (class in `kombu.transport.qpid`), 139
- `Transport.Connection.Channel` (class in `kombu.transport.pyamqp`), 112
- `Transport.Connection.Channel` (class in `kombu.transport.qpid`), 140
- `Transport.Connection.Channel.Message` (class in `kombu.transport.pyamqp`), 112
- `Transport.Connection.Channel.Message` (class in `kombu.transport.qpid`), 141
- `Transport.Connection.Channel.QoS` (class in `kombu.transport.qpid`), 141
- `TRANSPORT_ALIASES` (in module `kombu.transport`), 96
- `transport_options` (`kombu.connection.Connection` attribute), 87
- `transport_options` (`kombu.transport.azurestoragequeues.Channel` attribute), 111
- `transport_options` (`kombu.transport.azurestoragequeues.Transport.Channel` attribute), 109
- `transport_options` (`kombu.transport.azurestoragequeues.Transport.Channel` attribute), 108
- `transport_options` (`kombu.transport.azurestoragequeues.Transport.Channel` attribute), 107
- `transport_options` (`kombu.transport.filesystem.Channel` attribute), 192
- `transport_options` (`kombu.transport.filesystem.Transport.Channel` attribute), 191
- `transport_options` (`kombu.transport.SLMQ.Channel` attribute), 202
- `transport_options` (`kombu.transport.SLMQ.Transport.Channel` attribute), 202
- `transport_options` (`kombu.transport.SQS.Channel` attribute), 200
- `transport_options` (`kombu.transport.SQS.Transport.Channel` attribute), 198
- `tref` (`kombu.asynchronous.timer.Entry` attribute), 208
- `tref` (`kombu.asynchronous.timer.Timer.Entry` attribute), 208
- `ttl` (`kombu.transport.mongodb.Channel` attribute), 185
- `ttl` (`kombu.transport.mongodb.Transport.Channel` attribute), 183
- `tx_commit()` (`kombu.transport.pyamqp.Connection.Channel` method), 133
- `tx_rollback()` (`kombu.transport.pyamqp.Connection.Channel` method), 133
- `tx_select()` (`kombu.transport.pyamqp.Connection.Channel` method), 134
- `type` (`kombu.Exchange` attribute), 47, 50
- `type` (`kombu.pidbox.Mailbox` attribute), 78
- `type` (`kombu.transport.virtual.exchange.DirectExchange` attribute), 104
- `type` (`kombu.transport.virtual.exchange.ExchangeType` attribute), 106
- `type` (`kombu.transport.virtual.exchange.FanoutExchange` attribute), 105
- `type` (`kombu.transport.virtual.exchange.TopicExchange` attribute), 105
- `typeof()` (`kombu.transport.qpid.Channel` method), 171
- `typeof()` (`kombu.transport.qpid.Connection.Channel` method), 162
- `typeof()` (`kombu.transport.qpid.Transport.Connection.Channel` method), 149
- `typeof()` (`kombu.transport.virtual.Channel` method), 101
- ## U
- `uamqp_keep_alive_interval` (`kombu.transport.azurestoragequeues.Channel` attribute), 111
- `uamqp_keep_alive_interval` (`kombu.transport.azurestoragequeues.Transport.Channel` attribute), 109
- `unacked_index_key` (`kombu.transport.redis.Channel` attribute), 180
- `unacked_index_key` (`kombu.transport.redis.Channel.QoS` attribute), 179
- `unacked_index_key` (`kombu.transport.redis.Transport.Channel` attribute), 178
- `unacked_index_key` (`kombu.transport.redis.Transport.Channel.QoS` attribute), 176
- `unacked_key` (`kombu.transport.redis.Channel` attribute), 181
- `unacked_key` (`kombu.transport.redis.Channel.QoS` attribute), 179
- `unacked_key` (`kombu.transport.redis.Transport.Channel` attribute), 178
- `unacked_key` (`kombu.transport.redis.Transport.Channel.QoS` attribute), 176
- `unacked_mutex_expire` (`kombu.transport.redis.Channel` attribute), 181
- `unacked_mutex_expire` (`kombu.transport.redis.Channel.QoS` attribute), 179
- `unacked_mutex_expire`

- (kombu.transport.redis.Transport.Channel attribute)*, 178
 - unacked_mutex_expire**
 - (kombu.transport.redis.Transport.Channel.QoS attribute)*, 176
 - unacked_mutex_key**
 - (kombu.transport.redis.Channel attribute)*, 181
 - (kombu.transport.redis.Channel.QoS attribute)*, 179
 - (kombu.transport.redis.Transport.Channel attribute)*, 178
 - (kombu.transport.redis.Transport.Channel.QoS attribute)*, 176
 - unacked_restore_limit**
 - (kombu.transport.redis.Channel attribute)*, 181
 - (kombu.transport.redis.Transport.Channel attribute)*, 178
 - unbind_from()** *(kombu.Exchange method)*, 50
 - unbind_from()** *(kombu.Queue method)*, 55
 - unregister()** *(in module kombu.matcher)*, 63
 - unregister()** *(in module kombu.serialization)*, 95
 - unregister()** *(kombu.matcher.MatcherRegistry method)*, 62
 - update()** *(kombu.utils.functional.LRUCache method)*, 224
 - update()** *(kombu.utils.scheduling.round_robin_cycle method)*, 229
 - uri_prefix** *(kombu.Connection attribute)*, 42
 - uri_prefix** *(kombu.connection.Connection attribute)*, 87
 - URL**, 282
 - url** *(kombu.asynchronous.aws.connection.AsyncHTTPSCConnection.Request attribute)*, 87
 - url** *(kombu.asynchronous.http.base.Request attribute)*, 215
 - url** *(kombu.asynchronous.http.Request attribute)*, 211
 - url_to_parts()** *(in module kombu.utils.url)*, 229
 - urlparts** *(class in kombu.utils.url)*, 230
 - use_gzip** *(kombu.asynchronous.aws.connection.AsyncHTTPSCConnection.Request attribute)*, 219
 - use_gzip** *(kombu.asynchronous.http.base.Request attribute)*, 215
 - use_gzip** *(kombu.asynchronous.http.Request attribute)*, 211
 - user_agent** *(kombu.asynchronous.aws.connection.AsyncHTTPSCConnection.Request attribute)*, 219
 - user_agent** *(kombu.asynchronous.http.base.Request attribute)*, 215
 - user_agent** *(kombu.asynchronous.http.Request attribute)*, 211
 - userid** *(kombu.Connection attribute)*, 41
 - userid** *(kombu.connection.Connection attribute)*, 87
 - username** *(kombu.utils.url.urlparts property)*, 230
 - uuid()** *(in module kombu.common)*, 61
 - uuid()** *(in module kombu.utils.uuid)*, 230
- ## V
- validate_cert** *(kombu.asynchronous.aws.connection.AsyncHTTPSCConnection attribute)*, 219
 - validate_cert** *(kombu.asynchronous.http.base.Request attribute)*, 215
 - validate_cert** *(kombu.asynchronous.http.Request attribute)*, 211
 - value** *(kombu.clocks.LamportClock attribute)*, 67
 - values()** *(kombu.utils.functional.LRUCache method)*, 225
 - verify_connection()**
 - (kombu.transport.consul.Transport method)*, 186
 - (kombu.transport.etcd.Transport method)*, 188
 - (kombu.transport.pyamqp.Transport method)*, 114
 - (kombu.transport.qpid.Transport method)*, 152
 - version** *(kombu.transport.sqlalchemy.models.Message attribute)*, 194
 - version_string_as_tuple()** *(in module kombu.utils.text)*, 229
 - VHOST**, 282
 - virtual_host** *(kombu.Connection attribute)*, 41
 - virtual_host** *(kombu.connection.Connection attribute)*, 87
 - visibility_timeout** *(kombu.transport.redis.Channel attribute)*, 181
 - visibility_timeout** *(kombu.transport.redis.Channel.QoS attribute)*, 179
 - visibility_timeout** *(kombu.transport.redis.Transport.Channel attribute)*, 178
 - visibility_timeout** *(kombu.transport.redis.Transport.Channel.QoS attribute)*, 176
 - visibility_timeout** *(kombu.transport.SLMQ.Channel attribute)*, 202
 - visibility_timeout** *(kombu.transport.SLMQ.Transport.Channel attribute)*, 202
 - visibility_timeout** *(kombu.transport.SQS.Channel attribute)*, 200
 - visibility_timeout** *(kombu.transport.SQS.Transport.Channel attribute)*, 198
 - visible** *(kombu.transport.sqlalchemy.models.Message attribute)*, 194

W

`wait()` (*kombu.compat.Consumer* method), 73

`wait()` (*kombu.transport.pyamqp.Connection* method), 136

`wait()` (*kombu.transport.pyamqp.Connection.Channel* method), 134

`wait_time_seconds` (*kombu.transport.azure_servicebus.Channel* attribute), 111

`wait_time_seconds` (*kombu.transport.azure_servicebus.Transport.Channel* attribute), 110

`wait_time_seconds` (*kombu.transport.SQS.Channel* attribute), 200

`wait_time_seconds` (*kombu.transport.SQS.Transport* attribute), 199

`wait_time_seconds` (*kombu.transport.SQS.Transport.Channel* attribute), 198

`warn()` (*kombu.log.LogMixin* method), 80

`when_bound()` (*kombu.abstract.MaybeChannelBound* method), 93

`when_bound()` (*kombu.Queue* method), 55

`wildcards` (*kombu.transport.virtual.exchange.TopicExchange* attribute), 105

`with_traceback()` (*kombu.compat.Consumer.ContentDisallowed* method), 70

`with_traceback()` (*kombu.compat.ConsumerSet.ContentDisallowed* method), 74

`with_traceback()` (*kombu.transport.pyamqp.Connection.Channel.Message.MessageStateError* method), 114

`with_traceback()` (*kombu.transport.virtual.Message.MessageStateError* method), 101

`WRITE` (*kombu.asynchronous.Hub* attribute), 205

`WRITE` (*kombu.asynchronous.hub.Hub* attribute), 206

`write()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* method), 222

`write_batch()` (*kombu.asynchronous.aws.sqs.queue.AsyncQueue* method), 222