**Deploying Flask 🌶️ over Kubernetes ☸️**

**Arya Chakraborty**

Flask is a lightweight web framework for Python. It provides tools and libraries for building web applications, focusing on simplicity and flexibility. Flask supports extensions for added functionality and is known for its ease of use, making it popular for both beginners and experienced developers in web development.

**OVERVIEW:** Deploying Flask over Kubernetes involves several steps to ensure your application is scalable and resilient. First, containerize your Flask application using Docker by writing a ‘Dockerfile’ that specifies the environment and dependencies. Then, push this image to a container registry like Docker Hub. Next, create Kubernetes manifests, including Deployment and Service YAML files, to define the desired state of your application and expose it. Use ‘kubectl’ to apply these configurations to your Kubernetes cluster. Optionally, implement a load balancer and configure auto-scaling policies to manage traffic and resources dynamically, ensuring high availability and performance.

**SETUP:**

**Step 1: Setting up Docker**

Docker is an open-source platform that enables developers to automate the deployment, scaling, and management of applications using containerization. Containers encapsulate an application and its dependencies into a single, portable unit that can run consistently across different computing environments. Docker provides tools and utilities to create, distribute, and run containers, ensuring that applications run the same regardless of where they are deployed, whether on a developer's local machine, a data center, or in the cloud.

For installing docker on your device, install the docker desktop application. Since we have the Linux distro installed in our system (Ubuntu LTS <latest version>), go to settings then resources. Under resources there will be an option of WSL integration. Turn on the WSL integration over Ubuntu LTS as we will be using this Linux distro for completing the task.

**Step 2: Installing Kubernetes**

Kubernetes is an open-source container orchestration platform that automates the deployment, scaling, and management of containerized applications. It provides a robust framework for running distributed systems resiliently, handling scaling, failover, and deployment patterns. Kubernetes groups containers into logical units for easy management and discovery, using abstractions like pods, services, and deployments. It supports self-healing, load balancing, and secret and configuration management, making it a powerful tool for ensuring applications run reliably and efficiently across diverse environments, from local development setups to large-scale production deployments.

Now we have to install a Kubernetes distribution. Since we are deploying the Kubernetes cluster locally, we can install minikube. For install minikube on our Linux distro use the following commands:

****

This will give us access to the Kubernetes distribution minikube and the command line tool ‘kubectl’. ‘kubectl’ is a command-line tool for interacting with Kubernetes clusters. It allows users to deploy applications, manage cluster resources, inspect, and troubleshoot applications, and view logs. ‘kubectl’ facilitates the control of Kubernetes clusters by providing commands to create, update, delete, and get information about resources.

**Step 3: Setting Up the Flask Application**

First, we setup the flask application. If flask is not already installed, install flask by using:****

Now we setup the flask application. In this case we want a simple flask application that handles both GET and POST requests and returns an answer. So, the code for the same is:

****

**Explanation:** The code here upon receiving the GET request renders the “index.html” page using the render\_template(*<html file name>*) function. Next, we move on to the POST request section. Upon receiving a POST request on the same “/” URL, we return the JSON file as the answer to the request using the jsonify(*<JSON name>*) function.

host='0.0.0.0': Makes the server accessible externally, not just from the local machine (required for external requests) and port=5000: Sets the port number to 5000. This ensures the app runs on the specified host and port when executed directly.

The directory structure should be:



This follows the convention as stated by the flask documentation. Once the flask application is ready. Test it by running locally using the command:

**python -m flask run**

If the application is accessible from <http://localhost:5000> then the flask application is deployed correctly. The full flask documentation is available at: <https://flask.palletsprojects.com/en/3.0.x/>

**Step 4: Building the Dockerfile and requirements file**

A Dockerfile is a script that contains a series of instructions on how to build a Docker image. Each instruction in a Dockerfile creates a layer in the image, defining the base image, copying files, installing dependencies, and setting up the environment. When the Dockerfile is processed, Docker executes these instructions to create a final image that can be run as a container. This allows for consistent and reproducible deployment of applications across different environments.

A requirements.txt file is used in Python projects to list all the dependencies needed to run the project. Each line in this file specifies a package and its version, allowing for consistent installation of dependencies. When the command **pip install -r requirements.txt** is run, it installs all the listed packages, ensuring that the environment is set up with the correct versions required by the project. This file is crucial for reproducibility and sharing Python projects.

Here the Dockerfile consists of the following code:



**Explanation:** This Dockerfile sets up a Docker image to run a Flask application by starting with the lightweight Python 3.9-slim base image. It sets the working directory to /app, where all subsequent commands will be executed. The requirements.txt file, which lists the application's dependencies, is copied into this directory and then installed using pip install -r requirements.txt. Following this, the entire application code from the current directory on the host machine is copied into the /app directory in the container. Environment variables are set to specify the Flask application entry point as app.py and to make the Flask server accessible externally by setting the host to 0.0.0.0. Finally, the default command flask run is specified to start the Flask development server when the container launches. This setup ensures that the Flask application is containerized with all necessary dependencies and configurations for consistent deployment.

Since we are using flask, we need to include that in the **requirements.txt** file. Therefore, the contents will be:

**flask**

This indicates that the dependencies of the app only include flask. Therefore, the new directory structure will be:



This completes the initial file setup for building the docker image.

**Step 5: Setup a GitHub** **Repository**

Create a new repository in GitHub and add the complete directory as it to it. The directory structure should be the same as the one just created. Create a public repo to avoid the authentication steps when we try to access it in the next step.

The repo will be saved in the form of **https://github.com/<*username*> /<*repo name*>.git**

**Step 6: Building the Docker image**

Open the Linux distro and use the following commands to clone the repo locally that you just created in GitHub:



Where username is the GitHub username and repo name is the name of the repository in which the flask application is saved. The cd command helps to get into the folder which contains the Dockerfile which is necessary before the next steps.

Once we are in the same folder that contains the Dockerfile, open the Docker Hub website (<https://hub.docker.com/>) and setup an account there using the username and the password method. Now in the Linux distro execute the following commands:

**docker build -t <*repo name*> .**

This will build the docker image using the docker file present in the directory. To check if the image has been formed, use the command:

**docker images**

This should show the formed images. Next use the command:

**docker login**

Use the credentials for authentication you have for the account just created on the Docker Hub. Before pushing the image to Docker Hub, we must tag it. Therefore use:

**docker tag flask-kubernetes-deploy <*dockerhub\_username*>/<*repo name*>**

Finally push the image to Docker Hub using:

**docker push <*dockerhub\_username*>/<*repo name*>**

Now to check whether the image has been pushed or not, go the Docker Hub website and under repositories find the image just pushed. Add a description if required to the image. The docker image build is now complete.

**Step 7: Deployment of the Application**

Now to deploy the application over Kubernetes we need to configure the required YAML files. Since we are required to use the ENV variables using ConfigMaps and Secrets, we first create the same files for the application.

A ConfigMap is a Kubernetes resource used to store non-sensitive configuration data in key-value pairs, enabling applications to retrieve necessary settings without embedding them in the container image. This separation enhances portability and flexibility, allowing changes to configurations without rebuilding images. ConfigMaps can hold data such as configuration files, command-line arguments, or environment variables, making it easy to manage and modify application settings dynamically. By using ConfigMaps, you can ensure that configuration data is consistently and centrally managed, facilitating streamlined updates and maintenance of Kubernetes-deployed applications.

A Secret is a Kubernetes resource designed to store and manage sensitive information securely, such as passwords, API keys, and TLS certificates. Secrets are encoded in base64 to add a layer of obfuscation, preventing direct exposure of plaintext credentials. This allows for secure handling of sensitive data, ensuring it is not embedded in application code or container images. By leveraging Secrets, you can securely pass confidential information to your applications, either through environment variables or mounted volumes, maintaining the security and integrity of sensitive data within a Kubernetes cluster.

The **configMap** of the application is as follows:



The ENV variable FLASK APP NAME is provided under the data section of the file as key and value pair. Similarly, the **Secrets file** contain the following:



The environment variables FLASK RUN HOST and FLASK RUN PORT are provided under the data section in base 64 encoding which provides an additional layer of security.

A deployment.yaml file is a configuration file used in Kubernetes to define and manage the deployment of applications. It specifies how many replicas of a particular application should run, the container image to use, and other settings related to application updates, scaling, and rollout strategies. This file ensures that applications are deployed consistently and can be easily managed and updated within a Kubernetes cluster.

This deployment.yaml file defines a Kubernetes Deployment for a Flask application, specifying that it uses the apps/v1 API version and is of the kind Deployment, named flask-kubernetes-app. It sets the desired state to run a single replica of the application, with the pods identified by the label app: flask-kubernetes-app. The pod template includes metadata with matching labels and a specification for the container. The container is named flask-kubernetes-app and uses the Docker image “docker.io/aryachakraborty/flask-over-kubernetes:latest”, exposing port 5000 where the Flask app will listen for requests. The environment variables FLASK\_RUN\_HOST and FLASK\_RUN\_PORT are set from a Kubernetes Secret named flask-kubernetes-secret, using the keys flask-run-host and flask-run-port, respectively, while FLASK\_APP is set from a ConfigMap named flask-kubernetes-configmap using the key flask-app-name. This configuration ensures that the Flask application runs with the specified settings and environment variables, with the Deployment managing the application's lifecycle to maintain the desired state within the Kubernetes cluster. The docker image used should be the same as the one present on the Docker Hub website (along with tags if any formed). The deployment file is as follows:



Now we need to add a flask service. To do that we need a service YAML file which contains the type of service (LoadBalancer), the ports (including the protocol and target port) etc. This service file will allow us to access the flask application from our browser (external request). The contents of the service file are as follows:



All the files are not ready. Next, we need to apply them one by one. First, we have to start the minikube cluster using:

**minikube start**

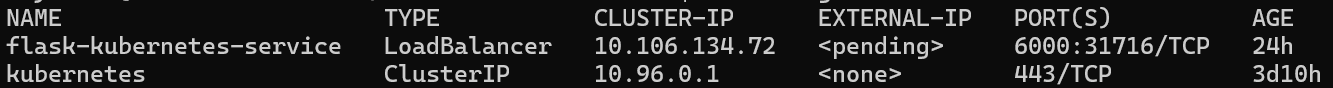
Next for applying the changes we need to use the kubectl ‘apply’ command. Since the deployment file contains information dependent on the secret and configMap, we need to create them first. Therefore,



Now check if the pods and services are running using kubectl ‘get’ command.



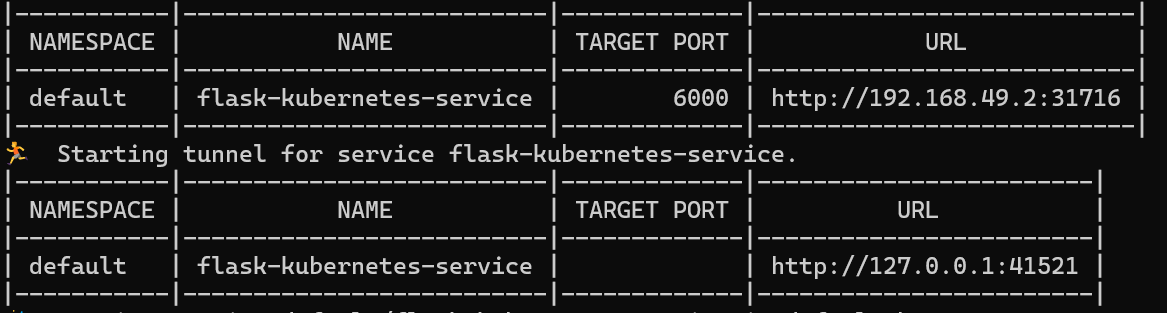
An example of the service should like:



Now to get an external IP for our service and to be able to access it from our browser we use the command:

**minikube service <flask service name>**

When executed, Minikube finds the service within the cluster, determines its accessible URL, and opens this URL in your default web browser, if available. This command facilitates quick access to and testing of services like a Flask application, ensuring they are running correctly and can be accessed via the local development environment provided by Minikube. An example of the terminal output should be:



Using a tunnel for a service, provides a straightforward way to access LoadBalancer services in a Minikube cluster from your local machine. It facilitates local development and testing by creating a network route, making services easily accessible as if they were local applications.

Therefore, to access the application in browser, just use the URL after the tunneling. If everything is done correctly then the page made for “/” should appear. The directory structure that could be kept for easy maintenance is:



To modify the application necessary changes can be made and the docker image should be rebuilt with the ‘latest’ tag. Then the process is similar i.e. apply the YAML files over the cluster and create the external IP using the command ‘minikube service’.

That is the deployment of a flask application over Kubernetes.