How to implement CICD for Django project using Make

In the previous articles, we explored the steps to create, run, build, and deploy a Django project. However, a “well-run” software production process encompasses many more steps such as: development/production environment setup, package updates, tests, and so on. Most of these steps are repetitive and take a lot of manual effort. In many cases, this entire process also requires establishing a “traceable path” for future auditing processes as well. While there are many ways to achieve a “well-run” process, at PeriShip, we are using one of the most common development utilities, Make tool, to accomplish this. In this article, I will share how we can leverage the Make tool to run, test, build, and deploy the software product to your favorite CI/CD environment and the local machine.

In this article, the running environment is Ubuntu. If you are using a Mac, you should have no issue. For Windows, I would suggest using WSL or Virtualbox to achieve the same result.

Base project

Let’s use this project as the base. It’s mentioned in my previous article How to use AdminLTE in Django.

Step 1. Run project

In the root folder, create a **Makefile** file. Edit it so that the context is as follows:

```
run: venv
	. venv/bin/activate && python manage.py runserver

venv: requirements.txt
	python3.8 -m venv venv
	. venv/bin/activate && pip install --upgrade pip && \
	pip install -r requirements.txt

clean-all:
	rm -rf venv/ db.sqlite3
```

The **run** target executes Django **runserver** command to launch the application. The **venv** target creates the virtual environment folder. The **clean-all** target removes the **venv** folder and the **db.sqlite3** file.

We probably don't want to commit the **venv** folder and sqlite file to the Github. Thus, in the **.gitignore** file, let's add the following.

```
venv/
db.sqlite3
```

Now, we can type the following command to run the project.

```
make run
```

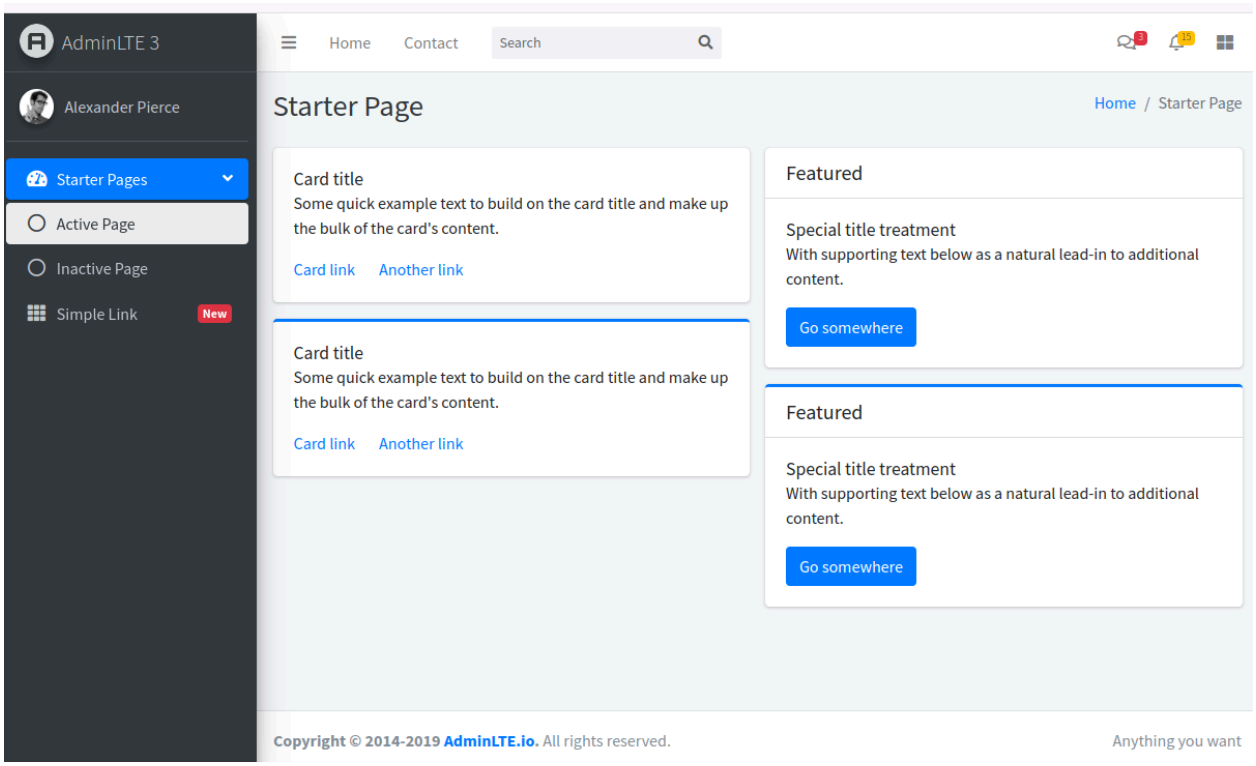
We should see the output console as follows.

```
python3.8 -m venv venv
. venv/bin/activate && pip install --upgrade pip && \
pip install -r requirements.txt
Requirement already satisfied: pip in ./venv/lib/python3.8/site-packages
(23.0.1)
Collecting pip
  Using cached pip-24.2-py3-none-any.whl (1.8 MB)
Installing collected packages: pip
  Attempting uninstall: pip
    Found existing installation: pip 23.0.1
    Uninstalling pip-23.0.1:
      Successfully uninstalled pip-23.0.1
Successfully installed pip-24.2
Collecting django==2.2.24 (from -r requirements.txt (line 1))
  Using cached Django-2.2.24-py3-none-any.whl.metadata (3.6 kB)
Collecting pytz (from django==2.2.24->-r requirements.txt (line 1))
  Using cached pytz-2024.1-py2.py3-none-any.whl.metadata (22 kB)
Collecting sqlparse>=0.2.2 (from django==2.2.24->-r requirements.txt (line
1))
  Using cached sqlparse-0.5.1-py3-none-any.whl.metadata (3.9 kB)
Using cached Django-2.2.24-py3-none-any.whl (7.5 MB)
Using cached sqlparse-0.5.1-py3-none-any.whl (44 kB)
Using cached pytz-2024.1-py2.py3-none-any.whl (505 kB)
Installing collected packages: pytz, sqlparse, django
Successfully installed django-2.2.24 pytz-2024.1 sqlparse-0.5.1
```

```
. venv/bin/activate && python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 17 unapplied migration(s). Your project may not work properly
until you apply the migrations for app(s): admin, auth, contenttypes,
sessions.
Run 'python manage.py migrate' to apply them.

August 19, 2024 - 03:58:20
Django version 2.2.24, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CONTROL-C.
```

Since **venv** is the prerequisite of **run** target, the **venv** folder will be created before application is launched.

To verify the running application, we can open up http://127.0.0.1:8000/polls/ in the browser. It should look like below.



Step 2. Test project

The base project has no test to start with. Let's add one test. In the **polls/tests.py** file, add the following.

```
from django.test import Client
from django.test import TestCase


class IndexViewTestCase(TestCase):
    def setUp(self):
        self.client = Client()

    def test_happy_path_should_return_200(self):
        response = self.client.get('/polls/')
        self.assertEqual(response.status_code, 200)
```

In the **requirements.txt** file, add **coverage** package to measure code coverage.

```
coverage==7.6.1
```

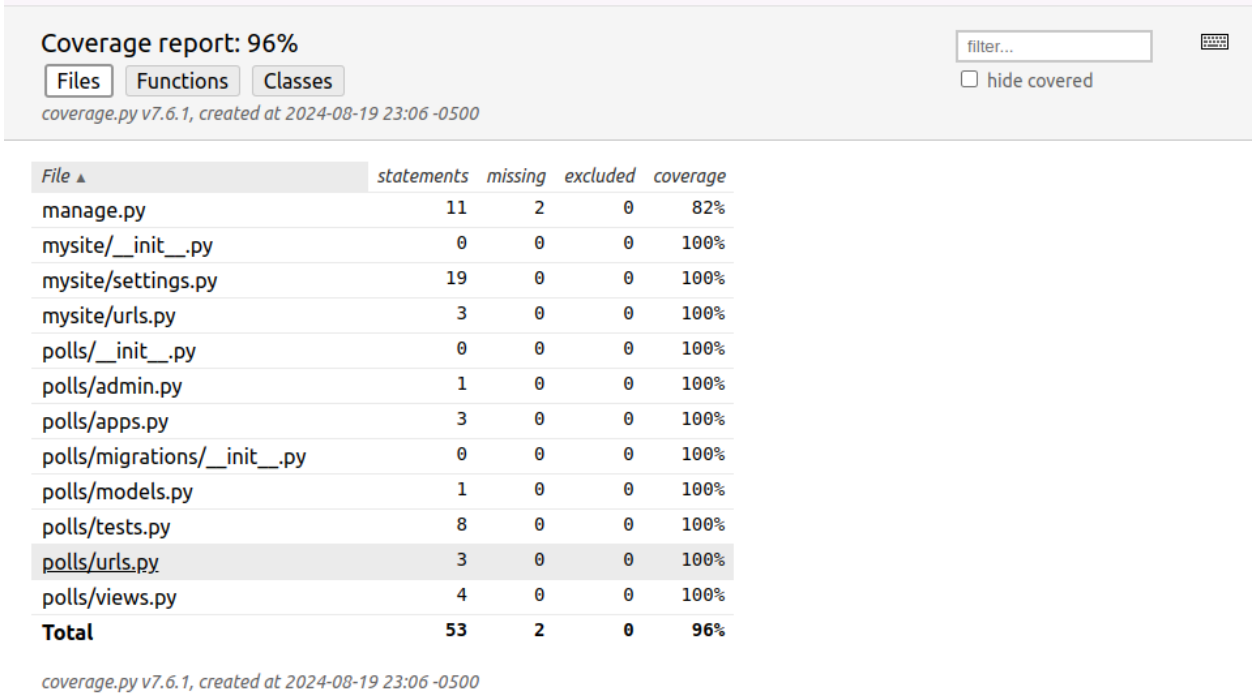
In the **Makefile** file, add the **test** target as below.

```
test: venv
	. venv/bin/activate && coverage run --source='.' manage.py test &&
coverage html
```

Now we can test the project by running the following command.

```
make test
```

To review the coverage report, we can open the **htmlcov/index.html** file in the browser. We should see something as below.



Coverage report: 96%

Files | Functions | Classes

filter...

hide covered

coverage.py v7.6.1, created at 2024-08-19 23:06 -0500

File ▲	statements	missing	excluded	coverage
manage.py	11	2	0	82%
mysite/__init__.py	0	0	0	100%
mysite/settings.py	19	0	0	100%
mysite/urls.py	3	0	0	100%
polls/__init__.py	0	0	0	100%
polls/admin.py	1	0	0	100%
polls/apps.py	3	0	0	100%
polls/migrations/__init__.py	0	0	0	100%
polls/models.py	1	0	0	100%
polls/tests.py	8	0	0	100%
polls/urls.py	3	0	0	100%
polls/views.py	4	0	0	100%
Total	**53**	**2**	**0**	**96%**

coverage.py v7.6.1, created at 2024-08-19 23:06 -0500

We probably want to clean up artifacts that are created by test tools and exclude them from Github. Thus, let's edit the **Makefile** file as follows.

```
...

clean:
	rm -rf htmlcov/ .coverage

clean-all: clean
	rm -rf venv/ db.sqlite3
```

Notice that we make the **clean** target a prerequisite to the **clean-all** target based on how often we clean up.

In the **.gitignore** file, add the following.

```
htmlcov/
.coverage
```

Step 3. Build project

In this article, we are going to containerize this project. In the root folder create a **Dockerfile** file. The content is as follows.

```
FROM python:3.8-alpine
WORKDIR /code

COPY requirements.txt requirements.txt
RUN pip install --upgrade pip && pip install -r requirements.txt

COPY mysite/ mysite/
COPY polls/ polls/
COPY static/ static/
COPY manage.py manage.py
```

In the **Makefile** file, add a **build** target as follows.

```
...

build: Dockerfile
	docker build . -t mysite

...
```

Now, run the following command to build the Docker image.

```
make build
```

Step 4. Group concept in make

By default, if no target is provided, Make would run the very first target in the Makefile. As developers, we probably want to run make so that it tests and builds every time. In this case, let's add a **default** target at the beginning of the **Makefile** file. It looks like below.

```
default: test build
```

Now, we can iterate our development process by running the command below. Run it as much as we want to.

```
make
```

Step 5. Deployment

In this article, we are going to deploy using docker compose. Create a **docker-compose.yaml** file in the root directory. Edit the file as follows.

```
services:
    web:
      image: mysite:latest
      command: python manage.py runserver 0.0.0.0:8000
      ports:
        - "8000:8000"
```

Then edit the **Makefile** file to add the **deploy** and **stop** targets. They are as follows.

```
...

deploy: docker-compose.yaml
      docker compose up -d

stop: docker-compose.yaml
      docker compose down

...
```

To deploy the build, we can run the following command. We should see the web page http://127.0.0.1:8000/polls/ in the browser.

```
make deploy
```

To stop the build, we can run the following command.

```
make stop
```

At the end, the final **Makefile** file is as follows.

```
default: test build

run: venv
	. venv/bin/activate && python manage.py runserver

test: venv
	. venv/bin/activate && coverage run manage.py test && coverage html

build: Dockerfile
	docker build . -t mysite

venv: requirements.txt
	python3.8 -m venv venv
	. venv/bin/activate && pip install --upgrade pip && \
	pip install -r requirements.txt

deploy: docker-compose.yaml
	docker compose up -d

stop: docker-compose.yaml
	docker compose down

clean:
	rm -rf htmlcov/ .coverage

clean-all: clean
	rm -rf venv/
```

Summary

In this article, we have gone through how to run, test, build, and deploy a project using Make. A common practice of CI/CD also contains the concept of packaging. So we may consider adding **make package** into the Makefile. We probably also want to add **make lint** to do static program analysis. Sometimes we can break down a target into smaller components—for example, tests.

```
test: test-unit test-integration

test-unit: venv
	# command to run unit tests
```

```
test-integration: venv
	# command to run integration tests
```

Not limited to CICD, Make can be used in other areas. For example, we can use Make to install Python. Download the Gzipped source tarball source release from Python official page, and unzip it. Following the instructions below in the **README.rst**, we should install Python successfully. If you are interested in this approach, please let me know in the comments. I can write a post to share how to.

```
Build Instructions
------------------

On Unix, Linux, BSD, macOS, and Cygwin::

    ./configure
    make
    make test
    sudo make install

This will install Python as ``python3``.
```

How would you like to use Make to build concept in your CICD tool? Let me know your thoughts in the comments. In addition, I want to thank Carlos Sandoval for introducing Make. Thanks for reading. Stay tuned.

For anyone interested in the source code, I've uploaded it to my Git repository as follows.
https://github.com/slow999/DjangoAndCICD