**DevOps Assignment**

* **Using Terraform, build the infrastructure resources such as PVC, subnets, instances, elb, security group, etc., in the most optimal way with Production Grade security measures.**

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

The infrastructure Terraform can manage includes low-level components such as compute instances, storage, and networking, as well as high-level components such as DNS entries, SaaS features, etc.

**Infrastructure as Code**

Infrastructure is described using a high-level configuration syntax. This allows a blueprint of your datacenter to be versioned and treated as you would any other code. Additionally, infrastructure can be shared and re-used.

**Execution Plans**

Terraform has a "planning" step where it generates an execution plan. The execution plan shows what Terraform will do when you call apply. This lets you avoid any surprises when Terraform manipulates infrastructure.

**Resource Graph**

Terraform builds a graph of all your resources, and parallelism the creation and modification of any non-dependent resources. Because of this, Terraform builds infrastructure as efficiently as possible, and operators get insight into dependencies in their infrastructure.

**Change Automation**

Complex changesets can be applied to your infrastructure with minimal human interaction. With the previously mentioned execution plan and resource graph, you know exactly what Terraform will change and in what order, avoiding many possible human errors.

**Terraform Workflow**

The core Terraform workflow has three steps:

* Write - Author infrastructure as code.
* Plan - Preview changes before applying.
* Apply - Provision reproducible infrastructure.

**Step1: Prepare your Machine:**

IAM allows us to manage users, groups and roles and their level of access to the AWS Console. The IAM user must have the following permissions:

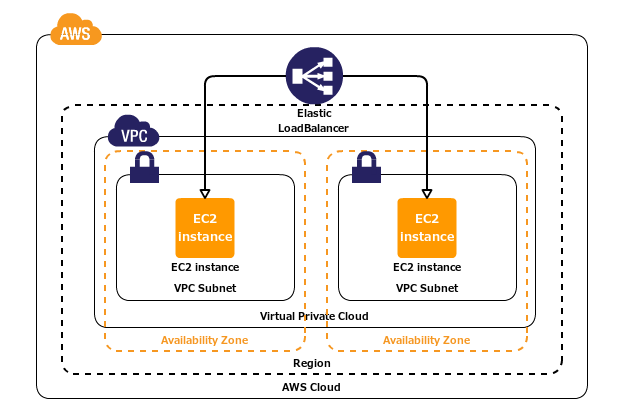
1. AmazonEC2FullAccess
2. AmazonRoute53FullAccess
3. AmazonS3FullAccess
4. IAMFullAccess
5. AmazonVPCFullAccess
6. AWSCertificateManagerFullAccess

**Step2: Prepare Terraform Scripts:**

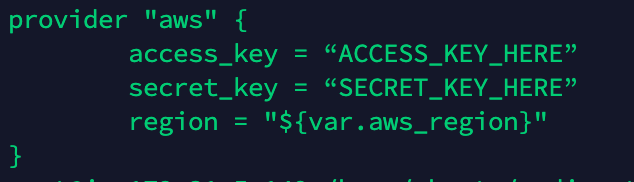
Terraform scans for all files with extensions \*.tf in the current folder and its subfolders recursively. It combines them all into a single file before executing it. In our example, we are using the following files:

* variables.tf is the representation of those variables in Terraform format
* vpc.tf is the actual script that provisions a VPC

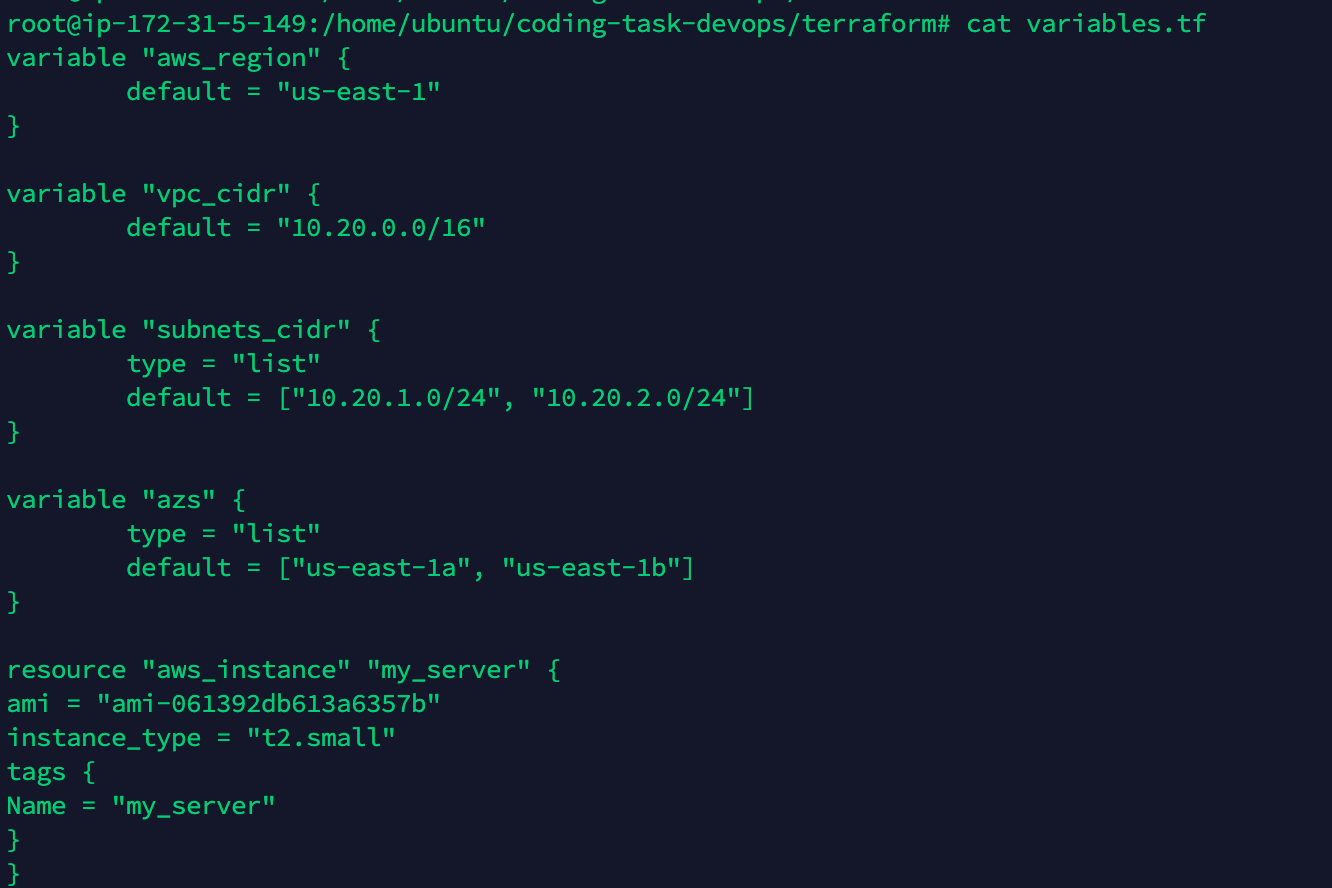
**Virtual Private Cloud**



**Provider Details:**



**Variables:**



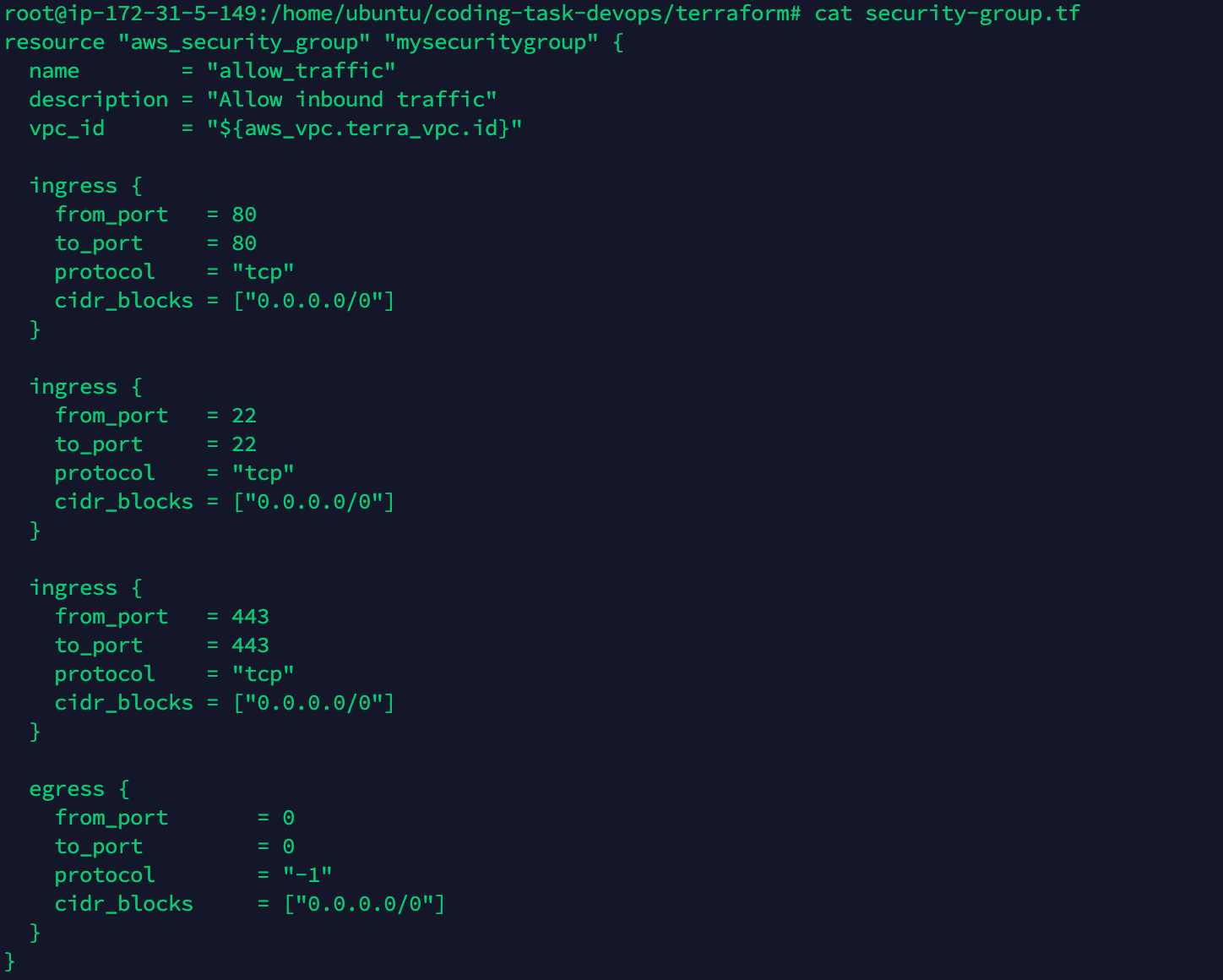
**VPC Details:**

In order for the resources in a VPC to send and receive traffic from the internet, the followings should be done:

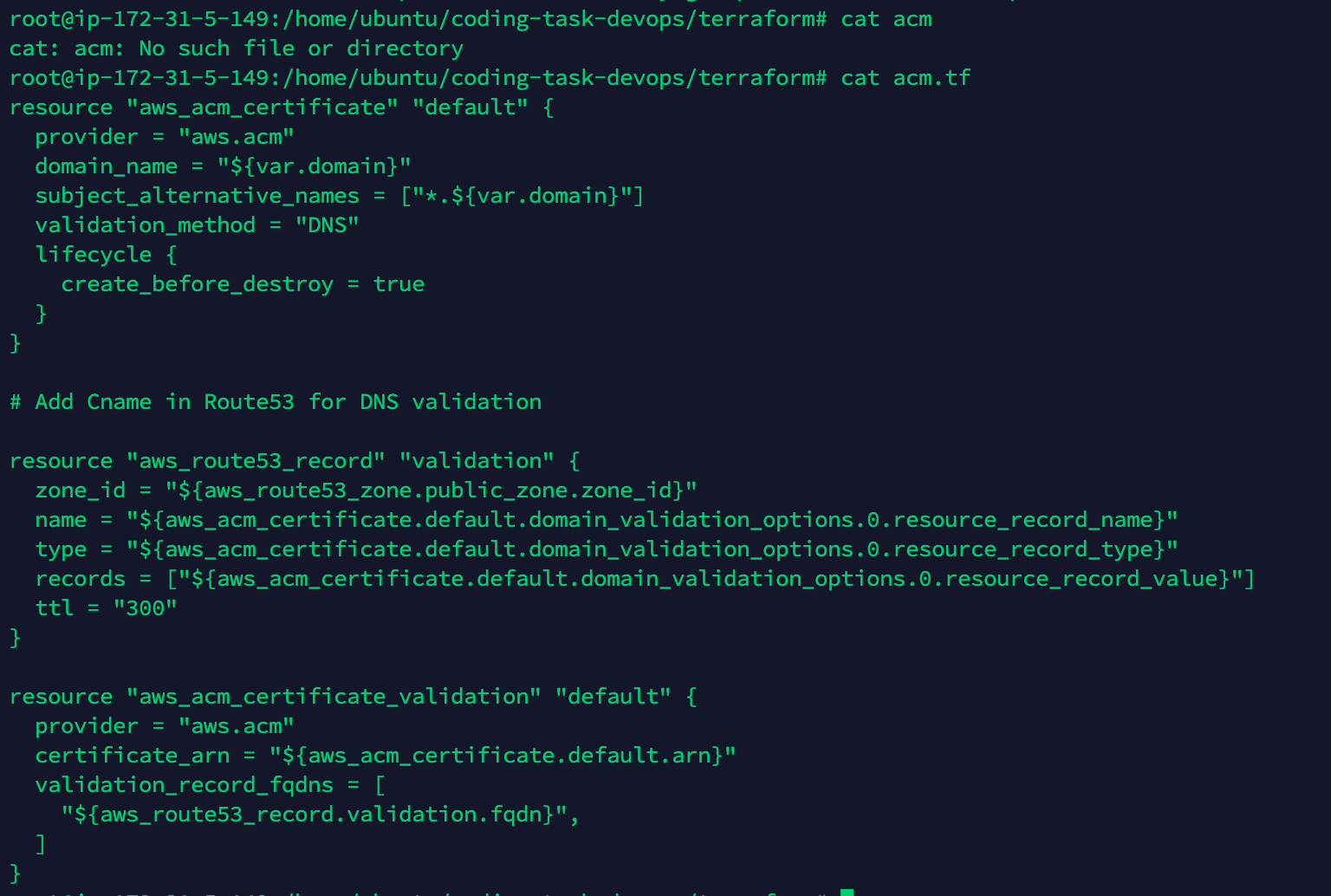
* An internet gateway must be attached to the VPC.
* The route tables associated with our public subnet (including custom route tables) must have a route to the internet gateway.



**Security Groups:**

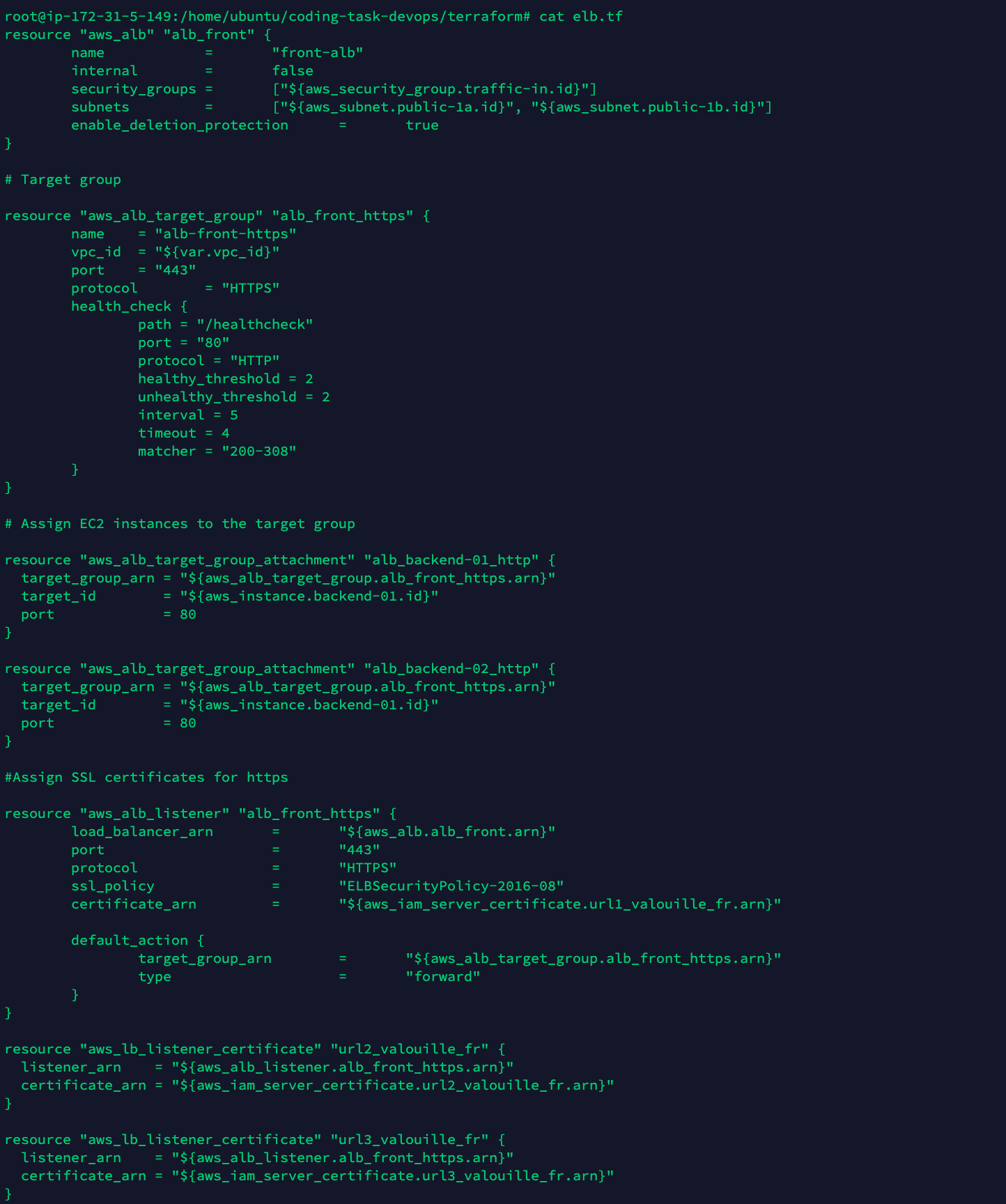


**AWS Certificate Manager:**

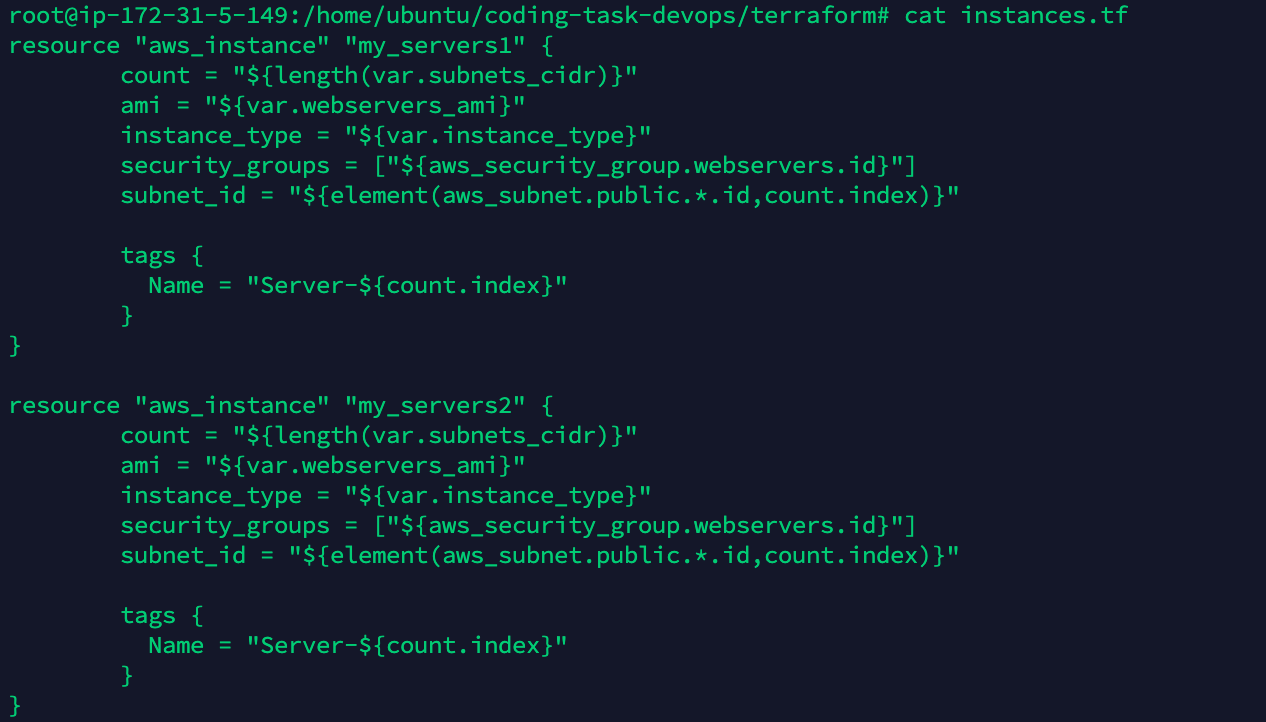


**Elastic Load Balancer:**

create one ALB and one target group, exposed over HTTPS and with two HTTP backends. For better security, it’s possible to also use the SSL certificates on the backends to do end-to-end encryption (from the ALB to the EC2 instances). We then create a rule to associate the target group to the ALB, and we attach the EC2 backend instances to the target group



**Instances:**



**Terraform init**

The first command to be used is 'terraform init'. This command downloads and installs plugins for providers used within the configuration.

**Terraform plan**

The second command to be used is 'terraform plan'. This command is used to see the changes that will take place on the infrastructure.

**Terraform apply**

Terraform apply command will create the resources on the AWS mentioned in the main.tf file. You will be prompted to provide your input to create the resources.

**Terraform destroy**

If you no longer require resources you created using the configuration mentioned in the \*.tf file. You can use the "terraform destroy" command to delete all those resources.

* **Build a Kubernetes Cluster in this VPC in an internal subnet, You can use any tool todo this.**

**Kubernetes Cluster Deployment with KOPS:**

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

In this section we are going to install a Kubernetes cluster on AWS using KOPS.

**Tools Stack:**

1. Terraform

* Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. Terraform can manage existing and popular service providers as well as custom in-house solutions.

1. KOPS

* KOPS helps us to create, destroy, upgrade and maintain production-grade, highly available, Kubernetes clusters from the command line.
* Fully automated installation
* Uses DNS to identify clusters
* Self-healing: everything runs in Auto-Scaling Groups
* Multiple OS support (Debian, Ubuntu 16.04 supported, CentOS & RHEL, Amazon Linux and CoreOS)
* High-Availability support
* Can directly provision, or generate terraform manifests

1. Kubectl

* Kubectl is a tool to interact with K8S cluster

1. S3 Buckets

* Terraform and KOPS need S3 buckets as back-end to store their states.

1. Route53

* Route53 Domain for the K8S cluster

So, what we are trying to achieve here is to create AWS related infra VPC, S3, Security Groups, ELB, Subnets etc using Terraform and we will use KOPS to launch EC2 instances for K8S Master and Nodes to set up a K8S cluster.

**AWS infra related steps are described in detail**  
In this part of the guide we will be more focused on the KOPS related tasks and steps.

1. Install Kops
2. Create Route53 domain
3. Create S3 Bucket to store the cluster state
4. Building cluster configuration
5. Create the cluster in AWS
6. Cleanup

**Install KOPS (For Linux):**

KOPS can be downloaded the latest release with the command:

curl -LO https://github.com/kubernetes/kops/releases/download/**$(**curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag\_name | cut -d '"' -f 4**)**/kops-linux-amd64

To download a specific version of kops, replace the following portion of the command with the specific kops version.

**$(**curl -s https://api.github.com/repos/kubernetes/kops/releases/latest | grep tag\_name | cut -d '"' -f 4**)**

Download kops version v1.15.0 type:

curl -LO https://github.com/kubernetes/kops/releases/download/1.15.0/kops-linux-amd64

Make the kops binary executable

chmod +x kops-linux-amd64

Move the kops binary into your PATH.

sudo mv kops-linux-amd64 /usr/local/bin/kops

You can also install kops using Homebrew.

brew update && brew install kops

**Create a Route53 Domain for the cluster:**

kops uses DNS for discovery, both inside the cluster and outside, so that you can reach the kubernetes API server from clients.

kops has a strong opinion on the cluster name: it should be a valid DNS name. By doing so you will no longer get your clusters confused, you can share clusters with your colleagues unambiguously, and you can reach them without relying on remembering an IP address.

You can, and probably should, use subdomains to divide your clusters. As our example we will use **dev.test.com**. The API server endpoint will then be **api.dev.test.com**.

A Route53 hosted zone can serve subdomains. Your hosted zone could be **dev.test.com**, but also **dev.test.com** or even **test.com**. kops works with any of these, so typically you choose for organization reasons.

Let's assume you're using dev.test.com as your hosted zone. You create that hosted zone using the normal process, or with a command such as aws **route53 create-hosted-zone --name dev.test.com --caller-reference 1**.

You must then set up your NS records in the parent domain, so that records in the domain will resolve. Here, you would create NS records in **test.com** for **dev.** If it is a root domain name you would configure the NS records at your domain registrar This step is easy to mess up.

You should see the 4 NS records that Route53 assigned your hosted zone.

**Create an S3 Bucket to store your clusters state:**

KOPS lets you manage your clusters even after installation. To do this, it must keep track of the clusters that you have created, along with their configuration, the keys they are using etc. This information is stored in an S3 bucket. S3 permissions are used to control access to the bucket.

Multiple clusters can use the same S3 bucket, and you can share an S3 bucket between your colleagues that administer the same clusters - this is much easier than passing around kubecfg files. But anyone with access to the S3 bucket will have administrative access to all your clusters, so you don't want to share it beyond the operations team.

So typically you have one S3 bucket for each ops team

In our example, we chose **dev.test.com** as our hosted zone, so let's pick **clusters.test.com** as the S3 bucket name.

* Export AWS\_PROFILE (if you need to select a profile for the AWS CLI to work)
* Create the S3 bucket using aws s3 mb **s3://clusters.test.com**
* You can export **KOPS\_STATE\_STORE=s3://clusters.dev.example.com** and then kops will use this location by default. We suggest putting this in your bash\_profile or similar.

As VPC and other AWS resources have already been created in the terraform section we are ready to go now to create k8s cluster on top of it with Kops.

**Configure KOPS state:**

export KOPS\_STATE\_STORE=s3://kops-state.${DOMAIN}

**KOPS Create Cluster:**

This command will only prepare configuration of the cluster and store it in the s3 bucket we specified via KOPS\_STATE\_STORE env variable.

kops create cluster \

--vpc=$(terraform output vpc\_id) \

--master-zones=$(terraform output -json networks | jq -r '.[].availability\_zone' | paste -sd, -) \

--zones=$(terraform output -json networks | jq -r '.[].availability\_zone' | paste -sd, -) \

--subnets=$(terraform output -json subnet\_ids | jq -r 'join(",")') \

--networking=calico \

--node-count=3 \

--master-size=t2.medium \

--node-size=t2.medium \

--dns-zone=${PROJECT\_NAME}.${DOMAIN} \

--dns=private \

--name=${PROJECT\_NAME}.${DOMAIN}

“dns=private” is set in order to disable DNS resolution verification as we don’t have a real domain name.

As you can see vpc, zones, master-zones, subnets come from terraform output because VPC and Subnets already exist.

Also we don’t need to set “target=terraform” flag that doesn’t make sense as it would create additional Terraform configuration that would require additional “Terraform apply” and state and so forth.

Review Kops cluster configuration:

kops edit cluster --name ${PROJECT\_NAME}.${DOMAIN}

**Apply Kops configuration:**

This will deploy k8s cluster:

kops update cluster --name ${PROJECT\_NAME}.${DOMAIN} --yes

Check the cluster (will fail):

kubectl cluster-info

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

Unable to connect to the server: dial tcp: lookup api.dev.your.domain on 8.8.8.8:53: no such host

The error arose because the hosted zone that contains “api.test.com” DNS record is private and the domain name cannot be resolved.

In order to fix it get api endpoint ip from route53 in AWS console:

Fix it via hosts file:

sudo bash -c ‘echo “34.205.156.35 api.test.com” >> /etc/hosts’

Check cluster again:

kubectl cluster-info

Kubernetes master is running at<https://api.test.com>

KubeDNS is running at [https://api.test.com/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy](https://api.dev.your.domain/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy)

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.

Check the nodes:

kubectl get nodes

NAME STATUS ROLES AGE VERSION

ip-10-0-0-156.ec2.internal Ready master 14m v1.18.0

ip-10-0-1-234.ec2.internal Ready master 14m v1.18.0

ip-10-0-2-157.ec2.internal Ready master 14m v1.18.0

### Explore other add-ons

See the [list of add-ons](https://kubernetes.io/docs/concepts/cluster-administration/addons/) to explore other add-ons, including tools for logging, monitoring, network policy, visualization, and control of your Kubernetes cluster.

## Cleanup

*kops* delete cluster --name ${PROJECT\_NAME}.${DOMAIN} --yes

terraform destroy -var "project\_name=${PROJECT\_NAME}" -var "domain=${DOMAIN}"

aws s3 rb s3://terraform-state.${DOMAIN} --force

aws s3 rb s3://kops-state.${DOMAIN} --force

* **Build the Docker image using GitHub actions / CircleCI / Drone CI etc.**

# **GitHub Actions**

Automate, customize, and execute your software development workflows right in your repository with GitHub Actions. You can discover, create, and share actions to perform any job you'd like, including CI/CD, and combine actions in a completely customized workflow.

### **Workflow syntax**

The workflow file is written in YAML. In the YAML workflow file, you can use expression syntax to evaluate contextual information, literals, operators, and functions. Contextual information includes workflow, environment variables, secrets, and the events that triggered the workflow. When you use run in a workflow step to run shell commands, you can use specific workflow command syntax to set environment variables, set output parameters for subsequent steps, and set error or debug messages.

### **Events**

You can configure workflows to run when specific GitHub events occur, at a scheduled time, manually, or when events outside of GitHub occur.

### **Authentication and secrets**

GitHub provides a token that you can use to authenticate on behalf of GitHub Actions. You can also store sensitive information as a secret in your organization or repository. GitHub encrypts all secrets.

### **GitHub-hosted runners**

GitHub offers hosted virtual machines to run workflows. The virtual machine contains an environment with tools, packages, and environment variables for GitHub Actions to use.

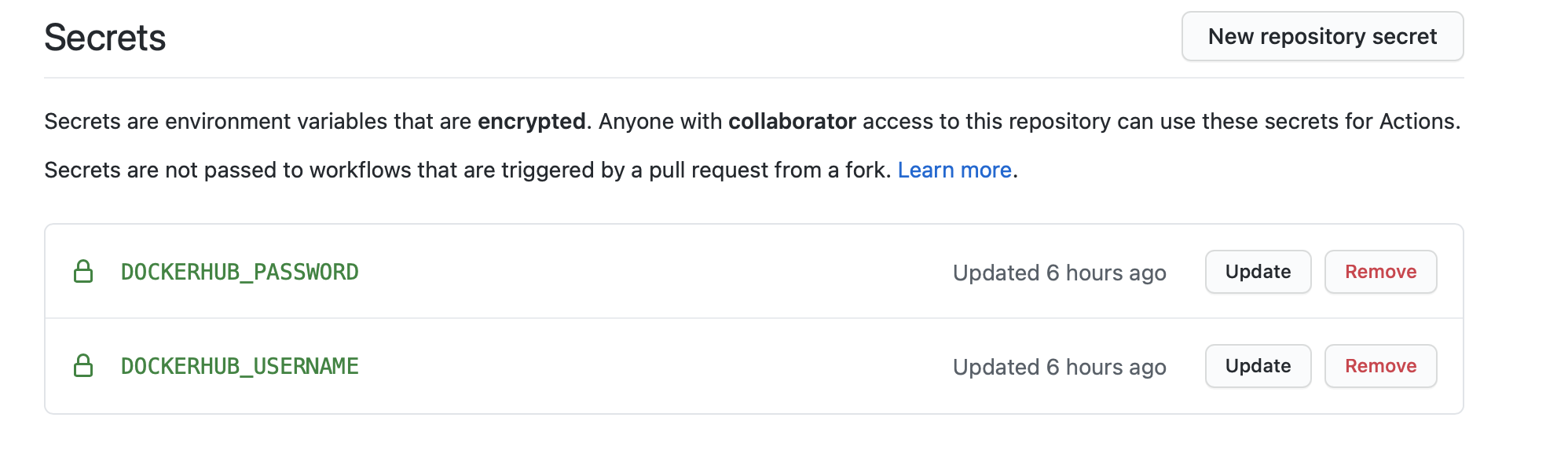
### **Administration**

When you run workflows on GitHub-hosted runners, there are usage limits and potential usage charges. You can also disable or restrict the usage of GitHub Actions in a repository and organization.

**Details**

The first thing we need to do is to ensure that we will be able to access Docker Hub from any workflow created, to do this we need to add DockerID and a Personal Access Token (PAT) as secrets into GitHub.

Add access token and username as secrets into the GitHub “Secrets UI” :



We can now start to set up our action workflow to build and store our images in docker hub. In this CI flow we are using two Docker actions, the first allows us to log in to Docker Hub using secrets stored in our GitHub Repository.

The second is the build and push action, in this we are setting the push flag to true (as I want to push!) and adding in my tag simply to always go to the latest.



* **Deploy the Ingress controller such as Traefik / Kong and configure the Ingress accordingly.**

The Kubernetes Ingress Controller. The Traefik Kubernetes Ingress provider is a Kubernetes Ingress controller; that is to say, it manages access to cluster services by supporting the Ingress specification.

The main purpose of Ingress is to expose HTTP and HTTPS routes from outside the cluster to services running in that cluster. This is the same as to say that Ingress controls how the external traffic is routed to the cluster.

We have several microservices (small applications communicating with each other) in our Kubernetes cluster. These services can be accessed from within the cluster, but we might also want our users to access them from outside the cluster as well. What we therefore need to do is to associate each HTTP(S) route with the corresponding backend using the reverse proxy and load balance between different instances of this service. At the same time, given the ever-changing nature of Kubernetes, we would want to track changes in service backends to be able to re-associate those HTTP routes to new Pod instances when new Pods are added or removed.

# **Ingress Resource**

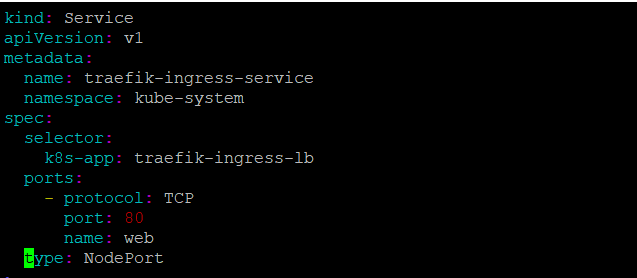
The Ingress manifest above contains a set of HTTP rules that tell the controller how to route the traffic.

Each rule contains the following info:

* **An optional host**.
  + If the host is not specified, the rule applies to all inbound HTTP traffic through the IP address specified. If a host is provided ( test.com ), the rule applies only to that host.
* **A list of paths**
  + That point to the associated backends defined with a serviceName and servicePort . Obviously, a Service that connects multiple Pods together should be created for the rule to work.
* **A backend**.
  + HTTP (and HTTPS) requests to the Ingress matching the host and path of a given rule will be routed to the backend Service specified in that rule.

# **Create NodePorts for External Access**

Create a Service to access Traefik from outside of the cluster. We need a Service that exposes the node port on 80.

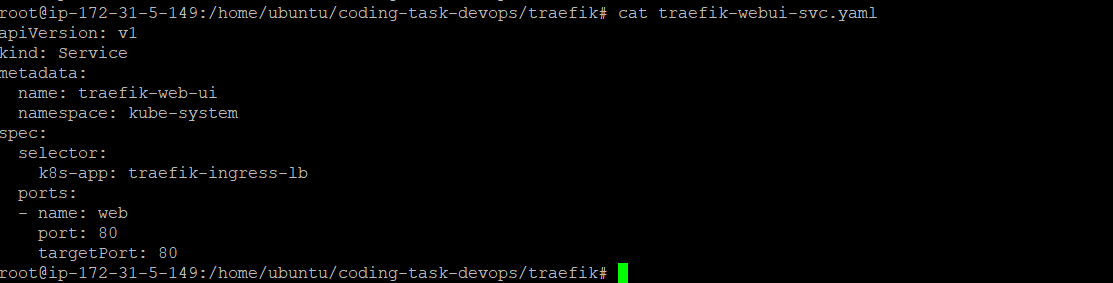


Save this manifest to traefik-svc.yaml and create the Service:

* kubectl create -f traefik-svc.yaml

# **Adding Ingress to the Cluster**

Now we have Traefik as the Ingress Controller in the Kubernetes cluster. However, we still need to define the Ingress resource and a Service that exposes Traefik Web UI.



Save this manifest to traefik-webui-svc.yaml and run:

* kubectl create -f traefik-webui-svc.yaml