Learning Objectives

- **Define a task signature**
- **Create a periodic task**
- **Differentiate between interval, crontab, solar, and clocked schedules**
- **Explain how Celery Beat works**
- **Schedule tasks that run on an interval or on a specified date/time**

Clone Project 4 Repo

Clone Project 4 Repo

Before we continue, you need to clone the `course4_proj` repo so you have all of your code. You will need the SSH information for your repo.

In the Terminal

- Clone the repo. Your command should look something like this:

```
git clone git@github.com:<your_github_username>/course4_proj.git
```

- You should see a `course4_proj` directory appear in the file tree.

You are now ready for the next assignment.

Scheduled and Periodic Tasks

Scheduled and Periodic Tasks

It's often useful to be able to run certain tasks at specific dates and times, or at periodic intervals. We might want to be able to clean up old data every month, trigger an email of a summary of events in the past week, or send daily happy birthday emails to users.

You can use a system tool like `cron` to schedule tasks, whether one-off or periodic. This is tied into your OS and so can only execute commands that can be triggered from the command line (i.e. you can't just tell `cron` to run a Python function, however it can run a Python script).

Sometimes you might want finer-grained control over task execution, or not have access to `cron` in your environment, or perhaps just want the task to be dispatched to a group of workers rather than run on a single server. These are some of the reasons for using *celery beat*

Celery beat is a scheduler that starts Celery tasks. There are two ways tasks can be added to the scheduler: by calling the `add_periodic_task()` method on the Celery app, or by adding tasks to a backend (for example, the Django database) which the scheduler reads from periodically. When it's time to execute the task, Celery beat triggers a task to be executed by a Celery worker. Thus to use Celery beat, you must run a "beat" process as well as worker process(es).

Since we're only going to be adding tasks using the Django database (with the *django-celery-beat* package) we'll just take a brief look at `add_periodic_task()` method, but first we need to make a short detour to discuss task *signatures*.

Task Signatures

In general, to call a function or method, you need both a reference to the function, and the arguments to pass to it. Celery provides a `Signature` object that combined the task and arguments into a single object. The `Signature` can then be executed, for example, by using `delay()`.

For example, here's a function that accepts a function and its arguments as arguments, then calls it with `delay()`:

```python
def function_executor(fn, *args, **kwargs):
    return fn.delay(args, **kwargs)
```

So we could do something like:

```
function_executor(notify_of_new_search_term, "lord of the
        rings")
```

We could construct a function that works on `Signature` objects instead:

```
def signature_executor(s):
    return s.delay()
```

And use it like this:

```
task_signature = notify_of_new_search_term.s("lord of the
        rings")

signature_executor(task_signature)
```

The result is the same, but passing around `Signatures` can be more convenient than managing multiple variables representing the same information. Furthermore, signatures are used in `add_periodic_task()`, which we'll return to now.

Add Periodic Task

The `add_periodic_task()` method has two required arguments:

- `schedule`: This can be an integer (in seconds), a `timedelta`, or a schedule object (more on them soon)
- `sig`: The task signature

A `name` for the scheduled task can also be provided as a keyword argument.

▼ **Documentation**
There are other more complex options, which you can read on the official documentation.

For the `schedule` argument, if an integer (number of seconds) or `timedelta` is passed, then the task will be executed repeatedly, with a delay of the time period specified. For example, this would schedule a `search_and_save` with the argument `star wars` every minute:

```
app.add_periodic_task(60.0, search_and_save.s("star wars"))
```

There are other types of schedules that can be used. Celery provides `crontab` and `solar` schedules. `crontab` schedule (`celery.schedules.crontab`) allows setting up schedules based on crontab style expressions. We mentioned `cron` earlier, and *crontab* is the format for specifying its schedule. The full ins and outs of the crontab format are beyond the scope of this course. Basically, rules can be set to specify which minute, hour, day of week, day of month, and/or month. If a rule is not specified for a given time unit, then the task will execute for any time it's valid for the other units.

For example, a task specified to execute at minute `0` with no other rules, would start on the hour, every hour, every day of the year. A task specified to execute at hour `10`, with no other rules, would execute at every minute of 10 AM (10:00, 10:01, 10:02, up to 10:59), on every day of the year. For this reason crontabs usually contain values for more than one unit.

Here's an example that executes `notify_of_new_search_term` at 7:30 AM on the first of the month, every month:

```
from celery.schedules import crontab

app.add_periodic_task(crontab(
    minute=30,
    hour=7,
    day_of_month=1
), search_and_save.s("star wars"))
```

The Celery Crontab schedules documentation has some more examples of various crontab rules that can be used.

The `solar` schedule (`celery.schedules.solar`) allows events to be scheduled based on solar events such as sunrise, sunset and solar noon. It even supports civil, nautical and astronomical times for both dawn and dusk. These are calculated based on a given latitude and longitude.

For example, here's how to schedule a task that executes at sunrise in Auckland, New Zealand.

```
from celery.schedules import solar

app.add_periodic_task(
    solar("sunrise", -36.848461, 174.763336),
    search_and_save.s("star wars")
)
```

Since solar scheduling is somewhat of a niche application, we'll defer to the official documentation for all the options.

Now let’s look at how to set up and schedule tasks with *django-celery-beat*.

Django Celery Beat

Django Celery Beat

django-celery-beat works with the Django database or cache as a store for scheduled tasks (although we'll just use it with the database). It works by repeatedly querying the database looking for new, updated or deleted scheduled tasks. It then schedules them in Celery beat for execution at the correct time.

django-celery-beat provides a number of models. All of these are in the module `django_celery_beat.models`

Periodic Task

This model is how *django-celery-beat* refers to a Celery task. It stores the task and arguments, so if you wanted to execute the same task with different arguments, you would need to add multiple `PeriodicTasks`. It also has fields for enabling or disabling the task (`enabled`), setting when the task should start running (`start_time`) and whether it should be run multiple times or just once (`one_off`). It also stores the last time the task was executed (`last_run_at`). There are a few other fields related to scheduling that we won't go into. For example, if you have multiple queues of different priority, you can set which queue/priority the `PeriodicTask` should have.

Before a `PeriodicTask` can be scheduled it must be associated with a schedule object. The schedule for the `PeriodicTask` is set on one of the fields `interval`, `crontab`, `solar` or `clocked`, which are related to the models `IntervalSchedule`, `CrontabSchedule`, `SolarSchedule` or `ClockedSchedule`, respectively. Exactly one of these fields must be set, with the rest set to `null`. Let's look at these models in more depth.

Interval Schedule

This is used when you need to run a `PeriodicTask` as a set interval. It has two fields:

- `period`: This is the unit to schedule at, which can be one of `IntervalSchedule.DAYS`, `IntervalSchedule.HOURS`, `IntervalSchedule.MINUTES`, `IntervalSchedule.SECONDS` or `IntervalSchedule.MICROSECONDS`.
- `every`: The number of intervals of `period` to wait between executions.

For example, to run a task every 30 minutes, create an `IntervalSchedule` like this:

```
schedule = IntervalSchedule(period=IntervalSchedule.MINUTES,
        every=30)
```

Crontab Schedule

Used to define a schedule with crontab rules. The fields are `minute`, `hour`, `day_of_week`, `day_of_month`, `month_of_year`. Each field defaults to `*`, which means run on any occurrence of that unit.

Like our previous crontab example above, here's a schedule for a task at 7:30 AM on the first of each month.

```
schedule = CrontabSchedule(minute="30", hour="7",
        day_of_month="1")
```

Note that these are `CharField`s, so that special crontab rules can be used. For example `minute="*/15"` to run every 15 minutes, or `hour="1,3,5,7,9,11,13,15,17,19,21,23"` to run only on odd hours, or `day_of_month="1-7"` to run on the first week of the month. Once again we defer to crontab specific documentation instead of going into the full detail here; Crontab Guru is a good site for experimenting with crontab expressions

Solar Schedule

For scheduling tasks based on solar events, at a given latitude and longitude. Its fields are `event` (`CharField`), latitude (`DecimalField`), and longitude (`DecimalField`). The valid values for `event` (the solar event type) are:

- `dawn_astronomical`
- `dawn_civil`
- `dawn_nautical`
- `dusk_astronomical`
- `dusk_civil`
- `dusk_nautical`
- `solar_noon`
- `sunrise`
- `sunset`

An explanation for what these different events are can be found in the official documentation.

Clocked Schedule

This is a very simple schedule: it just runs the task at the time set by the `clocked_time` (`DateTimeField`) field. The `PeriodicTask` would therefore only run once.

Scheduling Periodic Tasks

Now that we've introduced the models, let's look at how to set up and schedule a `PeriodicTask`. First an instance of one of our schedule models is required. For example, an `IntervalSchedule` which executes once a day.

```
from django_celery_beat.models import IntervalSchedule


day_schedule, created =
        IntervalSchedule.objects.get_or_create(period=IntervalSchedule.DAYS,
        every=1)
```

Then create a `PeriodicTask`. The `task` field stores the importable module path of the task. For example, our tasks would have the path `movies.tasks.search_and_save` and `movies.tasks.notify_of_new_search_term`.

The `PeriodicTask` stores the `args` and `kwargs` that are passed to the task on `TextFields` with those same names. They are set as a JSON encoded `list` or `dict`, respectively. This means that you can only pass arguments that can be JSON encoded and decoded (`lists`, `dicts`, `str` and number type). If your task requires arguments of more complex types, you will have to change their signature so they are able to parse or reconstruct arguments from these simple types.

A `name` for the `PeriodicTask` can also be provided to help keep track of what it does.

Then, let's say we wanted to keep our database of movies about *python* up to date by searching for them every day. Here's how the `PeriodicTask` would be set up:

```
from django_celery_beat.models import PeriodicTask
import json

args = json.dumps(["python"])

pt = PeriodicTask.objects.create(
    name="Daily python movie search",
    interval=day_schedule,
    args=args,
    task="movies.tasks.search_and_save"
)
```

Note that we're using the `interval` field since we created an `IntervalSchedule`. If we'd created a `CrontabSchedule` we'd save it on the `crontab` field, and likewise with the other schedule types.

That’s all that needs to be done, the `PeriodicTask` is now scheduled and ready to run. Next we’ll see how to run the Celery beat process to start the task at the right time.

Celery Beat Process

Celery Beat Process

Starting Celery beat is similar to starting a Celery worker. In fact, we just start `celery` with the `beat` command instead of `worker`:

```
$ celery -A course4_proj beat -l INFO
celery beat v5.1.2 (sun-harmonics) is starting.
__    -    ... __   -        _
LocalTime -> 2021-10-01 01:51:08
Configuration ->
    . broker -> redis://localhost:6379/0
    . loader -> celery.loaders.app.AppLoader
    . scheduler -> celery.beat.PersistentScheduler
    . db -> celerybeat-schedule
    . logfile -> [stderr]@%INFO
    . maxinterval -> 5.00 minutes (300s)
[2021-10-01 01:51:08,957: INFO/MainProcess] beat: Starting...
```

However, if we did this, we'd only run tasks that have been scheduled by calling the `add_periodic_task()` method. If we want Celery beat to read the schedule from the Django database (or a different schedule store) we need to use the `--scheduler` argument. To use *django-celery-beat* we'll specify the scheduler `django_celery_beat.schedulers:DatabaseScheduler`:

```
$ celery -A course4_proj beat -l INFO --scheduler
        django_celery_beat.schedulers:DatabaseScheduler
celery beat v5.1.2 (sun-harmonics) is starting.
__    -    ... __   -        _
LocalTime -> 2021-10-01 01:53:19
Configuration ->
    . broker -> redis://localhost:6379/0
    . loader -> celery.loaders.app.AppLoader
    . scheduler ->
        django_celery_beat.schedulers.DatabaseScheduler

    . logfile -> [stderr]@%INFO
    . maxinterval -> 5.00 seconds (5s)
[2021-10-01 01:53:19,169: INFO/MainProcess] beat: Starting...
```

As we mentioned, Celery beat doesn’t actually execute the tasks, instead it triggers the task (adds them to the task queue) for a Celery worker to run. Therefore, a Celery worker needs to be started in a new terminal (in the usual way) to perform the execution.

Now you’ll set up Celery beat and *django-celery-beat*, then schedule a task.

Try It Out

Try It Out

First `django-celery-beat` needs to be installed, using `pip`:

```
pip3 install django-celery-beat
```

Then, `django_celery_beat` must be added to `INSTALLED_APPS` in `settings.py`.

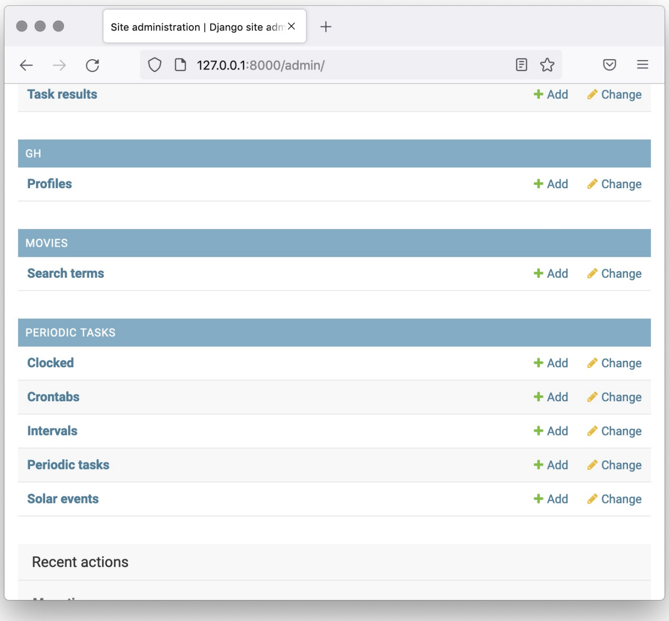
Open settings.py

Next, run the `migrate` management command with `manage.py` to set up the database tables for *django-celery-beat*.

```
python3 manage.py migrate
```

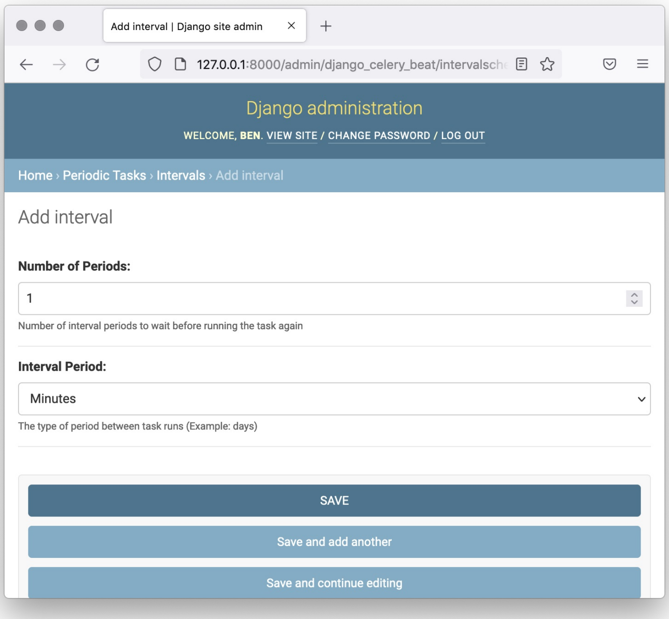
Instead of setting up the `PeriodicTasks` programmatically, we'll do it through Django admin, so start the Django dev server and log in to the Django Admin. You'll see the *Periodic Tasks* section on the main page.

View Admin Panel



periodic tasks section

Go into the *Intervals* section and choose **Add Interval**. We'll create an `IntervalSchedule` with a small delay just for testing, so enter `1` in the **Number of Periods** field and select **Interval Period** of *Minutes*. Click **Save**.


Add interval | Django site admin

127.0.0.1:8000/admin/django_celery_beat/intervalsch

Django administration

WELCOME, BEN. VIEW SITE / CHANGE PASSWORD / LOG OUT

Home › Periodic Tasks › Intervals › Add interval

Add interval

Number of Periods: 1

Number of interval periods to wait before running the task again

Interval Period: Minutes

The type of period between task runs (Example: days)

add interval

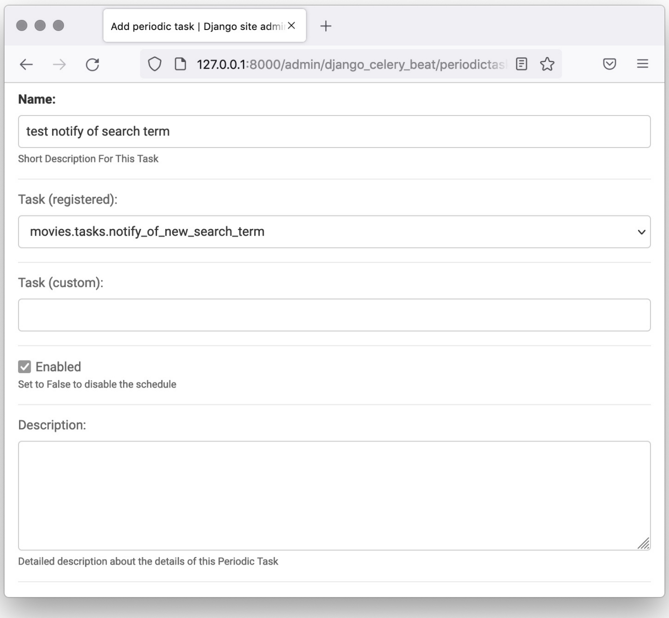
You'll now see it under the list of intervals as *every minute*.

Now return to the *Periodic Tasks* section of the Django admin and choose **Add** next to *Periodic tasks*.

We'll just set up a `PeriodicTask` that calls `notify_of_new_search_term()` every minute so that we see some output in the Celery console.

The *Name* you enter doesn't matter, call it something like *test notify of search term*.

Since our tasks have been registered to the Celery app, it will display them in the list under *Task (registered)*. Select **movies.tasks.notify_of_new_search_term**.


Add periodic task | Django site admin

127.0.0.1:8000/admin/django_celery_beat/periodictas

Name: test notify of search term

Short Description For This Task

Task (registered): movies.tasks.notify_of_new_search_term

Task (custom):

☑ Enabled

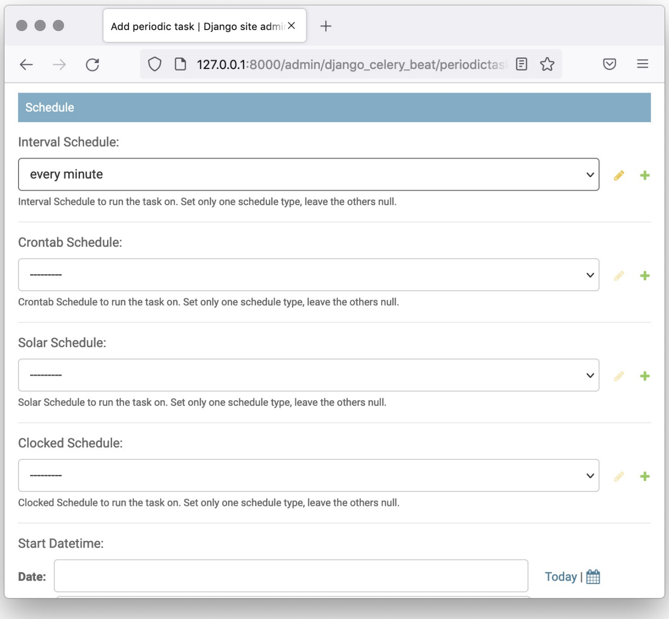
Set to False to disable the schedule

Description:

Detailed description about the details of this Periodic Task

periodic task name and task

Scroll down to the *Schedule* section. Select the **every minute** schedule under *Interval Schedule*.


Add periodic task | Django site admin

127.0.0.1:8000/admin/django_celery_beat/periodictas

Schedule

Interval Schedule: every minute

Interval Schedule to run the task on. Set only one schedule type, leave the others null.

Crontab Schedule: ---------

Crontab Schedule to run the task on. Set only one schedule type, leave the others null.

Solar Schedule: ---------

Solar Schedule to run the task on. Set only one schedule type, leave the others null.

Clocked Schedule: ---------

Clocked Schedule to run the task on. Set only one schedule type, leave the others null.

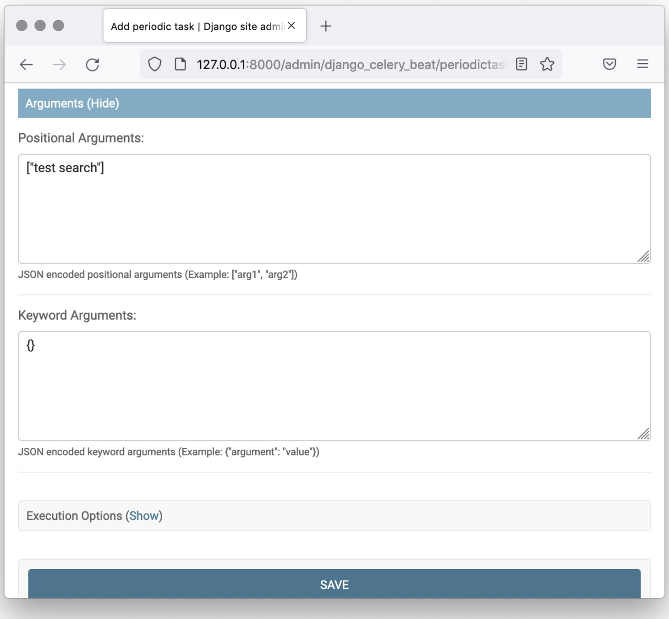
Start Datetime:

Date: Today |

periodic task interval selected

The last thing we need to enter is an argument to be passed to the task. Let’s just call it with the string `"test search"`. Expand the **Arguments** section and enter this JSON string in *Positional Arguments*:

```
["test search"]
```


Add periodic task | Django site admi

127.0.0.1:8000/admin/django_celery_beat/periodictas

Arguments (Hide)

Positional Arguments:

```
["test search"]
```

JSON encoded positional arguments (Example: ["arg1", "arg2"])

Keyword Arguments:

```
{}
```

JSON encoded keyword arguments (Example: {"argument": "value"})

Execution Options (Show)

SAVE

periodic tasks arguments

That’s all we need to enter to set up a task, so scroll to the bottom of the page and click **Save**.

Now in a new terminal, start up the Celery worker process:

Open a terminal

```
celery -A course4_proj worker -l DEBUG
```

Your output should look something like this:

```
celery -A course4_proj worker -l INFO


 -------------- celery@BensMBP.local v5.1.2 (sun-harmonics)
--- ***** -----
-- ******* ---- macOS-10.14.6-x86_64-i386-64bit 2021-10-01
        08:25:15
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app:         course4_proj:0x108296bb0
- ** ---------- .> transport:   redis://localhost:6379/0
- ** ---------- .> results:
- *** --- * --- .> concurrency: 8 (prefork)
-- ******* ---- .> task events: OFF (enable -E to monitor tasks
        in this worker)
--- ***** -----
 -------------- [queues]
                .> celery           exchange=celery(direct)
        key=celery



[tasks]
  . movies.tasks.notify_of_new_search_term
  . movies.tasks.search_and_save
... etc
```

Then in another terminal, start the Celery beat process.

info

Opening a Second Terminal

To open a second terminal, click *Tools* in the Codio menu bar. Then click *Terminal*. Change directories to `course4_proj`.

```
cd course4_proj
```

Then enter the following command in the terminal to launch the beat process.

```
celery -A course4_proj beat -l INFO --scheduler
        django_celery_beat.schedulers:DatabaseScheduler
```

You should see output similar to the following:

```
celery beat v5.1.2 (sun-harmonics) is starting.
__    -    ... __   -        _
LocalTime -> 2021-10-01 08:25:56
Configuration ->
    . broker -> redis://localhost:6379/0
    . loader -> celery.loaders.app.AppLoader
    . scheduler ->
        django_celery_beat.schedulers.DatabaseScheduler

    . logfile -> [stderr]@%INFO
    . maxinterval -> 5.00 seconds (5s)
[2021-10-01 08:25:56,411: INFO/MainProcess] beat:
        Starting...
```

Interval tasks start executing from either the start time of the Celery beat process or the last execution time if they've been run before. In our case, we have to wait a minute before the task is first executed. You should see the output in the Celery beat terminal when it's started:

```
[2021-10-01 08:28:26,635: INFO/MainProcess] Scheduler: Sending
        due task test notify of search term
        (movies.tasks.notify_of_new_search_term)
```

And then if you switch to the Celery worker terminal, you'll see the email output:

```
[2021-10-01 08:28:26,656: INFO/MainProcess] Task
        movies.tasks.notify_of_new_search_term[7d39a0b7-f97c-
        4880-9a68-8dd6cda8b6b7] received
[2021-10-01 08:28:26,903: WARNING/ForkPoolWorker-8] Content-
        Type: text/plain; charset="utf-8"
MIME-Version: 1.0
Content-Transfer-Encoding: 7bit
Subject: [Django] New Search Term
From: root@localhost
... etc
```

Wait another minute, and the process will be kicked off again, and you'll see the email output once more.

Once you've see it in execute a couple of times, you can stop the worker and beat processes by typing **Control-C** in each of the terminals.

Wrap-Up

As you've seen, Celery beat is a flexible way of scheduling tasks with Celery, and its Django integration makes it easy to dynamically create, modify and disable tasks with a GUI. As with the Celery worker, if you want the beat process to run automatically at system startup you will need to use something like `supervisord`, `systemd` or `init` to execute it.

That brings us to the end of this module about Celery, and to the end of the learning portions of the courses. In the next two modules you'll work on the capstone project Movie Night, which will bring together all of the things you've learned so far.

Pushing to GitHub

Pushing to GitHub

Before continuing, you must push your work to GitHub. In the terminal:

- Commit your changes:

```
git add .
git commit -m "Finish scheduling with celery"
```

- Push to GitHub:

```
git push
```