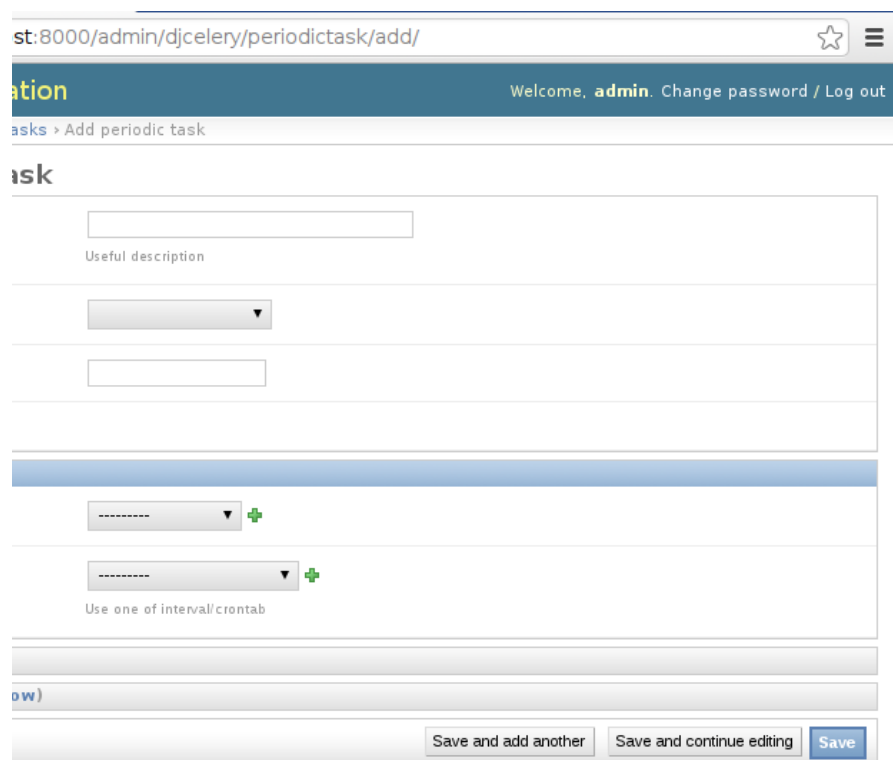


Started Scheduling Tasks with

celery, how-to, scheduling tasks

or

on f



The screenshot shows a Django admin page for adding a periodic task. The URL is `st:8000/admin/djcelery/periodictask/add/`. The page header includes a welcome message for 'admin' and a 'Log out' link. The breadcrumb trail is 'tasks > Add periodic task'. The form has several fields: a text input for 'Useful description', a dropdown menu, a text input, and two more dropdown menus with '+' icons. The bottom of the form has three buttons: 'Save and add another', 'Save and continue editing', and 'Save'.

Started Using Celery for Scheduling

Applications can make good use of being able to schedule work, or just not blocking the request thread.

There are multiple ways to schedule tasks in your Django app, but there are some advantages to using Celery. It's supported, scales well, and works well with Django. Given its wide use, there are lots of resources to help learn and use it. And once learned, that knowledge is likely to be useful on other projects.

Celery versions 3.0.x

This documentation applies to Celery 3.0.x. Earlier or later versions of Celery

Subscribe to our newsletter

youremail@email.com



The purpose of Celery is to allow you to run some code later, or regularly according to a schedule.

Why might this be useful? Here are a couple of common cases.

First, suppose a web request has come in from a user, who is waiting for the request to complete so a new page can load in their browser. Based on their request, you have some code to run that's going to take a while (longer than the person might want to wait for a web page), but you don't really need to run that code before responding to the web request. You can use Celery to have your long-running code called later, and go ahead and respond immediately to the web request.

This is common if you need to access a remote server to handle the request. Your app has no control over how long the remote server will take to respond, or the remote server might be down.

Another common situation is wanting to run some code regularly. For example, maybe every hour you want to look up the latest weather report and store the data. You can write a task to do that work, then ask Celery to run it every hour. The task runs and puts the data in the database, and then your Web application has access to the latest weather report.

A [task](#) is just a Python function. You can think of scheduling a task as a time-delayed call to the function. For example, you might ask Celery to call your function `task1` with arguments `(1, 3, 3)` after five minutes. Or you could have your function `batchjob` called every night at midnight.

We'll set up Celery so that your tasks run in pretty much the same environment as the rest of your application's code, so they can access the same database and Django settings. There are a few differences to keep in mind, but we'll cover those later.

When a task is ready to be run, Celery puts it on a [queue](#), a list of tasks that are ready to be run. You can have many queues, but we'll assume a single queue here for simplicity.

Putting a task on a queue just adds it to a to-do list, so to speak. In order for the task to be executed, some other process, called a [worker](#), has to be watching that queue for tasks. When it sees tasks on the queue, it'll pull off the first and execute it, then go back to wait for more. You can have many workers, possibly on many different servers, but we'll assume a single worker for now.

We'll talk more later about the queue, the workers, and another important

process that we haven't mentioned yet, but that's enough for now, let's do some work.

Installing celery locally

Installing celery for [local use with Django](#) is trivial - just install [django-celery](#):

```
$ pip install django-celery
```

Configuring Django for Celery

To get started, we'll just get Celery configured to use with runserver. For the Celery [broker](#), which we will explain more about later, we'll use a [Django database broker implementation](#). For now, you just need to know that Celery needs a broker and we can get by using Django itself during development (but you **must** use something more robust and better performing in production).

In your Django settings.py file:

1. Add these lines:

```
import djcelery
djcelery.setup_loader()
BROKER_URL = 'django://'
```

The first two lines are always needed. Line 3 configures Celery to use its Django broker.

Important: Never use the Django broker in production. We are only using it here to save time in this tutorial. In production you'll want to use RabbitMQ, or maybe Redis.

2. Add djcelery and kombu.transport.django to INSTALLED_APPS:

```
INSTALLED_APPS = (
    ...
    'djcelery',
    'kombu.transport.django',
    ...
)
```

djcelery is always needed. **kombu.transport.django** is the Django-based broker, for use mainly during development.

3. Create celery's database tables. If using [South](#) for schema migrations:

```
$ python manage.py migrate
```

Otherwise:

```
$ python manage.py syncdb
```

Writing a task

As mentioned before, a task can just be a Python function. However, Celery does

need to know about it. That's pretty easy when using Celery with Django. Just add a `tasks.py` file to your application, put your tasks in that file, and decorate them. Here's a trivial `tasks.py`:

```
from celery import task

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

When `djcelery.setup_loader()` runs from your settings file, Celery will look through your `INSTALLED_APPS` for `tasks.py` modules, find the functions marked as tasks, and register them for use as tasks.

Marking a function as a task doesn't prevent calling it normally. You can still call it: `z = add(1, 2)` and it will work exactly as before. Marking it as a task just gives you additional ways to call it.

Scheduling it

Let's start with the simple case we mentioned above. We want to run our task soon, we just don't want it to hold up our current thread. We can do that by just adding `.delay` to the name of our task:

```
from myapp.tasks import add

add.delay(2, 2)
```

Celery will add the task to its queue ("**worker, please call myapp.tasks.add(2, 2)**") and return immediately. As soon as an idle worker sees it at the head of the queue, the worker will remove it from the queue, then execute it:

```
import myapp.tasks.add

myapp.tasks.add(2, 2)
```

A warning about import names

It's important that your task is always imported and referred to using the **same package name**. For example, depending on how your Python path is set up, it might be possible to refer to it as either `myproject.myapp.tasks.add` or `myapp.tasks.add`. Or from `myapp.views`, you might import it as `.tasks.add`. But Celery has no way of knowing those are all the same task.

`djcelery.setup_loader()` will register your task using the package name of your app in `INSTALLED_APPS`, plus `.tasks.functionname`. Be sure when you schedule your task, you also import it using that same name, or very confusing bugs can occur.

Testing it

Start a worker

As we've already mentioned, a separate process, the **worker**, has to be running to actually execute your Celery tasks. Here's how we can start a worker for our

development needs.

First, open a new shell or window. In that shell, set up the same Django development environment - activate your virtual environment, or add things to your Python path, whatever you do so that you **could** use runserver to run your project.

Now you can [start a worker](#) in that shell:

```
$ python manage.py celery worker --loglevel=info
```

The worker will run in that window, and send output there.

Run your task

Back in your first window, start a Django shell and run your task:

```
$ python manage.py shell
>>> from myapp.tasks import add
>>> add.delay(2, 2)
```

You should see output in the worker window indicating that the worker has run the task:

```
[2013-01-21 08:47:08,076: INFO/MainProcess] Got task from broker: myapp.tasks.add[e080e047-b2a2-43a7-af74-d7d9d98b02fc]
[2013-01-21 08:47:08,299: INFO/MainProcess] Task myapp.tasks.add[e080e047-b2a2-43a7-af74-d7d9d98b02fc] succeeded in 0.183349132538s: 4
```

An Example

Earlier we mentioned using Celery to avoid delaying responding to a web request. Here's a simplified Django view that uses that technique:

```
# views.py

def view(request):
    form = SomeForm(request.POST)
    if form.is_valid():
        data = form.cleaned_data
        # Schedule a task to process the data later
        do_something_with_form_data.delay(data)
    return render_to_response(...)

# tasks.py

@task
def do_something_with_form_data(data):
    call_slow_web_service(data['user'], data['text'], ...)
```

Troubleshooting

It can be frustrating trying to get Celery tasks working, because multiple parts have to be present and communicating with each other. Many of the usual tips still apply:

- Get the simplest possible configuration working first.

- Use the python debugger and print statements to see what's going on.
- Turn up logging levels (e.g. `--loglevel debug` on the worker) to get more insight.

There are also some tools that are unique to Celery.

Eager scheduling

In your Django settings, you can add:

```
CELERY_ALWAYS_EAGER = True
```

and Celery will **bypass the entire scheduling mechanism** and call your code directly.

In other words, with `CELERY_ALWAYS_EAGER = True`, these two statements run just the same:

```
add.delay(2, 2)
add(2, 2)
```

You can use this to get your core logic working before introducing the complication of Celery scheduling.

Peek at the Queue

As long as you're using Django itself as your broker for development, your queue is stored in a Django database. That means you can look at it easily. Add a few lines to `admin.py` in your application:

```
from kombu.transport.django import models as kombu_models
site.register(kombu_models.Message)
```

Now you can go to `/admin/django/message/` to see if there are items on the queue. Each **message** is a request from Celery for a worker to run a task. The contents of the message are rather inscrutable, but just knowing if your task got queued can sometimes be useful. The messages tend to stay in the database, so seeing a lot of messages there doesn't mean your tasks aren't getting executed.

Check the results

Anytime you schedule a task, Celery returns an `AsyncResult` object. You can save that object, and then use it later to see if the task has been executed, whether it was successful, and what the result was.

```
result = add.delay(2, 2)
...
if result.ready():
    print "Task has run"
    if result.successful():
        print "Result was: %s" % result.result
    else:
        if isinstance(result.result, Exception):
            print "Task failed due to raising an exception"
            raise result.result
        else:
            print "Task failed without raising exception"
```

```
else:  
    print "Task has not yet run"
```

Periodic Scheduling

Another common case is running a task on a regular schedule. Celery implements this using another process, [celerybeat](#). Celerybeat runs continually, and whenever it's time for a scheduled task to run, celerybeat queues it for execution.

For obvious reasons, only one celerybeat process should be running (unlike workers, where you can run as many as you want and need).

Starting celerybeat is similar to starting a worker. Start another window, set up your Django environment, then:

```
$ python manage.py celery beat
```

There are several ways to tell celery to run a task on a schedule. We're going to look at [storing the schedules in a Django database table](#). This allows you to easily change the schedules, even while Django and Celery are running.

Add this setting:

```
CELERYBEAT_SCHEDULER = 'djcelery.schedulers.DatabaseScheduler'
```

You can now add schedules by opening the Django admin and going to [/admin/djcelery/periodictask/](#). See the image above for what adding a new periodic task looks like, and here's how the fields are used:

- **Name** — Any name that will help you identify this scheduled task later.
- **Task (registered)** — This should give a choice of any of your defined tasks, as long as you've started Django at least once after adding them to your code. If you don't see the task you want here, it's better to figure out why and fix it than use the next field.
- **Task (custom)** — You can enter the full name of a task here (e.g. `myapp.tasks.add`), but it's better to use the registered tasks field just above this.
- **Enabled** — You can uncheck this if you don't want your task to actually run for some reason, for example to disable it temporarily.
- **Interval** — Use this if you want your task to run repeatedly with a certain delay in between. You'll probably need to use the green "+" to define a new schedule. This is pretty simple, e.g. to run every 5 minutes, set "Every" to 5 and "Period" to minutes.
- **Crontab** — Use [crontab](#), instead of [Interval](#), if you want your task to run at specific times. Use the green "+" and fill in the minute, hour, day of week,

day of month, and day of year. You can use "*" in any field in place of a specific value, but be careful - if you use "*" in the Minute field, your task will run every minute of the hour(s) selected by the other fields. Examples: to run every morning at 7:30 am, set Minute to "30", Hour to "7", and the remaining fields to "*".

- **Arguments** — If you need to pass arguments to your task, you can open this section and set `*args` and `**kwargs`.
- **Execution Options** — Advanced settings that we won't go into here.

Default schedules

If you want some of your tasks to have default schedules, and not have to rely on someone setting them up in the database after installing your app, you can use Django fixtures to provide your schedules as [initial data](#) for your app.

- Set up the schedules you want in your database.
- Dump the schedules in json format:

```
$ python manage.py dumpdata djcelery --indent=2 --exclude=djcelery.taskmeta >filename.json
```

- Create a fixtures directory inside your app
- If you never want to edit the schedules again, you can copy your json file to `initial_data.json` in your fixtures directory. Django will load it every time syncdb is run, and you'll either get errors or lose your changes if you've edited the schedules in your database. (You can still add new schedules, you just don't want to change the ones that came from your initial data fixture.)
- If you just want to use these as the initial schedules, name your file something else, and load it when setting up a site to use your app:

```
$ python manage.py loaddata <your-app-label/fixtures/your-filename.json
```

Hints and Tips

Don't pass model objects to tasks

Since tasks don't run immediately, by the time a task runs and looks at a model object that was passed to it, the corresponding record in the database might have changed. If the task then does something to the model object and saves it, those changes in the database are overwritten by older data.

It's almost always safer to save the object, pass the record's key, and look up the object again in the task:

```
myobject.save()
mytask.delay(myobject.pk)
```

...


```
@task
def mytask(pk):
    myobject = MyModel.objects.get(pk=pk)
    ...
```

Schedule tasks in other tasks

It's perfectly all right to schedule one task while executing another. This is a good way to make sure the second task doesn't run until the first task has done some necessary work first.

Don't wait for one task in another

If a task waits for another task, the first task's worker is blocked and cannot do any more work until the wait finishes. This is likely to lead to a deadlock, sooner or later.

If you're in Task A and want to schedule Task B, and after Task B completes, do some more work, it's better to create a Task C to do that work, and have Task B schedule Task C when it's done.

Next Steps

Once you understand the basics, parts of the Celery User's Guide are good reading. I recommend these chapters to start with; the others are either not relevant to Django users or more advanced:

- [Tasks](#)
- [Periodic Tasks](#)

Using Celery in production

The Celery configuration described here is for convenience in development, and should never be used in production.

The most important change to make in production is to stop using `kombu.transport.django` as the broker, and switch to [RabbitMQ](#) or something equivalent that is robust and scalable.



Show Comments

Sorry, the browser you are using is not currently supported. Disqus actively supports the following browsers:

- [Firefox](#)
- [Chrome](#)
- [Internet Explorer 11+](#)
- [Safari](#)



[Ryan Prater](#) • 1 year ago

Thanks! The "Don't pass model objects to a task" is important and often left out of articles like this. It's worth noting that django-celery is now deprecated, so it could be worth the time to update or create a new article with respective instructions - you've gotten great feedback!



[Eutychus Towett](#) • 1 year ago

Awesome post. It made me understand what celery is and how it works!



[James Gardiner](#) • 1 year ago

You might also like:

Posts:

Shiplt Day Recap Q2 2017

Technical > May 3, 2017

Building a Custom Block Template Tag

Technical > May 1, 2017

Digging Into Django QuerySets

Technical > April 5, 2017

Resources:

Setting Up JupyterHub for Distance Learning

Talk

You Belong with Me: Scraping Taylor Swift Lyrics with Python and Celery

Talk

Django 1.1 Testing and Debugging

Book

Get in touch

Contact

solutions@cactusgroup.com

T 919-951-0052

F 919-928-5516

108 Morris St, Suite 2
Durham, NC 27701

1111 Light St. 4th Flr.
c/o Betamore
Baltimore, MD 21230

Quick links

[Services](#)

[Our work](#)

[About us](#)

[Blog](#)

[Careers](#)

[Events](#)

[Talks](#)

[Press](#)

Monthly newsletter

youremail@email.com



Get Django tips, learn about our clients, and stay up to date on Cactus news.

[Privacy Policy](#)

Stay connected

