eecs441§3&4

Chatter Back End

Cover Page

DUE Wed, 09/15, 2 pm

At the end of the lab, you'll be prompted to keep a clean copy of your working solution and it will form the starting point for all subsequent labs.

A hosted server

You need an Internet-accessible server running

- Ubuntu 20.04 or later or equivalent
- Django 3.1.3 or later
- Python 3.8 or later
- Nginx, Gunicorn, and PostgreSQL (latest default version)

You can use a real physical host or a virtual machine on AWS, Digital Ocean, MS Azure, GoogleCloud, AlibabaCloud, etc. as long as they support the above systems.

We will use AWS in this spec though the steps here have been verified to work on a plain Ubuntu 20.04 host. Following are the instructions to set up an AWS instance adapted from EECS 485's tutorial:

Create an account

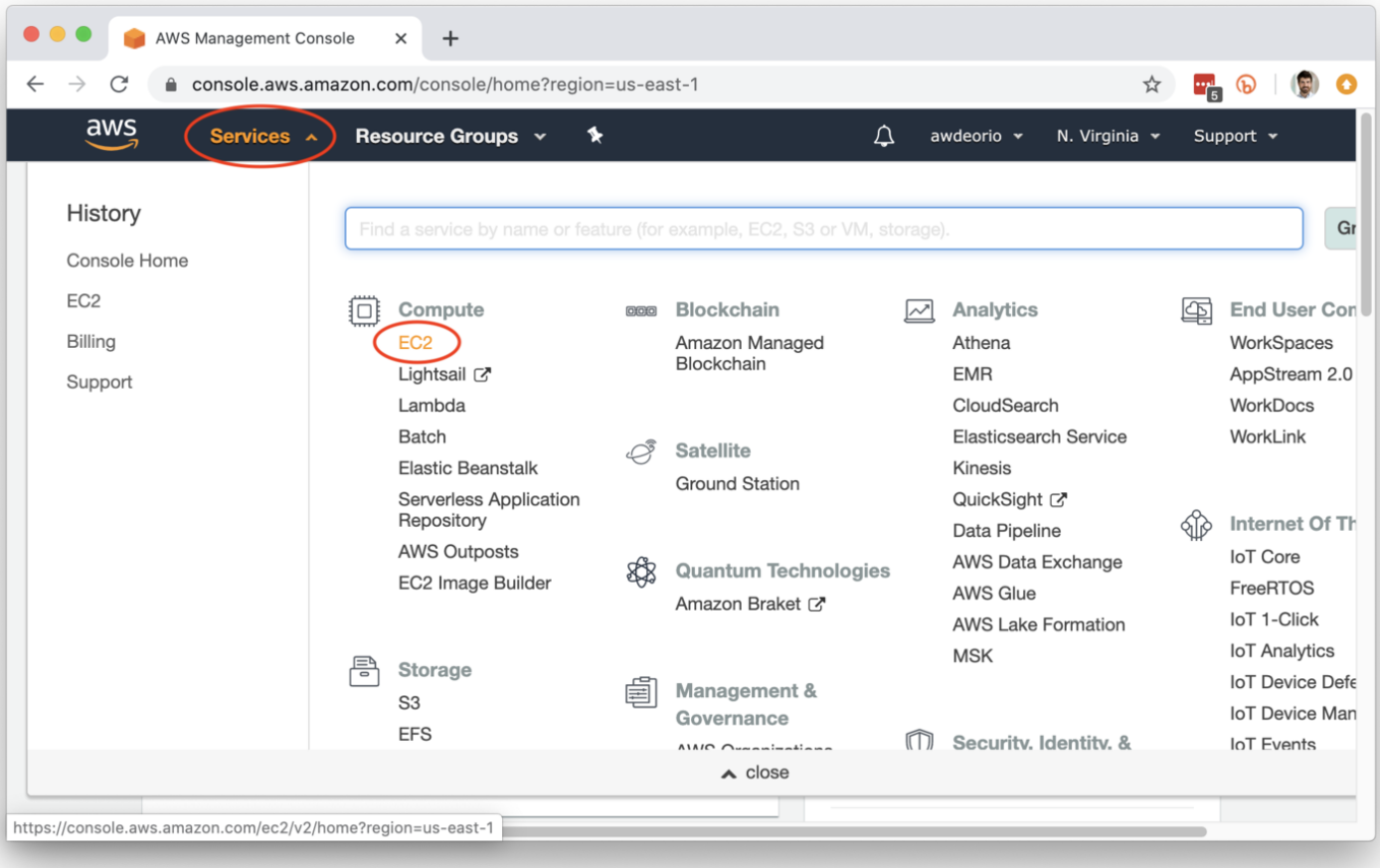
Create an AWS account at the AWS Registration. You should be eligible for their free tier, which means that you will be able to run an instance for free for the duration of the course.

Despite that, you will need to enter a credit card number on the account, even if you only use free tier resources. This is how Amazon charges, in case you request more resources than provided in the free tier. **Do not launch any additional instances** other than the one we specify in order to avoid additional charges.

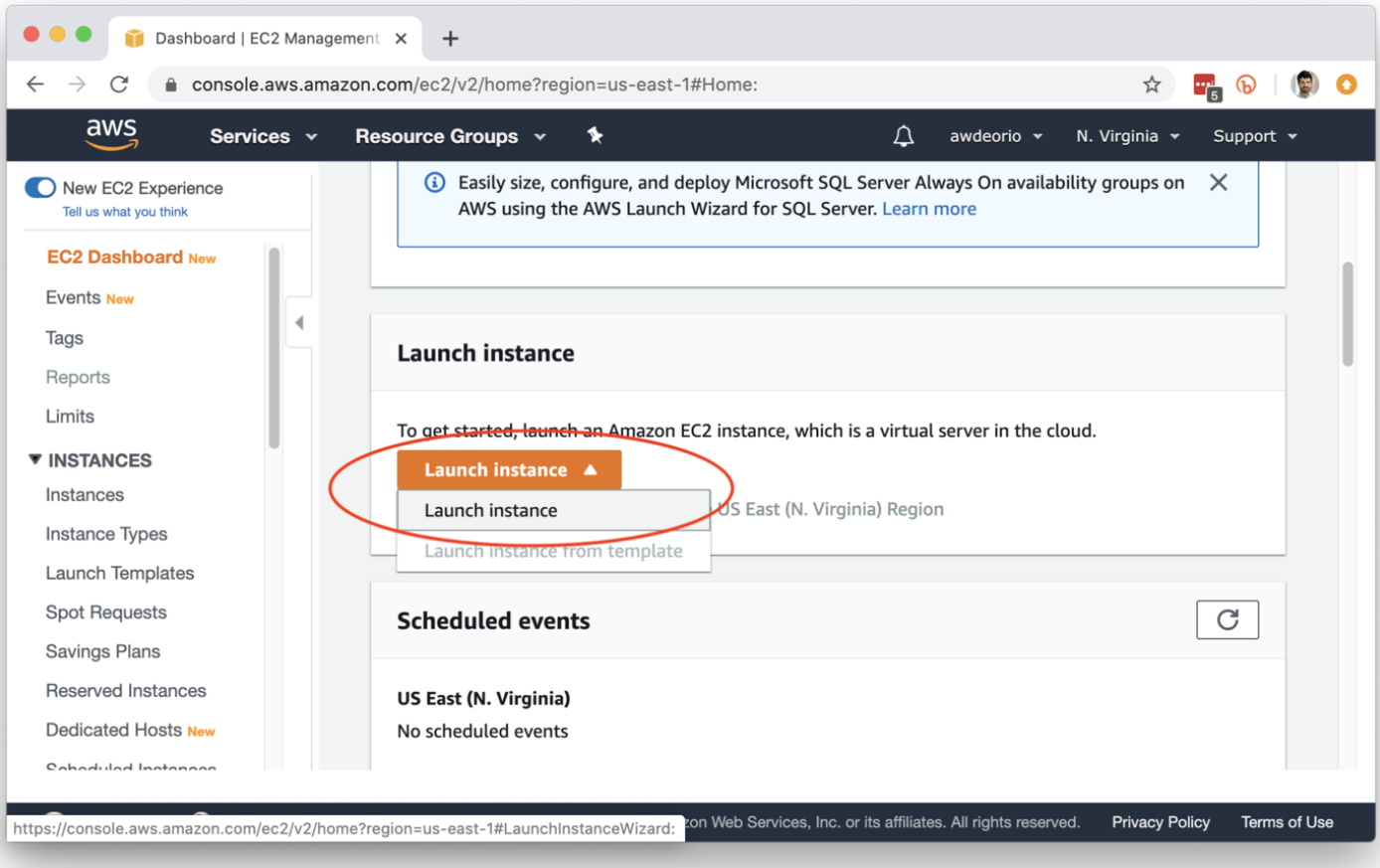
Optionally, you may redeem extra benefits as a student, including $100 in AWS credits.

Start instance

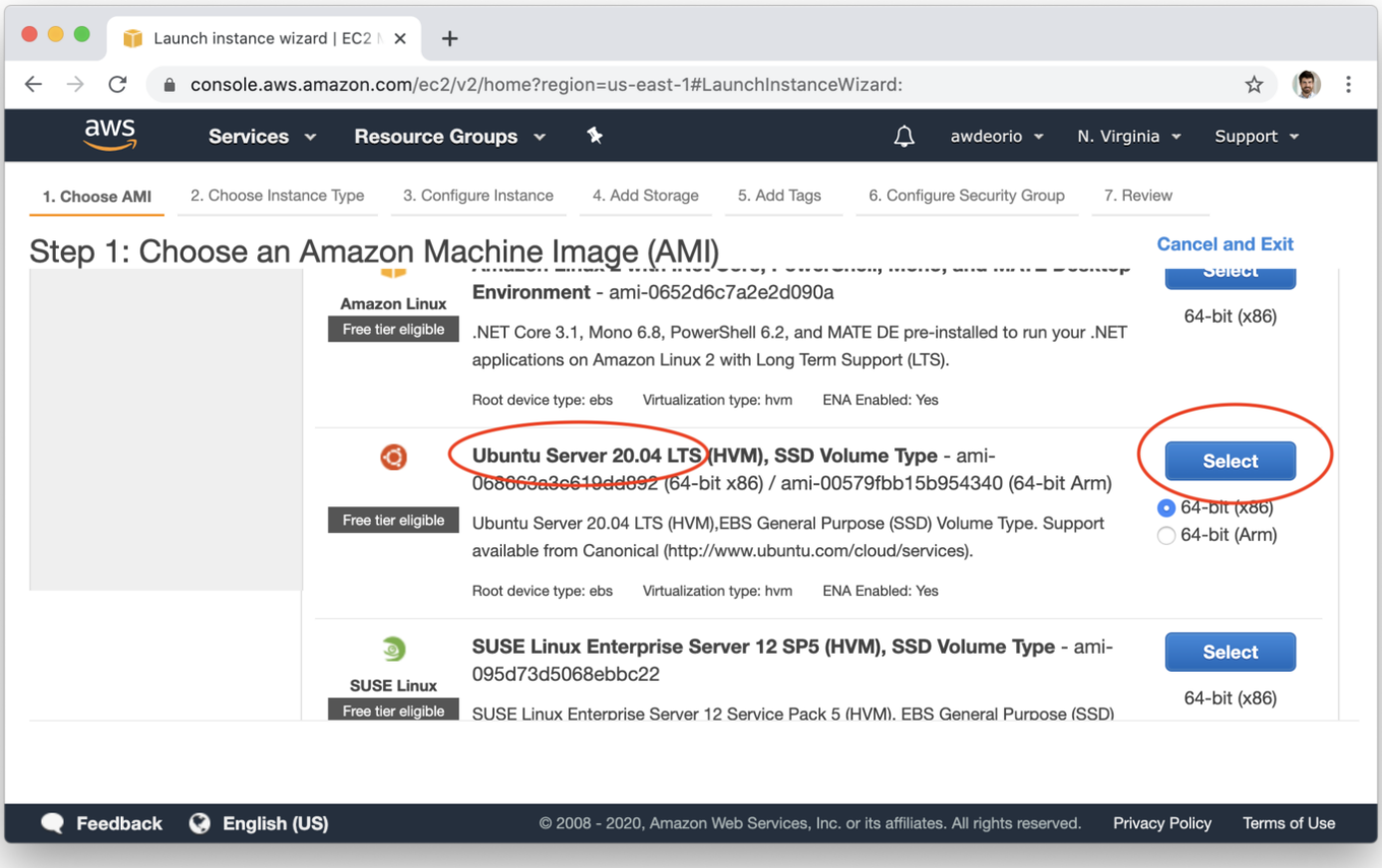
Navigate to the AWS Management Console. Select the "Services" dropdown menu, then "EC2". An EC2 "instance" is a virtual machine running on Amazon AWS hardware.



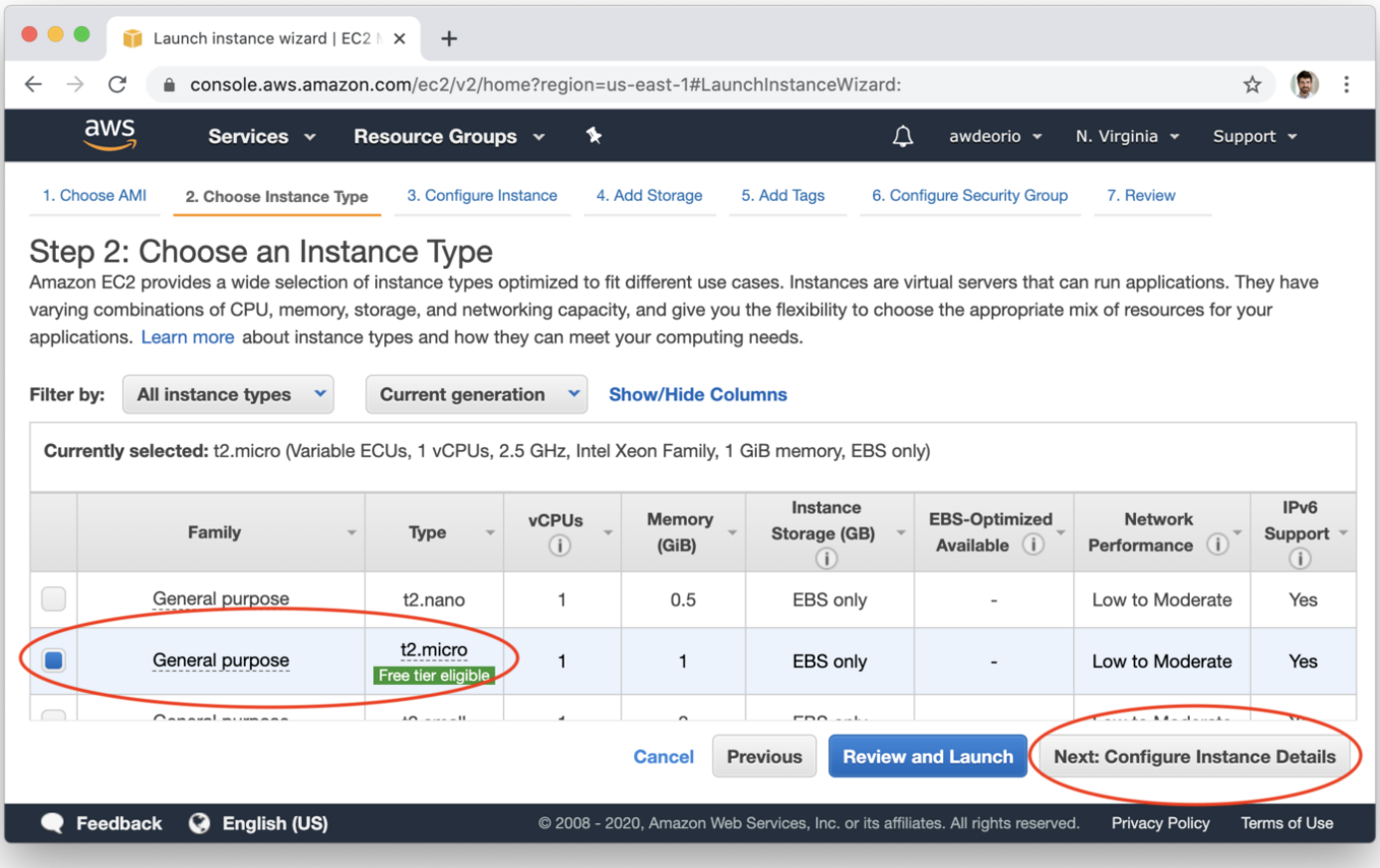
Click launch an instance. It may take a few minutes to initialize before it's running.



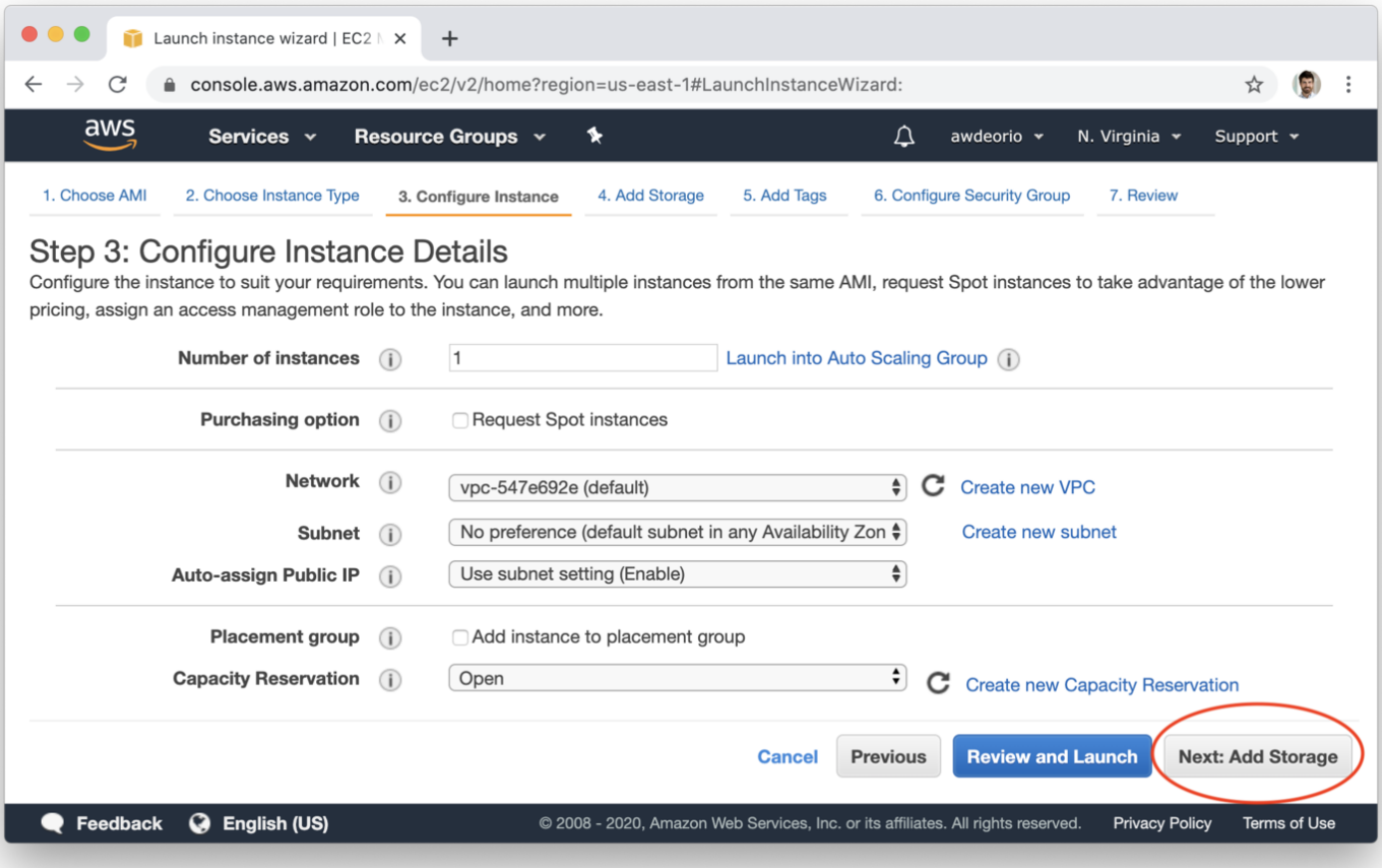
Select the "Ubuntu Server 20.04 LTS" Amazon Machine Image (AMI).



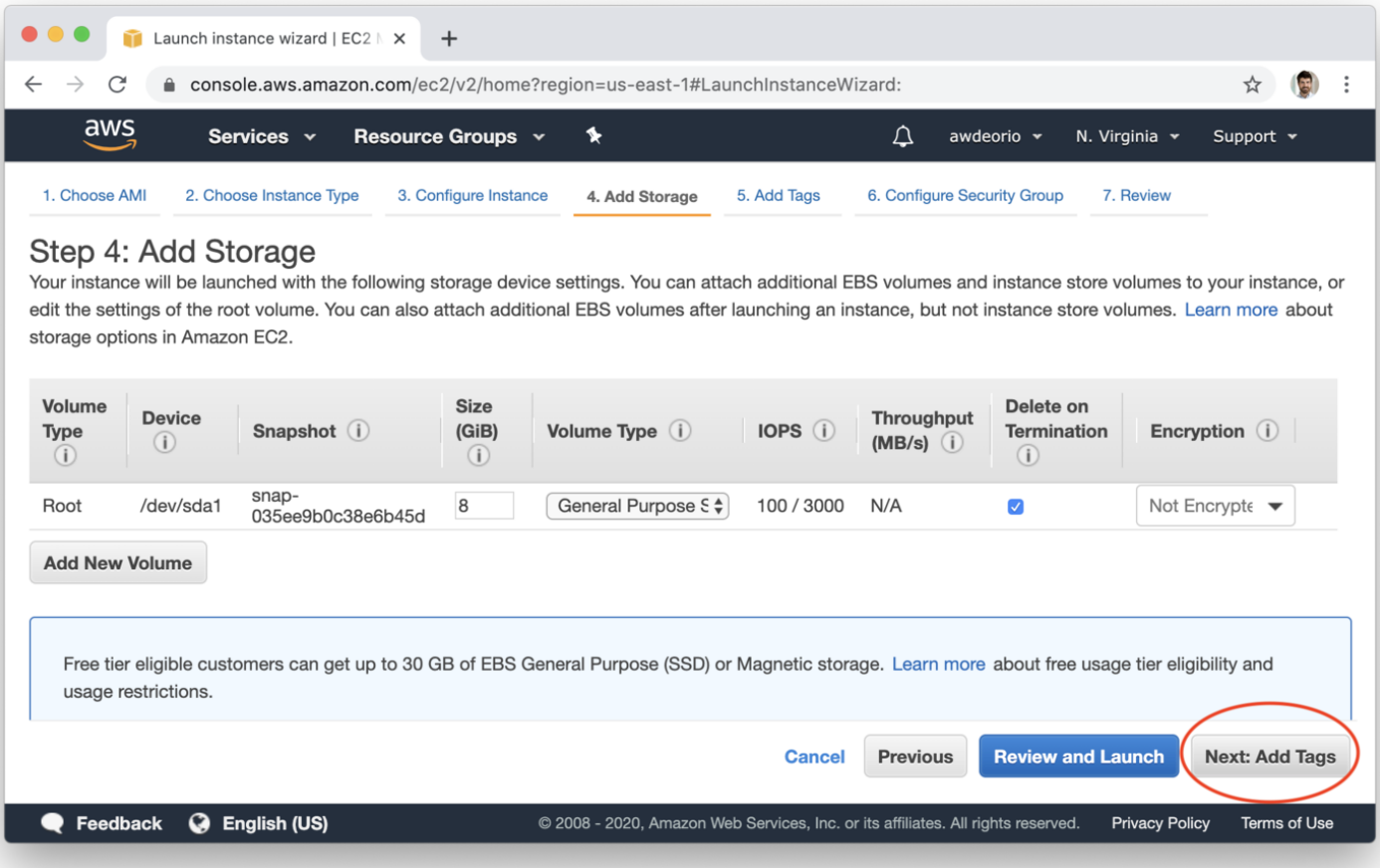
Select the “t2.micro” instance type. You should see “free tier eligible”. Click “Next”.



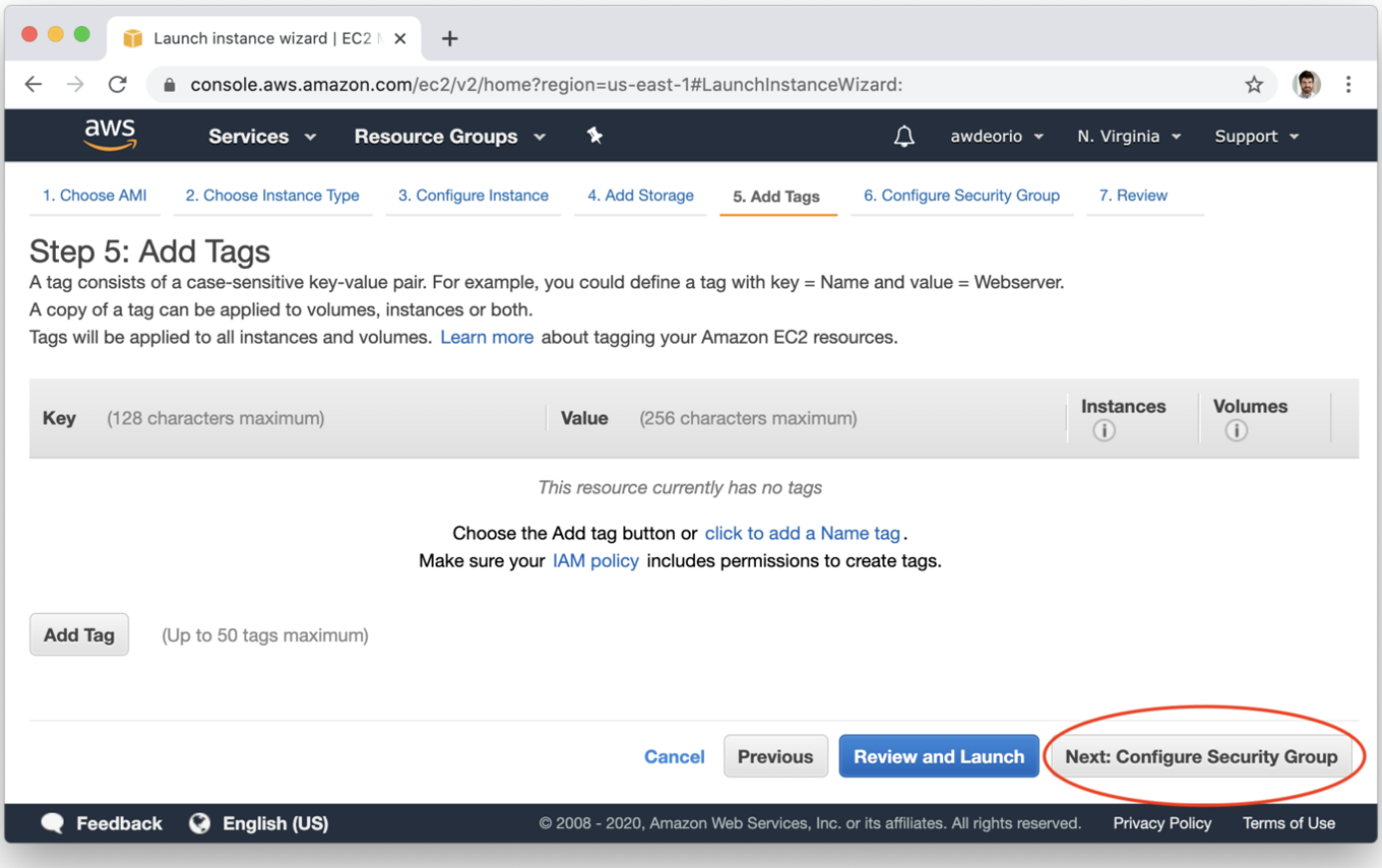
Click “Next”



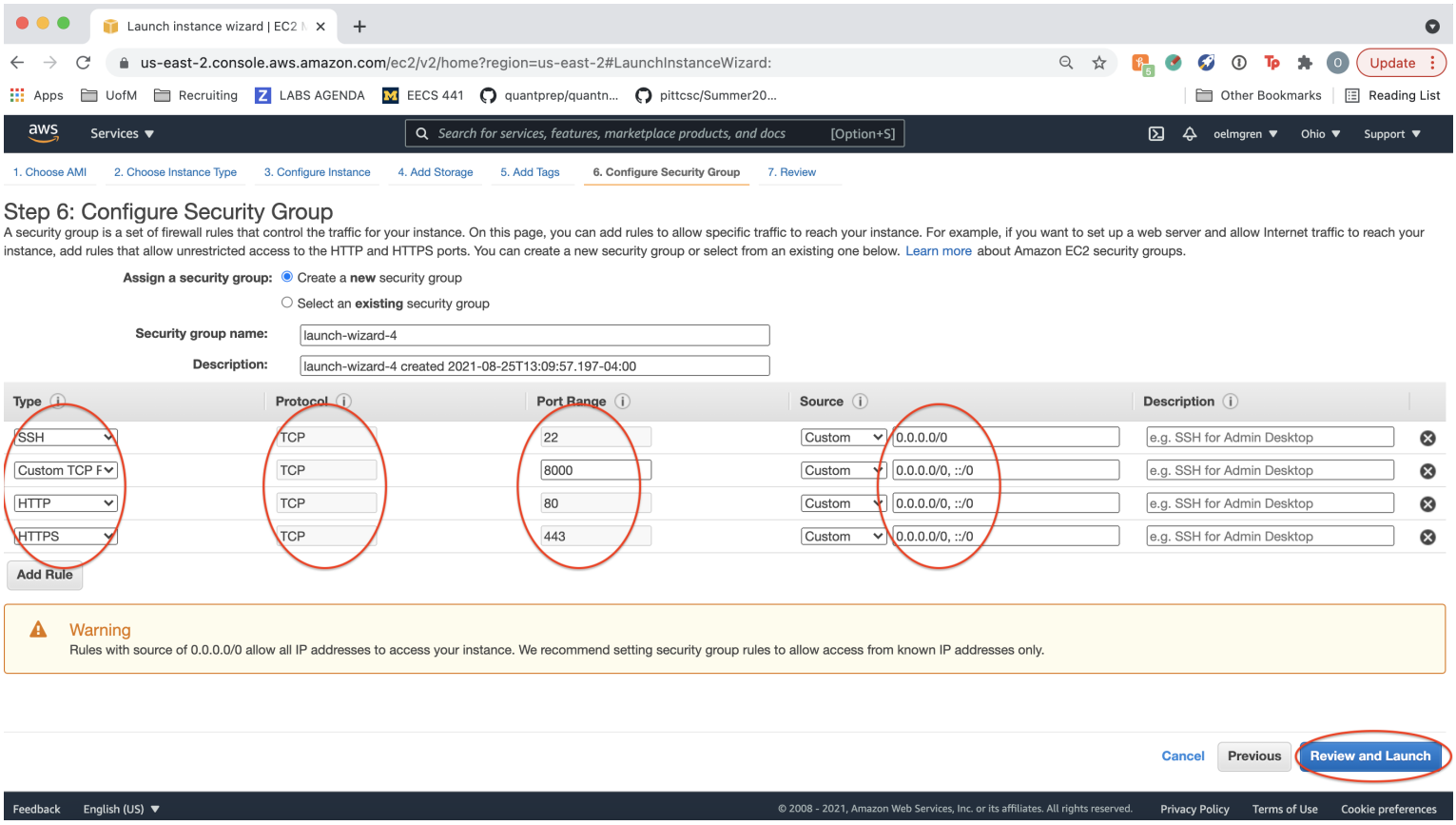
Click “Next”



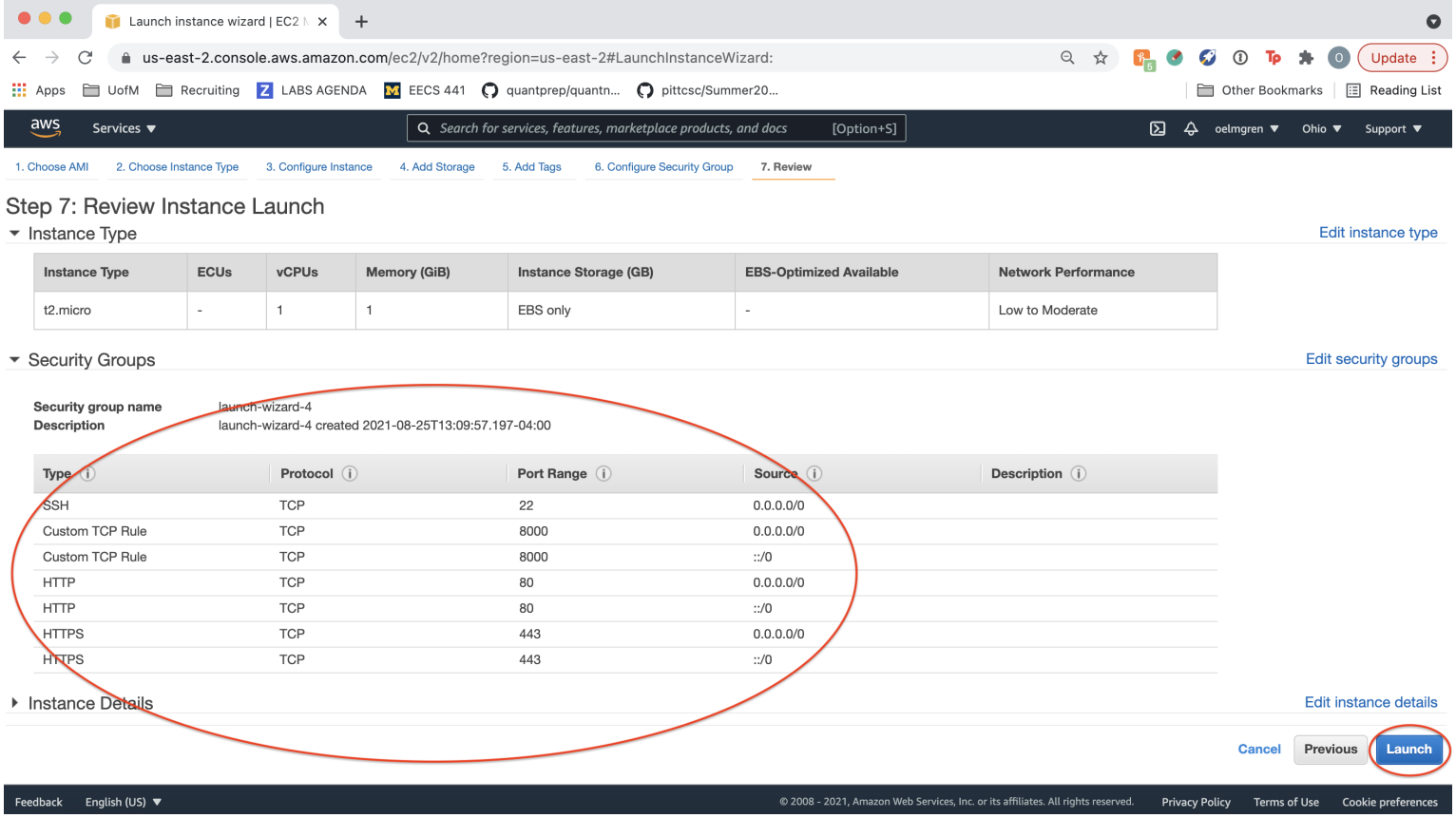
Click “Next”



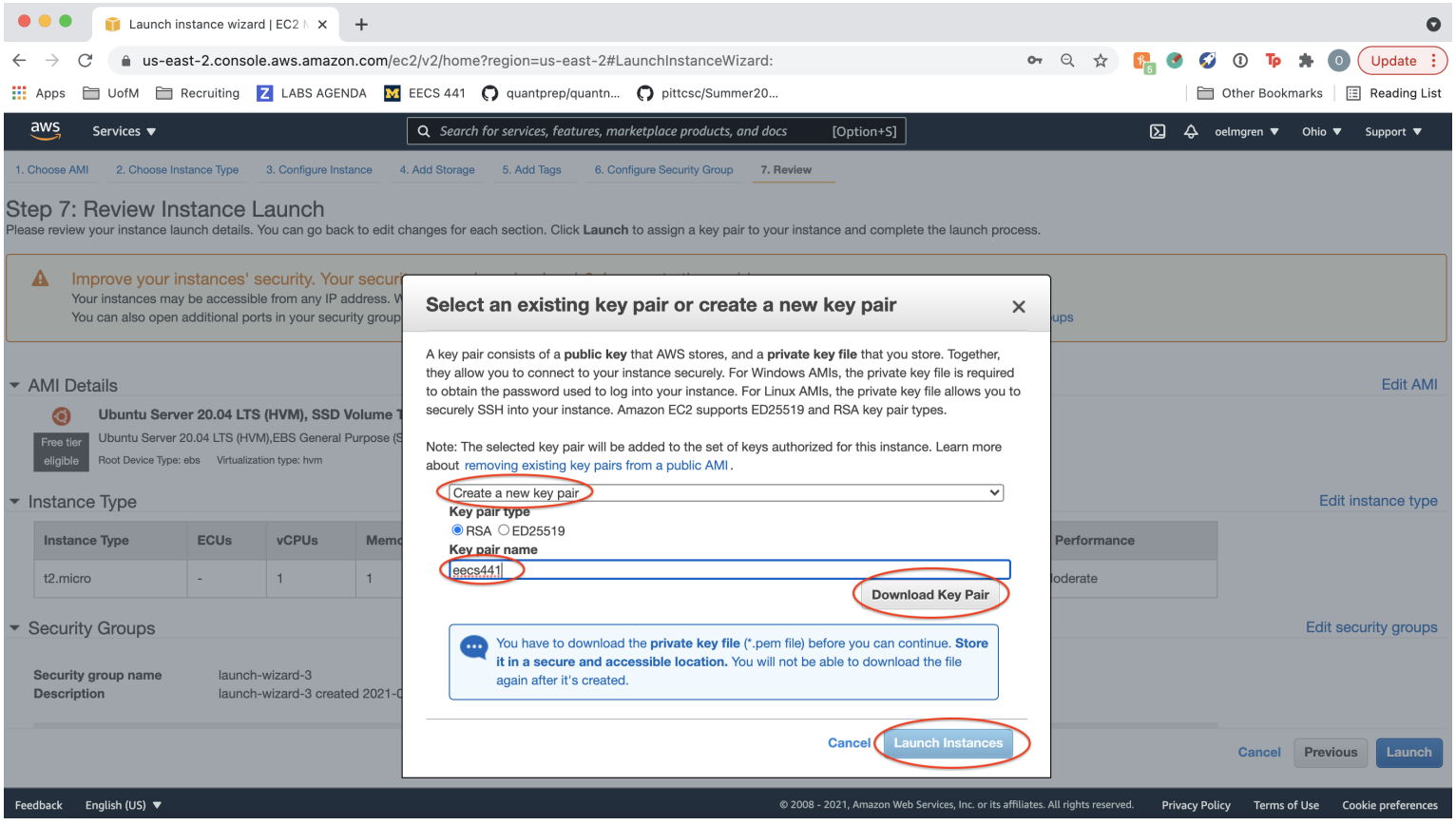
Add a rule to allow SSH, HTTP, HTTPS, and "Custom TCP Rule" for port 8000 (Django) traffic in and out of your instance. Then, click "Review and Launch".



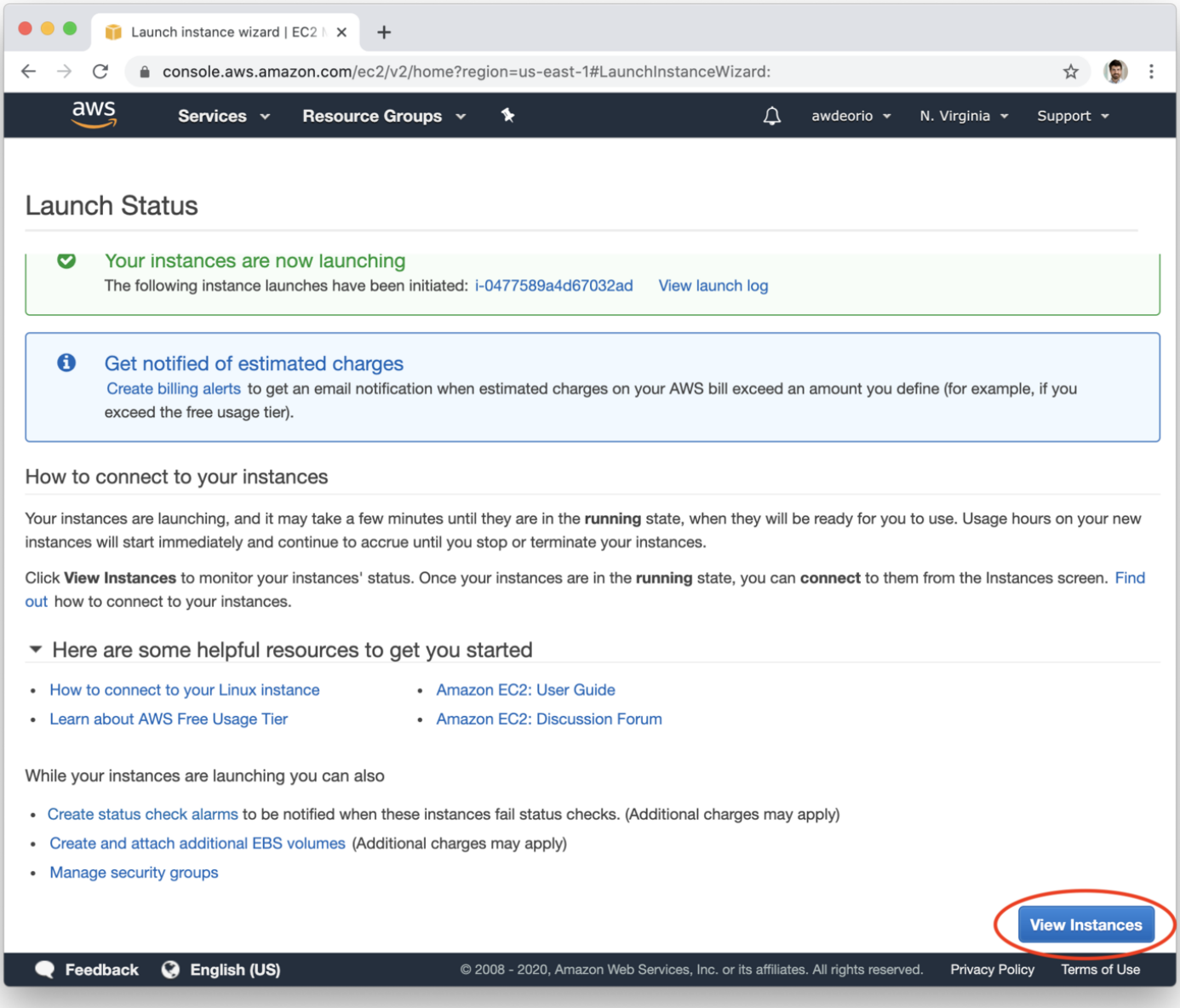
Click "Launch". Ignore a warning about "Improve your instances' security".



When you create an instance, AWS automatically creates user "ubuntu" for you on the instance. Create a key pair for user "ubuntu" and download it. You'll use this later to `ssh` to the instance. Finally, click "Launch Instances".

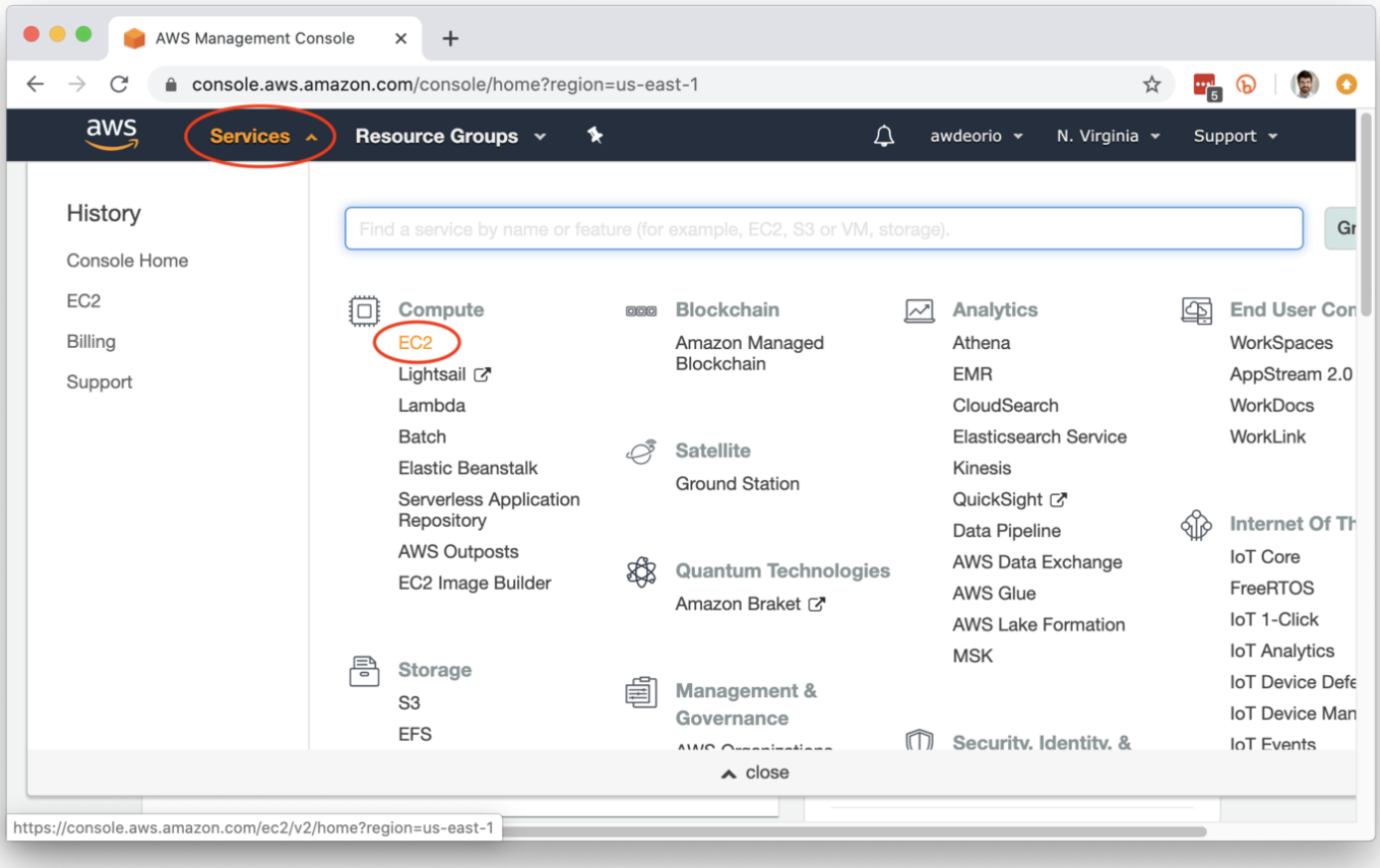


Click "View Instances".

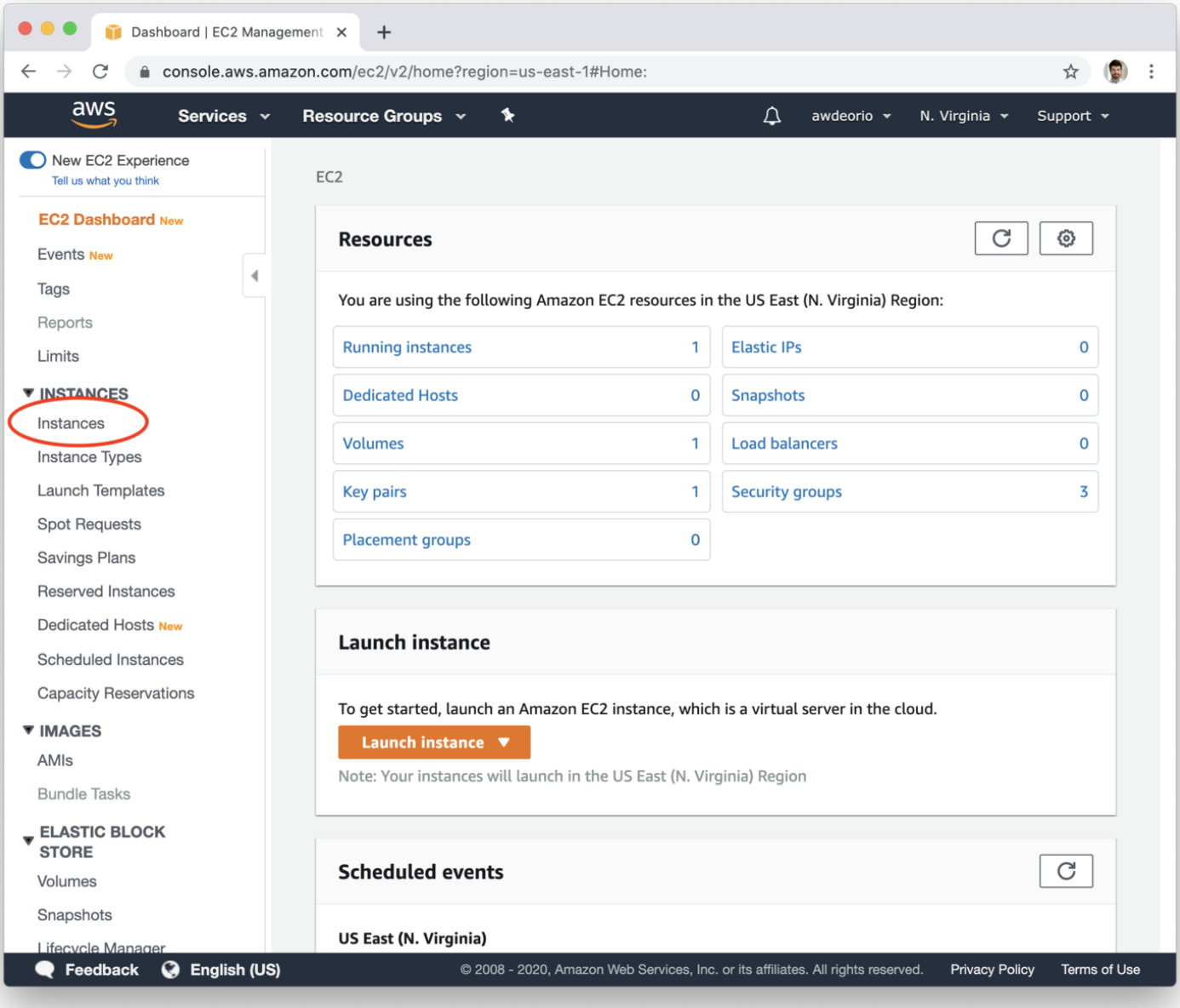


Instance status

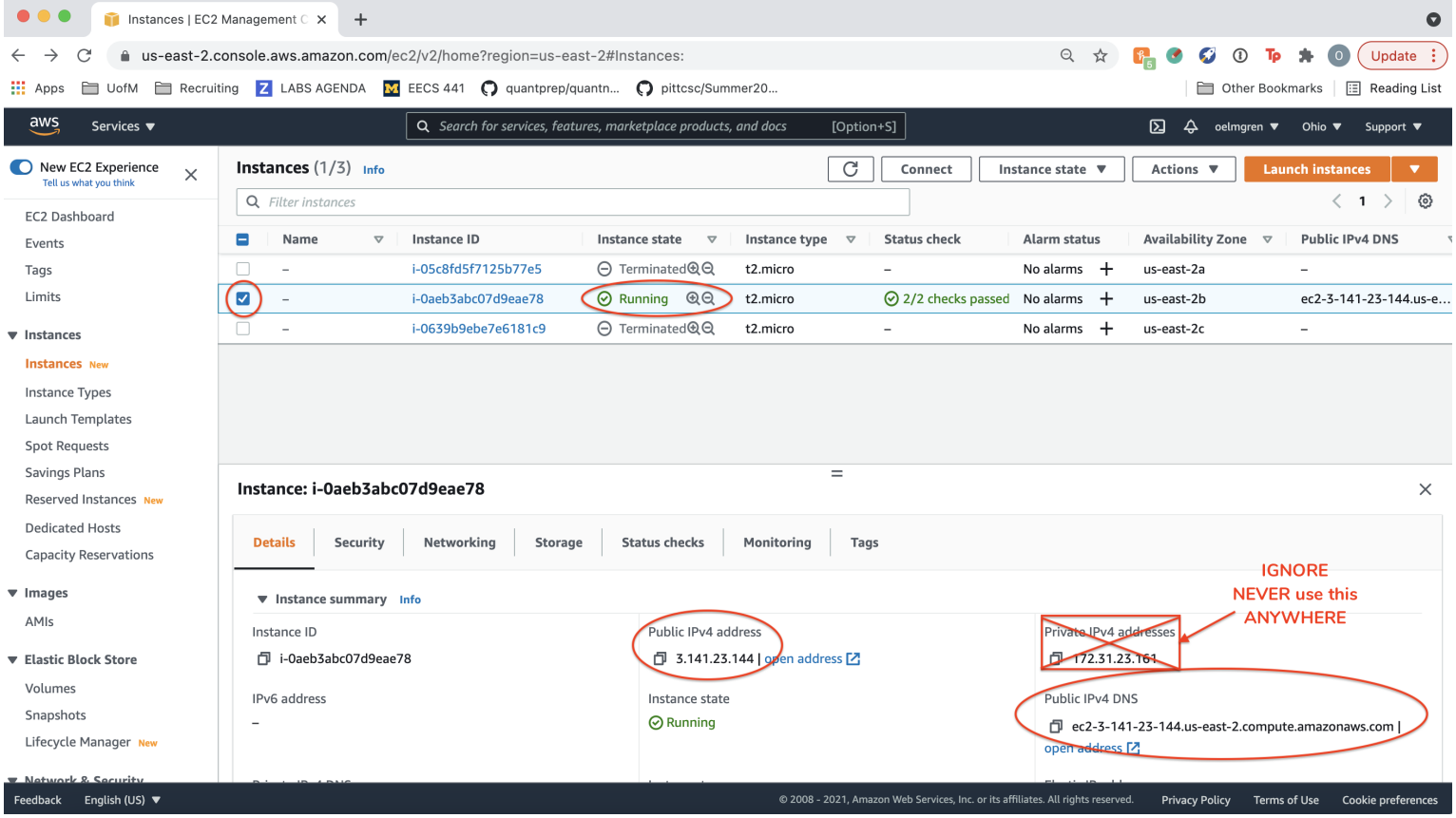
Navigate to the AWS Management Console. Select the “Services” dropdown menu, then “EC2”.



Click “Instances”.



Select one of the instances and view its status and Public DNS.



In the remainder of this spec, we will refer to your "IPv4 Public IP" as `YOUR_SERVER_IP` (in the image it's `3.141.23.144`) and "Public DNS (IPv4)" as `YOUR_SERVER_DNS` (in the image it's `ec2-3-141-23-144.us-east-2.compute.amazonaws.com`). Note that YOUR_SERVER_IP always follows `ec2` in YOUR_SERVER_DNS, with your "region" listed next (in this case `us-east-2`).

`ssh` to instance

On your development host (`laptop`):

> If AWS gave you a `eec441.cer` instead of `eecs441.pem`, just use `eecs441.cer` everywhere you see `eecs441.pem` in this spec.

MacOS on Terminal:

```
laptop$ cd YOUR_LABSFOLDER
laptop$ mv ~/Downloads/eecs441.pem eecs441.pem
laptop$ chmod 400 eecs441.pem
laptop$ ssh -i eecs441.pem ubuntu@YOUR_SERVER_IP
```

Windows on PowerShell [thanks to Jad Beydoun (F21) for use of `icacls`]:

```
PS laptop> cd YOUR_LABSFOLDER
PS laptop> mv ~\Downloads\eecs441.pem eecs441.pem
PS laptop> icacls eecs441.pem /grant "$($env:username):(r)" /inheritance:r
PS laptop> ssh -i eecs441.pem ubuntu@YOUR_SERVER_IP
```

In both cases, what the above does:

1. change working directory to `YOUR_LABSFOLDER`,
2. move the private `ssh` key you created and downloaded earlier to `YOUR_LABSFOLDER`,
3. set its permissions to read-only, and
4. `ssh` to your AWS instance as user "ubuntu" using the downloaded private key.

⚠ You **must** use `YOUR_SERVER_IP`, **not** `YOUR_SERVER_DNS`, to `ssh` to your AWS instance. The DNS hostname assigned by AWS is too long for `ssh`.

And make sure your instance is running. See Instance status.

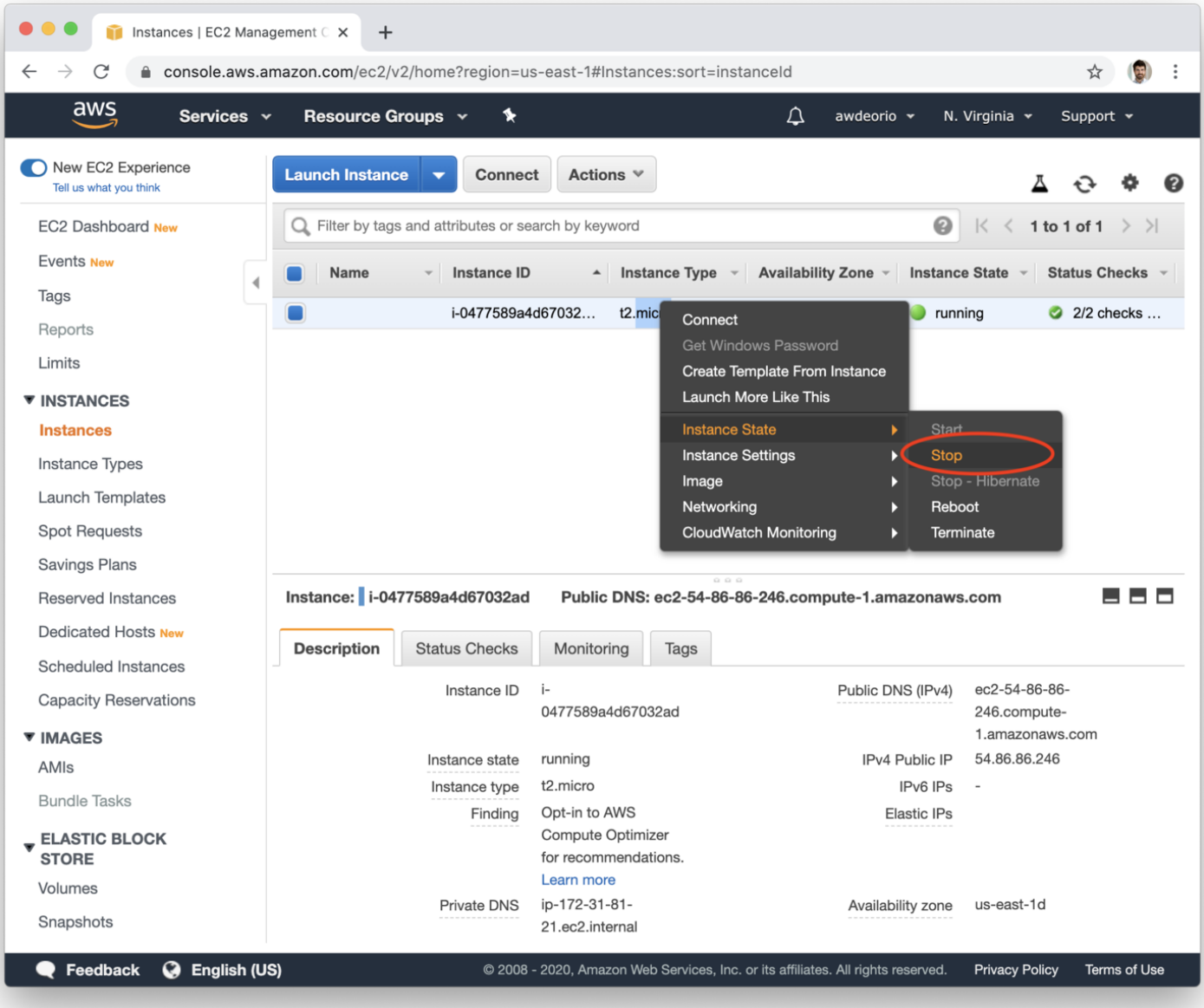
Stop instance

❗ Please leave your EC2 instance running for grading purposes. **DO NOT STOP YOUR INSTANCE**. Stopping your instance will change its alloted IP address and undo some of the customizations you've done following this spec. When we're done with all the labs, after lab4 has been graded, **in about 2.5 months**, and if you don't need your instance for your course project, then you can stop your instance, to avoid using your AWS credits.

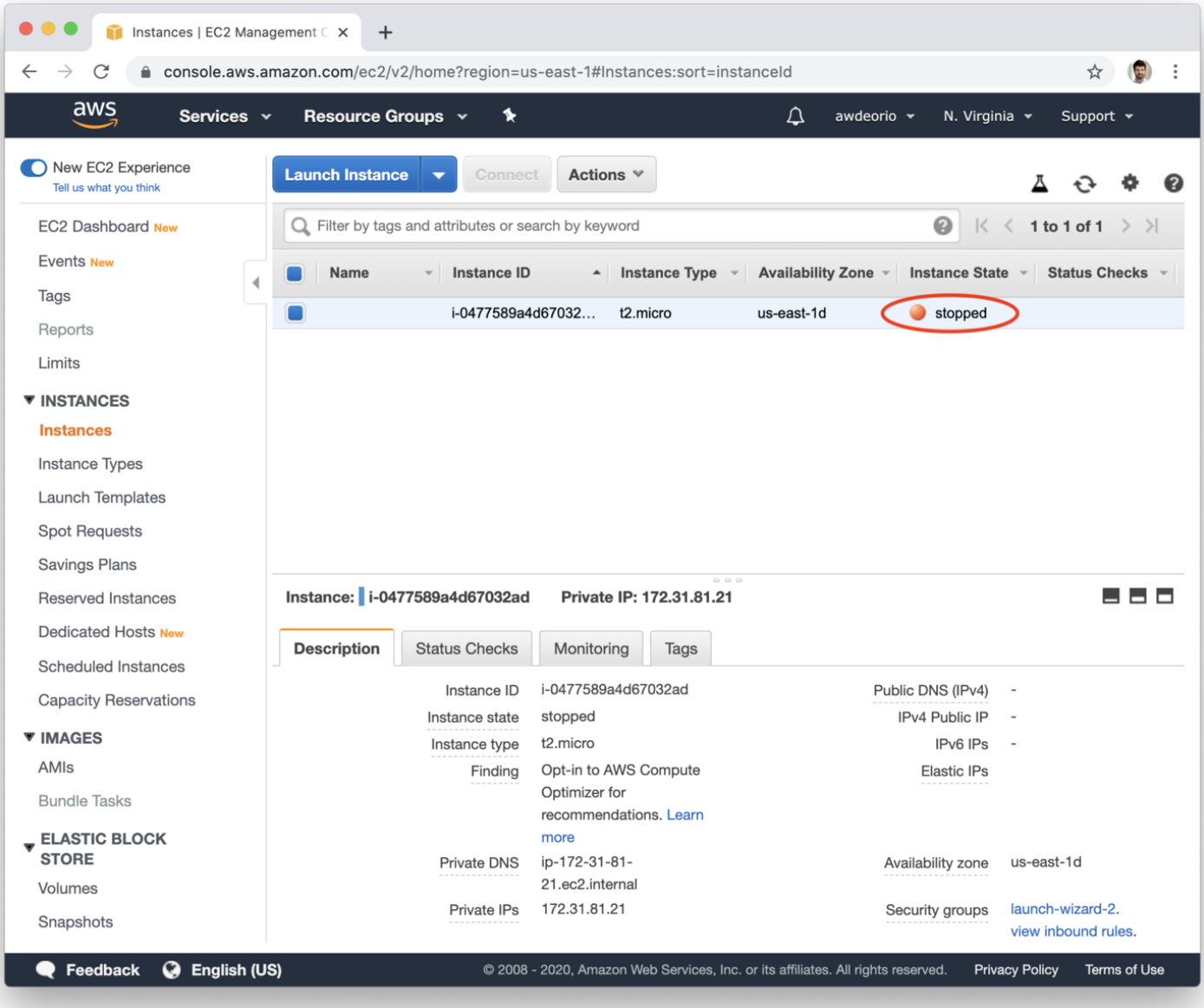
The AWS free credit refreshes every month. So don't fret if you get an email from AWS near the end of a month saying you've used up 85% of your free credit. It should reset when the new month rolls around.

Check your Instance status.

Right click on your instance `> Instance State >Stop`.



You should now see that your instance is stopped.



Install updates

Every time you ssh to your server, you will see something like:

```
N updates can be installed immediately.
```

if `N` is not 0, run the following:

```
server$ sudo apt update
server$ sudo apt upgrade
```

Failure to update your packages could lead to the lab back end not performing correctly and also make you vulnerable to security hacks.

If you see `*** System restart required ***` when you ssh to your server, please run:

```
server$ sync
server$ sudo reboot
```

Your ssh session will be ended at the server. Wait a few minutes for the system to reboot before you ssh to your server again.

Web server

The following are based on DigitalOcean's How To Set Up Django with Postgres, Nginx, and Gunicorn on Ubuntu 20.04 though the instructions have been customized to the specifics of our `Chatter` project, especially when it comes to directory and variable naming, for the sake of narrative consistency across all our labs.

Installing packages

We next need to install and setup the following packages using the `apt` package manager:

- Python 3.6 or later
- PostgreSQL (latest version)
- Nginx (latest version)

We will later use the Python package manager `pip` to install:

- Django 3.0.5 or later
- Gunicorn (latest version)

Before we start, we first ensure that you have the latest `apt` package manager's index. Then we download and install the packages and set python3 to be your default python:

```
server$ sudo apt update
server$ sudo apt install python3-pip python3-dev python3-venv libpq-dev postgresql postgresql-contri
server$ sudo ln -s /usr/bin/python3 /usr/bin/python
```

▶ Troubleshooting python

PostgreSQL

- Log into an interactive Postgres (`psql`) session as user `postgres` :

```
server$ sudo -u postgres psql
```

> You may receive the message: “could not change directory to “/root”: Permission denied”. You can safely ignore this message.

Your command-line prompt should change to `postgres=#`.

- Create a database for your project:

```
CREATE DATABASE chatterdb;
```

- Connect to the database you just created:

```
\connect chatterdb
```

> `\connect` may be shortened to `\c`.

Your command-line prompt should change to `chatterdb=#`.

- Create a database user for your project. Make sure to select a secure password.

```
CREATE USER chatter WITH PASSWORD 'chattchatt';
```

> **TIP**: Forgetting to do this is a common cause of getting HTTP error code `500 Internal Server Error`.

All SQL commands must end with a `;`.

- Use SQL commands from here to do what you need, e.g., creating tables, deleting all entries from a table (last command), etc.:

```
CREATE TABLE chatts (username varchar(255), message varchar(255), time timestamp DEFAULT CURRENT
INSERT INTO chatts  VALUES ('testuser1', 'Hello world');
SELECT * from chatts;

TRUNCATE TABLE chatts;
```

> **TIP**: Trying to send `message` longer than 255 characters is another common cause of getting HTTP error code `500 Internal Server Error`.

- You can issue the `\dt` command to list all tables:

```
\dt

         List of relations
 Schema |  Name  | Type  |  Owner
--------+--------+-------+----------
 public | chatts | table | postgres
(1 rows)
```

- Next give user `chatter` access to administer the new database. Your Django project will be known as `chatter` to the postgres database:

```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO chatter;
```

- When you are finished, exit PostgreSQL:

```
\q
```

> or hit `Ctl-d` (`^d`).

Clone your `441` GitHub repo to your back end server

We need to clone your `441` GitHub repo so that we can push your back-end files for submission:

- First, on your browser, navigate to your `441` GitHub repo
- Click on the green `Code` button and copy the URL to your clipboard by clicking the clipboard icon next to the URL
- Then on your back-end server:

```
server$ cd ~
server$ git clone <paste the URL you copied above> 441
```

> If you haven't, you would need to create a personal access token to use HTTPS Git.

If all goes well, your `441` repo should be cloned to `~/441` . Check that:

```
server$ ls ~/441
```

shows the content of your `441` git repo, including your lab0 front end.

Python virtual environment

We next install Python within a virtual environment for easier management.

- First confirm that your installed python is of version 3.6 or higher:

```
server$ python --version
```

> If your shell doesn't recognize the command or the output doesn't say `Python 3.6` or higher, you'd need to switch your python to version 3.6 or higher. If you don't know how to switch to a different version of python, try this tutorial.

- Create and change into a directory where we can keep our project files:

```
server$ mkdir ~/441/chatter
server$ cd ~/441/chatter
```

- Within the project directory, create a Python virtual environment:

```
server$ python -m venv env
```

This will create a directory called `env` within your `chatter` directory. Inside, it will install a local version of Python and a local version of `pip`. We can use this to install and configure an isolated Python environment for our project.

> **Note**: for this and subsequent labs, we will assume your folders/directories are named using the "canonical" names listed here. For example, we will always refer to the project directory as `~/441/chatter` and the directory where the python virtual environment is housed as `~/441/chatter/env`, without further explanations, from this lab to lab4. If you prefer to use your own naming scheme, you're welcome to do so, but be aware that you'd have to map your naming scheme to the canonical one in all the labs.

- Activate your virtual environment:

```
server$ source env/bin/activate
```

Your prompt should change to indicate that you are now operating within a Python virtual environment. It will look something like this: `(env) ubuntu@YOUR_SERVER_IP:~/441/chatter$`.

Henceforth we will shorten the prompt to just `(env):chatter$` to indicate being inside the python virtual environment.

- Within your activated virtual environment, install `django`, `gunicorn`, and the `psycopg2` PostgreSQL adaptor using the local instance of `pip`:

```
(env):chatter$ pip install django gunicorn psycopg2-binary
```

You should now have all of the software packages needed to start a Django project.

> You can check whether Django is installed and which version is installed with:
>
> ```
> server$ python -m django --version
> ```

Django web framework

- Create a Django project:

```
(env):chatter$ django-admin startproject routing ~/441/chatter
```

At this point your project directory (`~/441/chatter`) should have the following content:

- `~/441/chatter/env/` : the virtual environment directory we created earlier.
- `~/441/chatter/manage.py` : the Django project management script.
- `~/441/chatter/routing/` : the Django project package. This should contain the `asgi.py` , `__init__.py` , `settings.py` , `urls.py` , and `wsgi.py` files.

- Edit the project settings:

```
(env):chatter$ vi routing/settings.py
```

> To edit a file, use your favorite editor, such as `nano` or `vi` (or `vim` or `nvim`). In this and all subsequent labs, we will assume `vi` because it has the shortest name 😊. You can replace `vi` with your favorite editor every time you see it. `Nano` has on-screen help and may be easier to pick up.

At the top of the file add:

```
import os
```

Next locate the `ALLOWED_HOSTS` directive. This defines a list of addresses or domain names clients may use to connect to the Django server instance. Any incoming requests with a **Host** header that is not in this list will raise an exception. Django requires that you set this to prevent a certain class of security vulnerability.

In the square brackets, list the IP addresses or domain names that are associated with your Django server. Each item should be listed in single quotes, with entries separated by a comma. In the snippet below, a few commented out examples are provide as examples. Those with a period prepended to the beginning of an entry serves an entire domain and its subdomains.

> **Note**: Be sure to include '**localhost**' as one of the options, for testing. You can also add '**127.0.0.1**', the IP address indicating localhost. Some online examples may use it instead of 'localhost'.

```
. . .
# The simplest case: just add the domain name(s) and IP addresses of your Django server
# ALLOWED_HOSTS = [ 'example.com', '203.0.113.5']
# To respond to 'example.com' and any subdomains, start the domain with a dot
# ALLOWED_HOSTS = ['.example.com', '203.0.113.5']
# ALLOWED_HOSTS = ['your_server_domain_or_IP', 'second_domain_or_IP', . . ., 'localhost']
ALLOWED_HOSTS = ['YOUR_SERVER_IP', 'localhost', '127.0.0.1']
```

> AWS: You don't need to list YOUR_SERVER_DNS nor your instance's Private IP and DNS.

Next find the `DATABASES` configuration. The default configuration in the file is for a SQLite database. We want to use a PostgreSQL database for our project, so we need to change the settings to our PostgreSQL database information. We tell Django to use the `psycopg2` adaptor we installed with `pip` . We need to give the database name, the database username, the database user's password, and then

specify that the database is located on the local computer. You can leave the `PORT` setting as an empty string:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'chatterdb',
        'USER': 'chatter',
        'PASSWORD': 'chattchatt',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Scroll to the bottom of the file and add a setting indicating where the static files should be placed. The Nginx web server is optimized to serve static files really fast, calling your python code to serve dynamic content only when necessary. Here we're telling Django to put static files for Nginx in a directory called `static` in the base project directory (`~/441/chatter/static/`):

```
STATIC_URL = '/static/'
STATIC_ROOT = BASE_DIR / 'static'   # added line
```

Save and close the `settings.py` file when you're done.

- We can now migrate Django's administrative database schema to our PostgreSQL database using the management script (and the exptected output):

  ```
  (env):chatter$ ./manage.py makemigrations
  # output:
  No changes detected
  (env):chatter$ ./manage.py migrate
  # output:
  Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions
  Running migrations:
  No migrations to apply.
  ```

- Create an administrative user for the project:

  ```
  (env):chatter$ ./manage.py createsuperuser
  ```

  You can use your uniqname@umich.edu, and choose and confirm a password.

- Collect all static content into the static directory we configured:

  ```
  (env):chatter$ ./manage.py collectstatic
  # output:
  128 static files copied to '/home/ubuntu/chatter/static'.
  ```

Testing Nginx

- Following the initial server setup guide, you should have a `ufw` firewall protecting your server. In order to test our back end, we now allow access to port 8000 which we'll be using for testing:

```
(env):chatter$ sudo ufw allow 8000
```

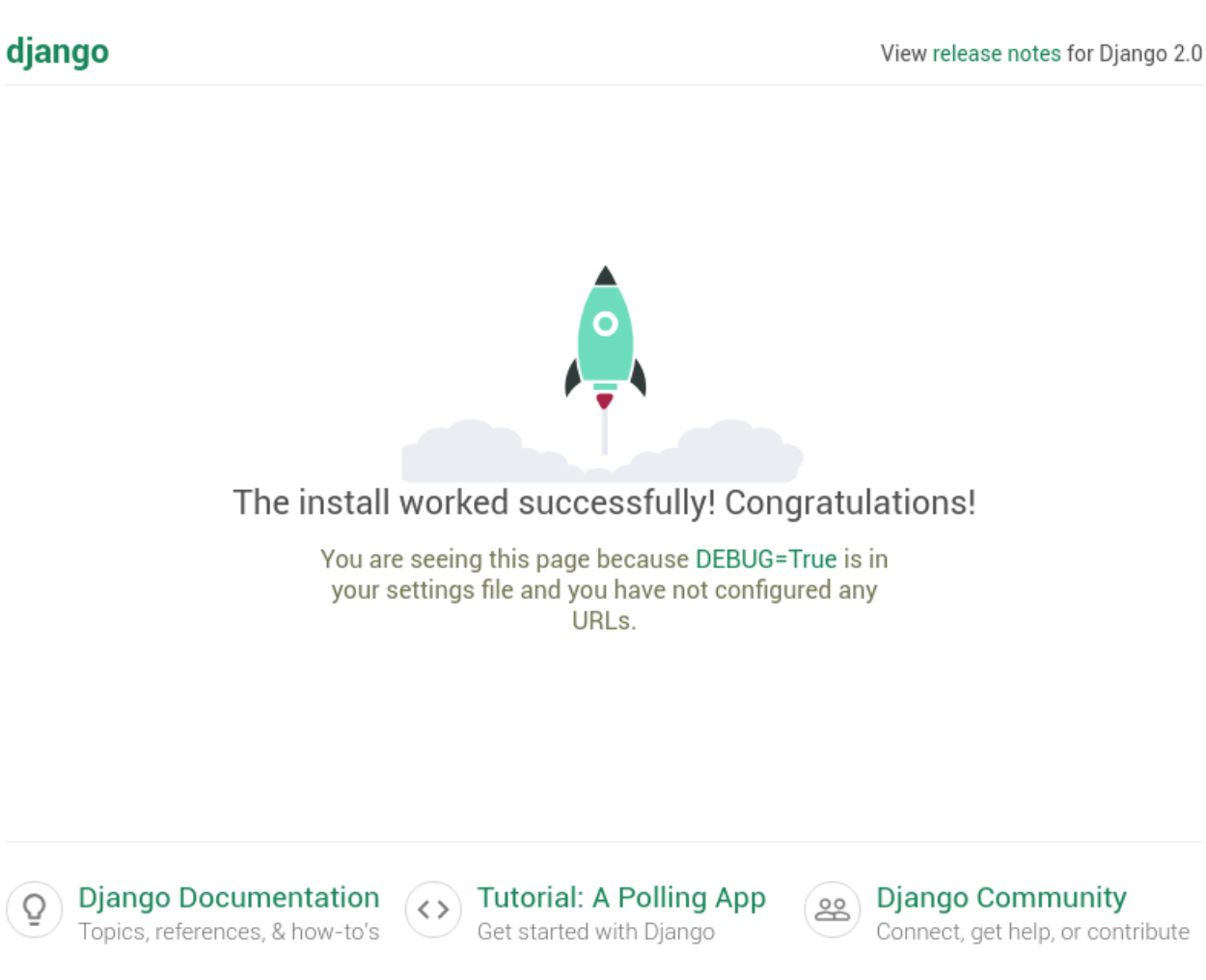
- You can now test run and test your web server:

```
(env):chatter$ ./manage.py runserver YOUR_SERVER_DNS:8000
```

> ❗ Use YOUR_SERVER_DNS **not** YOUR_SERVER_IP with `runserver`, but you can use either with the browser below:

In your web browser on your `laptop`, visit `http://YOUR_SERVER_IP:8000/`.

You should see the default Django index page:



When you are satisfied, hit `Ctl-C` in `server`'s terminal window to shut down the development server.

TIP: should you need to debug any of your Django python code later in the term, remember that running `runserver` on port 8000 from the command line and connecting to it from your browser, or app, allows

you to view error messages from your python code, including any debug printouts from your code. Remember to set your firewall to allow access to port 8000: `server$ sudo ufw allow 8000` .

Testing Gunicorn

- The last thing we want to do before leaving our virtual environment is test Gunicorn to make sure that it can serve the application. We can do this by entering our project directory and using `gunicorn` to load the project's WSGI (Web Server Gateway Interface) module:

  ```
  (env):chatter$ gunicorn --bind 0.0.0.0:8000 routing.wsgi
  ```

  This will start Gunicorn on the same interface that the Django development server was running on. You should see output similar to this:

  ```
  [2020-07-21 21:43:14 -0400] [32224] [INFO] Starting gunicorn 20.0.4
  [2020-07-21 21:43:14 -0400] [32224] [INFO] Listening at: http://0.0.0.0:8000 (32224)
  [2020-07-21 21:43:14 -0400] [32224] [INFO] Using worker: sync
  [2020-07-21 21:43:14 -0400] [32227] [INFO] Booting worker with pid: 32227
  ```

- When you are done testing, hit `Ctl-C` in the `server` 's terminal window to stop Gunicorn.
- We're done configuring the Django application. Exit the virtual environment:

  ```
  (env):chatter$ deactivate
  ```

System-level setup

Configuring Gunicorn

We have tested that Gunicorn can interact with our Django application, we now implement a more robust way of starting and stopping the application server. For this, we'll rely on `systemd` service and socket files.

The Gunicorn socket will be created at boot and will listen for connections. When a connection occurs, `systemd` will automatically start the Gunicorn process to handle the connection.

- Start by creating and opening a `systemd` socket file for Gunicorn with sudo privileges:

  ```
  server$ sudo vi /etc/systemd/system/gunicorn.socket
  ```

- Inside, we will create a `[Unit]` section to describe the socket, a `[Socket]` section to specify the socket location, and an `[Install]` section to make sure the socket is created at the right time:

```
[Unit]
Description=gunicorn socket

[Socket]
```

```
ListenStream=/run/gunicorn.sock

[Install]
WantedBy=sockets.target
```

Save and close the file when you are finished.

- Next, create and open a `systemd` service file for Gunicorn, with `sudo` privileges. The service filename must match the socket filename, except for the extension:

```
server$ sudo vi /etc/systemd/system/gunicorn.service
```

- Enter the following into your `gunicorn.service` file:

```
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target

[Service]
User=ubuntu
Group=www-data
WorkingDirectory=/home/ubuntu/441/chatter
ExecStart=/home/ubuntu/441/chatter/env/bin/gunicorn \
          --access-logfile - \
          --workers 3 \
          --bind unix:/run/gunicorn.sock \
          routing.wsgi:application

[Install]
WantedBy=multi-user.target
```

With that, our `systemd` service file is complete. Save and close it.

- We can now start and enable the Gunicorn socket. This will create the socket file at `/run/gunicorn.sock` now and at boot. When a connection is made to that socket, `systemd` will automatically start the `gunicorn.service` to handle it:

```
server$ sudo systemctl start gunicorn.socket
server$ sudo systemctl enable gunicorn.service
```

We can confirm that the operation was successful by checking for the socket file.

```
server$ file /run/gunicorn.sock
# output:
/run/gunicorn.sock: socket
```

Testing socket

- To test the socket activation mechanism, send a connection to the socket through `curl`:

```
server$ curl --unix-socket /run/gunicorn.sock localhost
```

- You should see the HTML output from your application in the terminal. This indicates that Gunicorn was started and was able to serve your Django application. You can also verify that the Gunicorn service is running with:

```
server$ systemctl status gunicorn
```

which should output something like:

```
● gunicorn.service
     Loaded: loaded (/etc/systemd/system/gunicorn.service; disabled; vendor preset: enabled)
     **Active: active (running)** since Wed 2020-05-27 16:47:07 UTC; 18s ago
   Main PID: 22195 (gunicorn)
     Tasks: 4 (limit: 1152)
     CGroup: /system.slice/gunicorn.service
             ├─22195 /home/ubuntu/chatter/env/bin/python /home/ubuntu/chatter/env/bin/gunicorn --
             ├─22211 /home/ubuntu/chatter/env/bin/python /home/ubuntu/chatter/env/bin/gunicorn --
             ├─22213 /home/ubuntu/chatter/env/bin/python /home/ubuntu/chatter/env/bin/gunicorn --
             └─22215 /home/ubuntu/chatter/env/bin/python /home/ubuntu/chatter/env/bin/gunicorn --

May 27 16:47:07 ubuntu-s-1vcpu-1gb-nyc1-01 systemd[1]: Started gunicorn.service.
May 27 16:47:08 ubuntu-s-1vcpu-1gb-nyc1-01 gunicorn[22195]: [2020-05-27 16:47:08 +0000] [22195]
May 27 16:47:08 ubuntu-s-1vcpu-1gb-nyc1-01 gunicorn[22195]: [2020-05-27 16:47:08 +0000] [22195]
May 27 16:47:08 ubuntu-s-1vcpu-1gb-nyc1-01 gunicorn[22195]: [2020-05-27 16:47:08 +0000] [22195]
May 27 16:47:08 ubuntu-s-1vcpu-1gb-nyc1-01 gunicorn[22195]: [2020-05-27 16:47:08 +0000] [22211]
May 27 16:47:08 ubuntu-s-1vcpu-1gb-nyc1-01 gunicorn[22195]: [2020-05-27 16:47:08 +0000] [22213]
May 27 16:47:08 ubuntu-s-1vcpu-1gb-nyc1-01 gunicorn[22195]: [2020-05-27 16:47:08 +0000] [22215]
lines 1-18
```

Make sure Gunicorn's status reported on the third line is `Active: active (running)`.

▶ Troubleshooting

Nginx2Gunicorn

Now that Gunicorn is set up, we need to configure Nginx to pass traffic to it.

- Start by creating and opening a web site configuration file we'll call `chatter` in Nginx's `sites-available` directory:

```
server$ sudo vi /etc/nginx/sites-available/chatter
```

- Inside, open up a new server block. We will start by specifying that the server will listen on the normal port 80 and that it should respond to YOUR_SERVER_IP (replace YOUR_SERVER_IP with yours):

```
server {
  listen 80;
  server_name YOUR_SERVER_IP;

  location = /favicon.ico { access_log off; log_not_found off; }
  location /static/ {
    root /home/ubuntu/441/chatter;
  }

  location / {
    include proxy_params;
    proxy_pass http://unix:/run/gunicorn.sock;
  }
}
```

- Save and close the file. Now, we can enable the file by linking it to the `sites-enabled` directory:

```
server$ sudo ln -s /etc/nginx/sites-available/chatter /etc/nginx/sites-enabled
```

- Test your Nginx configuration for syntax errors:

```
server$ sudo nginx -t
# output:
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

- If no errors are reported, go ahead and restart Nginx:

```
server$ sudo systemctl restart nginx
```

- Finally, we need to open up our firewall to normal traffic on port 80:

```
server$ sudo ufw allow 'Nginx Full'
```

- You should now be able to, on your browser, browse `http://YOUR_SERVER_IP/` to view the Django default page as before, except now served through gunicorn.

▶ Troubleshooting

Summary: configuration files

In summary, here are the configuration files for the three packages we rely on to provide our back-end service:

nginx: web server that listens on port 80
file: `/etc/nginx/sites-enabled/chatter`, if modified run:

```
server$ sudo nginx -t
server$ sudo systemctl restart nginx
```

gunicorn: serves Django project
file: `/etc/systemd/system/gunicorn.service`, if modified run:

```
server$ sudo systemctl daemon-reload
server$ sudo systemctl restart gunicorn
```

django: framework to route HTTP requests to your python code
directory: `~/441/chatter/routing/`, in particular `urls.py` (see below). If modified run:

```
server$ sudo systemctl restart gunicorn
```

Server-side HTTPS

Begining in 2017, Apple started requiring that apps use HTTPS, the secure version of HTTP. In Aug. 2018, with the release of Android Pie (API level 28), Android also defaulted to blocking all cleartext (HTTP) traffic. To support HTTPS, we need to enable it on the hosting server.

Obtaining a public key

To that end, we must first obtain a public key signed by a Certification Authority (CA). Since obtaining such a certificate usually requires a host with a fully qualified domain name (FQDN), such as `www.eecs.umich.edu`, which your server does not have, we have decided to be our own CA and generate and use a self-signed certificate in this course. Typically, a self-signed certificate is used only during development.

Starting with iOS 13 (and macOS 10.15 Catalina), Apple added some security requirements that all server certificates must comply with. To support both iOS and Android clients, a back-end server must thus comply with these security requirements also.

- To generate a self-signed certificate that satisfies the new requirements, you first need to add them to the openssl configuration file on the server:

  ```
  server$ cd /etc/ssl
  server$ sudo cp openssl.cnf selfsigned.cnf
  ```

- Open `selfsigned.cnf` (with `sudo`), search for the label `v3_ca` and in the `[v3_ca]` section add the following two lines:

```
extendedKeyUsage = serverAuth
subjectAltName = IP:YOUR_SERVER_IP    # or DNS:YOUR_SERVER_FQDN
```

> Due to a bug in `openssl`, it is **CRUCIAL** that the above two lines be added in the `[v3_ca]` section and not outside of it or in any other section of the file.

▶ DNS instead of IP

- Next, search for `copy_extensions` in `selfsigned.cnf` and uncomment it:

```
# Extension copying option: use with caution.
copy_extensions = copy
```

- Now create a self-signed key and certificate pair with OpenSSL using the following command:

```
server$ sudo openssl req -x509 -days 100 -nodes -newkey rsa:2048 -config selfsigned.cnf -keyout
```

You will be asked to fill out a series of prompts, which will look something like:

```
Country Name (2 letter code) [AU]:US
State or Province Name (full name) [Some-State]:MI
Locality Name (eg, city) []:AA
Organization Name (eg, company) [Internet Widgits Pty Ltd]:UM
Organizational Unit Name (eg, section) []:CSE
Common Name (e.g. server FQDN or YOUR name) []:YOUR_SERVER_IP
Email Address []:admin@your_domain.com
```

The **most important information** is `Common Name (e.g. server FQDN or YOUR name)`. Make sure to use the **YOUR_SERVER_IP** address.

> Or your FQDN if you have specified an FQDN as `subjectAltName` above. But if you're on AWS, you **must** use YOUR_SERVER_IP and **not** YOUR_SERVER_DNS because the DNS provided by AWS is too long for Nginx.

- Verify that the generated certificate has the right entries:

```
server$ sudo openssl x509 -text -in certs/selfsigned.cert -noout
```

It must have the following lines:

```
    X509v3 extensions:

        X509v3 Extended Key Usage:
            TLS Web Server Authentication
```

```
X509v3 Subject Alternative Name:
    IP Address:YOUR_SERVER_IP         # or DNS:YOUR_SERVER_FQDN
```

Congrats! You have generated a public-key and put it inside a self-signed certificate! Now to use it.

HTTPS web server

Adapted from Self-Signed SSL Cert.

Our next step is to modify our Nginx configuration to use the new key and certificate files.

- Go to the `/etc/nginx/sites-available` directory and edit the `chatter` server configuration file you created earlier:

```
server$ cd /etc/nginx/sites-available
server$ sudo vi chatter
```

- In your configuration file, update the listen statement to use port `443` and `ssl`, and provide pointers to our self-signed certificate and key, as follows. First update the listen statement:

```
server {
  listen 443 ssl;
  listen [::]:443 ssl;  # add support for IPv6
  # . . .
}
```

- Now, add the following pointers directly under the updated listen statements.

```
  ssl_certificate     /etc/ssl/certs/selfsigned.cert;
  ssl_certificate_key /etc/ssl/private/selfsigned.key;
  ssl_protocols TLSv1.2;
  ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM-SHA384:DHE-
```

Your updated `chatter` server configuration file should now look something like this:

```
server {
  listen 443 ssl;
  listen [::]:443 ssl;  # add support for IPv6

  ssl_certificate     /etc/ssl/certs/selfsigned.cert;
  ssl_certificate_key /etc/ssl/private/selfsigned.key;
  ssl_protocols TLSv1.2;
  ssl_ciphers ECDHE-RSA-AES256-GCM-SHA512:DHE-RSA-AES256-GCM-SHA512:ECDHE-RSA-AES256-GCM-SHA384:DHE-

  server_name YOUR_SERVER_IP;

  location = /favicon.ico { access_log off; log_not_found off; }
```

```
  # . . .
}
```

- So that visitors to your web site who accidentally entered `http` instead of `https` wouldn't be confronted with an error message, we will automatically redirect them to `https`. In your server configuration file, create a second server block after the closing bracket of the first server block (in the code, replace `YOUR_SERVER_IP` with yours):

```
# . . .
server {
  listen 80;
  listen [::]:80;  # IPv6

  server_name YOUR_SERVER_IP;

  return 302 https://$server_name$request_uri; # temporary redirect
}
```

Now save and close the file.

- To enable the new server setup, first check to make sure we don't have any syntax error in the configuration file:

```
server$ sudo nginx -t
# output:
nginx: the configuration file /etc/nginx/nginx.conf syntax is ok
nginx: configuration file /etc/nginx/nginx.conf test is successful
```

- Now restart Nginx so that it can reinitialize with the new server configuration:

```
server$ sudo systemctl restart nginx
```

> Be sure to do this **every time** you make changes to the `chatter` server configuration file.

- To test, you can use `curl`:

```
laptop$ curl --insecure https://YOUR_SERVER_IP/
```

or, from your **Chrome** browser on your laptop, browse to `https://YOUR_SERVER_IP/`. It will warn you that "Your connection is not private". Click the `Advanced` button. Then bravely click `Proceed to YOUR_SERVER_IP (unsafe)`. You should see the Nginx rocket.

> On macOS, Safari won't allow you to visit the site, you have to use Chrome.

HTTPS redirect

- You can verify that your redirect for HTTP to HTTPS functions correctly by accessing `http://YOUR_SERVER_IP/` (note use of `http` not `https`). Again, you must do this from a Chrome browser or `curl`.
- If your redirect worked correctly, to allow only encrypted traffic, modify your server configuration to make the redirect permanent. In your configuration file find `return 302` and change it to `return 301`:

  ```
  return 301 https://$server_name$request_uri; # permanent redirect
  ```

  Save and close the file. Once you have verified that you have no syntax errors, restart nginx.

Congratulations! Your host is all set up for the rest of the term! Now to implement the `Chatter` back end.

`Chatter` back end

We will be creating a python app to serve as `Chatter` back end.

Start by creating the model-view-controller (MVC) framework expected by Django for all python projects:

```
server$ cd ~/441/chatter
server$ source env/bin/activate
(env):chatter$ ./manage.py startapp app
(env):chatter$ deactivate
```

This will create a directory `~/441/chatter/app` with the necessary python files in it. In your `~/441/chatter` project directory, you should now have two directories that were created by Django:

- `~/441/chatter/routing/` created with `startproject` earlier. It contains the Django web framework for your app. We will be modifying `settings.py` and `urls.py` in this directory.
- `~/441/chatter/app/` that we just created with `startapp`. It contains your app's domain/business logic and views and controllers. We will be modifying `views.py` in this directory.

These are distinct directories and both must be retained (don't delete or merge them!).

`Chatter`'s API

`Chatter` is a simple CRUD app. We have previously created a `chatts` table in our PostgreSQL database to hold submitted `chatts`. It consists of three columns: `username`, `message`, and `time`, with `time` being automatically filled in by the database when an entry is added. We associate a `username` with each `message` in a `chatt`. The `getchatts` API is for retrieving posted `chatts` (HTTP GET). To create and post a `chatt`, the front end will use the `postchatt` API (HTTP POST). `Chatter` doesn't provide "replace" (HTTP PUT) and "delete" (HTTP DELETE) functions.

To start with, `chatter` has only two APIs:

- `getchatts`: query the database and use HTTP GET to retrieve all found `chatts`
- `postchatt`: use HTTP POST to post a `chatt` as JSON object

The protocol handshakes:

```
url
<- request
-> response

/getchatts/
<- HTTP GET {}
-> { list of chatts } 200 OK

/postchatt/
<- HTTP POST { username, message }
-> {} 200 OK
```

Data formats

The `getchatts` API will send back all accumulated chatts in the form of a JSON object with the key being "chatts" and the value being an array of string arrays. Each string array consists of three elements "username", "message", and "timestamp". For example:

```
{
    "chatts": [["username0", "message0", "timestamp0"],
               ["username1", "message1", "timestamp1"],
               ...
              ]
}
```

Each element of the string array may have a value of JSON `null` or the empty string (`""`).

To post a `chatt` with the `postchatt` API, the front-end client sends a JSON object consisting of "username" and "message". For example:

```
{
   "username": "ubuntu",
   "message": "Hello world!"
}
```

Both APIs will be implemented in `~/441/chatter/app/views.py` .

Adding `getchatts`

Add to `~/441/chatter/app/views.py` :

```
from django.http import JsonResponse, HttpResponse

def getchatts(request):
    if request.method != 'GET':
        return HttpResponse(status=404)
    response = {}
```

```
    response['chatts'] = ['Replace Me', 'DUMMY RESPONSE'] # **DUMMY response!**
    return JsonResponse(response)
```

Routing `getchatts`

In `~/441/chatter/routing/urls.py` add:

```
from app import views
```

and add to the `urlpatterns` array:

```
    path('getchatts/', views.getchatts, name='getchatts'),
```

Your `urls.py` should now look like this:

```
from django.contrib import admin
from django.urls import path
from app import views

urlpatterns = [
    path('getchatts/', views.getchatts, name='getchatts'),
    path('admin/', admin.site.urls),
]
```

Testing `getchatts`

Everytime you make changes to either `views.py` or `urls.py`, you **MUST** restart Gunicorn:

```
server$ sudo systemctl restart gunicorn
```

Test `getchatts` with:

```
  laptop$ curl --insecure https://YOUR_SERVER_IP/getchatts/
```

or by browsing, on **Chrome**, to `https://YOUR_SERVER_IP/getchatts/`. You should see displayed:

```
{"chatts": [["Replace Me", "DUMMY RESPONSE", "1970-01-01T00:00:00.000"]]}
```

Retrieving `chatts`

Add to `app/views.py`:

```
from django.db import connection
```

We will use the database cursor to retrieve `chatts`. We will add the following to your python `getchatts(request)` function:

```
    cursor = connection.cursor()
    cursor.execute('SELECT * FROM chatts ORDER BY time DESC;')
    rows = cursor.fetchall()
```

The following shows where to add the above code to `getchatts()`. Once you have retrieved all the rows from the database, you need to insert it into the `response` dictionary to be returned to the front end. Don't forget to replace the dummy response above with the `rows` retrieved from the database.

```
def getchatts(request):
    if request.method != 'GET':
        return HttpResponse(status=404)

    cursor = connection.cursor()
    cursor.execute('SELECT * FROM chatts ORDER BY time DESC;')
    rows = cursor.fetchall()

    response = {}
    response['chatts'] = rows       # <<<<< NOTE: REPLACE dummy response WITH chatts <<<
    return JsonResponse(response)
```

Adding `postchatt`

Django wants CSRF (cross-site request forgery) cookies by default, since we're not implementing it, we ask for exemption. In `views.py` add:

```
from django.views.decorators.csrf import csrf_exempt
import json

@csrf_exempt
def postchatt(request):
    if request.method != 'POST':
        return HttpResponse(status=404)
    json_data = json.loads(request.body)
    username = json_data['username']
    message = json_data['message']
    cursor = connection.cursor()
    cursor.execute('INSERT INTO chatts (username, message) VALUES '
                   '(%s, %s);', (username, message))
    return JsonResponse({})
```

> For Python-PostgreSQL interaction, see Passing parameters to SQL queries.

Once we have the view define, we need to tell Django how to route to it. In `routing/urls.py` add the following entry to the `urlpatterns` array:

```
    path('postchatt/', views.postchatt, name='postchatt'),
```

As before, everytime you make changes to either `app/views.py` or `routing/urls.py`, you need to restart Gunicorn:

```
server$ sudo systemctl restart gunicorn
```

Testing `postchatt`

To test `HTTP POST` (or `HTTP PUT` or other) requests, we can again use `curl`:

```
laptop$ curl -X POST -d '{ "username": "Test", "message": "Hello World" }' --insecure https://YOUR_S
```

We can't test using a browser unfortunately. Instead, we could use Postman, which has a nice graphical interface and is free. The instructions for Postman in this and subsequent labs are intended for the desktop version of Postman. If you're proficient with the web version, you can use the web version.

To test with Postman, first disable `Preferences > SSL certificate verification` in Postman. Your certificate wasn't signed by a trusted certification authority, it was self signed.

- Then in the main Postman screen, next to the `Launchpad` tab, click the `+` tab. You get a new `Untitled Request` screen. You should see `GET` listed under `Untitled Request`.
- Click on `GET` to show the drop down menu and select `POST`.
- Enter `https://YOUR_SERVER_IP/postchatt/` in the field next to `POST`.
- Below that there's a menu with `Body` being the fourth element. Click on `Body`.
- You should now see a submenu under `Body`. Click on `raw`.
- At the end of the submenu, click on `TEXT` and select it with `JSON` in the drop down menu.
- You can now enter the following in the box under the submenu:

  ```
  {
      "username": "Postman",
      "message": "Ring! Ring!"
  }
  ```

  and click the big blue `Send` button.

  If everything works as expected, the bottom pane of Postman should say to the right of its menu line, `Status: 200 OK` and the pane should simply display `{}`.

- You can create a new request in Postman to do `GET` with `https://YOUR_SERVER_IP/getchatts` and click the big blue `Send` button. It should return something like:

  ```
  {
      "chatts": [
  ```

```
            [
                "Postman",
                "Ring! Ring!",
                "2020-07-22T17:33:25.947"
            ]
        ]
    }
```

If you're familiar with HTTPie, you can also use it to test from the command line:

```
laptop$ echo '{ "username": "weepie", "message": "Yummy!" }' | http --verify=no POST https://YOU
# output:
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 2
Content-Type: application/json
Date: Wed, 22 Jul 2020 17:45:53 GMT
Server: nginx/1.14.0 (Ubuntu)
X-Content-Type-Options: nosniff
X-Frame-Options: DENY

{}
```

The `--verify=no` option is to tell HTTPie to not try to verify our self-signed certificate. Incidentally, you can also use HTTPie to test `getchatts`:

```
laptop$ http --verify=no https://YOUR_SERVER_IP/getchatts/
# output:
HTTP/1.1 200 OK
Connection: keep-alive
Content-Length: 116
Content-Type: application/json
Date: Wed, 22 Jul 2020 17:46:32 GMT
Server: nginx/1.14.0 (Ubuntu)
X-Content-Type-Options: nosniff
X-Frame-Options: DENY

{
    "chatts": [
        [
            "Postman",
            "Ring! Ring!",
            "2020-07-22T17:33:25.947"
        ],
        [
            "weepie",
            "Yummy!",
            "2020-07-22T17:45:53.177"
        ]
    ]
}
```

Front-end `Chatter`

To test your back-end server with your `Chatter` front end, you need to install your self-signed server certificate on your device (or emulator or simulator). Some versions of both iOS and Android require the certificate to be in binary (DER) format, so we will first convert our certificate to the binary format.

```
server$ cd ~/441/chatter
server$ sudo openssl x509 -inform PEM -outform DER -in /etc/ssl/certs/selfsigned.cert -out selfsigne
server$ sudo chown ubuntu selfsigned.crt
```

Next download a copy of `selfsigned.crt` to your laptop, to `YOUR_LABSFOLDER`:

```
laptop$ cd YOUR_LABSFOLDER
laptop$ scp -i eecs441.pem ubuntu@YOUR_SERVER_IP:441/chatter/selfsigned.crt selfsigned.crt
```

And change the `serverUrl` property of your `ChattStore` class from `mobapp.eecs.umich.edu` to YOUR_SERVER_IP.

If you're on Android, skip to installation on Android emulator and device.

On iOS simulator

Drag `selfsigned.crt` on your laptop and drop it on the home screen of your iOS device simulator.

That's it. To test it, launch a web browser on the simulator and access your `chatt` server at `https://YOUR_SERVER_IP/getchatts/`.

> You can use ADVTrustStore to list and remove installed certificates on your simulated devices.

On iOS device

AirDrop `selfsigned.crt` to your iPhone or email it to yourself.

Then **on your device**:

1. If you emailed the certificate to yourself, view your email and tap the attached `selfsigned.crt`. iOS will tell you, `Profile Downloaded`. If you AirDropped it, just continue to next step.
2. Go to `Settings > General > VPN & Device Management` and tap on the profile with your `YOUR_SERVER_IP`.
3. At the upper right corner of the screen, tap `Install`.
4. Enter your passcode.
5. Tap `Install` at the upper right corner of the screen **again**.
6. And tap the somewhat dimmed out `Install` button.
7. Tap `Done` on the upper right corner of screen.

8. **Go back to `Settings > General`**
9. **Go to `[Settings > General > ]About > Certificate Trust Settings`**
10. **Bravely slide the toggle button next to `YOUR_SERVER_IP` to enable full trust of your self-signed certificate**

> **IT IS A COMMON ERROR TO MISS THE LAST THREE STEPS**

To test the installed self-signed certificate, launch the web browser on your device and access your server at `https://YOUR_SERVER_IP/getchatts/` .

You can retrace your steps to remove the certificate when you don't need it anymore.

> If you run into problem using HTTPS on your device, the error code displayed by Xcode may help you debug. This post has a list of them near the end of the thread.

iOS `Chatter`

Build and run your app and you should now be able to connect your mobile front end to your back end via HTTPS.

On Android emulator and device

1. Get `selfsigned.crt` onto your emulator or device:
   - for the emulator: drag `selfsigned.crt` on your laptop and drop it on the home screen of the emulator (a dialog box saying `Copying file` pops up)
   - for the device: email `selfsigned.crt` to yourself, then on the device, view your email and tap the attached `selfsigned.crt`

Then regardless of whether you're on the emulator or device, continue with the following steps:

1. Swipe up to reveal the `Settings` button.
2. Go to `Settings > Security > Encryption & credentials > Install a certificate > CA certificate > Install anyway >` tap on `selfsigned.crt` .

You can verify that your certificate is installed in `Settings > Security > Encryption & credentials > Trusted credentials > USER` (tab at the top right corner).

To test the installion, launch a web browser on the emulator or device and access your server at `https://YOUR_SERVER_IP/getchatts/` .

Android `Chatter`

Next, we need to tell Android to trust the self-signed certificate we have installed above. First we create an xml file `network_security_config.xml` to tell `chatter` where to find the self-signed certificate:

1. In Android Studio, right click on your `/app/res/` folder and select `New > Android Resource Directory` in the drop-down menu. Change the `Resource type:` to `xml`. The `Directory name:` should automatically change to `xml` also. Leave the `Source set:` to `main` and click `OK`.

   > `/app/res/` may show up as `/app/src/main/res/`, depending on your Android Studio setting.

2. Right click the new `xml` folder, choose `New > File` in the drop-down menu. Name the new xml file `network_security_config.xml` and put the following content in the file:

```
<?xml version="1.0" encoding="utf-8"?>
<network-security-config>
    <domain-config>
        <domain includeSubdomains="true">YOUR_SERVER_IP</domain>
        <trust-anchors>
            <certificates src="user"/>
        </trust-anchors>
    </domain-config>
</network-security-config>
```

> It is important that you limit the use of the self-signed certificate to only your own back-end server IP. In particular do **not** use the `<base-config>` tag because it will cause your app to try and fail to use the self-signed certificate with other services, such as Google Maps.

1. Finally add the following line that accounts for the new file to your `AndroidManifest.xml`:

```
  <application
      android:allowBackup="true"
      // ... other items
      android:networkSecurityConfig="@xml/network_security_config">
```

Build and run your app and you should now be able to connect your mobile front end to your back end via HTTPS.

Submission guideline

- Navigate to your back end folder:

  ```
  server$ cd ~/441/chatter
  server$ cp /etc/nginx/sites-enabled/chatter nginx-site
  ```

  and tell git to add the files we've added or changed:

  ```
  server$ git add selfsigned.crt nginx-site app/views.py routing/urls.py routing/settings.py
  ```

  You can check the git status by running:

  ```
  server$ git status
  ```

  and you will see the newly added files.

- Commit new changes to the local repo with:

  ```
  server$ git commit -m "lab0 back end"
  ```

  and push new changes to the remote GitHub repo with:

  ```
  server$ git push
  ```

- If `git push` fails due to new changes made to the remote repo, you will need to run `git pull` first. Then you may have to resolve any conflicts before you can `git push` again.

Go the GitHub website to confirm that your back end files have been uploaded to your GitHub repo. Also double check and confirm that your repo has a folder structure similar to the following (you will not see `swiftChatter` if you're building `kotlinChatter` and vice versa), otherwise our script will not pick up your submission and you will further have problems getting started with latter labs:

```
441
  |— chatter
      |— app
      |— nginx-site
      |— routing
      |— selfsigned.crt
  |— lab0
      |— kotlinChatter
          |— app
      |— swiftChatter
          |— swiftChatter.xcodeproj
          |— swiftChatter
```

If you haven't, remember to invite `eecs441staff@umich.edu` to your GitHub repo as collaborator and enter your uniqname (and that of your lab team mate's) and the link to your GitHub repo on the Lab Links sheet.

Resources and References

Accounts:

- GitHub student pack

Setup:

- Ubuntu setup
- Nginx+Gunicorn+Django+PostgreSQL setup
  - Django NameError: name 'os' is not defined
- AWS Services You Should Know When Deploying Your Django App

Intro:

- Django introduction
- PostgreSQL tutorial
- Postman or HTTPie
- To read more about how nginx, gunicorn, and django work together:
  - Django Runserver is not your Production Server
  - Gunicorn and Nginx in a Nutshell
  - What is Gunicorn, and What does it do?

HTTPS, SSL, TLS

- What is HTTPS
- Everything about HTTPS and SSL

Working with self-signed certificate

- New self-signed SSL Certificate for iOS 13
  - Requirements for trusted certificates in iOS 13 and macOS 10.15
  - Missing X.509 extensions with an openssl-generated certificate
  - iOS SSL-related error code
- Self-Signed SSL Certificate for Nginx
- Connecting mobile apps (iOS and Android) to backends for development with SSL
  - Android Network Security Configuration
  - Network security config for range of ip addresses?
- How to do the CHMOD 400 Equivalent Command on Windows
  - icacls

Appendix: AWS command line tools

To administer AWS EC2 instance from the Ubuntu command line, install the following:

```
server$ sudo apt install cloud-utils awscli
```

Useful commands:

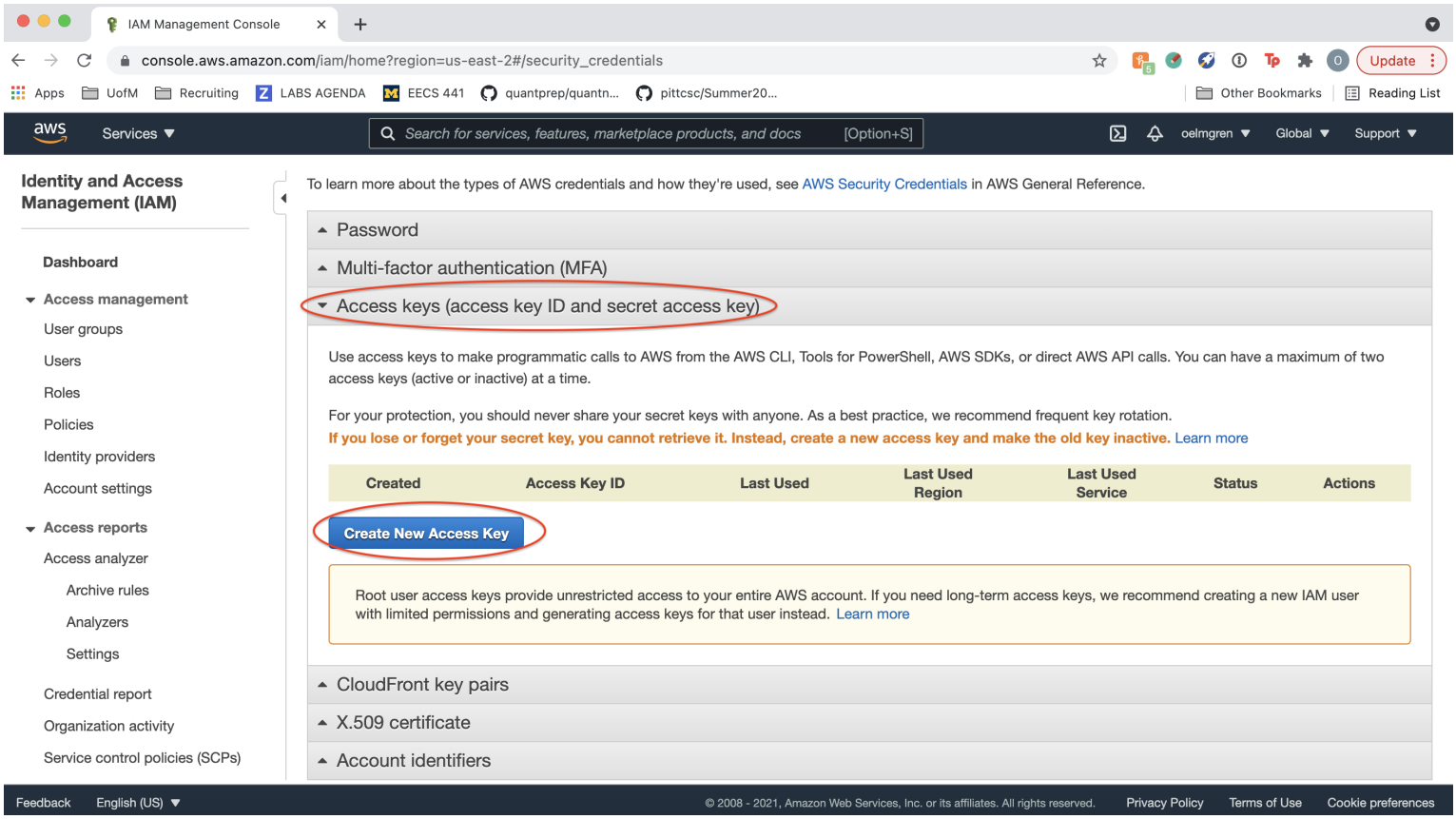
```
server$ ec2metadata
server$ aws configure
server$ aws ec2 help
```

The command `ec2metadata` shows the instance's `public-ip` and `public-hostname`.

The command `aws configure` asks for `AWS Access Key ID`, which can be obtained from:

```
server$ aws iam list-access-keys
```

It also asks for `AWS Secret Access Key`, which is shown only at creation time at the IAM console.



The `Default region name` is listed in the `public-hostname` following the `public-ip`.

The command `aws ec2` is the main interface to configure ec2. The `help` sub-command lists all the sub-commands such as `describe-security-groups`, from which one can obtain the group name/id needed to run sub-command `authorize-security-group-ingress`, for example.

To add IPv6 CIDR block use `--ip-permissions`, e.g.,

```
server$ aws ec2 authorize-security-group-ingress --group-id YOUR_GROUP_ID --ip-permissions IpProtocc
```