**Cloud Development and Deployment Project**

CISC5550: Cloud Computing

Arna Sadia

**Summary**

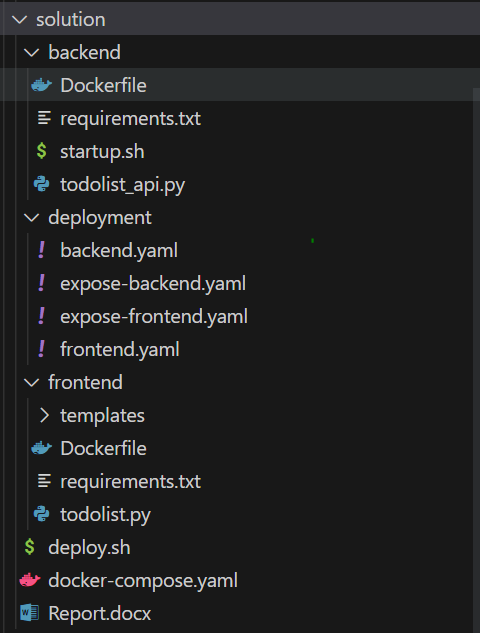
In this project, I have extended homework #4 with the following features:

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**

**Report**

1. **Folder Structure Changes**

The previous homework submission contained all the source codes 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. backend
2. frontend
3. deployment

**The backend folder:**

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

**The frontend folder:**

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:**

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
3. **Dockerizing Front-End and Back-End App**

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

**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 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 arnasadia/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 build the docker image appropriately.

**New Docker file for Backend API:**

1. The new containerized app will run with the latest version of Python (v3.11) with the same base image used as **python:3.11-slim.**
2. All the app dependencies will be installed from the requirements.txt file now which makes dependency management easier.
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 built 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 arnasadia/cisc5550todoapi .

1. **Docker Compose File**

Docker compose is a tool that 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, in order to have the app working end-to-end locally, the below steps are followed:

1. Build a frontend image
2. Build a 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 the 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. The updated file is located in the source code.

services:

  web: -> frontend service

    image: arnasadia/cisc5550todoapp

    build:

      context: frontend

      args:

        api\_ip: api

    ports:

      - '5000:5000'

  api: -> backend service

    image: arnasadia/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 we can relate to why the build arguments for the frontend are written as API.

**Some commands to work with:**

To check version

docker compose version

To build images:

docker compose build

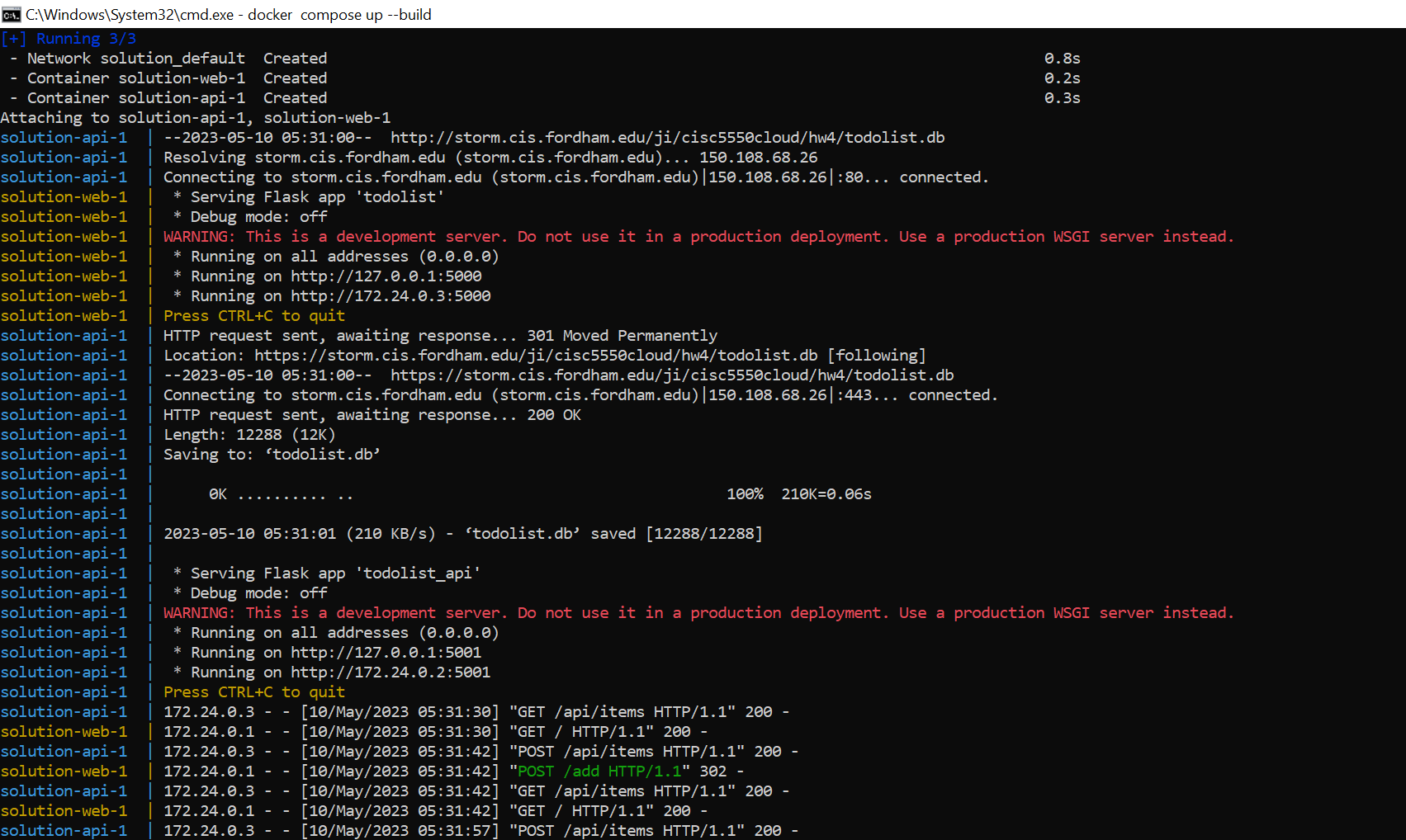
To run the app locally:

docker compose up

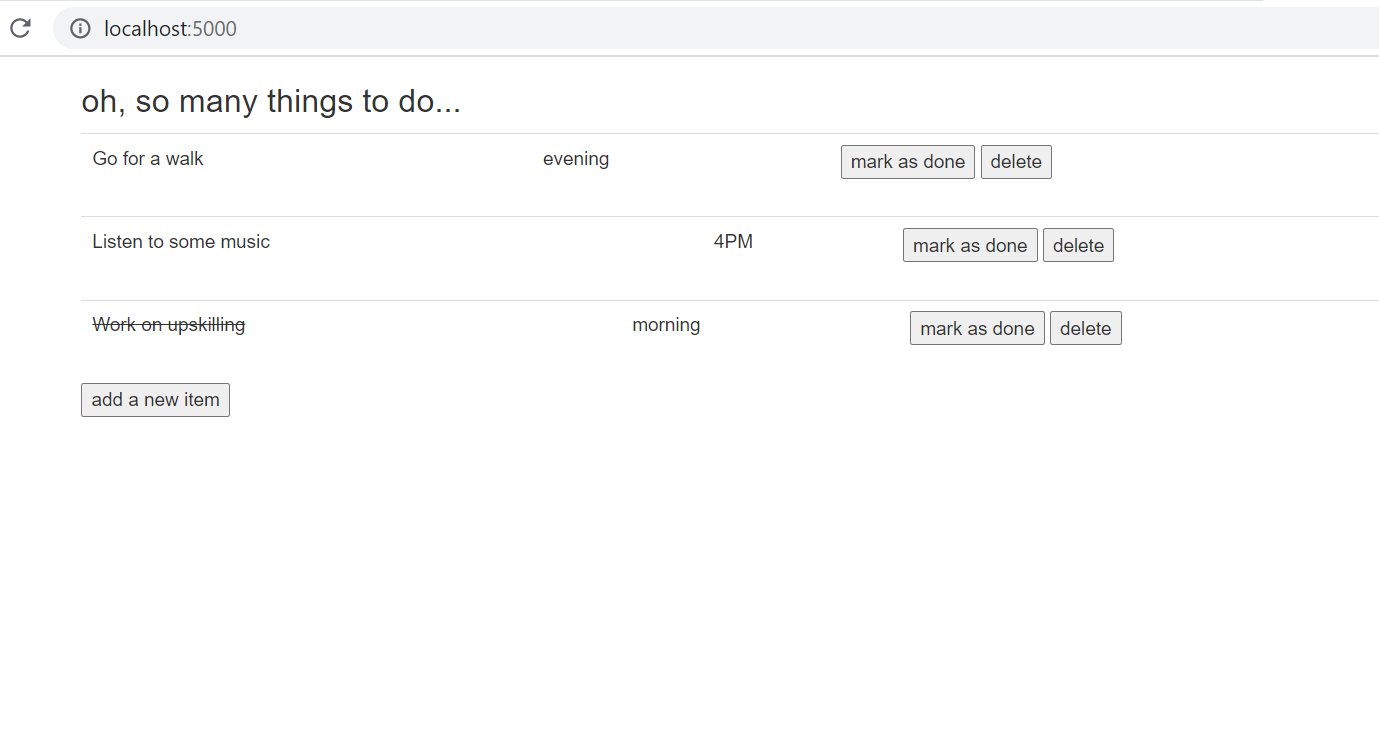
To build & run the app locally:

docker compose up --build

Application Logs:



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



To clean up the local app:

docker compose down --rmi all

1. **Kubernetes YAML**

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

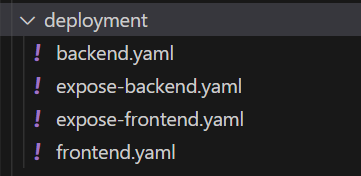
kubectl create deployment cc5550 --image=arnasadia/cisc5550todoapp --port=5000

kubectl expose deployment cc5550 --type="LoadBalancer"

These commands are usually self-understandable and are called imperative commands. The drawbacks of using these are they are hard to clean up. 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 the only exception that we create Kubernetes objects instead of cloud resources.

The `deployment` folder inside the source code contains all the required YAML files to deploy the 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.

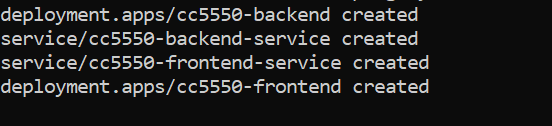
The **expose-backend.yaml** contains a service configuration that exposes all of our backend pods to the cluster. The service is of type **ClusterIP** which means all pods inside the same namespace can communicate to the backend pod via a hostname (i.e. the service name). The backend pod is not exposed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

The **expose-frontend.yaml** contains a 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 an external loadbalancer (IP) which can be used to see the user interface (connects to frontend pods).

Create Kubernetes Objects using the declarative command:

kubectl apply -f deployment

Logs:

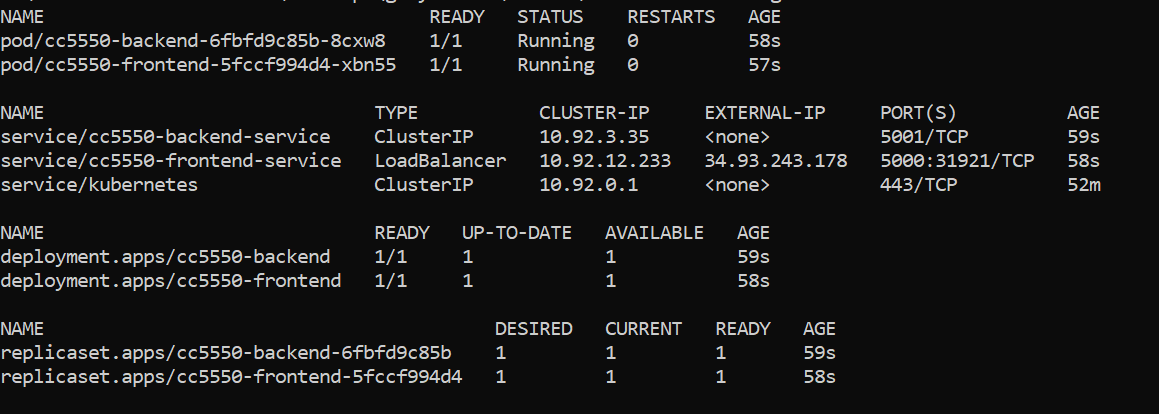


Scaling of the backend pod is not trivial as the SQLite file is not shared between multiple backend pods. One possible solution can be to create a K8s persistent volume and mount the file to share across the pods.

See all Kubernetes Objects:

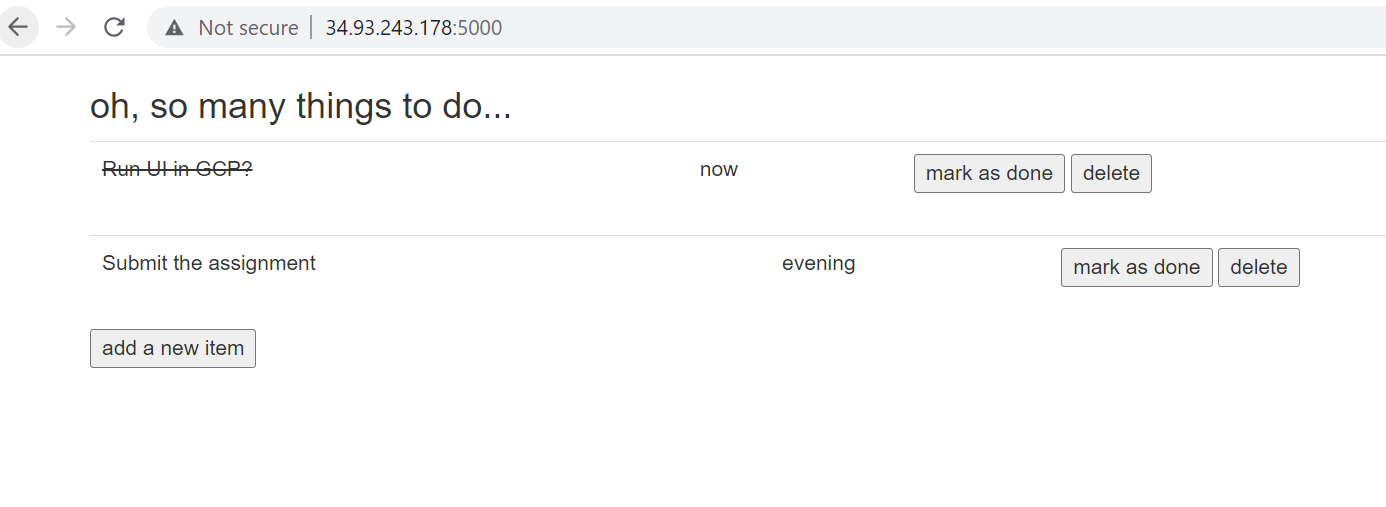
kubectl get all

Logs:



The external IP in cc5550-frontend-service can be used to access the user interface.

App User Interface in GCP K8s Cluster:



Cleanup Kubernetes Objects using the declarative command:

kubectl delete -f deployment

1. **Conclusion**

Here are some final thoughts & outcomes:

* The project was extended into a maintainable solution with directories corresponding to different apps with better dependency management. Now I can have separate Frontend & Backend teams working under their respective work stream.
* An easy local build and run option is provided via the docker compose tool. Just docker and docker-compose tool is sufficient to run the app. Docker compose is also a good option to deploy inside of development environments but is 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.