SHRADDHA SAINI
R171217055
500062194
B.TECH CSE DEVOPS VII

DJANGO | GIT | JENKINS | DOCKER | TERRAFORM

SYSTEM PROVISIONING AND CONFIGURATION MANAGEMENT

django

BUILDING A BLOG APPLICATION

CREATED TWO NODE -

1. ADMIN
2. BLOG

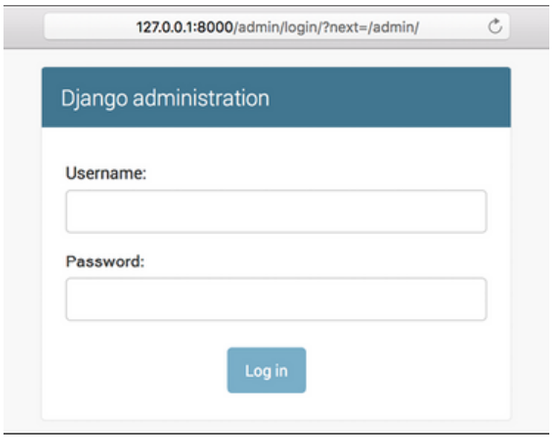
```
from django.db import models
from django.utils import timezone
from django.contrib.auth.models import User
class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250,
                            unique_for_date='publish')
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
                              choices=STATUS_CHOICES,
                              default='draft')
    class Meta:
        ordering = ('-publish',)
    def __str__(self):
        return self.title
```

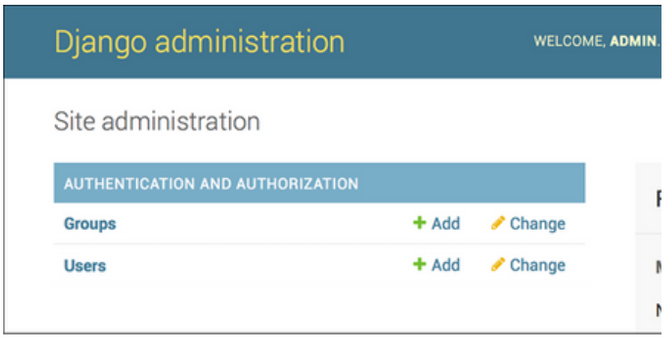
BLOG STRUCTURE

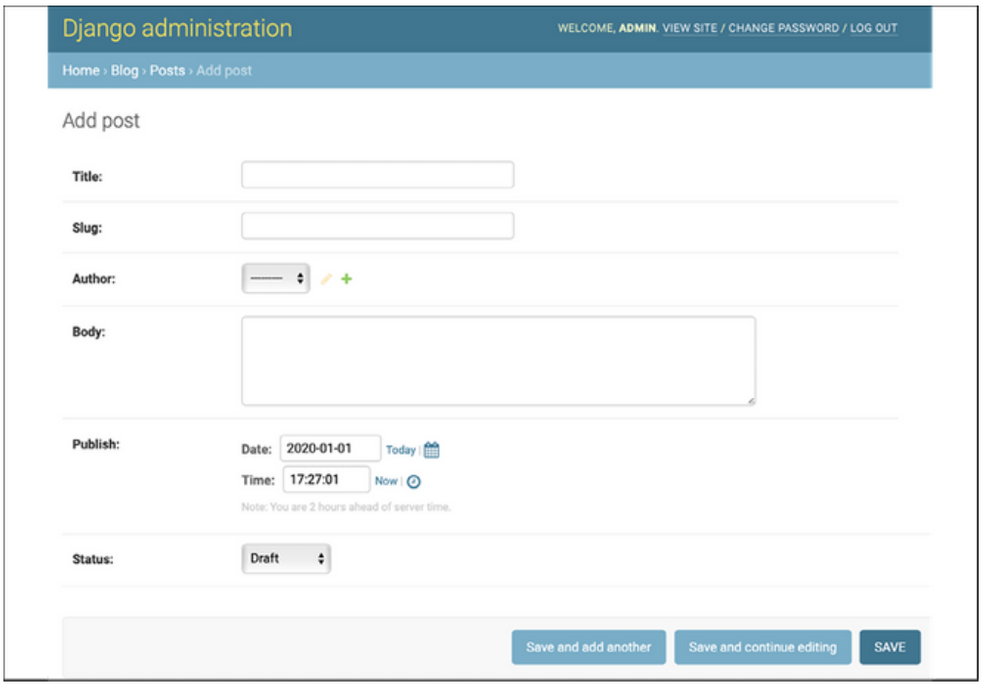
This is your model for blog posts. Let's take a look at the fields you just defined for this model:

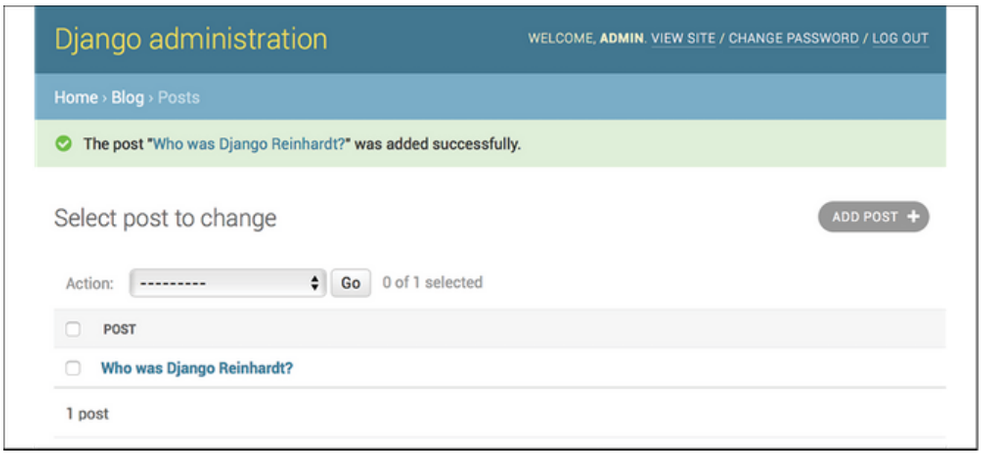
title: This is the field for the post title. This field is CharField, which translates into a VARCHAR column in the SQL database.
slug: This is a field intended to be used in URLs. A slug is a short label that contains only letters, numbers, underscores, or hyphens. You will use the slug field to build beautiful, SEO-friendly URLs for your blog posts. You have added the unique_for_date parameter to this field so that you can build URLs for posts using their publish date and slug. Django will prevent multiple posts from having the same slug for a given date.
author: This field defines a many-to-one relationship, meaning that each post is written by a user, and a user can write any number of posts. For this field, Django will create a foreign key in the database using the primary key of the related model. In this case, you are relying on the User model of the Django authentication system. The on_delete parameter specifies the behavior to adopt when the referenced object is deleted. This is not specific to Django; it is an SQL standard. Using CASCADE, you specify that when the referenced user is deleted, the database will also delete all related blog posts. You can take a look at all the possible options at https://docs.djangoproject.com/en/3.0/ref/models/fields/#django.db.models.ForeignKey.on_delete. You specify the name of the reverse relationship, from User to Post, with the related_name attribute. This will allow you to access related objects easily. You will learn more about this later.
body: This is the body of the post. This field is a text field that translates into a TEXT column in the SQL database.
publish: This datetime indicates when the post was published. You use Django's timezone now method as the default value. This returns the current datetime in a timezone-aware format. You can think of it as a timezone-aware version of the standard Python datetime.now method.
created: This datetime indicates when the post was created. Since you are using auto_now_add here, the date will be saved automatically when creating an object.
updated: This datetime indicates the last time the post was updated. Since you are using auto_now here, the date will be updated automatically when saving an object.
status: This field shows the status of a post. You use a choices parameter, so the value of this field can only be set to one of the given choices.

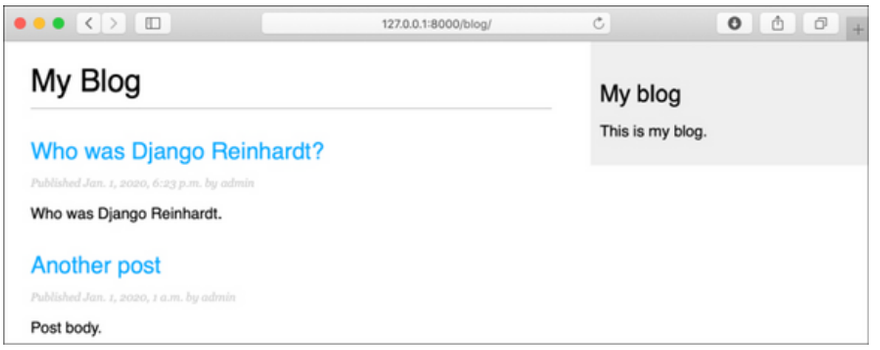
PREVIEW OF THE BLOG APPLICATION











DOCKERING DJANGO WITH DOCKER-COMPOSE

IF YOU WANT A COMBINATION OF EASE-OF-USE AND REPLICATION OF A DEPLOYMENT ENVIRONMENT, THERE'S NO BETTER SOLUTION THAN USING DOCKER COMPOSE.

Virtualenv was the tool of choice for Python developers before Docker came along,as it allowed you to separate package dependencies for different applications you're working on without having to create separate virtual machines for each one. It worked very well for me. However, there are still some annoying things the developers will have to deal with. Such things include having to install additional applications and services required by the project such as PostgreSQL, RabbitMQ, and Solr, for example. Also, some Python packages won't install/compile properly without additional libraries installed on the developer's machine. There is also the issue of predictability when you deploy to production. You may be deploying it on Windows but your production server runs Ubuntu.

Of course, virtual machines can solve these issues. But VMs are heavy.

With Docker, these issues go away. You can have all these services in isolated Docker containers that are lightweight and start up very quickly. You can use base images for different Linux distros, preferably the same distro and version you use in production.

THIS BLOG SHOWS YOU HOW TO SET UP A DJANGO APPLICATION AND DEVELOPMENT ENVIRONMENT USING DOCKER.
1. INSTALL DOCKER ENGINE:
The first step is to install the Docker Engine for your platform.

2. INSTALL DOCKER COMPOSE
DEFINING THE COMPONENTS:
Create a Dockerfile,
Python dependencies file
docker-compose.yml file.
Create an Empty Project Directory: This directory is the context for your application image. The directory should only contain resources to build that image.
Create a Dockerfile: The Dockerfile defines an application's image content via one or more build commands that configure that image. Once built, you can run the image in a container.
Add the following content to the Dockerfile:

```
FROM python:3
ENV PYTHONUNBUFFERED 1
RUN mkdir /code
WORKDIR /code
COPY requirements.txt /code/
RUN pip install -r requirements.txt
COPY . /code/
```

Save and close the Dockerfile.
In your project directory create a requirements.txt: This file is used by the RUN pip install -r requirements.txt command in your Dockerfile.
Add the required softwares in the file:

```
Django>=2.0,<3.0
psycopg2>=2.7,<3.0
```

Save and close requirements.txt file.
Create a file called docker-compose.yml: This file describes the services that make your app. In this example those services are a web server and database. The compose file also describes which Docker images these services use, how they link together, any volumes they might need mounted inside the containers. Finally, the docker-compose.yml file describes which ports these services expose.
Add the following configuration to the file:

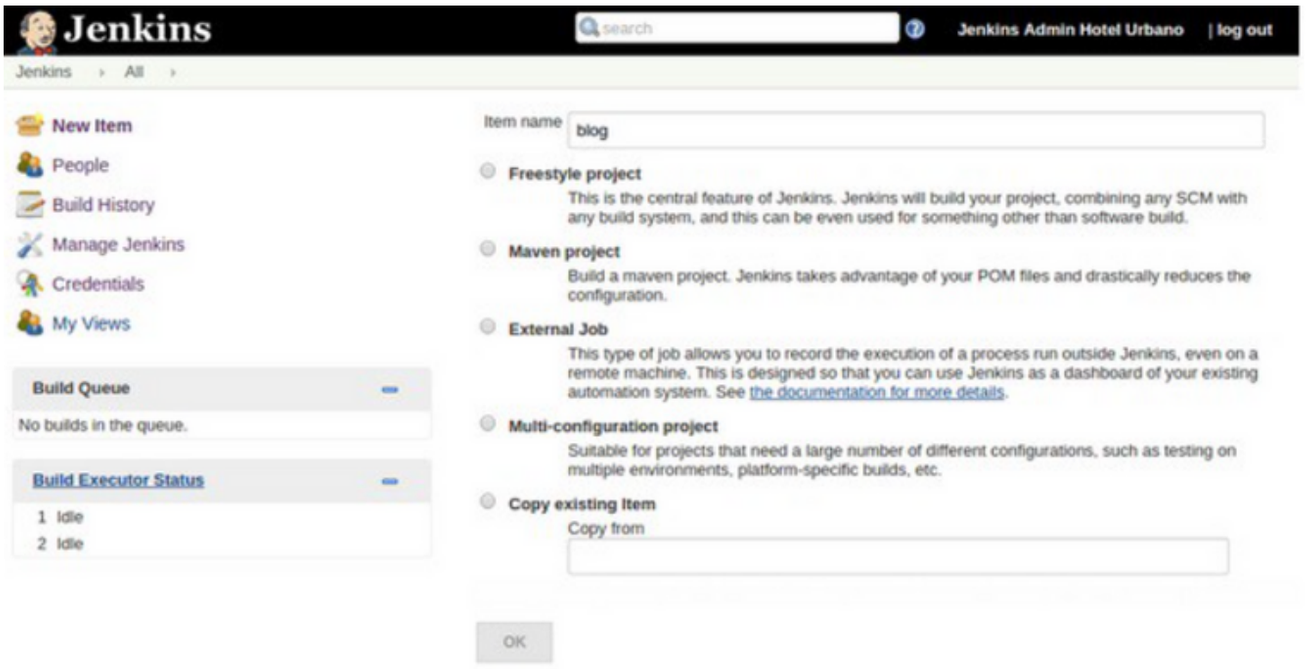
```
version: '3'

services:
  db:
    image: postgres
  web:
    build: .
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - .:/code
    ports:
      - "8000:8000"
    depends_on:
      - db
```

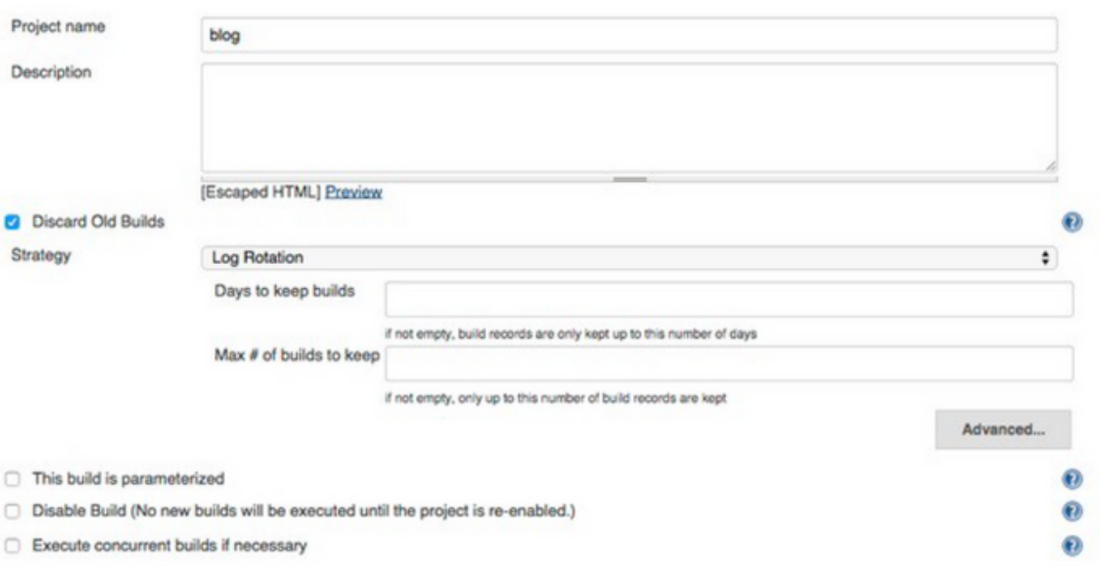
Save and close this file.
CREATE A DJANGO PROJECT:
You'll need to use the docker-compose run command to start your Django project. Of course, if you've already got a project started this step is unnecessary, but it may still be helpful to read through. You create a Django starter project by building the image from the build context defined in the previous procedure.

ASSEMBLING A CI SERVICE FOR A DJANGO PROJECT ON JENKINS

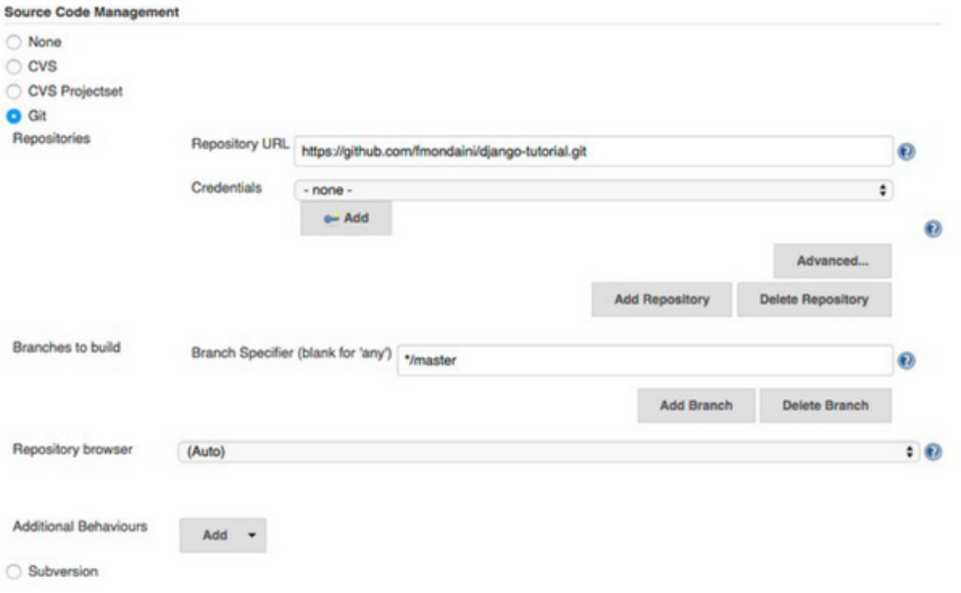
Configuring our Job
Now with our Jenkins server installed with the Git plugin, we must configure our job for Jenkins to be able to poll our repository for changes.
Initial configuration
On Jenkins homepage, click on “New Item” to create a new Job. Write the job name and select the “Freestyle project” option:



On the next page, select the “Discard Old Builds” option so the older builds to be discarded automatically.



Accessing the repository:
This step is vital. Here we are going to select the type of the repository (Git, Subversion, CVS), address, which branch to pick up and so on. As our project is a git repository, select the Git option and type the Repository URL:



Build Triggers
Here we can schedule the Job according to our needs. In this case, I chose to poll the repository on every 15 minutes. If there is a change, Jenkins will download it and do the tests.

Build
After downloading the newer version, Jenkins must run the tests to finally validate our project. Let's add a step on Build. In order to configure we will add a small shell script.

USING TERRAFORM

How to Deploy a load balanced Dockerized Python web application on AWS using terraform for automation and nginx for loadbalancing.

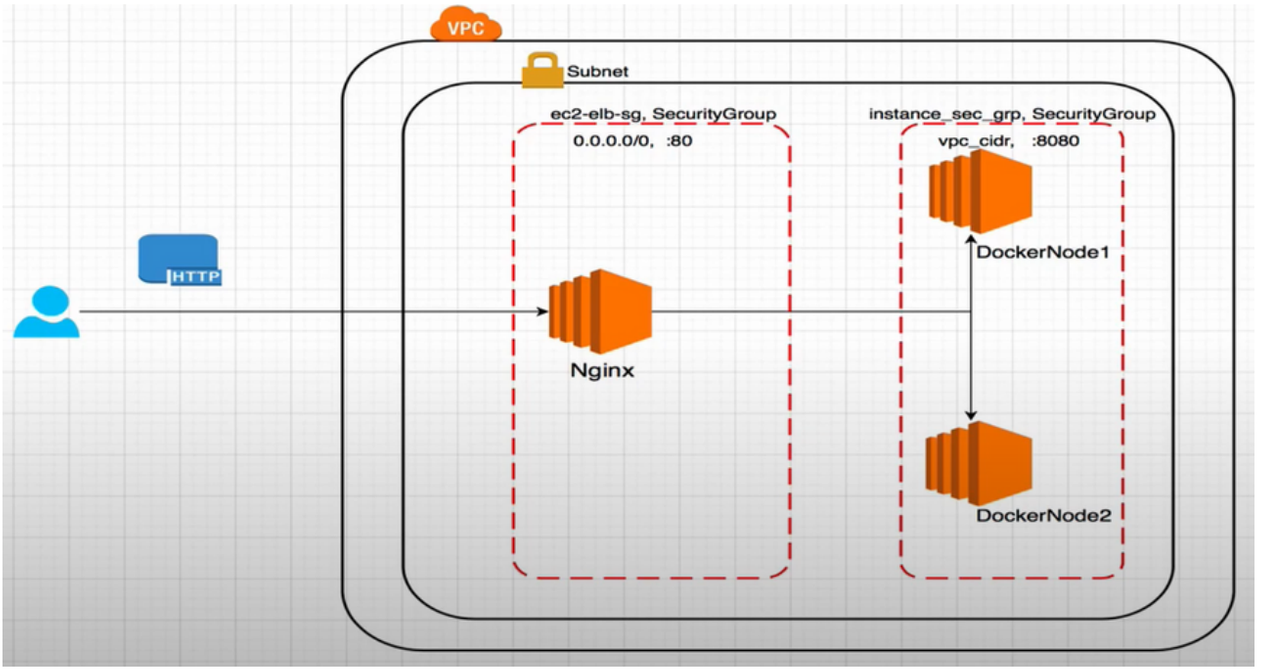
Complete end-to-end deployment including custom VPC, Subnets, Security Groups and ECS-optimized Alinux2 based EC2's

A Sample blog app

- Python app
- Dockerfile

Terraform code

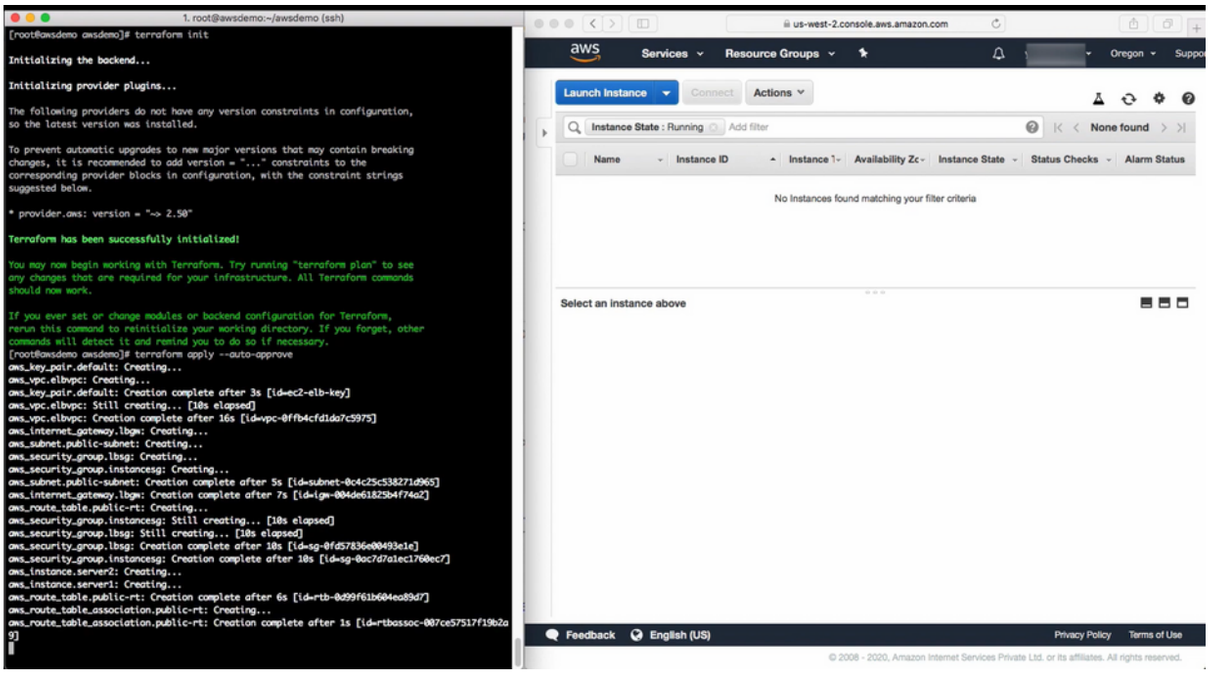
- Overall Deployment Architecture
- Terraform file layout
- Individual resources used



THE ARCHITECTURE

The Deployment

- Terraform Docker setup with Nginx for Load Balancing
- ECS-optimized Linux AMI used for all EC2 nodes.



Terraform docker setup.

Terraform docker setup using nginx LB at front.
Worker-nodes have port 8080 exposed only to internal VPC IPs and Nginx node has only 80 port exposed.
AMI being used in ECS-optimised Alinux2 (as this has most of docker and docker dependencies pre-installed).
IPs added to /etc/hosts file using provisioner 'remote-exec'. Updating nginx conf files through user data.
Outputs Docker nodes private IP and Nginx nodes Public DNS name
To access webapp, user URL <Nginx_Public_dns>/app1
Docker nodes can run multiple dockers at different port and nginx can be made to point them.
Working with provider.aws ~> 2.50 and TF > 12.xx
To run as is, run below commands

$ cd terraform
$ terraform init
$ terraform plan
$ terraform apply