Create a new project directory along with a new Django project:

$ mkdir testprojectfolder && cd testprojectfolder

$ mkdir app && cd app

$ python3.9 -m venv env

$ source env/bin/activate

(env)$

(env)$ pip install django==3.2.6

(env)$ django-admin.py startproject testproject .

(env)$ python manage.py migrate

(env)$ python manage.py runserver

Navigate to <http://localhost:8000/> to view the Django welcome screen. Kill the server once done. Then, exit from and remove the virtual environment. We now have a simple Django project to work with.

Create a requirements.txt file in the "app" directory and add Django as a dependency:

Pip freeze > requirements.txt

Pip freeze -r requirements.txt

The contents of requirements.txt are like below

Django==3.2.6

Since we'll be moving to Postgres, go ahead and remove the db.sqlite3 file from the "app" directory.

Your project directory should look like:

└── app

├── testproject

│   ├── \_\_init\_\_.py

│   ├── asgi.py

│   ├── settings.py

│   ├── urls.py

│   └── wsgi.py

├── manage.py

└── requirements.txt

Docker

Install [Docker](https://docs.docker.com/install/), if you don't already have it, then add a *Dockerfile* to the "app" directory:

*# pull official base image*

**FROM** python:3.9.6-alpine

*# set work directory*

**WORKDIR** /usr/src/app

*# set environment variables*

**ENV** PYTHONDONTWRITEBYTECODE 1

**ENV** PYTHONUNBUFFERED 1

*# install dependencies*

**RUN** pip install --upgrade pip

**COPY** ./requirements.txt .

**RUN** pip install -r requirements.txt

*# copy project*

**COPY** . .

So, we started with an [Alpine](https://github.com/gliderlabs/docker-alpine)-based [Docker image](https://hub.docker.com/_/python/) for Python 3.9.6. We then set a [working directory](https://docs.docker.com/engine/reference/builder/#workdir) along with two environment variables:

1. PYTHONDONTWRITEBYTECODE: Prevents Python from writing pyc files to disc (equivalent to python -B [option](https://docs.python.org/3/using/cmdline.html#id1))
2. PYTHONUNBUFFERED: Prevents Python from buffering stdout and stderr (equivalent to python -u [option](https://docs.python.org/3/using/cmdline.html#cmdoption-u))

Finally, we updated Pip, copied over the *requirements.txt* file, installed the dependencies, and copied over the Django project itself.

Review [Docker for Python Developers](https://mherman.org/presentations/dockercon-2018) for more on structuring Dockerfiles as well as some best practices for configuring Docker for Python-based development.

Next, add a *docker-compose.yml* file to the project root:

**version**: '3.8'

**services**:

**web**:

**build**: ./app

**command**: python manage.py runserver 0.0.0.0:8000

**volumes**:

- ./app/:/usr/src/app/

**ports**:

- 8000:8000

**env\_file**:

- ./.env.dev

Review the [Compose file reference](https://docs.docker.com/compose/compose-file/) for info on how this file works.

Update the SECRET\_KEY, DEBUG, and ALLOWED\_HOSTS variables in *settings.py*:

SECRET\_KEY = os.environ.get("SECRET\_KEY")

DEBUG = int(os.environ.get("DEBUG", default=0))

*# 'DJANGO\_ALLOWED\_HOSTS' should be a single string of hosts with a space between each.*

*# For example: 'DJANGO\_ALLOWED\_HOSTS=localhost 127.0.0.1 [::1]'*

ALLOWED\_HOSTS = os.environ.get("DJANGO\_ALLOWED\_HOSTS").split(" ")

Make sure to add the import to the top:

**import** **os**

Then, create a *.env.dev* file in the project root to store environment variables for development:

DEBUG=1

SECRET\_KEY=foo

DJANGO\_ALLOWED\_HOSTS=localhost 127.0.0.1 [::1]

Build the image:

$ docker-compose build

Once the image is built, run the container:

$ docker-compose up -d

Navigate to <http://localhost:8000/> to again view the welcome screen.

Check for errors in the logs if this doesn't work via docker-compose logs -f.

Postgres

To configure Postgres, we'll need to add a new service to the *docker-compose.yml* file, update the Django settings, and install [Psycopg2](http://initd.org/psycopg/).

First, add a new service called db to *docker-compose.yml*:

**version**: '3.8'

**services**:

**web**:

**build**: ./app

**command**: python manage.py runserver 0.0.0.0:8000

**volumes**:

- ./app/:/usr/src/app/

**ports**:

- 8000:8000

**env\_file**:

- ./.env.dev

**depends\_on**:

- db

**db**:

**image**: postgres:13.0-alpine

**volumes**:

- postgres\_data:/var/lib/postgresql/data/

**environment**:

- POSTGRES\_USER=testproject

- POSTGRES\_PASSWORD=testproject

- POSTGRES\_DB=testproject\_dev

**volumes**:

**postgres\_data**:

To persist the data beyond the life of the container we configured a volume. This config will bind postgres\_data to the "/var/lib/postgresql/data/" directory in the container.

We also added an environment key to define a name for the default database and set a username and password.

Review the "Environment Variables" section of the [Postgres Docker Hub page](https://hub.docker.com/_/postgres) for more info.

We'll need some new environment variables for the web service as well, so update *.env.dev* like so:

DEBUG=1

SECRET\_KEY=foo

DJANGO\_ALLOWED\_HOSTS=localhost 127.0.0.1 [::1]

SQL\_ENGINE=django.db.backends.postgresql

SQL\_DATABASE=testporject\_dev

SQL\_USER=testproject

SQL\_PASSWORD=testproject

SQL\_HOST=db

SQL\_PORT=5432

Update the DATABASES dict in *settings.py*:

DATABASES = {

"default": {

"ENGINE": os.environ.get("SQL\_ENGINE", "django.db.backends.sqlite3"),

"NAME": os.environ.get("SQL\_DATABASE", BASE\_DIR / "db.sqlite3"),

"USER": os.environ.get("SQL\_USER", "user"),

"PASSWORD": os.environ.get("SQL\_PASSWORD", "password"),

"HOST": os.environ.get("SQL\_HOST", "localhost"),

"PORT": os.environ.get("SQL\_PORT", "5432"),

}

}

Here, the database is configured based on the environment variables that we just defined. Take note of the default values.

Update the Dockerfile to install the appropriate packages required for Psycopg2:

*# pull official base image*

**FROM** python:3.9.6-alpine

*# set work directory*

**WORKDIR** /usr/src/app

*# set environment variables*

**ENV** PYTHONDONTWRITEBYTECODE 1

**ENV** PYTHONUNBUFFERED 1

*# install psycopg2 dependencies*

**RUN** apk update **\**

&& apk add postgresql-dev gcc python3-dev musl-dev

*# install dependencies*

**RUN** pip install --upgrade pip

**COPY** ./requirements.txt .

**RUN** pip install -r requirements.txt

*# copy project*

**COPY** . .

Add Psycopg2 to *requirements.txt*:

Django==3.2.6

psycopg2-binary==2.9.1

Review [this GitHub Issue](https://github.com/psycopg/psycopg2/issues/684) for more info on installing Psycopg2 in an Alpine-based Docker Image.

Build the new image and spin up the two containers:

$ docker-compose up -d --build

Run the migrations:

$ docker-compose exec web python manage.py migrate --noinput

Get the following error?

django.db.utils.OperationalError: FATAL: database "hello\_django\_dev" does not exist

Run docker-compose down -v to remove the volumes along with the containers. Then, re-build the images, run the containers, and apply the migrations.

Ensure the default Django tables were created:

$ docker-compose exec db psql --username=testproject --dbname=testproject\_dev

psql (13.0)

Type "help" **for** help.

hello\_django\_dev=*# \l*

List of databases

Name | Owner | Encoding | Collate | Ctype | Access privileges

------------------+--------------+----------+------------+------------+-------------------------------

testproject\_dev | testproject | UTF8 | en\_US.utf8 | en\_US.utf8 |

postgres | testproject | UTF8 | en\_US.utf8 | en\_US.utf8 |

template0 | hello\_django | UTF8 | en\_US.utf8 | en\_US.utf8 | =c/hello\_django +

| | | | | hello\_django=CTc/hello\_django

template1 | hello\_django | UTF8 | en\_US.utf8 | en\_US.utf8 | =c/hello\_django +

| | | | | hello\_django=CTc/hello\_django

(4 rows)

hello\_django\_dev=*# \c hello\_django\_dev*

You are now connected to database "hello\_django\_dev" as user "hello\_django".

hello\_django\_dev=*# \dt*

List of relations

Schema | Name | Type | Owner

--------+----------------------------+-------+--------------

public | auth\_group | table | hello\_django

public | auth\_group\_permissions | table | hello\_django

public | auth\_permission | table | hello\_django

public | auth\_user | table | hello\_django

public | auth\_user\_groups | table | hello\_django

public | auth\_user\_user\_permissions | table | hello\_django

public | django\_admin\_log | table | hello\_django

public | django\_content\_type | table | hello\_django

public | django\_migrations | table | hello\_django

public | django\_session | table | hello\_django

(10 rows)

hello\_django\_dev=*# \q*

You can check that the volume was created as well by running:

$ docker volume inspect django-on-docker\_postgres\_data

You should see something similar to:

[

{

"CreatedAt": "2021-08-23T15:49:08Z",

"Driver": "local",

"Labels": {

"com.docker.compose.project": "django-on-docker",

"com.docker.compose.version": "1.29.2",

"com.docker.compose.volume": "postgres\_data"

},

"Mountpoint": "/var/lib/docker/volumes/django-on-docker\_postgres\_data/\_data",

"Name": "django-on-docker\_postgres\_data",

"Options": null,

"Scope": "local"

}

]

Next, add an *entrypoint.sh* file to the "app" directory to verify that Postgres is healthy *before* applying the migrations and running the Django development server:

*#!/bin/sh*

**if** [ "$DATABASE" = "postgres" ]

**then**

echo "Waiting for postgres..."

**while** ! nc -z $SQL\_HOST $SQL\_PORT; **do**

sleep 0.1

**done**

echo "PostgreSQL started"

**fi**

python manage.py flush --no-input

python manage.py migrate

exec "$@"

Update the file permissions locally:

$ chmod +x app/entrypoint.sh

Then, update the Dockerfile to copy over the *entrypoint.sh* file and run it as the Docker [entrypoint](https://docs.docker.com/engine/reference/builder/" \l "entrypoint) command:

*# pull official base image*

**FROM** python:3.9.6-alpine

*# set work directory*

**WORKDIR** /usr/src/app

*# set environment variables*

**ENV** PYTHONDONTWRITEBYTECODE 1

**ENV** PYTHONUNBUFFERED 1

*# install psycopg2 dependencies*

**RUN** apk update **\**

&& apk add postgresql-dev gcc python3-dev musl-dev

*# install dependencies*

**RUN** pip install --upgrade pip

**COPY** ./requirements.txt .

**RUN** pip install -r requirements.txt

*# copy entrypoint.sh*

**COPY** ./entrypoint.sh .

**RUN** sed -i 's/\r$//g' /usr/src/app/entrypoint.sh

**RUN** chmod +x /usr/src/app/entrypoint.sh

*# copy project*

**COPY** . .

*# run entrypoint.sh*

**ENTRYPOINT** ["/usr/src/app/entrypoint.sh"]

Add the DATABASE environment variable to *.env.dev*:

DEBUG=1

SECRET\_KEY=foo

DJANGO\_ALLOWED\_HOSTS=localhost 127.0.0.1 [::1]

SQL\_ENGINE=django.db.backends.postgresql

SQL\_DATABASE=hello\_django\_dev

SQL\_USER=hello\_django

SQL\_PASSWORD=hello\_django

SQL\_HOST=db

SQL\_PORT=5432

DATABASE=postgres

Test it out again:

1. Re-build the images
2. Run the containers
3. Try <http://localhost:8000/>

Notes

First, despite adding Postgres, we can still create an independent Docker image for Django as long as the DATABASE environment variable is not set to postgres. To test, build a new image and then run a new container:

$ docker build -f ./app/Dockerfile -t hello\_django:latest ./app

$ docker run -d **\**

-p 8006:8000 **\**

-e "SECRET\_KEY=please\_change\_me" -e "DEBUG=1" -e "DJANGO\_ALLOWED\_HOSTS=\*" **\**

hello\_django python /usr/src/app/manage.py runserver 0.0.0.0:8000

You should be able to view the welcome page at [http://localhost:8006](http://localhost:8006/)

Second, you may want to comment out the database flush and migrate commands in the *entrypoint.sh* script so they don't run on every container start or re-start:

*#!/bin/sh*

**if** [ "$DATABASE" = "postgres" ]

**then**

echo "Waiting for postgres..."

**while** ! nc -z $SQL\_HOST $SQL\_PORT; **do**

sleep 0.1

**done**

echo "PostgreSQL started"

**fi**

*# python manage.py flush --no-input*

*# python manage.py migrate*

exec "$@"

Instead, you can run them manually, after the containers spin up, like so:

$ docker-compose exec web python manage.py flush --no-input

$ docker-compose exec web python manage.py migrate

Gunicorn

Moving along, for production environments, let's add [Gunicorn](https://gunicorn.org/), a production-grade WSGI server, to the requirements file:

Django==3.2.6

gunicorn==20.1.0

psycopg2-binary==2.9.1

Curious about WSGI and Gunicorn? Review the [WSGI](https://testdriven.io/courses/python-web-framework/wsgi/) chapter from the [Building Your Own Python Web Framework](https://testdriven.io/courses/python-web-framework/) course.

Since we still want to use Django's built-in server in development, create a new compose file called *docker-compose.prod.yml* for production:

**version**: '3.8'

**services**:

**web**:

**build**: ./app

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**ports**:

- 8000:8000

**env\_file**:

- ./.env.prod

**depends\_on**:

- db

**db**:

**image**: postgres:13.0-alpine

**volumes**:

- postgres\_data:/var/lib/postgresql/data/

**env\_file**:

- ./.env.prod.db

**volumes**:

**postgres\_data**:

If you have multiple environments, you may want to look at using a [docker-compose.override.yml](https://docs.docker.com/compose/extends/) configuration file. With this approach, you'd add your base config to a *docker-compose.yml* file and then use a *docker-compose.override.yml* file to override those config settings based on the environment.

Take note of the default command. We're running Gunicorn rather than the Django development server. We also removed the volume from the web service since we don't need it in production. Finally, we're using [separate environment variable files](https://docs.docker.com/compose/env-file/) to define environment variables for both services that will be passed to the container at runtime.

*.env.prod*:

DEBUG=0

SECRET\_KEY=change\_me

DJANGO\_ALLOWED\_HOSTS=localhost 127.0.0.1 [::1]

SQL\_ENGINE=django.db.backends.postgresql

SQL\_DATABASE=hello\_django\_prod

SQL\_USER=hello\_django

SQL\_PASSWORD=hello\_django

SQL\_HOST=db

SQL\_PORT=5432

DATABASE=postgres

*.env.prod.db*:

POSTGRES\_USER=hello\_django

POSTGRES\_PASSWORD=hello\_django

POSTGRES\_DB=hello\_django\_prod

Add the two files to the project root. You'll probably want to keep them out of version control, so add them to a *.gitignore* file.

Bring [down](https://docs.docker.com/compose/reference/down/) the development containers (and the associated volumes with the -v flag):

$ docker-compose down -v

Then, build the production images and spin up the containers:

$ docker-compose -f docker-compose.prod.yml up -d --build

Verify that the hello\_django\_prod database was created along with the default Django tables. Test out the admin page at <http://localhost:8000/admin>. The static files are not being loaded anymore. This is expected since Debug mode is off. We'll fix this shortly.

Again, if the container fails to start, check for errors in the logs via docker-compose -f docker-compose.prod.yml logs -f.

Production Dockerfile

Did you notice that we're still running the database [flush](https://docs.djangoproject.com/en/2.2/ref/django-admin/#flush) (which clears out the database) and migrate commands every time the container is run? This is fine in development, but let's create a new entrypoint file for production.

*entrypoint.prod.sh*:

*#!/bin/sh*

**if** [ "$DATABASE" = "postgres" ]

**then**

echo "Waiting for postgres..."

**while** ! nc -z $SQL\_HOST $SQL\_PORT; **do**

sleep 0.1

**done**

echo "PostgreSQL started"

**fi**

exec "$@"

Update the file permissions locally:

$ chmod +x app/entrypoint.prod.sh

To use this file, create a new Dockerfile called *Dockerfile.prod* for use with production builds:

*###########*

*# BUILDER #*

*###########*

*# pull official base image*

**FROM** python:3.9.6-alpine **as** builder

*# set work directory*

**WORKDIR** /usr/src/app

*# set environment variables*

**ENV** PYTHONDONTWRITEBYTECODE 1

**ENV** PYTHONUNBUFFERED 1

*# install psycopg2 dependencies*

**RUN** apk update **\**

&& apk add postgresql-dev gcc python3-dev musl-dev

*# lint*

**RUN** pip install --upgrade pip

**RUN** pip install flake8==3.9.2

**COPY** . .

**RUN** flake8 --ignore=E501,F401 .

*# install dependencies*

**COPY** ./requirements.txt .

**RUN** pip wheel --no-cache-dir --no-deps --wheel-dir /usr/src/app/wheels -r requirements.txt

*#########*

*# FINAL #*

*#########*

*# pull official base image*

**FROM** python:3.9.6-alpine

*# create directory for the app user*

**RUN** mkdir -p /home/app

*# create the app user*

**RUN** addgroup -S app && adduser -S app -G app

*# create the appropriate directories*

**ENV** HOME=/home/app

**ENV** APP\_HOME=/home/app/web

**RUN** mkdir $APP\_HOME

**WORKDIR** $APP\_HOME

*# install dependencies*

**RUN** apk update && apk add libpq

**COPY** --from=builder /usr/src/app/wheels /wheels

**COPY** --from=builder /usr/src/app/requirements.txt .

**RUN** pip install --no-cache /wheels/\*

*# copy entrypoint.prod.sh*

**COPY** ./entrypoint.prod.sh .

**RUN** sed -i 's/\r$//g' $APP\_HOME/entrypoint.prod.sh

**RUN** chmod +x $APP\_HOME/entrypoint.prod.sh

*# copy project*

**COPY** . $APP\_HOME

*# chown all the files to the app user*

**RUN** chown -R app:app $APP\_HOME

*# change to the app user*

**USER** app

*# run entrypoint.prod.sh*

**ENTRYPOINT** ["/home/app/web/entrypoint.prod.sh"]

Here, we used a Docker [multi-stage build](https://docs.docker.com/develop/develop-images/multistage-build/) to reduce the final image size. Essentially, builder is a temporary image that's used for building the Python wheels. The wheels are then copied over to the final production image and the builder image is discarded.

You could take the [multi-stage build approach](https://stackoverflow.com/a/53101932/1799408) a step further and use a single *Dockerfile* instead of creating two Dockerfiles. Think of the pros and cons of using this approach over two different files.

Did you notice that we created a non-root user? By default, Docker runs container processes as root inside of a container. This is a bad practice since attackers can gain root access to the Docker host if they manage to break out of the container. If you're root in the container, you'll be root on the host.

Update the web service within the *docker-compose.prod.yml* file to build with *Dockerfile.prod*:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**ports**:

- 8000:8000

**env\_file**:

- ./.env.prod

**depends\_on**:

- db

Try it out:

$ docker-compose -f docker-compose.prod.yml down -v

$ docker-compose -f docker-compose.prod.yml up -d --build

$ docker-compose -f docker-compose.prod.yml exec web python manage.py migrate --noinput

Nginx

Next, let's add Nginx into the mix to act as a [reverse proxy](https://www.nginx.com/resources/glossary/reverse-proxy-server/) for Gunicorn to handle client requests as well as serve up static files.

Add the service to *docker-compose.prod.yml*:

**nginx**:

**build**: ./nginx

**ports**:

- 1337:80

**depends\_on**:

- web

Then, in the local project root, create the following files and folders:

└── nginx

├── Dockerfile

└── nginx.conf

*Dockerfile*:

**FROM** nginx:1.21-alpine

**RUN** rm /etc/nginx/conf.d/default.conf

**COPY** nginx.conf /etc/nginx/conf.d

*nginx.conf*:

upstream hello\_django {

server web:8000;

}

server {

listen 80;

location / {

proxy\_pass http://hello\_django;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

}

}

Review [Using NGINX and NGINX Plus as an Application Gateway with uWSGI and Django](https://docs.nginx.com/nginx/admin-guide/web-server/app-gateway-uwsgi-django/) for more info on configuring Nginx to work with Django.

Then, update the web service, in *docker-compose.prod.yml*, replacing ports with expose:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**expose**:

- 8000

**env\_file**:

- ./.env.prod

**depends\_on**:

- db

Now, port 8000 is only exposed internally, to other Docker services. The port will no longer be published to the host machine.

For more on ports vs expose, review [this](https://stackoverflow.com/questions/40801772/what-is-the-difference-between-docker-compose-ports-vs-expose) Stack Overflow question.

Test it out again.

$ docker-compose -f docker-compose.prod.yml down -v

$ docker-compose -f docker-compose.prod.yml up -d --build

$ docker-compose -f docker-compose.prod.yml exec web python manage.py migrate --noinput

Ensure the app is up and running at [http://localhost:1337](http://localhost:1337/).

Your project structure should now look like:

├── .env.dev

├── .env.prod

├── .env.prod.db

├── .gitignore

├── app

│ ├── Dockerfile

│ ├── Dockerfile.prod

│ ├── entrypoint.prod.sh

│ ├── entrypoint.sh

│   ├── hello\_django

│   │   ├── \_\_init\_\_.py

│   │   ├── asgi.py

│   │   ├── settings.py

│   │   ├── urls.py

│   │   └── wsgi.py

│ ├── manage.py

│ └── requirements.txt

├── docker-compose.prod.yml

├── docker-compose.yml

└── nginx

├── Dockerfile

└── nginx.conf

Bring the containers down once done:

$ docker-compose -f docker-compose.prod.yml down -v

Since Gunicorn is an application server, it will not serve up static files. So, how should both static and media files be handled in this particular configuration?

Static Files

Update *settings.py*:

STATIC\_URL = "/static/"

STATIC\_ROOT = BASE\_DIR / "staticfiles"

Development

Now, any request to http://localhost:8000/static/\* will be served from the "staticfiles" directory.

To test, first re-build the images and spin up the new containers per usual. Ensure static files are still being served correctly at <http://localhost:8000/admin>.

Production

For production, add a volume to the web and nginx services in *docker-compose.prod.yml* so that each container will share a directory named "staticfiles":

**version**: '3.8'

**services**:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**volumes**:

- static\_volume:/home/app/web/staticfiles

**expose**:

- 8000

**env\_file**:

- ./.env.prod

**depends\_on**:

- db

**db**:

**image**: postgres:13.0-alpine

**volumes**:

- postgres\_data:/var/lib/postgresql/data/

**env\_file**:

- ./.env.prod.db

**nginx**:

**build**: ./nginx

**volumes**:

- static\_volume:/home/app/web/staticfiles

**ports**:

- 1337:80

**depends\_on**:

- web

**volumes**:

**postgres\_data**:

**static\_volume**:

We need to also create the "/home/app/web/staticfiles" folder in *Dockerfile.prod*:

...

*# create the appropriate directories*

**ENV** HOME=/home/app

**ENV** APP\_HOME=/home/app/web

**RUN** mkdir $APP\_HOME

**RUN** mkdir $APP\_HOME/staticfiles

**WORKDIR** $APP\_HOME

...

Why is this necessary?

Docker Compose normally mounts named volumes as root. And since we're using a non-root user, we'll get a permission denied error when the collectstatic command is run if the directory does not already exist

To get around this, you can either:

1. Create the folder in the Dockerfile ([source](https://github.com/docker/compose/issues/3270#issuecomment-206214034))
2. Change the permissions of the directory after it's mounted ([source](https://stackoverflow.com/a/40510068/1799408))

We used the former.

Next, update the Nginx configuration to route static file requests to the "staticfiles" folder:

upstream hello\_django {

server web:8000;

}

server {

listen 80;

location / {

proxy\_pass http://hello\_django;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

}

location /static/ {

alias /home/app/web/staticfiles/;

}

}

Spin down the development containers:

$ docker-compose down -v

Test:

$ docker-compose -f docker-compose.prod.yml up -d --build

$ docker-compose -f docker-compose.prod.yml exec web python manage.py migrate --noinput

$ docker-compose -f docker-compose.prod.yml exec web python manage.py collectstatic --no-input --clear

Again, requests to http://localhost:1337/static/\* will be served from the "staticfiles" directory.

Navigate to <http://localhost:1337/admin> and ensure the static assets load correctly.

You can also verify in the logs -- via docker-compose -f docker-compose.prod.yml logs -f -- that requests to the static files are served up successfully via Nginx:

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /admin/ HTTP/1.1" 302 0 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /admin/login/?next=/admin/ HTTP/1.1" 200 2214 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/css/base.css HTTP/1.1" 304 0 "http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/css/nav\_sidebar.css HTTP/1.1" 304 0 "http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/css/responsive.css HTTP/1.1" 304 0 "http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/css/login.css HTTP/1.1" 304 0 "http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/js/nav\_sidebar.js HTTP/1.1" 304 0 "http://localhost:1337/admin/login/?next=/admin/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/css/fonts.css HTTP/1.1" 304 0 "http://localhost:1337/static/admin/css/base.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/fonts/Roboto-Regular-webfont.woff HTTP/1.1" 304 0 "http://localhost:1337/static/admin/css/fonts.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

nginx\_1 | 192.168.144.1 - - [23/Aug/2021:20:11:00 +0000] "GET /static/admin/fonts/Roboto-Light-webfont.woff HTTP/1.1" 304 0 "http://localhost:1337/static/admin/css/fonts.css" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_15\_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/92.0.4515.159 Safari/537.36" "-"

Bring the containers once done:

$ docker-compose -f docker-compose.prod.yml down -v

Media Files

To test out the handling of media files, start by creating a new Django app:

$ docker-compose up -d --build

$ docker-compose exec web python manage.py startapp upload

Add the new app to the INSTALLED\_APPS list in *settings.py*:

INSTALLED\_APPS = [

"django.contrib.admin",

"django.contrib.auth",

"django.contrib.contenttypes",

"django.contrib.sessions",

"django.contrib.messages",

"django.contrib.staticfiles",

"upload",

]

*app/upload/views.py*:

**from** **django.shortcuts** **import** render

**from** **django.core.files.storage** **import** FileSystemStorage

**def** image\_upload(request):

**if** request.method == "POST" **and** request.FILES["image\_file"]:

image\_file = request.FILES["image\_file"]

fs = FileSystemStorage()

filename = fs.save(image\_file.name, image\_file)

image\_url = fs.url(filename)

print(image\_url)

**return** render(request, "upload.html", {

"image\_url": image\_url

})

**return** render(request, "upload.html")

Add a "templates", directory to the "app/upload" directory, and then add a new template called *upload.html*:

{% block content %}

<**form** action="{% url "upload" %}" method="post" enctype="multipart/form-data">

{% csrf\_token %}

<**input** type="file" name="image\_file">

<**input** type="submit" value="submit" />

</**form**>

{% if image\_url %}

<**p**>File uploaded at: <**a** href="{{ image\_url }}">{{ image\_url }}</**a**></**p**>

{% endif %}

{% endblock %}

*app/hello\_django/urls.py*:

**from** **django.contrib** **import** admin

**from** **django.urls** **import** path

**from** **django.conf** **import** settings

**from** **django.conf.urls.static** **import** static

**from** **upload.views** **import** image\_upload

urlpatterns = [

path("", image\_upload, name="upload"),

path("admin/", admin.site.urls),

]

**if** bool(settings.DEBUG):

urlpatterns += static(settings.MEDIA\_URL, document\_root=settings.MEDIA\_ROOT)

*app/hello\_django/settings.py*:

MEDIA\_URL = "/media/"

MEDIA\_ROOT = BASE\_DIR / "mediafiles"

Development

Test:

$ docker-compose up -d --build

You should be able to upload an image at <http://localhost:8000/>, and then view the image at <http://localhost:8000/media/IMAGE_FILE_NAME>.

Production

For production, add another volume to the web and nginx services:

**version**: '3.8'

**services**:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

**expose**:

- 8000

**env\_file**:

- ./.env.prod

**depends\_on**:

- db

**db**:

**image**: postgres:13.0-alpine

**volumes**:

- postgres\_data:/var/lib/postgresql/data/

**env\_file**:

- ./.env.prod.db

**nginx**:

**build**: ./nginx

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

**ports**:

- 1337:80

**depends\_on**:

- web

**volumes**:

**postgres\_data**:

**static\_volume**:

**media\_volume**:

Create the "/home/app/web/mediafiles" folder in *Dockerfile.prod*:

...

*# create the appropriate directories*

**ENV** HOME=/home/app

**ENV** APP\_HOME=/home/app/web

**RUN** mkdir $APP\_HOME

**RUN** mkdir $APP\_HOME/staticfiles

**RUN** mkdir $APP\_HOME/mediafiles

**WORKDIR** $APP\_HOME

...

Update the Nginx config again:

upstream hello\_django {

server web:8000;

}

server {

listen 80;

location / {

proxy\_pass http://hello\_django;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

}

location /static/ {

alias /home/app/web/staticfiles/;

}

location /media/ {

alias /home/app/web/mediafiles/;

}

}

Re-build:

$ docker-compose down -v

$ docker-compose -f docker-compose.prod.yml up -d --build

$ docker-compose -f docker-compose.prod.yml exec web python manage.py migrate --noinput

$ docker-compose -f docker-compose.prod.yml exec web python manage.py collectstatic --no-input --clear

Test it out one final time:

1. Upload an image at <http://localhost:1337/>.
2. Then, view the image at <http://localhost:1337/media/IMAGE_FILE_NAME>.

If you see an 413 Request Entity Too Large error, you'll need to [increase the maximum allowed size of the client request body](https://stackoverflow.com/a/28476755/1799408) in either the server or location context within the Nginx config.

Example:

location / {

proxy\_pass http://hello\_django;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $host;

proxy\_redirect off;

client\_max\_body\_size 100M;

}

There are a number of different ways to secure a containerized Django app with HTTPS. Arguably, the most popular approach is to add a new service to your Docker Compose file that utilizes [Certbot](https://certbot.eff.org/) for issuing and renewing SSL certificates. While this is perfectly valid, we'll take a slightly different approach and use the following projects:

1. [nginx-proxy](https://github.com/nginx-proxy/nginx-proxy) - used to automatically build your Nginx proxy configuration for running containers where each container is treated as a single virtual host
2. [letsencrypt-nginx-proxy-companion](https://github.com/nginx-proxy/docker-letsencrypt-nginx-proxy-companion) - used to issue and renew Let's Encrypt SSL certificates for each of the containers proxied by nginx-proxy

Together, these projects simplify the management of your Nginx configuration and SSL certificates.

Another option is to use [Traefik](https://traefik.io/) instead of Nginx. In short, Traefik works with Let's Encrypt to issue and renew certificates. For more, check out [Dockerizing Django with Postgres, Gunicorn, and Traefik](https://testdriven.io/blog/django-docker-traefik/).

Let's Encrypt

When the app is deployed for the first time, you should follow these two steps to avoid issues with certificates:

1. Start by issuing the certificates from Let's Encrypt's [staging environment](https://letsencrypt.org/docs/staging-environment/)
2. Then, when all is running as expected, switch to Let's Encrypt's production environment

Why?

To protect their servers, Let's Encrypt enforces [rate limitations](https://letsencrypt.org/docs/rate-limits/) on their production validation system:

1. 5 validation failures per account, per hostname, per hour
2. 50 certificates may be created per domain per week

If you make a typo in your domain name or in a DNS entry or anything similar, your request will fail, which will count against your rate limit, and you'll have to attempt to issue a new certificate.

To avoid being rate limited, during development and testing, you should use Let's Encrypt's staging environment for testing their validation system. The rate limits are much [higher](https://letsencrypt.org/docs/staging-environment/) on the staging environment, which is better for testing. Just be aware that the issued certificates in staging are not trusted publicly, so once everything is working, you should switch over to their production environment.

Project Setup

First, clone down the contents from the GitHub project repo:

$ git clone https://github.com/testdrivenio/django-on-docker django-on-docker-letsencrypt

$ cd django-on-docker-letsencrypt

This repository contains everything that you need to deploy a Dockerized Django app minus the SSL certificates, which we'll be adding in this tutorial.

Django Configuration

First, to run the Django app behind an HTTPS proxy you'll need to add the [SECURE\_PROXY\_SSL\_HEADER](https://docs.djangoproject.com/en/3.2/ref/settings/#secure-proxy-ssl-header) setting to *settings.py*:

SECURE\_PROXY\_SSL\_HEADER = ("HTTP\_X\_FORWARDED\_PROTO", "https")

In this tuple, when X-Forwarded-Proto is set to https the request is secure.

Docker Compose

It's time to configure Docker Compose.

Let's add a new Docker Compose file for testing purposes called *docker-compose.staging.yml*:

**version**: '3.8'

**services**:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

**expose**:

- 8000

**env\_file**:

- ./.env.staging

**depends\_on**:

- db

**db**:

**image**: postgres:13.0-alpine

**volumes**:

- postgres\_data:/var/lib/postgresql/data/

**env\_file**:

- ./.env.staging.db

**nginx-proxy**:

**container\_name**: nginx-proxy

**build**: nginx

**restart**: always

**ports**:

- 443:443

- 80:80

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- /var/run/docker.sock:/tmp/docker.sock:ro

**depends\_on**:

- web

**nginx-proxy-letsencrypt**:

**image**: jrcs/letsencrypt-nginx-proxy-companion

**env\_file**:

- ./.env.staging.proxy-companion

**volumes**:

- /var/run/docker.sock:/var/run/docker.sock:ro

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- acme:/etc/acme.sh

**depends\_on**:

- nginx-proxy

**volumes**:

**postgres\_data**:

**static\_volume**:

**media\_volume**:

**certs**:

**html**:

**vhost**:

**acme**:

Add a *.env.staging.db* file for the db container:

POSTGRES\_USER=hello\_django

POSTGRES\_PASSWORD=hello\_django

POSTGRES\_DB=hello\_django\_prod

Change the values of POSTGRES\_USER and POSTGRES\_PASSWORD to match your user and password.

We already looked at the web and db services in the previous [tutorial](https://testdriven.io/blog/dockerizing-django-with-postgres-gunicorn-and-nginx/), so let's dive into the nginx-proxy and nginx-proxy-letsencrypt services.

Databases are [critical services](https://vsupalov.com/database-in-docker/). Adding additional layers, such us Docker, adds unnecessary risk in production. To simplify tasks such as minor version updates, regular backups, and scaling, it's recommended to use a managed service like [AWS RDS](https://aws.amazon.com/rds/), [Google Cloud SQL](https://cloud.google.com/sql), or [DigitalOcean's Managed Database](https://www.digitalocean.com/products/managed-databases/).

Nginx Proxy Service

For this service, the [nginx-proxy](https://github.com/nginx-proxy/nginx-proxy) project is used for generating a reverse proxy configuration for the web container using virtual hosts for routing.

Be sure to review the README on the [nginx-proxy](https://github.com/nginx-proxy/nginx-proxy) repo.

Once up, the container associated with nginx-proxy automatically detects containers (in the same network) that have the VIRTUAL\_HOST environment variable set and dynamically updates its virtual hosts configuration.

Go ahead and add a *.env.staging* file for the web container:

DEBUG=0

SECRET\_KEY=change\_me

DJANGO\_ALLOWED\_HOSTS=<YOUR\_DOMAIN.COM>

SQL\_ENGINE=django.db.backends.postgresql

SQL\_DATABASE=hello\_django\_prod

SQL\_USER=hello\_django

SQL\_PASSWORD=hello\_django

SQL\_HOST=db

SQL\_PORT=5432

DATABASE=postgres

VIRTUAL\_HOST=<YOUR\_DOMAIN.COM>

VIRTUAL\_PORT=8000

LETSENCRYPT\_HOST=<YOUR\_DOMAIN.COM>

Notes:

1. Change <YOUR\_DOMAIN.COM> to your actual domain, and change the default values of SQL\_USER and SQL\_PASSWORD to match POSTGRES\_USER and POSTGRES\_PASSWORD (from *.env.staging.db*).
2. As mentioned, VIRTUAL\_HOST (and VIRTUAL\_PORT) are needed by nginx-proxy to auto create the reverse proxy configuration.
3. LETSENCRYPT\_HOST is there so the nginx-proxy-companion can issue Let's Encrypt certificate for your domain.
4. Since the Django app will be listening on port 8000, we also set the VIRTUAL\_PORT environment variable.
5. The /var/run/docker.sock:/tmp/docker.sock:ro volume in *docker-compose.staging.yml* is used to listen for newly registered/de-registered containers.

For testing/debugging purposes you may want to use a \* for DJANGO\_ALLOWED\_HOSTS the first time you deploy to simplify things. Just don't forget to limit the allowed hosts once testing is complete.

So, requests made to the specified domain will be [proxied](https://timothy-quinn.com/using-nginx-as-a-reverse-proxy-for-multiple-sites/) to the container that has the domain set as the VIRTUAL\_HOST environment variable.

Next, let's update the Nginx configuration in the "nginx" folder.

First, add directory called "vhost.d". Then, add a file called *default* inside that directory to serve static and media files:

location /static/ {

alias /home/app/web/staticfiles/;

add\_header Access-Control-Allow-Origin \*;

}

location /media/ {

alias /home/app/web/mediafiles/;

add\_header Access-Control-Allow-Origin \*;

}

Requests that match any of these patterns will be served from static or media folders. They won't be proxied to other containers. The web and nginx-proxy containers share the volumes in which the static and media files are located:

static\_volume:/home/app/web/staticfiles

media\_volume:/home/app/web/mediafiles

Add a *custom.conf* file to the "nginx" folder to hold custom proxy-wide configuration:

client\_max\_body\_size 10M;

Update *nginx/Dockerfile*:

**FROM** jwilder/nginx-proxy:0.9

**COPY** vhost.d/default /etc/nginx/vhost.d/default

**COPY** custom.conf /etc/nginx/conf.d/custom.conf

Remove *nginx.conf*.

Your "nginx" directory should now look like this:

└── nginx

├── Dockerfile

├── custom.conf

└── vhost.d

└── default

Let's Encrypt Nginx Proxy Companion Service

While the nginx-proxy service handles routing, nginx-proxy-letsencrypt (via [letsencrypt-nginx-proxy-companion](https://github.com/nginx-proxy/docker-letsencrypt-nginx-proxy-companion)) handles the creation, renewal, and use of Let's Encrypt certificates for proxied Docker containers.

To issue and renew certificates for proxied containers, the LETSENCRYPT\_HOST environment variable needs to be added to each of them (which we've already done). It must also have the same value as VIRTUAL\_HOST.

This container must share the following volumes with nginx-proxy:

1. certs:/etc/nginx/certs stores certificates, private keys, and ACME account keys
2. html:/usr/share/nginx/html writes [http-01](https://tools.ietf.org/html/draft-ietf-acme-acme-03#section-7.2) challenge files
3. vhost:/etc/nginx/vhost.d changes the configuration of vhosts

For more, review the [official documentation](https://github.com/nginx-proxy/docker-letsencrypt-nginx-proxy-companion#basic-usage-with-the-nginx-proxy-container).

Add a *.env.staging.proxy-companion* file:

DEFAULT\_EMAIL=youremail@yourdomain.com

ACME\_CA\_URI=https://acme-staging-v02.api.letsencrypt.org/directory

NGINX\_PROXY\_CONTAINER=nginx-proxy

Notes:

1. DEFAULT\_EMAIL is the email that Let's Encrypt will use to send you notifications about your certificates (including renewals).
2. ACME\_CA\_URI is the URL used to issue certificates. Again, use staging until you're 100% sure that everything works.
3. NGINX\_PROXY\_CONTAINER is the name of nginx-proxy container.

Running the Containers

Everything is ready to go for deployment.

It's time to move to your Linux instance.

Assuming you have a project directory created on your instance, like */home/myuser/django-on-docker*, copy the files and folders over with SCP:

$ scp -r **$(**pwd**)**/{app,nginx,.env.staging,.env.staging.db,.env.staging.proxy-companion,docker-compose.staging.yml} user@your-ip-or-domain:/path/to/django-on-docker

Connect to your instance via SSH and move to the project directory:

$ ssh user@your-ip-or-domain

$ cd /path/to/django-on-docker

When that, you're ready to build the images and spin up the containers:

$ docker-compose -f docker-compose.staging.yml up -d --build

Once the containers are up and running, navigate to your domain in your browser. You should see something like:

Graphical user interface, text, application, email

Description automatically generated

This is expected. This screen is shown because the certificate was issued from a [staging environment](https://letsencrypt.org/docs/staging-environment/), which, again, doesn't have the same [rate limits](https://letsencrypt.org/docs/staging-environment/#rate-limits) as a production environment. It's similar to a [self-signed HTTPS certificate](https://en.wikipedia.org/wiki/Self-signed_certificate). Always use a staging environment until you're sure that everything is working as expected.

How do you know if everything works?

Click on "Advanced" and then on "Proceed". You should now see your app. Upload an image, and then make sure you can view the image at https://yourdomain.com/mediafiles/IMAGE\_FILE\_NAME.

Issue the Production Certificate

Now, that everything works as expected, we can switch over to Let's Encrypt's production environment.

Bring down the existing containers and exit your instance:

$ docker-compose -f docker-compose.staging.yml down -v

$ exit

Back on your local machine, update *docker-compose.prod.yml*:

**version**: '3.8'

**services**:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

**expose**:

- 8000

**env\_file**:

- ./.env.prod

**depends\_on**:

- db

**db**:

**image**: postgres:13.0-alpine

**volumes**:

- postgres\_data:/var/lib/postgresql/data/

**env\_file**:

- ./.env.prod.db

**nginx-proxy**:

**container\_name**: nginx-proxy

**build**: nginx

**restart**: always

**ports**:

- 443:443

- 80:80

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- /var/run/docker.sock:/tmp/docker.sock:ro

**depends\_on**:

- web

**nginx-proxy-letsencrypt**:

**image**: jrcs/letsencrypt-nginx-proxy-companion

**env\_file**:

- ./.env.prod.proxy-companion

**volumes**:

- /var/run/docker.sock:/var/run/docker.sock:ro

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- acme:/etc/acme.sh

**depends\_on**:

- nginx-proxy

**volumes**:

**postgres\_data**:

**static\_volume**:

**media\_volume**:

**certs**:

**html**:

**vhost**:

**acme**:

The only difference here, compared to *docker-compose.staging.yml*, is that we used different environment files.

*.env.prod*:

DEBUG=0

SECRET\_KEY=change\_me

DJANGO\_ALLOWED\_HOSTS=<YOUR\_DOMAIN.COM>

SQL\_ENGINE=django.db.backends.postgresql

SQL\_DATABASE=hello\_django\_prod

SQL\_USER=hello\_django

SQL\_PASSWORD=hello\_django

SQL\_HOST=db

SQL\_PORT=5432

DATABASE=postgres

VIRTUAL\_HOST=<YOUR\_DOMAIN.COM>

VIRTUAL\_PORT=8000

LETSENCRYPT\_HOST=<YOUR\_DOMAIN.COM>

*.env.prod.db*:

POSTGRES\_USER=hello\_django

POSTGRES\_PASSWORD=hello\_django

POSTGRES\_DB=hello\_django\_prod

*.env.prod.proxy-companion*:

DEFAULT\_EMAIL=youremail@yourdomain.co

NGINX\_PROXY\_CONTAINER=nginx-proxy

Update them appropriately.

Did you spot the difference, from the staging versions? The ACME\_CA\_URI environment variable is not set since the letsencrypt-nginx-proxy-companion image uses Let's Encrypt's production environment by default.

Copy the new files and folders to your instance with SCP:

$ scp **$(**pwd**)**/{.env.prod,.env.prod.db,.env.prod.proxy-companion,docker-compose.prod.yml} user@your-ip-or-domain:/path/to/django-on-docker

Like before, connect to your instance via SSH and move to the project directory:

$ ssh user@your-ip-or-domain

$ cd /path/to/django-on-docker

Build the images and spin up the containers:

$ docker-compose -f docker-compose.prod.yml up -d --build

Navigate to your domain again. You should no longer see a warning.

Congrats! You're now using a production Let's Encrypt certificate.

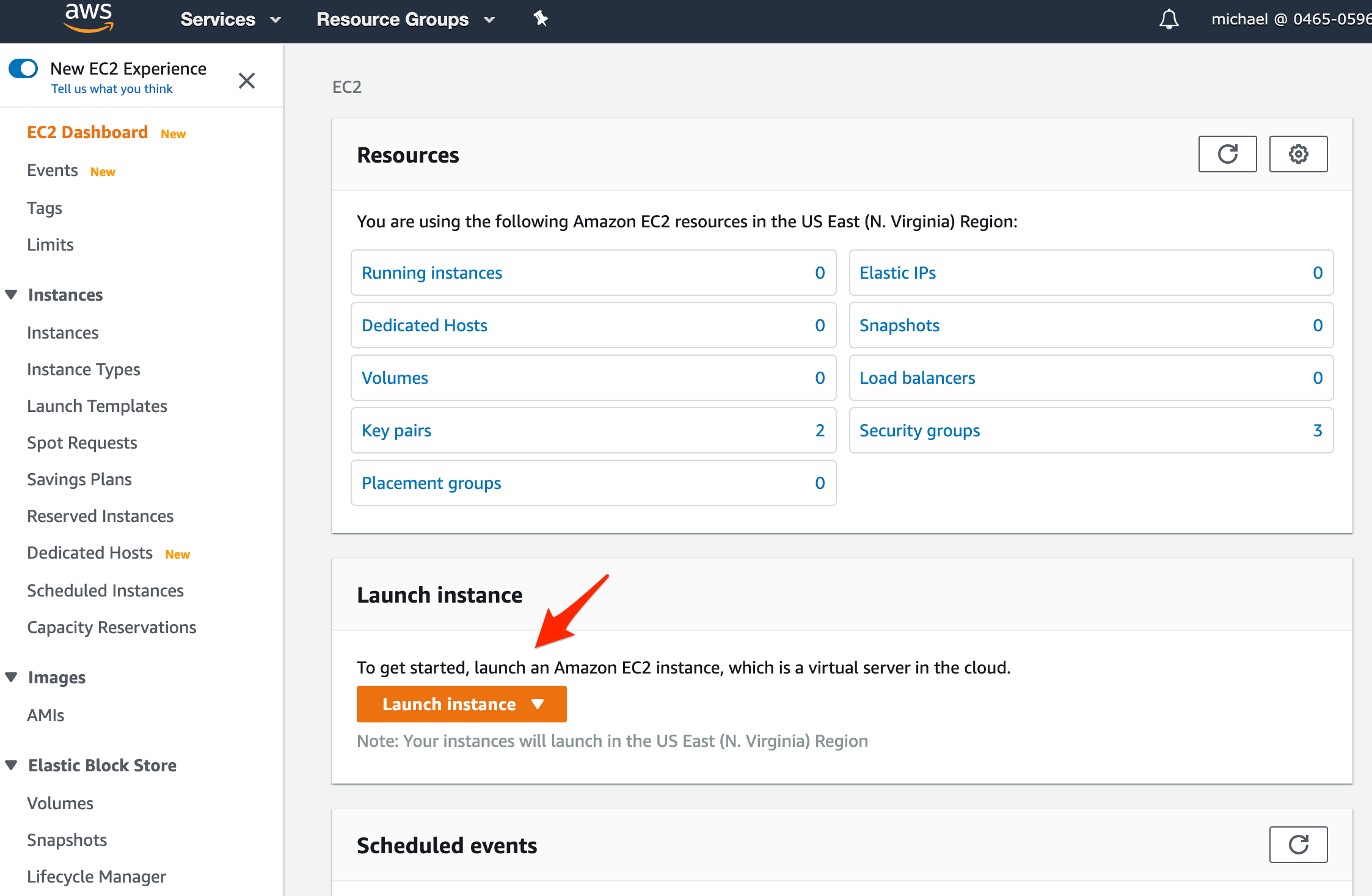
Want to see the certificate creation process in action, check out the logs:

$ docker-compose -f docker-compose.prod.yml logs nginx-proxy-letsencrypt

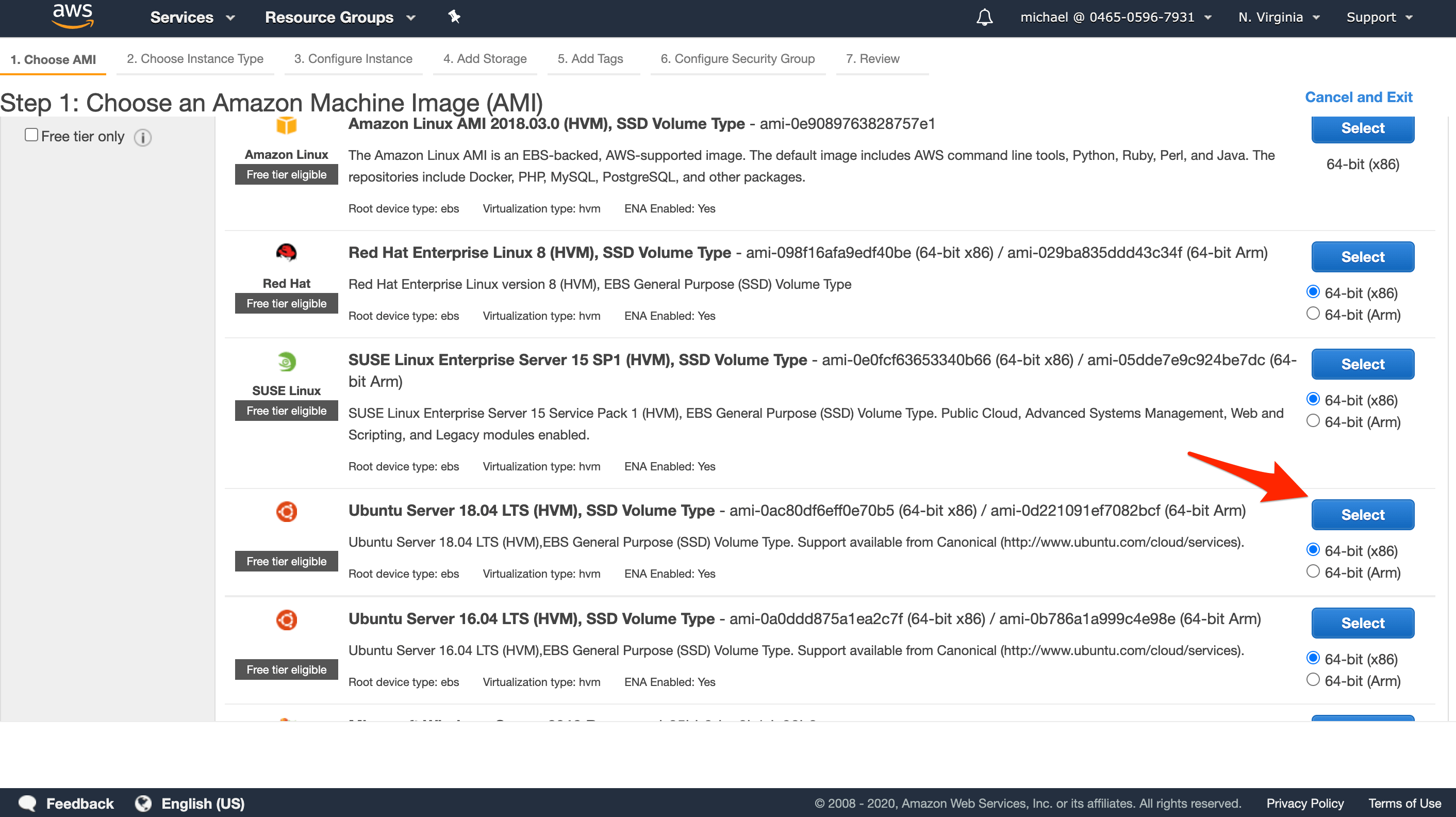
AWS EC2

First, create an [AWS](https://portal.aws.amazon.com/billing/signup#/start) account if you don't already have one.

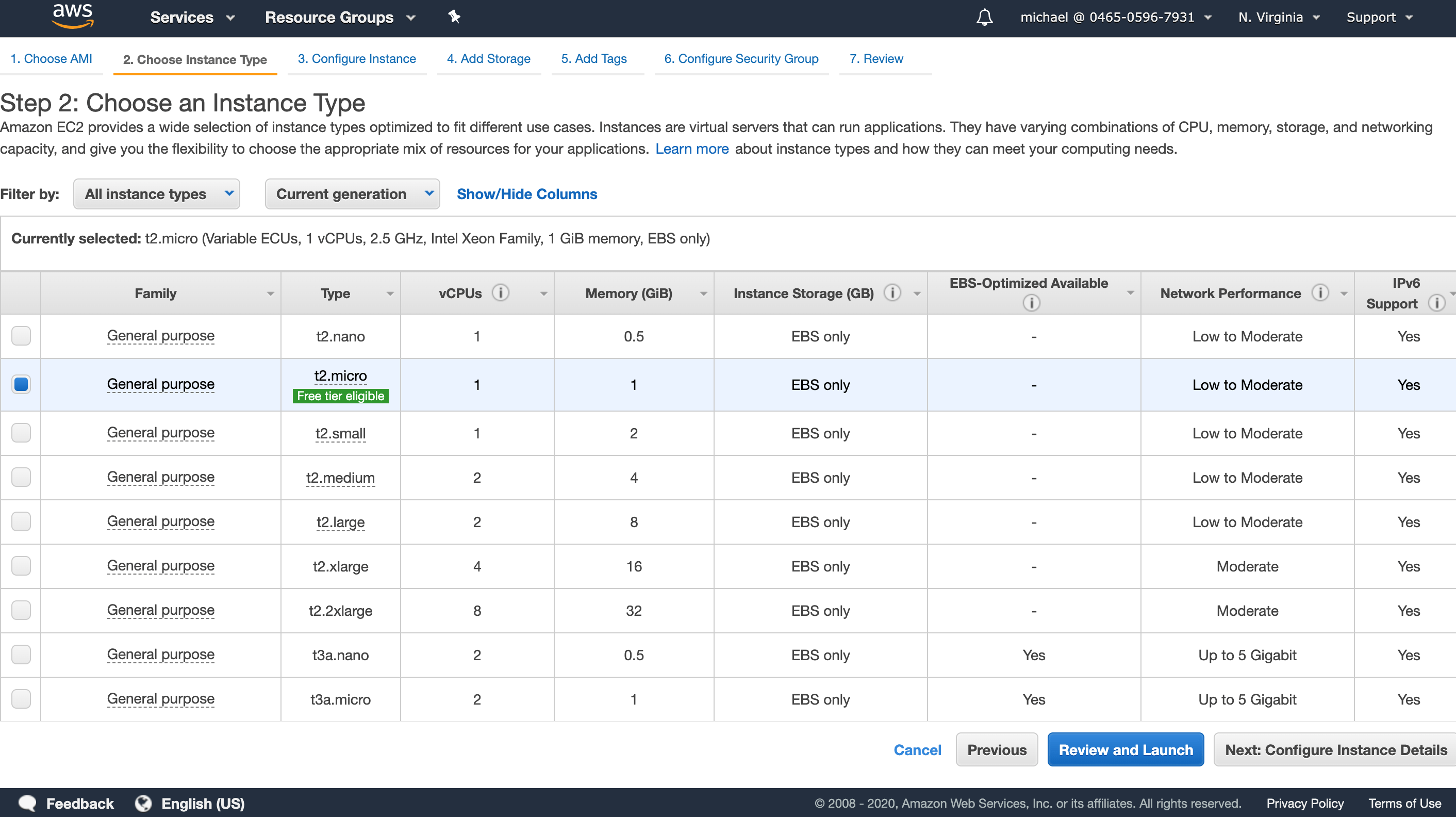
Next, navigate to the [EC2 console](https://console.aws.amazon.com/ec2/) and click **Launch instance**:



Use **Ubuntu Server 18.04 LTS (HVM)** for the server image (AMI):



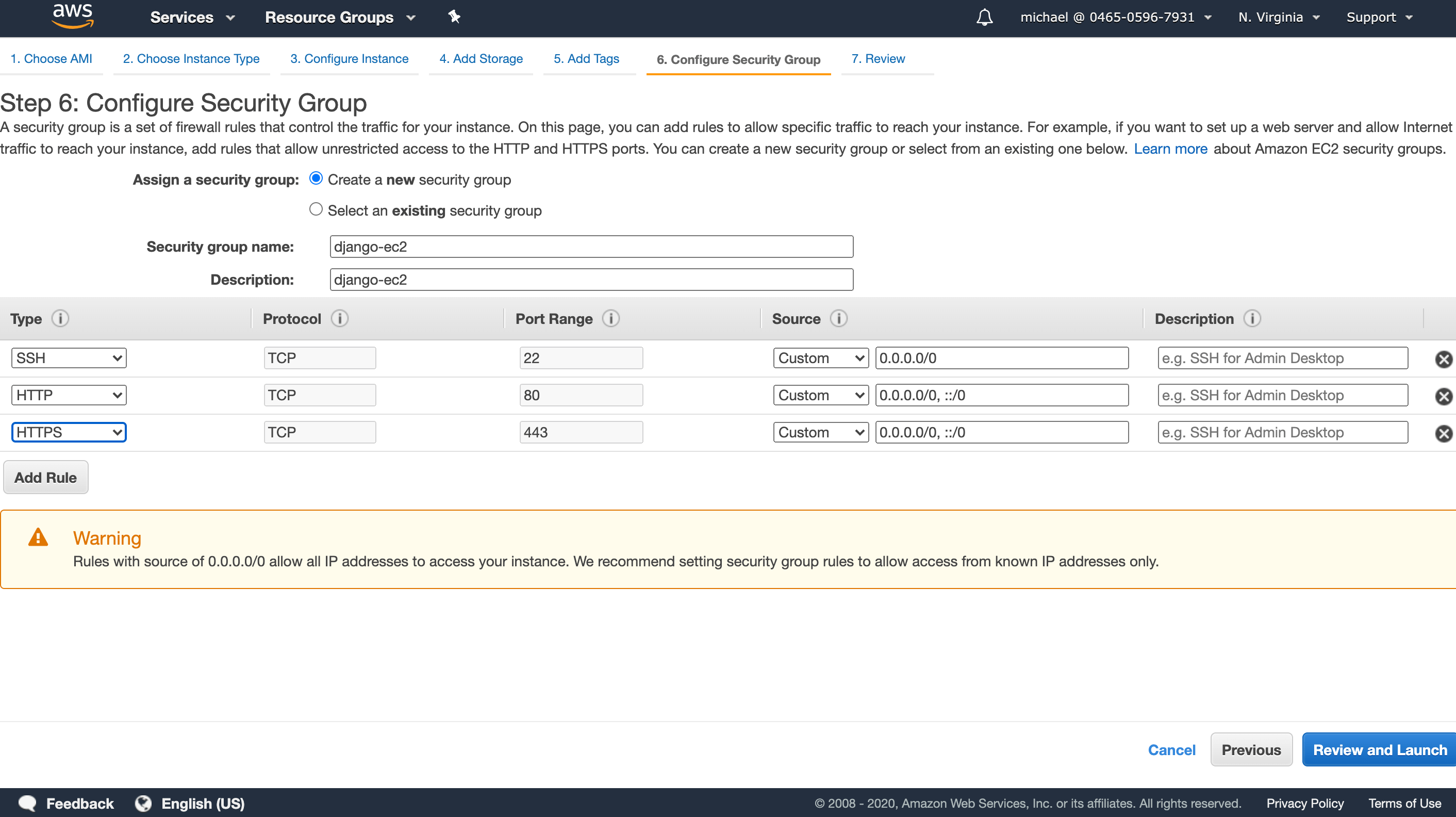
In the next step, stick with the **t2.micro** instance. Click on **Next: Configure Instance Details**:



At the **Configure Instance Details** step, leave everything as it is to keep things simple. Then click **Next** a few times until you're at the **Configure Security Group** step.

With **Create a new security group** selected, set the name and description to django-ec2 and add two rules:

* HTTP -> Anywhere
* HTTPS -> Anywhere

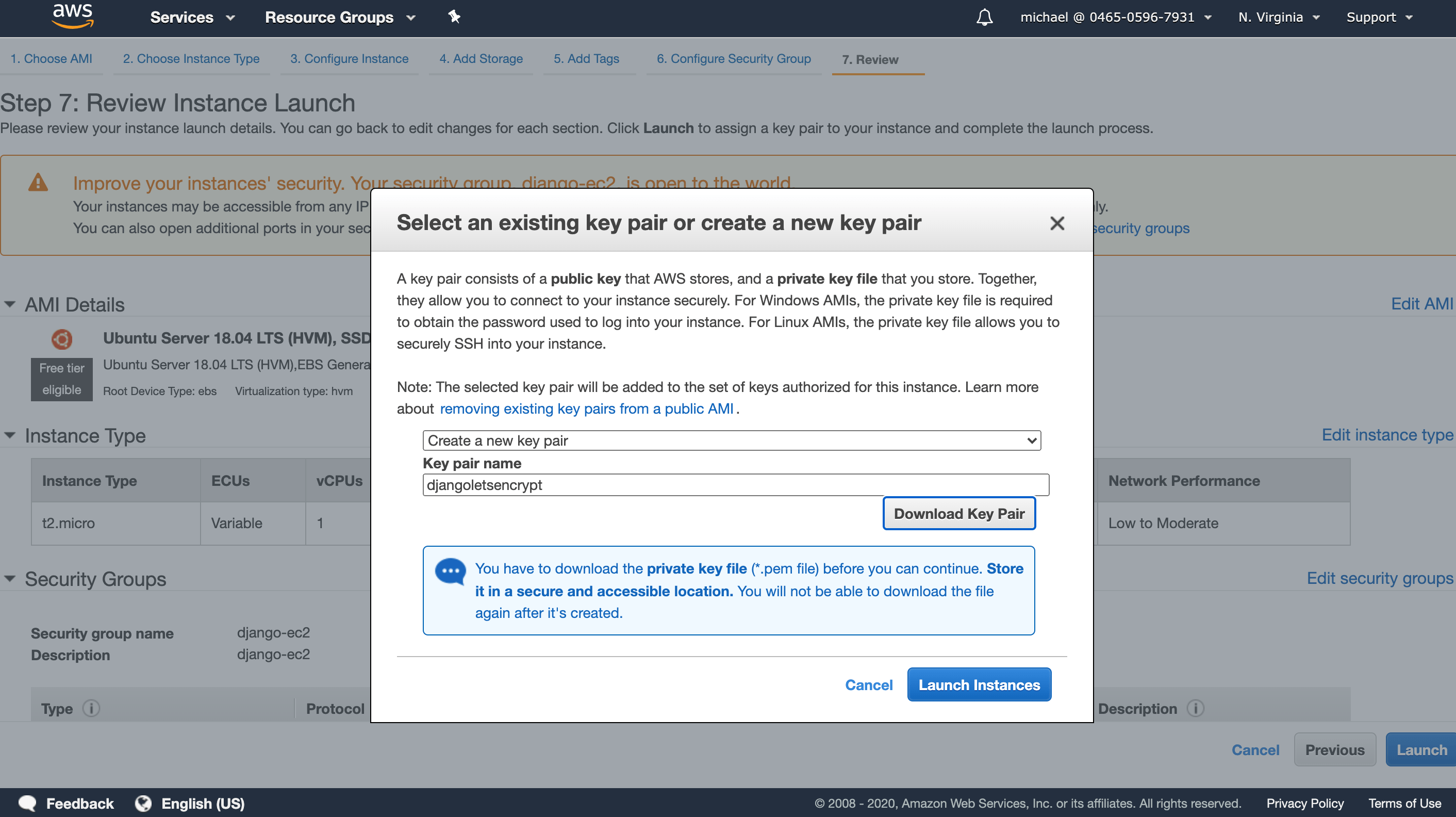


These rules are needed to issue certificates and to access the app.

Security group *inbound* rules are used to limit access to your instance from the internet. Unless you have some additional security requirements, you'll probably want to allow HTTP and HTTPS traffic from anywhere for instances hosting web apps. SSH must be allowed for you to connect to the instance for set up and deployment.

Click **Review and Launch**. On the next screen, click **Launch**.

You'll be prompted to select a key pair. You need it for SSH connection to your instance. Select **Create a new key pair** and name it djangoletsencrypt. Then click **Download key pair**. After the key pair is downloaded, click on **Launch Instances**:



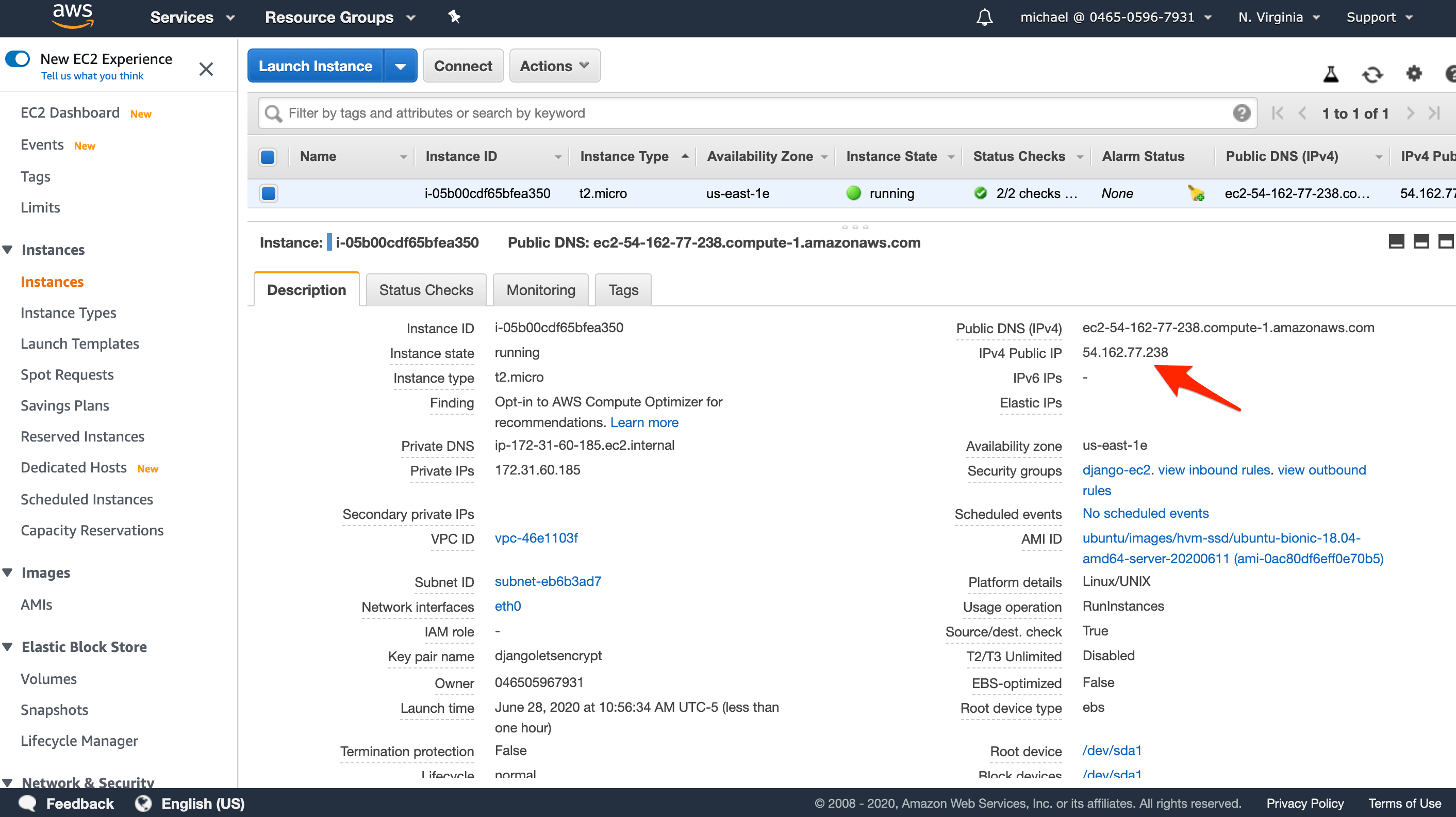
It will take a few minutes for the instance to spin up.

Configure EC2 Instance

In this section, we'll install Docker on the instance, add an Elastic IP, and configure an IAM role.

Install Docker

Navigate back to the [EC2 console](https://console.aws.amazon.com/ec2/), select the newly created instance, and grab the public IP address:



Connect to your EC2 instance using the .pem key that we downloaded in the "AWS EC2" step.

$ ssh -i /path/to/your/djangoletsencrypt.pem ubuntu@public-ip-or-domain-of-ec2-instance

Your .pem was probably downloaded into path like ~/Downloads/djangoletsencrypt.pem. If you're not sure where to store it, move it into the *~/.ssh* directory. You may have to also change the permissions -- i.e., chmod 400 -i /path/to/your/djangoletsencrypt.pem.

Start by installing the latest version of Docker and version 1.29.2 of Docker Compose:

$ sudo apt update

$ sudo apt install apt-transport-https ca-certificates curl software-properties-common

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

$ sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"

$ sudo apt update

$ sudo apt install docker-ce

$ sudo usermod -aG docker **${**USER**}**

$ sudo curl -L "https://github.com/docker/compose/releases/download/1.29.2/docker-compose-**$(**uname -s**)**-**$(**uname -m**)**" -o /usr/local/bin/docker-compose

$ sudo chmod +x /usr/local/bin/docker-compose

$ docker -v

Docker version 20.10.8, build 3967b7d

$ docker-compose -v

docker-compose version 1.29.2, build 5becea4c

Install AWS CLI

First, install unzip:

$ sudo apt install unzip

Download AWS CLI ZIP:

$ curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip"

Unzip its content:

$ unzip awscliv2.zip

Install AWS CLI:

$ sudo ./aws/install

Verify installation:

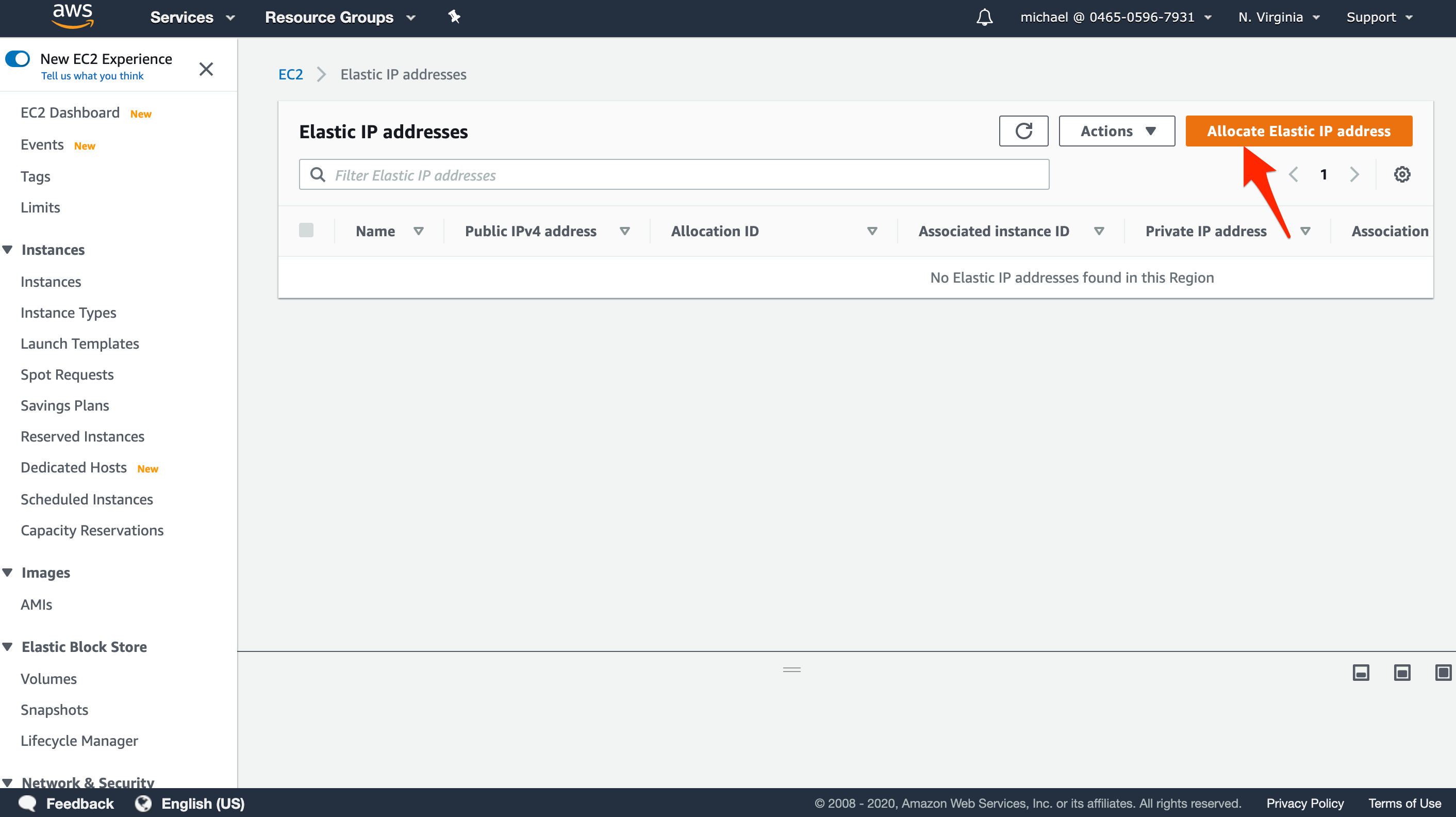
$ aws --version

Elastic IP

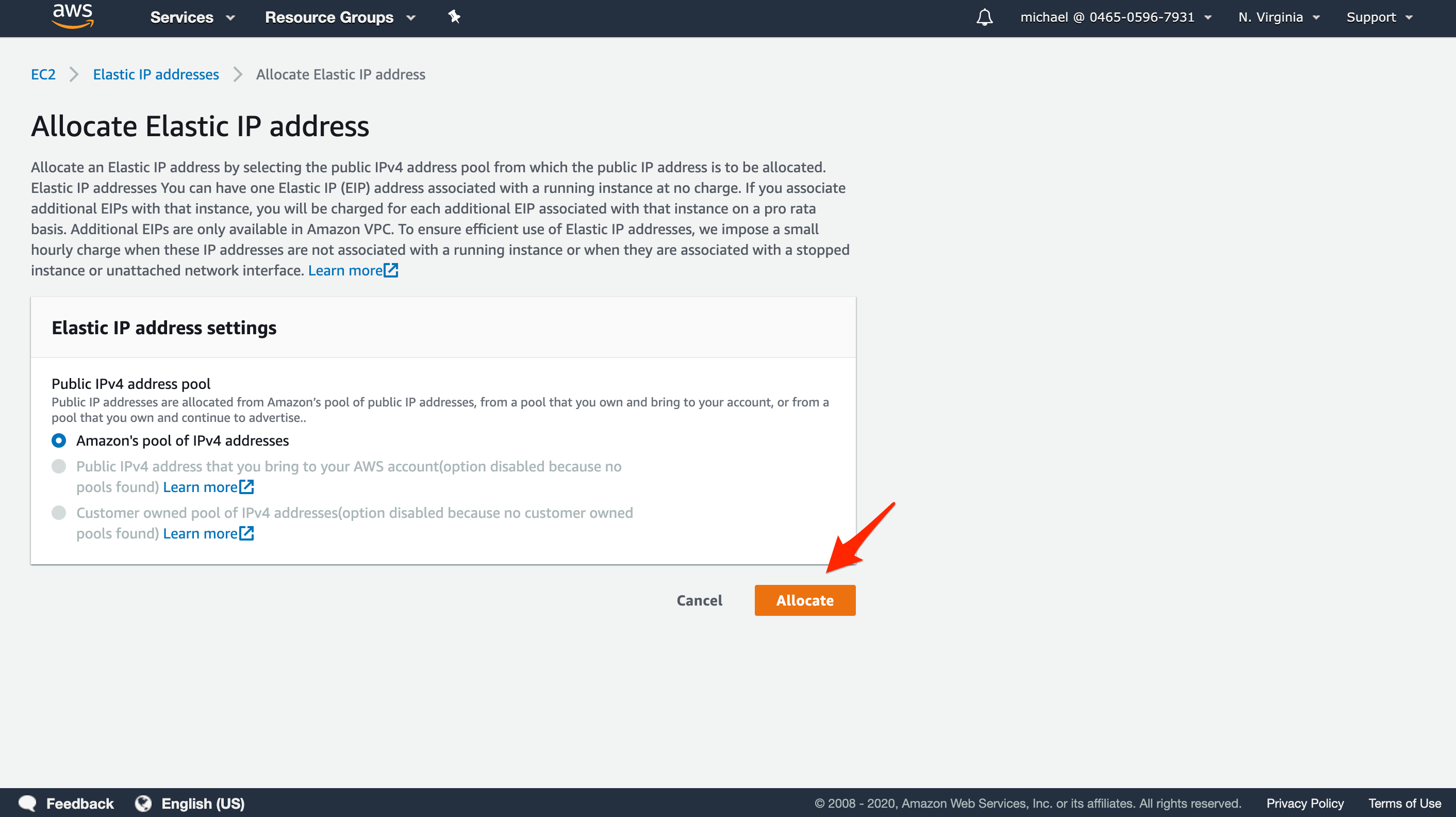
By default, instances receive new public IP address every time they start and re-start.

[Elastic IP](https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html) allows you to allocate static IPs for your EC2 instances, so the IP stays the same all the time and can be re-associated between instances. It's recommended to use one for your production setup.

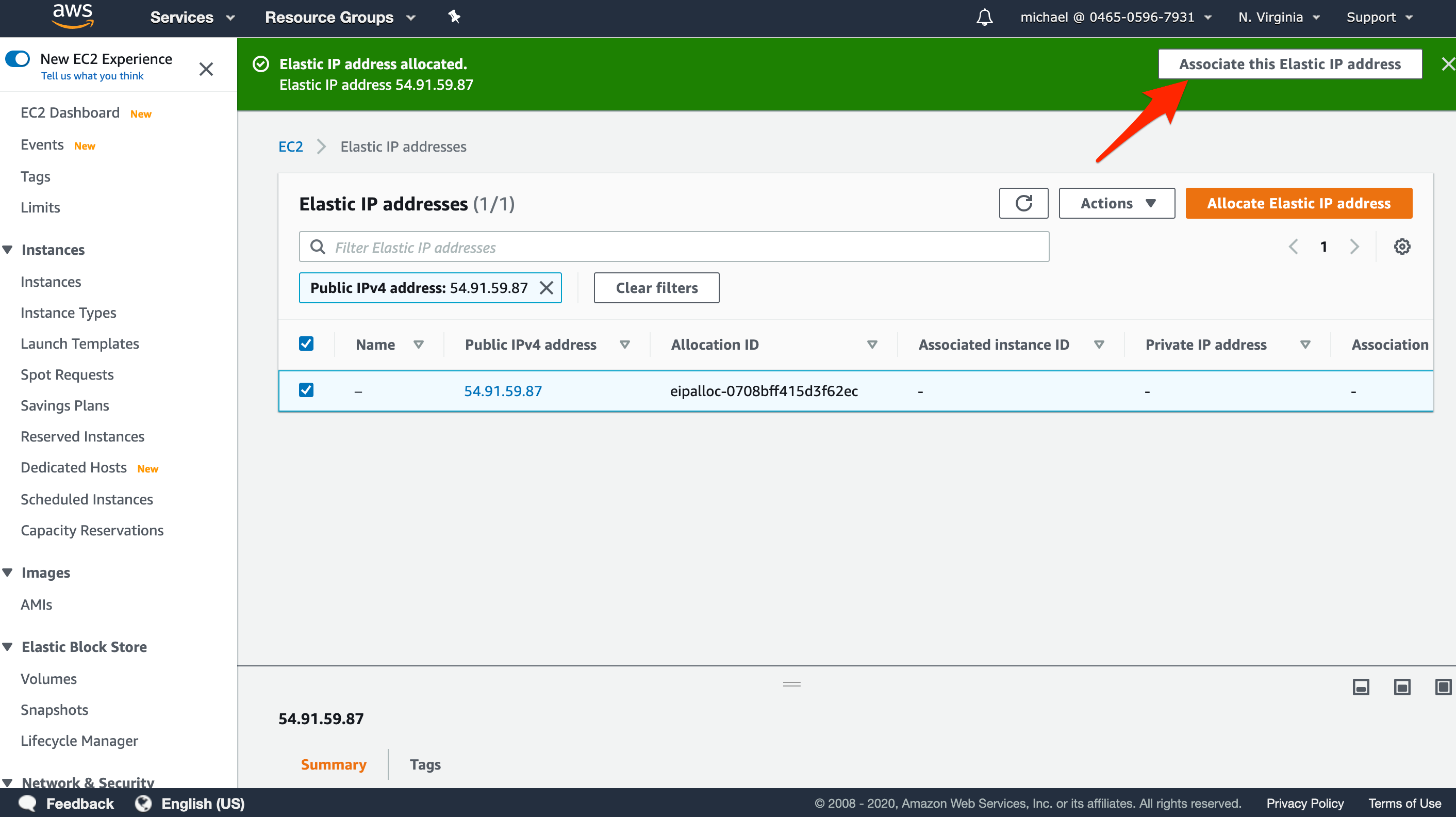
Navigate to [Elastic IPs](https://console.aws.amazon.com/ec2/v2/home?#Addresses) and click **Allocate Elastic IP address**:



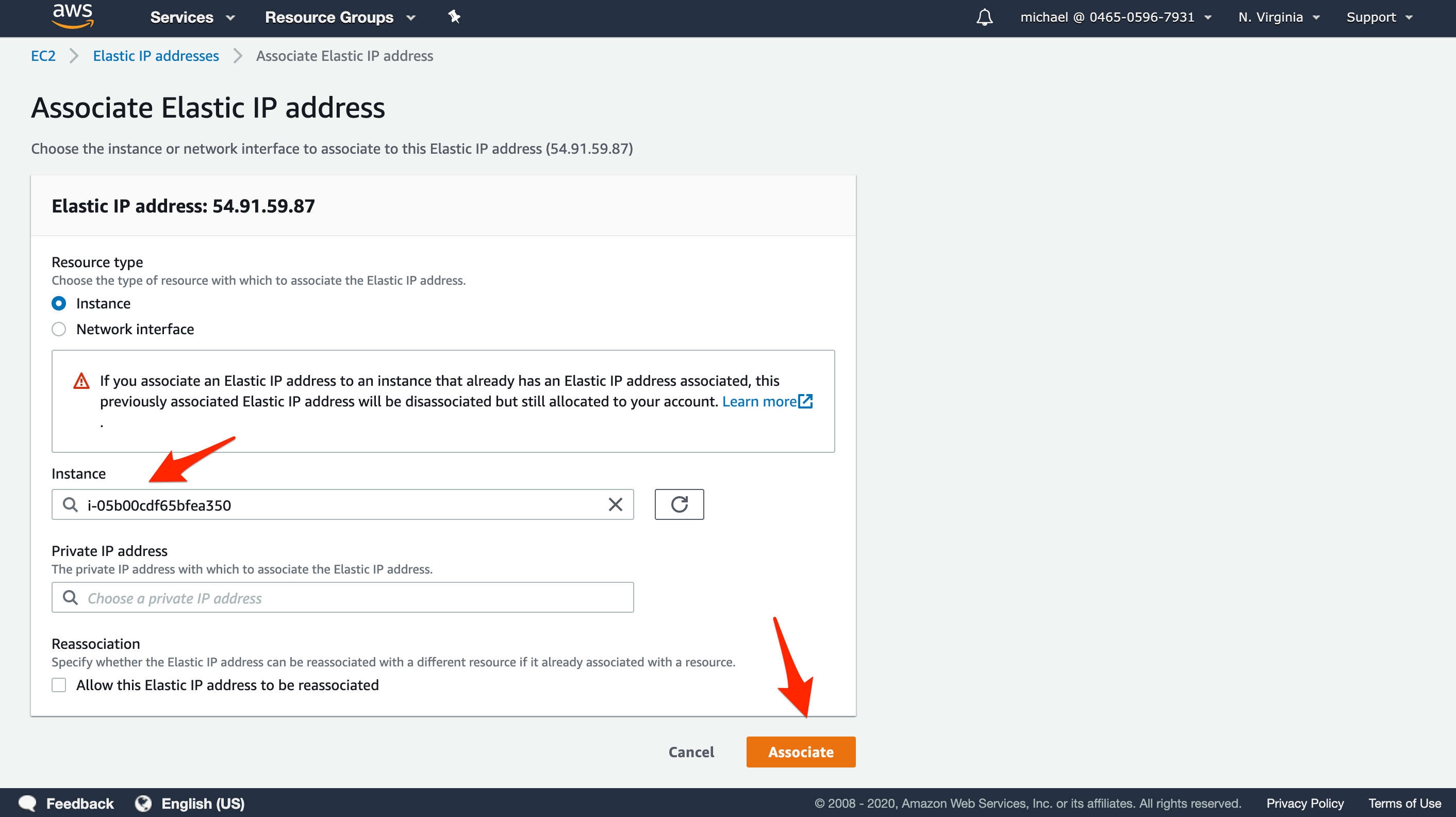
Then, click **Allocate**:



Click on **Associate this Elastic IP address**:



Select your instance and click **Associate**:

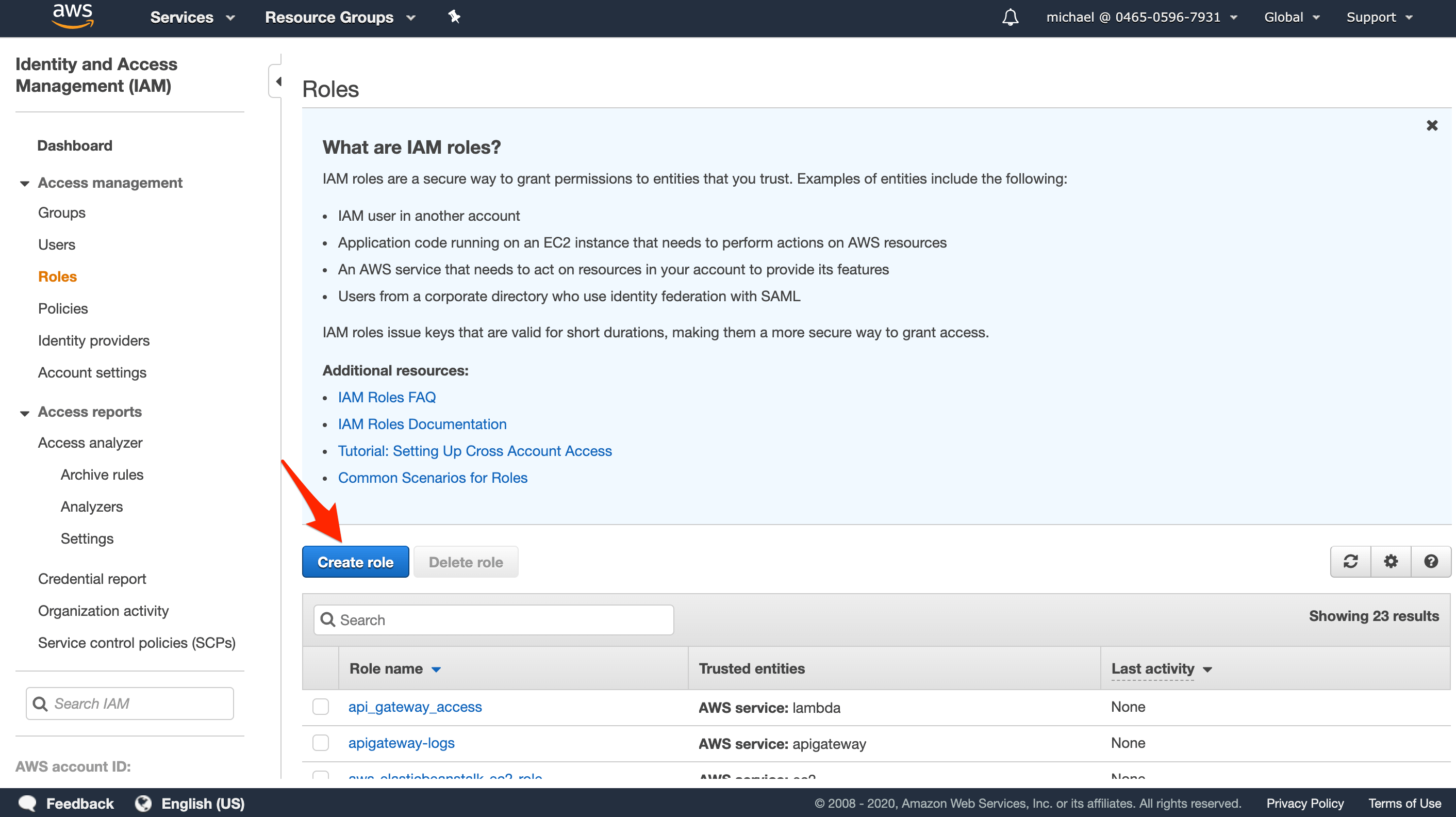


IAM Role

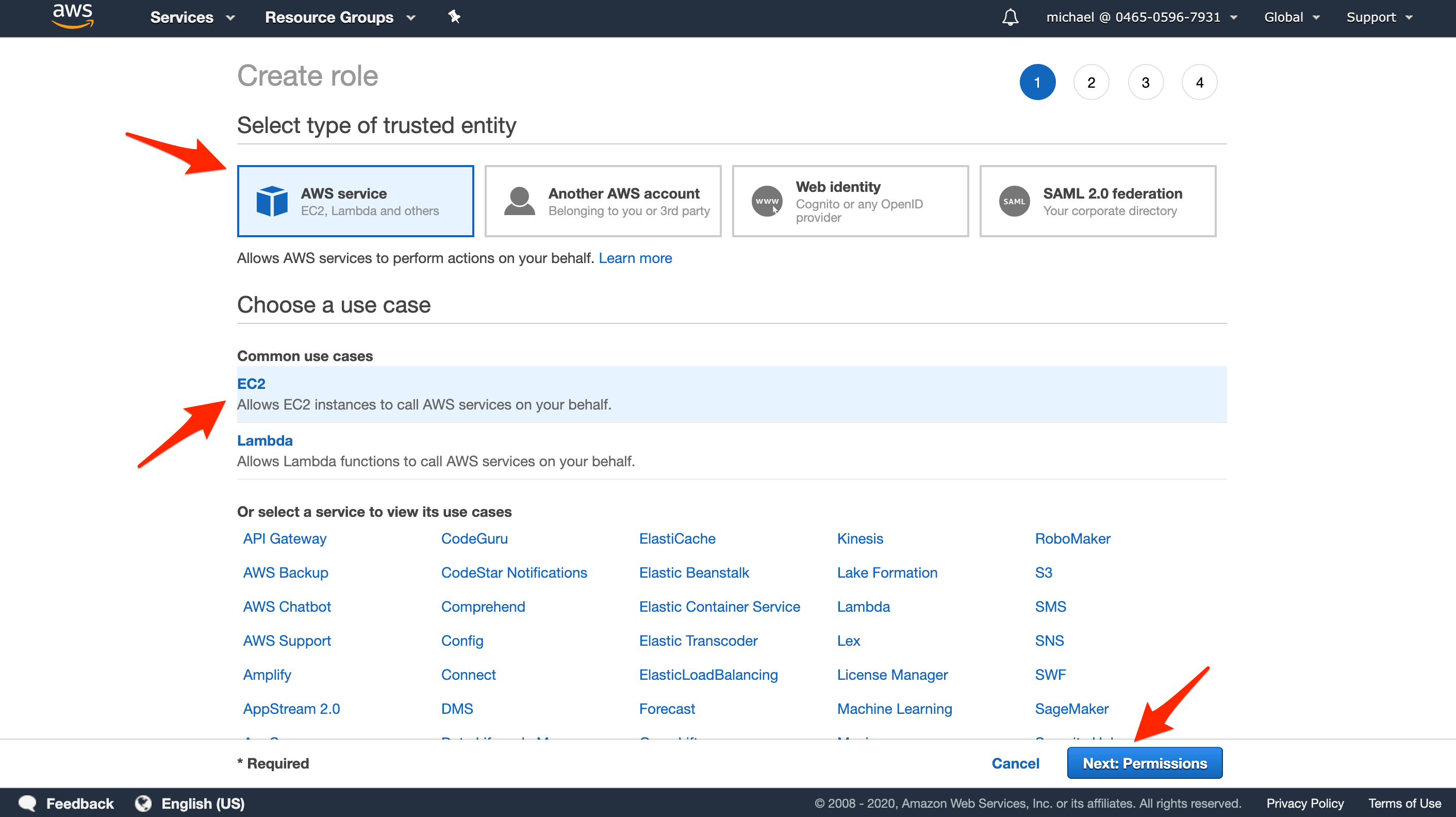
We'll be using AWS ECR to pull images from AWS ECR to our EC2 instance during deployment. Since we won't be allowing public access to the Docker image on ECR, you'll need to create an IAM role with permissions to pull Docker images from ECR and attach it to your EC2 instance.

Navigate to the [IAM console](https://console.aws.amazon.com/iam/home#/home).

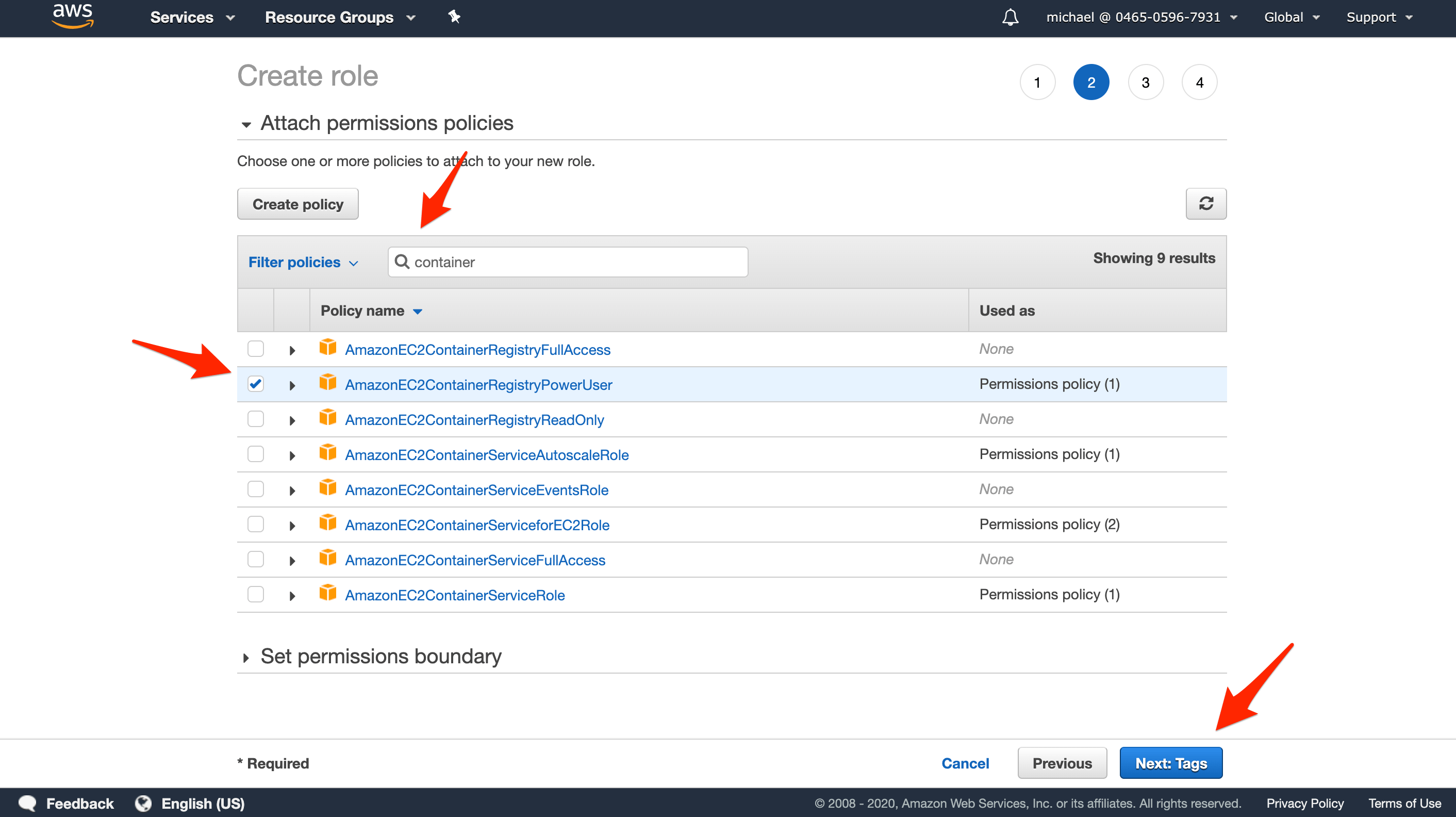
Click **Roles** in the left sidebar and then **Create role**:



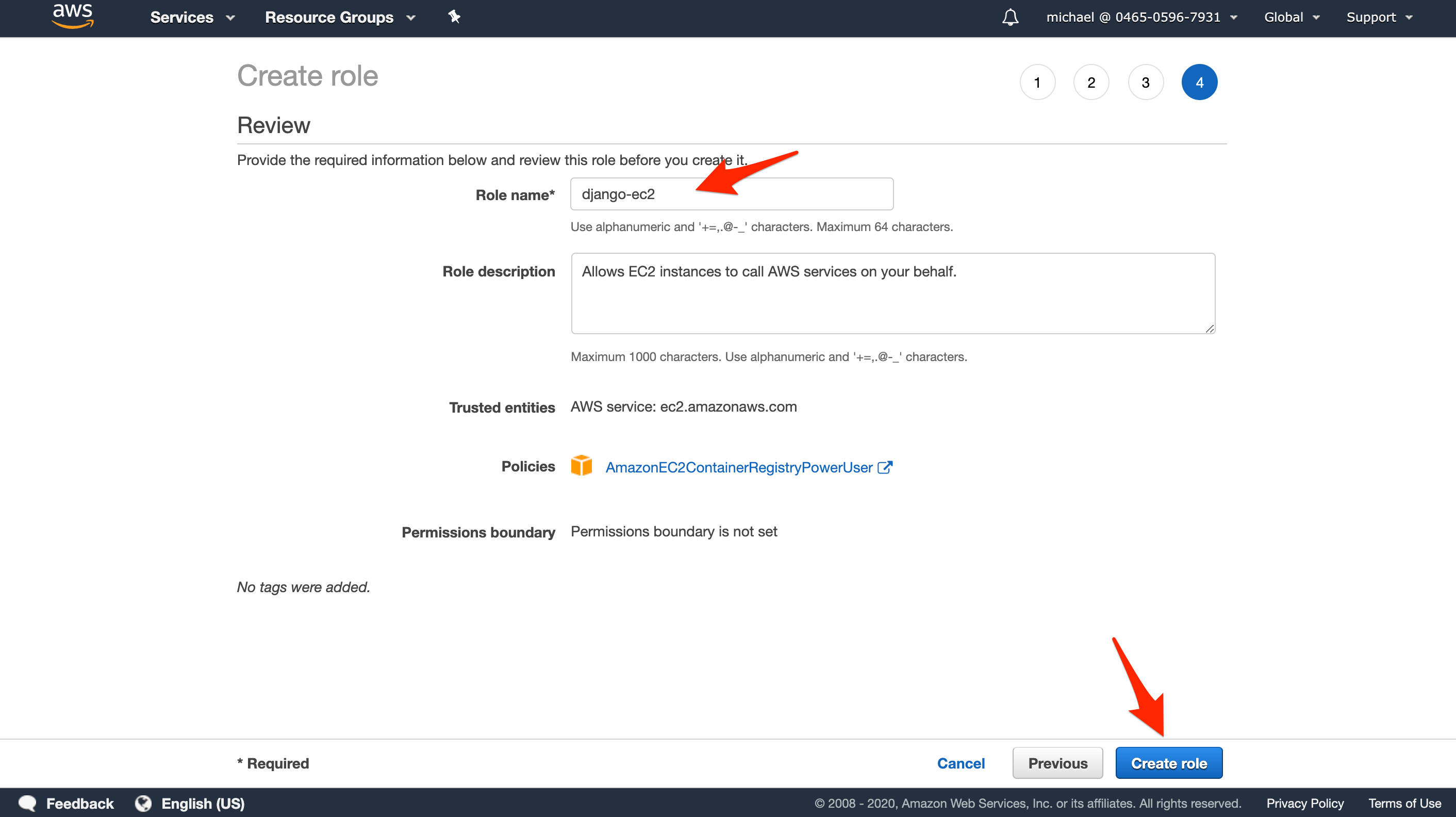
Select **AWS Service** and **EC2**, then click on **Next: Permissions**:



Enter container in the search box, select the **AmazonEC2ContainerRegistryPowerUser** policy, and click **Next: Tags**:

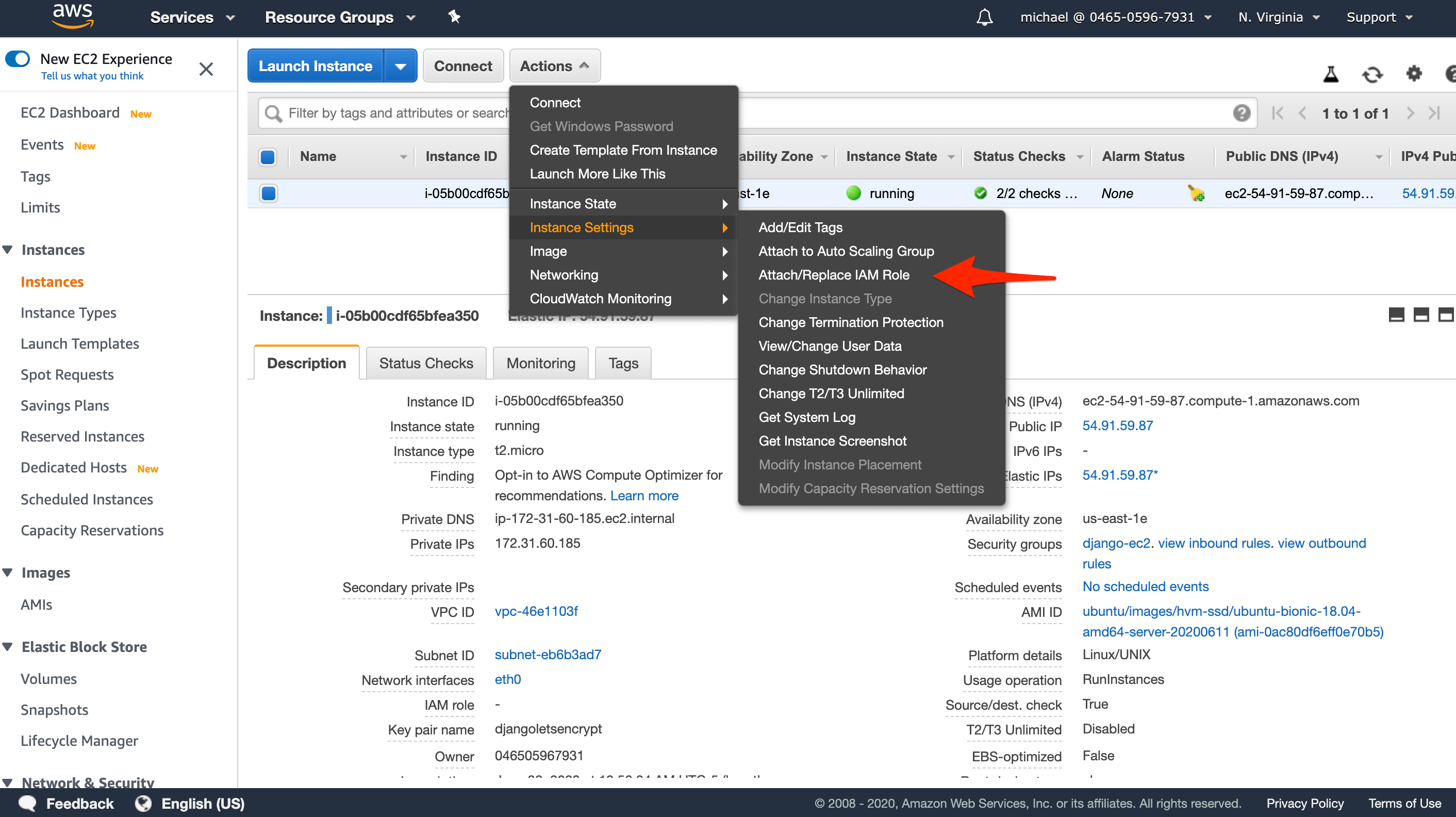


Click **Next: Review**. Use django-ec2 for the name, and click on **Create role**:

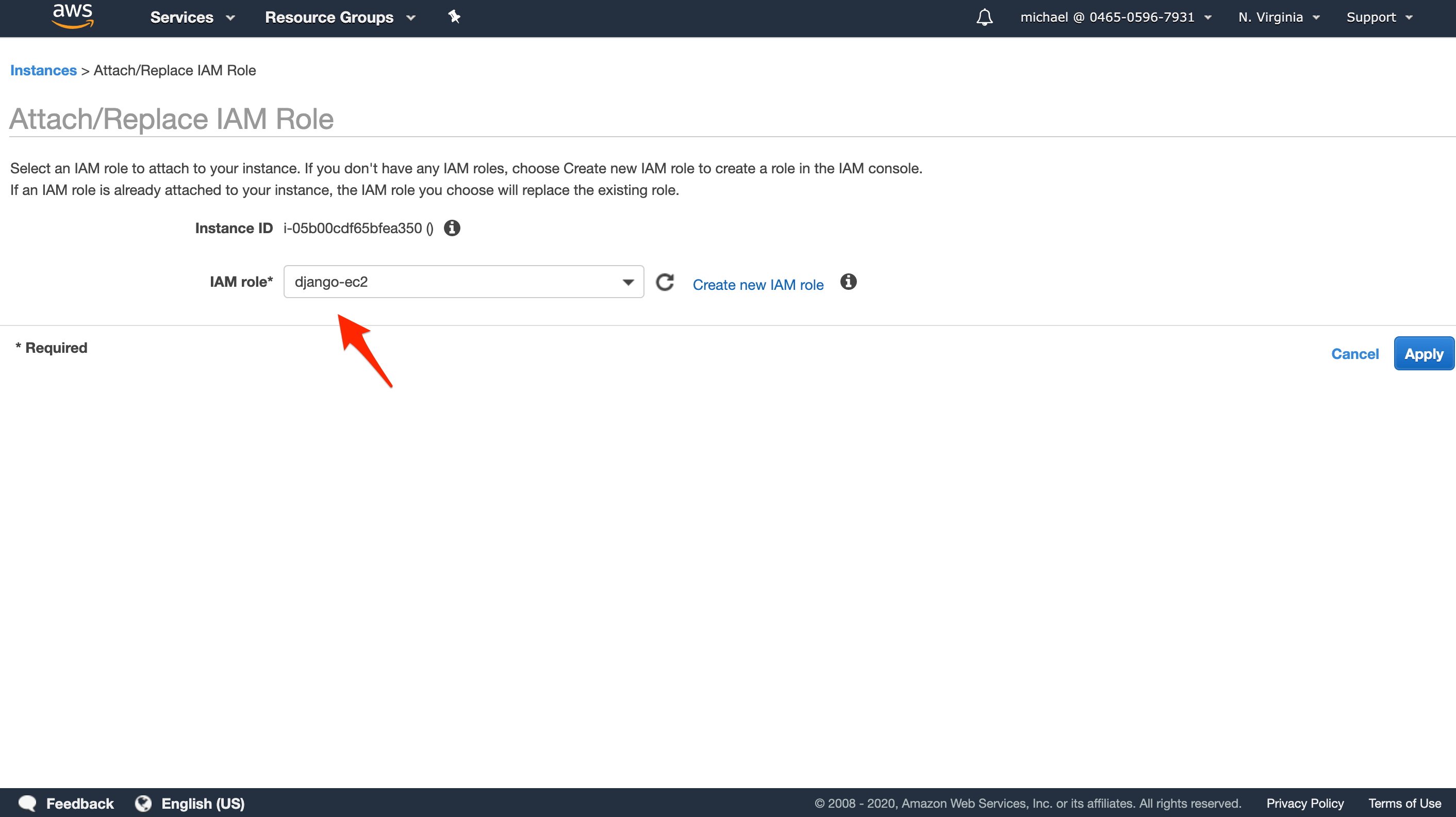


Now you need to attach the new role to your EC2 instance.

Back in the [EC2 console](https://console.aws.amazon.com/ec2/), click **Instances**, and then select your instance. Click the **Actions** dropdown -> **Instance settings** -> **Attach/Replace IAM Role**:



Select the **django-ec2** role, and then click **Apply**.



Click **Close**.

Add DNS Record

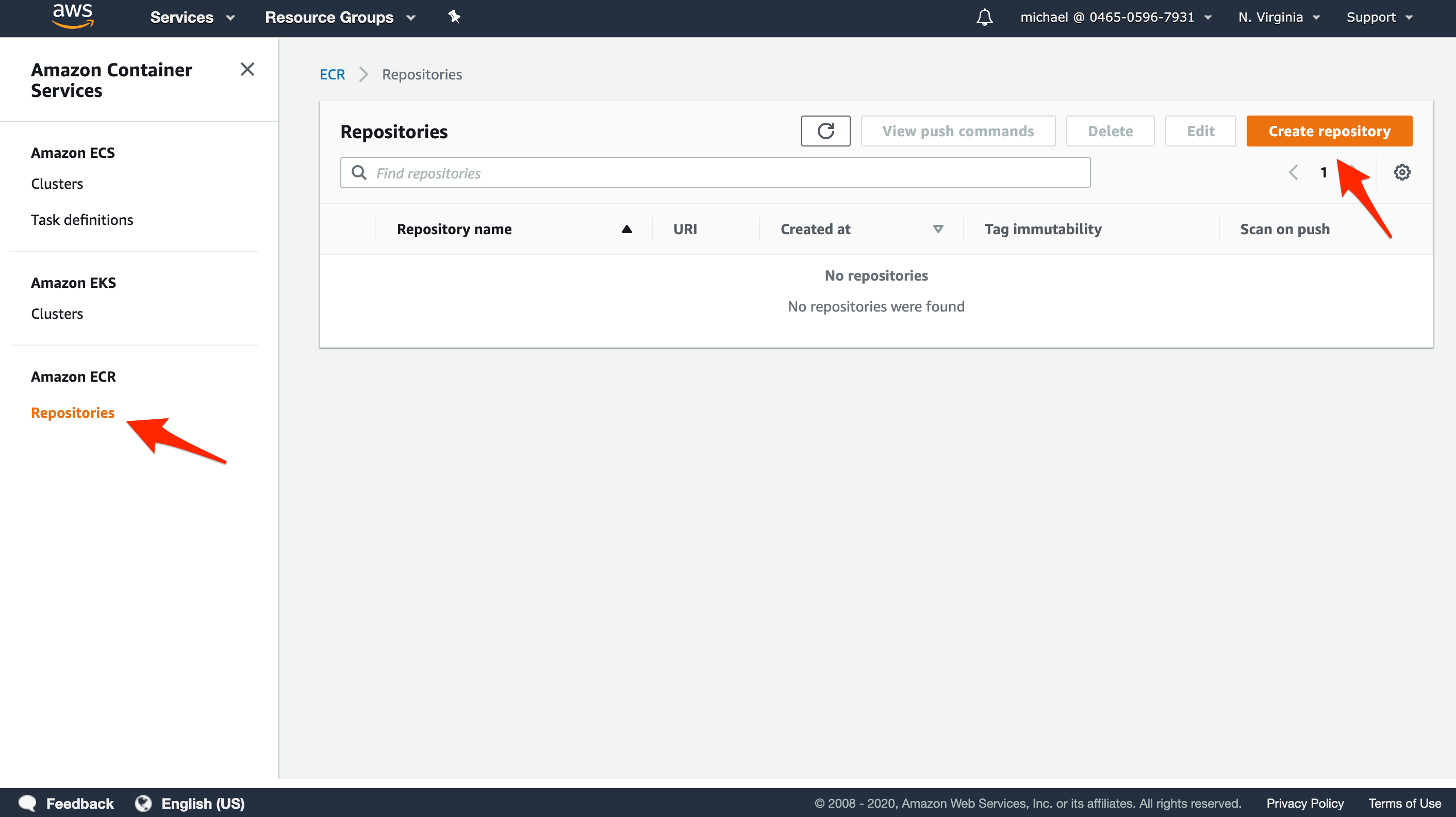
Add an A record to your DNS, for the domain that you are using, to point to your EC2 instance's public IP.

It's the Elastic IP you've associated to your instance.

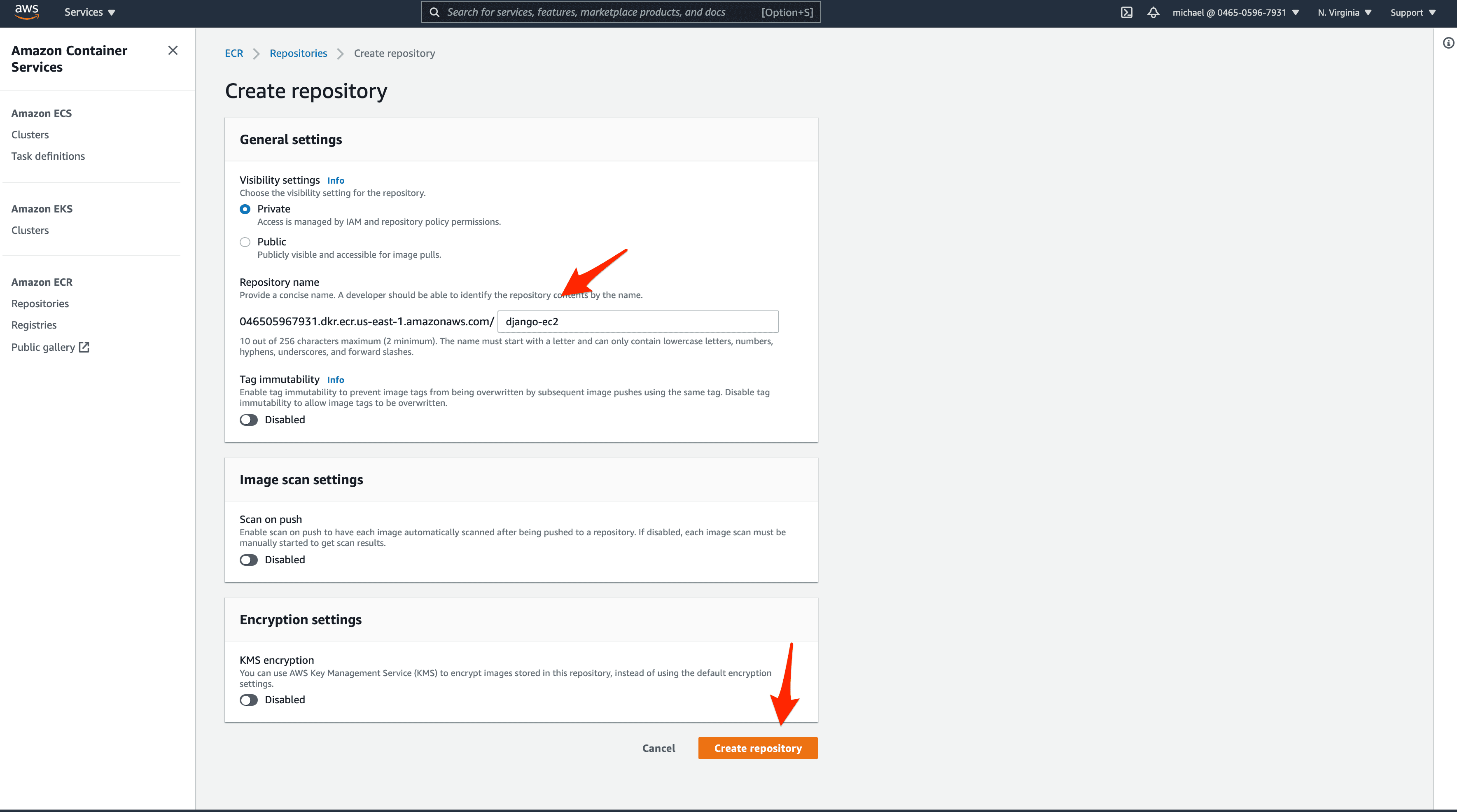
AWS ECR

[Amazon Elastic Container Registry](https://aws.amazon.com/ecr/) (ECR) is a fully-managed Docker image registry that makes it easy for developers to store and manage images. For private images, access is managed via IAM users and roles.

Navigate to the [ECR console](https://console.aws.amazon.com/ecr/). Click **Repositories** in the sidebar and then on **Create repository**:



Set the name to django-ec2 and click on **Create repository**:

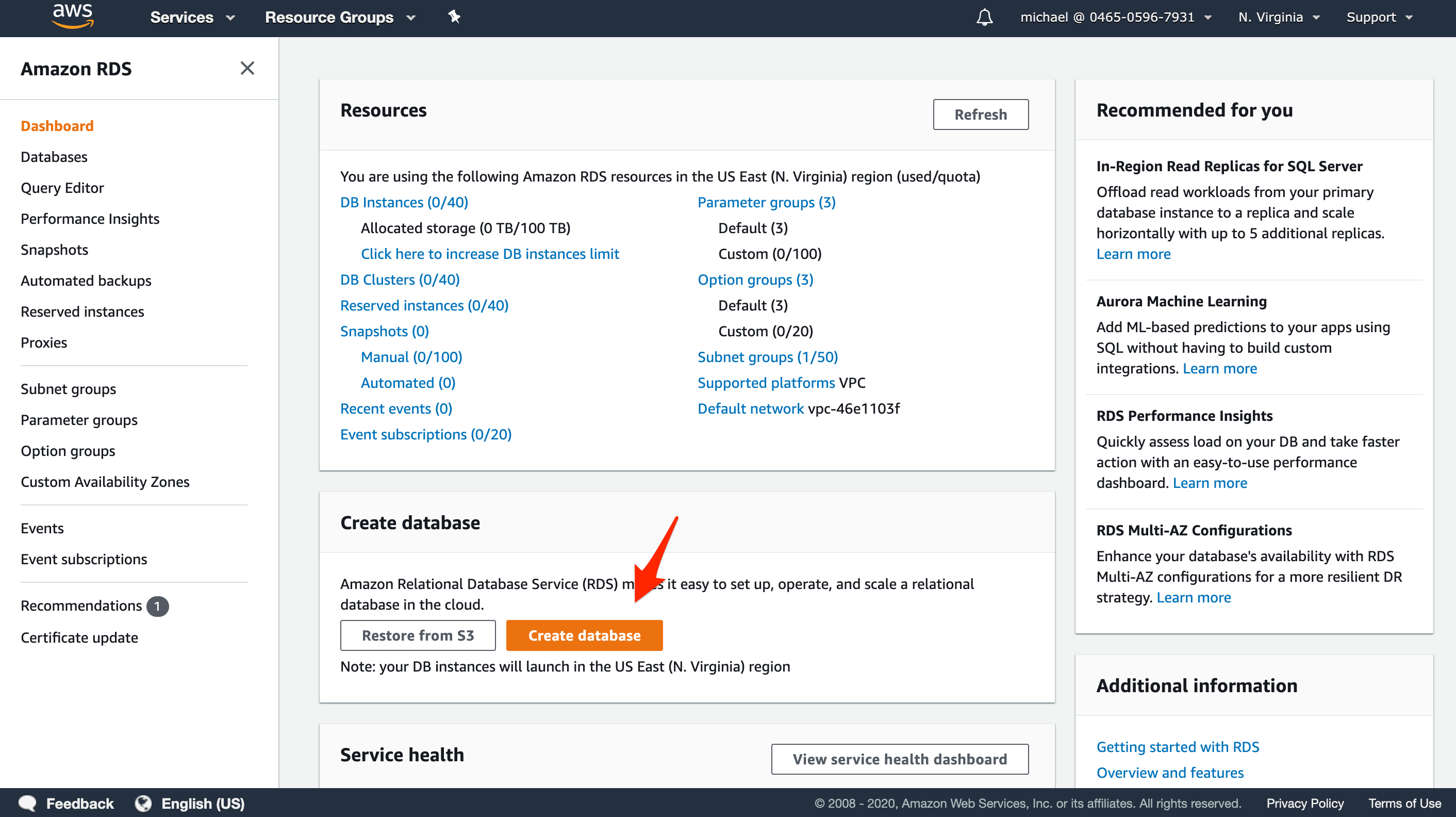


AWS RDS

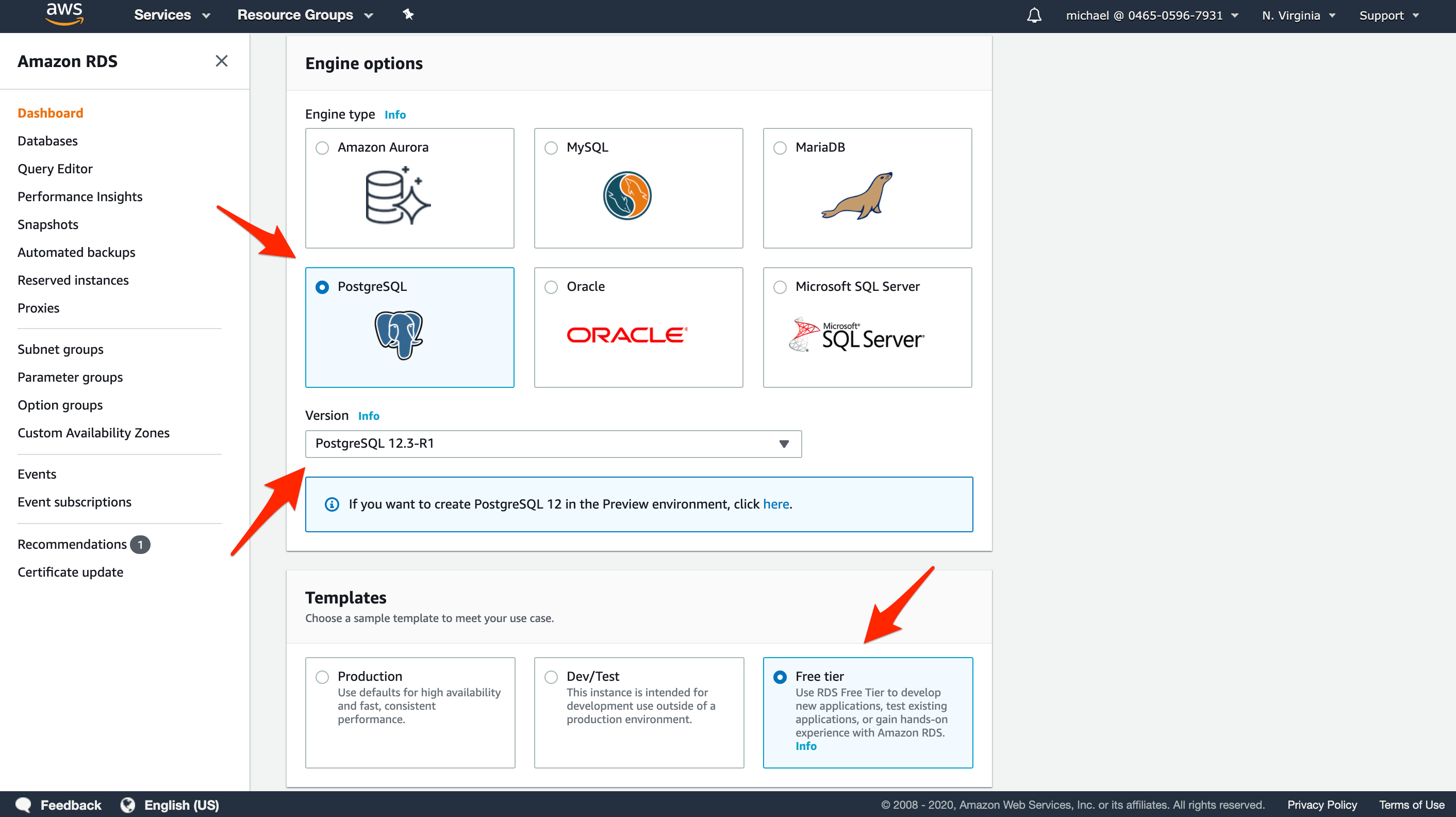
Now we can configure an RDS Postgres database.

While you can run your own Postgres database in a container, since databases are [critical services](https://vsupalov.com/database-in-docker/), adding additional layers, such us Docker, adds unnecessary risk in production. To simplify tasks such as minor version updates, regular backups, and scaling, it's recommended to use a managed service. So, we'll use [RDS](https://aws.amazon.com/rds/).

Navigate to the [RDS console](https://console.aws.amazon.com/rds/). Click on **Create database**:



Select the latest version of Postgres with the **Free tier** template:



Under **Settings**, set:

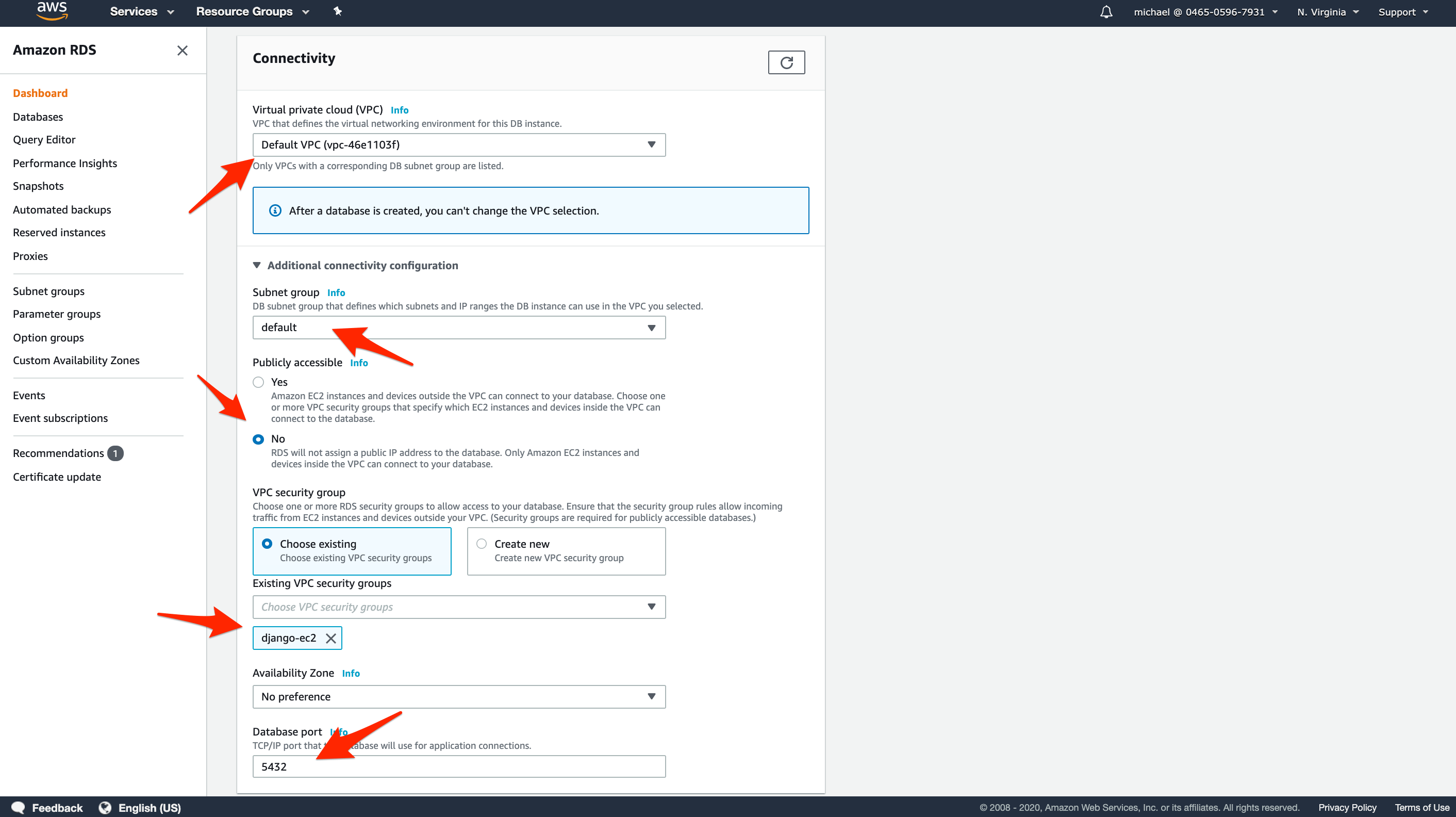
* DB Instance identifier: djangoec2
* Master username: webapp
* Select **Auto generate a password**

Stick with default settings for:

* DB instance size
* Storage
* Availability & durability

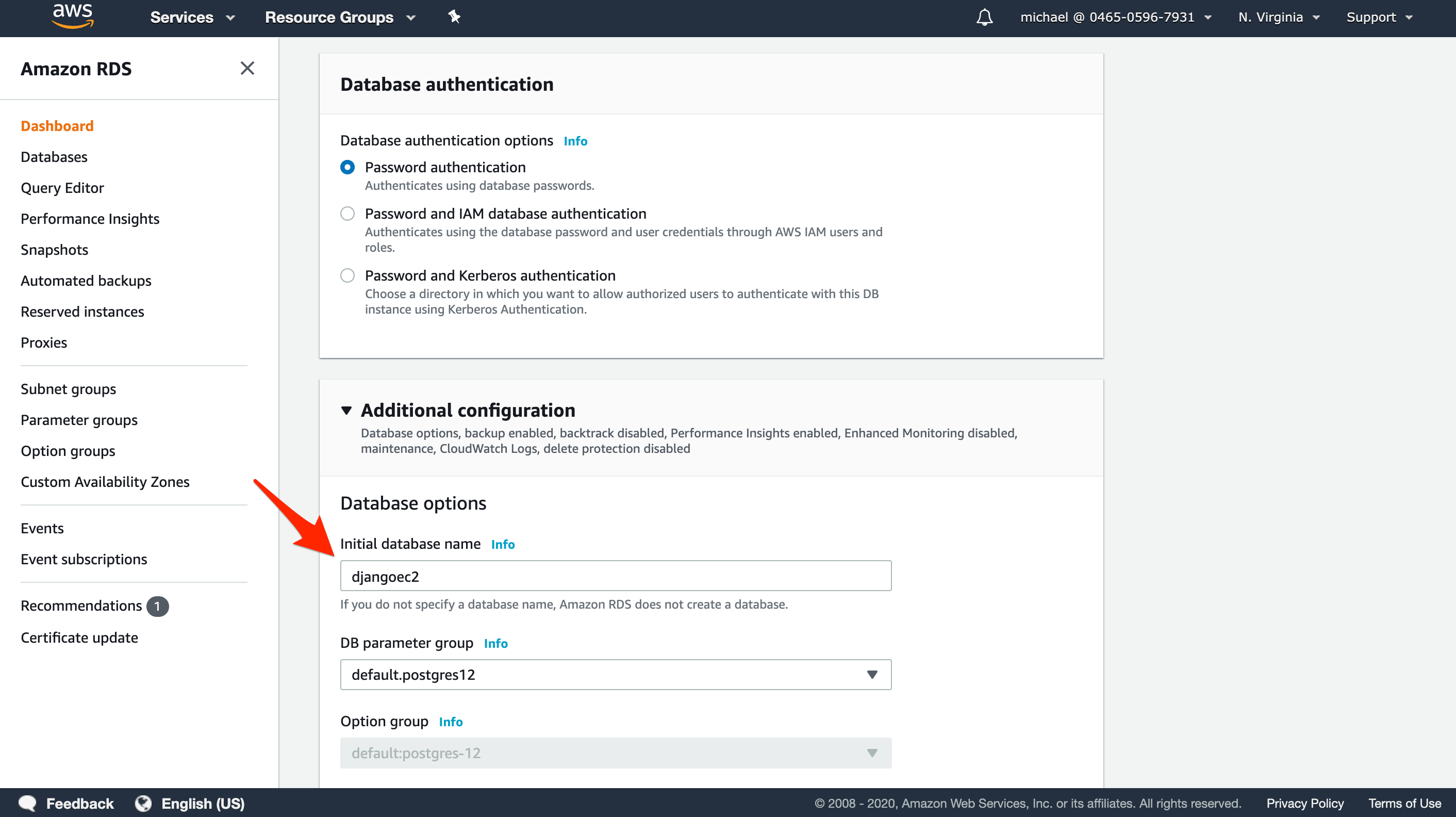
Skip down to the **Connectivity** section and set the following:

* Virtual private cloud (VPC): Default
* Subnet group: default
* Publicly accessible: No
* VPC security group: django-ec2
* Database port: 5432



Leave **Database authentication** as it is.

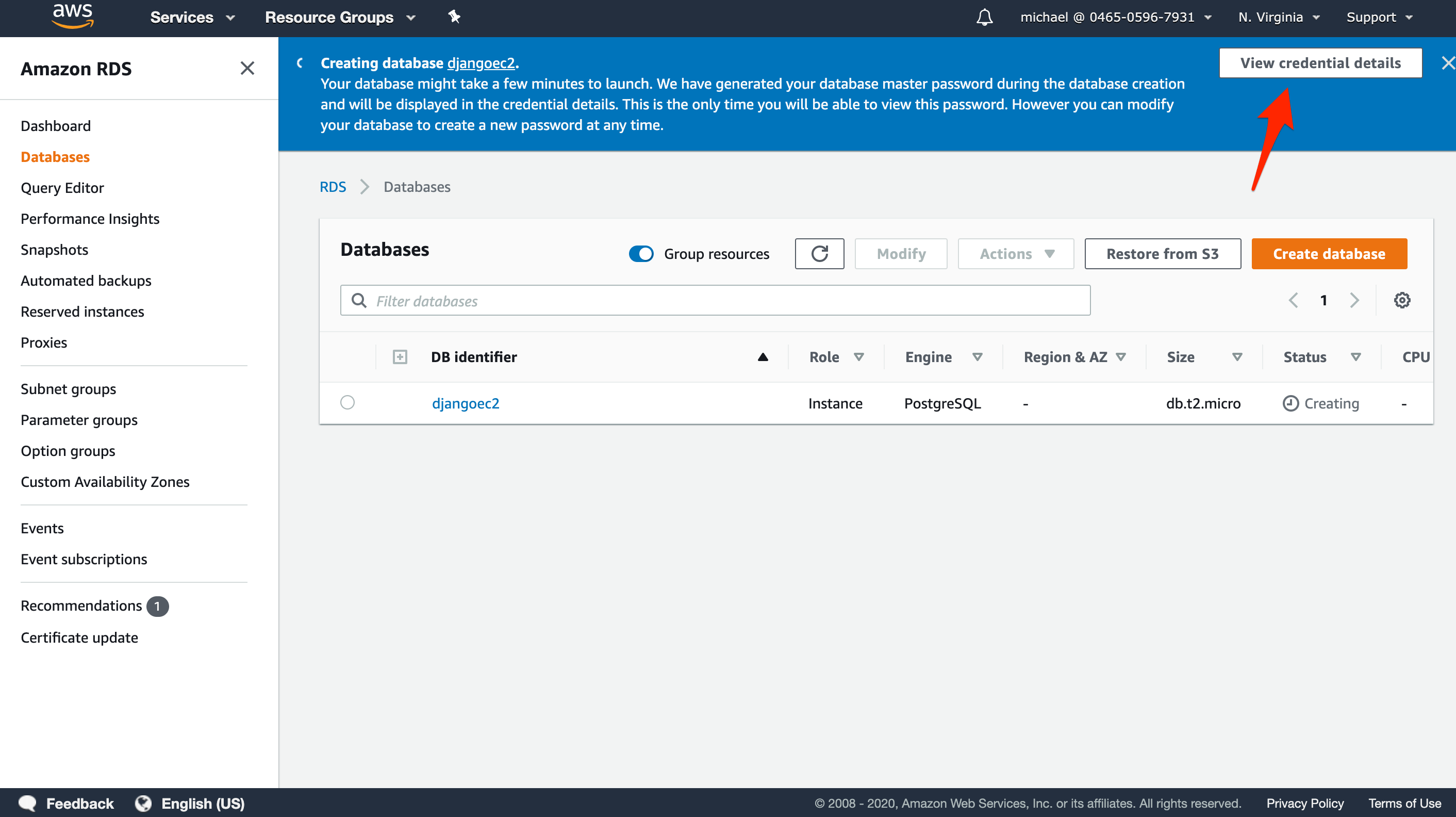
Open **Additional configuration** and change **Initial database name** to djangoec2:



Leave the other settings as they are.

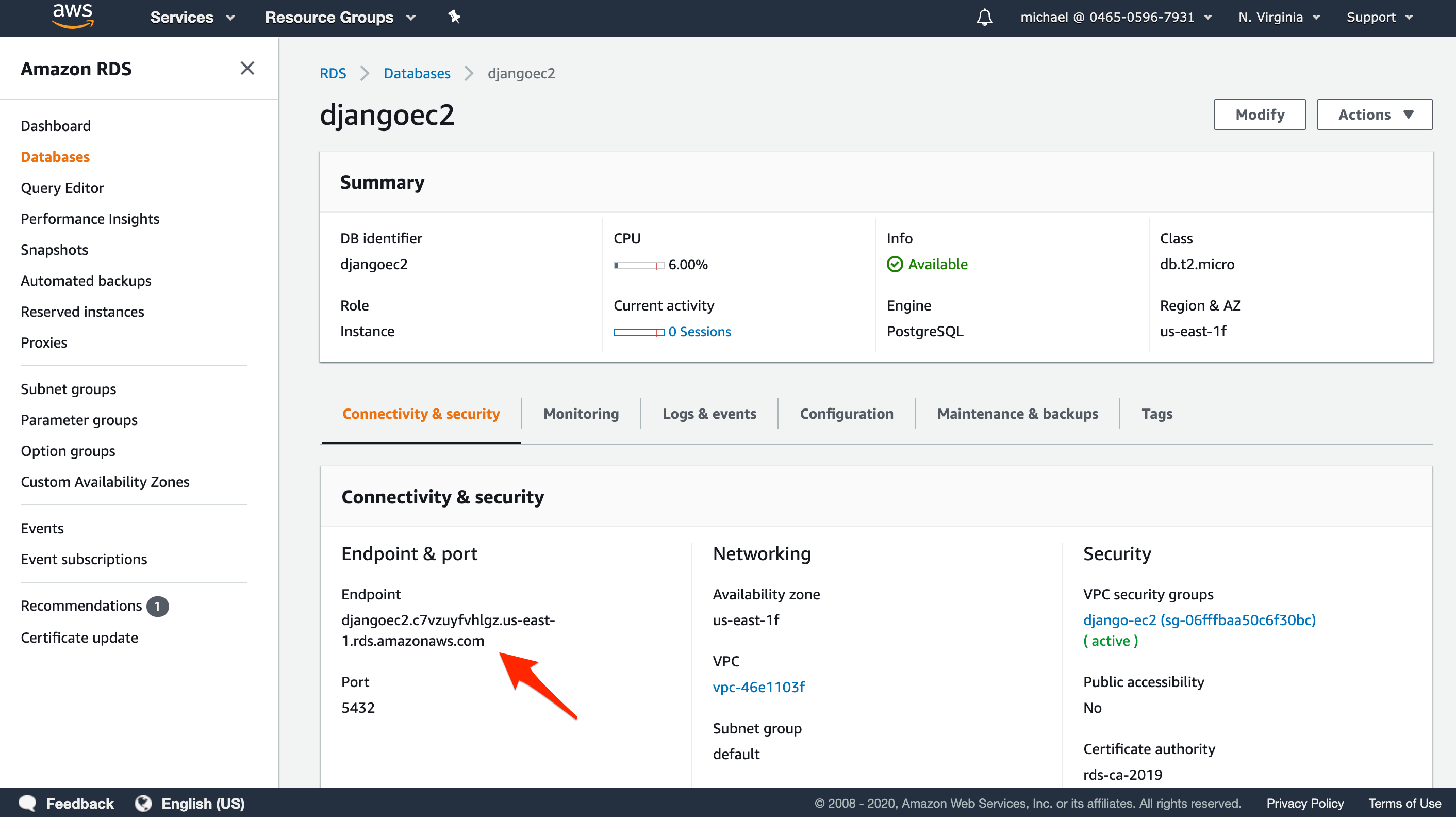
Finally, click **Create database**.

Click on **View credentials details** to see the generated password for the **webapp** user:



Store this password somewhere safe. You'll need to provide it to the Django application here shortly.

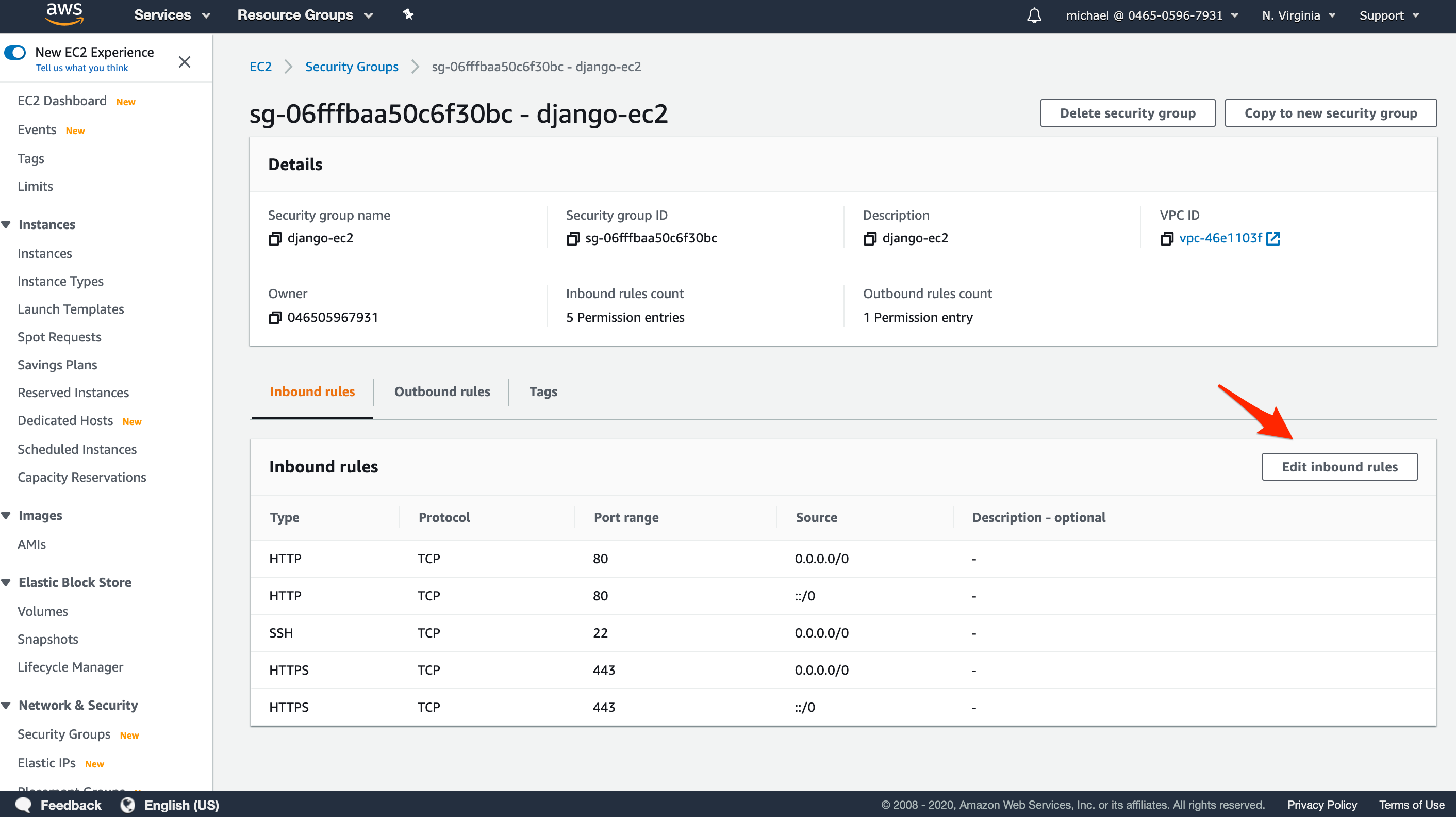
It will take a few of minutes for the instance to spin up. Once up, click on the DB Identifier of the newly created database to see its details. Take note of the database endpoint; you'll need to set it in your Django app.



AWS Security Group

Within the [EC2 console](https://console.aws.amazon.com/ec2/), click **Security Groups** in the sidebar. Find and click the ID of the **django-ec2** group to edit its details.

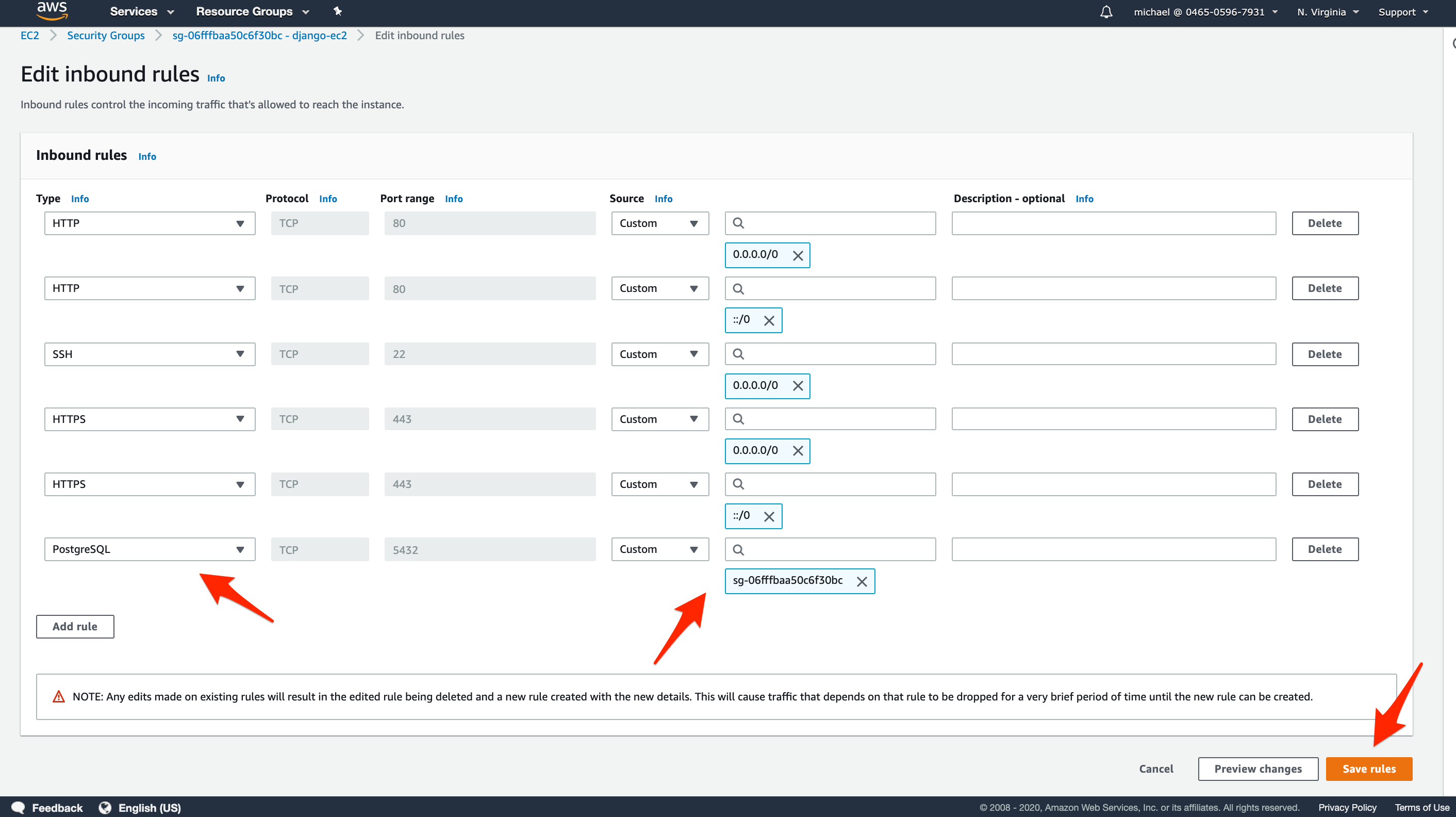
Click on **Edit inbound rules**:



Add an inbound rule that will allow Postgres connections to instances inside that Security Group. To do that:

* click on **Add rule**
* select **PostgreSQL** for the rule type
* select **Custom** for the rule source
* click into the search field and select the **django-ec2** Security Group
* click on **Save rules**

To limit access to your database, only connections from instances inside the same Security Group are allowed. Our application can connect because we set the same Security Group, *django-ec2*, for both the RDS and EC2 instances. Instances inside other Security Groups are therefore not allowed to connect.



Project Config

With the AWS infrastructure set up, we now need to configure our Django project locally before deploying it.

First, clone down the contents from the GitHub project repo:

$ git clone https://github.com/testdrivenio/django-on-docker-letsencrypt django-on-docker-letsencrypt-aws

$ cd django-on-docker-letsencrypt-aws

This repository contains everything that you need to deploy a Dockerized Django with Let's Encrypt HTTPS certificates.

Docker Compose

When the app is deployed for the first time, you should follow these two steps to avoid issues with certificates:

1. Start by issuing the certificates from Let's Encrypt's staging environment
2. Then, when all is running as expected, switch to Let's Encrypt's production environment

You can read more about Let's Encrypt's limitations on production environments in the [Let's Encrypt](https://testdriven.io/blog/django-lets-encrypt/#lets-encrypt) section from the previous post, [Securing a Containerized Django Application with Let's Encrypt](https://testdriven.io/blog/django-lets-encrypt/).

For testing, update the *docker-compose.staging.yml.* file like so:

**version**: '3.8'

**services**:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**image**: <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com/django-ec2:web

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

**expose**:

- 8000

**env\_file**:

- ./.env.staging

**nginx-proxy**:

**container\_name**: nginx-proxy

**build**: nginx

**image**: <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com/django-ec2:nginx-proxy

**restart**: always

**ports**:

- 443:443

- 80:80

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- /var/run/docker.sock:/tmp/docker.sock:ro

**depends\_on**:

- web

**nginx-proxy-letsencrypt**:

**image**: jrcs/letsencrypt-nginx-proxy-companion

**env\_file**:

- ./.env.staging.proxy-companion

**volumes**:

- /var/run/docker.sock:/var/run/docker.sock:ro

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- acme:/etc/acme.sh

**depends\_on**:

- nginx-proxy

**volumes**:

**static\_volume**:

**media\_volume**:

**certs**:

**html**:

**vhost**:

**acme**:

For the web and nginx-proxy services, update the image properties to use images from ECR (which we'll add shortly).

Examples:

**image**: 123456789.dkr.ecr.us-east-1.amazonaws.com/django-ec2:web

**image**: 123456789.dkr.ecr.us-east-1.amazonaws.com/django-ec2:nginx-proxy

The values consist of the repository URL (123456789.dkr.ecr.us-east-1.amazonaws.com) along with the image name (django-ec2) and tags (web and nginx-proxy).

To keep things simple we're using a single registry to store both images. We used the web and nginx-proxy to differentiate between the two. Ideally, you should use two registries: one for web and one for nginx-proxy. Update this on your own if you'd like it.

Other than the image properties, we also removed the db service (and related volume) since we're using RDS rather than managing Postgres in a container.

Environments

It's time to set up the environment files for the web and nginx-proxy-letsencrypt containers.

First, add an *.env.staging* file for the web container:

DEBUG=0

SECRET\_KEY=change\_me

DJANGO\_ALLOWED\_HOSTS=<YOUR\_DOMAIN.COM>

SQL\_ENGINE=django.db.backends.postgresql

SQL\_DATABASE=djangoec2

SQL\_USER=webapp

SQL\_PASSWORD=<PASSWORD-FROM-AWS-RDS>

SQL\_HOST=<DATABASE-ENDPOINT-FROM-AWS-RDS>

SQL\_PORT=5432

DATABASE=postgres

VIRTUAL\_HOST=<YOUR\_DOMAIN.COM>

VIRTUAL\_PORT=8000

LETSENCRYPT\_HOST=<YOUR\_DOMAIN.COM>

Notes:

1. Change <YOUR\_DOMAIN.COM> to your actual domain.
2. Change SQL\_PASSWORD and SQL\_HOST to match those created in the RDS section.
3. Change SECRET\_KEY to some long random string.
4. The VIRTUAL\_HOST and VIRTUAL\_PORT are needed by nginx-proxy container to auto create the reverse proxy configuration.
5. LETSENCRYPT\_HOST is there so the nginx-proxy-companion can issue Let's Encrypt certificate for your domain.

For testing/debugging purposes you may want to use a \* for DJANGO\_ALLOWED\_HOSTS the first time you deploy to simplify things. Just don't forget to limit the allowed hosts once testing is complete.

Second, add an *.env.staging.proxy-companion* file, making sure to update the DEFAULT\_EMAIL value:

DEFAULT\_EMAIL=youremail@yourdomain.com

ACME\_CA\_URI=https://acme-staging-v02.api.letsencrypt.org/directory

NGINX\_PROXY\_CONTAINER=nginx-proxy

Review the [Let's Encrypt Nginx Proxy Companion Service](https://testdriven.io/blog/django-lets-encrypt/#lets-encrypt-nginx-proxy-companion-service) section from the [Securing a Containerized Django Application with Let's Encrypt](https://testdriven.io/blog/django-lets-encrypt/) post to learn more about the above mentioned environment variables.

Build and Push Docker Images

Now we're ready to build the Docker images:

$ docker-compose -f docker-compose.staging.yml build

It may take a few minutes to build. Once done, we're ready to push the images up to ECR.

First, assuming that you have the [awscli installed](https://docs.aws.amazon.com/cli/latest/userguide/install-cliv2.html) and that you've set your [AWS credentials](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-configure.html), log in to the ECR Docker repository:

$ aws ecr get-login-password --region <aws-region> | docker login --username AWS --password-stdin <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com

*# aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin 123456789.dkr.ecr.us-east-1.amazonaws.com*

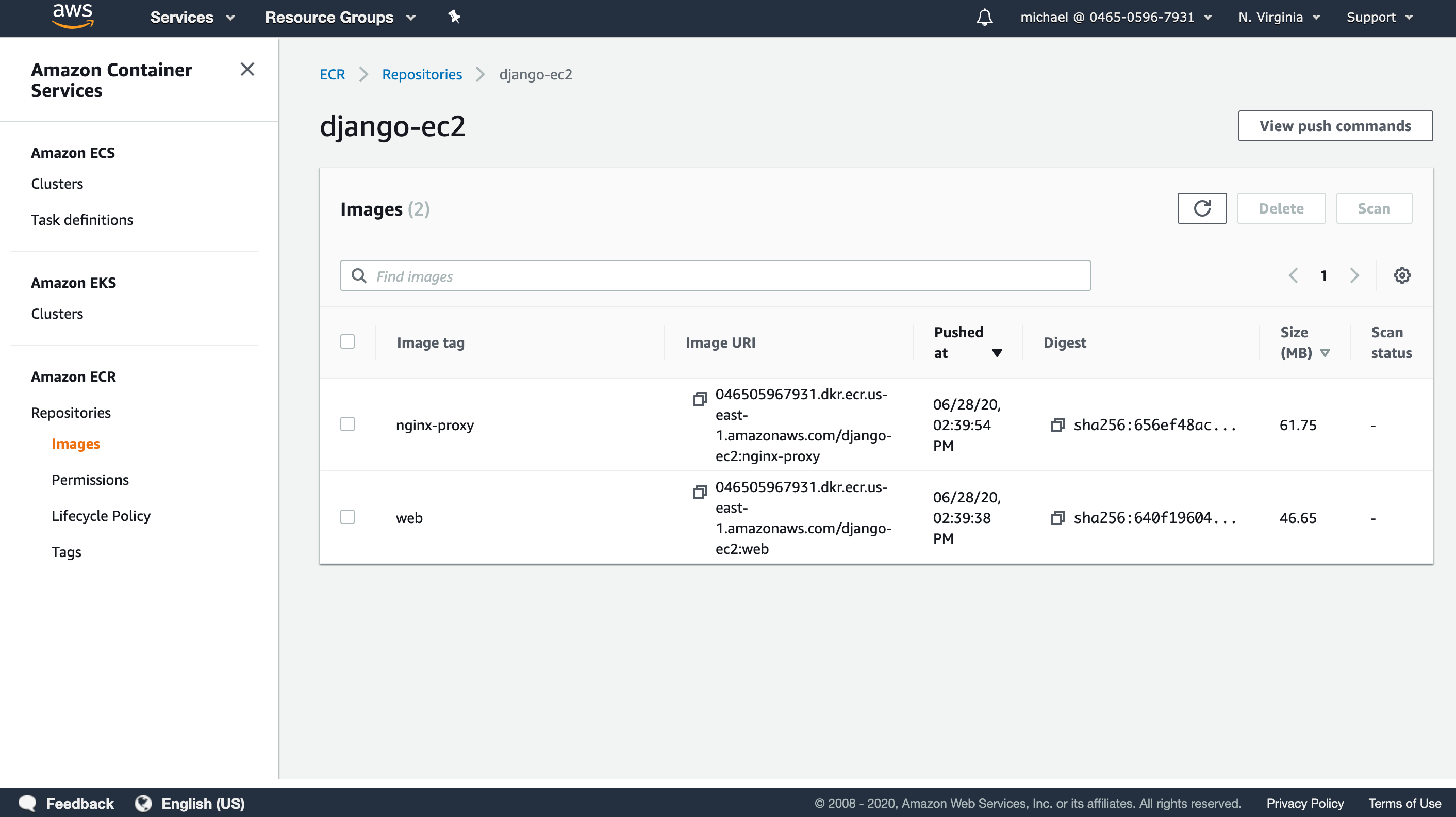
You should see:

Login Succeeded

Then push the images to ECR:

$ docker-compose -f docker-compose.staging.yml push

Open your django-ec2 ECR repository to see the pushed images:



Running the Containers

Everything is set up for deployment.

It's time to move to your EC2 instance.

Assuming you have a project directory created on your instance, like */home/ubuntu/django-on-docker*, copy the files and folders over with SCP:

$ scp -i /path/to/your/djangoletsencrypt.pem **\**

-r **$(**pwd**)**/{app,nginx,.env.staging,.env.staging.proxy-companion,docker-compose.staging.yml} **\**

ubuntu@public-ip-or-domain-of-ec2-instance:/path/to/django-on-docker

Next, connect to your instance via SSH and move to the project directory:

$ ssh -i /path/to/your/djangoletsencrypt.pem ubuntu@public-ip-or-domain-of-ec2-instance

$ cd /path/to/django-on-docker

Login to ECR Docker repository.

$ aws ecr get-login-password --region <aws-region> | docker login --username AWS --password-stdin <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com

Pull the images:

$ docker pull <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com/django-ec2:web

$ docker pull <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com/django-ec2:nginx-proxy

With that, you're ready to spin up the containers:

$ docker-compose -f docker-compose.staging.yml up -d

Once the containers are up and running, navigate to your domain in your browser. You should see something like:

Graphical user interface, text, application, email

Description automatically generated

This is expected. This screen is shown because the certificate was issued from a [staging environment](https://letsencrypt.org/docs/staging-environment/).

How do you know if everything works?

Click on "Advanced" and then on "Proceed". You should now see your app. Upload an image, and then make sure you can view the image at https://yourdomain.com/media/IMAGE\_FILE\_NAME.

Issue the Production Certificate

Now, that everything works as expected, we can switch over to Let's Encrypt's production environment.

Bring down the existing containers and exit your instance:

$ docker-compose -f docker-compose.staging.yml down -v

$ exit

Back on your local machine, open *docker-compose.prod.yml* and make the same changes that you did for the staging version:

1. Update the ìmage properties to match your AWS ECR URLs for the ẁeb and nginx-proxy services
2. Remove the db service along with the related volume

**version**: '3.8'

**services**:

**web**:

**build**:

**context**: ./app

**dockerfile**: Dockerfile.prod

**image**: 046505967931.dkr.ecr.us-east-1.amazonaws.com/django-ec2:web

**command**: gunicorn hello\_django.wsgi:application --bind 0.0.0.0:8000

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

**expose**:

- 8000

**env\_file**:

- ./.env.prod

**nginx-proxy**:

**container\_name**: nginx-proxy

**build**: nginx

**image**: 046505967931.dkr.ecr.us-east-1.amazonaws.com/django-ec2:nginx-proxy

**restart**: always

**ports**:

- 443:443

- 80:80

**volumes**:

- static\_volume:/home/app/web/staticfiles

- media\_volume:/home/app/web/mediafiles

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- /var/run/docker.sock:/tmp/docker.sock:ro

**depends\_on**:

- web

**nginx-proxy-letsencrypt**:

**image**: jrcs/letsencrypt-nginx-proxy-companion

**env\_file**:

- ./.env.prod.proxy-companion

**volumes**:

- /var/run/docker.sock:/var/run/docker.sock:ro

- certs:/etc/nginx/certs

- html:/usr/share/nginx/html

- vhost:/etc/nginx/vhost.d

- acme:/etc/acme.sh

**depends\_on**:

- nginx-proxy

**volumes**:

**static\_volume**:

**media\_volume**:

**certs**:

**html**:

**vhost**:

**acme**:

Next create an *.env.prod* file by duplicating the *.env.staging* file. You don't need to make any changes to it.

Finally, add an *.env.prod.proxy-companion* file:

DEFAULT\_EMAIL=youremail@yourdomain.com

NGINX\_PROXY\_CONTAINER=nginx-proxy

Build and push images again:

$ docker-compose -f docker-compose.prod.yml build

$ aws ecr get-login-password --region <aws-region> | docker login --username AWS --password-stdin <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com

$ docker-compose -f docker-compose.prod.yml push

Copy the new files and folders to your instance with SCP:

$ scp -i /path/to/your/djangoletsencrypt.pem **\**

**$(**pwd**)**/{.env.prod,.env.prod.proxy-companion,docker-compose.prod.yml} **\**

ubuntu@public-ip-or-domain-of-ec2-instance:/path/to/django-on-docker

Like before, connect to your instance via SSH and move to the project directory:

$ ssh -i /path/to/your/djangoletsencrypt.pem ubuntu@public-ip-or-domain-of-ec2-instance

$ cd /path/to/django-on-docker

Log in to your ECR Docker repository again:

$ aws ecr get-login-password --region <aws-region> | docker login --username AWS --password-stdin <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com

Pull the images:

$ docker pull <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com/django-ec2:web

$ docker pull <aws-account-id>.dkr.ecr.<aws-region>.amazonaws.com/django-ec2:nginx-proxy

And finally spin up the containers:

$ docker-compose -f docker-compose.prod.yml up -d

Navigate to your domain again. You should no longer see a warning.

Congrats! You're now using a production Let's Encrypt certificate for your Django application running on AWS EC2.

Want to see the certificate creation process in action, check out the logs:

$ docker-compose -f docker-compose.prod.yml logs nginx-proxy-letsencrypt