**art 1: Creating your application**

**Prerequisite:** Install Flask, Flask-PyMongo and MongoDB server instance on your machine if you wish to run the app locally once. You can skip this step as we will be containerizing both the app and the database.

The app that we will be building here is a simple tasks app where you can create, update and delete simple one liner tasks. These data will be stored inside a MongoDB database, an open source database that stores flexible JSON-like documents. By default, when a MongoDB Server instance is started on a machine, it starts to listen on port 27017. Flask-PyMongo module helps us to bridge Flask and MongoDB and provides some convenience helpers. ObjectId module is a tool for working with MongoDB ObjectId, which is the default value of \_id field of each document, generated during the creation of any document. We will be using curl commands to talk to our app, which will be discussed later once we deploy our app onto a cluster.

Currently, our working directory tasksapp-python consists of:

1. requirements.txt - consists of Flask and Flask-PyMongo entries which will be installed when we build our Docker image.

Flask  
Flask-PyMongo

2. app.py - which when run on any host (python app.py), can be accessed at http://localhost:5000/ inside it.

from flask import Flask, request, jsonify  
from flask\_pymongo import PyMongo  
from bson.objectid import ObjectId  
import socketapp = Flask(\_\_name\_\_)  
app.config["MONGO\_URI"] = "mongodb://mongo:27017/dev"  
mongo = PyMongo(app)  
db = mongo.db@app.route("/")  
def index():  
 hostname = socket.gethostname()  
 return jsonify(  
 message="Welcome to Tasks app! I am running inside {} pod!".format(hostname)  
 )@app.route("/tasks")  
def get\_all\_tasks():  
 tasks = db.task.find()  
 data = []  
 for task in tasks:  
 item = {  
 "id": str(task["\_id"]),  
 "task": task["task"]  
 }  
 data.append(item)  
 return jsonify(  
 data=data  
 )@app.route("/task", methods=["POST"])  
def create\_task():  
 data = request.get\_json(force=True)  
 db.task.insert\_one({"task": data["task"]})  
 return jsonify(  
 message="Task saved successfully!"  
 )@app.route("/task/<id>", methods=["PUT"])  
def update\_task(id):  
 data = request.get\_json(force=True)["task"]  
 response = db.task.update\_one({"\_id": ObjectId(id)}, {"$set": {"task": data}})  
 if response.matched\_count:  
 message = "Task updated successfully!"  
 else:  
 message = "No Task found!"  
 return jsonify(  
 message=message  
 )@app.route("/task/<id>", methods=["DELETE"])  
def delete\_task(id):  
 response = db.task.delete\_one({"\_id": ObjectId(id)})  
 if response.deleted\_count:  
 message = "Task deleted successfully!"  
 else:  
 message = "No Task found!"  
 return jsonify(  
 message=message  
 )@app.route("/tasks/delete", methods=["POST"])  
def delete\_all\_tasks():  
 db.task.remove()  
 return jsonify(  
 message="All Tasks deleted!"  
 )if \_\_name\_\_ == "\_\_main\_\_":  
 app.run(host="0.0.0.0", port=5000)

In app.py file, we first import all the required modules and create instances of the Flask class (our app) and the PyMongo class (our database). Note that the hostname in the MONGO\_URI Flask configuration variable is defined as ‘mongo’ instead of ‘localhost’. This is because ‘mongo’ will be the name of our database container, and containers in the same Docker network can talk to each other by their names.

Use app.config[“MONGO\_URI”] = “mongodb://localhost:27017/dev” if you wish to run the app locally.

Our app consists of six functions which are assigned URLs by @app.route() Python decorator. At first glance, it is easy to understand that the decorator is telling our app that whenever a user visits our @app domain at the given route(), execute the underlying function.

1. index() - displays a welcome message whenever you hit the app. Also displays the hostname of the machine where our app is running. This is useful to understand that we will be hitting a random pod each time we try to access our app.
2. get\_all\_tasks() - displays all the tasks that are available in the database as a list of dictionaries.
3. create\_task() - creates a new task which is stored in the database with a unique ID.
4. update\_task(id) - modifies any existing task. If no task is found with the queried ID, appropriate message is returned.
5. delete\_task(id) - removes that entry of the task having the queried ID from the database. Returns appropriate message if no task with the specified ID is found.
6. delete\_all\_tasks() - removes all the tasks. Returns an empty list.

In the final section, where we run the app, we define the host parameter as ‘0.0.0.0’ to make the server publicly available, running on the machine’s IP address. The machine here I am talking about is a Pod, which we will discuss later.

**Part 2: Containerizing your application**

**Prerequisite:**Install Docker on your machine and create a Docker Hub account, if you don’t have one. Authorize Docker to connect to the Docker Hub account using docker login command.

Now, let us build a Docker image of our app which can be pushed to the Docker Hub registry. Inside our working directory tasksapp-python, a Dockerfile with the following contents, is to be created:

FROM python:alpine3.7  
COPY . /app  
WORKDIR /app  
RUN pip install -r requirements.txt  
ENV PORT 5000  
EXPOSE 5000  
ENTRYPOINT [ "python" ]  
CMD [ "app.py" ]

We are using the official Python image, based on Alpine Linux project, as the base image and copying the contents of our working directory to a new directory on the image. We are instructing the image to expose the port 5000 when run as a container, on which we can access our app. Finally, our app container is configured to run python app.py automatically when it is created.

Here, we build our image with tag <username>/<image-name>:<version> format using the below command:

>docker build -t varunkumarg/tasksapp-python:1.0.0 .

and push it to Docker Hub registry:

>docker push varunkumarg/tasksapp-python:1.0.0

*Note: Ensure to replace <varunkumarg> with your Docker Hub username.*

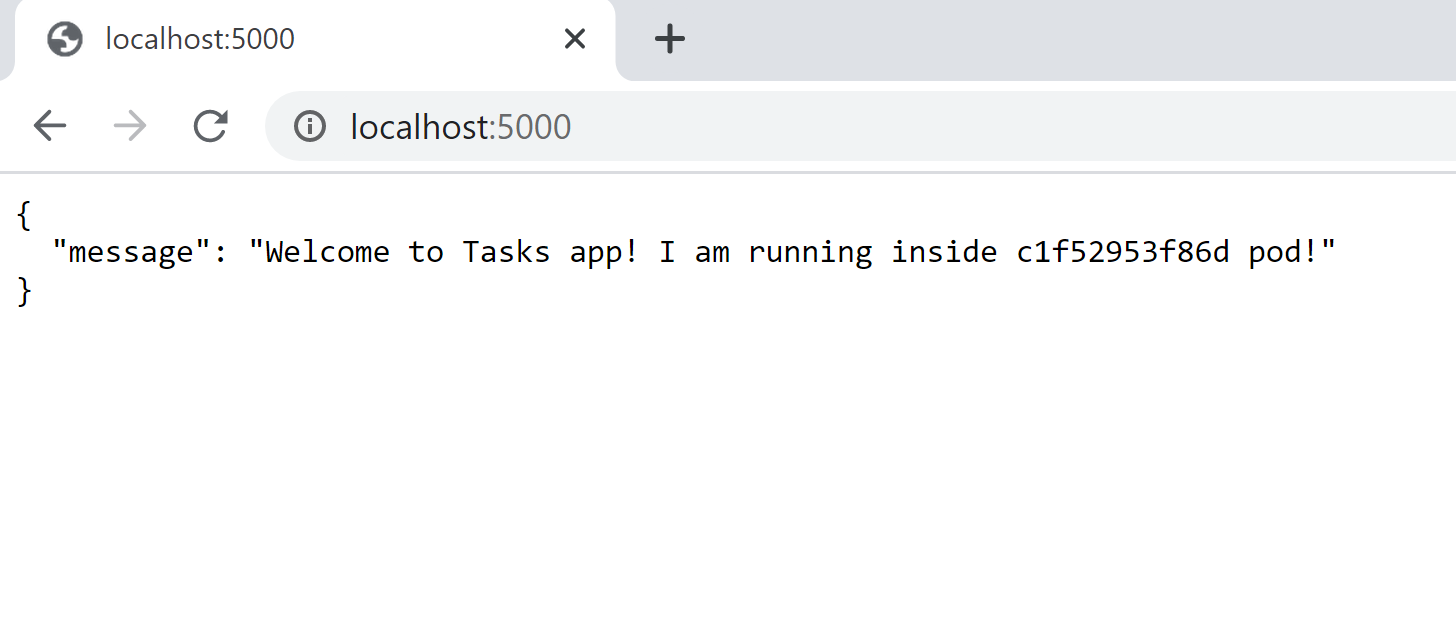
Our image is publicly available on Docker Hub and anybody in the world can download and run it.

Now that our app is containerized, what about our database? How can we containerize that? We don’t have to worry about it as we can easily use the official mongo Docker image and run it on the same network as the app container.

Run the below commands to test our image locally:

>docker network create tasksapp-net  
>docker run --name=mongo --rm -d --network=tasksapp-net mongo  
>docker run –-name=tasksapp-python --rm -p 5000:5000 -d –-network=tasksapp-net varunkumarg/tasksapp-python:1.0.0

Image for post



‘c1f52953f86d’ is the Container ID of the tasksapp-python container

The next step is to run this containerized app on a container orchestrator.

**Part 3: Deploying your application**

**Prerequisite:** A Kubernetes cluster created using installation tools like Minikube (single node cluster) or kubeadm (multi node cluster) or a managed Kubernetes service like GKE, EKS or AKS. Here, I am using a cluster consisting 1 master and 3 workers deployed using kubeadm on Ubuntu 18.04 VMs.

myuser@master1:~$ kubectl get nodes  
NAME STATUS ROLES AGE VERSION  
master1 Ready master 25h v1.18.5  
worker1 Ready <none> 25h v1.18.5  
worker2 Ready <none> 25h v1.18.5  
worker3 Ready <none> 25h v1.18.5

To deploy our application on a Kubernetes cluster, we will be creating .yaml files for each Kubernetes resource and running a set of kubectl commands. The resources are to be deployed as follows:

1. tasksapp Deployment:

The tasksapp.yaml file defines the deployment of our app running in a pod on any worker node. The spec section defines the pod where we specify the image to be pulled and run. The port 5000 of the Pod is exposed.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: tasksapp  
 labels:  
 app: tasksapp  
spec:  
 replicas: 1  
 selector:  
 matchLabels:  
 app: tasksapp  
 template:  
 metadata:  
 labels:  
 app: tasksapp  
 spec:  
 containers:  
 - name: tasksapp  
 image: varunkumarg/tasksapp-python:1.0.0  
 ports:  
 - containerPort: 5000  
 imagePullPolicy: Always

Create the tasksapp deployment using kubectl create.

myuser@master1:~$ kubectl create -f tasksapp.yaml  
deployment.apps/tasksapp created  
myuser@master1:~$ kubectl get deployments  
NAME READY UP-TO-DATE AVAILABLE AGE  
tasksapp 1/1 1 1 61s

We can scale the deployment using kubectl scale using --replicas option.

myuser@master1:~$ kubectl scale deployment tasksapp --replicas=3  
deployment.apps/tasksapp scaled  
myuser@master1:~$ kubectl get pods -o wide  
NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE READINESS GATES  
tasksapp-645b67dfbc-2kqp5 1/1 Running 0 37m 10.38.0.1 worker2 <none> <none>  
tasksapp-645b67dfbc-mqh97 1/1 Running 0 32s 10.44.0.2 worker3 <none> <none>  
tasksapp-645b67dfbc-ncrt7 1/1 Running 0 32s 10.40.0.2 worker1 <none> <none>

2. tasksapp-svc LoadBalancer Service:

The LoadBalancer Service enables the pods in a deployment to be accessible from outside the cluster. Here, since we are using a custom Kubernetes cluster, we will be accessing the app from the master node at <service-ip>:<service-port>. The service is defined by tasksapp-svc.yaml. The advantage of using a Service is that it gives us a single consistent IP to access our app as many pods may come and go in our deployment.

apiVersion: v1  
kind: Service  
metadata:  
 name: tasksapp-svc  
spec:  
 selector:  
 app: tasksapp  
 ports:  
 - port: 8080  
 targetPort: 5000  
 type: LoadBalancer

Here, the port 8080 of the service tasksapp-svc is bound to the port 5000 of the pods attached to it.

myuser@master1:~$ kubectl create -f tasksapp-svc.yaml  
service/tasksapp-svc created  
myuser@master1:~$ kubectl get svc tasksapp-svc  
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE  
tasksapp-svc LoadBalancer 10.103.82.106 <pending> 8080:32156/TCP 13s

Now, we should be able to access our app at 10.103.82.106:8080.

myuser@master1:~$ curl 10.103.82.106:8080  
{  
 "message": "Welcome to Tasks app! I am running inside tasksapp-645b67dfbc-2kqp5 pod!"  
}  
myuser@master1:~$ curl 10.103.82.106:8080  
{  
 "message": "Welcome to Tasks app! I am running inside tasksapp-645b67dfbc-2kqp5 pod!"  
}  
myuser@master1:~$ curl 10.103.82.106:8080  
{  
 "message": "Welcome to Tasks app! I am running inside tasksapp-645b67dfbc-mqh97 pod!"  
}  
myuser@master1:~$ curl 10.103.82.106:8080  
{  
 "message": "Welcome to Tasks app! I am running inside tasksapp-645b67dfbc-ncrt7 pod!"  
}

We can clearly see that the LoadBalancer sends the traffic to any random pod each time we try to hit our app. At this point in time, other functions of our application cannot be used as our database pod is not ready yet.

3. mongo-pv PersistentVolume:

This creates a storage volume of 256 MB that is to be made available to the mongo container. The contents of this volume persist, even if the MongoDB pod is deleted or moved to a different node. It is defined in mongo-pv.yaml.

apiVersion: v1  
kind: PersistentVolume  
metadata:  
 name: mongo-pv  
spec:  
 capacity:  
 storage: 256Mi  
 accessModes:  
 - ReadWriteOnce  
 hostPath:  
 path: /tmp/db

I am using a local path /tmp/db on the host as the disk path for simplicity. It would be better if we use any persistent disk like gcePersistentDisk.

myuser@master1:~$ kubectl create -f mongo-pv.yaml  
persistentvolume/mongo-pv created  
myuser@master1:~$ kubectl get pv  
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE  
mongo-pv 256Mi RWO Retain Available 65s

You can notice that the status of the volume is Available for claim.

4. mongo-pvc PersistentVolumeClaim:

This is used to claim/obtain the storage created above and can be mounted on the mongo container. It is defined in mongo-pvc.yaml.

apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
 name: mongo-pvc  
spec:  
 accessModes:  
 - ReadWriteOnce  
 resources:  
 requests:  
 storage: 256Mi

Create the PersistentVolumeClaim and you can see that the status of the volume is now changed to Bound.

myuser@master1:~$ kubectl create -f mongo-pvc.yaml  
persistentvolumeclaim/mongo-pvc created  
myuser@master1:~$ kubectl get pvc  
NAME STATUS VOLUME CAPACITY ACCESS MODES STORAGECLASS AGE  
mongo-pvc Bound mongo-pv 256Mi RWO 20s  
myuser@master1:~$ kubectl get pv  
NAME CAPACITY ACCESS MODES RECLAIM POLICY STATUS CLAIM STORAGECLASS REASON AGE  
mongo-pv 256Mi RWO Retain Bound default/mongo-pvc 6m57s

5. mongo Deployment:

The mongo.yaml, similar to the tasksapp.yaml, is where we define the mongo deployment that creates a single instance of MongoDB server. Here, we expose the port 27017 which can be accessed by other pods. The persistent volume claimed can be mounted onto a directory on the container.

apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: mongo  
spec:  
 selector:  
 matchLabels:  
 app: mongo  
 template:  
 metadata:  
 labels:  
 app: mongo  
 spec:  
 containers:  
 - name: mongo  
 image: mongo  
 ports:  
 - containerPort: 27017  
 volumeMounts:  
 - name: storage  
 mountPath: /data/db  
 volumes:  
 - name: storage  
 persistentVolumeClaim:  
 claimName: mongo-pvc

Create the mongo deployment using kubectl create.

myuser@master1:~$ kubectl create -f mongo.yaml  
deployment.apps/mongo created  
myuser@master1:~$ kubectl get deployments  
NAME READY UP-TO-DATE AVAILABLE AGE  
mongo 1/1 1 1 59s  
tasksapp 3/3 3 3 44m

6. mongo Service:

This service, defined by mongo-svc.yaml, is similar to the one created for our app except the fact that it is of type ClusterIP(default type of Service in Kubernetes). This service makes the mongo pod accessible from within the cluster, but not from outside. The only resource that should have access to the MongoDB database is our app.

apiVersion: v1  
kind: Service  
metadata:  
 name: mongo  
spec:  
 selector:  
 app: mongo  
 ports:  
 - port: 27017  
 targetPort: 27017

Here, the port 27017 of the service mongo-svc is bound to the port 27017 of the mongo pod attached to it.

myuser@master1:~$ kubectl create -f mongo-svc.yaml  
service/mongo created  
myuser@master1:~$ kubectl get svc mongo  
NAME TYPE CLUSTER-IP EXTERNAL-IP PORT(S) AGE  
mongo ClusterIP 10.100.83.25 <none> 27017/TCP 8s

We can test whether the mongo pod is up and running using the following command.

myuser@master1:~$ curl 10.100.83.25:27017  
It looks like you are trying to access MongoDB over HTTP on the native driver port.

**Part 4: Testing your application**

**Prerequisite:** curl is to be installed on the master node.

Now that we can access our application at 10.103.82.106:8080, lets try to test all the functionalities that our app offers.

myuser@master1:~$ curl 10.103.82.106:8080  
{  
 "message": "Welcome to Tasks app! I am running inside tasksapp-645b67dfbc-ncrt7 pod!"  
}myuser@master1:~$ curl 10.103.82.106:8080/tasks  
{  
 "data": []  
}myuser@master1:~$ curl -X POST -d "{\"task\": \"Task 1\"}" http://10.103.82.106:8080/task  
{  
 "message": "Task saved successfully!"  
}  
myuser@master1:~$ curl -X POST -d "{\"task\": \"Task 2\"}" http://10.103.82.106:8080/task  
{  
 "message": "Task saved successfully!"  
}  
myuser@master1:~$ curl -X POST -d "{\"task\": \"Task 3\"}" http://10.103.82.106:8080/task  
{  
 "message": "Task saved successfully!"  
}myuser@master1:~$ curl 10.103.82.106:8080/tasks  
{  
 "data": [  
 {  
 "id": "5ef8bce7df44b8194ee30c9a",  
 "task": "Task 1"  
 },  
 {  
 "id": "5ef8bcef7bc5ec979ae73a43",  
 "task": "Task 2"  
 },  
 {  
 "id": "5ef8bcf742b0630f70328296",  
 "task": "Task 3"  
 }  
 ]  
}myuser@master1:~$ curl -X PUT -d "{\"task\": \"Task 1 Updated\"}" http://10.103.82.106:8080/task/5ef8bce7df44b8194ee30c9a  
{  
 "message": "Task updated successfully!"  
}myuser@master1:~$ curl 10.103.82.106:8080/tasks  
{  
 "data": [  
 {  
 "id": "5ef8bce7df44b8194ee30c9a",  
 "task": "Task 1 Updated"  
 },  
 {  
 "id": "5ef8bcef7bc5ec979ae73a43",  
 "task": "Task 2"  
 },  
 {  
 "id": "5ef8bcf742b0630f70328296",  
 "task": "Task 3"  
 }  
 ]  
}myuser@master1:~$ curl -X DELETE http://10.103.82.106:8080/task/5ef8bce7df44b8194ee30c9a  
{  
 "message": "Task deleted successfully!"  
}myuser@master1:~$ curl 10.103.82.106:8080/tasks  
{  
 "data": [  
 {  
 "id": "5ef8bcef7bc5ec979ae73a43",  
 "task": "Task 2"  
 },  
 {  
 "id": "5ef8bcf742b0630f70328296",  
 "task": "Task 3"  
 }  
 ]  
}myuser@master1:~$ curl -X POST http://10.103.82.106:8080/tasks/delete  
{  
 "message": "All Tasks deleted!"  
}myuser@master1:~$ curl 10.103.82.106:8080/tasks  
{  
 "data": []  
}