Getting Up and Running Locally

Setting Up Development Environment

Make sure to have the following on your host:

- Python 3.12
- PostgreSQL.
- Redis, if using Celery
- Cookiecutter

First things first.

1. Create a virtualenv:

```
$ python3.12 -m venv <virtual env path>
```

2. Activate the virtualenv you have just created:

```
$ source <virtual env path>/bin/activate
```

3. Generate a new cookiecutter-django project:

```
$ cookiecutter gh:cookiecutter/cookiecutter-django
```

   For more information refer to Project Generation Options.

4. Install development requirements:

```
$ cd <what you have entered as the project_slug at setup stage>
$ pip install -r requirements/local.txt
$ git init # A git repo is required for pre-commit to install
$ pre-commit install
```


Note

the *pre-commit* hook exists in the generated project as default. For the details of *pre-commit*, follow the pre-commit site.

5. Create a new PostgreSQL database using createdb:

```
$ createdb --username=postgres <project_slug>
```

`project_slug` is what you have entered as the project_slug at the setup stage.

Note

if this is the first time a database is created on your machine you might need an initial PostgreSQL set up to allow local connections & set a password for the `postgres` user. The postgres documentation explains the syntax of the config file that you need to change.

6. Set the environment variables for your database(s):

```
$ export DATABASE_URL=postgres://postgres:<password>@127.0.0.1:5432/<DB name given
to createdb>
```

Note

Check out the Settings page for a comprehensive list of the environments variables.

See also

To help setting up your environment variables, you have a few options:

- create an `.env` file in the root of your project and define all the variables you need in it. Then you just need to have `DJANGO_READ_DOT_ENV_FILE=True` in your machine and all the variables will be read.
- Use a local environment manager like direnv

7. Apply migrations:

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

8. If you're running synchronously, see the application being served through development server:

```
$ python manage.py runserver 0.0.0.0:8000
```

or if you're running asynchronously:

```
$ uvicorn config.asgi:application --host 0.0.0.0 --reload --reload-include '*.html'
```

If you've opted for Webpack or Gulp as frontend pipeline, please see the dedicated section below.

Creating Your First Django App

After setting up your environment, you're ready to add your first app. This project uses the setup from "Two Scoops of Django" with a two-tier layout:

- **Top Level Repository Root** has config files, documentation, *manage.py*, and more.
- **Second Level Django Project Root** is where your Django apps live.
- **Second Level Configuration Root** holds settings and URL configurations.

The project layout looks something like this:

```
<repository_root>/
├── config/
│   ├── settings/
│   │   ├── __init__.py
│   │   ├── base.py
│   │   ├── local.py
│   │   └── production.py
│   ├── urls.py
│   └── wsgi.py
├── <django_project_root>/
│   ├── <name_of_the_app>/
│   │   ├── migrations/
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   └── views.py
│   ├── __init__.py
│   └── ...
├── requirements/
│   ├── base.txt
│   ├── local.txt
│   └── production.txt
├── manage.py
├── README.md
└── ...
```

latest

Following this structured approach, here's how to add a new app:

1. **Create the app** using Django's `startapp` command, replacing `<name-of-the-app>` with your desired app name:

```
$ python manage.py startapp <name-of-the-app>
```

2. **Move the app** to the Django Project Root, maintaining the project's two-tier structure:

```
$ mv <name-of-the-app> <django_project_root>/
```

3. **Edit the app's apps.py** change `name = '<name-of-the-app>'` to `name = '<django_project_root>.<name-of-the-app>'`.
4. **Register the new app** by adding it to the `LOCAL_APPS` list in `config/settings/base.py`, integrating it as an official component of your project.

Setup Email Backend

Mailpit

Note

In order for the project to support Mailpit it must have been bootstrapped with `use_mailpit` set to `y`.

Mailpit is used to receive emails during development, it is written in Go and has no external dependencies.

For instance, one of the packages we depend upon, `django-allauth` sends verification emails to new users signing up as well as to the existing ones who have not yet verified themselves.

1. Download the latest Mailpit release for your OS.
2. Copy the binary file to the project root.
3. Make it executable:

```
$ chmod +x mailpit
```

4. Spin up another terminal window and start it there:

```
./mailpit
```

5. Check out http://127.0.0.1:8025/ to see how it goes.

Now you have your own mail server running locally, ready to receive whatever you send it.

Console

Note

If you have generated your project with `use_mailpit` set to `n` this will be a default setup.

Alternatively, deliver emails over console via `EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'`.

In production, we have Mailgun configured to have your back!

Celery

If the project is configured to use Celery as a task scheduler then, by default, tasks are set to run on the main thread when developing locally instead of getting sent to a broker. However, if you have Redis setup on your local machine, you can set the following in `config/settings/local.py`:

```
CELERY_TASK_ALWAYS_EAGER = False
```

Next, make sure *redis-server* is installed (per the Getting started with Redis guide) and run the server in one terminal:

```
$ redis-server
```

Start the Celery worker by running the following command in another terminal:

```
$ celery -A config.celery_app worker --loglevel=info
```

That Celery worker should be running whenever your app is running, typically as a background process, so that it can pick up any tasks that get queued. Learn m latest Celery Workers Guide.

The project comes with a simple task for manual testing purposes, inside *<project_slug>/users/tasks.py*. To queue that task locally, start the Django shell, import the task, and call *delay()* on it:

```
$ python manage.py shell
>> from <project_slug>.users.tasks import get_users_count
>> get_users_count.delay()
```

You can also use Django admin to queue up tasks, thanks to the django-celerybeat package.

Using Webpack or Gulp

If you've opted for Gulp or Webpack as front-end pipeline, the project comes configured with Sass compilation and live reloading. As you change your Sass/JS source files, the task runner will automatically rebuild the corresponding CSS and JS assets and reload them in your browser without refreshing the page.

1. Make sure that Node.js v18 is installed on your machine.
2. In the project root, install the JS dependencies with:

   ```
   $ npm install
   ```

3. Now - with your virtualenv activated - start the application by running:

   ```
   $ npm run dev
   ```

   This will start 2 processes in parallel: the static assets build loop on one side, and the Django server on the other.

4. Access your application at the address of the `node` service in order to see your correct styles. This is http://localhost:3000 by default.

   Note

   Do NOT access the application using the Django port (8000 by default), as it will result in broken styles and 404s when accessing static assets.

Summary

Congratulations, you have made it! Keep on reading to unleash full potential of Django.