Django Training - 9. Running Asynchronous Tasks using Celery

Introduction

- A lookback on what we've been doing

  So far all our work with Django has been done in a synchronous fashion, for a set of instructions in a view the labeled `I(0), I(1), ..., I(n-1), I(n)`, the instruction `I(n)` waits for the instruction `I(n-1)` to finish executing before `I(n)` can start executing.

- Example

  In our platform, let's say we optionally ask the user *(whether they are an artist or not)* during registration to provide their spotify playlist url if they wanted to, and if a user provided their playlist url we make a call to Spotify's API and fetch that playlist to get an idea of which genres that user likes listening to.

- Problem

  ```
  class RegisterView:
      def post(request):
          # I(0): validate user registration data
          # I(1): create user instance
          # I(2): fetch spotify playlist (heavy API call)
          # I(3): return Response(status=200)
  ```

  The call to Spotify's API can take a lot of time since we're fetching a big chunk of data, because we're using synchronous django, during that call the frontend will still be waiting for a `200 OK` response from the API, technically we should return that `200 OK` response if the user provided registration data that passed our validation, and the call to Spotify's API is not an integral part of the user registration, it's optional, and the user can use our platform regardless of their spotify playlist.

- A possible solution

  We can make the call `I(2)` asynchronous and run it in the background, without blocking `I(3)`, to do that we'll use Celery Task Queue.

  Note: Django already has asynchronous support but for this tutorial we want to focus on Celery

- Extension

  What if we also wanted to fetch the user's spotify playlist every fixed amount of time *(say 14 days)* to see if they added any new songs? Celery also allows us to do that in a process called Periodic Scheduling.

Material

- Celery
  - Using Redis
  - User Guide
  - Django
- Redis
- django-environ
- Sending email

Use case

An example of a heavy task is sending emails, we want to send an artist user an email in these situations: 1. When an artist creats an album, we want to send an email to their address congratulating them. 2. We want to run a periodic task every 24 hours to see if any artists haven't created an album in the past 30 days, if so, we send them an email letting them know that their inactivity is causing their popularity on our platform to decrease.

Task

1. Install `redis-server` on your machine
2. Install `celery` as a project dependency
3. Install `redis` as a project dependency
4. Integrate celery with the project
    - Allow for defining celery config options in `settings.py` module
    - Celery config options should have the prefix `CELERY_CONF`
        - for example: `CELERY_CONF_TIMEZONE`, `CELERY_CONF_RESULT_BACKEND`
5. Setup a gmail email to use for this project to send emails from
6. Use `django-environ` to import your secure environment variables like the redis server address or email credentials
7. Define a task in `albums/tasks.py` or whatever `related_name` you provided for celery's `autodiscover_tasks()` method
8. Define a task that receives the artist and album data you need as arguments and send the artist a congratulation email.
    - remember: the data the task receives must be serializable
9. Use a post-save signal or override `Album.save()` and each time an album is created **asynchronously call** the task that you defined in the previous step
10. Define another task that achieves `Use case 2.` and use celery's default beat scheduler `PersistentScheduler` to run the task every 24 hours
    - Define the `beat_schedule` in `settings.py` like so: `CELERY_CONF_BEAT_SCHEDULE = { 'add-every-30-seconds': { 'task': 'tasks.add', 'schedule': 30.0, 'args': (16, 16) }, }`
    - Read: There is another way to define periodic tasks during runtime by storing the beat schedule in django's database and using django admin to define each task's schedule. See `django-celery-beat`

Guidelines

1. Use this gitignore template
2. List your environment variables (if any) in a `.env.example` file
3. Don't forget to run `manage.py makemigrations` and `python manage.py migrate`
4. You are allowed to refer to any articles, documentation source, questions on StackOverFlow
5. You are not allowed to consult other people through any medium of communication
6. Your grade will be affected if your code isn't clean, you should maintain code cleanliness as much as you can
7. **I should be able to run your tasks when I do the following**:
    1. Defining redis url variable
    2. Defining email variables
    3. Running a celery worker
    4. Running a celery beat instance