# Cloud Development and Deployment Project

## Summary of work

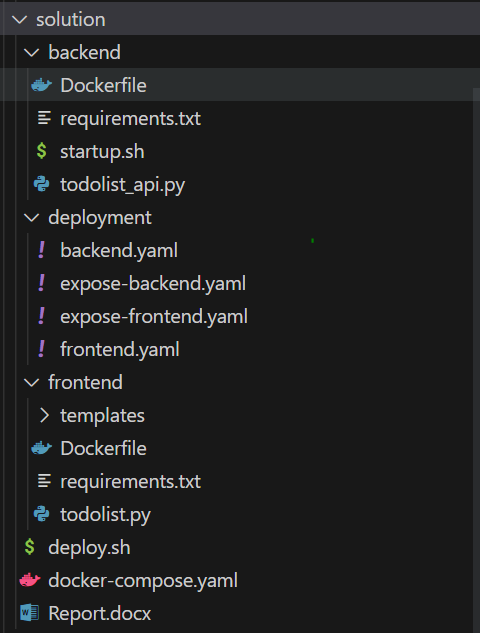
Extended the project done as homework #4 with the following features

1. Using **docker compose** for running the app (both web and api) locally
2. Deploying apps to Kubernetes Cluster using **declarative commands** and **Deployment YAML**

## Detailed Report

#### Folder Structure Changes

In the previous home work submission, All the source code where contained in a single folder. Now they are rearranged as shown below:



To have a better maintainable solution, I have split the source code into 3 directories namely:

1. frontend
2. backend
3. deployment

The backend folder now contains

1. **todolist\_api.py -** Flask app that hosts the /api/\* resources
2. **requirements.txt -** Python app dependencies
3. **startup.sh -** Bash file to download initial SQLite (DB) file and then runs the app
4. **Dockerfile** - Instructions for docker to create image

The frontend folder now contains

1. **todolist.py -** Flask app that hosts User Interface
2. **templates/index.html** - Template html files for flask serve
3. **requirements.txt -** Python app dependencies

The deployment folder now contains

1. **\*.yaml -** Files for Kubernetes Objects Deployment in Cluster

Other files:

1. **docker-compose.yaml** - Contains configuration to build images and run apps with Docker locally
2. **deploy.sh** - Bash script with line-by-line instructions to deploy the app

#### Dockerizing Frontend and Backend App

The frontend app was already dockerized in the last homework submission. I have made a few changes again to improve few places.

**Changes made to Frontend Dockerfile:**

1. The new containerized app will run with the **latest version of python** (v3.11)
2. **Reduced the image size**. The base image used is **python:3.11-slim** around 140MB in size. Makes the push and deployment more faster.
3. All the app dependencies will be installed from **requirements.txt** file now. Easier dependency management.

To build the docker image:

cd frontend

docker build -t zhouji2018/cisc5550todoapp --build-arg api\_ip=localhost .

In the previous submission for backend only startup.sh script was made available to build and run the api solution. I have migrated the instructions to building docker image appropriately.

**New Dockerfile for Backend API:**

1. The new containerized app will run with the **latest version of python** (v3.11) with same base image used as **python:3.11-slim.**
2. All the app dependencies will be installed from **requirements.txt** file now. Easier dependency management.
3. I have modified the startup.sh script and used it as the entry command for this docker image. So once the docker image is build and then run as a container, the app **first downloads** the fresh Sqlite **DB file** and then runs the App. Commands are given below:

#!/bin/bash

wget http://storm.cis.fordham.edu/ji/cisc5550cloud/hw4/todolist.db

python todolist\_api.py

To build the docker image:

cd backend

docker build -t zhouji2018/cisc5550todoapi .

#### Docker Compose File

Docker compose is a tool which works on top of Docker. Mostly used to run and debug the docker containers locally. It makes it much easier to manage multiple containerized apps in docker along with volume and network configurations.

Currently I have 2 dockerized images, if need to have the app working end-to-end locally we should follow the below steps:

1. Build the frontend image
2. Build backend image
3. Create a docker network
4. Run the backend container inside the new network, exposing port 5001
5. Create an alias Record (DNS) type name link to the backend container. This allows other containers to communicate with backend container via a hostname (instead of IPs).
6. Run the frontend container inside the same network, exposing port 5000

Instead of doing all the above steps, a single configuration file is written inside of docker-compose.yaml. Kindly refer the updated file in source code.

services:

  web: -> frontend service

    image: zhouji2018/cisc5550todoapp

    build:

      context: frontend

      args:

        api\_ip: api

    ports:

      - '5000:5000'

  api: -> backend service

    image: zhouji2018/cisc5550todoapi

    build:

      context: backend

    ports:

      - '5001:5001'

The above config **creates a new network by default** and run all containers inside of it. Also all the containers are allowed to be **communicated via** their host name (I.e) their service names (**web** and **api**). Now you could relate why the build arguments for frontend is written as api.

**Some commands to work with**

To check version

docker compose version

To build images

docker compose build

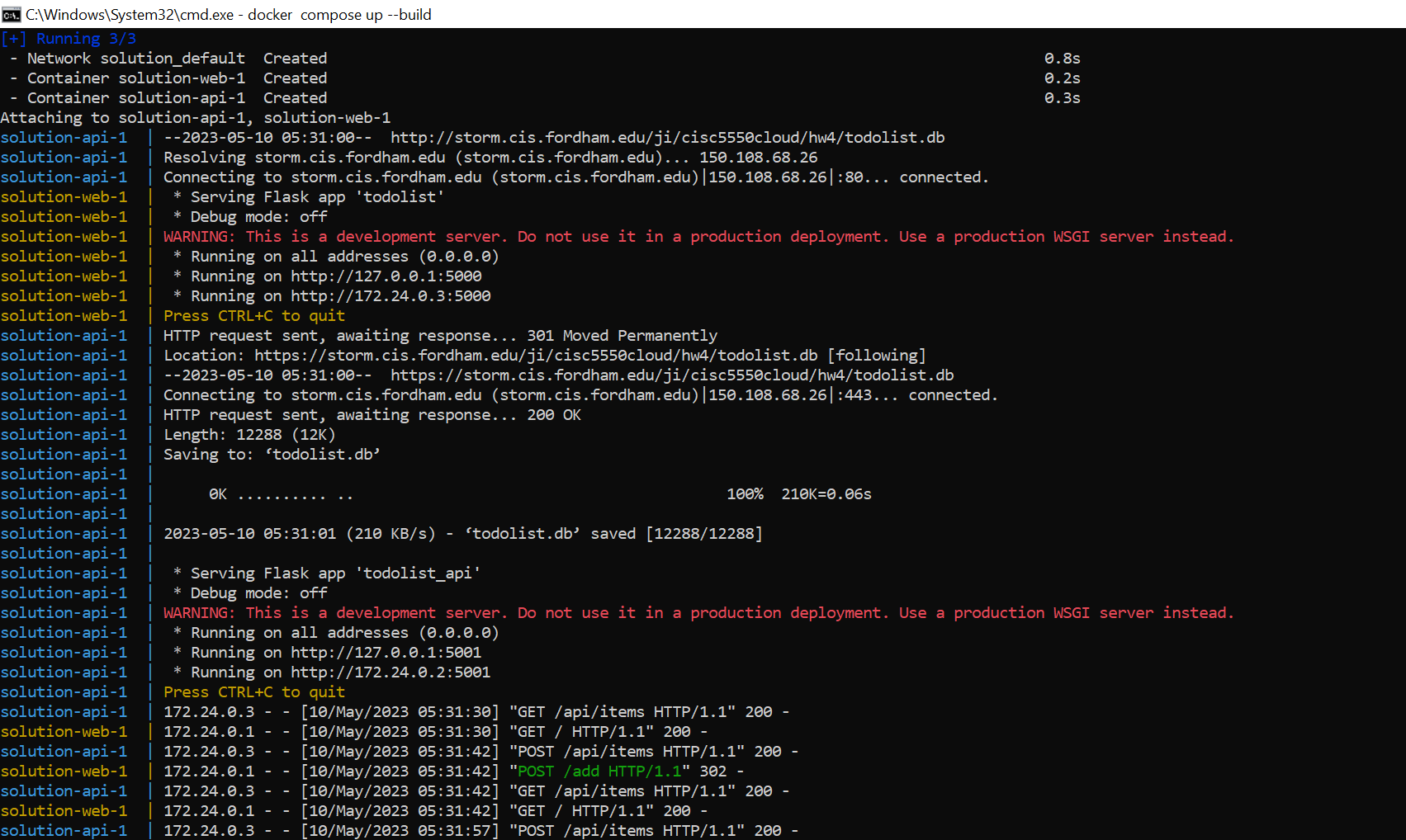
To run app locally

docker compose up

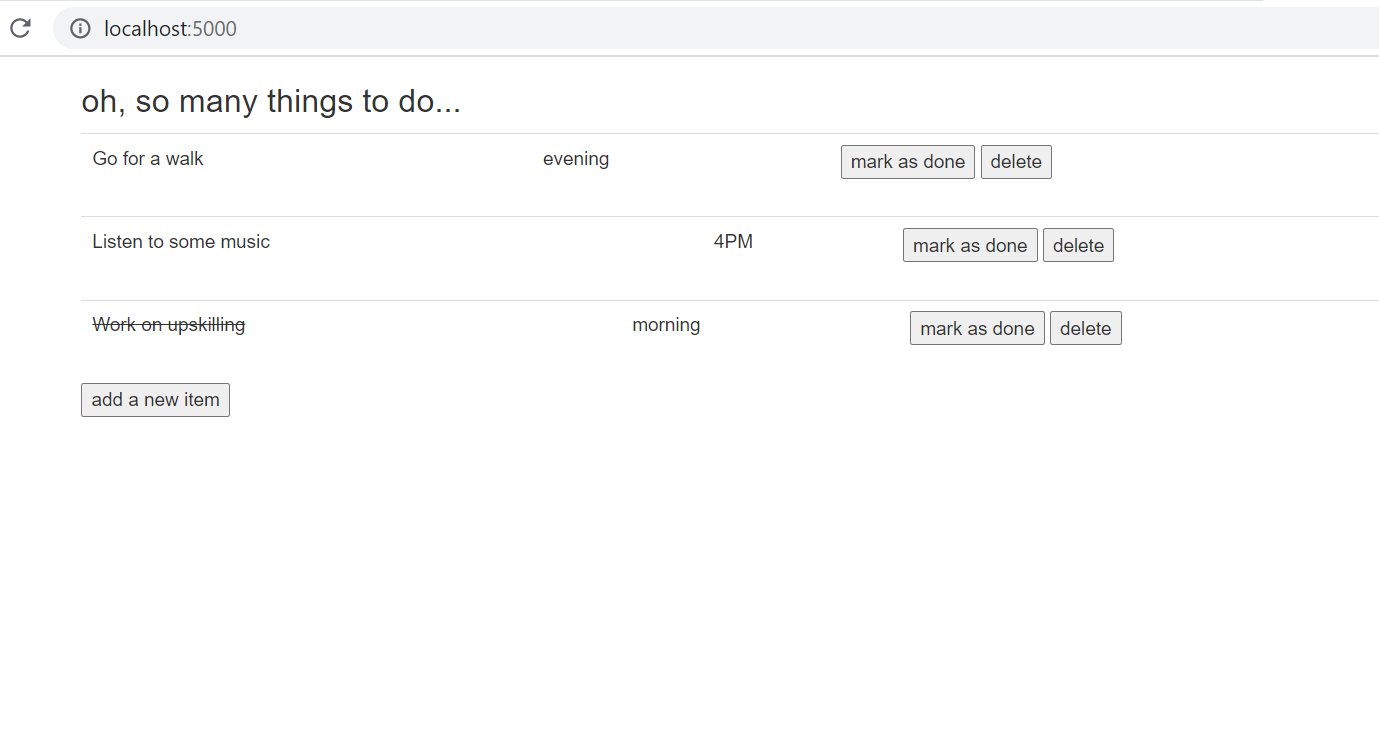
To build & run app locally

docker compose up --build

Application Logs:



The user interface for the app becomes available at port 5000 of localhost. Attached screenshots of some tasks added during development.



To clean up the local app

docker compose down --rmi all

#### Kubernetes YAML

In the previous submission, the final GKE deployment happens by running the below commands:

kubectl create deployment cc5550 --image=zhouji2018/cisc5550todoapp --port=5000

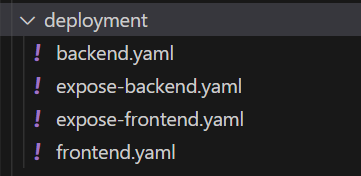
kubectl expose deployment cc5550 --type="LoadBalancer"

These commands are usually self understandable and are called the imperative commands. The drawbacks of using these are they are hard to cleanup.

For example, if we created 5 deployments with imperative commands we have to keep track of them and delete exact deployments for cleaning up.

A great way to improve this scenario is to use **declarative commands & k8s object YAML** files. This is very similar to Infrastructure as a Code concept, with only exception that we create kubernetes objects instead of cloud resources.

Kindly refer to the `deployment` folder inside of source code which contains all the required YAML files to deploy our solution.



The **backend.yaml** contains deployment configuration which creates a single backend pod.

The **frontend.yaml** contains deployment configuration which creates a single frontend pod.

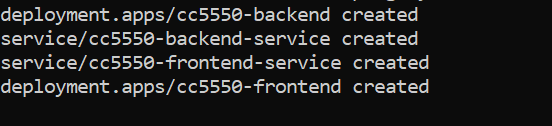
The **expose-backend.yaml** contains service configuration which exposes all of our backend pods to the cluster. The service is of type **ClusterIP** which means all pods inside the same namespace can communicate to the backend pod via a host name (i.e the service name). Backend pod is not exposed to external network.

The **expose-frontend.yaml** contains service configuration which exposes all of our frontend pods to the end user. The service is of type **LoadBalancer** which means a port gets allocated on external loadbalancer (IP) which can be used to see the user interface (connects to frontend pods).

Create Kubernetes Objects using declarative command:

kubectl apply -f deployment

Logs:

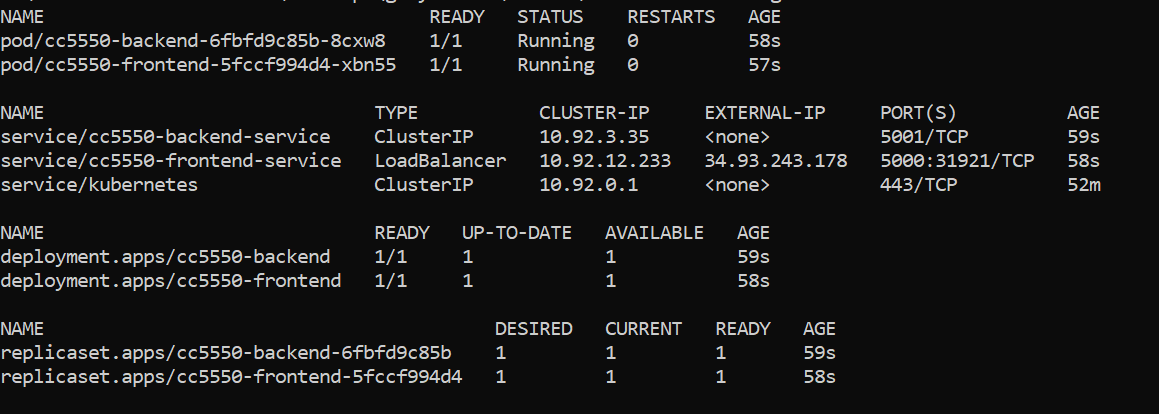


Scaling of backend pod is not trivial as SQLite file is not shared between multiple backend pods. One possible solution can be to to create a K8s persistent volume and mount the file to share across the pods. I have not implemented this due to time constraints.

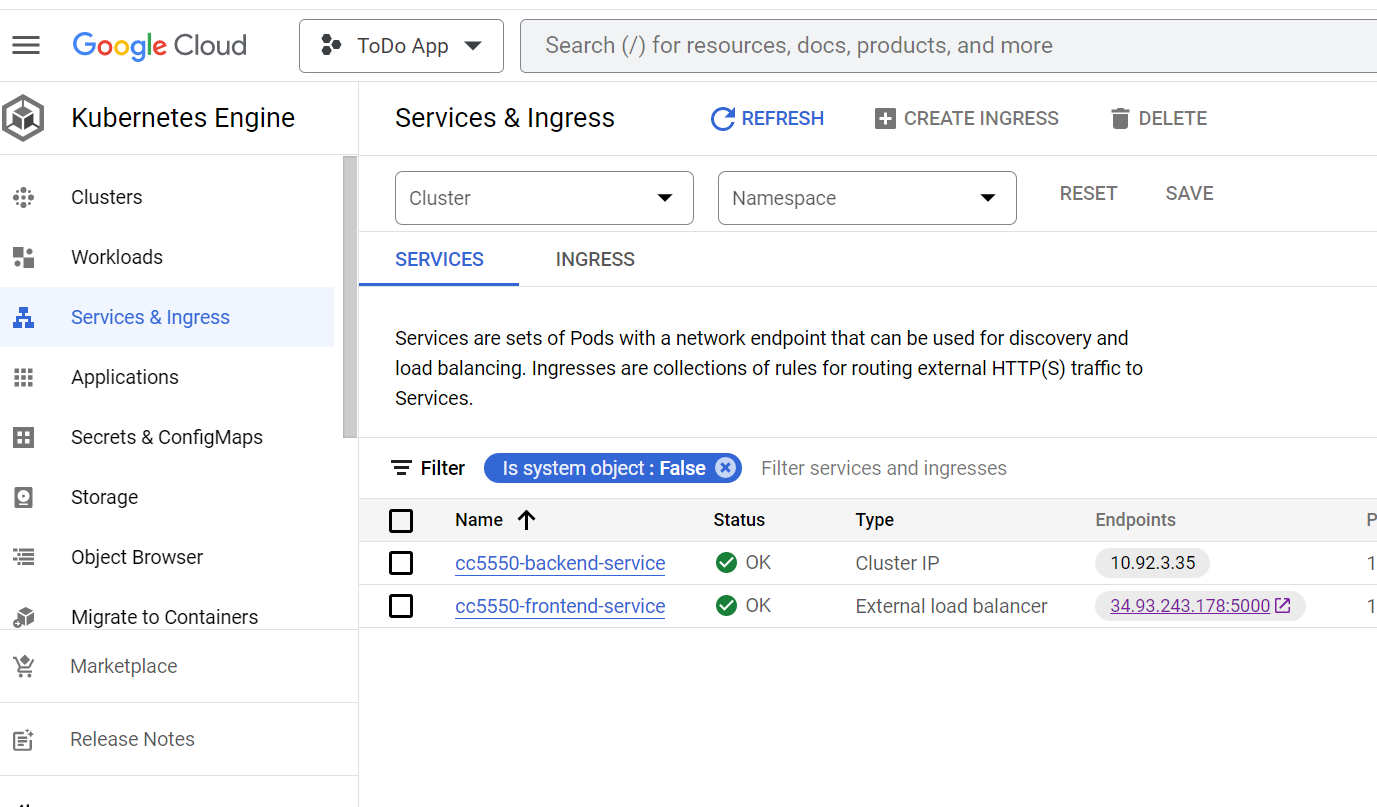
See all Kubernetes Objects:

kubectl get all

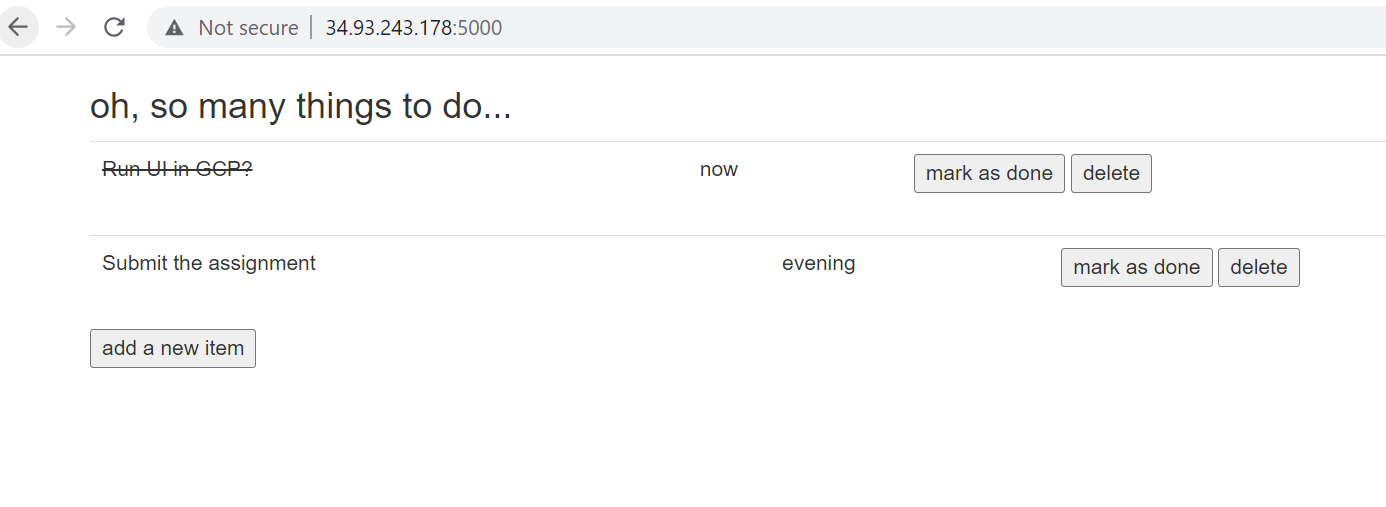
Logs:



Look for the external IP in cc5550-frontend-service to access user interface. You can also get this access via GCP Console.



App User Interface in GCP K8s Cluster:



Cleanup Kubernetes Objects using declarative command:

kubectl delete -f deployment

## Conclusion

Sharing final thoughts & Outcomes for this assignment,

* the project was extended into a **maintainable solution with directories** corresponding to different apps with **better dependency management**. Now we can have separate Frontend & Backend teams working under their respective work stream.
* An **easy local build and run** option is provided via docker compose tool. Just docker and docker-compose tool is sufficient to run the app. New developers don’t have to install any additional tools to run the app. Docker compose is also a good option to deploy inside of development environments but **not recommended for production.**
* A more reliable deployment way was developed by means of Kubernetes YAML files, All the required K8s objects can be deployed and cleared from a cluster with simple commands. This makes it **easier to integrate with tools like Helms & Helmsman**. One important note is that the **scaling of backend pod is not recommended** as the data is not yet stored in a persistent storage solution.