In this tutorial, you will learn how to build a a RESTful Flask API for Kubernetes, by building a production-ready Docker container.

The solution provided by this post will use Kubernetes Deployments, which will allow you to scale your application. Deployments also provide additional features to control how new image updates are deployed and are handled on failures.

You will also learn how to create services for your Pods to expose them, as well as how to use secrets for sensitive information.

uWSGI Application Server

In order to run the Flask API an application server is required. There are a few servers available, however, this tutorial will cover uWSGI. Another popular choice is gUnicorn.

NGINX Web Server

A web server will handle all incoming requests, and then reverse proxy them to the application server. While the container could run the application server alone, a web server provides more control over the traffic hitting our Flask API.

NGINX is a highly efficient, event-driven web server that is capable of handling high volumes of traffic. It is also very simple to add uWSGI backend support.

A Basic Flask API

The following is a basic Flask REST API, which will be used for demonstration purposes. The application will be written as simple CRUD api that will connect with a MySQL database.

|  |  |
| --- | --- |
|  | import pymysql |
|  | from db\_config import mysql |
|  | from flask import jsonify |
|  | from flask import flash, request |
|  | from werkzeug import generate\_password\_hash, check\_password\_hash |
|  |  |
|  | from flaskext.mysql import MySQL |
|  | import os |
|  |  |
|  |  |
|  | from flask import Flask |
|  |  |
|  | app = Flask(\_\_name\_\_) |
|  |  |
|  | mysql = MySQL() |
|  |  |
|  | # MySQL configurations |
|  | app.config['MYSQL\_DATABASE\_USER'] = os.environ['db\_username'] |
|  | app.config['MYSQL\_DATABASE\_PASSWORD'] = os.environ['db\_password'] |
|  | app.config['MYSQL\_DATABASE\_DB'] = os.environ['db\_name'] |
|  | app.config['MYSQL\_DATABASE\_HOST'] = '192.168.1.213' |
|  | mysql.init\_app(app) |
|  |  |
|  |  |
|  | @app.route('/') |
|  | def index(): |
|  | return "Hello, world!" |
|  |  |
|  | @app.route('/add', methods=['POST']) |
|  | def add\_user(): |
|  | try: |
|  | \_json = request.json |
|  | \_name = \_json['name'] |
|  | \_email = \_json['email'] |
|  | \_password = \_json['pwd'] |
|  | # validate the received values |
|  | if \_name and \_email and \_password and request.method == 'POST': |
|  | #do not save password as a plain text |
|  | \_hashed\_password = generate\_password\_hash(\_password) |
|  | # save edits |
|  | sql = "INSERT INTO tbl\_user(user\_name, user\_email, user\_password) VALUES(%s, %s, %s)" |
|  | data = (\_name, \_email, \_hashed\_password,) |
|  | conn = mysql.connect() |
|  | cursor = conn.cursor() |
|  | cursor.execute(sql, data) |
|  | conn.commit() |
|  | resp = jsonify('User added successfully!') |
|  | resp.status\_code = 200 |
|  | return resp |
|  | else: |
|  | return not\_found() |
|  | except Exception as e: |
|  | print(e) |
|  | finally: |
|  | cursor.close() |
|  | conn.close() |
|  |  |
|  | @app.route('/users') |
|  | def users(): |
|  | try: |
|  | conn = mysql.connect() |
|  | cursor = conn.cursor(pymysql.cursors.DictCursor) |
|  | cursor.execute("SELECT \* FROM tbl\_user") |
|  | rows = cursor.fetchall() |
|  | resp = jsonify(rows) |
|  | resp.status\_code = 200 |
|  | return resp |
|  | except Exception as e: |
|  | print(e) |
|  | finally: |
|  | cursor.close() |
|  | conn.close() |
|  |  |
|  | @app.route('/user/') |
|  | def user(id): |
|  | try: |
|  | conn = mysql.connect() |
|  | cursor = conn.cursor(pymysql.cursors.DictCursor) |
|  | cursor.execute("SELECT \* FROM tbl\_user WHERE user\_id=%s", id) |
|  | row = cursor.fetchone() |
|  | resp = jsonify(row) |
|  | resp.status\_code = 200 |
|  | return resp |
|  | except Exception as e: |
|  | print(e) |
|  | finally: |
|  | cursor.close() |
|  | conn.close() |
|  |  |
|  | @app.route('/update', methods=['POST']) |
|  | def update\_user(): |
|  | try: |
|  | \_json = request.json |
|  | \_id = \_json['id'] |
|  | \_name = \_json['name'] |
|  | \_email = \_json['email'] |
|  | \_password = \_json['pwd'] |
|  | # validate the received values |
|  | if \_name and \_email and \_password and \_id and request.method == 'POST': |
|  | #do not save password as a plain text |
|  | \_hashed\_password = generate\_password\_hash(\_password) |
|  | # save edits |
|  | sql = "UPDATE tbl\_user SET user\_name=%s, user\_email=%s, user\_password=%s WHERE user\_id=%s" |
|  | data = (\_name, \_email, \_hashed\_password, \_id,) |
|  | conn = mysql.connect() |
|  | cursor = conn.cursor() |
|  | cursor.execute(sql, data) |
|  | conn.commit() |
|  | resp = jsonify('User updated successfully!') |
|  | resp.status\_code = 200 |
|  | return resp |
|  | else: |
|  | return not\_found() |
|  | except Exception as e: |
|  | print(e) |
|  | finally: |
|  | cursor.close() |
|  | conn.close() |
|  |  |
|  | @app.route('/delete/') |
|  | def delete\_user(id): |
|  | try: |
|  | conn = mysql.connect() |
|  | cursor = conn.cursor() |
|  | cursor.execute("DELETE FROM tbl\_user WHERE user\_id=%s", (id,)) |
|  | conn.commit() |
|  | resp = jsonify('User deleted successfully!') |
|  | resp.status\_code = 200 |
|  | return resp |
|  | except Exception as e: |
|  | print(e) |
|  | finally: |
|  | cursor.close() |
|  | conn.close() |
|  |  |
|  | @app.errorhandler(404) |
|  | def not\_found(error=None): |
|  | message = { |
|  | 'status': 404, |
|  | 'message': 'Not Found: ' + request.url, |
|  | } |
|  | resp = jsonify(message) |
|  | resp.status\_code = 404 |
|  |  |
|  | return resp |
|  |  |
|  | if \_\_name\_\_ == "\_\_main\_\_": |
|  | app.run() |

[**view raw**](https://gist.github.com/serainville/a468c733c03ac44dc184e7e519d8557d/raw/140f499bfb81926e01280740145d2b2caeb7f980/flaskapi.py)[**flaskapi.py**](https://gist.github.com/serainville/a468c733c03ac44dc184e7e519d8557d#file-flaskapi-py) hosted with  by [**GitHub**](https://github.com/)

Configurations are stored in a config.py file, which is where the database connections settings will be stored.

|  |  |
| --- | --- |
|  | from flaskext.mysql import MySQL |
|  | import os |
|  |  |
|  |  |
|  | from flask import Flask |
|  |  |
|  | app = Flask(\_\_name\_\_) |
|  |  |
|  | mysql = MySQL() |
|  |  |
|  | # MySQL configurations |
|  | app.config['MYSQL\_DATABASE\_USER'] = os.environ['db\_username'] |
|  | app.config['MYSQL\_DATABASE\_PASSWORD'] = os.environ['db\_password'] |
|  | app.config['MYSQL\_DATABASE\_DB'] = os.environ['db\_name'] |
|  | app.config['MYSQL\_DATABASE\_HOST'] = '192.168.1.213' |
|  | mysql.init\_app(app) |

[**view raw**](https://gist.github.com/serainville/50b3e34160bf74ce4376de0fa217a8f1/raw/e7f909b599d6145933ccdc3ce0b6aa24feef11e0/flaskapi-config.py)[**flaskapi-config.py**](https://gist.github.com/serainville/50b3e34160bf74ce4376de0fa217a8f1#file-flaskapi-config-py) hosted with  by [**GitHub**](https://github.com/)

Notice that none of the environmental information is statically stored in config.py. As a best practice with Docker and, by extension, Kubernetes, environment specific information should not be stored in a Docker container.

Even more important is not storing credentials in the container or your code. All sensitive information, called secrets, should be handled by a secrets vault, such as [Hashicorp Vault](https://www.vaultproject.io/" \t "_blank), or in the case of this tutorial [Kubernetes Secrets](https://kubernetes.io/docs/concepts/configuration/secret/).

Secrets can be fetched from a Vault or from Kubernetes Secrets at runtime.

A Note on Kubernetes Secrets

Kubernetes Secrets provides simple storage of your sensitive data and files. While used in this tutorial as an example of handling secrets, you must keep in mind that its purpose is just to transport secrets to pods, for consumption by containers.

A strongly recommended setup is to use Hashicorp Vault and Kubernetes secrets in concert. Hashicorp Vault will be the authority of all secrets, and those secrets will be synced to Kubernetes Secrets.

Unfortunately, this is outside of the scope of this tutorial.

Building a Docker Image

Before we can create a Docker image we are going to need a Dockerfile. Let’s take a minute to understand what the Dockerfiles will need to support the REST API we’ve written.

Flask has a built-in web server that is useful during development, however, it is strongly recommended that you not use it for production. The reason is simple, the built-in server is single processed and single threaded.

The lack of threads is an issue with concurrency. The REST API won’t be able to handle much traffic.

Following best practices, the Flask app in this tutorial will use uWSGI as the server.

Getting back to the Dockerfile, there are no official images for running Flask applications. Rather, we will use the official Python3 image and build upon it.

Create a new filed named Dockerfile in your project’s directory

touch Dockerfile

Start the Dockerfile with the following line

FROM python:3.6-slim

This image will provide an Ubuntu-based container that is slimmed down for running Python. There are much smaller images available provided by the community.

The Dockerfile should also include lines to install packages updates, to address bug fixes and vulnerabilites, as well as install any prerequisites. The prerequisites for a Flask application are: python3-dev and build-essentials.

RUN apt-get clean \

&& apt-get -y update

RUN apt-get -y install \

nginx \

python3-dev \

build-essential

Add any configurations to services installed in the container below the lines above.

COPY nginx.conf /etc/nginx/nginx.com

The application will run from the directory named /app within the container. Set this as the working directory.

WORKDIR /app

Now let’s add the requirements.txt file to the container, and then install our Flask application’s dependencies.

COPY requirements.txt /app/requirements.txt

RUN pip install -r requirements.txt --src /usr/local/src

The container is still missing the Flask application itself, which will be added with the following lines.

COPY app.py /app/app.py

COPY config.py /app/config.py

COPY uwsgi.ini /app/wsgi.ini

COPY startup.sh /app/startup.sh

The startup script will need to be executable, so the chmod command will need to be ran against it.

RUN chmod +x ./startup.sh

Finally, let’s expose the container over port 80 and have the container run as the startup.sh script.

EXPOSE 80

CMD [ "./startup.sh" ]

Publishing the Image to a Registry

In order to deploy your Flask Docker image in Kubernetes, it must published to a container registry accessible to Kubernetes. It’s important to push new builds to the registry.

Dockerhub

To deploy your images to Dockerhub, you will need to authenticate first.

docker login

When prompted, enter your Dockerhub log in credentials. Once you are successfully authenticated, an authentication token will be stored locally, which will permit you to run commands against Dockerhub without including your credentials.

Any image being pushed to a repository must include repository information in its name. For Dockerhub, it the image must start with your Dockerhub ID.

Use the following command to add a tag for your repository.

docker -t flask-api:1.0.0 serverlab/flask-api:1.0.0

Our image can be pushed up to Dockerhub using the new tag we applied to it.

docker push serverlab/flask-api:1.0.0

Storing Database Credentials in Kubernetes Secrets

Secrets should never be stored as plain text, in your code or in any of your Kubernetes manifests. Passwords, certificates and any other sensitive data should be stored away from your application.

In this tutorial, we will use Kubernetes Secrets as our secrets store. The Flask application’s database credentials can be stored here, and retrieved when its Docker image is deployed as a Pod.

Secrets are stored as Base64 encoded strings. Let’s encode our database’s username and password in base64.

echo -n 'db-username' | base64

echo -n 'my-super-secret-password' | base64

Both commands will output a string of seemingly random characters, such as the example below. Keep note of them.

c3VwZXItc2VjcmV0LXBhc3N3b3JkCg==

Now create a new file named flaskapi-secrets.yml, and add the following contents to it.

---

apiVersion: v1

kind: Secret

metadata:

name: flaskapi-secrets

type: Opaque

data:

db\_username: ZGItdXNlcm5hbWU=

db\_password: c3VwZXItc2VjcmV0LXBhc3N3b3JkCg==

Two keys have been added under data. One key will store the value for the database username, and the other the database password.

The values for the keys are the base64 encoded strings created earlier.

Add the secrets to Kubernetes using the kubectl apply command.

kubectl apply -f flaskapi-secrets.yml

Once created, store this file away from your application and any other manifests. It should always be heavily protected. Alternative, you may delete this file when done.

Creating a Kubernetes Deployment

The Flask application used in this tutorial is stateless, meaning a sessions is not tied to a specific instance, nor is any user session information stored. It is a simple API, and that means we can scale it up or down without worrying about state.

A Kubernetes Deployment is the ideal way for deploying applications in Kubernetes. Deployments allow you to scale your application up or down, easily as demand requires it.

Deployments allows you to set deployment strategies, which are used to handle pod failures and Docker images updates.

When the Deployment manifest for your Flask application is updated to use a new image version, and subsequentially applied, the Deployment controller allows you to set a deployment strategy. By default, new versions of your applications must be deployed successfully before the older versions are taken down.

The following is an example of a Kubernetes Deployment. It will be used to deploy our Flask application with 3 replicas.

Create a new filed named flaskapp-deployment.yml, and add the following to it. Ensure you replace the image name, currently serverlab/flask-api:1.0.0, with that of your own.

|  |  |
| --- | --- |
|  | --- |
|  | apiVersion: apps/v1 |
|  | kind: Deployment |
|  | metadata: |
|  | name: flaskapi-deployment |
|  | labels: |
|  | app: flaskapi |
|  | spec: |
|  | replicas: 3 |
|  | selector: |
|  | matchLabels: |
|  | app: flaskapi |
|  | template: |
|  | metadata: |
|  | labels: |
|  | app: flaskapi |
|  | spec: |
|  | containers: |
|  | - name: flaskapi |
|  | image: serverlab/flask-api:1.0.4 |
|  | ports: |
|  | - containerPort: 80 |
|  | env: |
|  | - name: db\_username |
|  | valueFrom: |
|  | secretKeyRef: |
|  | name: flaskapi-secrets |
|  | key: db\_username |
|  | - name: db\_password |
|  | valueFrom: |
|  | secretKeyRef: |
|  | name: flaskapi-secrets |
|  | key: db\_password |
|  | - name: db\_name |
|  | value: flaskapi\_demo |

[**view raw**](https://gist.github.com/serainville/9586755e50f1a5fe97ce67bb4693b600/raw/2b12277d07e28c69653bdca67b342819ae25970b/flaskapp-k8s-deployment.yml)[**flaskapp-k8s-deployment.yml**](https://gist.github.com/serainville/9586755e50f1a5fe97ce67bb4693b600#file-flaskapp-k8s-deployment-yml) hosted with  by [**GitHub**](https://github.com/)

Notice the the ENV key under the CONTAINER key in the manifest. The Flask API fetches environment variables for the environmental settings. This allows us to use the exact same image in any environment, from DEV to UAT and Production.

Deploying Your Deployment

Creating the deployment on the Kubernetes cluster is done using the kubectl apply command. There is also a create command, however, the apply command allows us to more easily apply updated manifests.

kubectl apply -f flaskapi-deployment.yml

To verify your deployment is created successfully, use the kubectl get svc command.

kubectl get svc

And finally, to verify the Pods the deployment created are healthy, use the kubectl get pods command.

kubectl get pods

If any Pods are in a failed state, the kubectl get pods command will only show you basic information. For more detailed information about the Pod, you use the describe command.

kubectl describe pod <pod-name>

Pod Logs

Logging is essential for identifying and troubleshooting any errors. To view the logs from your containers, you use the kubectl log <pod name> command.

kubectl log flaskapi-abcd1234

It is important to remember that Docker, and Kubernetes, will only output logs sent to STDOUT. Always make sure that your applications any service running your application export logs as STDOUT.

Creating a Kubernetes Service

Services allow us to expose our Pods, internally to other Pods and externally to the Public. Each service is assigned a statically assigned internal IP address, which allows us to connect to the service.

Pods, on the other hand, are ephemeral, and new pods replacing old or failed ones will not re-use IP addresses. For this this reason, it is always advised that you connect to services rather than Pods.

To create a service for the Flask application, create a new file named flaskapi-service.yml

Add the following lines to it.

|  |  |
| --- | --- |
|  | --- |
|  | apiVersion: v1 |
|  | kind: Service |
|  | metadata: |
|  | name: flaskapi-service |
|  | spec: |
|  | selector: |
|  | app: flaskapi |
|  | ports: |
|  | - protocol: TCP |
|  | port: 80 |
|  | targetPort: 80 |

[**view raw**](https://gist.github.com/serainville/24c190ac29e34da2a5c51ca29c124b06/raw/e8ffdf397a6a7faec1a1d00525c97840c607f3fe/flaskapi-k8s-service.yml)[**flaskapi-k8s-service.yml**](https://gist.github.com/serainville/24c190ac29e34da2a5c51ca29c124b06#file-flaskapi-k8s-service-yml) hosted with  by [**GitHub**](https://github.com/)