Learning Objectves

- **Explain the benefits of Celery**
- **Differentiate between Celery and Redis**
- **Register a Celery task**
- **Fetch a completed task**
- **View task results in the Admin GUI**

Clone Course 4 Project Repo

Clone Course 4 Project 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.

Intro

Background Tasks

When running a complex web application, sometimes we need to do things that take a little while to execute. Some examples are fetching data from a remote API (e.g. movie searching), sending out emails to many users, running a reindex on a search database, and more.

Trying to run long operations in the context of a webserver can be troublesome. Usually execution times of web processes are limited to 30, 60, 90 or 120 seconds (although they can be just about any arbitrary time). Any process you try to start that takes longer than the timeout might be cut short, and fail to complete. Furthermore, your users will be kept waiting until the operation completes, so they could be sitting in front of their browsers for two minutes wondering if everything is working. It's not a good experience!

This is where Celery comes in. It's a library that is used to run tasks outside of the main web process. At a high level, tasks are put into a *queue* (the time to write the task to the queue is fast). The main webserver process can then continue, and return a response to the client. Meanwhile, a *broker* reads from the queue and passes the task on to *worker(s)* that process/execute the task and puts the results back into a queue or database. When a task has queued a reference to the task is returned. Later, we can use this reference to get the result of the task. For example, we could return the reference to the browser and it could make requests to an endpoint to check the result is available, which means the task has finished. Sometimes we don't need to use the task reference, we just assume that the task will complete successfully and exceptions will be logged to the usual places on failure.

Let's see how to set up Celery.

Celery Setup

Celery Setup

Before setting up Celery, let's first look at broker selection.

Broker Selection

Celery supports a number of brokers for queueing messages. You can choose to run your own backend queue software like Redis or RabbitMQ, or use a cloud service like Amazon SQS. Each particular service will have its own pros and cons. If there's one you or your organization is already familiar with then you're probably best sticking with it.

We're going to use Redis as the broker, and the *django-celery-results* project to store the task results. This will allow us to query task results in using standard ORM methods and see them in the Django admin.

▼ **Redis**
Redis also works as a key-value store/cache as well as having queueing features for message brokering. You might consider using Redis as your general cache for Django if you're using it for your message brokering as well.

Try It Out:

Before we start configuring Celery in our Django project, let's get Redis installed. Use the `apt` command to install it into your Codio environment.

```
sudo apt install -y redis
```

It will install and start automatically. To check that it's working, you can issue it the `ping` command using the `redis-cli` tool.

```
redis-cli ping
```

You should get a `PONG` result:

```
PONG
```

Now let's look at how to install and set up Celery in our project.

Installation and Configuration

Celery is installed like most Python libraries, using `pip`. To store the results in the Django database, `django-celery-results` is also required. We'll also need the `redis` package for communication with the Redis server. They can all be installed at the same time:

```
pip3 install celery django-celery-results redis
```

Then we need to make some changes to `settings.py`.

Open settings.py

First `django_celery_results` must be added to `INSTALLED_APPS`. Then we can start adding settings for Celery. Celery has a number of settings available that can be configured in different ways, including being read from `settings.py`. The convention is that the Celery setting name is uppercased and prefixed with `CELERY_`.

For example, the `broker_url` setting would be defined in `settings.py` as `CELERY_BROKER_URL`.

The two settings that we'll use are `CELERY_RESULT_BACKEND` and `CELERY_BROKER_URL`

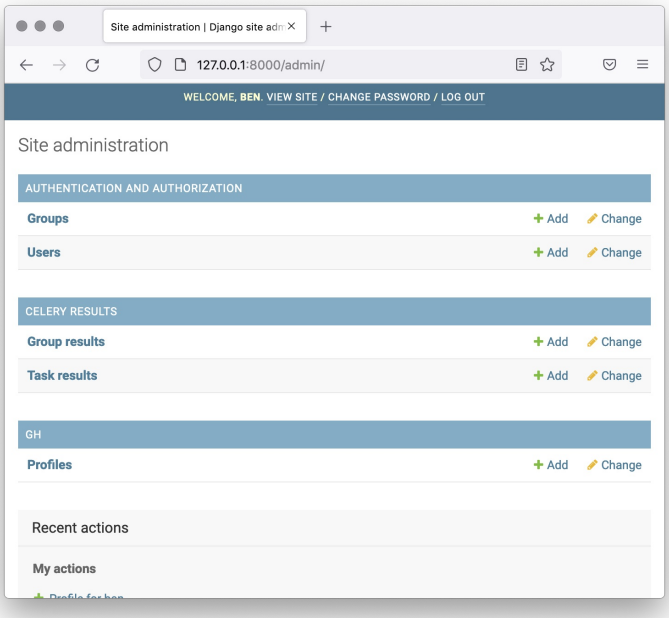
```
    CELERY_RESULT_BACKEND = "django-db"
    CELERY_BROKER_URL = "redis://localhost:6379/0"
```

▼ **Celery configuration options**
The full list of Celery configuration options can be found at the Celery Configuration Documentation.

After the configuration is complete the Celery database models need to be set up with the `migrate` management command:

```
python3 manage.py migrate
```

After this is done, the *Celery Results* models are available in the Django admin.



celery results

Validate your changes by starting the Django dev server and checking that you can see the *Celery Results* section, as per the image above.

View Project

Next we'll look at how to execute tasks.

Executing Celery Tasks

Executing Celery Tasks

To use Celery with Django you must first define an instance of the Celery library, called an *app*. Normally this is done in a file called `celery.py` inside your project module directory (the same directory as `settings.py`).

▼ **celery.py**
Think of this file kind of like `wsgi.py`. WSGI compatible web servers can load the `application` variable from `wsgi.py` to run. Celery knows how to load the `app` variable from `celery.py` to start the Celery worker.

The Celery documentation provides an example `celery.py` file that we can use as a basis for ours, however we need to make some changes so it works with Django Configurations.

We'll take a look at the file that we'll be using, and then go through and explain how it works:

```
import os

from celery import Celery
from django.conf import settings

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
        "course4_proj.settings")
os.environ.setdefault("DJANGO_CONFIGURATION", "Dev")

import configurations

configurations.setup()

app = Celery("course4_proj")
app.config_from_object("django.conf:settings",
        namespace="CELERY")
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

The `app` variable is an instance of the `celery.Celery` class, so we need to import that, among other modules that you would have seen before. Then, we need to set up the environment variables for Django. First

`DJANGO_SETTINGS_MODULE` for the Django settings and then `DJANGO_CONFIGURATION` so that Django Configurations knows which settings class to use.

We then import `configurations` and run `configurations.setup()`, to enable Django Configurations. We'll get an exception without this step.

Then we instantiate the `Celery` class with a name – in this case we use the project name. We call `config_from_object()` on the instance, and tell it to load the settings from the `settings` variable in the `django.conf` module. The `namespace` argument essentially means the prefix for the Celery settings, e.g. `broker_url` comes from `CELERY_BROKER_URL`.

Finally we call `autodiscover_tasks()` on the `app`. This will go through our `INSTALLED_APPS` and look inside the files `tasks.py` and `models.py` for each of them, loading any tasks it finds. We'll talk about what makes a *task* soon.

The last piece of configuration is to define `celery_app` as part of the `__all__` contents of the project model. This will allow Celery to find the `app` variable by importing `celery_app` from the Django project and allow us to use a special `shared_task` decorator - which we'll also cover soon.

The contents of `__init__.py` in the project directory should be:

```
from .celery import app as celery_app

__all__ = ("celery_app",)
```

The Celery worker can then be started, with this command:

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

Where:

- `-A` is the application argument, in this case we want to import it from the `course4_proj` module.
- `worker` is the command to run, which means start a worker instance
- `-l` sets the log level, we set it to INFO. You could also use DEBUG, WARN, ERROR, etc.

Upon running this command the Celery worker should start 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 2021-09-27
        02:36:58
- *** --- * ---
- ** ---------- [config]
- ** ---------- .> app:         course4_proj:0x1039993d0
- ** ---------- .> 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]
```

And a lot more output. It should finish with a line like:

```
[2021-09-27 02:37:00,390: INFO/MainProcess] celery@BensMBP.local
ready.
```

Indicating that the worker has started successfully and is ready to execute tasks.

The worker can be stopped by typing **Control-C** in the terminal. Next we'll get Celery configured in your `course4_proj` project, and then add some tasks.

Try It Out

Start by creating a file called `celery.py` in the `course4_proj` module directory. Insert this content:

```
import os

from celery import Celery
from django.conf import settings

os.environ.setdefault("DJANGO_SETTINGS_MODULE",
        "course4_proj.settings")
os.environ.setdefault("DJANGO_CONFIGURATION", "Dev")

import configurations

configurations.setup()

app = Celery("course4_proj")
app.config_from_object("django.conf:settings",
        namespace="CELERY")
app.autodiscover_tasks(lambda: settings.INSTALLED_APPS)
```

Then open `__init__.py` in the same directory and insert this content:

Open **init**.py

```
from .celery import app as celery_app

__all__ = ("celery_app",)
```

Then you can try out Celery by running this command:

Open the terminal

```
celery -A course4_proj worker -l INFO
```

Confirm that you get similar output as was shown above, finishing with the `ready` message.

Then, type **Control-C** to stop the Celery worker.

Next we'll see how to define and run Celery tasks.

Defining and Running Celery Tasks

Defining and Running Celery Tasks

Just about any function can be turned into a Celery task. All you need to do is decorate it (and then call it a bit differently).

For a function to be turned into a task it must be registered to the Celery app. The `app` instance has a `task` method that's used a decorator to do this (`@app.task`). For example, we could do something like this:

```
from course4_proj.celery import app


@app.task
def my_long_running_function(arg1, arg2):
    # do something
```

However, with Django, this is not the preferred method of registering tasks. Sometimes you might be writing tasks inside Django apps that are reused by different projects. For example, we could open-source our `movies` app and make it available to the community. When installed by a third party, their Django project would not be called `course4_proj` and so trying to import from `course4_proj` would fail for them.

For this reason, Celery provides a special decorator called `shared_task` (`celery.shared_task`). This will register the task to the app defined by `celery_app` in `__init__.py`.

Our example above would become this:

```
from celery import shared_task


@shared_task
def my_long_running_function(arg1, arg2):
    # do something
```

Decorated functions now have a new attribute: `delay()`. This is used to send the function off to be run in the Celery worker rather than the current process.

Normally a function would be called like this:

```
my_long_running_function("arg1", 2)
```

To have Celery execute the function, it's called like this:

```
my_long_running_function.delay("arg1", 2)
```

This will return immediately, so how do we get the result if a function returns something? A *delayed* function actually returns an `AsyncResult` instance (`celery.result.AsyncResult`). The `AsyncResult` has a `get()` method which, when called, waits for the result of the tasks and returns it.

Therefore these two pieces of code will result in the same output; first the "normal way":

```
value = my_long_running_function("arg1", 2)
print(value)
```

And the "Celery way":

```
res = my_long_running_function.delay("arg1", 2)
value = res.get()
print(value)
```

When we're looking at this second block of code, we can see it's obvious how to fetch the result if you have access to the returned `AsyncResult`. But you're essentially making asynchronous code synchronous again. How can we come back later and access the result? We'll look at that in the next section.

Fetching Results Later

Fetching Results Later

Each `AsyncResult` has an `id` attribute containing a UUID identifying the task. We can store a reference to this UUID and later use it to retrieve the result. The simplest method to fetch a task result is to use the `app.AsyncResult` class alias. It takes a UUID and uses the Celery app's settings to figure out where to fetch the result from (i.e. in which backend the results are stored).

Fetching the ID of an `AsyncResult` is easy:

```
res = my_long_running_function.delay("arg1", 2)
task_id = res.id
```

Assuming that `task_id` has been stored somewhere, later on the `AsyncResult` can be retrieved like this:

```
from course4_proj.celery import app  # get a reference to the
        celery app

res = app.AsyncResult(task_id)  # load the AsyncResult
value = res.get()  # get the value
```

Get takes a `timeout` argument which defaults to `None` if not passed in. This means the method will wait forever for a result, blocking the rest of your code from running. The `timeout` argument can be provided, and indicates the number of seconds that `get()` should block for. If no result is received before the timeout, then `celery.exceptions.TimeoutError` is raised. By using this argument, you can wait for a result for a short amount of time, then give the user some feedback that the result is not ready, and depending on your use case, you can try again or tell the user to try again.

For example, here's how you could loop and keep waiting two seconds for the result of the task until it was received. For every loop, print out a message that the result is still pending.

```
from celery.exceptions import TimeoutError


res = my_long_running_function.delay("arg1", 2)

while True:
    try:
        value = res.get(timeout=2)  # wait two seconds
        break  # once we have a value, break from the loop
    except TimeoutError:
        # print a message if no result after two seconds, then
        continue looping
        print("Task not finished after another two seconds,
        waiting again.")
```

This allows us to give some feedback to the user so they know the program hasn't crashed.

Another feature of `get()` is that it will raise any uncaught exceptions inside the delayed function. That is, in our example, if `my_long_running_function()` raises an exception it should be caught with a handler around `res.get()`.

Now that we've seen how to schedule tasks and get their results later, let's implement this in the `course4_proj`.

Try It Out

Try It Out

We'll make the movie `search_and_save()` function into a task. Then we'll add three views for search.

- A *search* view. This will initiate a search with `search_and_save()`. If the results are returned within two seconds we'll redirect the user straight to the results view. Otherwise we'll redirect the user to a waiting view until the results are ready.
- A *waiting* view. This will contain the task ID in the URL and we'll use it to fetch the `AsyncResult` and check if it's finished. If it's finished, we'll redirect to the search results page. Otherwise, we'll advise the user to refresh.
- A *results* view. This will perform the search against the database and show the results as plain text.

Start by creating a file called `tasks.py` in the `movies` app directory. It will first need to import the `shared_task` decorator and the `omdb_integration` module so we have access to `search_and_save`:

```
from celery import shared_task

from movies import omdb_integration
```

Then below the imports, reimplement `search_and_save()` as a task. It will just call the original `search_and_save()` function, but we'll add the `shared_task` decorator so it's loaded by Celery.

```
@shared_task
def search_and_save(search):
    return omdb_integration.search_and_save(search)
```

▼ **Separate files**
Since task functions can be called normally (i.e. even if decorated we can still call `search_and_save()` instead of `search_and_save.delay()`) we could choose to move the functions from `omdb_integration.py` to `tasks.py`. Having them segregated like this can make it more clear which functions you intend to run as tasks. It can also make the namespacing of your code strange if you just have `tasks.py` as a "catch-all" for many different functions across different domains.

Now we'll implement the views. Open the `movies` app's `views.py`. We won't be using templates, so you can remove the `from django.shortcuts import render` line, and instead add these imports:

Open movies/views.py

```
import urllib.parse

from celery.exceptions import TimeoutError
from django.http import HttpResponse
from django.shortcuts import redirect
from django.urls import reverse

from course4_proj.celery import app
from movies.models import Movie
from movies.tasks import search_and_save
```

The first view is `search()`. It retrieves the search term from the `search_term` query string parameter, then uses it to `search_and_save.delay(search_term)`. It then waits two seconds for a result, and if none is received, it redirects to the waiting view, with the ID of the task for later reference. If search results are received within two seconds (for example, if there aren't many results or the results have already been cached) then it redirects straight to the search results view. Here's the code to implement it:

```
def search(request):
    search_term = request.GET["search_term"]
    res = search_and_save.delay(search_term)
    try:
        res.get(timeout=2)
    except TimeoutError:
        return redirect(
            reverse("search_wait", args=(res.id,))
            + "?search_term="
            + urllib.parse.quote_plus(search_term)
        )
    return redirect(
        reverse("search_results")
        + "?search_term="
        + urllib.parse.quote_plus(search_term),
        permanent=False,
    )
```

Next the `search_wait()` view. It accepts the task UUID as an argument, then uses that to fetch the `AsyncResult()`. It tries to get the result. By using a `timeout` of `-1` it will return immediately if there's no result: using a `timeout` of `0` actually means it will wait forever for a result which is not what we want.

If the `get()` does timeout (i.e. there's no result available immediately) then a `TimeoutError` will be raised. It's caught and a message is returned to the browser telling the user to refresh the page to check again. If a result is returned (which means the task has finished), then a redirect to the results pages is returned. Here's the `search_wait()` view:

```
def search_wait(request, result_uuid):
    search_term = request.GET["search_term"]
    res = app.AsyncResult(result_uuid)

    try:
        res.get(timeout=-1)
    except TimeoutError:
        return HttpResponse("Task pending, please refresh.",
        status=200)

    return redirect(
        reverse("search_results")
        + "?search_term="
        + urllib.parse.quote_plus(search_term)
    )
```

Finally the `search_results`. It queries the database for the search term and returns all the results as a plain text list:

```
def search_results(request):
    search_term = request.GET["search_term"]
    movies = Movie.objects.filter(title__icontains=search_term)
    return HttpResponse(
        "\n".join([movie.title for movie in movies]),
        content_type="text/plain"
    )
```

▼ **Minimal implementation**
Note that the intention of these views is to be a minimal implementation to demonstrate Celery. You'll probably spot several ways to raise exceptions. For example, if `search_term` is missing. If we were implementing this in a production app we'd have much better error handling.

The final file to set up is `urls.py` inside the `course4_proj` module directory.

Open urls.py

Add the `movies.views` import:

```
import movies.views
```

Then add these URL patterns:

```
    path("search/", movies.views.search, name="search"),
    path(
        "search-wait/<uuid:result_uuid>/",
        movies.views.search_wait, name="search_wait"
    ),
    path("search-results/", movies.views.search_results,
        name="search_results")
```

You're ready to try it out, but there's one last thing you can change to make testing easier. In `omdb_integrations.py` in the `search_and_save()` function, we `return` if the `search_term` was already used. You can comment out the `return` inside the check, which will allow us to repeatedly make the same search. This is not necessary, but can make seeing the behavior easy because you can repeat long searches.

Before these new views will work, you'll need to make sure a Celery worker is started:

Open a terminal

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

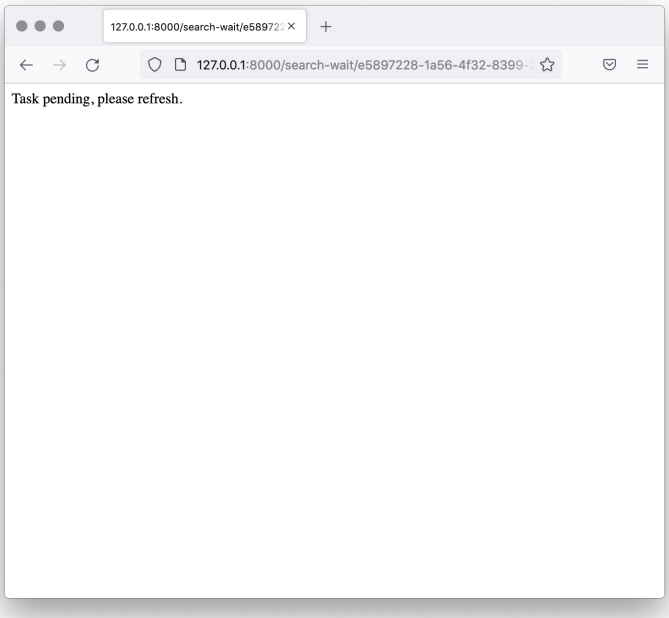
▼ **Reloading code changes**

Note that Celery doesn't automatically reload when code changes are made, like Django does. If you edit the `search_and_save()` function in particular, you'll need to stop then start the Celery worker for the changes to be picked up.

You should see the `search_and_save` task listed during startup:

```
[tasks]
  . movies.tasks.search_and_save
```

Start the Django dev server too (in another terminal), then start a search by visiting the search page. For example: `/search/?search_term=star+wars`. This is a good search as it has a lot of results. The page should load for two seconds and then you'll be redirected to the waiting page.



waiting page

info

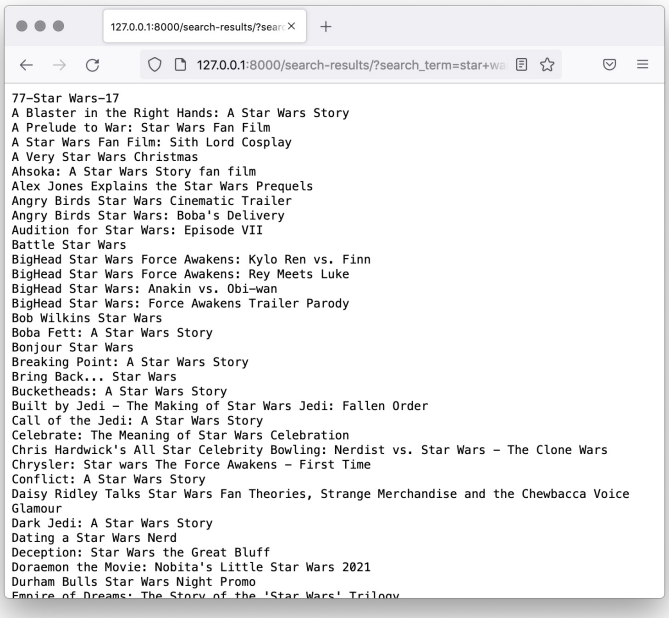
Opening a Second Terminal

To open a second terminal, click *Tools* in the Codio menu bar. Then click *Terminal*. Enter the following command in the terminal to launch the dev server.

```
python3 course4_proj/manage.py runserver 0.0.0.0:8000
```

View Project

Notice how the ID of the task is used in the URL. You can now refresh the waiting page, and can do so repeatedly until the search is finished (you can watch the Celery output terminal to see when the search has completed). Once it's finished you will be redirected to the results view.



results page

Make sure you re-enable the early return for repeated searches in `search_and_save()` (if you disabled it) and retry the search. You should get redirected to the results straight away. Or, if you try a more precise search that returns all the results within two seconds, you'll be redirected without going to the waiting view.

Improvements to the Implementation

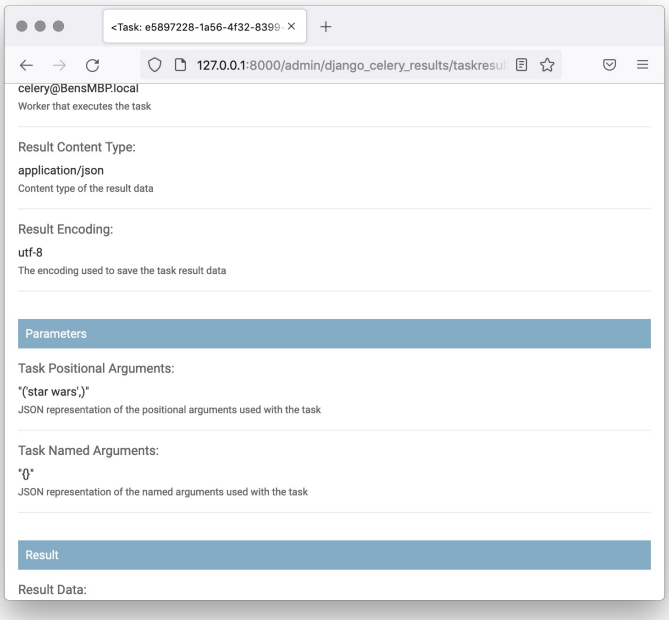
As mentioned earlier, this is a bit of a quick-and-dirty implementation to demonstrate Celery. Here are some things to consider if using these techniques in production:

- Use JavaScript to check if the results are ready on the waiting page. `fetch()` can be called multiple times and then a redirect can be performed in the browser, meaning the user doesn't have to manually refresh the page.
- When a search result is initiated, store the task ID in the database for the result (for example, in a new field on the `SearchTerm` model). If another user performs a search while one is already active, we could reuse the same task ID to prevent starting another query for the same results.
- The UUID is exposed to the user. In our case this is not a big security concern, and UUIDs are nearly impossible to guess, so the chance of being able to guess another user's search based on UUID is fairly low. However it's a better practice to not allow fetching of arbitrary tasks based on ID without check that they belong to the logged in user. This is something that you'd need to store in the database or session yourself.

Task Results and Production

Task Results

Since we're using *django-celery-results* we can see results of tasks in Django admin. Log into Django admin and navigate to the *Task Results* list, you should have some populated now. You can see the results of the task, what parameters it was called with, which worker executed the task, and more.



task results detail

View Project

This can make it easier to debug problems with Celery tasks. Debugging asynchronous tasks can sometimes be quite difficult as minute differences between the web environment and Celery environment can cause the tasks to execute differently, so any extra information can help.

Celery in Production

When we take Django to production, we don't just start up the Django dev server in a terminal and leave it running. Similarly with Celery, we don't just run `celery` in a terminal. Instead it should be set up within your

operating system's daemon system (`supervisord`, `systemd`, `init`, etc). More information on this can be found at Celery's Daemonizing Guide. The exact process will depend on your OS.

In the next section we'll look at Django signals and how to use them with Celery.

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 celery introduction"
```

- Push to GitHub:

```bash
git push
```