Cloud portfolio

Preface

The application code, project report and pipeline configuration can also be found in the public repository on GitHub[1].

Introduction

For this assignment we were tasked to create a delivery pipeline using CI/CD to automating testing and deployment of our application code. We were to choose our desired tools and create the pipeline as well as creating a working infrastructure of code to test and build.

Goals

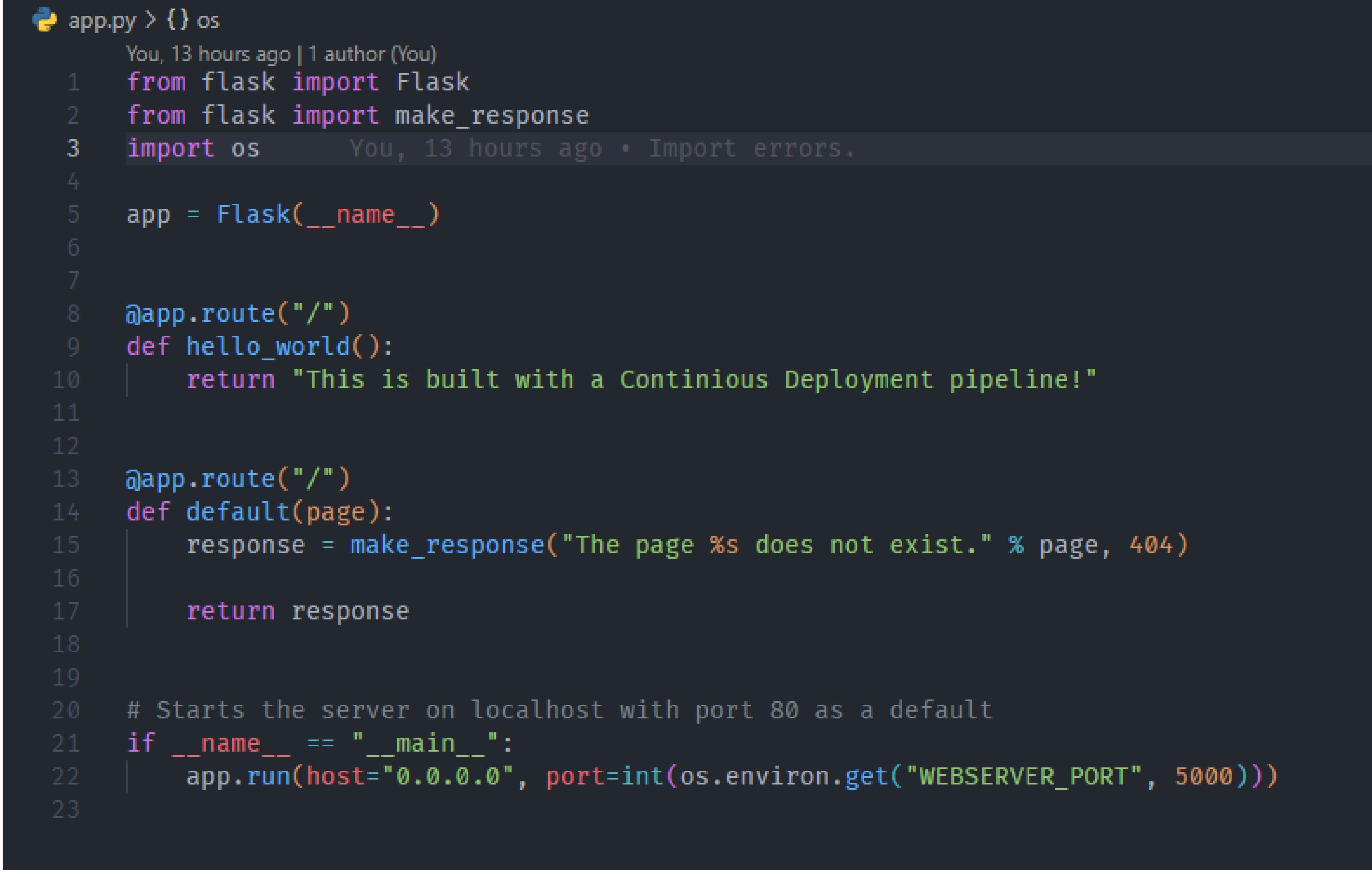
This report will showcase a possible approach to this and justify the decisions made by creating this type of pipeline as well as the underlying principles it is built on. The goal for this approach is to minimize the costs to stay as a free tier subscription on the cloud provider as well as incorporating relevant technologies to become familiarized. Additionally the intention was to create a Continuous deployment pipeline meaning the whole process of releasing changes into production is entirely automated removing any manual tasks related to this and being able to focus on the application code and its features.

Given that the focus was on developing a pipeline the application itself is relatively simple. It is comprised of a flask-webserver "Hello world!" application. The application and the tests is a modified version of the provided examples from the course textbook (Agarwal, 2021, Building a CI pipeline with GitHub Actions)[2].

Implementation

Python & Flask

As stated earlier the application is simple in nature to allow focus to be directed towards the Continuous deployment itself and remove any complexities from the application itself. The application is simply made up of a flask webserver returning a greeting message upon a visit to the hosted site.



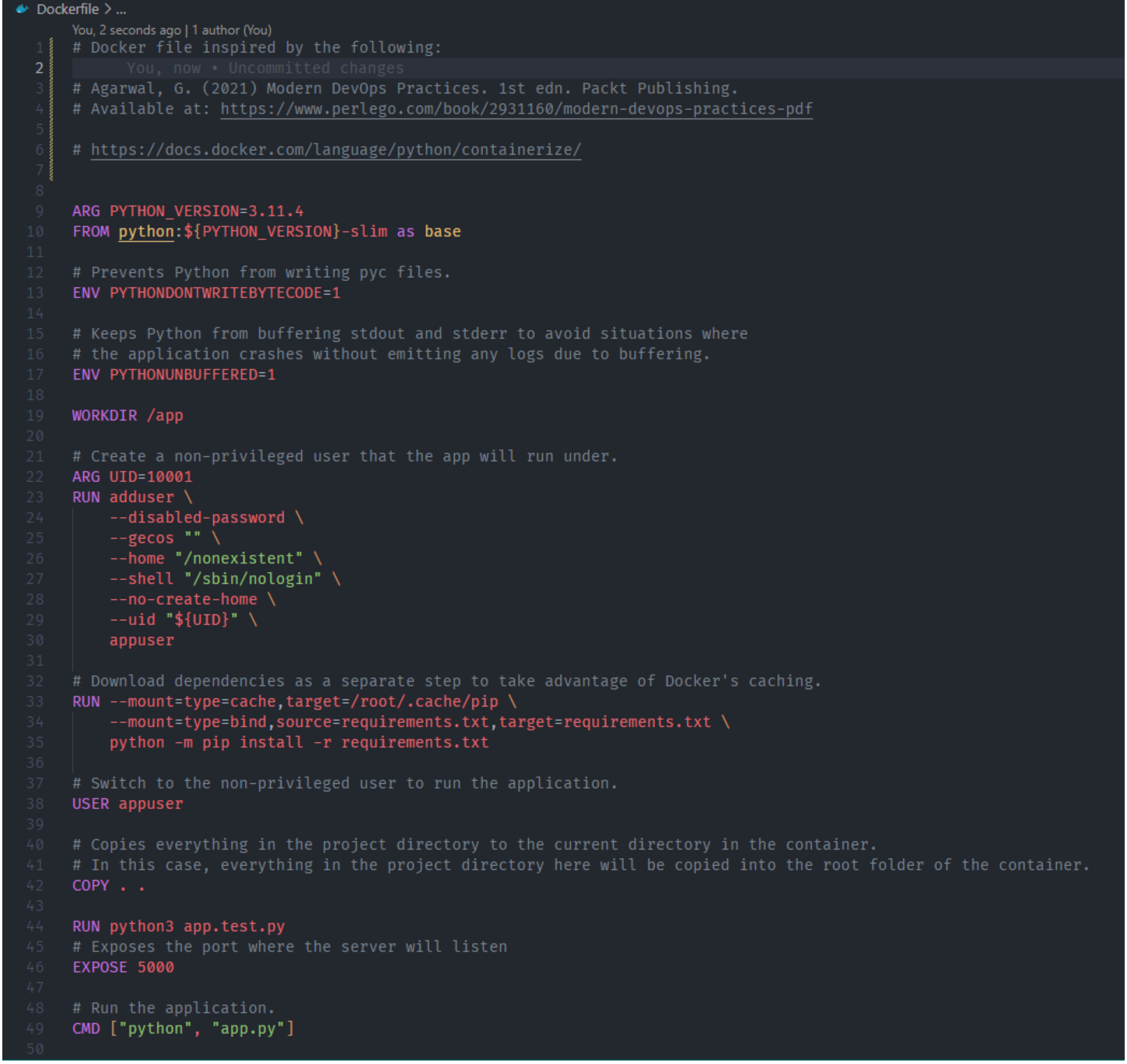
app.py > {} os

```python
You, 13 hours ago | 1 author (You)
from flask import Flask
from flask import make_response
import os       You, 13 hours ago • Import errors.

app = Flask(__name__)


@app.route("/")
def hello_world():
    return "This is built with a Continious Deployment pipeline!"


@app.route("/")
def default(page):
    response = make_response("The page %s does not exist." % page, 404)

    return response


# Starts the server on localhost with port 80 as a default
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=int(os.environ.get("WEBSERVER_PORT", 5000)))

```

Docker

This is encapsulated to run on a Docker container which has many advantages when developing an application. The first being that the container itself is CI compliant meaning the unit tests can easily be incorporated during the build process centring the relevant logic in the same place[2-1]. Having the application run in a Docker container is also a beneficial solution in the long-term as this can be scaled horizontally by creating multiple instances of this container giving consistency as well as being portable as they can be run on any system capable of running Docker.

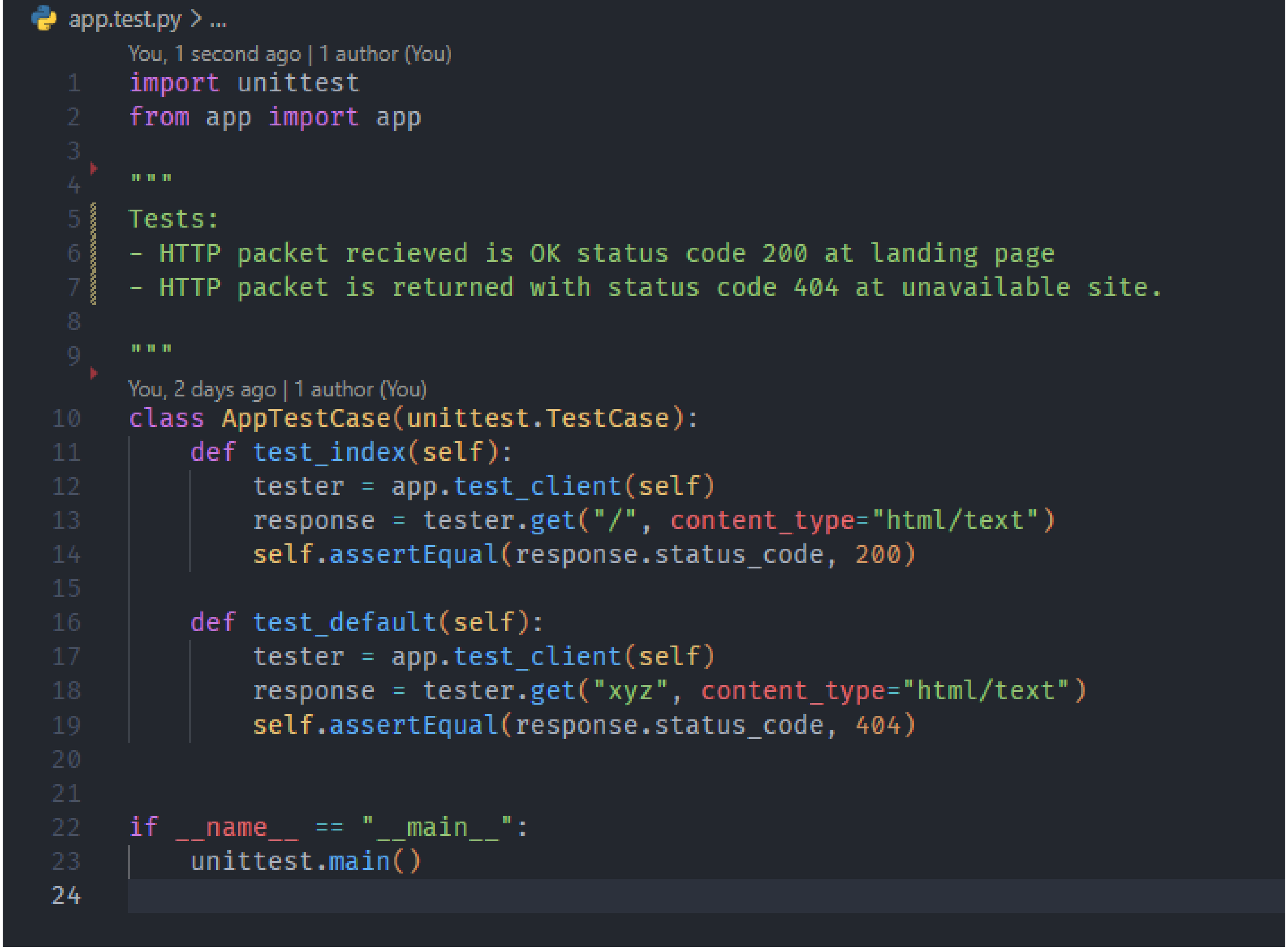
The properties of the container is specified in a Dockerfile which makes up the Image that defines how the container will operate and its dependencies .



```
Dockerfile > ...
    You, 2 seconds ago | 1 author (You)
 1  # Docker file inspired by the following:
 2        You, now • Uncommitted changes
 3  # Agarwal, G. (2021) Modern DevOps Practices. 1st edn. Packt Publishing.
 4  # Available at: https://www.perlego.com/book/2931160/modern-devops-practices-pdf
 5
 6  # https://docs.docker.com/language/python/containerize/
 7
 8
 9  ARG PYTHON_VERSION=3.11.4
10  FROM python:${PYTHON_VERSION}-slim as base
11
12  # Prevents Python from writing pyc files.
13  ENV PYTHONDONTWRITEBYTECODE=1
14
15  # Keeps Python from buffering stdout and stderr to avoid situations where
16  # the application crashes without emitting any logs due to buffering.
17  ENV PYTHONUNBUFFERED=1
18
19  WORKDIR /app
20
21  # Create a non-privileged user that the app will run under.
22  ARG UID=10001
23  RUN adduser \
24      --disabled-password \
25      --gecos "" \
26      --home "/nonexistent" \
27      --shell "/sbin/nologin" \
28      --no-create-home \
29      --uid "${UID}" \
30      appuser
31
32  # Download dependencies as a separate step to take advantage of Docker's caching.
33  RUN --mount=type=cache,target=/root/.cache/pip \
34      --mount=type=bind,source=requirements.txt,target=requirements.txt \
35      python -m pip install -r requirements.txt
36
37  # Switch to the non-privileged user to run the application.
38  USER appuser
39
40  # Copies everything in the project directory to the current directory in the container.
41  # In this case, everything in the project directory here will be copied into the root folder of the container.
42  COPY . .
43
44  RUN python3 app.test.py
45  # Exposes the port where the server will listen
46  EXPOSE 5000
47
48  # Run the application.
49  CMD ["python", "app.py"]
50
```

This Dockerfile is a modified version of the one provided in the course textbook[2-2]

As can be seen on line 44 of the provided image, this approach allows for integrating the in this case low-level tests in the outline for the container itself. The tests are basic as there is no application complexity it ensures that the landing page is available and that it will return correct status code upon an attempt to access non existent address.


app.test.py > ...
You, 1 second ago | 1 author (You)
You, 2 days ago | 1 author (You)

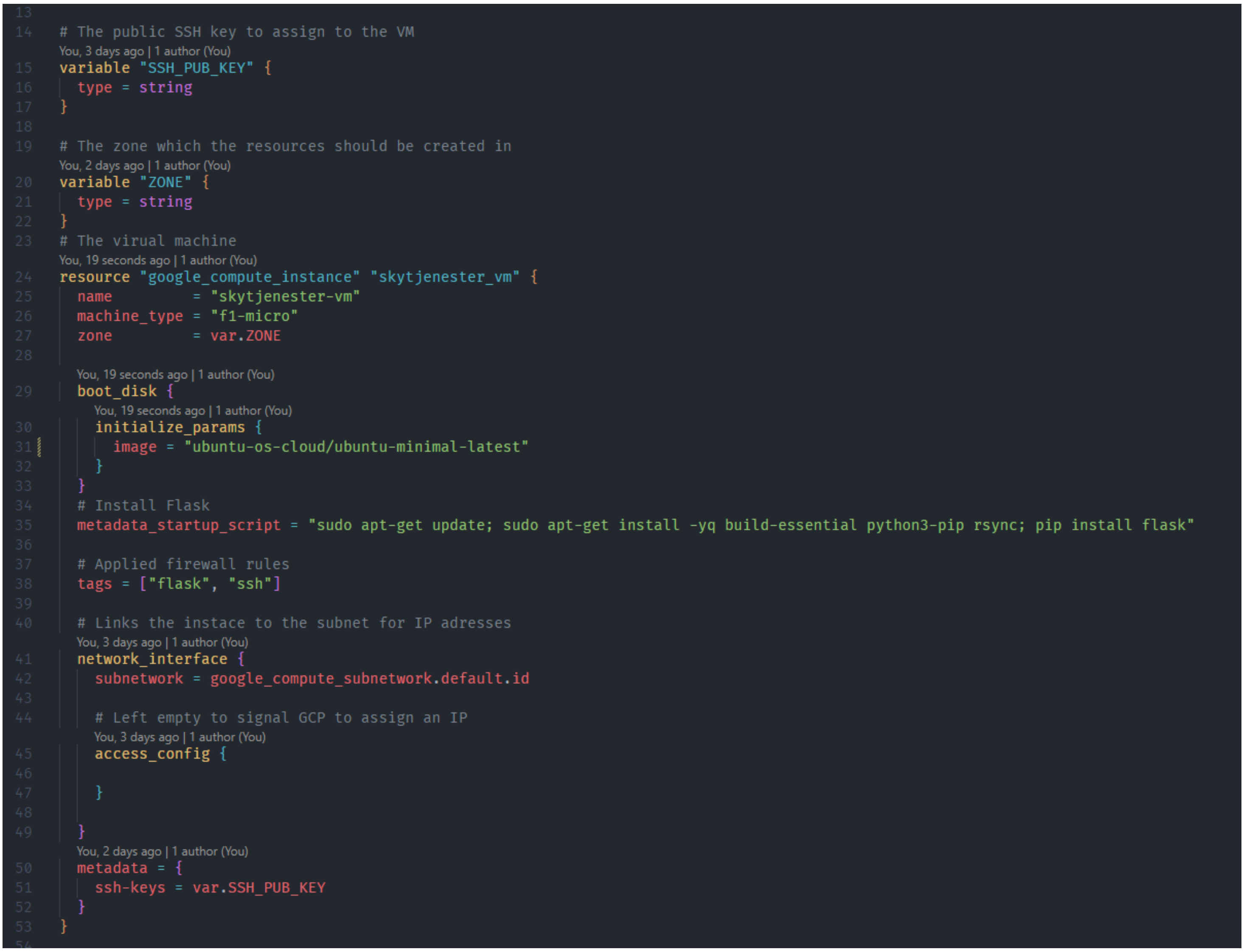
```python
import unittest
from app import app

"""
Tests:
- HTTP packet recieved is OK status code 200 at landing page
- HTTP packet is returned with status code 404 at unavailable site.

"""
class AppTestCase(unittest.TestCase):
    def test_index(self):
        tester = app.test_client(self)
        response = tester.get("/", content_type="html/text")
        self.assertEqual(response.status_code, 200)

    def test_default(self):
        tester = app.test_client(self)
        response = tester.get("xyz", content_type="html/text")
        self.assertEqual(response.status_code, 404)


if __name__ == "__main__":
    unittest.main()

```

The low-level tests consisting of unit tests

Terraform

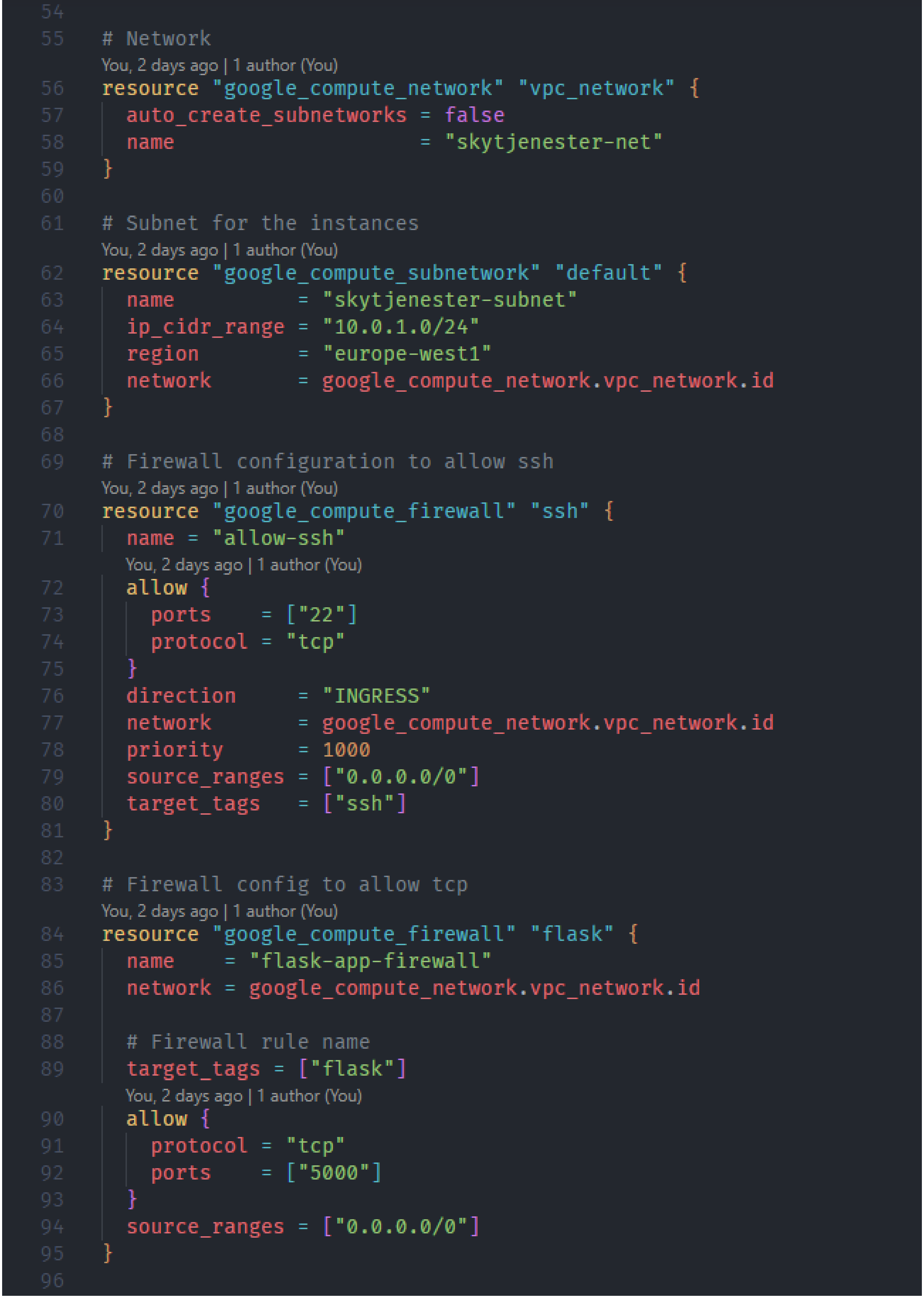
In order to host this application it needs the sufficient infrastructure to do so. This is where Terraform comes in as it defines the necessary resources that the application requires.

For starters it needs a virtual machine to run the container. Which in this case is a arbitrary Linux machine using the latest version to combat the risk of becoming outdated.



```
# The public SSH key to assign to the VM
You, 3 days ago | 1 author (You)
variable "SSH_PUB_KEY" {
  type = string
}

# The zone which the resources should be created in
You, 2 days ago | 1 author (You)
variable "ZONE" {
  type = string
}
# The virual machine
You, 19 seconds ago | 1 author (You)
resource "google_compute_instance" "skytjenester_vm" {
  name         = "skytjenester-vm"
  machine_type = "f1-micro"
  zone         = var.ZONE

  You, 19 seconds ago | 1 author (You)
  boot_disk {
    You, 19 seconds ago | 1 author (You)
    initialize_params {
      image = "ubuntu-os-cloud/ubuntu-minimal-latest"
    }
  }
  # Install Flask
  metadata_startup_script = "sudo apt-get update; sudo apt-get install -yq build-essential python3-pip rsync; pip install flask"

  # Applied firewall rules
  tags = ["flask", "ssh"]

  # Links the instace to the subnet for IP adresses
  You, 3 days ago | 1 author (You)
  network_interface {
    subnetwork = google_compute_subnetwork.default.id

    # Left empty to signal GCP to assign an IP
    You, 3 days ago | 1 author (You)
    access_config {

    }

  }
  You, 2 days ago | 1 author (You)
  metadata = {
    ssh-keys = var.SSH_PUB_KEY
  }
}
```

In order for the VM to be able to communicate publicly it needs to be connected to a network that allows for this. Therefore a network with firewalls allowing for both SSH to manually connect to the computer and TCP for listening at port 5000 to host the webserver . This is similar to the approach used in the Assignment 4 only that this makes it so it has eternal communication.

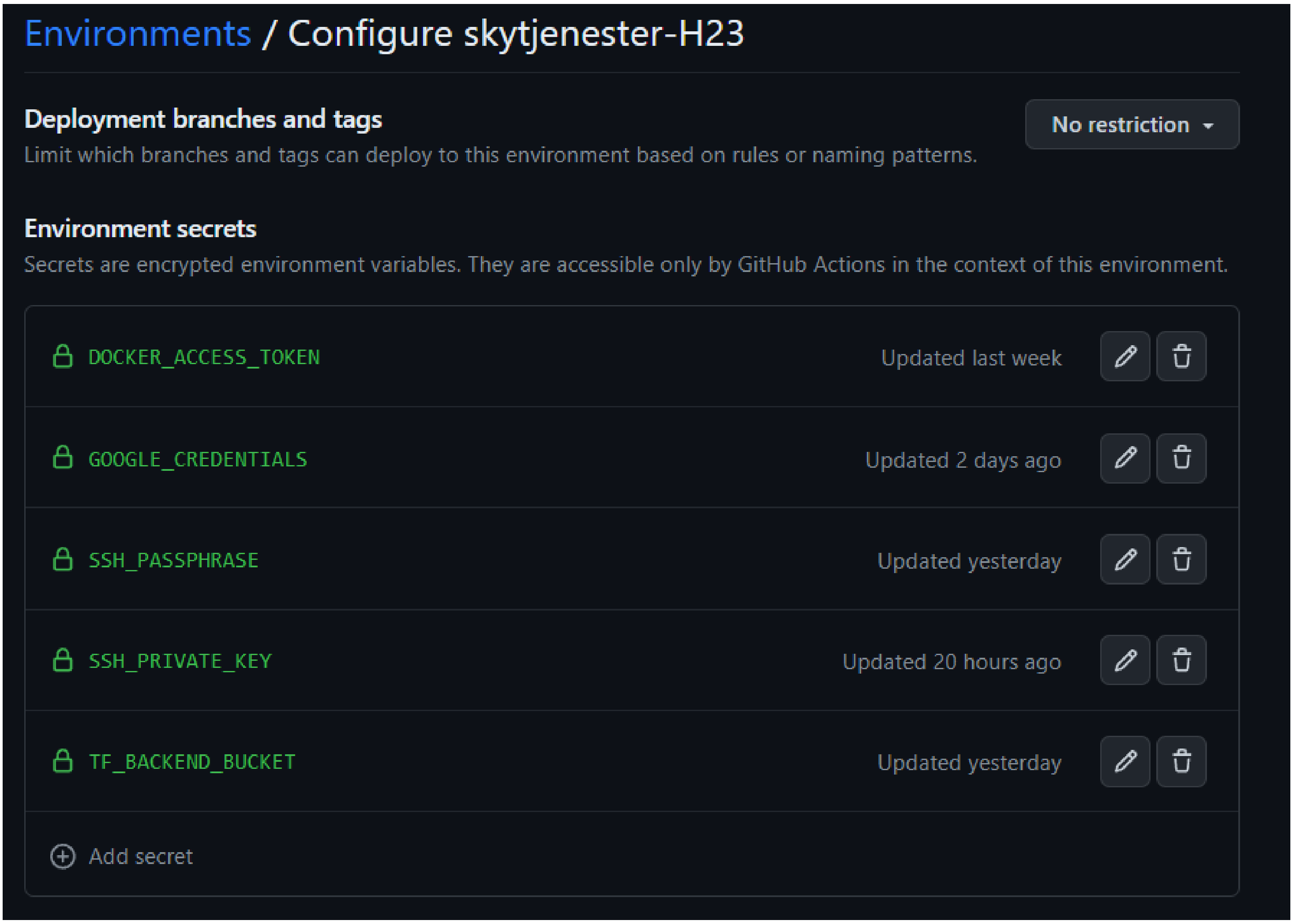


```
# Network
You, 2 days ago | 1 author (You)
resource "google_compute_network" "vpc_network" {
  auto_create_subnetworks = false
  name                    = "skytjenester-net"
}

# Subnet for the instances
You, 2 days ago | 1 author (You)
resource "google_compute_subnetwork" "default" {
  name          = "skytjenester-subnet"
  ip_cidr_range = "10.0.1.0/24"
  region        = "europe-west1"
  network       = google_compute_network.vpc_network.id
}

# Firewall configuration to allow ssh
You, 2 days ago | 1 author (You)
resource "google_compute_firewall" "ssh" {
  name = "allow-ssh"
  You, 2 days ago | 1 author (You)
  allow {
    ports    = ["22"]
    protocol = "tcp"
  }
  direction     = "INGRESS"
  network       = google_compute_network.vpc_network.id
  priority      = 1000
  source_ranges = ["0.0.0.0/0"]
  target_tags   = ["ssh"]
}

# Firewall config to allow tcp
You, 2 days ago | 1 author (You)
resource "google_compute_firewall" "flask" {
  name    = "flask-app-firewall"
  network = google_compute_network.vpc_network.id

  # Firewall rule name
  target_tags = ["flask"]
  You, 2 days ago | 1 author (You)
  allow {
    protocol = "tcp"
    ports    = ["5000"]
  }
  source_ranges = ["0.0.0.0/0"]
}
```

The tags specified are applied to the VM in the same script

GitHub Actions

All these components make up the pipeline which is orchestrated by the GitHub Actions which is specified in the workflow file. As GitHub actions only assigns a temporary machine which is not necessarily the same each time, creating an environment for the workflow is required to be able to execute the tasks independent of local environment.



Environments / Configure skytjenester-H23

Deployment branches and tags

Limit which branches and tags can deploy to this environment based on rules or naming patterns.

No restriction ▾

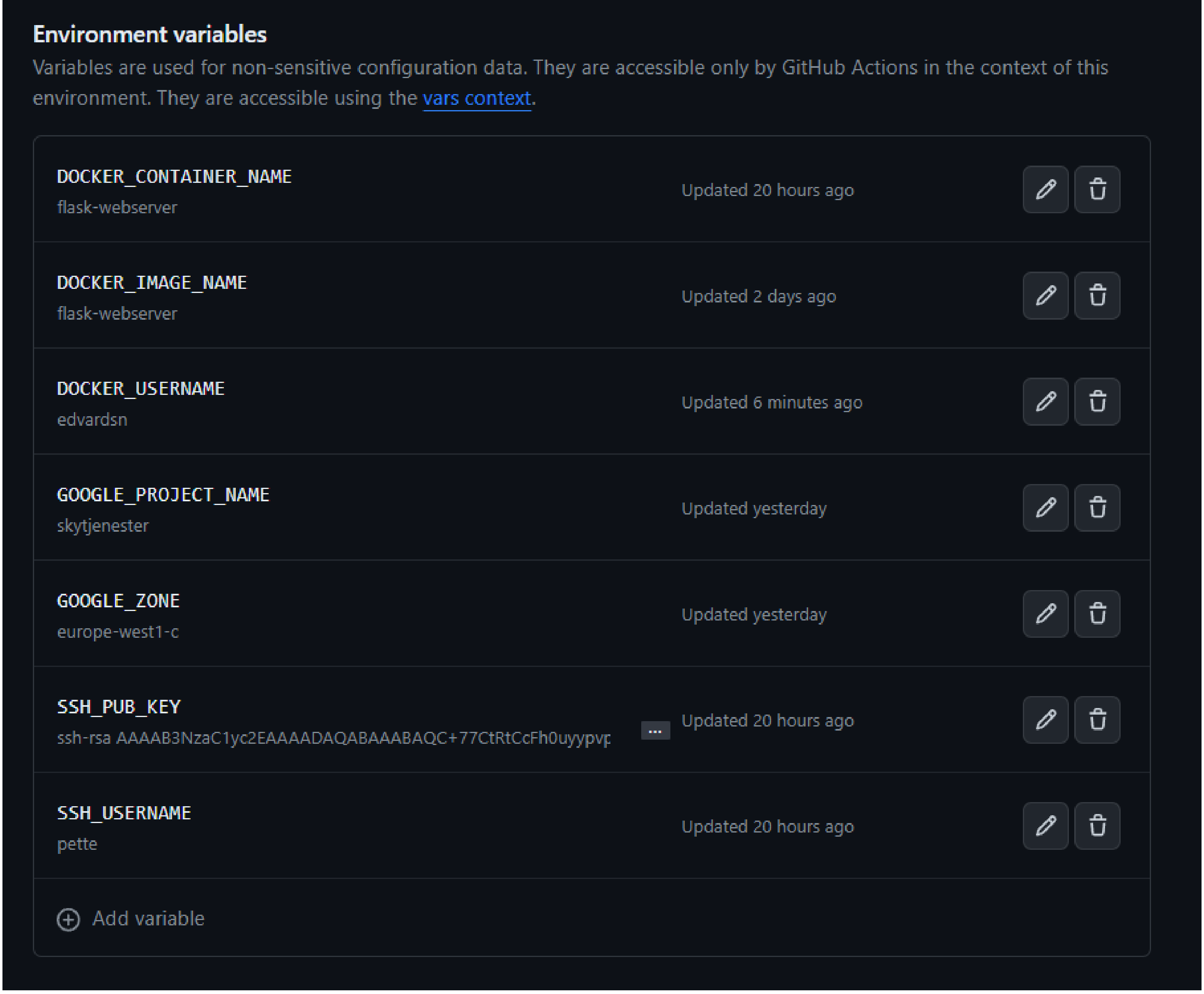
Environment secrets

Secrets are encrypted environment variables. They are accessible only by GitHub Actions in the context of this environment.

DOCKER_ACCESS_TOKEN	Updated last week
GOOGLE_CREDENTIALS	Updated 2 days ago
SSH_PASSPHRASE	Updated yesterday
SSH_PRIVATE_KEY	Updated 20 hours ago
TF_BACKEND_BUCKET	Updated yesterday

⊕ Add secret

Contains secret information such as authentication details etc. that needs to be encoded (Terraform bucket does not strictly speaking need to be secret)



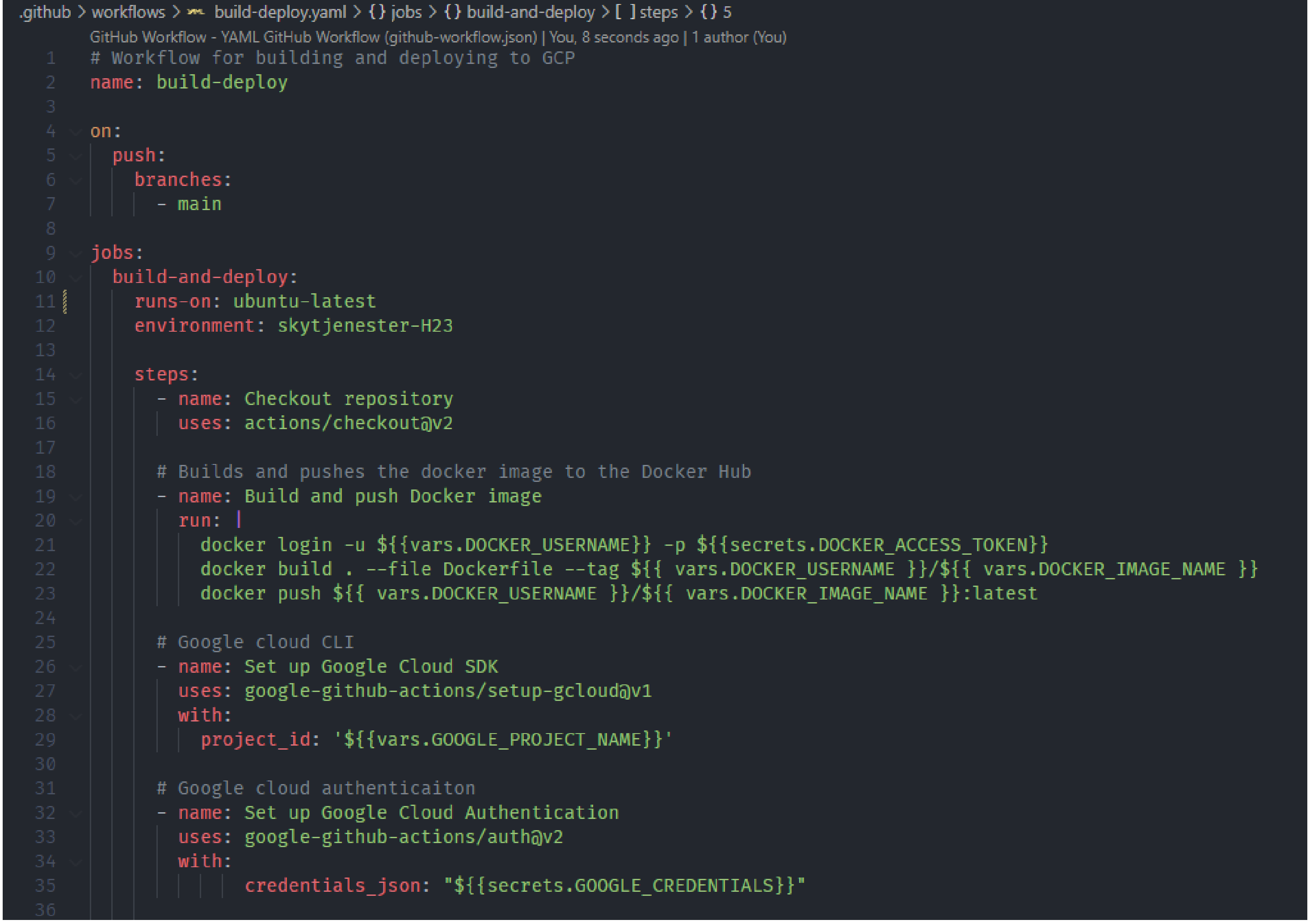
Environment variables

Variables are used for non-sensitive configuration data. They are accessible only by GitHub Actions in the context of this environment. They are accessible using the vars context.

DOCKER_CONTAINER_NAME flask-webserver	Updated 20 hours ago	
DOCKER_IMAGE_NAME flask-webserver	Updated 2 days ago	
DOCKER_USERNAME edvardsn	Updated 6 minutes ago	
GOOGLE_PROJECT_NAME skytjenester	Updated yesterday	
GOOGLE_ZONE europe-west1-c	Updated yesterday	
SSH_PUB_KEY ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQC+77CtRtCcFh0uyypvp	…	Updated 20 hours ago
SSH_USERNAME pette	Updated 20 hours ago	

Contains non-sensitive information

The first step is dedicated to testing and building the application. As mentioned earlier given the application size and complexity the low-level tests in the form of unit tests are conducted when building the image for the container that is to be ran. Assuming the tests are passed the Image will be pushed to Docker Hub with the new application code ready to be pulled.



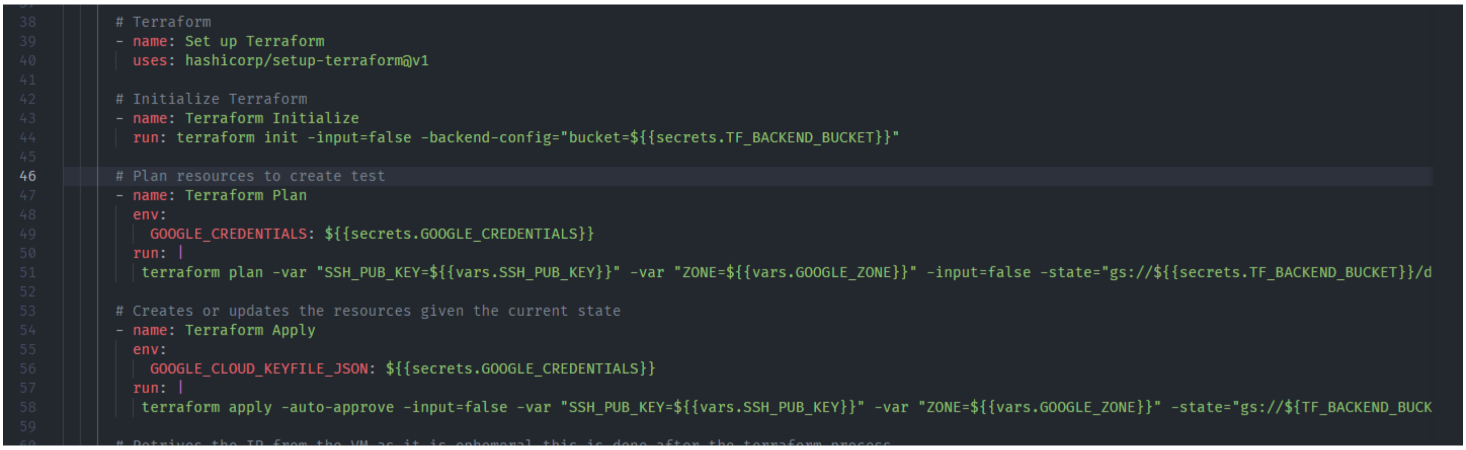
.github > workflows > build-deploy.yaml > {} jobs > {} build-and-deploy > [ ] steps > {} 5

GitHub Workflow - YAML GitHub Workflow (github-workflow.json) | You, 8 seconds ago | 1 author (You)

```
# Workflow for building and deploying to GCP
name: build-deploy

on:
  push:
    branches:
      - main

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    environment: skytjenester-H23

    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      # Builds and pushes the docker image to the Docker Hub
      - name: Build and push Docker image
        run: |
          docker login -u ${{vars.DOCKER_USERNAME}} -p ${{secrets.DOCKER_ACCESS_TOKEN}}
          docker build . --file Dockerfile --tag ${{ vars.DOCKER_USERNAME }}/${{ vars.DOCKER_IMAGE_NAME }}
          docker push ${{ vars.DOCKER_USERNAME }}/${{ vars.DOCKER_IMAGE_NAME }}:latest

      # Google cloud CLI
      - name: Set up Google Cloud SDK
        uses: google-github-actions/setup-gcloud@v1
        with:
          project_id: '${{vars.GOOGLE_PROJECT_NAME}}'

      # Google cloud authenticaiton
      - name: Set up Google Cloud Authentication
        uses: google-github-actions/auth@v2
        with:
            credentials_json: "${{secrets.GOOGLE_CREDENTIALS}}"

```

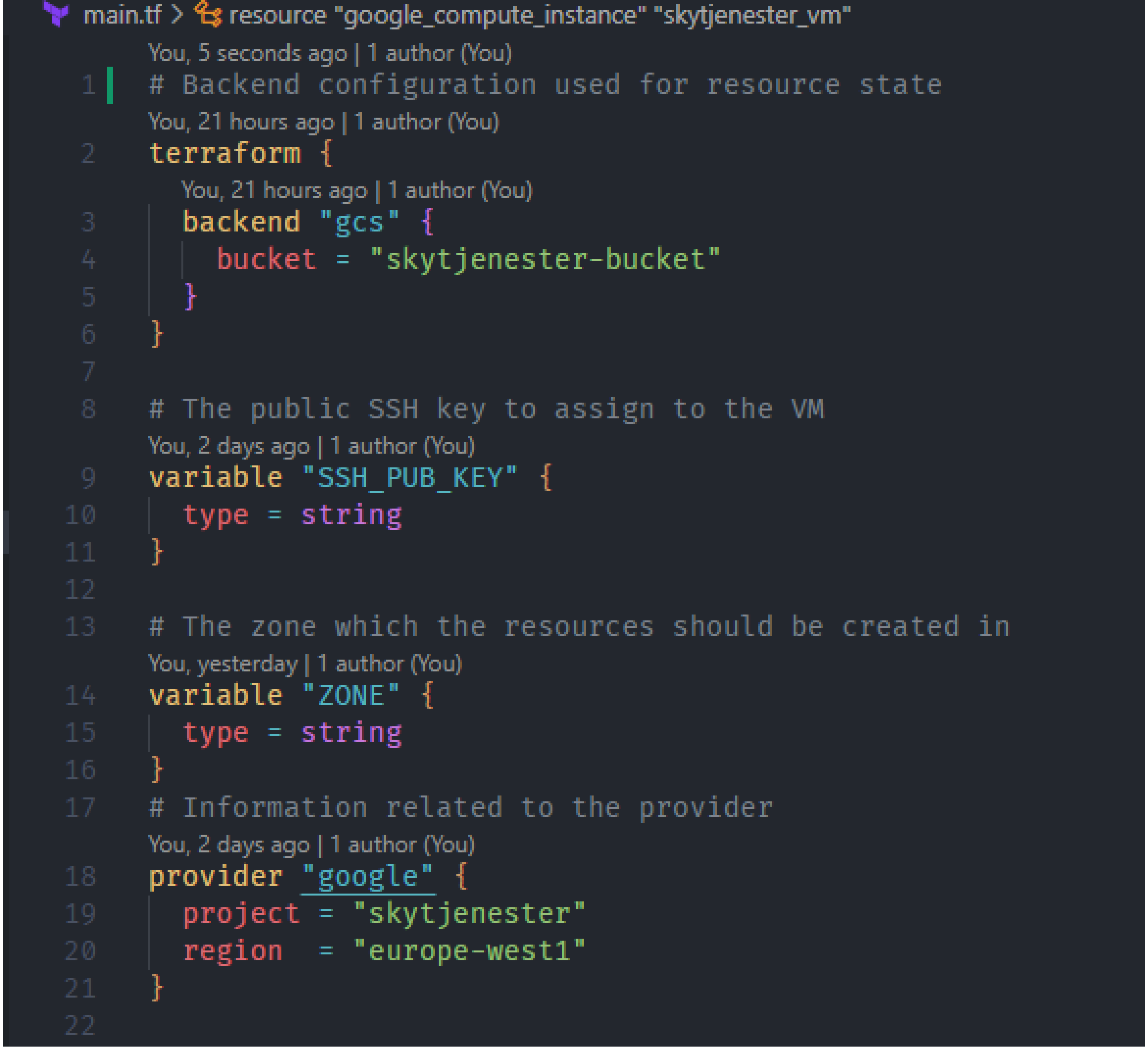
Additionally it extracts the necessary authentication for executing GCP commands for administering resources. Which for this approach is a JSON file manually saved as a secret.

Terraform



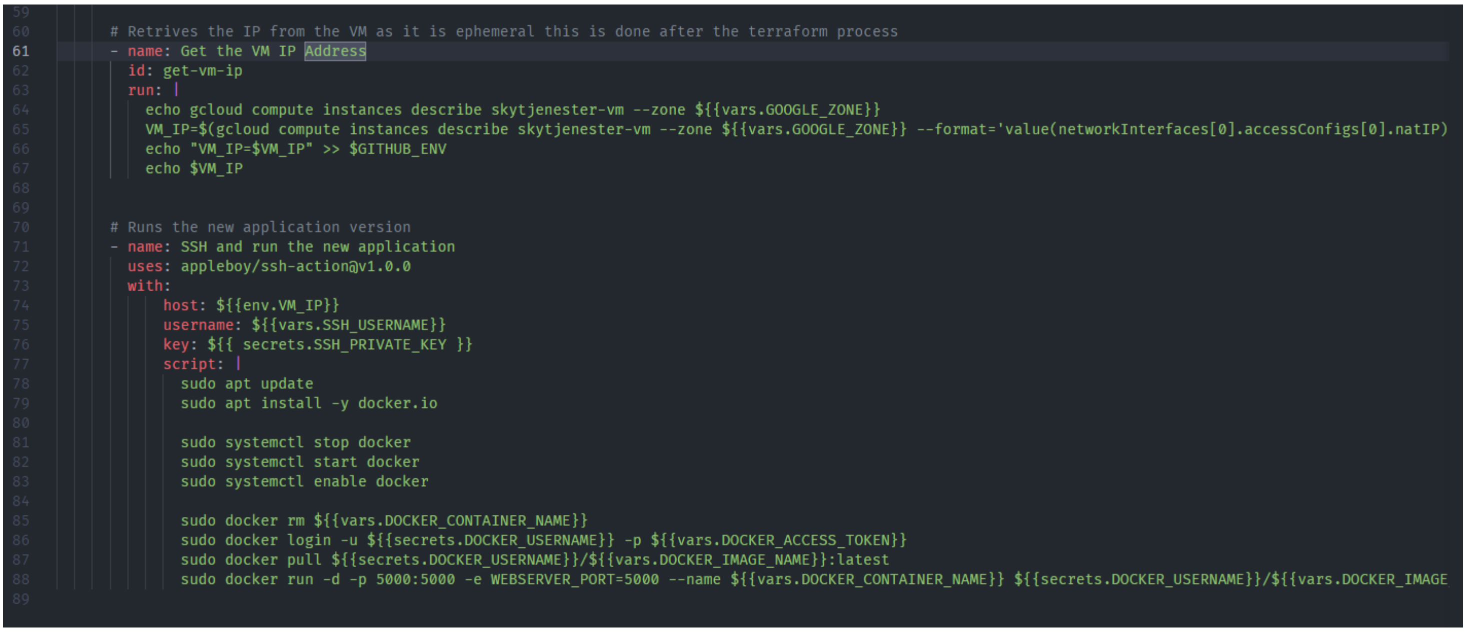
```
      # Terraform
      - name: Set up Terraform
        uses: hashicorp/setup-terraform@v1

      # Initialize Terraform
      - name: Terraform Initialize
        run: terraform init -input=false -backend-config="bucket=${{secrets.TF_BACKEND_BUCKET}}"

      # Plan resources to create test
      - name: Terraform Plan
        env:
          GOOGLE_CREDENTIALS: ${{secrets.GOOGLE_CREDENTIALS}}
        run: |
         terraform plan -var "SSH_PUB_KEY=${{vars.SSH_PUB_KEY}}" -var "ZONE=${{vars.GOOGLE_ZONE}}" -input=false -state="gs://${{secrets.TF_BACKEND_BUCKET}}/d

      # Creates or updates the resources given the current state
      - name: Terraform Apply
        env:
          GOOGLE_CLOUD_KEYFILE_JSON: ${{secrets.GOOGLE_CREDENTIALS}}
        run: |
         terraform apply -auto-approve -input=false -var "SSH_PUB_KEY=${{vars.SSH_PUB_KEY}}" -var "ZONE=${{vars.GOOGLE_ZONE}}" -state="gs://${TF_BACKEND_BUCK

      # Retrives the IP from the VM as it is ephemeral this is done after the terraform process
```

From there Terraform will check if the specified resources or rather the state of the resources are inline with the specified resources. As it will be a different environment each time the workflow is executed the state of the resources has to be stored remotely in a "Bucket" to ensure that Terraform can be run idempotent. The bucket is for this approach manually created and stored on GCP using its Cloud Storage API.


main.tf > resource "google_compute_instance" "skytjenester_vm"
You, 5 seconds ago | 1 author (You)
You, 21 hours ago | 1 author (You)
You, 21 hours ago | 1 author (You)
You, 2 days ago | 1 author (You)
You, yesterday | 1 author (You)
You, 2 days ago | 1 author (You)

```
# Backend configuration used for resource state
terraform {
  backend "gcs" {
    bucket = "skytjenester-bucket"
  }
}

# The public SSH key to assign to the VM
variable "SSH_PUB_KEY" {
  type = string
}

# The zone which the resources should be created in
variable "ZONE" {
  type = string
}
# Information related to the provider
provider "google" {
  project = "skytjenester"
  region  = "europe-west1"
}

```

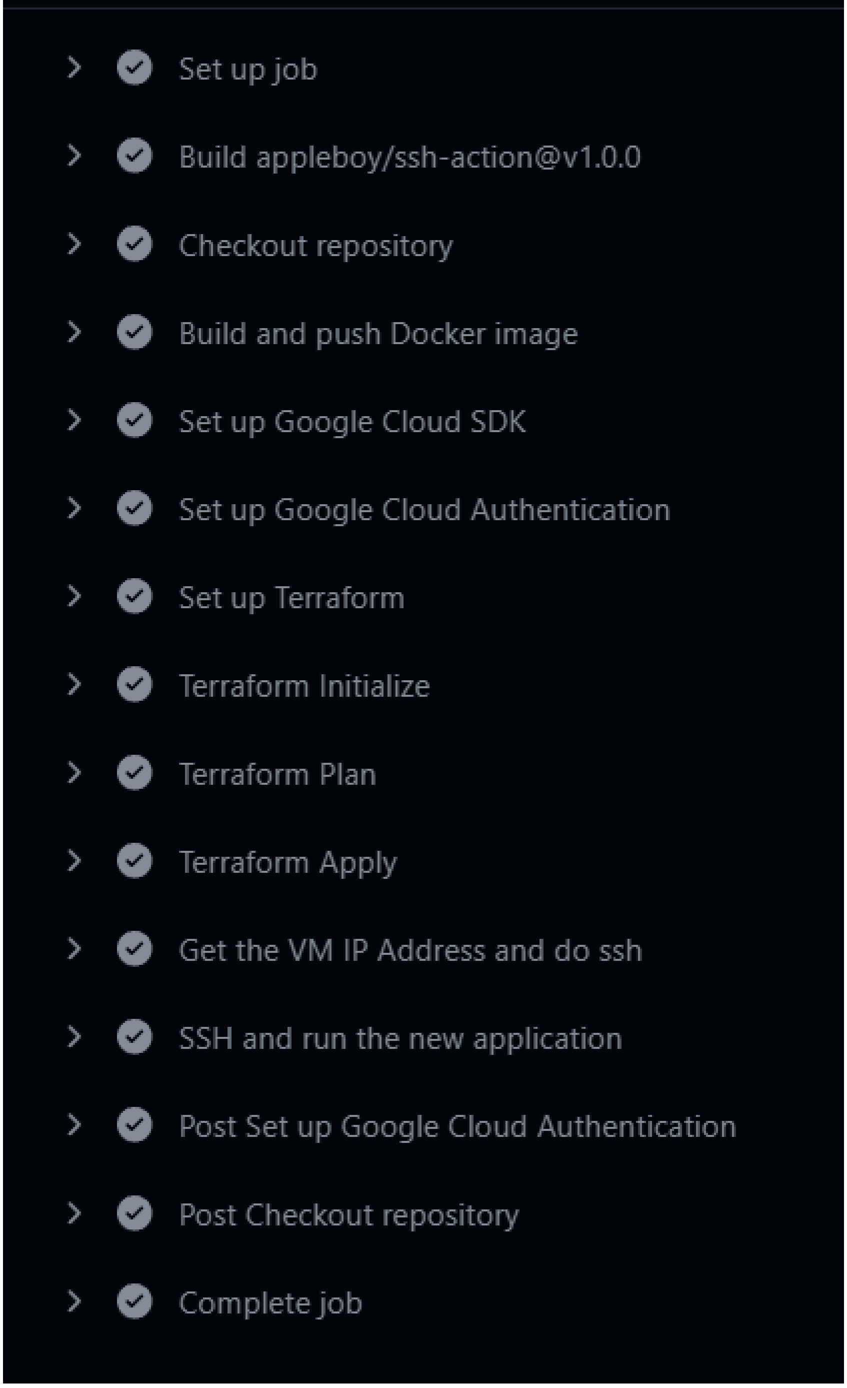
SSH



```

      # Retrives the IP from the VM as it is ephemeral this is done after the terraform process
      - name: Get the VM IP Address
        id: get-vm-ip
        run: |
          echo gcloud compute instances describe skytjenester-vm --zone ${{vars.GOOGLE_ZONE}}
          VM_IP=$(gcloud compute instances describe skytjenester-vm --zone ${{vars.GOOGLE_ZONE}} --format='value(networkInterfaces[0].accessConfigs[0].natIP)
          echo "VM_IP=$VM_IP" >> $GITHUB_ENV
          echo $VM_IP


      # Runs the new application version
      - name: SSH and run the new application
        uses: appleboy/ssh-action@v1.0.0
        with:
            host: ${{env.VM_IP}}
            username: ${{vars.SSH_USERNAME}}
            key: ${{ secrets.SSH_PRIVATE_KEY }}
            script: |
              sudo apt update
              sudo apt install -y docker.io

              sudo systemctl stop docker
              sudo systemctl start docker
              sudo systemctl enable docker

              sudo docker rm ${{vars.DOCKER_CONTAINER_NAME}}
              sudo docker login -u ${{secrets.DOCKER_USERNAME}} -p ${{vars.DOCKER_ACCESS_TOKEN}}
              sudo docker pull ${{secrets.DOCKER_USERNAME}}/${{vars.DOCKER_IMAGE_NAME}}:latest
              sudo docker run -d -p 5000:5000 -e WEBSERVER_PORT=5000 --name ${{vars.DOCKER_CONTAINER_NAME}} ${{secrets.DOCKER_USERNAME}}/${{vars.DOCKER_IMAGE

```

The final step of the workflow is retrieving the IP address from the possibly newly created VM as the external IPs are ephemeral in the free tier of GCP.

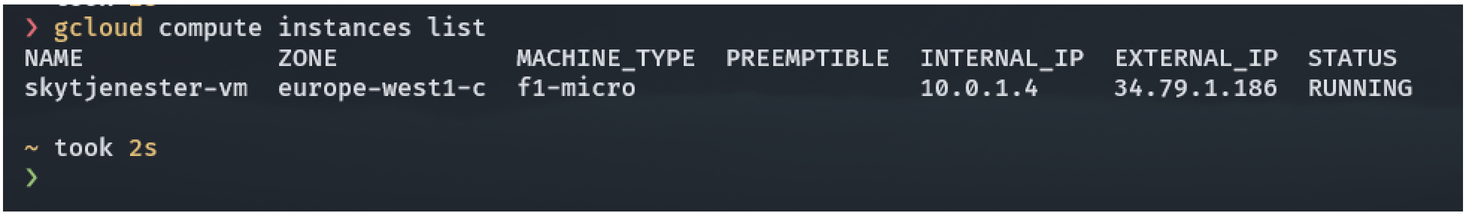
From there its simply to use the private SSH key belonging to the previously injected public SSH key to attach to the VM and restart Docker and clean up the previous iteration as well as pulling and starting the new version.



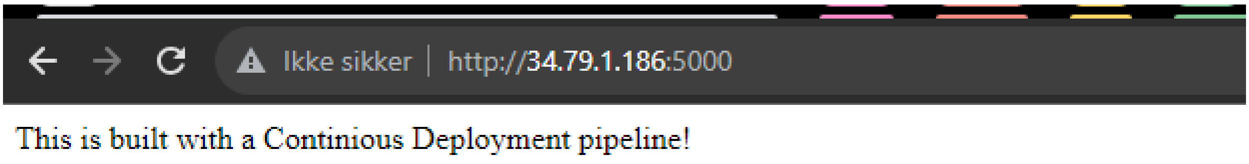
- Set up job
- Build appleboy/ssh-action@v1.0.0
- Checkout repository
- Build and push Docker image
- Set up Google Cloud SDK
- Set up Google Cloud Authentication
- Set up Terraform
- Terraform Initialize
- Terraform Plan
- Terraform Apply
- Get the VM IP Address and do ssh
- SSH and run the new application
- Post Set up Google Cloud Authentication
- Post Checkout repository
- Complete job

Timeline which showcases the steps taken in the pipeline

Showcase



```
❯ gcloud compute instances list
NAME            ZONE            MACHINE_TYPE  PREEMPTIBLE  INTERNAL_IP  EXTERNAL_IP  STATUS
skytjenester-vm europe-west1-c  f1-micro                   10.0.1.4     34.79.1.186  RUNNING

~ took 2s
❯
```



Ikke sikker | http://34.79.1.186:5000

This is built with a Continious Deployment pipeline!

Reflection

The are multiple areas of this approach which can be improved upon such as the deployment. Using SSH to deploy is not an ideal strategy as this can be complex if the number of instances increases. Deploying using Spinmaker and Kubernetes or similar technologies can strive to keep the deployment process more manageable and simply the process of incorporating more sophisticated deployment processes such as Blue/Green deployment but require more resource which are not optimal when trying to stay within the free tier (Agarwal, 2021, Continuous Deployment and Automation).

Another point of improvement is the manual configurations used to run this pipeline such as the GCP Bucket which ideally should be replaced with HashiCorp cloud storage or similar methods. Additionally it might be beneficial to incorporate a security vault as the number of necessary secrets could increase when scaled.

Lastly as the application is simplistic there are not much to test. As the complexity of the application increases incorporating both mid-level and high-level tests into the test suite and workflow will prove beneficial.

As a final note the pipeline manages to deploy new application code as well as be idempotent. This automates the manual process of testing, building and deploying new features of the application. This has been a time consuming process of configuring the pipeline but very educational and accomplishes the initial goals for the project.

References

---

[1] https://github.com/Edvardsn/InfrastructreAsCode↩

[2] Agarwal, G. (2021) Modern DevOps Practices. 1st edn. Packt Publishing. (Accessed: 5 December 2023).↩↩↩