Kombu Documentation

Release 2.2.0rc2

Ask Solem

June 06, 2012

CONTENTS

Contents:

CHAPTER

ONE

KOMBU - MESSAGING FRAMEWORK FOR PYTHON

Version 2.2.0rc2

1.1 Synopsis

Kombu is an AMQP messaging framework for Python.

AMQP is the Advanced Message Queuing Protocol, an open standard protocol for message orientation, queuing, routing, reliability and security.

One of the most popular implementations of AMQP is RabbitMQ.

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

1.2 Features

- Allows application authors to support several message server solutions by using pluggable transports.
  - AMQP transports for both the amqplib (sync) and pika (sync + async) clients.
  - Virtual transports makes it really easy to add support for non-AMQP transports. There is already built-in support for Redis, Beanstalk, Amazon SQS, CouchDB, and MongoDB.
  - SQLAlchemy and Django ORM transports exists as plug-ins (kombu-sqlalchemy and django-kombu).
  - 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
amqplib	Native	Yes	Yes	Yes
pika	Native	Yes	Yes	Yes
redis	Virtual	Yes	Yes	Yes (PUB/SUB)
mongodb	Virtual	Yes	Yes	Yes
beanstalk	Virtual	Yes	Yes [1]	No
SQS	Virtual	Yes	Yes [1]	Yes [2]
couchdb	Virtual	Yes	Yes [1]	No
in-memory	Virtual	Yes	Yes [1]	No
django	Virtual	Yes	Yes [1]	No
sqlalchemy	Virtual	Yes	Yes [1]	No

1.3.1 Documentation

Kombu is using Sphinx, and the latest documentation is available at GitHub:

http://ask.github.com/kombu

1.3.2 Quick overview

```
from kombu import BrokerConnection, 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 BrokerConnection("amqp://guest:guest@localhost//") as conn:

    # Declare the video queue so that the messages can be delivered.
    # It is a best practice in Kombu to have both publishers and
    # consumers declare the queue.
    video_queue(conn.channel()).declare()
```

[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.

```
    # produce
    with conn.Producer(exchange=media_exchange,
                       serializer="json", routing_key="video") as producer:
        producer.publish({"name": "/tmp/lolcat1.avi", "size": 1301013})

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

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

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

Or handle channels manually:

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

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

```
from kombu import BrokerConnection, Consumer, Producer

connection = BrokerConnection()
    # ...
connection.close()

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


producer = Producer(channel_or_connection, ...)
    # ....
producer.close()
```

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:

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

>>> connection = BrokerConnection()
>>> channel = connection.channel()
>>> bound_exchange = exchange(channel)
>>> 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 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
```

1.5 Terminology

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

- Producers

  Producers sends messages to an exchange.

- Exchanges

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

- Consumers

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

- Queues

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

- Routing keys

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

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

  - Direct exchange

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

  - Fan-out exchange

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

  - Topic exchange

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

1.6 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/ask/kombu/issues/

1.8 Contributing

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

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

1.9 License

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

CHAPTER

TWO

USER GUIDE

Release 2.2

Date June 06, 2012

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 the banks messaging central. The messaging central then record and coordinate 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 will not 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 (amqplib, pika, redis, in-memory), and you can even create your own. The default transport is amqplib.

Create a connection using the default transport:

```
>>> from kombu import BrokerConnection
>>> connection = BrokerConnection("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 BrokerConnection() as connection:
    # work with connection
```

2.2.2 URLs

Connection parameters can be provided as an 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 virtual host '/foo'
amqp://localhost//foo

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

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:

```
>>> BrokerConnection("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 `BrokerConnection` 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.pyamqplib.Transport`), or one of the aliases: `amqplib`, `pika`, `redis`, `memory`, and so on.

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

insist Insist on connecting to a server. In a configuration with multiple load-sharing servers, the insist option tells the server that the client is insisting on a connection to the specified server. Default is `False`. Only supported by the amqp and pika transports, and not by AMQP 0-9-1.

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 Transport Comparison

Client	Type	Direct	Topic	Fanout
amqplib	Native	Yes	Yes	Yes
pika	Native	Yes	Yes	Yes
redis	Virtual	Yes	Yes [1]	Yes (PUB/SUB)
beanstalk	Virtual	Yes	Yes [1]	No
SQS	Virtual	Yes	Yes [1]	Yes [2]
mongodb	Virtual	Yes	Yes [1]	No
couchdb	Virtual	Yes	Yes [1]	No
in-memory	Virtual	Yes	Yes [1]	No

2.3 Producers

2.3.1 Basics

2.3.2 Serialization

See *Serialization*.

[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. Can be disabled by setting the `supports_fanout` transport option.

2.3.3 Reference

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

Message Producer.

Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing_key** – Optional default routing key.
- **serializer** – Default serializer. Default is *"json"*.
- **compression** – Default compression method. Default is no compression.
- **auto_declare** – Automatically declare the default exchange at instantiation. Default is `True`.
- **on_return** – 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.

Producer.**auto_declare** = True

By default the exchange is declared at instantiation. If you want to declare manually then you can set this to `False`.

Producer.**channel** = None

The connection channel used.

Producer.**compression** = None

Default compression method. Disabled by default.

Producer.**declare**()

Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

Producer.**exchange** = None

Default exchange.

Producer.**maybe_declare**(*entity*, *retry=False*, ***retry_policy*)

Declare the exchange if it hasn't already been declared during this session.

Producer.**on_return** = None

Basic return callback.

Producer.**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=[]*, ***properties*)

Publish message to the specified exchange.

Parameters

- **body** – Message body.
- **routing_key** – Message routing key.
- **delivery_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.

- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content_type** – Content type. Default is auto-detect.
- **content_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- ****properties** – Additional message properties, see AMQP spec.

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

`Producer.`**`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.

Draining events from a single consumer:

```
with Consumer(connection, queues):
    connection.drain_events(timeout=1)
```

Draining events from several consumers:

```
from kombu.utils import nested

with connection.channel(), connection.channel() as (channel1, channel2):
    consumers = [Consumer(channel1, queues1),
                 Consumer(channel2, queues2)]
    with nested(\*consumers):
        connection.drain_events(timeout=1)
```

Or using `ConsumerMixin`:

```
from kombu.mixins import ConsumerMixin

class C(ConsumerMixin):

    def __init__(self, connection):
```

```
        self.connection = connection

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

    def on_message(self, body, message):
        print("RECEIVED MESSAGE: %r" % (body, ))
        message.ack()

C(connection).run()
```

and with multiple channels again:

```
from kombu.messaging 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]),
                Consumer(self.channel2, queues2,
                         callbacks=[self.on_special_message])]

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

C(connection).run()
```

2.4.2 Reference

class kombu.messaging.**Consumer**(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*)

Message consumer.

Parameters

- **channel** – see `channel`.
- **queues** – see `queues`.
- **no_ack** – see `no_ack`.
- **auto_declare** – see `auto_declare`
- **callbacks** – see `callbacks`.
- **on_decode_error** – see `on_decode_error`.

Consumer.**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`.

Consumer.**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 (a subclass of Message).

Consumer.**cancel**()
End all active queue consumers.

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

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

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

Consumer.**close**()
End all active queue consumers.

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

Consumer.**declare**()
Declare queues, exchanges and bindings.

This is done automatically at instantiation if auto_declare is set.

Consumer.**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.

Consumer.**no_ack** = None
Flag for message acknowledgment disabled/enabled. Enabled by default.

Consumer.**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.

Consumer.**purge**()
Purge messages from all queues.

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

Consumer.**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** – 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** – Specify the prefetch window in terms of whole messages.
- **apply_global** – Apply new settings globally on all channels. Currently not supported by RabbitMQ.

`Consumer.`**`queues`** = **None**
A single `Queue`, or a list of queues to consume from.

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

This dispatches to the registered `callbacks`.

Parameters

- **body** – The decoded message body.
- **message** – The *Message* instance.

Raises NotImplementedError If no consumer callbacks have been registered.

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

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

Parameters **requeue** – 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.

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

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

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

2.5 Examples

2.5.1 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 __future__ import with_statement

from kombu.mixins import ConsumerMixin
from kombu.utils import kwdict, reprcall

from queues import task_queues


class Worker(ConsumerMixin):

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

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

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

if __name__ == "__main__":
    from kombu import BrokerConnection
    from kombu.utils.debug import setup_logging
    setup_logging(loglevel="INFO")

    with BrokerConnection("amqp://guest:guest@localhost:5672//") as conn:
        try:
            Worker(conn).run()
        except KeyboardInterrupt:
            print("bye bye")
```

tasks.py:

```
def hello_task(who="world"):
    print("Hello %s" % (who, ))
```

client.py:

```
from __future__ import with_statement

from kombu.common import maybe_declare
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:
        maybe_declare(task_exchange, producer.channel)
        producer.publish(payload, serializer="pickle",
                                  compression="bzip2",
                                  routing_key=routing_key)

if __name__ == "__main__":
    from kombu import BrokerConnection
    from tasks import hello_task

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

2.6 Simple Interface

`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 `BrokerConnection` 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 import SimpleQueue, SimpleBuffer

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

2.6.1 Sending and receiving messages

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

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

```
from __future__ import with_statement

from socket import gethostname
from time import time

from kombu import BrokerConnection


class Logger(object):

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

    def log(self, message, level="INFO", context={}):
        self.queue.put({"message": message,
                        "level": level,
                        "context": context,
                        "hostname": socket.gethostname(),
                        "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 BrokerConnection("amqp://guest:guest@localhost:5672//") as conn:
        with closing(Logger(connection)) 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.fromtimestamp(entry["timestamp"])
                print("[%s %s %s] %s %r" % (date,
                                            entry["hostname"],
                                            entry["level"],
                                            entry["message"],
                                            entry["context"]))
```

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

2.7 Connection and Producer Pools

2.7.1 Default Pools

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

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

The connection pool group

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

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

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

Let's acquire and release a connection:

```
from kombu import BrokerConnection
from kombu.pools import connections

connection = BrokerConnection("redis://localhost:6379")

with connections[connection].acquire(block=True) as conn:
    print("Got connection: %r" % (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 BrokerConnection
from kombu.pools import connections

c1 = BrokerConnection("amqp://")
c2 = BrokerConnection("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 BrokerConnection, Exchange
from kombu.common import maybe_declare
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 = BrokerConnection("amqp://guest:guest@localhost:5672//")

with producers[connection].acquire(block=True) as producer:
    # maybe_declare knows what entities have already been declared
    # so we don't have to do so multiple times in the same process.
    maybe_declare(news_exchange)
    producer.publish(article, routing_key="domestic",
                              serializer="json",
                              compression="zlib")
```

Setting pool limits

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

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

Resetting all pools

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

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

2.7.3 Custom Pool Groups

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

```
from kombu import pools
from kombu import BrokerConnection

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

connection = BrokerConnection("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.

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.

***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, then *Kombu* will not waste cycles serializing/deserializing the data.

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

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

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

CHAPTER

THREE

FREQUENTLY ASKED QUESTIONS

3.1 Questions

3.1.1 Q: Message.reject doesn't work?

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

3.1.2 Q: Message.requeue doesn't work?

Answer: See Message.reject doesn't work?

CHAPTER

FOUR

API REFERENCE

Release 2.2

Date June 06, 2012

4.1 kombu.connection

Broker connection and pools.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.1.1 Connection

class `kombu.connection.`**`BrokerConnection`**(*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*, ***kwargs*)

A connection to the broker.

Parameters

- **URL** – Connection URL.
- **hostname** – Default host name/address 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.
- **ssl** – Use SSL to connect to the server. Default is `False`. May not be supported by the specified transport.

- **transport** – Default transport if not specified in the URL.
- **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.
- **insist** – *Deprecated*

Note: The connection is established lazily when needed. If you need the connection to be established, then force it to do so using `connect()`:

```
>>> conn.connect()
```

Remember to always close the connection:

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

Attributes

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

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

connection
The underlying connection object.

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

Methods

connect()
Establish connection to server immediately.

channel()
Request a new channel.

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

> **Parameters** **timeout** – Timeout in seconds before we give up. Raises `socket.timeout` if the timeout is exceeded.

Usually used from an event loop.

release()
Close the connection (if open).

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

If not retry establishing the connection with the settings specified.

Parameters

- **errback** – 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** – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval_start** – The number of seconds we start sleeping for.
- **interval_step** – How many seconds added to the interval for each retry.
- **interval_max** – Maximum number of seconds to sleep between each retry.
- **callback** – Optional callback that is called for every internal iteration (1 s)
- **callback** – Optional callback that is called for every internal iteration (1 s).

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.

Will retry by establishing the connection, and reapplying the function.

Parameters

- **fun** – Method to apply.
- **errback** – 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** – Maximum number of times to retry. If this limit is exceeded the connection error will be re-raised.
- **interval_start** – The number of seconds we start sleeping for.
- **interval_step** – How many seconds added to the interval for each retry.
- **interval_max** – Maximum number of seconds to sleep between each retry.

Example

This is an example ensuring a publish operation:

```
>>> def errback(exc, interval):
...     print("Couldn't publish message: %r. Retry in %ds" % (
...             exc, interval))
>>> publish = conn.ensure(producer, producer.publish,
...                       errback=errback, max_retries=3)
>>> publish(message, routing_key)
```

create_transport()

get_transport_cls()

Get the currently used transport class.

clone(***kwargs*)

Create a copy of the connection with the same connection settings.

info()

Get connection info.

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

See ConnectionPool.

Parameters

- **limit** – Maximum number of active connections. Default is no limit.
- **preload** – Number of connections to preload when the pool is created. Default is 0.

Example usage:

```
>>> pool = connection.Pool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
```

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

See ChannelPool.

Parameters

- **limit** – Maximum number of active channels. Default is no limit.
- **preload** – Number of channels to preload when the pool is created. Default is 0.

Example usage:

```
>>> pool = connection.ChannelPool(2)
>>> c1 = pool.acquire()
>>> c2 = pool.acquire()
>>> c3 = pool.acquire()
>>> c1.release()
>>> c3 = pool.acquire()
```

SimpleQueue(*name*, *no_ack=None*, *queue_opts=None*, *exchange_opts=None*, *channel=None*, ***kwargs*)
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** – Name of the queue/or a Queue.
- **no_ack** – Disable acknowledgements. Default is false.
- **queue_opts** – Additional keyword arguments passed to the constructor of the automatically created Queue.
- **exchange_opts** – Additional keyword arguments passed to the constructor of the automatically created Exchange.
- **channel** – Channel to use. If not specified a new channel from the current connection will be used. Remember to call close() when done with the object.

SimpleBuffer(*name*, *no_ack=None*, *queue_opts=None*, *exchange_opts=None*, *channel=None*, ***kwargs*)
Create new SimpleQueue using a channel from this connection.

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 acknowledgements are disabled (`no_ack`).

4.1.2 Pools

See Also:

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

class `kombu.connection.`**`ConnectionPool`**(*connection*, *limit=None*, *preload=None*)

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

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

Parameters

- **block** – If the limit is exceeded, block until there is an available item.
- **timeout** – 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*)

class `kombu.connection.`**`ChannelPool`**(*connection*, *limit=None*, *preload=None*)

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

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

Parameters

- **block** – If the limit is exceeded, block until there is an available item.
- **timeout** – 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*)

4.2 kombu.simple

Simple interface.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.2.1 Persistent

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

channel
Current channel

producer
Producer used to publish messages.

consumer
Consumer used to receive messages.

no_ack
flag to enable/disable acknowledgements.

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()

4.2.2 Buffer

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

channel
Current channel

producer
Producer used to publish messages.

consumer
Consumer used to receive messages.

no_ack
flag to enable/disable acknowledgements.

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()

4.3 kombu.messaging

Sending and receiving messages.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.3.1 Message Producer

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

Message Producer.

Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing_key** – Optional default routing key.
- **serializer** – Default serializer. Default is *"json"*.
- **compression** – Default compression method. Default is no compression.
- **auto_declare** – Automatically declare the default exchange at instantiation. Default is True.

- **on_return** – 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 = None
The connection channel used.

exchange = None
Default exchange.

routing_key = ''

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

compression = None
Default compression method. Disabled by default.

auto_declare = True
By default the exchange is declared at instantiation. If you want to declare manually then you can set this to `False`.

on_return = None
Basic return callback.

declare()
Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

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=[]*, ***properties*)
Publish message to the specified exchange.

Parameters

- **body** – Message body.
- **routing_key** – Message routing key.
- **delivery_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content_type** – Content type. Default is auto-detect.
- **content_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.

- **retry_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- ****properties** – Additional message properties, see AMQP spec.

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

4.3.2 Message Consumer

class kombu.messaging.**Consumer**(*channel*, *queues=None*, *no_ack=None*, *auto_declare=None*, *callbacks=None*, *on_decode_error=None*)
Message consumer.

Parameters

- **channel** – see `channel`.
- **queues** – see `queues`.
- **no_ack** – see `no_ack`.
- **auto_declare** – see `auto_declare`
- **callbacks** – see `callbacks`.
- **on_decode_error** – see `on_decode_error`.

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

queues = None
A single `Queue`, or a list of queues to consume from.

no_ack = None
Flag for message acknowledgment disabled/enabled. Enabled 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 (a subclass of `Message`).

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.

declare()
Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

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

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

consume(*no_ack=None*)

cancel()
End all active queue consumers.

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.

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** – 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** – Specify the prefetch window in terms of whole messages.
- **apply_global** – Apply new settings globally on all channels. Currently not supported by RabbitMQ.

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

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

Parameters **requeue** – 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** – The decoded message body.
- **message** – The *Message* instance.

Raises NotImplementedError If no consumer callbacks have been registered.

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

4.4 kombu.entity

Exchange and Queue declarations.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.4.1 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()`:

```
>>> bound_exchange.declare()
>>> message = bound_exchange.Message("Cure for cancer found!")
>>> bound_exchange.publish(message, routing_key="news.science")
>>> bound_exchange.delete()
```

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

Parameters

- **name** – See `name`.
- **type** – See `type`.
- **channel** – See `channel`.
- **durable** – See `durable`.
- **auto_delete** – See `auto_delete`.
- **delivery_mode** – See `delivery_mode`.
- **arguments** – See `arguments`.

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

type
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.

This description of AMQP exchange types was shamelessly stolen from the blog post AMQP in 10 minutes: Part 4 by Rajith Attapattu. This article is recommended reading.

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

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

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

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).

arguments
Additional arguments to specify when the exchange is declared.

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

Message(*body*, *delivery_mode=None*, *priority=None*, *content_type=None*, *content_encoding=None*, *properties=None*, *headers=None*)
Create message instance to be sent with `publish()`.

Parameters

- **body** – Message body.
- **delivery_mode** – Set custom delivery mode. Defaults to `delivery_mode`.
- **priority** – Message priority, 0 to 9. (currently not supported by RabbitMQ).
- **content_type** – 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** – 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** – Message properties.
- **headers** – Message headers.

PERSISTENT_DELIVERY_MODE = 2

TRANSIENT_DELIVERY_MODE = 1

attrs = ((‘name’, None), (‘type’, None), (‘arguments’, None), (‘durable’, <type ‘bool’>), (‘auto_delete’, <type ‘bool’>), (‘

auto_delete = False

can_cache_declaration

declare(*nowait=False*)
Declare the exchange.

Creates the exchange on the broker.

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

delivery_mode = 2

durable = True

name = ‘’

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

Parameters

- **message** – `Message()` instance to publish.

- **routing_key** – Routing key.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.

type = 'direct'

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

class `kombu.entity.`**Queue**(*name=''*, *exchange=None*, *routing_key=''*, *channel=None*, ***kwargs*)

A Queue declaration.

Parameters

- **name** – See `name`.
- **exchange** – See `exchange`.
- **routing_key** – See `routing_key`.
- **channel** – See `channel`.
- **durable** – See `durable`.
- **exclusive** – See `exclusive`.
- **auto_delete** – See `auto_delete`.
- **queue_arguments** – See `queue_arguments`.
- **binding_arguments** – See `binding_arguments`.

name

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

exchange

The `Exchange` the queue binds to.

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"*.

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

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`.

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

Default is `False`.

auto_delete
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.

queue_arguments
Additional arguments used when declaring the queue.

binding_arguments
Additional arguments used when binding the queue.

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

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

attrs = ((‘name’, None), (‘exchange’, None), (‘routing_key’, None), (‘queue_arguments’, None), (‘binding_arguments’, N

auto_delete = False

can_cache_declaration

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** – 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** – If set messages received does not have to be acknowledged.
- **nowait** – Do not wait for a reply.
- **callback** – callback called for each delivered message

declare(*nowait=False*)
Declares the queue, the exchange and binds the queue to the exchange.

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

Parameters

- **if_unused** – 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** – 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** – Do not wait for a reply.

durable = True

exchange = <unbound Exchange (direct)>

exclusive = False

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

Returns the message instance if a message was available, or `None` otherwise.

Parameters **no_ack** – If set messages received does not have to be acknowledged.

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.

name = ''

no_ack = False

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

queue_bind(*nowait=False*)
Create the queue binding on the server.

queue_declare(*nowait=False*, *passive=False*)
Declare queue on the server.

Parameters

- **nowait** – Do not wait for a reply.
- **passive** – 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.

routing_key = ''

unbind()
Delete the binding on the server.

when_bound()

4.5 Common Utilities - kombu.common

4.5.1 kombu.common

Common Utilities.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.common.**Broadcast**(*name=None*, *queue=None*, ***kwargs*)
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** – This is used as the name of the exchange.
- **queue** – By default a unique id is used for the queue name for every consumer. You can specify a custom queue name here.
- ****kwargs** – See Queue for a list of additional keyword arguments supported.

kombu.common.**entry_to_queue**(*queue*, ***options*)

kombu.common.**maybe_declare**(*entity*, *channel*, *retry=False*, ***retry_policy*)

kombu.common.**uuid**()
Generate a unique id, having - hopefully - a very small chance of collision.

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

kombu.common.**itermessages**(*conn*, *channel*, *queue*, *limit=1*, *timeout=None*, *Consumer=<class 'kombu.messaging.Consumer'>*, *callbacks=None*, ***kwargs*)

kombu.common.**send_reply**(*exchange*, *req*, *msg*, *producer=None*, ***props*)

kombu.common.**isend_reply**(*pool*, *exchange*, *req*, *msg*, *props*, ***retry_policy*)

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

kombu.common.**insured**(*pool*, *fun*, *args*, *kwargs*, *errback=None*, *on_revive=None*, ***opts*)
Ensures function performing broker commands completes despite intermittent connection failures.

kombu.common.**ipublish**(*pool*, *fun*, *args=()*, *kwargs={}*, *errback=None*, *on_revive=None*, ***retry_policy*)

4.6 Mixin Classes - kombu.mixins

4.6.1 kombu.mixins

Useful mixin classes.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class `kombu.mixins.`**`ConsumerMixin`**

Convenience mixin for implementing consumer threads.

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 `BrokerConnection` instance, and define a `get_consumers()` method that returns a list of `kombu.messaging.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],
                         callback=[self.on_task])]

    def on_task(self, body, message):
        print("Got task: %r" % (body, ))
        message.ack()
```

Additional handler methods:

- `extra_context()`

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

  Takes arguments `(connection, channel)`.

- `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.

- `on_connection_revived()`

  Handler called when the connection is re-established after connection failure.

  Takes no arguments.

- 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.

- on_consume_end()

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

- on_iteration()

  Handler called for every iteration while draining events.

  Takes no arguments.

- 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(**args*, ***kwds*)

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*)

establish_connection(**args*, ***kwds*)

extra_context(**args*, ***kwds*)

get_consumers(*Consumer*, *channel*)

maybe_conn_error(*fun*)
Applies function but ignores any connection or channel errors raised.

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()

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.

4.7 Clocks and Synchronization - kombu.clocks

4.7.1 kombu.clocks

Logical Clocks and Synchronization.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class `kombu.clocks.`**`LamportClock`**(*initial_value=0*)

Lamport's logical clock.

From Wikipedia:

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

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

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

See Also:

- Lamport timestamps
- Lamports distributed mutex

Usage

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

`adjust`(*other*)

`forward`()

`value = 0`

The clocks current value.

4.8 kombu.compat

Carrot compatible interface for `Publisher` and `Producer`.

See http://packages.python.org/pypi/carrot for documentation.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.8.1 Publisher

Replace with `kombu.messaging.Producer`.

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

auto_declare = True

auto_delete = False

backend

channel = None

close()

compression = None

connection

declare()
Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

durable = True

exchange = ''

exchange_type = 'direct'

maybe_declare(*entity*, *retry=False*, ***retry_policy*)
Declare the exchange if it hasn't already been declared during this session.

on_return = None

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

Parameters

- **body** – Message body.
- **routing_key** – Message routing key.
- **delivery_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.

- **priority** – Message priority. A number between 0 and 9.
- **content_type** – Content type. Default is auto-detect.
- **content_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- ****properties** – Additional message properties, see AMQP spec.

release()

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

routing_key = ''

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

serializer = None

4.8.2 Consumer

Replace with `kombu.messaging.Consumer`.

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

add_queue(*queue*)

auto_declare = True

auto_delete = False

callbacks = None

cancel()
End all active queue consumers.

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

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

channel = None

close()

connection

consume(*no_ack=None*)

consuming_from(*queue*)

declare()
Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

discard_all(*filterfunc=None*)

durable = True

exchange = ''

exchange_type = 'direct'

exclusive = False

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

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

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

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

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

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

no_ack = None

on_decode_error = None

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** – 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** – Specify the prefetch window in terms of whole messages.
- **apply_global** – Apply new settings globally on all channels. Currently not supported by RabbitMQ.

queue = ''

queues = None

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

This dispatches to the registered `callbacks`.

Parameters

- **body** – The decoded message body.
- **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** – 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.

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

revive(*channel*)

routing_key = ''

wait(*limit=None*)

4.8.3 ConsumerSet

Replace with `kombu.messaging.Consumer`.

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

add_consumer(*consumer*)

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

add_queue(*queue*)

auto_declare = True

callbacks = None

cancel()
End all active queue consumers.

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

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

channel = None

close()

connection

consume(*no_ack=None*)

consuming_from(*queue*)

declare()
Declare queues, exchanges and bindings.

This is done automatically at instantiation if `auto_declare` is set.

discard_all()

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

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

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

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

no_ack = None

on_decode_error = None

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** – 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** – Specify the prefetch window in terms of whole messages.
- **apply_global** – Apply new settings globally on all channels. Currently not supported by RabbitMQ.

queues = None

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

This dispatches to the registered `callbacks`.

Parameters

- **body** – The decoded message body.

- **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 – 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.

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

revive(*channel*)

4.9 kombu.pidbox

Generic process mailbox.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.9.1 Introduction

Creating the applications Mailbox

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

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

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

Example Node

```
>>> connection = kombu.BrokerConnection()
>>> 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)
```

4.9.2 Mailbox

class kombu.pidbox.**Mailbox**(*namespace*, *type='direct'*, *connection=None*)

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={}*, *timeout=None*, *callback=None*, *channel=None*)

cast(*destination*, *command*, *kwargs={}*)

abcast(*command*, *kwargs={}*)

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

get_reply_queue(*ticket*)

get_queue(*hostname*)

4.9.3 Node

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

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*, ***options*)

handler(*fun*)

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

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

dispatch_from_message(*message*)

handle_call(*method*, *arguments*)

handle_cast(*method*, *arguments*)

handle(*method*, *arguments={}*)

handle_message(*body*, *message*)

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

4.10 kombu.exceptions

Exceptions.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

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 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.

4.11 Logging - kombu.log

class kombu.log.**LogMixin**

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

logger_name

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

kombu.log.**get_loglevel**(*level*)

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

4.12 kombu.transport

Built-in transports.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.12.1 Data

kombu.transport.**DEFAULT_TRANSPORT**
Default transport used when no transport specified.

kombu.transport.**TRANSPORT_ALIASES**
Mapping of transport aliases/class names.

4.12.2 Functions

kombu.transport.**get_transport_cls**(*transport=None*)
Get transport class by name.

The transport string is the full path to a transport class, e.g.:

```
"kombu.transport.amqplib.Transport"
```

If the name does not include "." (is not fully qualified), the alias table will be consulted.

kombu.transport.**resolve_transport**(*transport=None*)

4.13 kombu.transport.amqplib

amqplib transport.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.13.1 Transport

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

class Connection(**args*, ***kwargs*)

channel(*channel_id=None*)

drain_events(*timeout=None*)
Wait for an event on a channel.

read_timeout(*timeout=None*)

Transport.**channel_errors** = (<class 'kombu.exceptions.StdChannelError'>, <class 'amqplib.client_0_8.exceptions

Transport.**close_connection**(*connection*)
Close the AMQP broker connection.

Transport.**connection_errors** = (<class 'amqplib.client_0_8.exceptions.AMQPConnectionException'>, <class 'soc

Transport.**create_channel**(*connection*)

Transport.**default_connection_params**

Transport.**default_port** = 5672

Transport.**drain_events**(*connection*, ***kwargs*)

Transport.**establish_connection**()
Establish connection to the AMQP broker.

Transport.**eventmap**(*connection*)

Transport.**get_manager**(*hostname=None*, *port=None*, *userid=None*, *password=None*)

Transport.**is_alive**(*connection*)

Transport.**nb_keep_draining** = **True**

Transport.**on_poll_init**(*poller*)

Transport.**on_poll_start**()

Transport.**verify_connection**(*connection*)

4.13.2 Connection

class kombu.transport.amqplib.**Connection**(**args*, ***kwargs*)

channel(*channel_id=None*)

close(*reply_code=0*, *reply_text=''*, *method_sig=(0, 0)*)
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.

dispatch_method(*method_sig*, *args*, *content*)

drain_events(*timeout=None*)
Wait for an event on a channel.

read_timeout(*timeout=None*)

wait(*allowed_methods=None*)
Wait for a method that matches our allowed_methods parameter (the default value of None means match any method), and dispatch to it.

4.13.3 Channel

class kombu.transport.amqplib.**Channel**(**args*, ***kwargs*)

class Message(*channel*, *msg*, ***kwargs*)

Channel.**basic_cancel**(*consumer_tag*, ***kwargs*)

Channel.**basic_consume**(**args*, ***kwargs*)

Channel.**close**()

Channel.**events = {'basic_return': []}**

Channel.**message_to_python**(*raw_message*)
Convert encoded message body back to a Python value.

Channel.**prepare_message**(*message_data*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*)
Encapsulate data into a AMQP message.

4.13.4 Message

class kombu.transport.amqplib.**Message**(*channel*, *msg*, ***kwargs*)

4.14 kombu.transport.pika

4.15 kombu.transport.memory

In-memory transport.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.15.1 Transport

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

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

after_reply_message_received(*queue*)

do_restore = False

queues = {}

Transport.**state = <kombu.transport.virtual.BrokerState object at 0x4ee1150>**
memory backend state is global.

4.15.2 Channel

class kombu.transport.memory.**Channel**(*connection*, ***kwargs*)

after_reply_message_received(*queue*)

do_restore = False

queues = {}

4.16 kombu.transport.redis

Redis transport.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.16.1 Transport

class kombu.transport.redis.**Transport**(**args*, ***kwargs*)

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

class QoS(**args*, ***kwargs*)

ack(*delivery_tag*)

append(*message*, *delivery_tag*)

client

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

restore_at_shutdown = True

restore_by_tag(*tag*)

restore_unacked()

restore_visible(*start=0*, *num=10*, *interval=10*)

unacked_index_key

unacked_key

visibility_timeout

Transport.Channel.**active_queues**

Transport.Channel.**basic_cancel**(*consumer_tag*)

Transport.Channel.**basic_consume**(*queue*, **args*, ***kwargs*)

Transport.Channel.**client**

Transport.Channel.**close**()

Transport.Channel.**from_transport_options = ('body_encoding', 'deadletter_queue', 'unacked_key', 'u**

Transport.Channel.**get_table**(*exchange*)

Transport.Channel.**keyprefix_queue = '_kombu.binding.%s'**

Transport.Channel.**pipeline**()

Transport.Channel.**priority**(*n*)

Transport.Channel.**priority_steps = [0, 3, 6, 9]**

Transport.Channel.**sep = '\x06\x16'**

Transport.Channel.**subclient**

Transport.Channel.**supports_fanout = True**

Transport.Channel.**unacked_index_key = 'unacked_index'**

Transport.Channel.**unacked_key = 'unacked'**

Transport.Channel.**visibility_timeout = 18000**

Transport.**default_port = 6379**

Transport.**handle_event**(*fileno*, *event*)

Transport.**on_poll_init**(*poller*)

Transport.**on_poll_start**()

Transport.**polling_interval = None**

4.16.2 Channel

class kombu.transport.redis.**Channel**(**args*, ***kwargs*)

class QoS(**args*, ***kwargs*)

ack(*delivery_tag*)

append(*message*, *delivery_tag*)

client

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

restore_at_shutdown = True

restore_by_tag(*tag*)

restore_unacked()

restore_visible(*start=0*, *num=10*, *interval=10*)

unacked_index_key

unacked_key

visibility_timeout

Channel.**active_queues**

Channel.**basic_cancel**(*consumer_tag*)

Channel.**basic_consume**(*queue*, **args*, ***kwargs*)

Channel.**client**

Channel.**close**()

Channel.**from_transport_options = ('body_encoding', 'deadletter_queue', 'unacked_key', 'unacked_index_key',**

Channel.**get_table**(*exchange*)

Channel.**keyprefix_queue = '_kombu.binding.%s'**

Channel.**pipeline**()

Channel.**priority**(*n*)

Channel.**priority_steps = [0, 3, 6, 9]**

Channel.**sep = '\x06\x16'**

Channel.**subclient**

Channel.**supports_fanout = True**

Channel.**unacked_index_key = 'unacked_index'**

Channel.**unacked_key = 'unacked'**

Channel.**visibility_timeout = 18000**

4.17 kombu.transport.django

Kombu transport using the Django database as a message store.

4.17.1 Transport

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

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

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

refresh_connection()

Transport.**channel_errors** = (<class 'kombu.exceptions.StdChannelError'>, <class 'django.core.exceptions.Object

Transport.**connection_errors** = ()

Transport.**default_port** = 0

Transport.**polling_interval** = 5.0

4.17.2 Channel

class kombu.transport.django.**Channel**(*connection*, ***kwargs*)

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

refresh_connection()

4.18 Django Models - kombu.transport.django.models

class kombu.transport.django.models.**Message**(**args*, ***kwargs*)
Message(id, visible, sent_at, payload, queue_id)

exception DoesNotExist

exception Message.**MultipleObjectsReturned**

Message.**objects** = <kombu.transport.django.managers.MessageManager object at 0x4bf9c50>

Message.**queue**

class kombu.transport.django.models.**Queue**(**args*, ***kwargs*)
Queue(id, name)

exception DoesNotExist

exception Queue.**MultipleObjectsReturned**

Queue.**messages**

Queue.**objects** = <kombu.transport.django.managers.QueueManager object at 0x4bf9750>

4.19 Django Managers - kombu.transport.django.managers

class kombu.transport.django.managers.**MessageManager**

cleanup()

cleanup_every = 10

connection_for_write()

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

class kombu.transport.django.managers.**QueueManager**

fetch(*queue_name*)

publish(*queue_name*, *payload*)

purge(*queue_name*)

size(*queue_name*)

kombu.transport.django.managers.**select_for_update**(*qs*)

4.20 Django Management - clean_kombu_messages

members

undoc-members

4.21 kombu.transport.sqlalchemy

4.22 kombu.transport.base

Base transport interface.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.22.1 Message

class kombu.transport.base.**Message**(*channel*, *body=None*, *delivery_tag=None*, *content_type=None*, *content_encoding=None*, *delivery_info={}*, *properties=None*, *headers=None*, *postencode=None*, ***kwargs*)

Base class for received messages.

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()
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()
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.

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.

4.22.2 Transport

class kombu.transport.base.**Transport**(*client*, ***kwargs*)
Base class for transports.

client = None
The BrokerConnection owning this instance.

default_port = None
Default port used when no port has been specified.

connection_errors = ()
Tuple of errors that can happen due to connection failure.

channel_errors = ()
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*)

4.23 kombu.transport.virtual

Virtual transport implementation.

Emulates the AMQ API for non-AMQ transports.

4.23.1 Transports

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

Virtual transport.

Parameters **client** – BrokerConnection instance

Channel = <class 'kombu.transport.virtual.Channel'>

Cycle = <class 'kombu.transport.virtual.scheduling.FairCycle'>

polling_interval = 1.0

Time to sleep between unsuccessful polls.

default_port = None

port number used when no port is specified.

state = <kombu.transport.virtual.BrokerState object at 0x37b7090>

BrokerState containing declared exchanges and bindings (set by constructor).

cycle = None

FairCycle instance used to fairly drain events from channels (set by constructor).

establish_connection()

close_connection(*connection*)

create_channel(*connection*)

close_channel(*channel*)

drain_events(*connection*, *timeout=None*)

4.23.2 Channel

class kombu.transport.virtual.**AbstractChannel**

This is an abstract class defining the channel methods you'd usually want to implement in a virtual channel.

Do not subclass directly, but rather inherit from Channel instead.

class kombu.transport.virtual.**Channel**(*connection*, ***kwargs*)
Virtual channel.

Parameters **connection** – The transport instance this channel is part of.

Message = <class 'kombu.transport.virtual.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 = {'topic': <class 'kombu.transport.virtual.exchange.TopicExchange'>, 'fanout': <class 'kombu.trans
mapping of exchange types and corresponding classes.

exchange_declare(*exchange*, *type='direct'*, *durable=False*, *auto_delete=False*, *arguments=None*, *nowait=False*)
Declare exchange.

exchange_delete(*exchange*, *if_unused=False*, *nowait=False*)
Delete *exchange* and all its bindings.

queue_declare(*queue*, *passive=False*, ***kwargs*)
Declare queue.

queue_delete(*queue*, *if_unusued=False*, *if_empty=False*, ***kwargs*)
Delete queue.

queue_bind(*queue*, *exchange*, *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*, ***kwargs*)
Get message by direct access (synchronous).

basic_ack(*delivery_tag*)
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.

Only *prefetch_count* is supported.

get_table(*exchange*)
Get table of bindings for *exchange*.

typeof(*exchange*)
Get the exchange type instance for *exchange*.

drain_events(*timeout=None*)

prepare_message(*message_data*, *priority=None*, *content_type=None*, *content_encoding=None*, *headers=None*, *properties=None*)
Prepare message data.

message_to_python(*raw_message*)
Convert raw message to `Message` instance.

flow(*active=True*)
Enable/disable message flow.

Raises NotImplementedError as flow is not implemented by the base virtual implementation.

close()
Close channel, cancel all consumers, and requeue unacked messages.

4.23.3 Message

class kombu.transport.virtual.**Message**(*channel*, *payload*, ***kwargs*)

exception MessageStateError
The message has already been acknowledged.

args

message

Message.**ack**()
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.

Message.**ack_log_error**(*logger*, *errors*)

Message.**acknowledged**
Set to true if the message has been acknowledged.

Message.**body**

Message.**channel**

Message.**content_encoding**

Message.**content_type**

Message.**decode**()
Deserialize the message body, returning the original python structure sent by the publisher.

Message.**delivery_info**

Message.**delivery_tag**

Message.**headers**

Message.**payload**
The decoded message body.

Message.**properties**

Message.**reject**()
Reject this message.

The message will be discarded by the server.

Raises MessageStateError If the message has already been acknowledged/requeued/rejected.

Message.**reject_log_error**(*logger*, *errors*)

Message.**requeue**()
Reject this message and put it back on the queue.

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.

Message.**serializable**()

4.23.4 Quality Of Service

class kombu.transport.virtual.**QoS**(*channel*, *prefetch_count=0*)
Quality of Service guarantees.

Only supports *prefetch_count* at this point.

Parameters

- **channel** – AMQ Channel.
- **prefetch_count** – 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()
Returns true if the channel can be consumed from.

Used to ensure the client adhers to currently active prefetch limits.

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()
Restores all unacknowledged message at shutdown/gc collect.

Will only be done once for each instance.

4.23.5 In-memory State

class kombu.transport.virtual.**BrokerState**(*exchanges=None*, *bindings=None*)

bindings = None
active bindings.

clear()

exchanges = None
exchange declarations.

4.24 kombu.transport.virtual.exchange

Implementations of the standard exchanges defined by the AMQ protocol (excluding the *headers* exchange).

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.24.1 Direct

class kombu.transport.virtual.exchange.**DirectExchange**(*channel*)
The *direct* exchange routes based on exact routing keys.

deliver(*message*, *exchange*, *routing_key*, ***kwargs*)

lookup(*table*, *exchange*, *routing_key*, *default*)

type = 'direct'

4.24.2 Topic

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

prepare_bind(*queue*, *exchange*, *routing_key*, *arguments*)

type = 'topic'

wildcards = {'#': '.*?', '*': '.*?[^\\.]'}
map of wildcard to regex conversions

4.24.3 Fanout

class kombu.transport.virtual.exchange.**FanoutExchange**(*channel*)
The *fanout* exchange implements broadcast messaging by delivering copies of all messages to all queues bound the 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 the redis backend for an example implementation of these methods.

deliver(*message*, *exchange*, *routing_key*, ***kwargs*)

lookup(*table*, *exchange*, *routing_key*, *default*)

type = 'fanout'

4.24.4 Interface

class kombu.transport.virtual.exchange.**ExchangeType**(*channel*)
Implements the specifics for an exchange type.

Parameters **channel** – AMQ Channel

equivalent(*prev*, *exchange*, *type*, *durable*, *auto_delete*, *arguments*)
Returns true if *prev* and *exchange* is equivalent.

lookup(*table*, *exchange*, *routing_key*, *default*)
Lookup all queues matching *routing_key* in *exchange*.

Returns *default* if no queues matched.

prepare_bind(*queue*, *exchange*, *routing_key*, *arguments*)
Returns tuple of *(routing_key, regex, queue)* to be stored for bindings to this exchange.

type = None

- kombu.transport.virtual.scheduling

4.25 kombu.transport.virtual.scheduling

Consumer utilities.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.transport.virtual.scheduling.**FairCycle**(*fun*, *resources*, *predicate=<type 'exceptions.Exception'>*)
Consume from a set of resources, where each resource gets an equal chance to be consumed from.

close()

get(***kwargs*)

4.26 kombu.serialization

Serialization utilities.

copyright

3. 2009 - 2012 by Ask Solem

license BSD, see LICENSE for more details.

4.26.1 Overview

Centralized support for encoding/decoding of data structures. Contains json, pickle, msgpack, and yaml serializers.

Optionally installs support for YAML if the PyYAML package is installed.

Optionally installs support for msgpack if the msgpack-python package is installed.

4.26.2 Exceptions

exception kombu.serialization.**SerializerNotInstalled**
Support for the requested serialization type is not installed

4.26.3 Serialization

kombu.serialization.**encode**(*self*, *data*, *serializer=None*)

decode(data, content_type, content_encoding):
Deserialize a data stream as serialized using *encode* based on *content_type*.

Parameters

- **data** – The message data to deserialize.
- **content_type** – The content-type of the data. (e.g., *application/json*).
- **content_encoding** – The content-encoding of the data. (e.g., *utf-8*, *binary*, or *us-ascii*).

Returns The unserialized data.

kombu.serialization.**decode**(*self*, *data*, *content_type*, *content_encoding*, *force=False*)

register(name, encoder, decoder, content_type,
content_encoding="utf-8"):
Register a new encoder/decoder.

Parameters

- **name** – A convenience name for the serialization method.
- **encoder** – 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** – 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** – The mime-type describing the serialized structure.
- **content_encoding** – The content encoding (character set) that the *decoder* method will be returning. Will usually be utf-8', *us-ascii*, or *binary*.

`kombu.serialization.`**`raw_encode`**(*data*)
Special case serializer.

4.26.4 Registry

`kombu.serialization.`**`register`**(*self*, *name*, *encoder*, *decoder*, *content_type*, *content_encoding='utf-8'*)

`unregister(name):`
`Unregister registered encoder/decoder.`

Parameters **name** – Registered serialization method name.

`kombu.serialization.`**`registry = <kombu.serialization.SerializerRegistry object at 0x36fb710>`**

`kombu.serialization.`**`encode`**(*data*, *serializer=default_serializer*)
Serialize a data structure into a string suitable for sending as an AMQP message body.

Parameters

- **data** – The message data to send. Can be a list, dictionary or a string.
- **serializer** – 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.

Raises SerializerNotInstalled If the serialization method requested is not available.

4.27 kombu.compression

Compression utilities.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

4.27.1 Encoding/decoding

kombu.compression.**compress**(*body*, *content_type*)
Compress text.

Parameters

- **body** – The text to compress.
- **content_type** – mime-type of compression method to use.

kombu.compression.**decompress**(*body*, *content_type*)
Decompress compressed text.

Parameters

- **body** – Previously compressed text to uncompress.
- **content_type** – mime-type of compression method used.

4.27.2 Registry

kombu.compression.**encoders**()
Returns 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*=[])
Register new compression method.

Parameters

- **encoder** – Function used to compress text.
- **decoder** – Function used to decompress previously compressed text.
- **content_type** – The mime type this compression method identifies as.
- **aliases** – A list of names to associate with this compression method.

4.28 General Pools - kombu.pools

4.28.1 kombu.pools

Public resource pools.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.pools.**ProducerPool**(*connections*, **args*, ***kwargs*)

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

Message Producer.

Parameters

- **channel** – Connection or channel.
- **exchange** – Optional default exchange.
- **routing_key** – Optional default routing key.
- **serializer** – Default serializer. Default is *"json"*.
- **compression** – Default compression method. Default is no compression.
- **auto_declare** – Automatically declare the default exchange at instantiation. Default is `True`.
- **on_return** – 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

channel = None

close()

compression = None

connection

declare()

Declare the exchange.

This happens automatically at instantiation if `auto_declare` is enabled.

exchange = None

maybe_declare(*entity*, *retry=False*, ***retry_policy*)

Declare the exchange if it hasn't already been declared during this session.

on_return = None

publish(*body*, *routing_key=None*, *delivery_mode=None*, *mandatory=False*, *immediate=False*, *priority=0*, *content_type=None*, *content_encoding=None*, *serializer=None*, *headers=None*, *compression=None*, *exchange=None*, *retry=False*, *retry_policy=None*, *declare=[]*, ***properties*)

Publish message to the specified exchange.

Parameters

- **body** – Message body.

- **routing_key** – Message routing key.
- **delivery_mode** – See `delivery_mode`.
- **mandatory** – Currently not supported.
- **immediate** – Currently not supported.
- **priority** – Message priority. A number between 0 and 9.
- **content_type** – Content type. Default is auto-detect.
- **content_encoding** – Content encoding. Default is auto-detect.
- **serializer** – Serializer to use. Default is auto-detect.
- **compression** – Compression method to use. Default is none.
- **headers** – Mapping of arbitrary headers to pass along with the message body.
- **exchange** – Override the exchange. Note that this exchange must have been declared.
- **declare** – Optional list of required entities that must have been declared before publishing the message. The entities will be declared using `maybe_declare()`.
- **retry** – Retry publishing, or declaring entities if the connection is lost.
- **retry_policy** – Retry configuration, this is the keywords supported by `ensure()`.
- ****properties** – Additional message properties, see AMQP spec.

release()

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

routing_key = ''

serializer = None

ProducerPool.**create_producer**()

ProducerPool.**new**()

ProducerPool.**prepare**(*p*)

ProducerPool.**release**(*resource*)

ProducerPool.**setup**()

class kombu.pools.**PoolGroup**(*limit=None*)

create(*resource*, *limit*)

kombu.pools.**register_group**(*group*)

kombu.pools.**get_limit**()

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

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

4.29 kombu.compression

Object utilities.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

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.

channel
Current channel if the object is bound.

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.

4.30 Async Utilities - kombu.syn

4.30.1 kombu.syn

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

kombu.syn.**detect_environment**()

4.31 Utilities - kombu.utils

4.31.1 kombu.utils

Internal utilities.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.utils.**EqualityDict**

kombu.utils.**say**(*m*, **s*)

kombu.utils.**uuid**()
Generate a unique id, having - hopefully - a very small chance of collision.

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

kombu.utils.**kwdict**(*kwargs*)

kombu.utils.**maybe_list**(*v*)

kombu.utils.**fxrange**(*start=1.0*, *stop=None*, *step=1.0*, *repeatlast=False*)

kombu.utils.**fxrangemax**(*start=1.0*, *stop=None*, *step=1.0*, *max=100.0*)

kombu.utils.**retry_over_time**(*fun*, *catch*, *args=[]*, *kwargs={}*, *errback=None*, *max_retries=None*, *interval_start=2*, *interval_step=2*, *interval_max=30*, *callback=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** – The function to try
- **catch** – Exceptions to catch, can be either tuple or a single exception class.
- **args** – Positional arguments passed on to the function.
- **kwargs** – Keyword arguments passed on to the function.
- **errback** – Callback for when an exception in `catch` is raised. The callback must take two arguments: `exc` and `interval`, where `exc` is the exception instance, and `interval` is the time in seconds to sleep next..
- **max_retries** – Maximum number of retries before we give up. If this is not set, we will retry forever.
- **interval_start** – How long (in seconds) we start sleeping between retries.
- **interval_step** – By how much the interval is increased for each retry.
- **interval_max** – Maximum number of seconds to sleep between retries.

kombu.utils.**emergency_dump_state**(*state*, *open_file=<built-in function open>*, *dump=None*)

kombu.utils.**cached_property**
Property descriptor that caches the return value of the get function.

Examples

```
@cached_property
def connection(self):
    return Connection()

@connection.setter  # Prepares stored value
def connection(self, value):
    if value is None:
        raise TypeError("Connection must be a connection")
    return value

@connection.deleter
def connection(self, value):
    # Additional action to do at del(self.attr)
```

```
        if value is not None:
            print("Connection %r deleted" % (value, ))
```

kombu.utils.**reprkwargs**(*kwargs*, *sep=', '*, *fmt='%s=%s'*)

kombu.utils.**reprcall**(*name*, *args=()*, *kwargs=()*, *sep=', '*)

kombu.utils.**nested**(**args*, ***kwds*)
Combine multiple context managers into a single nested context manager.

4.32 Rate limiting - kombu.utils.limits

4.32.1 kombu.utils.limits

Token bucket implementation for rate limiting.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.utils.limits.**TokenBucket**(*fill_rate*, *capacity=1*)
Token Bucket Algorithm.

See http://en.wikipedia.org/wiki/Token_Bucket Most of this code was stolen from an entry in the ASPN Python Cookbook: http://code.activestate.com/recipes/511490/

Thread safety

This implementation may not be thread safe.

---

can_consume(*tokens=1*)
Returns `True` if *tokens* number of tokens can be consumed from the bucket.

capacity = 1
Maximum number of tokensin the bucket.

expected_time(*tokens=1*)
Returns the expected time in seconds when a new token should be available.

Warning

This consumes a token from the bucket.

---

fill_rate = None
The rate in tokens/second that the bucket will be refilled

timestamp = None
Timestamp of the last time a token was taken out of the bucket.

4.33 Compat. utilities - kombu.utils.compat

4.33.1 kombu.utils.compat

Helps compatibility with older Python versions.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.utils.compat.**CompatOrderedDict**(**args*, ***kwds*)
Dictionary that remembers insertion order

clear() → None. Remove all items from od.

copy() → a shallow copy of od

classmethod fromkeys(*S*[, *v*]) → New ordered dictionary with keys from S
and values equal to v (which defaults to None).

items()

iteritems()

iterkeys()

itervalues()

keys()

pop(*key*, *default=<object object at 0x3593150>*)

popitem() -> (*k*, *v*)
Return and remove a (key, value) pair. Pairs are returned in LIFO order if last is true or FIFO order if false.

setdefault(*key*, *default=None*)

update(*other=()*, ***kwds*)

values()

class kombu.utils.compat.**LifoQueue**(*maxsize=0*)

4.34 Debugging - kombu.utils.debug

4.34.1 kombu.utils.debug

Debugging support.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

kombu.utils.debug.**setup_logging**(*loglevel=10, loggers=['kombu.connection', 'kombu.channel']*)

class kombu.utils.debug.**Logwrapped**(*instance*, *logger=None*, *ident=None*)

4.35 String Encoding - kombu.utils.encoding

4.35.1 kombu.utils.encoding

Utilities to encode text, and to safely emit text from running applications without crashing with the infamous `UnicodeDecodeError` exception.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

kombu.utils.encoding.**bytes_to_str**(*s*)

kombu.utils.encoding.**default_encode**(*obj*)

kombu.utils.encoding.**default_encoding**()

kombu.utils.encoding.**ensure_bytes**(*s*)

kombu.utils.encoding.**from_utf8**(*s*, **args*, ***kwargs*)

kombu.utils.encoding.**safe_repr**(*o*, *errors='replace'*)

kombu.utils.encoding.**safe_str**(*s*, *errors='replace'*)

kombu.utils.encoding.**str_to_bytes**(*s*)

4.36 kombu.utils.functional

kombu.utils.functional.**maybe_promise**(*value*)
Evaluates if the value is a promise.

class kombu.utils.functional.**promise**(*fun*, **args*, ***kwargs*)
A promise.

Evaluated when called or if the `evaluate()` method is called. The function is evaluated on every access, so the value is not memoized (see `mpromise`).

Overloaded operations that will evaluate the promise: `__str__()`, `__repr__()`, `__cmp__()`.

evaluate()

4.37 Finalize - kombu.utils.finalize

4.37.1 kombu.utils.finalize

Execute cleanup handlers when objects go out of scope.

Taken from `multiprocessing.util.Finalize`.

copyright

3. 2009 - 2012 by Ask Solem.

license BSD, see LICENSE for more details.

class kombu.utils.finalize.**Finalize**(*obj*, *callback*, *args=()*, *kwargs=None*, *exitpriority=None*)
Object finalization using weakrefs.

cancel()
Cancel finalization of the object.

still_active()

4.38 kombu.utils.url

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

CHAPTER

FIVE

CHANGE HISTORY

5.1 2.1.7

release-date 2012-04-27 6:00 P.M BST

- compat consumerset now accepts optional channel argument.

5.2 2.1.6

release-date 2012-04-23 1:30 P.M BST

- 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.

5.3 2.1.5

release-date 2012-04-13 3:30 P.M BST

- The url parser removed more than the first leading slash (Issue #121).
- SQLAlchemy: Can now specify url using + separator

  Example:

  ```
  BrokerConnection("sqla+mysql://localhost/db")
  ```

- Better support for anonymous queues (Issue #116).

  Contributed by Michael Barrett.
- `Connection.as_uri` now quotes url parts (Issue #117).
- Beanstalk: Can now set message TTR as a message property.

  Contributed by Andrii Kostenko

5.4 2.1.4

release-date 2012-04-03 4:00 P.M GMT

- 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 `StdChannelError`.
- All kombu exceptions now inherit from a common `KombuError`.

5.5 2.1.3

release-date 2012-03-20 3:00 P.M GMT

by Ask Solem

- Fixes Jython compatibility issues.
- Fixes Python 2.5 compatibility issues.

5.6 2.1.2

release-date 2012-03-01 01:00 P.M GMT

by Ask Solem

- amqplib: Last version broke SSL support.

5.7 2.1.1

release-date 2012-02-24 02:00 P.M GMT

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 amqplib'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.

5.8 2.1.0

release-date 2012-02-04 10:38 P.M GMT

by Ask Solem

- MongoDB: Now supports fanout (broadcast) (Issue #98).

  Contributed by Scott Lyons.
- amqplib: 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:

  ```
  >>> BrokerConnection("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()`.

5.9 2.0.0

release-date 2012-01-15 18:34 P.M GMT

by Ask Solem

5.9.1 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 an 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 `djkombu` 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.

5.9.2 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.

5.9.3 Fixes

- Redis transport: Timeout was multiplied by 1000 seconds when using `select` for event I/O (Issue #86).

5.10 1.5.1

release-date 2011-11-30 01:00 P.M GMT

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")
```

5.11 1.5.0

release-date 2011-11-27 06:00 P.M GMT

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 = BrokerConnection("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 acknowledgements (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).

5.12 1.4.3

release-date 2011-10-27 10:00 P.M BST

- Fixes bug in ProducerPool where too many resources would be acquired.

5.13 1.4.2

release-date 2011-10-26 05:00 P.M BST

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
- BrokerConnection.Consumer had the wrong signature.

  Fix contributed by Pavel Skvazh

5.14 1.4.1

release-date 2011-09-26 04:00 P.M BST

by Ask Solem

- 1.4.0 broke the producer pool, resulting in new connections being established for every acquire.

5.15 1.4.0

release-date 2011-09-22 05:00 P.M BST

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.messaging.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 BrokerConnection:

  ```
  >>> connection.Producer(exchange)
  >>> connection.Consumer(queues=..., callbacks=...)
  ```

- BrokerConnection has aquired 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 establising 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.

5.16 1.3.5

release-date 2011-09-16 06:00 P.M BST

by Ask Solem

- Python 3: AMQP_PROTOCOL_HEADER must be bytes, not str.

5.17 1.3.4

release-date 2011-09-16 06:00 P.M BST

by Ask Solem

- Fixes syntax error in pools.reset

5.18 1.3.3

release-date 2011-09-15 02:00 P.M BST

by Ask Solem

- pools.reset did not support after forker arguments.

5.19 1.3.2

release-date 2011-09-10 01:00 P.M BST

by Mher Movsisyan

- Broke Python 2.5 compatibility by importing `parse_qsl` 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.

5.20 1.3.1

release-date 2011-10-07 03:00 P.M BST

- Last release broke after fork for pool reinitialization.
- Producer/Consumer now has a `connection` attribute, giving access to the `BrokerConnection` of the instance.
- Pika: Channels now have access to the underlying `BrokerConnection` 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`.

5.21 1.3.0

release-date 2011-10-05 01:00 P.M BST

- Broker connection info can be now be specified using URLs

  The broker hostname can now be given as an URL instead, of the format:

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

  for example the default broker is expressed as:

  ```
  >>> BrokerConnection("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:

  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 `BrokerConnection`:

  ```
  >>> from kombu import BrokerConnection, connections
  >>> connection = BrokerConnection("amqp://guest:guest@localhost//")
  >>> with connections[connection].acquire(block=True):
  ...     # do something with connection
  ```

  To acquire a producer using the connection parameters from a `BrokerConnection`:

  ```
  >>> from kombu import BrokerConnection, producers
  >>> connection = BrokerConnection("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.

5.22 1.2.1

release-date 2011-07-29 12:52 P.M BST

- 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).
- `BrokerConnection.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.

- `BrokerConnection.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.

5.23 1.2.0

release-date 2011-07-15 12:00 P.M BST

- 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

5.24 1.1.6

release-date 2011-06-13 04:00 P.M BST

- 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:

```
>>> BrokerConnection(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.
- amqplib: Now uses *localhost* as default hostname instead of raising an error.

5.25 1.1.5

release-date 2011-06-07 06:00 P.M BST

- Fixes compatibility with redis-py 2.4.4.

5.26 1.1.4

release-date 2011-06-07 04:00 P.M BST

- Redis transport: Now requires redis-py version 2.4.4 or later.
- New Amazon SQS transport added.

  Usage:

  ```
  >>> conn = BrokerConnection(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.
- amqplib transport: Now supports *login_method* for SSL auth.

  `BrokerConnection` now supports the *login_method* keyword argument.

  Default *login_method* is `AMQPLAIN`.

5.27 1.1.3

release-date 2011-04-21 16:00 P.M CEST

- 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 cancelling 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.
- amqplib: `AttributeError` has been added to the list of known connection related errors (`Connection.connection_errors`).
- amqplib: Now converts `SSLError` timeout errors to `socket.timeout` (http://bugs.python.org/issue10272)
- Ensures cyclic references are destroyed when the connection is closed.

5.28 1.1.2

release-date 2011-04-06 16:00 P.M CEST

- 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.

5.29 1.1.1

release-date 2011-04-05 15:51 P.M CEST

- 1.1.0 started using `Queue.LifoQueue` which is only available in Python 2.6+ (Issue #33). We now ship with our own LifoQueue.

5.30 1.1.0

release-date 2011-04-05 01:05 P.M CEST

5.30.1 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:

  ```
  BrokerConnection(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.

- BrokerConnection: 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 `BrokerConnection` (Issue #30).
- BrokerConnection: Default autentication credentials are now delegated to the individual transports.

  This means that the `userid` and `password` arguments to BrokerConnection 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`.

5.31 1.0.7

release-date 2011-03-28 05:45 P.M CEST

- 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 `BrokerConnection.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.

- BrokerConnection 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).

5.32 1.0.6

release-date 2011-03-22 04:00 P.M CET

- The `delivery_mode` aliases (persistent/transient) were not automatically converted to integer, and would cause a crash if using the amqplib 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 BrokerConnection
>>> conn = BrokerConnection()
>>> 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)
```

5.33 1.0.5

release-date 2011-03-17 04:00 P.M CET

- 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.

5.34 1.0.4

release-date 2011-02-28 04:00 P.M CET

- 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.

5.35 1.0.3

release-date 2011-02-12 04:00 P.M CET

- ConnectionPool: Re-connect if amqplib connection closed
- Adds `Queue.as_dict` + `Exchange.as_dict`.
- Copyright headers updated to include 2011.

5.36 1.0.2

release-date 2011-01-31 10:45 P.M CET

- amqplib: Message properties were not set properly.
- Ghettoq backend names are now automatically translated to the new names.

5.37 1.0.1

release-date 2011-01-28 12:00 P.M CET

- Redis: Now works with Linux (epoll)

5.38 1.0.0

release-date 2011-01-27 12:00 P.M CET

- Initial release

5.39 0.1.0

release-date 2010-07-22 04:20 P.M CET

- Initial fork of carrot

CHAPTER

SIX

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

k