Yehan Djoe Follow
Jun 16, 2017 · 7 min read

Celery 4 Periodic Task in Django

Automation in Django is a developer dream. Tedious work such as creating database backup, reporting annual KPI, or even blasting email could be made a breeze. Through Celery—a well-known software in Python for delegating task—such action made possible.

Preparation

Environment

Unfortunately, only a handful tutorial available for Celery 4—as the rest is older—which leads to confusion and messy configuration. After experimenting with **Ubuntu 16.04 LTS, Python 3.6.1, Django 1.11.2, Celery 4.0.2, and Redis 3.0.6**, I finally grasped how it works. Together with **virtualenv 15.1.0**, project environment is contained safely.

Redis (as a Broker)

Make sure you are set with Redis before proceeding; it acts as a message broker for Celery. Rose Hosting gives us a nice article on how to install Redis on Ubuntu.

Project Structure

This is the standard structure Django provides; very easy to maintain as it promotes modularity.

```
proj
├──proj
|  ├──__init__.py
|  ├──settings.py
|  └──celery.py
├──app1
|  └──tasks.py
└──app2
   └──tasks.py
```

PIP Library

First, install Celery by executing `pip install celery==4.0.2`. We do not need `django-celery` anymore as stated in the official Celery-Django guide below:

> *Since Celery 3.1, Django is supported without additional library*

However, feel free to use other libraries such as `django-celery-results` or `django-celery-beat` as complementary (I am not going to cover those).

Configuring Celery

celery.py

Create the following file at the `proj/proj/celery.py`:

```
from __future__ import absolute_import, unicode_literals
import os
from celery import Celery

# set the default Django settings module for the 'celery'
program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE',
'proj.settings')

app = Celery('proj')

# Using a string here means the worker don't have to
```

```
serialize
# the configuration object to child processes.
# - namespace='CELERY' means all celery-related
configuration keys
#   should have a `CELERY_` prefix.
app.config_from_object('django.conf:settings',
namespace='CELERY')

# Load task modules from all registered Django app configs.
app.autodiscover_tasks()


@app.task(bind=True)
def debug_task(self):
    print('Request: {0!r}'.format(self.request))
```

`Celery()` object contains Celery tasks and configurations. By defining `config_from_object` and `namespace` , we are going to set Celery configs inside Django's `settings.py` with any variables starts with `'CELERY_'` . I called this "Namespace Configuration".

There is another way called "Direct Configuration", where config values are assigned to `app` directly. However, that is not the scope at this time.

settings.py

Following the "Namespace Configuration", we shift focus towards `proj/proj/settings.py` and add the following lines:

```
# Other Django configurations...
# Celery application definition
#
http://docs.celeryproject.org/en/v4.0.2/userguide/configurat
ion.html

CELERY_BROKER_URL = 'redis://localhost:6379'
CELERY_RESULT_BACKEND = 'redis://localhost:6379'
CELERY_ACCEPT_CONTENT = ['application/json']
CELERY_RESULT_SERIALIZER = 'json'
CELERY_TASK_SERIALIZER = 'json'
CELERY_TIMEZONE = 'Asia/Makassar'
CELERY_BEAT_SCHEDULE = {}
```

The `CELERY_BEAT_SCHEDULE` is where we will define our scheduled task. Right now it's an empty dict, but we are going to fill it up once we create our task.

If `USE_TZ = True` (timezone is active), then set the corresponding `CELERY_TIMEZONE`.

__init__.py

The last step is to ensure Django loads `app` when it starts. Don't forget to add these lines at `proj/proj/__init__.py`:

```
from __future__ import absolute_import, unicode_literals

# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from .celery import app as celery_app
```

```
__all__ = ['celery_app']
```

Creating Task

Great! Now it’s time to create any task to automate. For example, at `proj/app1/tasks.py` , create the following method:

```
from __future__ import absolute_import, unicode_literals
from celery import task

@task()
def task_number_one():
    # Do something...
```

And at `proj/app2/tasks.py` :

```
from __future__ import absolute_import, unicode_literals
from celery import task


@task()
def task_number_two():
    # Do another thing...
```

The same method also applies when creating other tasks. Notice that we are using `@task`, not `@shared_task`. Don't worry, Celery is still be able to auto-discover your tasks.

Celery Beat Schedule

We have configured Celery and created tasks successfully. However, there is something left behind… *The beat schedule is still empty, isn't it?*

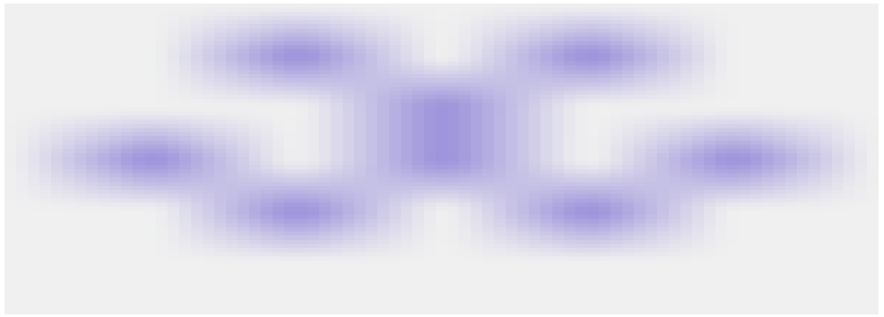
Back to `proj/proj/settings.py`, edit the `CELERY_BEAT_SCHEDULE`.

```
# The following lines may contains pseudo-code

from celery.schedules import crontab

# Other Celery settings
CELERY_BEAT_SCHEDULE = {
    'task-number-one': {
        'task': 'app1.tasks.task_number_one',
        'schedule': crontab(minute=59, hour=23),
        'args': (*args)
    },
    'task-number-two': {
        'task': 'app2.tasks.task_number_two',
        'schedule': crontab(minute=0, hour='*/3,10-19'),
        'args': (*args)
    }
}
```

Only add `*args` if your function contains arguments; otherwise remove `'args': (*args)` line. And for more information, read this doc.

Full Code Review

Phew! That's a lot of things! And now to synchronize our rhythm, here is the compilation:


Full Code Review

Have we beat at the same frequency yet?
> **Yes** to continue :)
> **No** to iterate ;P

Starting the Worker and the Beat

Configurations are all in places, tasks are defined, and what's left is to let it run. Celery requires *both* of the worker and the beat in order for tasks to execute as planned. When developing, use this command:

```
$ celery -A proj worker -l info -B
```

It will start both simultaneously. Despite its simplicity, the method is discouraged for production environment. We have to find another way.

Individual Service

Since the method above is only suitable for development, each of the worker and the beat service should be started separately by the following commands:

```
$ celery -A proj worker -l info
$ celery -A proj beat -l info
```

Starting service manually is nice, but redundant. The solution is daemonization—making the services automatically starts along with the system.

Daemonization

Although the offical doc provides daemonization, I found it quite impractical. I have stumbled upon a very nice article by Real Python. They utilize an Ubuntu library called **supervisor**. First, install it using apt-get:

```
$ sudo apt-get install supervisor
```

Then, add `/etc/supervisor/conf.d/celery_proj_worker.conf` file:

```
; ==================================
; celery worker supervisor example
; ==================================
; the name of your supervisord program
[program:projworker]

; Set full path to celery program if using virtualenv
command=/home/me/proj/venv/bin/celery -A proj worker -l
info

; The directory to your Django project
directory=/home/me/sites/proj

; If supervisord is run as the root user, switch users to
this UNIX user account before doing any processing.
user=me

; Supervisor will start as many instances of this program
as named by numprocs
numprocs=1
```

```
; Put process stdout output in this file
stdout_logfile=/var/log/celery/proj_worker.log

; Put process stderr output in this file
stderr_logfile=/var/log/celery/proj_worker.log

; If true, this program will start automatically when
supervisord is started
autostart=true

; May be one of false, unexpected, or true. If false, the
process will never be autorestarted. If unexpected, the
process will be restart when the program exits with an exit
code that is not one of the exit codes associated with this
process’ configuration (see exitcodes). If true, the
process will be unconditionally restarted when it exits,
without regard to its exit code.
autorestart=true

; The total number of seconds which the program needs to
stay running after a startup to consider the start
successful.
startsecs=10

; Need to wait for currently executing tasks to finish at
shutdown. ; Increase this if you have very long running
tasks.
stopwaitsecs = 600

; When resorting to send SIGKILL to the program to
terminate it ; send SIGKILL to its whole process group
instead, taking care of its children as well.
killasgroup=true

; if your broker is supervised, set its priority higher so
it starts first
priority=998
```

Also add `/etc/supervisor/conf.d/celery_proj_beat.conf` file:

```
; ================================
;  celery beat supervisor example
; ================================
; the name of your supervisord program
[program:projbeat]


; Set full path to celery program if using virtualenv
command=/home/me/proj/venv/bin/celery -A proj beat -l info


; The directory to your Django project
directory=/home/me/sites/proj


; If supervisord is run as the root user, switch users to
this UNIX user account before doing any processing.
user=me


; Supervisor will start as many instances of this program
as named by numprocs
numprocs=1


; Put process stdout output in this file
stdout_logfile=/var/log/celery/proj_beat.log


; Put process stderr output in this file
stderr_logfile=/var/log/celery/proj_beat.log


; If true, this program will start automatically when
supervisord is started
autostart=true
```

```
; May be one of false, unexpected, or true. If false, the
process will never be autorestarted. If unexpected, the
process will be restart when the program exits with an exit
code that is not one of the exit codes associated with this
process' configuration (see exitcodes). If true, the
process will be unconditionally restarted when it exits,
without regard to its exit code.
autorestart=true

; The total number of seconds which the program needs to
stay running after a startup to consider the start
successful.
startsecs=10

; if your broker is supervised, set its priority higher so
it starts first
priority=999
```

Although it seems overwhelming, just pay attention on 3 vital things:

1. Full path to Celery program;
2. The Django project directory;
3. And user.

The others are quite safe to be copied blindly.

Don't forget to create empty log files first, so they are registered as valid paths. Using touch commands:

```
$ sudo touch /var/log/celery/proj_worker.log
$ sudo touch /var/log/celery/proj_beat.log
```

Then update change for supervisor:

```
$ sudo supervisorctl reread
$ sudo supervisorctl update
```

Finally we can start/stop/restart the services or even check the status:

```
$ sudo supervisorctl start projworker
$ sudo supervisorctl stop projworker
$ sudo supervisorctl restart projworker
$ sudo supervisorctl status projworker

$ sudo supervisorctl start projbeat
$ sudo supervisorctl stop projbeat
$ sudo supervisorctl restart projbeat
$ sudo supervisorctl status projbeat
```

Summary

It takes quite a lot of configurations just to create a periodic task in Django. As complex as it seems, we can always break it down:

1. Configure Celery in Django by `celery.py` , `settings.py` , and `__init__.py` ;
2. Create `tasks.py` under each corresponding apps;
3. Start the Celery services for both worker and beat; either manually or by daemonization.

Final words, thank you for your time reading this tutorial and hope it helps! :)