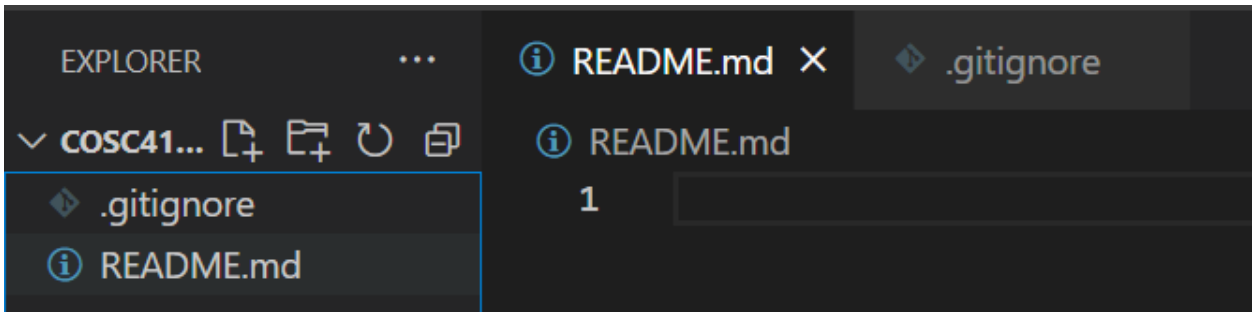
How to Setup a Django Application and Run Using Docker

By The Island Boys (Beto Estrada Jr and Ryan Seidel)

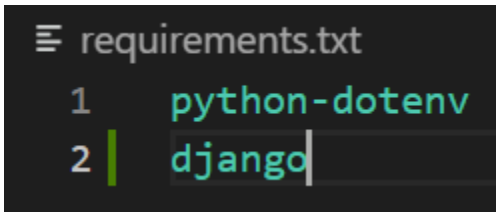
Table of Content

Setup, Installation, and Page Navigation

1. Open your project folder in Visual Studio Code and create a README.md file, requirements.txt, and a .gitignore file.

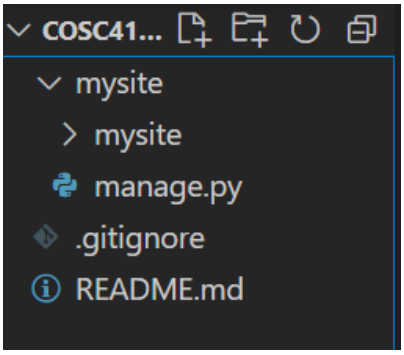


2. In 'requirements.txt' add the following lines



3. **This step is for Windows users only. MAC and Linux users can skip or research how to create their own virtual environment on their respective OS.** Open up a terminal and create a virtual environment using `python3 -m venv ~/venv/venv-name` and activate it using `source ~/venv/venv-name/Scripts/activate`. Note: You may need to use `source ~/venv/venv-name/bin/activate`.
4. Run `pip install -r requirements.txt`. You may need to use `pip3` instead of `pip`.
5. Try the `django-admin startproject mysite` command. This creates a project folder called 'mysite'. If django-admin is not found then you probably need to add django-admin to the PATH.
    a. For Windows this will usually be located in the Scripts directory inside your Python installation folder. For example: 'C:\PythonLocation\Scripts'
    b. For Linux, the django-admin executable is usually located in '/usr/local/bin'. Confirm the location using the command `sudo find / -name django-admin`

```
torial/Code/cosc4100-technical-tutorial
$ django-admin startproject mysite
```



6. Next change directories to the new project folder using `cd mysite`

```
torial/Code/cosc4100-technical-tutorial
$ cd mysite
```

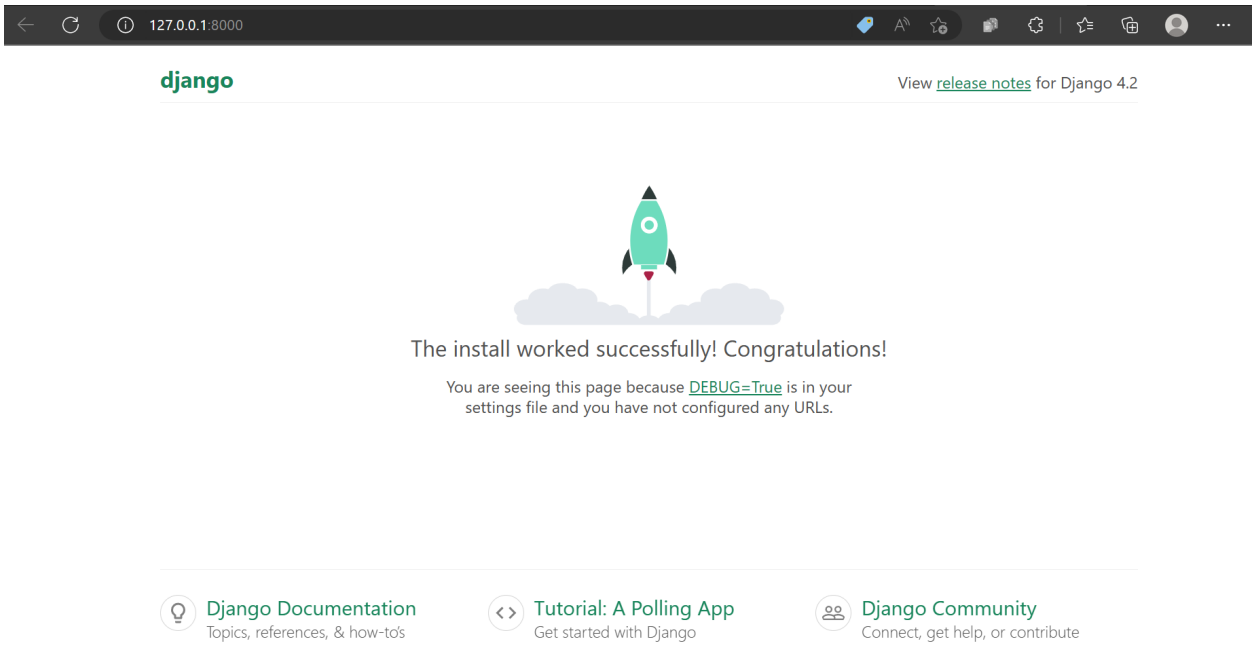
7. In 'settings.py' in the 'mysite' folder add the following code to import the `SECRET_KEY` from a .env file for extra security. You'll want to be sure to do this if you plan on pushing the repo to GitHub, or any online platform. Then be sure to create a .env file in the 'mysite' folder with your `SECRET_KEY`. You can use the key that Django created for you.

```
mysite > mysite > settings.py > ...
https://docs.djangoproject.com/en/4.2/ref/settings/
"""

from dotenv import load_dotenv
import os

from pathlib import Path

load_dotenv()

# SECURITY WARNING: keep the secret key used in production secret!
SECRET_KEY = os.environ.get('SECRET_KEY')
```

8. Then from that directory run the command `python manage.py runserver` to run a local server that will allow you to connect and view the website (though it does not have anything to show yet.)
    a. You can specify a specific port by adding the port after `runserver`. So like `python manage.py runserver 8100`

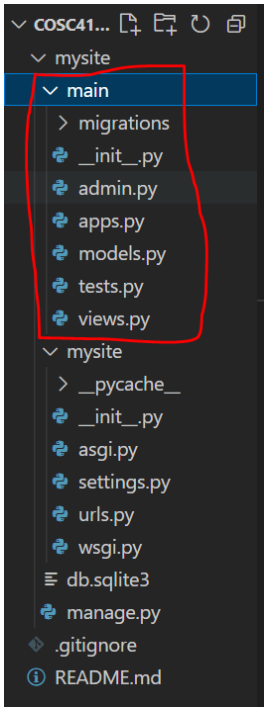
```
$ python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).

You have 18 unapplied migration(s). Your project may not work properly until you apply the migrations for app(s): admin, auth, conte
nttypes, sessions.
Run 'python manage.py migrate' to apply them.
April 12, 2023 - 21:08:31
Django version 4.2, using settings 'mysite.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

9. Click on or copy and paste the link ‘http://127.0.0.1:8000/’ to view the Django debugger page. Use `Ctrl+C` on the terminal to quit the server when you are done.



10. Create an app called ‘main’ using the command `python manage.py startapp main`. You can run the server again to make sure that everything is working.

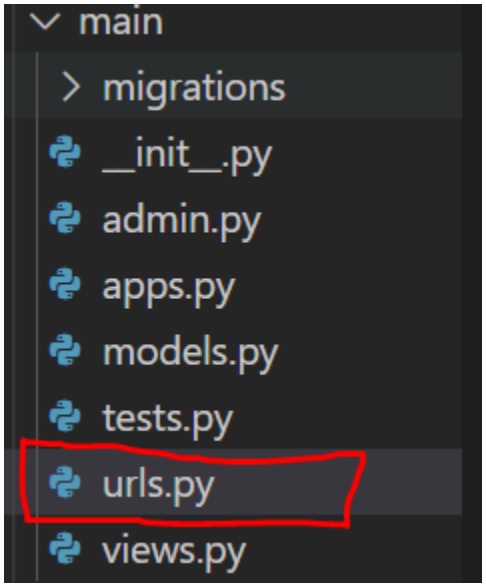
```
torial/Code/cosc4100-technical-tutorial/mysite
$ python manage.py startapp main
```



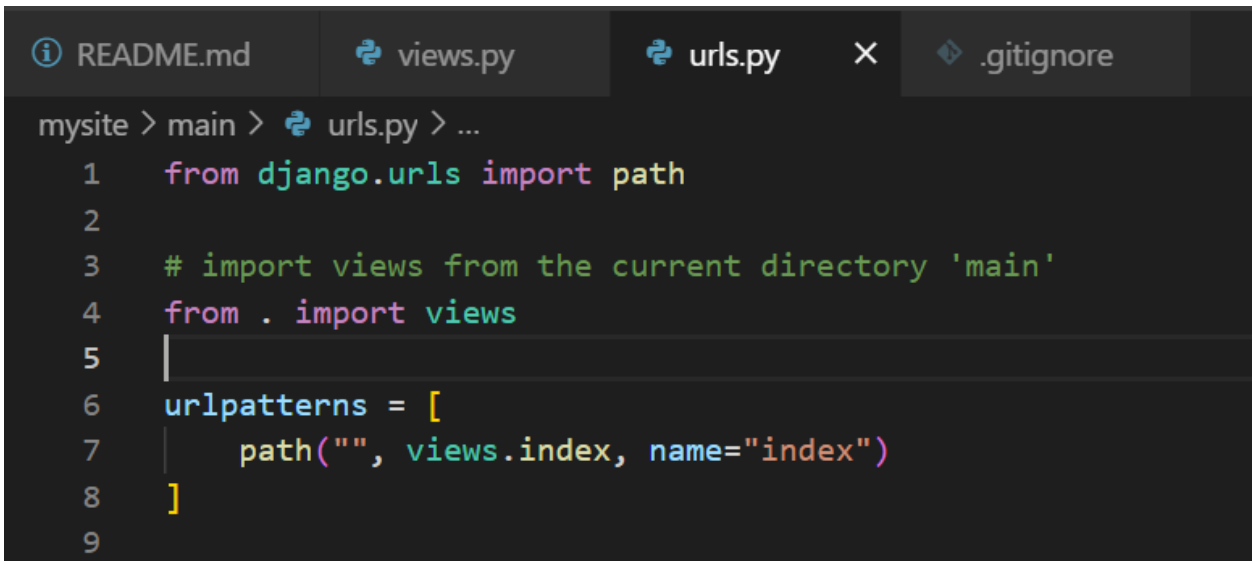
11. Next, using the Explorer in VS Code, open up the ‘views.py’ file in the ‘main’ folder. There you will be creating a simple view. Enter the code below.

```
mysite > main > views.py > index
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def index(response):
    return HttpResponse("Technical Tutorial!")
```

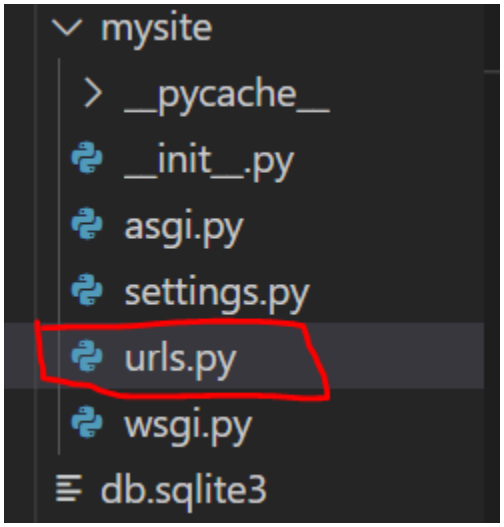
12. Inside the ‘main’ folder, create a file called ‘urls.py’. This will define the paths to the different webpages. Note that these correspond only to the views that are in ‘main’.



Add the code below to the file. This acts as a home page route for that app. So if the path that the user enters is empty, then it will take them to the index view that you created above

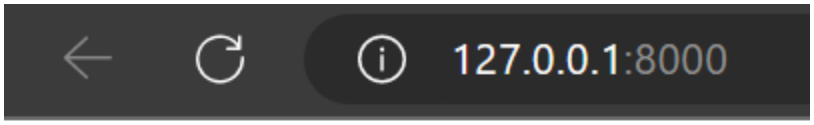


13. Now travel to the ‘urls.py’ file in the ‘mysite’ folder. NOT the ‘main’ folder. Add the code below to connect the path to the ‘main’ app paths. More specifically, this says that if the path the user enters is empty, it will take them to the ‘urls.py’ in ‘main’ and follow where the empty path takes you there, which in this case is the ‘index’ view.




```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path("admin/", admin.site.urls),
    path("", include("main.urls")),
]
```

14. Make sure to save all changes. Then restart the server and open or refresh the page from before to see the updated page. If you do not see the changes, go back a couple of steps and make sure that you followed all the steps correctly.



Technical Tutorial!

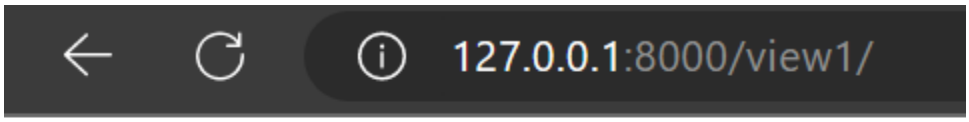
15. Now go to ‘views.py’ in ‘main’ and add a new view called ‘view1’. Notice that I also added HTML headers (<h1>) which you can add since this is an HttpResponse.

```
mysite > main > views.py > ...
from django.shortcuts import render
from django.http import HttpResponse

# Create your views here.

def index(response):
    return HttpResponse("<h1>Technical Tutorial!<h1>")

def view1(response):
    return HttpResponse("<h1>View 1!</h1>")
```

16. Go to ‘urls.py’ in ‘main’ and a path to the new view ‘view1’ that you created.

```
mysite > main > urls.py > ...
from django.urls import path

# import views from the current directory 'main'
from . import views

urlpatterns = [
    path("", views.index, name="index"),
    path("view1/", views.view1, name="view 1")
]
```

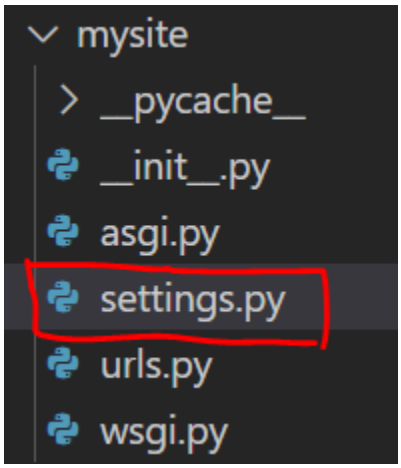
17. Run the server again and this time add ‘view1’ to the path to visit the new view that you created. It would look like this ‘http://127.0.0.1:8000/view1’



View 1!

SQLite3 Database Integration

18. Go to ‘settings.py’ in the ‘mysite’ folder and add the following code to tell Django that we have another application that has dependencies inside of our project.

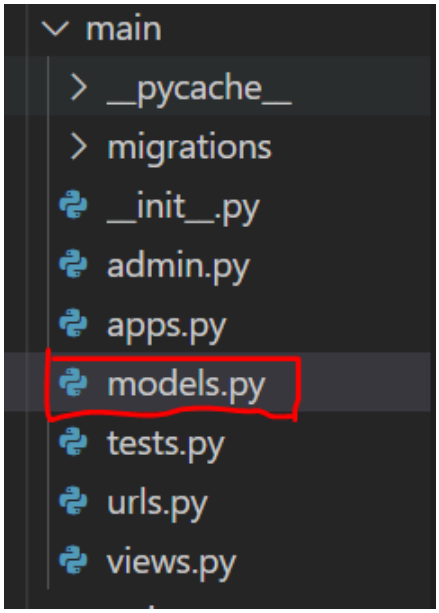



```
# Application definition

INSTALLED_APPS = [
    "django.contrib.admin",
    "django.contrib.auth",
    "django.contrib.contenttypes",
    "django.contrib.sessions",
    "django.contrib.messages",
    "django.contrib.staticfiles",
    "main.apps.MainConfig",
]
```

19. Next, `cd mysite` and run the command `python manage.py migrate`

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
Running migrations:
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying admin.0003_logentry_add_action_flag_choices... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying auth.0008_alter_user_username_max_length... OK
  Applying auth.0009_alter_user_last_name_max_length... OK
  Applying auth.0010_alter_group_name_max_length... OK
  Applying auth.0011_update_proxy_permissions... OK
  Applying auth.0012_alter_user_first_name_max_length... OK
  Applying sessions.0001_initial... OK
```

20. Next go to 'models.py' in the 'main' folder and create the models below. Here we create a simple 'ToDoList' object and an 'Item' object to go with it.



README.md | settings.py | models.py | views.py | urls.py ...\mai

mysite > main > models.py > ...

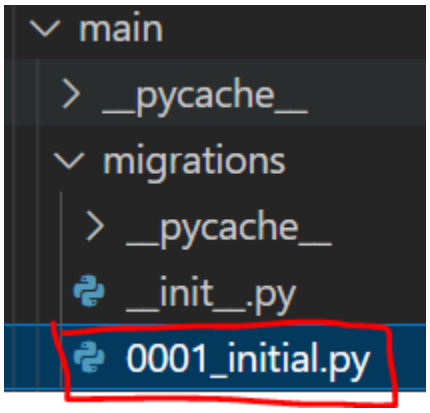
```python
from django.db import models

# Create your models here.
class ToDoList(models.Model):
    name = models.CharField(max_length=200)

    # Simple way to print the information from the object
    def __str__(self):
        return self.name

class Item(models.Model):
    todolist = models.ForeignKey(ToDoList, on_delete=models.CASCADE)
    text = models.CharField(max_length=300)
    complete = models.BooleanField()

    def __str__(self):
        return self.text
```

21. Next, while still in the 'mysite' folder, run the command `python manage.py makemigrations main` to stage the changes that you just made in your main app (creating the models). Similar to adding files to the staging area in Git.

```
$ python manage.py makemigrations main
Migrations for 'main':
  main\migrations\0001_initial.py
    - Create model ToDoList
    - Create model Item
```

22. Now run the command `python manage.py migrate` to apply the staged migrations. You can go to the 'migrations' folder in 'main' and select the most recent 'initial.py' file to verify that the correct migrations were made.

```
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, main, sessions
Running migrations:
  Applying main.0001_initial... OK
```



23. Next let's add objects to our database using the command line. Type the command `python manage.py shell`. You should see the text below with your own Python version.

```
$ python manage.py shell
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>
```

24. While in the shell, run the commands below to create a ToDoList object with the name "Bob's List" and save it to the database.

```
>>> from main.models import Item, ToDoList
>>> t = ToDoList(name="Bob's List")
>>> t.save()
```

25. Use the command `ToDoList.objects.all()` to view the new object in the database.

```
>>> ToDoList.objects.all()
<QuerySet [<ToDoList: Bob's List>]>
```

26. Use the command `t.item_set.create(text="Go to the mall", complete=False)` to create an item object in the ToDoList object `t` which is "Bob's Lists" that you just created. Use `exit()` to exit the shell.

```
>>> t.item_set.create(text="Go to the mall", complete=False)
<Item: Go to the mall>
```

27. Go to 'views.py' in 'main' to remove `view1` and go to `urls.py` to remove the path to `view1`. They are no longer needed.

```
def view1(response):
    return HttpResponse("<h1>View 1!</h1>")
```

```
urlpatterns = [
    path("", views.index, name="index"),
    path("view1/", views.view1, name="view 1")
]
```

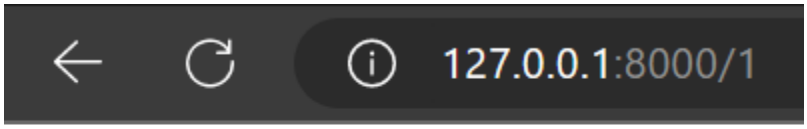
28. Next, in 'urls.py' in 'main' you can add `<int:id>` to the path to return an object with that specific id from the database.

```
urlpatterns = [
    path("<int:id>", views.index, name="index"),
]
```

29. Then add the following code to 'views.py' in 'main' to display the name of the object on the webpage based on `id`.

```
README.md   settings.py   models.py   0001_initial.py   views.py
mysite > main > views.py > index
from django.shortcuts import render
from django.http import HttpResponse
from .models import ToDoList, Item

# Create your views here.

def index(response, id):
    ls = ToDoList.objects.get(id=id)
    return HttpResponse("<h1>%s<h1>" %ls.name)
```

30. Now start the server up again using `python manage.py runserver` and add id # “1” to the path to retrieve the object where id = 1. Note that when you first open the webpage without any id in the path defined it will say “Page not found.” This is because we no longer have a view attached to the empty index path. Also, note that this only works for id = 1 since that is the only object we have defined so far.



Bob's List

31. Lastly, we will also display the item in “Bob’s list” on the webpage using the code below. Add this code ‘views.py’ in ‘main’. Note that this is a very simple example just to show the functionality.

```
def index(response, id):
    ls = ToDoList.objects.get(id=id)
    item = ls.item_set.get(id=1)
    return HttpResponse("<h1>%s</h1><br></br><p>%s</p>" %(ls.name, str(item.text)))
```

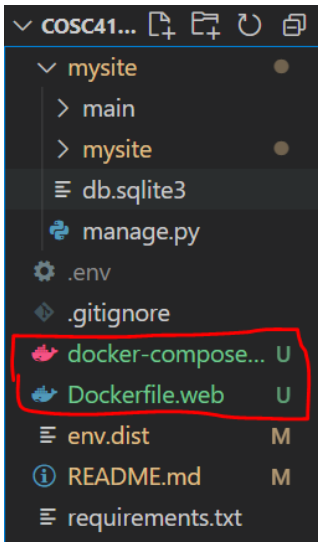
If the server is still up the changes will automatically show after saving. If not, you must start the server again.

Bob's List

Go to the mall

Containerize the Application Using Docker

32. In the parent directory (so outside main ‘mysite’ folder) create the files ‘Dockerfile.web’ and ‘docker-compose.yml’



33. In the ‘Dockerfile.web’ file add the following code:

```
FROM python:3.7-slim

WORKDIR /app

COPY ./requirements.txt ./requirements.txt
RUN pip install -r ./requirements.txt

COPY ./mysite .

# Disables output buffering, so the output from your Python
application is printed to the console in real-time.
ENV PYTHONUNBUFFERED=1
```

- `WORKDIR` sets the current working directory of the docker container.
- It’s best to copy and install ‘requirements.txt` before anything else to make testing easier and save space in case of any errors.
- `COPY ./mysite .` adds the rest of the files from ‘mysite’ to the working directory ‘.’.

34. In ‘docker-compose.yml’ add the following code.

```
version: '3.7'

services:
  web:
    build:
      dockerfile: Dockerfile.web
      context: ./

    restart: unless-stopped

    image: django-app
    container_name: "django-app"

    env_file:
      - .env
    environment:
      - SECRET_KEY=${SECRET_KEY}
      - DJANGO_ALLOWED_HOSTS=${DJANGO_ALLOWED_HOSTS}
      - APP_PORT=${APP_PORT}

    ports:
      - ${APP_PORT}:${APP_PORT}

    command: >
      sh -c "python manage.py makemigrations &&
      python manage.py migrate &&
      python manage.py runserver 0.0.0.0:${APP_PORT}"
```

- Here a service is created called `web` which builds the Dockerfile `Dockerfile.web`
- `env_file` sets the name of the environment file and `environment` maps the environment variables from the .env file.
- `ports:` maps the django server port to the docker container port. In this case it is set to the same port which means that the docker port will listen in and display the content from the Django app on the same port. These could be different, but I made them the same for simplicity.
- Lastly, `command: >` runs the commands listed above which start the django app. Notice how the host name is set to `0.0.0.0`, which listens for any host, and the port is set to the `${APP_PORT}` env variable. This is all done in the docker-compose file so that the env variables can be grabbed at runtime. This makes the container more secure.

35. Go to ‘settings.py’ in ‘mysite’ and make the following changes to `DEBUG` and `ALLOWED_HOSTS`:

```
# SECURITY WARNING: don't run with debug turned on in production!
DEBUG = False

ALLOWED_HOSTS = os.environ.get('DJANGO_ALLOWED_HOSTS', '').split(',')
```

    - Set `DEBUG = False` once your application is ready for production.
    - `ALLOWED_HOSTS` is set to the env variable `'DJANGO_ALLOWED_HOSTS'` which is a list of allowed host names that are separated by a comma. There must NOT be any spaces between the host names.
36. Be sure to add the env variables to your ‘.env’ file. For testing without a container, add the .env file to the ‘mysite’ folder that contains ‘manage.py’. For testing with docker, be sure to add the .env file to the parent directory that contains the ‘docker-compose.yml’ file. Of course, always be sure to add your .env file to the .gitignore file before pushing ANY changes to GitHub. The ALLOWED_HOSTS and APP_PORT should look like the code snippet below in your .env file.

```
DJANGO_ALLOWED_HOSTS="localhost,127.0.0.1"
APP_PORT="8000"
```

    - `127.0.0.1` is the default host name for Django apps and `8000` is the default port.
    - `APP_PORT` won’t be used when testing without Docker. You can just define the port in the command run: `python manage.py runserver 8100`
37. Lastly, to start the app using docker-compose, from the parent directory (directory that contains the docker-compose.yml file) run the command `docker-compose build` to build the containers and then run `docker-compose up` to run the container in the current terminal or `docker-compose up -d` to run the container in the background.
    - To access the app visit http://localhost:${APP_PORT}/1 where ${APP_PORT} is the port that you defined.

Here is the link to the GitHub repo: [BigToe33/cosc4100-technical-tutorial: Simple Django Application Containerized Using Docker (github.com)]