EC2 Django Deployment

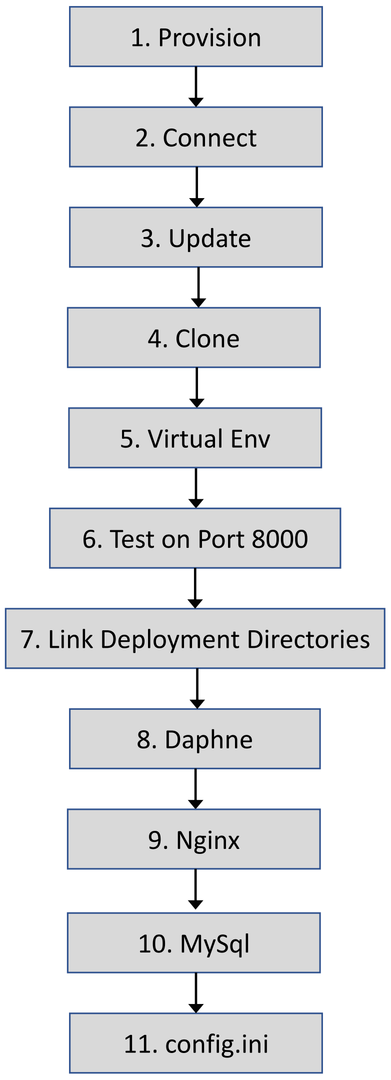
Web Application Development Course
Carnegie Mellon University
Updated on March 15, 2025

There are many ways to configure an EC2 instance to deploy your Django app. See the Django documentation for a discussion of the many options:

- https://docs.djangoproject.com/en/5.1/howto/deployment/

This guide uses the Daphne application server and the Nginx HTTP server for deploying the class examples or your homework. Daphne supports WSGI and ASGI, it will also work with all the class examples, including the WebSockets example (whereas application servers such as Gunicorn and Apache's mod_wsgi only support WSGI which cannot handle WebSockets).

Here is an outline of the steps you need to deploy your application using this guide.



1. Provision the EC2 Instance

Ubuntu is not the default Unix.
Be sure to select Ubuntu Unix.

Go to `aws.amazon.com`:

- Create account
- Create an EC2 Ubuntu Instance. The Ubuntu defaults are fine for the homework, unless you're using VSCode Remote Development Extensions. The defaults are version 24.04, size t2.micro, 8gb of storage. See notes below for larger needs (e.g., when using VSCode).
  - Review & Launch
  - Create Key Pair, download as <name>
    - This will download a PEM file you can use to connect (option #2 below)
  - When provisioning another time, you may reuse existing key pairs / security groups.
- Enable ports 80 and 8000
  - Select your server instance
  - Under the Security tab, click on your security group
  - Select Security Groups → <your security group>
    - Click "Inbound rules" Tab
    - Click "Edit inbound rules" → Add Rule
    - Add HTTP Rule (Port 80) with Source 0.0.0.0/0
    - Add Custom TCP Rule for Port 8000 with Source 0.0.0.0/0
    - Save

Careful not to delete Port 22.
That's for your SSH connections.

Notes on size settings when creating instances:

- The VSCode Remote Development Extensions seem to put a heavy load on the EC2 instance. You should allocate at least a t2.medium instance if using these.
- The t2 instances are using older technology and seem to be slower and less reliable than other choices, but t2.micro is free for new AWS developers. When using t2 instances for the course servers, they seem to crash once or twice a week. We believe this is due to load imposed by bots looking for vulnerabilities. We currently run www.cmu-webapps.org on an m7a.medium instance and it's been very stable. We have been running the AutoGrader servers on c7a.large instances, and they have been very stable. Pricing info can by found here: https://aws.amazon.com/ec2/pricing/on-demand/.
- The 8gb is the default amount storage. That will be fine for the homework and probably the project. For comparison, we allocate 16gb for the course server (www.cmu-webapps.com) and 100gb for AutoGrader instances (but we could get by with 70gb).

Notes on connecting to your instance, and its IP addresses:

- The EC2 console can be used to stop/start your EC2 instance, to find its IP address and its security group, and even to connect to the instance.
- When stopping and then starting your instance, the IP address will change.
- When working on your project, you may wish to allocate and assign an elastic IP address to your instance so that the IP address will not change. (AWS charges for elastic IP addresses that are not assigned to a running EC2 instance (to keep people from hoarding IP addresses).

2. Connecting to your EC2 instance

Here are <u>three options</u> you can use to access a shell on your EC2 instance:

Just works

a. In the EC2 console on aws.amazon.com, select your instance and click on Connect => "EC2 Instance Connect". This lets you use a shell from your browser.

Easier to re-connect

b. Connect to your instance from using SSH (on your laptop) and the key in your downloaded PEM file:
   i. Remove write access to the PEM file (on MAC: `chmod 400 <name>.pem`)
   ii. Then: `ssh -i <name>.pem ubuntu@<ip-address>`

Many steps but easiest reconnect & repo cloning

c. Connect to your instance with SSH using your laptop's public key:
   1. Set up SSH authentication for GitHub as per our Git Quickstart guide.
   2. Connect to your instance using options 2a or 2b, above.
      - Using vim (or other editor), add your SSH public key (as an additional line) to your instance's `~/.ssh/authorized_keys` file
      - Exit the instance's shell
   3. On your laptop, create (or add to) your `~/.ssh/config` file with these lines:

```
Host <nickname>
    Hostname <ip-address>
    User ubuntu
    ForwardAgent yes
```

   4. Now reconnect to your instance this way: `ssh <nickname>`
      - Test that GitHub recognizes your forwarded identity on your EC2 instance using: `ssh -T git@github.com`
      - If this command does not show your identity, return to your computer and run `ssh-add`. Then reconnect to your instance and see if the above command now works.

A note about usernames on your EC2 Ubuntu instance:
- The default user is "ubuntu". The connection instructions above will log you in as this user. We'll configure the Daphne server to run as the "ubuntu" user so that it has access to everything in `/home/ubuntu`.
- Many configuration steps (below) need to run with "superuser" privileges. On Ubuntu (and other Unix systems), the superuser is "root". Prefixing comments with "sudo" (for superuser do) will run commands as root.
- There's another user called "www-data" which the HTTP Server uses.

A note about file protections on Unix systems:
- Files are owned by a user.
- Files are also in a group. Group names are typically the same as usernames.
- Read, write, and execute permissions for files can be separately set for owner, group, and everyone else.
- The `ls -l` unix command will show a file's owner, group, and permissions.
- We'll configure the HTTP Server to run as the "www-data" user but in the "ubuntu" group so it will have access to files in `/home/ubuntu`.

3. Update the Software on your EC2 Instance

In shell on EC2 Instance, run the following commands:

```
sudo apt update
sudo apt upgrade
sudo apt install python3-pip python3.12-venv
sudo reboot
```

4. Cloning Repo

Just works

If you are connected to your EC2 instance using methods 2a or 2b, you can use either:

- Option 1: Use an HTTPS URL (https://github.com/cmu-webapps/...). You will be prompted for your GitHub username and "password". You must use a personal token for the password.
- Option 2: Create a new public/private key pair on the your EC2 instance and register the public key as (another one of) your public key(s) on GitHub. (See the SSH Credentials section in the Git Quickstart guide for the steps.) Then cloning SSH URLs will work.

After setup, easiest to use if you access GitHub often

If you set up Agent Forwarding when you connected to your EC2 instance using method #2c above, you can use:

- Option 3: `git clone` commands with SSH URLs should just work.

Examples of HTTPS URLs:

```
git clone https://github.com/cmu-webapps/image-example.git
git clone https://github.com/cmu-webapps/websockets-example.git
git clone https://github.com/cmu-webapps/<andrewid>.git
```

Examples of SSL URLs:

```
git clone git@github.com:cmu-webapps/image-example.git
git clone git@github.com:cmu-webapps/websockets-example.git
git clone git@github.com:cmu-webapps/<andrewid>.git
```

5. Create a Virtual Environment

In shell on EC2 Instance, run the following commands:

```
python3 -m venv ~/venv
```

Enter the virtual environment:

```
source ~/venv/bin/activate
```

(When reconnecting to your EC2 instance, remember to re-enter the venv before running Python.)

6. Test on Port 8000

1. Set the current working directory to the folder for the Django project you want to run.

   Examples:

   ```
   cd image-example
   ```

   or

   ```
   cd <andrewid>/hw7
   ```

2. Install Django and any other Python packages the project uses.

   Examples:

   ```
   pip install django
   ```

   or

   ```
   pip install -r requirements.txt
   ```

3. Initialize the database:

   ```
   python manage.py makemigrations
   python manage.py migrate
   ```

4. Add your IP address to ALLOWED_HOSTS in settings.py, if necessary.

5. Start the server. (Don't forget 0.0.0.0 so it listens for external requests.)

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

6. In your web browser, visit `http://<ip-address>:8000`
   - You should see the Django application running, using SQLite for the DB

7. Stop Django (type Control-C).

7. Link Deployment Folders

To simplify the steps in the configuration files set up in the later sections, below, we will make symbolic links to your Django application folder and your static folder (or folders). This will allow you to copy and paste the configurations in Sections 8 and 9 with less editing.

1. Create a symbolic link to your Django project folder. This is the folder that contains manage.py. The link must be called "project".

   ```
   cd ~
   ln -s <project-folder> project
   ```

   For a course example, such as the image-example, the `<project-folder>` would be `image-example`

   For HW7, `<project-folder>` would be `<andrewid>/hw7`

2. Create a symbolic link to your static file folder.

   ```
   cd ~
   ln -s <project-folder>/<appname>/static
   ```

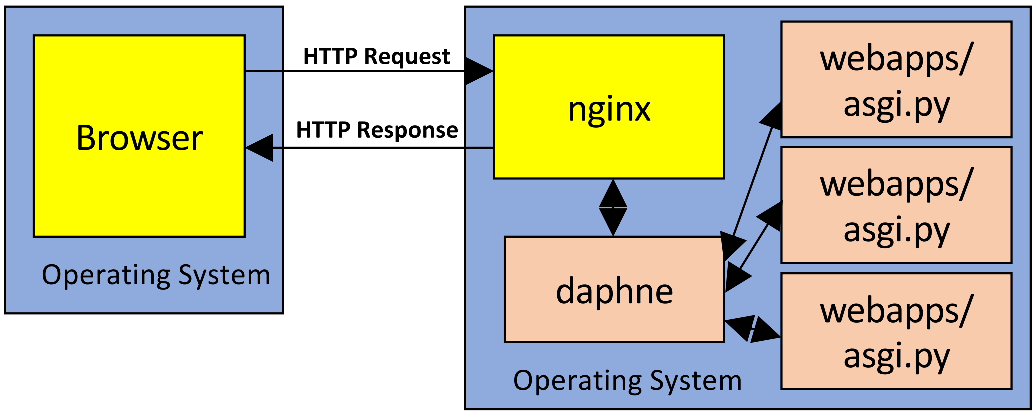
   Or, if you have multiple applications in your project, you should create a `static` folder and then link each applications' static subfolders into this folder. For example, the image-example has the `picture_list` application as well as the `welcome` application. So it would be like this:

   ```
   mkdir ~/static
   cd ~/static
   ln -s ~/image-example/picture_list/static/picture_list
   ln -s ~/image-example/welcome/static/welcome
   ```

If done correctly, running `ls ~/project` should show the files at the top of your project folder and `ls ~/static` should show your static files (or if you repeated the appname in the applications' static folders, then `ls ~/static/<appname>` should show your static files).

8. Daphne

This guides sets up Daphne as the application server to run your Django code.



Note: Daphne can be used for traditional (WSGI) web apps as well! It supports both ASGI & WSGI apps, so this guide will also work for apps that do not use WebSockets.

8.1 Install and Test Daphne

If your application is using WebSockets, you may have installed Daphne when you installed the packages to test on port 8000, but if not:

```
pip install daphne
```

Now we'll ensure that Daphne works and serve our app using the following command:

```
cd ~/project
daphne webapps.asgi:application -b 0.0.0.0 -p 8000
```

Visit `http://<ip-address>:8000` to verify that the application is running.

Note that the static files will not be served yet. (So you're CSS and any JS code won't work, unless these files have already been cached from testing using `runserver`. You can open the browser devtools with "Disable Cache" checked to ensure you're seeing that these files are not being served.)

Type Ctrl-C to stop Daphne.

8.2 Daemonizing Daphne using supervisord

Or use the command in the gray box below to copy this config file from the course server.

First install `supervisord` using the command

```
sudo apt install supervisor
```

Let's create a configuration to run make Daphne run Django:

```
sudo vim /etc/supervisor/conf.d/daphne.conf
```

```
[fcgi-program:daphne]
# TCP socket used by Nginx backend upstream
socket=tcp://localhost:8001

user=ubuntu
directory=/home/ubuntu/project
environment=PATH="/home/ubuntu/venv/bin:%(ENV_PATH)s"

# Each process needs to have a separate socket file, so we use process_num
command=daphne webapps.asgi:application -u /home/ubuntu/daphne-
sockets/daphne%(process_num)d.sock --fd 0 --access-log - --proxy-headers

# Number of processes to startup, roughly the number of CPUs you have
numprocs=3

# Give each process a unique name so they can be told apart
process_name=daphne%(process_num)d

# Choose where you want your log to go
stdout_logfile=/home/ubuntu/daphne.log
redirect_stderr=true
autostart=true
autorestart=true
```

Use this curl command to download the above configuration file from our course server and save it in the right place:

```
sudo curl -k https://www.cmu-webapps.org/deployment/daphne.conf -o /etc/supervisor/conf.d/daphne.conf
```

Create the folder for Daphne sockets to use for interprocess communication:

```
mkdir ~/daphne-sockets
```

Save the configuration and exit. Tell the supervisor to update its registry of services:

```
sudo supervisorctl reread
sudo supervisorctl update
```

Run the status command below to see that Daphne is running (and restart when fixing errors).

```
sudo supervisorctl status daphne:*
sudo supervisorctl restart daphne:*
```

To look for errors, see the log file: `/home/ubuntu/daphne.log`

Use the cat and/or tail commands to inspect the log file, e.g.:

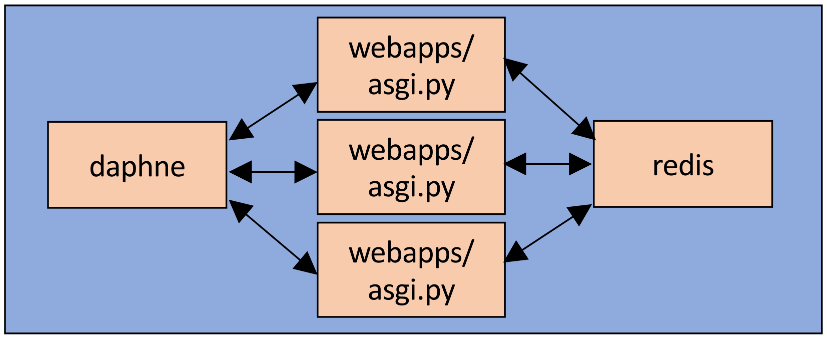
```
cat ~/daphne.log
tail -100 ~/daphne.log
```

8.3 Additional Configuration Steps for WebSockets Deployment

(If your web application is not using WebSockets, skip this part and go on to Section 9.)

When developing/testing with the Django development server, there is only one process running your web application so we can use the Channels `InMemoryChannelLayer`.

When deploying a WebSockets application running with multiple Django processes (accessed via `asgi.py`), you need to use a Channels backend which will support distribution of the WebSockets messages across all the Django processes. We recommend using the `RedisChannelLayer` and running the Redis server.



Install the Redis server (which will also start it):

```
sudo apt install redis-server
```

Install the Channels backend for Redis (in your virtual environment):

```
pip install channels_redis
```

Modify your settings.py file to configure Channels to use Redis:

```
CHANNEL_LAYERS = {
    "default": {
        "BACKEND": "channels_redis.core.RedisChannelLayer",
        "CONFIG": {
            "hosts": [("localhost", 6379)],
        },
    },
}
```

Restart Daphne (which is running instances of Django, as you've modified Django settings):

```
sudo supervisorctl restart daphne:*
```

9. Nginx

We now need to put an HTTP server in front of Daphne. We'll use Nginx. Nginx will directly serve the static files and to forward other requests to Daphne, which you set up, in Section 8.2, to listen for these requests on your instance's port 8001.

Nginx will listen for incoming HTTP requests on port 80 (the default port that browsers use for HTTP). When we cover SSL, we'll give you additional steps to configure Nginx to listen for HTTPS requests on port 443 (the default for HTTPS).

(If you don't recall the note on user ids in the box at the bottom of Section 2, you re-read it now.)

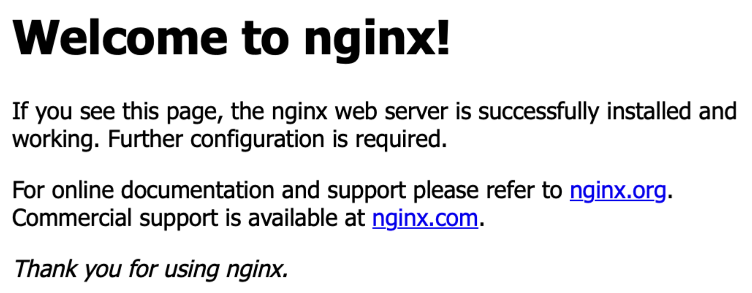
9.1 Install the Nginx Server

In shell on EC2 Instance:

```
sudo apt install nginx
```

In web browser, visit `http://<ip-address>`

- You should see the Nginx splash screen



Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

9.2 Configure Nginx to Serve your Django App and static files

Before we setup nginx for our app, we must get rid of the default nginx config.

```
sudo rm /etc/nginx/sites-enabled/default
```

Or use the command in the gray box below to copy this config file from the course server.

Now let's setup our own config. Paste the text in the box below into this file. Save and exit.

```
sudo vim /etc/nginx/sites-available/my-site.conf
```

```
upstream daphne {
    server localhost:8001;
}

server {
    listen 80;

    # If using a domain name, uncomment the line below and enter domain name
    # server_name my.domain-name.com;

    location /static/ {
        alias /home/ubuntu/static/;
    }

    location / {
        try_files $uri @proxy_to_app;
    }

    location @proxy_to_app {
        proxy_pass http://daphne;

        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";

        proxy_redirect off;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;

        # More info: http://en.wikipedia.org/wiki/X-Forwarded-For
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Host $server_name;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Use this curl command to download the above configuration file from our course server and save it in the right place:

```
sudo curl -k https://www.cmu-webapps.org/deployment/my-site.conf -o /etc/nginx/sites-available/my-site.conf
```

Now we need to create a symbolic link to this file in the sites-enabled folder:

```
cd /etc/nginx/sites-enabled
sudo ln -s ../sites-available/my-site.conf
```

By default, Nginx runs as the www-data user. To allow Nginx to access the static files in `/home/ubuntu/static`, edit the Nginx's main config file to run with "ubuntu" group access:

```
sudo vim /etc/nginx/nginx.conf
```

- Change the first line from:
  ```
  user www-data;
  ```
  to:
  ```
  user www-data ubuntu;
  ```

- Save and exit

Restart Nginx:

```
sudo service nginx restart
```

In web browser, visit `http://<ip-address>`

- You should now see your Django app running via Nginx
- Static files should work

To check the status of Nginx use:

```
sudo service nginx status
```

If you're getting errors with Nginx, check it's log file: `/var/log/nginx/error.log`

10. Install and Configure MySQL

In the shell on EC2 Instance:

```
sudo apt install pkg-config
sudo apt install mysql-server
sudo apt install libmysqlclient-dev

pip install mysqlclient
```

Don't forget you need to be in your venv.

```
sudo mysql
    create user ''@'localhost' identified by '';
    grant all privileges on *.* to ''@'localhost';
    quit;
```

You may specify a username and password if you wish.

```
mysql
    create database django character set utf8mb4;
    quit;
```

If you set username and password, use this command:
```
mysql -u <user> -p
```
to be prompted for your password.

Oddly, MySQL's utf8 character set only handles 3-byte Unicode. They have added utf8mb4 to handle 4-byte.

```
cd ~/project
vim webapps/settings.py
```

- Change DB config to use MySQL:

```
DATABASES = {
    'default': {
        'OPTIONS': {'charset': 'utf8mb4'},
        'ENGINE': 'django.db.backends.mysql',
        'NAME': 'django',
        'USER': '',
        'PASSWORD': '',

    }
}
```

If you set username and password, enter them here.

```
python3 manage.py migrate
sudo supervisorctl restart daphne:*
```

In web browser, visit `http://<ip-address>`

- Model data should now be stored in the database.

To view the data, in the shell on the EC2 Instance:

```
mysql
    use django;
    show tables;
    select * from <table_name>;
    quit;
```

If you set username and password, use this command:
```
mysql -u <user> -p
```
to be prompted for your password.
To use full Unicode from the mysql command line to run (in mysql):
```
set charset utf8mb4;
```

11. Removing Secrets from your Repo

You should not store secret data in your repo, such as passwords and other encryption keys. For EC2 deployments, you should put the secrets in a `config.ini` file like this:

```
[Django]
secret=p8xyz5&fxyzhyb5fxyz-@t#g!2=_yh_#^0y_9xyzk7+tgq+ts
```

Note that the % character is special in `.ini` files, so if you have any `%` characters in your secret, you must double it (use `%%`).

Then read the secrets in from your `config.ini` file using the Python `ConfigParser` class. For Django's `SECRET_KEY` in `settings.py`, it would look like this:

```
from configparser import ConfigParser
...
CONFIG = ConfigParser()
CONFIG.read(BASE_DIR / "config.ini")
...
SECRET_KEY = CONFIG.get("Django", "secret")
```

Do the same for any other secrets, as well as changeable configuration parameter that you don't want to hard code into your Python files, such as database usernames and passwords, authentication keys to cloud-based services, etc. You can have multiple sections in your .ini file, but starting each section with a new header in square brackets, like this:

```
[MYSQL]
user=root
password=monkeybreath
```

Since the `config.ini` file, must not be in your repo, we document its format with a "sample" file, such as `config.ini.sample`, which doesn't contain the secrets. Example:

```
# To generate a new secret key, use get_random_secret_key():
#    from django.core.management.utils import get_random_secret_key
#    print(get_random_secret_key())
# Warning: The % character is special to ConfigParser - use %%
# Warning: Changing secret keys invalidates existing sessions,
#          so may need to delete your DB tables and re-migrate

[Django]
secret=
```

12. Additional notes

The AutoGrader can't see the code you're actually running on your server. It checks the code in your repo on GitHub. Thus, it is important for you to push all the changes made in `webapps/settings.py` to your GitHub repo.

To upload files from your laptop to your server (e.g., a config.ini file), you can use `sftp`:

```
sftp –i <name>.pem ubuntu@<ip-address>        (or sftp <nickname>)
    cd <andrewid>/hw7
    put config.ini
    exit
```

To refresh your deployed app with changes that have been pushed to GitHub from elsewhere, simply run:

```
git pull
sudo supervisorctl restart daphne:*
```

If you need to delete all the data in your MySQL database, delete and remigrate with:

```
mysql
    drop database django;
    create database django character set utf8mb4;
    quit;
python manage.py migrate
```

If you need to completely reset your migrations:

```
mysql
    drop database django;
    create database django character set utf8mb4;
    quit;

rm -fr picture_list/migrations
python manage.py makemigrations picture_list
python manage.py migrate
```

For HW7, replace `picture_list` with `socialnetwork`

Note: If your DB is in SQLite, you can completely reset your migrations:

```
rm db.sqlite3
rm -fr picture_list/migrations
python manage.py makemigrations picture_list
python manage.py migrate
```

(Be careful with the `-fr` option on the `rm` command. It recursively deletes files without any prompting, so you need to be careful to get it right. But it should be OK since all your code is safely in your repo on GitHub.)