This document describes the current stable version of Celery (4.2). For development docs, go here.

Routing Tasks

Note:

Alternate routing concepts like topic and fanout is not available for all transports, please consult the transport comparison table.

Previous topic

Periodic Tasks

Next topic

Monitoring and Management Guide

This Page

Show Source

Quick search

Go

Basics

Automatic routing

The simplest way to do routing is to use the `task_create_missing_queues` setting (on by default).

With this setting on, a named queue that's not already defined in `task_queues` will be created automatically. This makes it easy to perform simple routing tasks.

Say you have two servers, *x*, and *y* that handles regular tasks, and one server *z*, that only handles feed related tasks. You can use this configuration:

```
task_routes = {'feed.tasks.import_feed': {'queue': 'feeds'}}
```

With this route enabled import feed tasks will be routed to the *"feeds"* queue, while all

other tasks will be routed to the default queue (named *"celery"* for historical reasons).

Alternatively, you can use glob pattern matching, or even regular expressions, to match all tasks in the `feed.tasks` name-space:

```
app.conf.task_routes = {'feed.tasks.*': {'queue': 'feeds'}}
```

If the order of matching patterns is important you should specify the router in *items* format instead:

```
task_routes = ([
    ('feed.tasks.*', {'queue': 'feeds'}),
    ('web.tasks.*', {'queue': 'web'}),
    (re.compile(r'(video|image)\.tasks\..*'), {'queue': 'media'}),
],)
```

Note:

The **`task_routes`** setting can either be a dictionary, or a list of router objects, so in this case we need to specify the setting as a tuple containing a list.

After installing the router, you can start server *z* to only process the feeds queue like this:

```
user@z:/$ celery -A proj worker -Q feeds
```

You can specify as many queues as you want, so you can make this server process the default queue as well:

```
user@z:/$ celery -A proj worker -Q feeds,celery
```

Changing the name of the default queue

You can change the name of the default queue by using the following configuration:

```
app.conf.task_default_queue = 'default'
```

How the queues are defined

The point with this feature is to hide the complex AMQP protocol for users with only basic needs. However – you may still be interested in how these queues are declared.

A queue named *"video"* will be created with the following settings:

```
{'exchange': 'video',
 'exchange_type': 'direct',
 'routing_key': 'video'}
```

The non-AMQP backends like *Redis* or *SQS* don't support exchanges, so they require the exchange to have the same name as the queue. Using this design ensures it will work for them as well.

Manual routing

Say you have two servers, *x*, and *y* that handles regular tasks, and one server *z*, that only handles feed related tasks, you can use this configuration:

```
from kombu import Queue

app.conf.task_default_queue = 'default'
app.conf.task_queues = (
    Queue('default',    routing_key='task.#'),
    Queue('feed_tasks', routing_key='feed.#'),
)
task_default_exchange = 'tasks'
task_default_exchange_type = 'topic'
task_default_routing_key = 'task.default'
```

`task_queues` is a list of `Queue` instances. If you don't set the exchange or exchange type values for a key, these will be taken from the `task_default_exchange` and `task_default_exchange_type` settings.

To route a task to the *feed_tasks* queue, you can add an entry in the `task_routes` setting:

```
task_routes = {
        'feeds.tasks.import_feed': {
            'queue': 'feed_tasks',
            'routing_key': 'feed.import',
        },
}
```

You can also override this using the *routing_key* argument to `Task.apply_async()`, or `send_task()`:

```
>>> from feeds.tasks import import_feed
>>> import_feed.apply_async(args=['http://cnn.com/rss'],
...                         queue='feed_tasks',
...                         routing_key='feed.import')
```

To make server *z* consume from the feed queue exclusively you can start it with the `celery worker -Q` option:

```
user@z:/$ celery -A proj worker -Q feed_tasks --hostname=z@%h
```

Servers *x* and *y* must be configured to consume from the default queue:

```
user@x:/$ celery -A proj worker -Q default --hostname=x@%h
user@y:/$ celery -A proj worker -Q default --hostname=y@%h
```

If you want, you can even have your feed processing worker handle regular tasks as well, maybe in times when there's a lot of work to do:

```
user@z:/$ celery -A proj worker -Q feed_tasks,default --hostname=z@%h
```

If you have another queue but on another exchange you want to add, just specify a custom exchange and exchange type:

```
from kombu import Exchange, Queue

app.conf.task_queues = (
    Queue('feed_tasks',    routing_key='feed.#'),
    Queue('regular_tasks', routing_key='task.#'),
    Queue('image_tasks',   exchange=Exchange('mediatasks', type='direct'
                           routing_key='image.compress'),
)
```

If you're confused about these terms, you should read up on AMQP.

See also:

In addition to the AMQP Primer below, there's Rabbits and Warrens, an excellent blog post describing queues and exchanges. There's also The *CloudAMQP tutorial*, For users of RabbitMQ the RabbitMQ FAQ could be useful as a source of information.

Special Routing Options

RabbitMQ Message Priorities

supported transports:
RabbitMQ

New in version 4.0.

Queues can be configured to support priorities by setting the `x-max-priority` argument:

```
from kombu import Exchange, Queue

app.conf.task_queues = [
    Queue('tasks', Exchange('tasks'), routing_key='tasks',
          queue_arguments={'x-max-priority': 10}),
]
```

A default value for all queues can be set using the `task_queue_max_priority` setting:

```
app.conf.task_queue_max_priority = 10
```

AMQP Primer

Messages

A message consists of headers and a body. Celery uses headers to store the content type of the message and its content encoding. The content type is usually the serialization format used to serialize the message. The body contains the name of the task to execute, the task id (UUID), the arguments to apply it with and some additional meta-data – like the number of retries or an ETA.

This is an example task message represented as a Python dictionary:

```
{'task': 'myapp.tasks.add',
 'id': '54086c5e-6193-4575-8308-dbab76798756',
 'args': [4, 4],
 'kwargs': {}}
```

Producers, consumers, and brokers

The client sending messages is typically called a *publisher*, or a *producer*, while the entity receiving messages is called a *consumer*.

The *broker* is the message server, routing messages from producers to consumers.

You're likely to see these terms used a lot in AMQP related material.

Exchanges, queues, and routing keys

1. Messages are sent to exchanges.
2. An exchange routes messages to one or more queues. Several exchange types exists, providing different ways to do routing, or implementing different messaging scenarios.
3. The message waits in the queue until someone consumes it.
4. The message is deleted from the queue when it has been acknowledged.

The steps required to send and receive messages are:

1. Create an exchange
2. Create a queue
3. Bind the queue to the exchange.

Celery automatically creates the entities necessary for the queues in **`task_queues`** to work (except if the queue's *auto_declare* setting is set to **`False`**).

Here's an example queue configuration with three queues; One for video, one for images, and one default queue for everything else:

```
from kombu import Exchange, Queue

app.conf.task_queues = (
    Queue('default', Exchange('default'), routing_key='default'),
    Queue('videos',  Exchange('media'),   routing_key='media.video'),
    Queue('images',  Exchange('media'),   routing_key='media.image'),
)
app.conf.task_default_queue = 'default'
app.conf.task_default_exchange_type = 'direct'
app.conf.task_default_routing_key = 'default'
```

Exchange types

The exchange type defines how the messages are routed through the exchange. The exchange types defined in the standard are *direct*, *topic*, *fanout* and *headers*. Also non-standard exchange types are available as plug-ins to RabbitMQ, like the last-value-cache plug-in by Michael Bridgen.

Direct exchanges

Direct exchanges match by exact routing keys, so a queue bound by the routing key *video* only receives messages with that routing key.

Topic exchanges

Topic exchanges matches routing keys using dot-separated words, and the wild-card characters: `*` (matches a single word), and `#` (matches zero or more words).

With routing keys like `usa.news`, `usa.weather`, `norway.news`, and `norway.weather`, bindings could be `*.news` (all news), `usa.#` (all items in the USA), or `usa.weather` (all USA weather items).

Related API commands

`exchange.declare(exchange_name, type, passive, durable, auto_delete, internal)`

Declares an exchange by name.

See **`amqp:Channel.exchange_declare`**.

Keyword Arguments:

- **passive** – Passive means the exchange won't be created, but you can use this to check if the exchange already exists.
- **durable** – Durable exchanges are persistent (i.e., they survive a broker restart).
- **auto_delete** – This means the exchange will be deleted by the broker when there are no more queues using it.

`queue.declare`(*queue_name*, *passive*, *durable*, *exclusive*, *auto_delete*)

Declares a queue by name.

See **amqp:Channel.queue_declare**

Exclusive queues can only be consumed from by the current connection. Exclusive also implies *auto_delete*.

`queue.bind`(*queue_name*, *exchange_name*, *routing_key*)

Binds a queue to an exchange with a routing key.

Unbound queues won't receive messages, so this is necessary.

See **amqp:Channel.queue_bind**

`queue.delete`(*name*, *if_unused=False*, *if_empty=False*)

Deletes a queue and its binding.

See **amqp:Channel.queue_delete**

`exchange.delete`(*name*, *if_unused=False*)

Deletes an exchange.

See **amqp:Channel.exchange_delete**

Note:

Declaring doesn't necessarily mean "create". When you declare you *assert* that the entity exists and that it's operable. There's no rule as to whom should initially create the exchange/queue/binding, whether consumer or producer. Usually the first one to need it will be the one to create it.

Hands-on with the API

Celery comes with a tool called **celery amqp** that's used for command line access to the AMQP API, enabling access to administration tasks like creating/deleting queues and exchanges, purging queues or sending messages. It can also be used for non-AMQP brokers, but different implementation may not implement all commands.

You can write commands directly in the arguments to **celery amqp**, or just start with no arguments to start it in shell-mode:

```
$ celery -A proj amqp
-> connecting to amqp://guest@localhost:5672/.
-> connected.
1>
```

Here `1>` is the prompt. The number 1, is the number of commands you have executed so far. Type `help` for a list of commands available. It also supports auto-completion, so you can start typing a command and then hit the *tab* key to show a list of possible matches.

Let's create a queue you can send messages to:

```
$ celery -A proj amqp
1> exchange.declare testexchange direct
ok.
2> queue.declare testqueue
ok. queue:testqueue messages:0 consumers:0.
3> queue.bind testqueue testexchange testkey
ok.
```

This created the direct exchange `testexchange`, and a queue named `testqueue`. The queue is bound to the exchange using the routing key `testkey`.

From now on all messages sent to the exchange `testexchange` with routing key `testkey` will be moved to this queue. You can send a message by using the `basic.publish` command:

```
4> basic.publish 'This is a message!' testexchange testkey
ok.
```

Now that the message is sent you can retrieve it again. You can use the `basic.get`` command here, that polls for new messages on the queue in a synchronous manner (this is OK for maintenance tasks, but for services you want to use `basic.consume` instead)

Pop a message off the queue:

```
5> basic.get testqueue
{'body': 'This is a message!',
 'delivery_info': {'delivery_tag': 1,
                   'exchange': u'testexchange',
                   'message_count': 0,
                   'redelivered': False,
                   'routing_key': u'testkey'},
 'properties': {}}
```

AMQP uses acknowledgment to signify that a message has been received and processed successfully. If the message hasn't been acknowledged and consumer channel is closed, the message will be delivered to another consumer.

Note the delivery tag listed in the structure above; Within a connection channel, every received message has a unique delivery tag, This tag is used to acknowledge the message. Also note that delivery tags aren't unique across connections, so in another client the delivery tag *1* might point to a different message than in this channel.

You can acknowledge the message you received using `basic.ack`:

```
6> basic.ack 1
ok.
```

To clean up after our test session you should delete the entities you created:

```
7> queue.delete testqueue
ok. 0 messages deleted.
8> exchange.delete testexchange
ok.
```

Routing Tasks

Defining queues

In Celery available queues are defined by the `task_queues` setting.

Here's an example queue configuration with three queues; One for video, one for images,

and one default queue for everything else:

```
default_exchange = Exchange('default', type='direct')
media_exchange = Exchange('media', type='direct')

app.conf.task_queues = (
    Queue('default', default_exchange, routing_key='default'),
    Queue('videos', media_exchange, routing_key='media.video'),
    Queue('images', media_exchange, routing_key='media.image')
)
app.conf.task_default_queue = 'default'
app.conf.task_default_exchange = 'default'
app.conf.task_default_routing_key = 'default'
```

Here, the **task_default_queue** will be used to route tasks that doesn't have an explicit route.

The default exchange, exchange type, and routing key will be used as the default routing values for tasks, and as the default values for entries in **task_queues**.

Multiple bindings to a single queue are also supported. Here's an example of two routing keys that are both bound to the same queue:

```
from kombu import Exchange, Queue, binding

media_exchange = Exchange('media', type='direct')

CELERY_QUEUES = (
    Queue('media', [
        binding(media_exchange, routing_key='media.video'),
        binding(media_exchange, routing_key='media.image'),
    ]),
)
```

Specifying task destination

The destination for a task is decided by the following (in order):

1. The Routers defined in **task_routes**.
2. The routing arguments to **Task.apply_async()**.
3. Routing related attributes defined on the **Task** itself.

It's considered best practice to not hard-code these settings, but rather leave that as configuration options by using Routers; This is the most flexible approach, but sensible defaults can still be set as task attributes.

Routers

A router is a function that decides the routing options for a task.

All you need to define a new router is to define a function with the signature `(name, args, kwargs, options, task=None, **kw)`:

```
def route_task(name, args, kwargs, options, task=None, **kw):
        if name == 'myapp.tasks.compress_video':
            return {'exchange': 'video',
                    'exchange_type': 'topic',
                    'routing_key': 'video.compress'}
```

If you return the `queue` key, it'll expand with the defined settings of that queue in **task_queues**:

```
{'queue': 'video', 'routing_key': 'video.compress'}
```

becomes –>

```
{'queue': 'video',
 'exchange': 'video',
 'exchange_type': 'topic',
 'routing_key': 'video.compress'}
```

You install router classes by adding them to the **`task_routes`** setting:

```
task_routes = (route_task,)
```

Router functions can also be added by name:

```
task_routes = ('myapp.routers.route_task',)
```

For simple task name -> route mappings like the router example above, you can simply drop a dict into **`task_routes`** to get the same behavior:

```
task_routes = {
    'myapp.tasks.compress_video': {
        'queue': 'video',
        'routing_key': 'video.compress',
    },
}
```

The routers will then be traversed in order, it will stop at the first router returning a true value, and use that as the final route for the task.

You can also have multiple routers defined in a sequence:

```
task_routes = [
    route_task,
    {
        'myapp.tasks.compress_video': {
            'queue': 'video',
            'routing_key': 'video.compress',
    },
]
```

The routers will then be visited in turn, and the first to return a value will be chosen.

Broadcast

Celery can also support broadcast routing. Here is an example exchange `broadcast_tasks` that delivers copies of tasks to all workers connected to it:

```
from kombu.common import Broadcast

app.conf.task_queues = (Broadcast('broadcast_tasks'),)
app.conf.task_routes = {
    'tasks.reload_cache': {
        'queue': 'broadcast_tasks',
        'exchange': 'broadcast_tasks'
    }
}
```

Now the `tasks.reload_cache` task will be sent to every worker consuming from this queue.

Here is another example of broadcast routing, this time with a **celery beat** schedule:

```python
from kombu.common import Broadcast
from celery.schedules import crontab

app.conf.task_queues = (Broadcast('broadcast_tasks'),)

app.conf.beat_schedule = {
    'test-task': {
        'task': 'tasks.reload_cache',
        'schedule': crontab(minute=0, hour='*/3'),
        'options': {'exchange': 'broadcast_tasks'}
    },
}
```

Broadcast & Results:

Note that Celery result doesn't define what happens if two tasks have the same task_id. If the same task is distributed to more than one worker, then the state history may not be preserved.

It's a good idea to set the `task.ignore_result` attribute in this case.