Heroku Deployment Steps

Once a Django project has been set up and developed, it can be deployed to Heroku using the following steps (this is how this project was deployed):

1. Ensure that all dependencies are in the requirements.txt file within the project using the python command “pip3 freeze > requirements.txt”:

2. Navigate to https://www.heroku.com/ and login

3. In the top right corner, select ‘New’ then ‘Create new app’

4. From the ‘Create New App’ screen, enter a unique App name and select Europe, then select ‘Create app’

5. An app is created and the dashboard is shown, from here navigate to the 'Resources tab'

6. The Postgres database needs to be connected to the app using an add-on. Search the 'Add-ons' in the Resources tab for 'Heroku Postgres' and select this add-on

7. On the pop-up for Heroku Postgres select a plan type (for this site 'Hobby Dev - Free' was selected)

8. Navigate to the 'Settings' tab for the app and select to 'Reveal Config Vars', a variable called `DATABASE_URL` will have been created by connecting the database add-on. Copy the ****value**** of this variable

9. Within the development environment, create a file called `env.py` at the top level

   - Ensure that this file is added to the .gitignore file. If your project does not have a .gitignore file then create one and add the `env.py` file to it

10. In the `env.py` file, import the `os` library and create a database variable using the value taken from Heroku:

~~~

os.environ ["DATABASE_URL"] = "<heroku database variable goes here>"

~~~

11. Whilst in the `env.py` file, create a `SECRET_KEY` variable which will be used later. To generate a new Django secret key, do a google search for a Django secret key generator and use one of the results to create a key. The variable can be created using:

~~~

os.environ ["SECRET_KEY"] = "<secret key goes here>"

~~~

12. Back in Heroku, add your secret key variable to the Config Vars by selecting 'Add' and entering 'SECRET_KEY' as the Key and your secret key value as the Value

13. Return to the development environment, and navigate to the `settings.py` file. Within this file, import the following:

~~~

from pathlib import Path

import os

import dj_database_url

if os.path.isfile('env.py'):

    import env

~~~

These imports will enable you to access the variables in your `env.py`

14. Find the `SECRET_KEY` variable and replace the assignment as follows:

~~~

SECRET_KEY = os.environ.get('SECRET_KEY')

~~~

15. Find the `DATABASES` variable and comment out the existing code (this will be used in a later step), add in the following code:

~~~

DATABASES = {

    'default': dj_database_url.parse(os.environ.get('DATABASE_URL'))

}
~~~

~~~

This will enable Heroku to connect to the database

16. As a new database is now being used, migrations need to be run again by using

~~~

python3 manage.py migrate

~~~

17. As this project uses Cloudinary for file storage, additional steps are needed to configure this. Start by creating another variable in the `env.py` file for your personal Cloudinary (which can be obtained from the Cloudinary dashboard):

~~~

os.environ["CLOUDINARY_URL"] = "<your cloudinary url>"

~~~

18. Next, go back to Heroku and add another Config Var for your cloudinary url using `CLOUDINARY_URL` for the Key and your cloudinary url as the Value

19. If your project does not have any static files then add another Config Var to Heroku using `DISABLE_COLLECTSTATIC` as the Key and '1' as the Value. This will prevent the deployment from failing if there are no static files. This variable has been removed for this project as there are static files.

20. Return to the development environment, and in the `settings.py` file find the `INSTALLED_APPS` variable add in 'cloudinary_storage' before 'django.contrib.staticfiles' and then add 'cloudinary' underneath 'django.contrib.staticfiles'.

21. To configure Django to use Cloudinary, find the static files section towards the bottom of the `settings.py` file and add the following code:

~~~

STATIC_URL = '/static/'

STATICFILES_STORAGE = 'cloudinary_storage.storage.StaticHashedCloudinaryStorage'

STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')]

STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')

MEDIA_URL = '/media/'

DEFAULT_FILE_STORAGE = 'cloudinary_storage.storage.MediaCloudinaryStorage'

~~~

22. Next, create a `TEMPLATES_DIR` variable in `settings.py` and set it to:

~~~

os.path.join(BASE_DIR, 'templates')

~~~

so that Django knows where to find templates

23. Find the `TEMPLATES` variable and for the `DIRS` key, set the value to be the `TEMPLATES_DIR` we have just created:

~~~

'DIRS': [TEMPLATES_DIR],

~~~

24. Find the `ALLOWED_HOSTS` variable in `settings.py` and add in the urls that should be able to access the project:

~~~

ALLOWED_HOSTS = ['<heroku-project-name>.herokuapp.com', 'localhost']

~~~

25. Within your project folder structure, create three directories at the top level for storing files: media, static, and templates

26. At the top level of your project structure, create a file called Procfile and add the following code to it:

~~~

web: gunicorn codestar.wsgi

~~~

Heroku uses this file to determine how to run the app

27. To prevent sensitive information being revealed when running the app on Heroku, the `DEBUG` variable should be set to False, however this can be set conditionally depending on whether you are in the development environment or not. To do this, create an environment variable within your development environment called `DEVELOPMENT` and set its value to True (for this project this was done in GitPod > Settings > Variables)

28. Next, in the `settings.py` file, add the following variable and assignment:

~~~

development = os.environ.get('DEVELOPMENT', False)

~~~

29. Set the `DEBUG` variable to be equal to development:

~~~

DEBUG = development

~~~

30. Find the `DATABASES` variable that was previously commented out and amend the databases section to be conditional on whether you are in the development environment or not. It should look like this:

~~~

```
if development:
    DATABASES = {
        'default': {
            'ENGINE': 'django.db.backends.sqlite3',
            'NAME': BASE_DIR / 'db.sqlite3',
        }
```

```
    }
else:
    DATABASES = {
        'default': dj_database_url.parse(os.environ.get('DATABASE_URL'))
    }
```

These steps will mean that `DEBUG` will be set to True whilst in the development environment, but False when running the Heroku app. It will also mean that Heroku will use the Heroku Postgres database, whereas the development environment will use the SQLite database. Note that any changes to the database will need to be migrated to both if both are to be used.

31. Commit your changes and push these to GitHub

32. Back in Heroku add another Config Var using `PORT` as the Key and '8000' as the Value.

33. Navigate to the Deploy tab and from the 'Deployment Method' section, select 'GitHub'

34. Allow Heroku to connect to GitHub by selecting 'Connect to GitHub'

35. Search for the repository by entering the name of the GitHub repository to deploy and selecting search

36. From the results, choose the relevant repository and select 'Connect'

37. To enable automatic deployment of the repository (which will automatically redeploy the project after every push to GitHub), select the 'Enable Automatic Deploys' option

38. To manually deploy the project, select 'Deploy Branch' from the Manual Deploy section

39. When the branch is manually deployed, Heroku will build and deploy the branch. Upon completion, a link to the deployed project will be generated

NB: The text was copied from:

https://github.com/adamhatton/john-breedon-bass-tuition-pp4/blob/main/README.md