**Guidance for setup Elastic Load Balancer and Certificate Manager (ELB + ACM)**

**Requirements:**

* Django Application
* Gunicorn
* Nginx
* Docker container
* AWS – RDS PostgreSQL instance **(Used – Paused)**
* AWS EC2 instances **(Used – Paused)**
* Route53 **(Used)**
* Loadbalancer **(Used)**
* Certificate Manager **(Used)**

**Before you start**

In this documentation, I will walk you through how to create a Django web application with the use of Nginx as the reverse proxy webserver and Gunicorn for production environment, with PostgreSQL as our database then use Docker to bundle our whole application. And the final step is to deploy it to AWS Cloud Platform.

First, let’s create a new virtual environment for our new project. And activate the virtual environment. Inside your project directory, install **virtualenv** with command **pip install virtualenv.** Then create a virtual environment with **virtualenv “virtualenv\_name”**. After this command, a folder named **“virtualenv\_name”** will be created. Then to activate it, locate the **activate** executable file inside folder **Scripts**.

Text

Description automatically generated

**Step 1: Install the required dependencies**

We are going to use **Django version 4.1.1**, **Gunicorn**, **python-decouple**. Since **Gunicorn** doesn’t support on Windows, we are going to use **Waitress** which is the same as **Gunicorn**. Within the virtual environment we created and activated earlier, start to do **pip install** for the above requirements. We will write down what libraries or dependencies we use into file **requirements.txt**

Our **requirements.txt** file will then look like this:

**Text

Description automatically generated**

To be more specific, you can see the correct version list of dependencies that I use here:

A picture containing text

Description automatically generated

**Step 2: Prepare our application**

Running a local server of Django is not a good practice in production environment because Django only provide a test server not a production ready server. Use **Gunicorn** and **Nginx** will give a better security and performance for our application in production environment.

First, let’s create a static directory to serve static files for our application. When we see these files are served while running our application, that means that **Nginx** must be configured correctly. In **settings.py,** create the URL for static paths as below:

Text

Description automatically generated

Then, create two folders **static** and **media** inside our application directory as below:

Graphical user interface

Description automatically generated

Later in the future, we will connect to our Database hosted in a cloud service platform (AWS), so let’s set up the secret key for connection configuration. Import **os, config, environ** and change your **SECRET\_KEY** and **DEBUG** as below. This is a good practice that the secret key is not published or exposed and be accessed/consumed as environment variables.

Text

Description automatically generated

Then create a **.env** file. The **.env** file is a simple text configuration file that provide applications environment constants, we will store the **.env** file in the same directory with **settings.py** to serve as below. Copy the value of **SECRET\_KEY** from **settings.py** over to **.env** file.Text

Description automatically generated

Inside **index.html** file, we will use Bootstrap CSS with a **style.css** file. Again, we want to do this so that we know if our static files are served, that means **Nginx** is set up properly. File **style.css** is stored inside folder **static.**

Text

Description automatically generated

Our **style.css** simply looks like this

A screenshot of a computer

Description automatically generated with medium confidence

**Step 3: Run Django server with Gunicorn on Docker**

Create a **Dockerfile** for our Django application:

Text

Description automatically generatedSince **Gunicorn** doesn’t support on **Windows**, we are going to use **Waitress** instead of **Gunicorn**.

Now start to build our **Gunicorn/Django** application image, then try to run the container from the image with command **docker run -p 8000:8000 dj\_prod:latest**

Text

Description automatically generated

Graphical user interface, text

Description automatically generated

We can see that our application container is running but we don’t see our css file **style.css** is running. This happens because we didn’t setup Nginx yet, so static file is not serving right now.

**Step 4: Django deployment with Gunicorn + Nginx and Docker-compose**

In this step, we will add **Nginx** to our application stack (Django + Gunicorn) and then dockerize our full application with Docker-compose.

First, create **Nginx** folder with **default.conf** file and **Dockerfile** file as below:

Text

Description automatically generated

As mentioned above, Nginx acts as a reverse proxy for Django. In the **default.conf,** the upstream module is used to defined group of servers that can be referenced by **proxy\_pass, uwsgi\_pass**,… directives.   
The **proxy\_pass** directives tells Nginx to send all requests for that location to a specified address. In our case, the Nginx server listens on port 80 and at directive **/, proxy\_pass** will send all requests from that to address [**http://django**](http://django).   
  
In other cases, we can point **proxy\_pass** at a Unix domain socket with **Gunicorn** listening on that socker.

Text

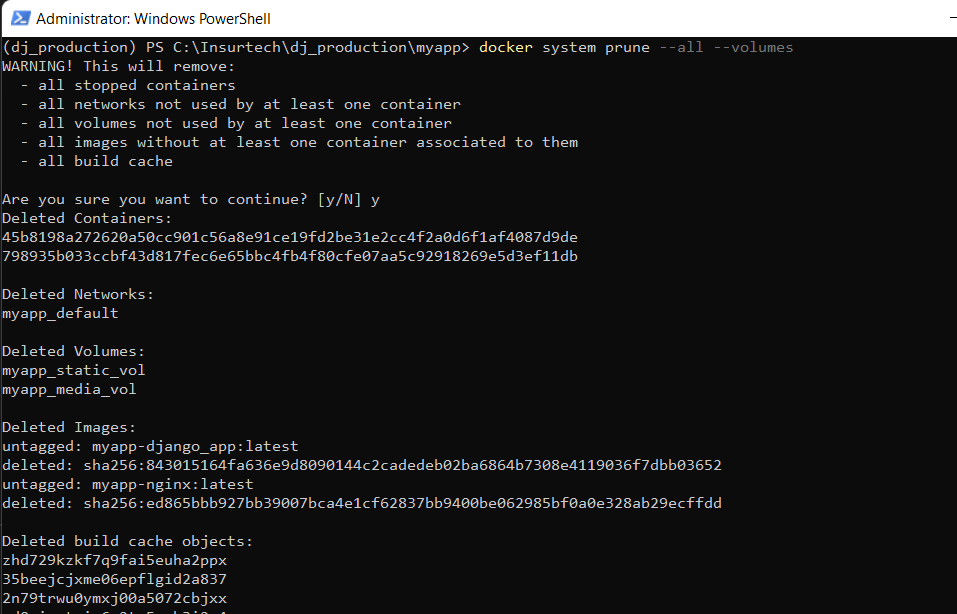
Description automatically generated

Now since Nginx is setup, we want to create a **docker-compose.yaml** file. Docker-compose is a practice to define and run multi-container Docker services. The structure is a YAML file and looks like this:

Text

Description automatically generated

Now, we will deploy our **Django** application with **Nginx** as a reverse proxy and **Gunicorn** as our WSGI. First, let’s clear other unused containers, etc with command **docker system prune –all --volumes**



The docker-compose file defines the list of services that our application is going to use. Here we have two services: (Django + Gunicorn) and Nginx. In each service’s definition, we provide the docker agent the path to dockerfile, the environment file, storage mapping volume between host and containers and the ports needed for each service.

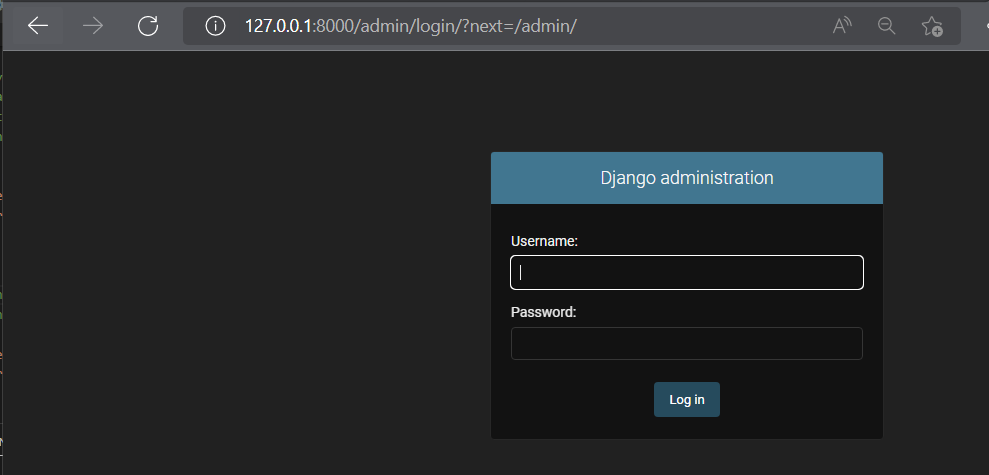
Then we will deploy our application with the command **docker-compose up –build**:

Text

Description automatically generated

Graphical user interface, website

Description automatically generated



As you can see, our application is running, and **Nginx** is also working because we can see the color from **style.css**.

**Step 5: Create a AWS – RDS PostgreSQL instance**

On your AWS RDS console, choose to create a database.

Create your database with standard create method and **PostgreSQL** engine type version **14.4-R1**. In this setup, we are going to use Free tier template. Give your RDS database a DB instance identifier with: master username and master password.

Graphical user interface, text, application

Description automatically generated

In **Connectivity,** for now we will use the default VPC with the default DB Subnet group and default VPC security group. Then click Yes on public access and finalize to create your RDS instance.

Head back to your RDS Console, and wait for the RDS database instance to complete the creation process. When the status is available, go to Configuration, you will see all the needed information there.

A screenshot of a computer

Description automatically generated

**Graphical user interface, application

Description automatically generated**

**Step 6: Connect Django application with AWS-RDS PostgreSQL**

In this step, we will establish a connection from our application to the created RDS database instance of AWS from above.

Inside **migrations** folder, delete **0001\_initial.py** file and **\_\_pycache\_\_** folder.

**Graphical user interface, application

Description automatically generated**

Then delete database file **db.sqlite3**

**Graphical user interface

Description automatically generated**

Then inside **settings.py** modify the database configuration as below

**Text

Description automatically generated**

As you can see, the credentials to connect to our RDS database instance will be supplied via environment variables in Django. In this step, we need to install **pip install django-environ** to our virtual environment. So go head and include this dependency into our **requirements.txt** file.

Then inside file **.env file** add these lines:

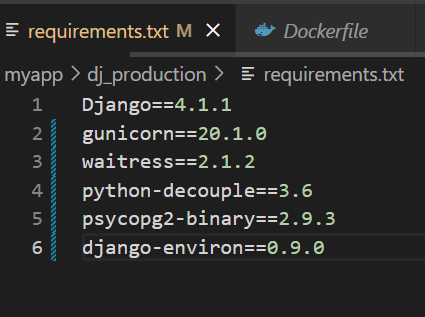
Text

Description automatically generated

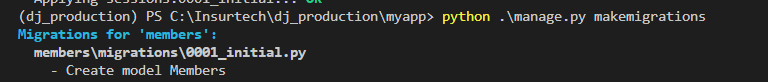
Next, we want to install **psycopg2** which is the PostgreSQl database adapter for Python. Use this command **pip install psycopg2-binary**. Then write **psycopg2-binary** to our **requirements.txt** file. Our dependencies requirement file now looks like this

**Text

Description automatically generated**

****

Migrate our model and database to our application with these two commands **python manage.py makemigrations** and **python manage.py migrate**

****

**Text

Description automatically generated**

Now try to run our application with **python manage.py runserver**:

Graphical user interface, text

Description automatically generated

As you can see here, our application is running along with Nginx is serving static files properly.

Let’s double check on our database connection, I will use **DBeaver** to connect to our AWS-RDS Database instance to see if the correct table and data were created and stored. Open **DBeaver** and use the information of RDS instance to connect to the database, then inside **django\_database,** table **Members** is automatically correctly createdwith the correct data present.

Graphical user interface, application

Description automatically generated

Now let’s close the local app and try to run our application again with **docker-compose** command:

**Text

Description automatically generated**

**Table

Description automatically generated with medium confidence**

**A screenshot of a computer

Description automatically generated**

Everything is working properly locally with docker-compose. Let’s go to next step and start our deployment to AWS

**Step 7: Deploy Django stack on EC2-Instance**

First, let’s create a Github repository and push our project to the repo. This way, we can then keep track of the changes we make and easier to clone the application to our EC2 instance or other platform.

The project is stored in this repository: [anhhavo/dj\_production (github.com)](https://github.com/anhhavo/dj_production)

Create a EC2 instance and connect to your EC2 instance.

After successfully connected to your EC2 instance, let’s install **git** to our EC2 instance:  
**sudo yum update -y  
sudo yum install git -y  
git version**

Install **docker** to our EC2 instance:  
**sudo amazon-linux-extras install docker  
sudo service docker start  
sudo usermod -a -G docker ec2-user**

Make **docker** auto-start  
**sudo chkconfig docker on  
sudo reboot**

Install **docker-compose** to your EC2 instance: **sudo curl -L** [**https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname%20-s)-$(uname%20-m)%20-o%20/usr/local/bin/docker-compose**](https://github.com/docker/compose/releases/latest/download/docker-compose-$(uname%20-s)-$(uname%20-m)%20-o%20/usr/local/bin/docker-compose) **sudo chmod +x /usr/local/bin/docker-compose  
docker-compose version**

Next, we want to attach a static IP address to our EC2 instance. In order to do this, we will attach an Elastic IP address to it.

Allocate an Elastic IP address and then associate it to our EC2 instance. Then clone our project inside our EC2 instance. Because we don’t want to push file **.env** to our Github, we will need to manually create file **.env** inside our EC2 instance. Then copy over the values for environment variables.

Text

Description automatically generated

When everything is ready, fire up **docker-compose up**:

Graphical user interface, text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

Graphical user interface, text

Description automatically generated

Great, our application is successfully deployed. Next, we want to register a domain and attach it to our web application then register SSL\TSL certificate to it.

**Step 8: Register DNS with Route53**

In Route53, register a domain name that you want our web application is at. Here I registered a DNS **dowcai.com**

**Graphical user interface, text, application, email

Description automatically generated**

The next step is to create a hosted zone with the same domain name **dowai.com**. After initially setup the hosted zone, you will see that there are 2 records type NS and SOA already created. In the description of type NS record, there are 4 name servers.

We will then perform these steps:

* Copy the name servers from our registered domain to our record type NS.
* And do the opposite, copy the name servers from our record type NS to our new registered domain.
* Create a new record type A with the value of our Elastic IP address.
* Create a new record type CNAME with the value of our Elastic IP address and the sub domain is **www**

At this point, your hosted zone will look like this except for the second CNAME type I will explain in the next step:

Graphical user interface, text, application

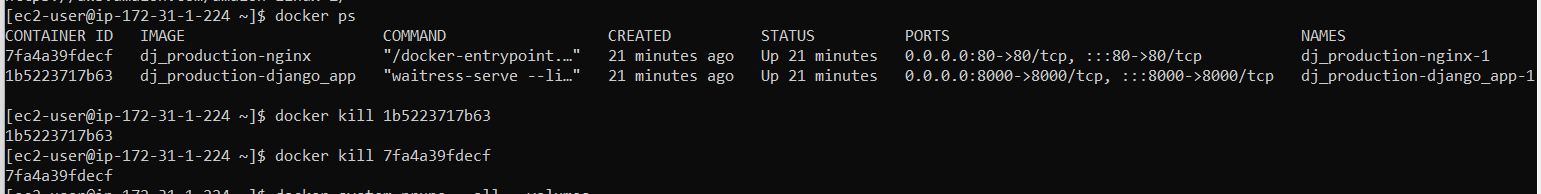
Description automatically generated

For the Domain name to get fully accepted, we will need to add this line in our **settings.py.** This way, any modification toward our database will be successfully done.



Later, you can remove other values except or [**https://dowcai.com**](https://dowcai.com) in **CSRF\_TRUSTED\_ORIGINS**.

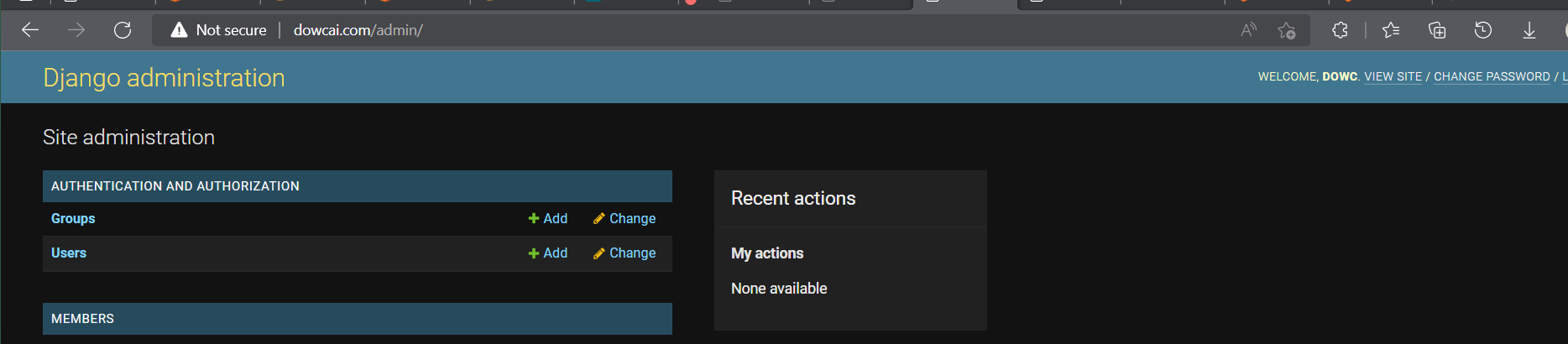
Remember to push your changes to Github from your local machine and then pull the new version to our EC2 instances. Before pulling, make sure you kill the running containers first.



After this step is done, fire up we can access our application via   
[**http://dowcai.com/members/**](http://dowcai.com/members/)or [**http://dowcai.com/admin/**](http://dowcai.com/admin/)

**Table

Description automatically generated**

****

**Step 9: Apply AWS Certificate Manager (ACM)**

In this step, we want to apply SSL\TLS certificate for our domain. When this step is succeeded, our site will be secured, and the connection will be encrypted. When requests to our site, it will start with **https** instead of **http**. Now look for **AWS Certificate Manager (ACM)** console on AWS Console website:

Text

Description automatically generated

Click **Request a certificate** and choose to request a public certificate:

Graphical user interface, text, application, email

Description automatically generated

After you request a certificate, inside your Certificate click **Create records in Route 53**.

Graphical user interface, text, application

Description automatically generated

In the above picture, I already finish the step but after you created one, inside your hosted zone you will see a new record added:

Graphical user interface, application

Description automatically generated

**Step 10: Create a Elastic Load Balancer**

Now inside your EC2 Instance Console, on the left panel click to **Load Blancers** and click **Create a Load Balancer.** Here we want to use Application Load BalancerDiagram

Description automatically generated

Apply a name for your load balancer choose the same virtual private cloud (VPC) that we are using for our RDS/EC2. Then select at least two available zones and subnet that match where you host your EC2:

Graphical user interface, text, application, email

Description automatically generated

For Security Group, I suggested to create a new security group policy for your load balancer and apply these inbound rules:

Graphical user interface, application

Description automatically generated

Then we will add two listeners, one for HTTP protocol on port 80 and one for HTTPS protocol on port 443. To make this easier, click on to create a target group.

Graphical user interface, text, application, email

Description automatically generated

The target group will help load balancers to route requests/traffic into a group of targets (EC2 instances, IP addresses, …). Follow the steps to group the list of instances you want Load Balancers to route to. Finally, create the load balancer.

Graphical user interface, text, application, email

Description automatically generated

As you can see, the load balancer has its own DNS name. Because of this, we want to modify our routing table inside Route 53 so that traffics will always come to the load balancer first.

Go back to our Route53 hosted zone, delete the old type A record and create a new record with type A but this time choose Alias – Load balancer:

Graphical user interface, text, application, email

Description automatically generated

After 10 minutes, our application should now be accessible via [**https://dowcai.com**](https://dowcai.com)

Table

Description automatically generated

A screenshot of a computer

Description automatically generated

Final step, we will modify the routing traffic in load balancer one more time so that whenever user type **dowcai.com** in the browser**,** the request will always direct to [**https://dowcai.com**](https://dowcai.com)and traffic that goes to[**http://dowcai.com**](http://dowcai.com)will also be redirected to [**https://dowcai.com**](https://dowcai.com)instead.

Now go to our Load Balancer description and modify the traffic from HTTP port 80 to be redirected to HTTPS port 443.