Flask Hello World Web Application

To create a web application using Flask that serves pages over HTTPS and blocks HTTP access while accepting a self-signed certificate, you can follow these steps:

First, make sure you have Flask installed. You can install it using pip if you haven't already. Additionally, you'll need Flask-Talisman for enforcing HTTPS. Install them using:

**pip install Flask Flask-Talisman**

Create a Python script (e.g., HelloWorld.py) for your Flask application.

**Note: Local Testing**

**python HelloWorld.py**

Access the application using the URL: http://127.0.0.1:5000

**python HelloWorld-Secure.py**

Access the Application using URL: <https://127.0.0.1:5000>

Note: Since we are using a self-signed certificate, we will likely see a security warning in your browser. You can proceed to the website.

**Generate Self Signed Certificates:**

You can use OpenSSL to generate a self-signed SSL certificate and key pair. Run the following commands to generate a certificate (make sure OpenSSL is installed):

**openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365**

This will create cert.pem and key.pem files in your directory. Note that the certs will be valid for 365 days i.e 1 year. It is configurable and can be modified using ***-days*** param

**Building docker image and testing the container:**

There are two docker files checked in named (dockerfile-plain for http and dockerfile-secure for https). If you want to test https or http follow these steps.

1. Rename the dockerfile-secure to dockerfile (similarly if you want to run http – rename dockerfile-plain to dockerfile)
2. Run the command to build the docker image.

**docker build -t hello-world-flask-app:latest .**

1. Running the docker image.

**docker run -d -p 5000:5000 hello-world-flask-app:latest**

1. Check the container is running

**docker ps**

1. If container is not running, run the following command to get into the container and run the python file locally, most of the time we will see the issue with the module versions that are being used in the code.

**docker run -it -p 5000:5000 hello-world-flask-app:latest bash**

1. If container is running, then access the application from the browser

<http://127.0.0.1:5000> for http and <https://127.0.0.1:5000> for https

**AWS Infrastructure:**

In this step we are creating the EC2 instance and hosting our jenkins server on this instance. Run the following steps to create the infrastructure.

1. Log in to AWS Management console and get the security credentials and configure the credentials on local machine using aws-cli.
2. Run the command to configure the aws credentials.

**aws configure**

1. TO test the configuration

**aws sts get-caller-identity** (it return the UserID, Account and Arn if configuration details are valid otherwise check and re-run)

1. GO to AWS-Infra-Terraform directory and run the following commands to provision the infrastructure on AWS

**terraform init** (If aws provider is previously downloaded on local machine if by pass downloading if not it download the aws provider plugins)

**terraform plan** (it will give the information about the infrastructure it is going to create under the plan)

**terraform apply** (This step will provision the resources)

1. As part of the resource’s creation EC2 instance run userdata script, in this step it will install the jenkins and starts it as a service. (Refer to install\_jenkins script under AWS-Infra-Terraform directory).

**Deployment of Application on EKS Cluster**

Copy the folders and files manually to EC2 instance.

Folders:

Prometheus

Fluentd

ssl\_certificates

templates

Files:

dockerfile-plain

dockerfile-secure

eks-cluster,sh

Helloworld-deployment.yaml

HelloWorld-Plain.py

HelloWorld-Secure.py

jenkinsfile

Running CI/CD:

As part of jenkins pipeline we are creating the following tasks

1. Creating Docker Image
2. Pushing the Docker Image to the ECR registry
3. Creating an EKS Cluster
4. Deploying the HelloWorld Flask Application in the cluster

**Assumptions:**

1. aws-cli and terraform installation on the local machine
2. Docker Account and configuring credentials in jenkins
3. Configuring the AWS Security Credentials in jenkins
4. Creating AWS ECR repository
5. Configuring the .kubeconfig to connect to connect to the cluster
6. Assuming we have already installed and configured the Elastic Search that we will be using it in the Fluentd configuration.

**Monitoring the Application using Prometheus**

Kubernetes cluster consists of pods, nodes, applications and services. To monitor these components as we are using Prometheus.

Prometheus consists of these major components.

1. Prometheus Server
2. Alert Manager for Notification
3. Grafana for Visualization

Go to the Prometheus folder and tun the following steps to deploy Prometheus

1. Create monitoring namespace for isolation

kubectl create namespace monitoring

1. Create Service Account, Cluster Role and Cluster Role Binding Objects.

kubectl apply -f Prometheus-rbac.yaml

1. Create the Persistent Volume and Persistent Volume Claim to store the scraping data

kubectl apply -f Prometheus-pvc.yaml

1. Create Config Map

kubectl apply -f Prometheus-cm.yaml

1. Now start Scraping the component
   1. Scrape Node
   2. Scrape cAdvisor
   3. Scrape API Server
   4. Scrape Pods
2. Create the Deployment

kubectl apply -f Prometheus-Deployment.yaml

1. Create a Service

kubectl apply -f Prometheus-service.yaml

From Prometheus dashboard look at Targets and configure the input metric cursor and add the graphs.

**Fluentd Logging:**

Fluentd to collect logs from different sources, each source may have different log format. Fluentd collects all these data and process into unified format. Tags, filter and parsers will pay key role in grouping and processing.

Follow these steps to configure the Fluentd deployment on the cluster

1. Create Service Account, Cluster Role and Cluster Role Binding objects.

kubectl apply -f Fluentd-rbac.yaml

1. Deploy Fluentd as DaemonSet

kubectl apply -f Fluentd-DS.yaml