Star 11,909

Please help support this community project with a donation:

Donate

Previous topic

User Guide

Next topic

Tasks

This Page

Show Source

Quick search

Go

YOUR AD HERE

Reach over 7 million devs each month when you advertise with Read the Docs.

Sponsored · Ads served ethically

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

Application

The Celery library must be instantiated before use, this instance is called an application (or *app* for short).

The application is thread-safe so that multiple Celery applications with different configurations, components, and tasks can co-exist in the same process space.

Let's create one now:

```
>>> from celery import Celery
>>> app = Celery()
>>> app
<Celery __main__:0x100469fd0>
```

The last line shows the textual representation of the application: including the name of the app class (`Celery`), the name of the current main module (`__main__`), and the memory address of the object (`0x100469fd0`).

Main Name

Only one of these is important, and that's the main module name. Let's look at why that is.

When you send a task message in Celery, that message won't contain any source code, but only the name of the task you want to execute. This works similarly to how host names work on the internet: every worker maintains a mapping of task names to their actual functions, called the *task registry*.

Whenever you define a task, that task will also be added to the local registry:

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

>>> add
<@task: __main__.add>

>>> add.name
__main__.add

>>> app.tasks['__main__.add']
<@task: __main__.add>
```

and there you see that `__main__` again; whenever Celery isn't able to detect what module the function belongs to, it uses the main module name to generate the beginning of the task name.

This is only a problem in a limited set of use cases:

1. If the module that the task is defined in is run as a program.
2. If the application is created in the Python shell (REPL).

For example here, where the tasks module is also used to start a worker with **`app.worker_main()`**:

`tasks.py`:

```
from celery import Celery
app = Celery()

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

if __name__ == '__main__':
    app.worker_main()
```

When this module is executed the tasks will be named starting with "`__main__`", but when the module is imported by another process, say to call a task, the tasks will be named starting with "`tasks`" (the real name of the module):

```
>>> from tasks import add
>>> add.name
tasks.add
```

You can specify another name for the main module:

```
>>> app = Celery('tasks')
>>> app.main
'tasks'

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

>>> add.name
tasks.add
```

See also:

Names

Configuration

There are several options you can set that'll change how Celery works. These options can be set directly on the app instance, or you can use a dedicated configuration module.

The configuration is available as **`app.conf`**:

```
>>> app.conf.timezone
'Europe/London'
```

where you can also set configuration values directly:

```
>>> app.conf.enable_utc = True
```

or update several keys at once by using the `update` method:

```
>>> app.conf.update(
...     enable_utc=True,
...     timezone='Europe/London',
...)
```

The configuration object consists of multiple dictionaries that are consulted in order:

1. Changes made at run-time.
2. The configuration module (if any)
3. The default configuration (`celery.app.defaults`).

You can even add new default sources by using the `app.add_defaults()` method.

See also:

Go to the Configuration reference for a complete listing of all the available settings, and their default values.

config_from_object

The `app.config_from_object()` method loads configuration from a configuration object.

This can be a configuration module, or any object with configuration attributes.

Note that any configuration that was previously set will be reset when `config_from_object()` is called. If you want to set additional configuration you should do so after.

Example 1: Using the name of a module

The `app.config_from_object()` method can take the fully qualified name of a Python module, or even the name of a Python attribute, for example: `"celeryconfig"`, `"myproj.config.celery"`, or `"myproj.config:CeleryConfig"`:

```
from celery import Celery

app = Celery()
app.config_from_object('celeryconfig')
```

The `celeryconfig` module may then look like this:

`celeryconfig.py`:

```
enable_utc = True
timezone = 'Europe/London'
```

and the app will be able to use it as long as `import celeryconfig` is possible.

Example 2: Passing an actual module object

You can also pass an already imported module object, but this isn't always recommended.

Tip:

Using the name of a module is recommended as this means the module does not need to be serialized when the prefork pool is used. If you're experiencing configuration problems or pickle errors then please try using the name of a module instead.

```
import celeryconfig

from celery import Celery

app = Celery()
app.config_from_object(celeryconfig)
```

Example 3: Using a configuration class/object

```
from celery import Celery

app = Celery()

class Config:
    enable_utc = True
    timezone = 'Europe/London'

app.config_from_object(Config)
# or using the fully qualified name of the object:
#   app.config_from_object('module:Config')
```

config_from_envvar

The **app.config_from_envvar()** takes the configuration module name from an environment variable

For example – to load configuration from a module specified in the environment variable named **CELERY_CONFIG_MODULE**:

```
import os
from celery import Celery

#: Set default configuration module name
os.environ.setdefault('CELERY_CONFIG_MODULE', 'celeryconfig')

app = Celery()
app.config_from_envvar('CELERY_CONFIG_MODULE')
```

You can then specify the configuration module to use via the environment:

```
$ CELERY_CONFIG_MODULE="celeryconfig.prod" celery worker -l info
```

Censored configuration

If you ever want to print out the configuration, as debugging information or similar, you may also want to filter out sensitive information like passwords and API keys.

Celery comes with several utilities useful for presenting the configuration, one is **humanize()**:

```
>>> app.conf.humanize(with_defaults=False, censored=True)
```

This method returns the configuration as a tabulated string. This will only contain changes to the configuration by default, but you can include the built-in default keys and values by enabling the `with_defaults` argument.

If you instead want to work with the configuration as a dictionary, you can use the **table()** method:

```
>>> app.conf.table(with_defaults=False, censored=True)
```

Please note that Celery won't be able to remove all sensitive information, as it merely uses a regular expression to search for commonly named keys. If you add custom settings containing sensitive information you should name the keys using a name that Celery identifies as secret.

A configuration setting will be censored if the name contains any of these sub-strings:

`API`, `TOKEN`, `KEY`, `SECRET`, `PASS`, `SIGNATURE`, `DATABASE`

Laziness

The application instance is lazy, meaning it won't be evaluated until it's actually needed.

Creating a **`Celery`** instance will only do the following:

1. Create a logical clock instance, used for events.
2. Create the task registry.
3. Set itself as the current app (but not if the `set_as_current` argument was disabled)
4. Call the **`app.on_init()`** callback (does nothing by default).

The **`app.task()`** decorators don't create the tasks at the point when the task is defined, instead it'll defer the creation of the task to happen either when the task is used, or after the application has been *finalized*,

This example shows how the task isn't created until you use the task, or access an attribute (in this case **`repr()`**):

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

>>> type(add)
<class 'celery.local.PromiseProxy'>

>>> add.__evaluated__()
False

>>> add        # <-- causes repr(add) to happen
<@task: __main__.add>

>>> add.__evaluated__()
True
```

Finalization of the app happens either explicitly by calling **`app.finalize()`** – or implicitly by accessing the **`app.tasks`** attribute.

Finalizing the object will:

1. Copy tasks that must be shared between apps

   Tasks are shared by default, but if the `shared` argument to the task decorator is disabled, then the task will be private to the app it's bound to.

2. Evaluate all pending task decorators.

3. Make sure all tasks are bound to the current app.

   Tasks are bound to an app so that they can read default values from the configuration.

The "default app"

Celery didn't always have applications, it used to be that there was only a module-

based API, and for backwards compatibility the old API is still there until the release of Celery 5.0.

Celery always creates a special app - the "default app", and this is used if no custom application has been instantiated.

The **`celery.task`** module is there to accommodate the old API, and shouldn't be used if you use a custom app. You should always use the methods on the app instance, not the module based API.

For example, the old Task base class enables many compatibility features where some may be incompatible with newer features, such as task methods:

```
from celery.task import Task   # << OLD Task base class.

from celery import Task        # << NEW base class.
```

The new base class is recommended even if you use the old module-based API.

Breaking the chain

While it's possible to depend on the current app being set, the best practice is to always pass the app instance around to anything that needs it.

I call this the "app chain", since it creates a chain of instances depending on the app being passed.

The following example is considered bad practice:

```
from celery import current_app

class Scheduler(object):

    def run(self):
        app = current_app
```

Instead it should take the `app` as an argument:

```
class Scheduler(object):

    def __init__(self, app):
        self.app = app
```

Internally Celery uses the **`celery.app.app_or_default()`** function so that everything also works in the module-based compatibility API

```
from celery.app import app_or_default

class Scheduler(object):
    def __init__(self, app=None):
        self.app = app_or_default(app)
```

In development you can set the **`CELERY_TRACE_APP`** environment variable to raise an exception if the app chain breaks:

```
$ CELERY_TRACE_APP=1 celery worker -l info
```

Evolving the API

Celery has changed a lot in the 7 years since it was initially created.

For example, in the beginning it was possible to use any callable as a task:

```
def hello(to):
    return 'hello {0}'.format(to)

>>> from celery.execute import apply_async

>>> apply_async(hello, ('world!',))
```

or you could also create a `Task` class to set certain options, or override other behavior

```
from celery.task import Task
from celery.registry import tasks

class Hello(Task):
    queue = 'hipri'

    def run(self, to):
        return 'hello {0}'.format(to)
tasks.register(Hello)

>>> Hello.delay('world!')
```

Later, it was decided that passing arbitrary call-able's was an anti-pattern, since it makes it very hard to use serializers other than pickle, and the feature was removed in 2.0, replaced by task decorators:

```
from celery.task import task

@task(queue='hipri')
def hello(to):
    return 'hello {0}'.format(to)
```

Abstract Tasks

All tasks created using the **`task()`** decorator will inherit from the application's base **`Task`** class.

You can specify a different base class using the `base` argument:

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

To create a custom task class you should inherit from the neutral base class: **`celery.Task`**.

```
from celery import Task

class DebugTask(Task):

    def __call__(self, *args, **kwargs):
        print('TASK STARTING: {0.name}[{0.request.id}]'.format(self))
        return super(DebugTask, self).__call__(*args, **kwargs)
```

Tip:

If you override the tasks `__call__` method, then it's very important that you also call super so that the base call method can set up the default request used when a task is called directly.

The neutral base class is special because it's not bound to any specific app yet. Once a task is bound to an app it'll read configuration to set default values, and so on.

To realize a base class you need to create a task using the `app.task()` decorator:

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

It's even possible to change the default base class for an application by changing its `app.Task()` attribute:

```
>>> from celery import Celery, Task

>>> app = Celery()

>>> class MyBaseTask(Task):
...    queue = 'hipri'

>>> app.Task = MyBaseTask
>>> app.Task
<unbound MyBaseTask>

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

>>> add
<@task: __main__.add>

>>> add.__class__.mro()
[<class add of <Celery __main__:0x1012b4410>>,
 <unbound MyBaseTask>,
 <unbound Task>,
 <type 'object'>]
```


© Copyright 2009-2018, Ask Solem & contributors.