Deployment 2 Documentation

Nasir York

Deployment goal:

Learn to set up a custom CI/CD Pipeline

Software and Tools Used:

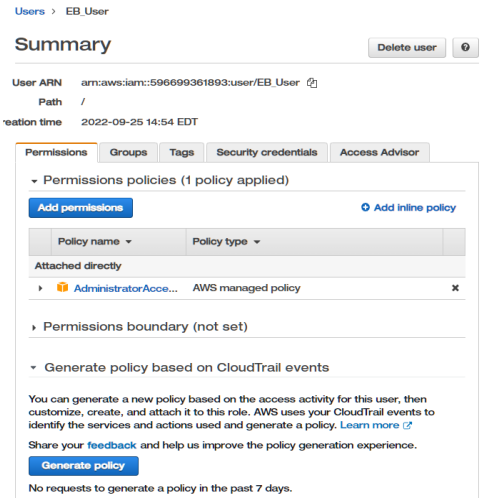
EC2, Jenkins, GitHub, IAM, AWS CLI, Elastic Beanstalk CLI, Elastic Beanstalk, Slack, DataDog

EC2:

First I started off by launching a new Amazon EC2 Instance. I opened ports 8080, 80, and 22 to allow for Jenkins to run, Http access, and SSH access. In order to run Jenkins I needed to Install its main dependency being Java. After my EC2 booted up I created a jenkins User. I needed to add the newly created Jenkins user to the sudoers list (located: ~/etc/sudoers) to grant it administrative permission. I then Installed Python3, Pip, and Python-venv since I was deploying a Python application as well as .AWS and Elastic Beanstalk CLI.

IAM:

I then launched Amazon IAM to create an Amazon user that we could assign permission to in order to configure and access the application we want to deploy.



AWS CLI:

I used the Access Key ID and Secret Access Key from the IAM user EB_User along with my region (us-east-1) and output format (json) to configure aws access to our EC2.

GitHub:

Github was where we stored the code for our application. Also Included is a file (Jenkinsfile) that Instructs jenkins on how we would like our pipeline to be constructed.

-This Pipeline consists of 3 stages: Build, Testing, and Deploy stage.

-The build stage creates a virtual environment for the application along with its requirements and runs it within a flask app.

-The test stage run py.test and posts its results.

-The Deploy stage will deploy or deploy our application once changes are made to the Github code.

```
pipeline {
  agent any
   stages {
    stage ('Build') {
      steps {
        sh '''#!/bin/bash
        python3 -m venv test3
        source test3/bin/activate
        pip install pip --upgrade
        pip install -r requirements.txt
        export FLASK_APP=application
        flask run &
        '''
```

```
stage ('test') {
  steps {
    sh '''#!/bin/bash
    source test3/bin/activate
    py.test --verbose --junit-xml test-reports/results.xml
    '''
  }

  post{
    always {
      junit 'test-reports/results.xml'
    }

  }
}
```

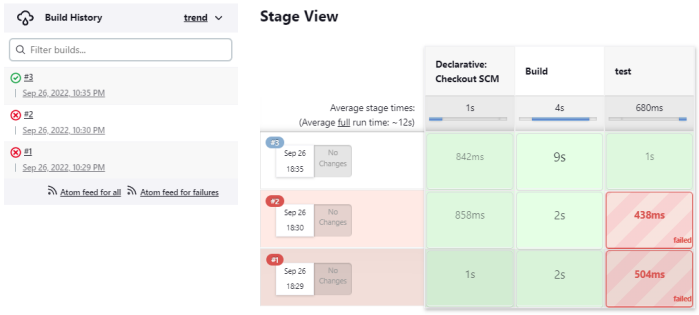
```
stage ('Deploy') {
  steps {
    sh '/var/lib/jenkins/.local/bin/eb deploy url-shortener-main-dev'
```

Jenkins:

Jenkins was used to track the entire pipeline for our application.

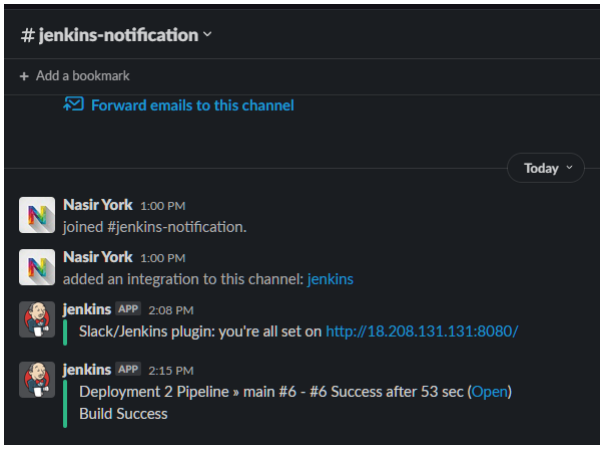
- I chose a multibranch pipeline and connected my forked repo to jenkins.

-Below is a successful deployment of all three stages created in the JenkinsFile.



Slack:

-Using a Jenkins Plugin I was able to configure jenkins to send me a notification with the result of the recent build.



Elastic Beanstalk CLI:

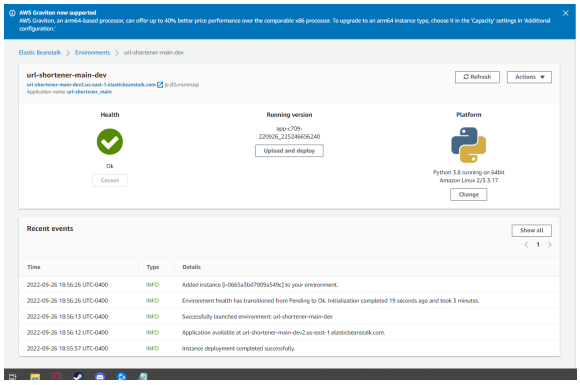
-With the success of the Jenkins deployment I was able to run eb init and eb create within the EC2 terminal to Launch the Elastic Beanstalk environment.

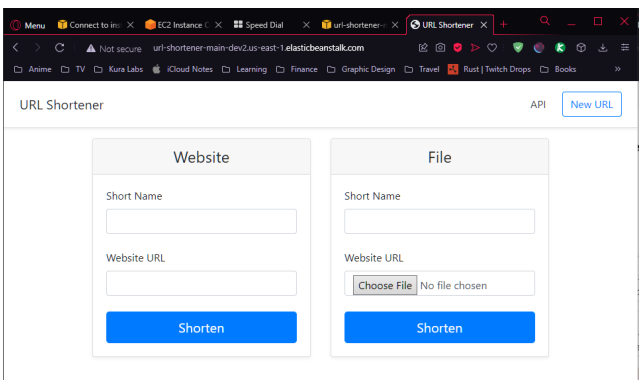
Elastic Beanstalk:

Here are the results of the successful deployments.

-Elastic Beanstalk created an environment to run and host the application.

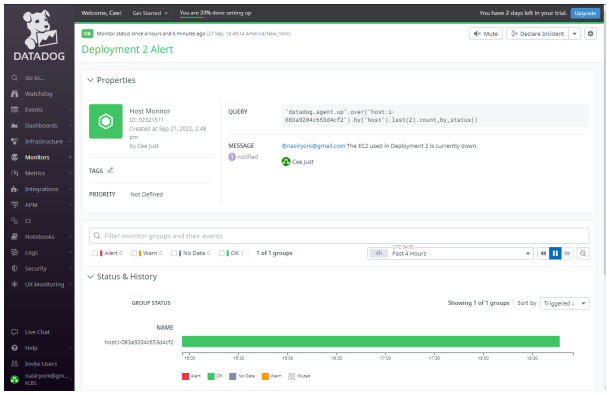
-The application was set up correctly and was accessible through the elastic beanstalk generated link.

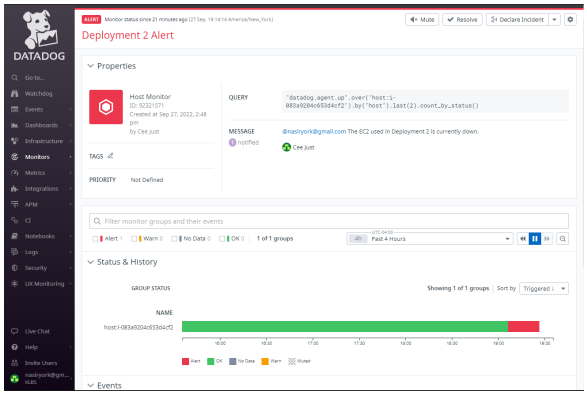


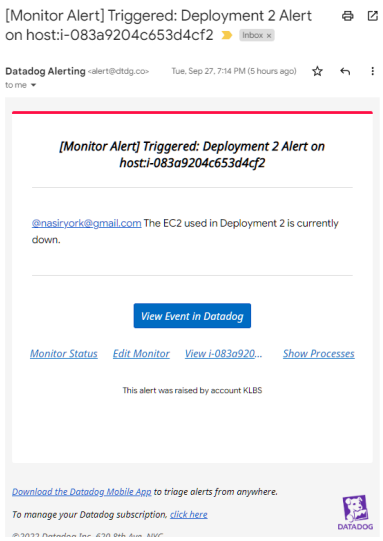


DataDog

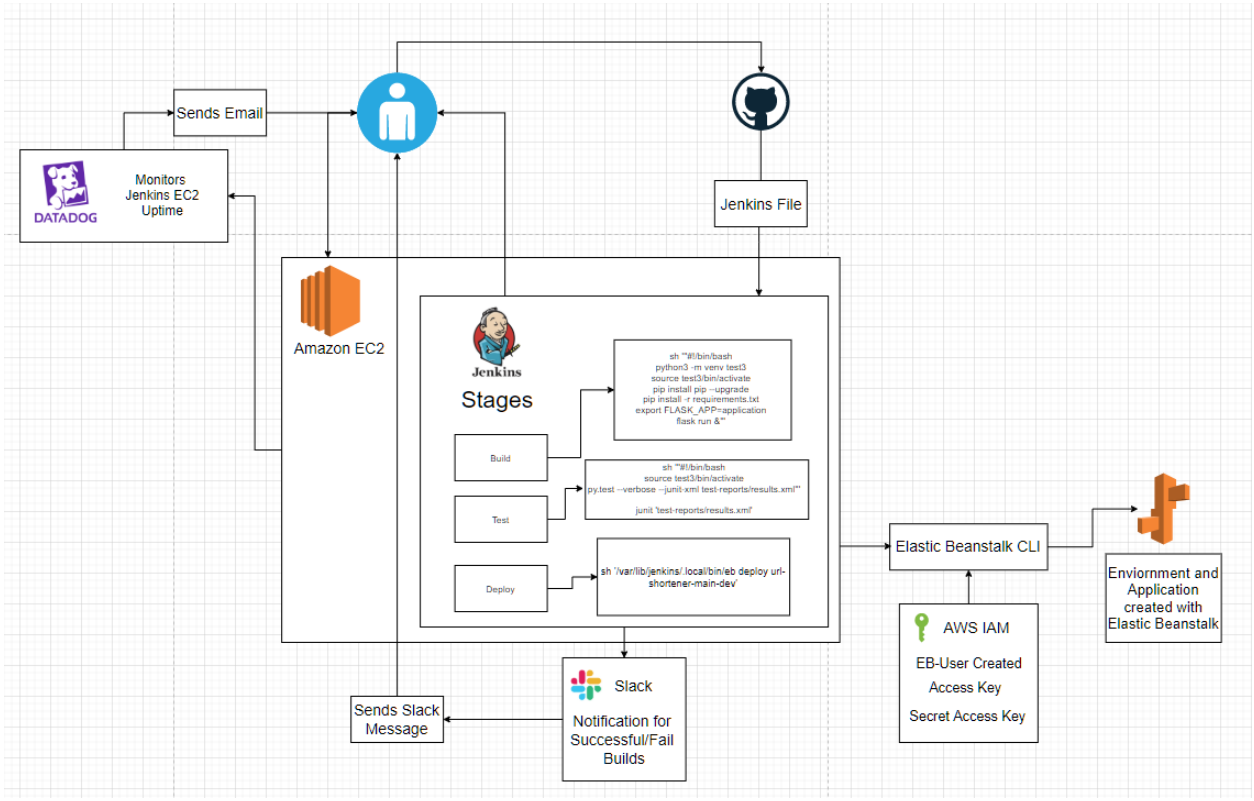
-I used DataDog to notify me via email if the EC2 running Jenkins was powered down or not responding.







Deployment Diagram:



Challenges:

Throughout this deployment there were 3 main stages where I got stuck on.

=Originally I could not get the path for my jenkins user to locate the elastic beanstalk cli download. This was solved once I created the ~/.profile, exported another path (export PATH=~/.local/bin:$PATH), and sourced the file.

-I ran into two problems in regards to my Jenkins pipeline. Initially I did not install Python's venv which didn't allow me to pass my test stage.

-The second problem that I ran into was the correct syntax for my Deploy stage.