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

Calling Tasks

- Basics
- Linking (callbacks/errbacks)
- On message
- ETA and Countdown
- Expiration
- Message Sending Retry
- Connection Error Handling
- Serializers
- Compression
- Connections
- Routing options
- Results options

Basics

This document describes Celery's uniform "Calling API" used by task instances and the canvas.

The API defines a standard set of execution options, as well as three methods:

- `apply_async(args[, kwargs[, …]])`

  Sends a task message.

- `delay(*args, **kwargs)`

  Shortcut to send a task message, but doesn't support execution options.

- *calling* (`__call__`)

  Applying an object supporting the calling API (e.g., `add(2, 2)`) means that the task will not be executed by a worker, but in the current process instead (a message won't be sent).

Quick Cheat Sheet

- `T.delay(arg, kwarg=value)`
  Star arguments shortcut to `.apply_async`. (`.delay(*args, **kwargs)` calls `.apply_async(args, kwargs)`).
- `T.apply_async((arg,), {'kwarg': value})`
- `T.apply_async(countdown=10)`
  executes in 10 seconds from now.
- `T.apply_async(eta=now + timedelta(seconds=10))`
  executes in 10 seconds from now, specified using `eta`
- `T.apply_async(countdown=60, expires=120)`
  executes in one minute from now, but expires after 2 minutes.
- `T.apply_async(expires=now + timedelta(days=2))`
  expires in 2 days, set using **`datetime`**.

Example

The **`delay()`** method is convenient as it looks like calling a regular function:

```
task.delay(arg1, arg2, kwarg1='x', kwarg2='y')
```

Using **`apply_async()`** instead you have to write:

```
task.apply_async(args=[arg1, arg2], kwargs={'kwarg1': 'x', 'kwarg2': 'y'
```

So *delay* is clearly convenient, but if you want to set additional execution options you have to use `apply_async`.

The rest of this document will go into the task execution options in detail. All examples use a task called *add*, returning the sum of two arguments:

Tip

If the task isn't registered in the current process you can use **`send_task()`** to call the task by name instead.

```
@app.task
def add(x, y):
    return x + y
```

There's another way…

You'll learn more about this later while reading about the Canvas, but **`signature`**'s are objects used to pass around the signature of a task invocation, (for example to send it over the network), and they also support the Calling API:

```
task.s(arg1, arg2, kwarg1='x', kwargs2='y').apply_async()
```

Linking (callbacks/errbacks)

Celery supports linking tasks together so that one task follows another. The callback task will be applied with the result of the parent task as a partial argument:

```
add.apply_async((2, 2), link=add.s(16))
```

Here the result of the first task (4) will be sent to a new task that adds 16 to the previous result, forming the expression $(2 + 2) + 16 = 20$

You can also cause a callback to be applied if task raises an exception (*errback*), but this behaves differently from a regular callback in that it will be passed the id of the parent task, not the result. This is because it may not always be possible to serialize the exception raised, and so this way the error callback requires a result backend to be enabled, and the task must retrieve the result of the task instead.

This is an example error callback:

What's `s`?

The `add.s` call used here is called a signature. If you don't know what they are you should read about them in the canvas guide. There you can also learn about **`chain`**: a simpler way to chain tasks together.

In practice the `link` execution option is considered an internal primitive, and you'll probably not use it directly, but use chains instead.

```
@app.task
def error_handler(uuid):
    result = AsyncResult(uuid)
    exc = result.get(propagate=False)
    print('Task {0} raised exception: {1!r}\n{2!r}'.format(
          uuid, exc, result.traceback))
```

it can be added to the task using the `link_error` execution option:

```
add.apply_async((2, 2), link_error=error_handler.s())
```

In addition, both the `link` and `link_error` options can be expressed as a list:

```
add.apply_async((2, 2), link=[add.s(16), other_task.s()])
```

The callbacks/errbacks will then be called in order, and all callbacks will be called with the return value of the parent task as a partial argument.

On message

Celery supports catching all states changes by setting on_message callback.

For example for long-running tasks to send task progress you can do something like this:

```
@app.task(bind=True)
def hello(self, a, b):
    time.sleep(1)
    self.update_state(state="PROGRESS", meta={'progress': 50})
    time.sleep(1)
    self.update_state(state="PROGRESS", meta={'progress': 90})
    time.sleep(1)
    return 'hello world: %i' % (a+b)
```

```
def on_raw_message(body):
    print(body)

r = hello.apply_async()
print(r.get(on_message=on_raw_message, propagate=False))
```

Will generate output like this:

```
{'task_id': '5660d3a3-92b8-40df-8ccc-33a5d1d680d7',
 'result': {'progress': 50},
 'children': [],
 'status': 'PROGRESS',
 'traceback': None}
{'task_id': '5660d3a3-92b8-40df-8ccc-33a5d1d680d7',
 'result': {'progress': 90},
 'children': [],
 'status': 'PROGRESS',
 'traceback': None}
{'task_id': '5660d3a3-92b8-40df-8ccc-33a5d1d680d7',
 'result': 'hello world: 10',
 'children': [],
 'status': 'SUCCESS',
 'traceback': None}
hello world: 10
```

ETA and Countdown

The ETA (estimated time of arrival) lets you set a specific date and time that is the earliest time at which your task will be executed. *countdown* is a shortcut to set ETA by seconds into the future.

```
>>> result = add.apply_async((2, 2), countdown=3)
>>> result.get()    # this takes at least 3 seconds to return
20
```

The task is guaranteed to be executed at some time *after* the specified date and time, but not necessarily at that exact time. Possible reasons for broken deadlines may include many items waiting in the queue, or heavy network latency. To make sure your tasks are executed in a timely manner you should monitor the queue for congestion. Use Munin, or similar tools, to receive alerts, so appropriate action can be taken to ease the workload. See Munin.

While *countdown* is an integer, *eta* must be a **datetime** object, specifying an exact date and time (including millisecond precision, and timezone information):

```
>>> from datetime import datetime, timedelta

>>> tomorrow = datetime.utcnow() + timedelta(days=1)
>>> add.apply_async((2, 2), eta=tomorrow)
```

Expiration

The *expires* argument defines an optional expiry time, either as seconds after task publish, or a specific date and time using **datetime**:

```
>>> # Task expires after one minute from now.
>>> add.apply_async((10, 10), expires=60)

>>> # Also supports datetime
>>> from datetime import datetime, timedelta
>>> add.apply_async((10, 10), kwargs,
...                 expires=datetime.now() + timedelta(days=1)
```

When a worker receives an expired task it will mark the task as **REVOKED** (**TaskRevokedError**).

Message Sending Retry

Celery will automatically retry sending messages in the event of connection failure, and retry behavior can be configured – like how often to retry, or a maximum number of retries – or disabled all together.

To disable retry you can set the `retry` execution option to `False`:

```
add.apply_async((2, 2), retry=False)
```

Related Settings

- **task_publish_retry**
- **task_publish_retry_policy**

Retry Policy

A retry policy is a mapping that controls how retries behave, and can contain the following keys:

- *max_retries*

Maximum number of retries before giving up, in this case the exception that caused the retry to fail will be raised.

A value of **`None`** means it will retry forever.

The default is to retry 3 times.

- *interval_start*

  Defines the number of seconds (float or integer) to wait between retries. Default is 0 (the first retry will be instantaneous).

- *interval_step*

  On each consecutive retry this number will be added to the retry delay (float or integer). Default is 0.2.

- *interval_max*

  Maximum number of seconds (float or integer) to wait between retries. Default is 0.2.

For example, the default policy correlates to:

```
add.apply_async((2, 2), retry=True, retry_policy={
    'max_retries': 3,
    'interval_start': 0,
    'interval_step': 0.2,
    'interval_max': 0.2,
})
```

the maximum time spent retrying will be 0.4 seconds. It's set relatively short by default because a connection failure could lead to a retry pile effect if the broker connection is down – For example, many web server processes waiting to retry, blocking other incoming requests.

Connection Error Handling

When you send a task and the message transport connection is lost, or the connection cannot be initiated, an **`OperationalError`** error will be raised:

```
>>> from proj.tasks import add
>>> add.delay(2, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "celery/app/task.py", line 388, in delay
        return self.apply_async(args, kwargs)
  File "celery/app/task.py", line 503, in apply_async
    **options
  File "celery/app/base.py", line 662, in send_task
    amqp.send_task_message(P, name, message, **options)
  File "celery/backends/rpc.py", line 275, in on_task_call
    maybe_declare(self.binding(producer.channel), retry=True)
  File "/opt/celery/kombu/kombu/messaging.py", line 204, in _get_channel
    channel = self._channel = channel()
  File "/opt/celery/py-amqp/amqp/connection.py", line 272, in connect
    self.transport.connect()
  File "/opt/celery/py-amqp/amqp/transport.py", line 100, in connect
    self._connect(self.host, self.port, self.connect_timeout)
  File "/opt/celery/py-amqp/amqp/transport.py", line 141, in _connect
    self.sock.connect(sa)
  kombu.exceptions.OperationalError: [Errno 61] Connection refused
```

If you have retries enabled this will only happen after retries are exhausted, or when disabled immediately.

You can handle this error too:

```
>>> from celery.utils.log import get_logger
>>> logger = get_logger(__name__)

>>> try:
...     add.delay(2, 2)
... except add.OperationalError as exc:
...     logger.exception('Sending task raised: %r', exc)
```

Serializers

Data transferred between clients and workers needs to be serialized, so every message in Celery has a `content_type` header that describes the serialization method used to encode it.

The default serializer is *JSON*, but you can change this using the **`task_serializer`** setting, or for each individual task, or even per message.

Security

The pickle module allows for execution of arbitrary functions, please see the security guide.

Celery also comes with a special serializer that uses cryptography to sign your messages.

There's built-in support for *JSON*, **`pickle`**, *YAML* and `msgpack`, and you can also add your own custom serializers by registering them into the Kombu serializer registry

See also:

Message Serialization in the Kombu user guide.

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 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.

Binary data will be transferred using Base64 encoding, increasing the size of the transferred data by 34% compared to an encoding format where native binary types are supported.

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.

See http://json.org for more information.

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.

See **`pickle`** for more information.

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.

See http://yaml.org/ for more information.

msgpack – msgpack is a binary serialization format that's closer to JSON
in features. It's very young however, and support should be considered experimental at this point.

See http://msgpack.org/ for more information.

The encoding used is available as a message header, so the worker knows how to deserialize any task. If you use a custom serializer, this serializer must be available for the worker.

The following order is used to decide the serializer used when sending a task:

1. The *serializer* execution option.
2. The **`Task.serializer`** attribute
3. The **`task_serializer`** setting.

Example setting a custom serializer for a single task invocation:

```
>>> add.apply_async((10, 10), serializer='json')
```

Compression

Celery can compress the messages using either *gzip*, or *bzip2*. You can also create your own compression schemes and register them in the **`kombu compression registry`**.

The following order is used to decide the compression scheme used when sending a task:

1. The *compression* execution option.
2. The **`Task.compression`** attribute.
3. The **`task_compression`** attribute.

Example specifying the compression used when calling a task:

```
>>> add.apply_async((2, 2), compression='zlib')
```

Connections

You can handle the connection manually by creating a publisher:

```
results = []
with add.app.pool.acquire(block=True) a
    with add.get_publisher(connection)
        try:
            for args in numbers:
                res = add.apply_async((
                results.append(res)
print([res.get() for res in results])
```

Though this particular example is much better expressed as a group:

Automatic Pool Support

Since version 2.3 there's support for automatic connection pools, so you don't have to manually handle connections and publishers to reuse connections.

The connection pool is enabled by default since version 2.5.

See the **`broker_pool_limit`** setting for more information.

```
>>> from celery import group

>>> numbers = [(2, 2), (4, 4), (8, 8), (16, 16)]
>>> res = group(add.s(i, j) for i, j in numbers).apply_async()

>>> res.get()
[4, 8, 16, 32]
```

Routing options

Celery can route tasks to different queues.

Simple routing (name <-> name) is accomplished using the `queue` option:

```
add.apply_async(queue='priority.high')
```

You can then assign workers to the `priority.high` queue by using the workers **`-Q`** argument:

```
$ celery -A proj worker -l info -Q celery,priority.high
```

See also:

Hard-coding queue names in code isn't recommended, the best practice is to use configuration routers (**`task_routes`**).

To find out more about routing, please see Routing Tasks.

Results options

You can enable or disable result storage using the `ignore_result` option:

```
result = add.apply_async(1, 2, ignore_result=True)
result.get() # -> None

# Do not ignore result (default)
result = add.apply_async(1, 2, ignore_result=False)
result.get() # -> 3
```

See also:

For more information on tasks, please see Tasks.

Advanced Options

These options are for advanced users who want to take use of AMQP's full routing capabilities. Interested parties may read the routing guide.

- exchange

  Name of exchange (or a **`kombu.entity.Exchange`**) to send the message to.

- routing_key

  Routing key used to determine.

- priority

  A number between *0* and *255*, where *255* is the highest priority.

Supported by: RabbitMQ, Redis (priority reversed, 0 is highest).