Deploy 8: CICD

Kenneth Tan

Goal: Create CICD Pipeline using Ansible, Docker, and Cypress for an application

Ansible:

- Used to provision the EC2s
- Used to download dependencies
- Encrypt different reports

Docker:

- Used to containerized application to de deployed on production EC2 after testing on Test EC2
- Pull image of the successful app onto the prod ec2

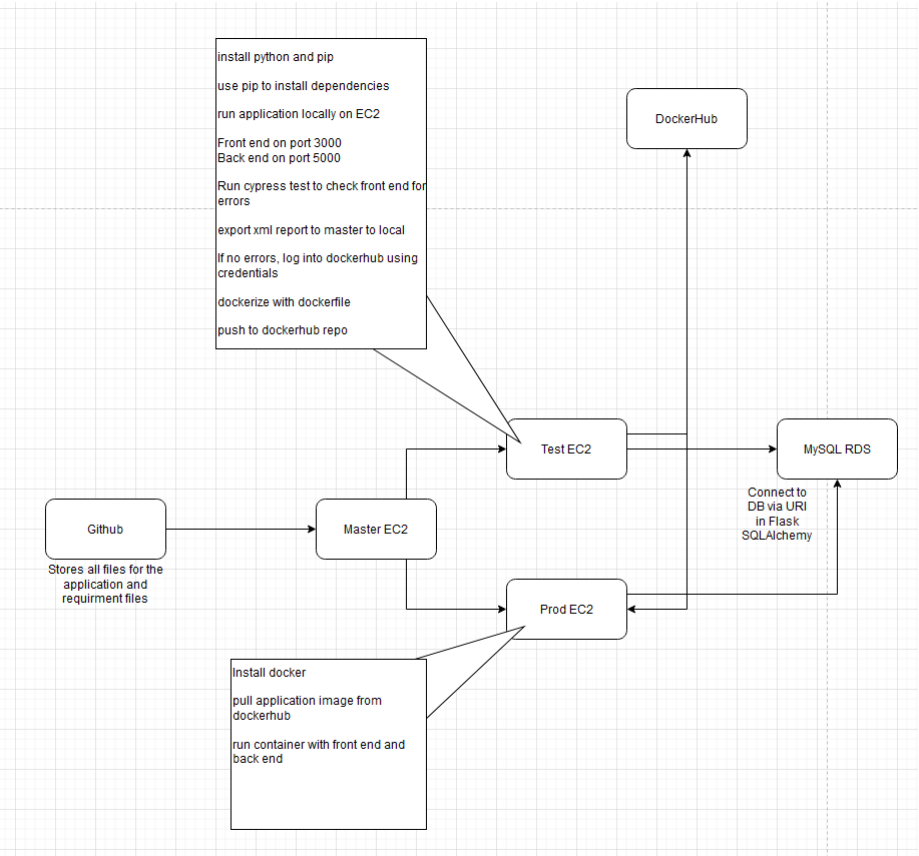
Cypress:

- Testing framework used in the Test EC2

Application:

- Application that was used in this deployment was a simple app that take in 2 fields of data and stores it in a database
- Used RDS as the database. This was connected via SQLAlchemy in Flask.

To begin creating the pipeline, we will follow the architecture as shown below.



The EC2s will be created by an ansible playbook. I named it new-ubuntu20-ec2.yaml. This will create 3 EC2s labelled Master, Test, Prod.

```yaml
---

- name: Spin up new ubuntu EC2
  hosts: localhost
  gather_facts: false

  tasks:
  - name: start MASTER ubuntu
    ec2_instance:
      name: "Master EC2"
      key_name: "ubuntu-aws"
      vpc_subnet_id: subnet-01afc7aabbb35ba9e
      instance_type: t2.micro
      security_group: default
      network:
        assign_public_ip: true
      image_id: ami-083654bd07b5da81d
      tags:
        Name: Ansible-Master
      state: present
      wait: yes

  - name: start TEST ubuntu
    ec2_instance:
      name: "Test EC2"
      key_name: "ubuntu-aws"
      vpc_subnet_id: subnet-01afc7aabbb35ba9e
      instance_type: t2.micro
      security_group: sg-0fbf51002134f5c7a
      network:
        assign_public_ip: true
      image_id: ami-083654bd07b5da81d
      tags:
        Name: Ansible-Test
      state: present
      wait: yes

  - name: start PROD ubuntu
    ec2_instance:
      name: "Prod EC2"
      key_name: "ubuntu-aws"
      vpc_subnet_id: subnet-01afc7aabbb35ba9e
      instance_type: t2.micro
      security_group: ansible-agent-sg
      network:
        assign_public_ip: true
      image_id: ami-083654bd07b5da81d
      tags:
        Name: Ansible-Prod
      state: present
      wait: yes
```

The EC2 dependencies will be installed via install-dependencies.yaml.

First, there will need to an update for the hosts file for ansible so it will be able to SSH in each instance and install their respective dependencies

Master will need to run Jenkins
Test will need to have Docker, and Java
Prod will need to have Docker, Java, and stress-ng

```
- name: Install dependencies for Master
  hosts: Master
  become: yes
  gather_facts: True
  tasks:
  - name: update
    shell: sudo apt update && sudo apt upgrade -y
  - name: jenkins resources
    shell: wget -q -O - https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo apt-key add -
  - name: jenkins resources
    shell: sudo sh -c 'echo deb http://pkg.jenkins.io/debian-stable binary/ > /etc/apt/sources.list.d/jenkins.list'
  - name: update with new packages
    shell: sudo apt update
  - name: install java
    shell: sudo apt install default-jre -y
  - name: install jenkins
    shell: sudo apt install jenkins -y
  - name: start jenkins
    shell: sudo systemctl start jenkins

- name: Install Java for Test
  hosts: Test
  become: yes
  gather_facts: True
  tasks:
  - name: update
    shell: sudo apt update && sudo apt upgrade -y
  - name: install java
    shell: sudo apt install default-jre -y

- name: Install Java and Docker for Prod
  hosts: Prod
  become: yes
  gather_facts: True
  tasks:
  - name: update
    shell: sudo apt update && sudo apt upgrade -y
  - name: install java
    shell: sudo apt install default-jre -y
  - name: install needed modules
    shell: sudo apt-get install \ ca-certificates \ curl \ gnupg \ lsb-release
  - name: add dockers key
    shell: curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
  - name: stable repo
    shell:  echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
  - name: update packages
    shell: sudo apt update
  - name: install docker engine
    shell: sudo apt install docker-ce \ docker-ce-cli \ containerd.io -y
```

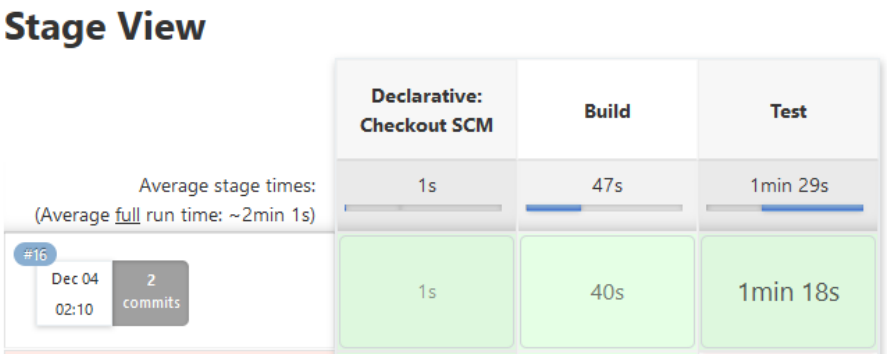
When Jenkins is configured on the Master, configure the Test and Prod Agents.

In the Test EC2, The application will be built, both frontend and backend. The application will also be tested with Cypress. There will need to be a directory added to the source repo, in this case the Github repo.

cypress/integration/test.spec.js will need to be added. This file will be what cypress refers to to know what to test and what to test for. In this case, testing for the header to say “Hello again react”
cypress.json will need to be added. This will be the configuration file for where the xml report that is generated will be. In this case in /results/cypress-results.xml. This xml file will be encrypted with ansible-vault. Ansible was installed on the EC2 and ran

ansible-vault encrypt cypress-results.xml

Encrypted file shown on Github.



After there is a successful test, we will dockerize the application in two parts, the react half and the flask half. These two images will be created and pushed to dockerhub. To do that, dockerhub credentials will need to be added to jenkins.

Errors

At this point, I began having issues dockerizing the front and back ends. I ran into a number of errors including

Key that suddenly lost permission, causing me to lose access to my EC2's. I ended up needing to remake the EC2's several times.

Below error when attempting to dockerize the images.

```
[91mError: error:0308010C:digital envelope routines::unsupported
    at new Hash (node:internal/crypto/hash:67:19)
    at Object.createHash (node:crypto:130:10)
    at module.exports (/app/node_modules/webpack/lib/util/createHash.js:135:53)
    at NormalModule._initBuildHash (/app/node_modules/webpack/lib/NormalModule.js:417:16)
    at handleParseError (/app/node_modules/webpack/lib/NormalModule.js:471:10)
    at /app/node_modules/webpack/lib/NormalModule.js:503:5
    at /app/node_modules/webpack/lib/NormalModule.js:358:12
    at /app/node_modules/loader-runner/lib/LoaderRunner.js:373:3
    at iterateNormalLoaders (/app/node_modules/loader-runner/lib/LoaderRunner.js:214:10)
    at iterateNormalLoaders (/app/node_modules/loader-runner/lib/LoaderRunner.js:221:10)
[0m[91m/app/node_modules/react-scripts/scripts/start.js:19
  throw err;
  ^

Error: error:0308010C:digital envelope routines::unsupported
    at new Hash (node:internal/crypto/hash:67:19)
    at Object.createHash (node:crypto:130:10)
    at module.exports (/app/node_modules/webpack/lib/util/createHash.js:135:53)
    at NormalModule._initBuildHash (/app/node_modules/webpack/lib/NormalModule.js:417:16)
    at /app/node_modules/webpack/lib/NormalModule.js:452:10
    at /app/node_modules/webpack/lib/NormalModule.js:323:13
    at /app/node_modules/loader-runner/lib/LoaderRunner.js:367:11
    at /app/node_modules/loader-runner/lib/LoaderRunner.js:233:18
    at context.callback (/app/node_modules/loader-runner/lib/LoaderRunner.js:111:13)
    at /app/node_modules/babel-loader/lib/index.js:59:103 {
  opensslErrorStack: [ 'error:03000086:digital envelope routines::initialization error' ],
  library: 'digital envelope routines',
  reason: 'unsupported',
  code: 'ERR_OSSL_EVP_UNSUPPORTED'
}
[0m[91m
Node.js v17.2.0
[0mThe command '/bin/sh -c npm run start' returned a non-zero code: 1
```

```
Step 1/5 : FROM node:latest
 ---> 058747996654
Step 2/5 : COPY . /app
Error processing tar file(exit status 1): write /app/node_modules/jsx-ast-utils/lib/values
/expressions/MemberExpression.js: no space left on device
[Pipeline] }
```

Below are notes for what I am planning to do.

Now that we have tested our application and have made it downloadable, the prod agent will pull the docker image and run 2 containers, one for the react image and one for the flask image. The reason we separate this is because together it may be too large for one container to run.

We can now connect these two containers with a docker bridge. This will allow the 2 to communicate with each other much like how it did in the test ec2.

When the containers are created, expose the appropriate ports

Docker run -p 5000:5000 deploy8-backend
Docker run -p 3000:3000 deploy8-frontend

Docker network create
Docker network connect

The application will now be available to run at the ip of the prod ec2 on port 3000.