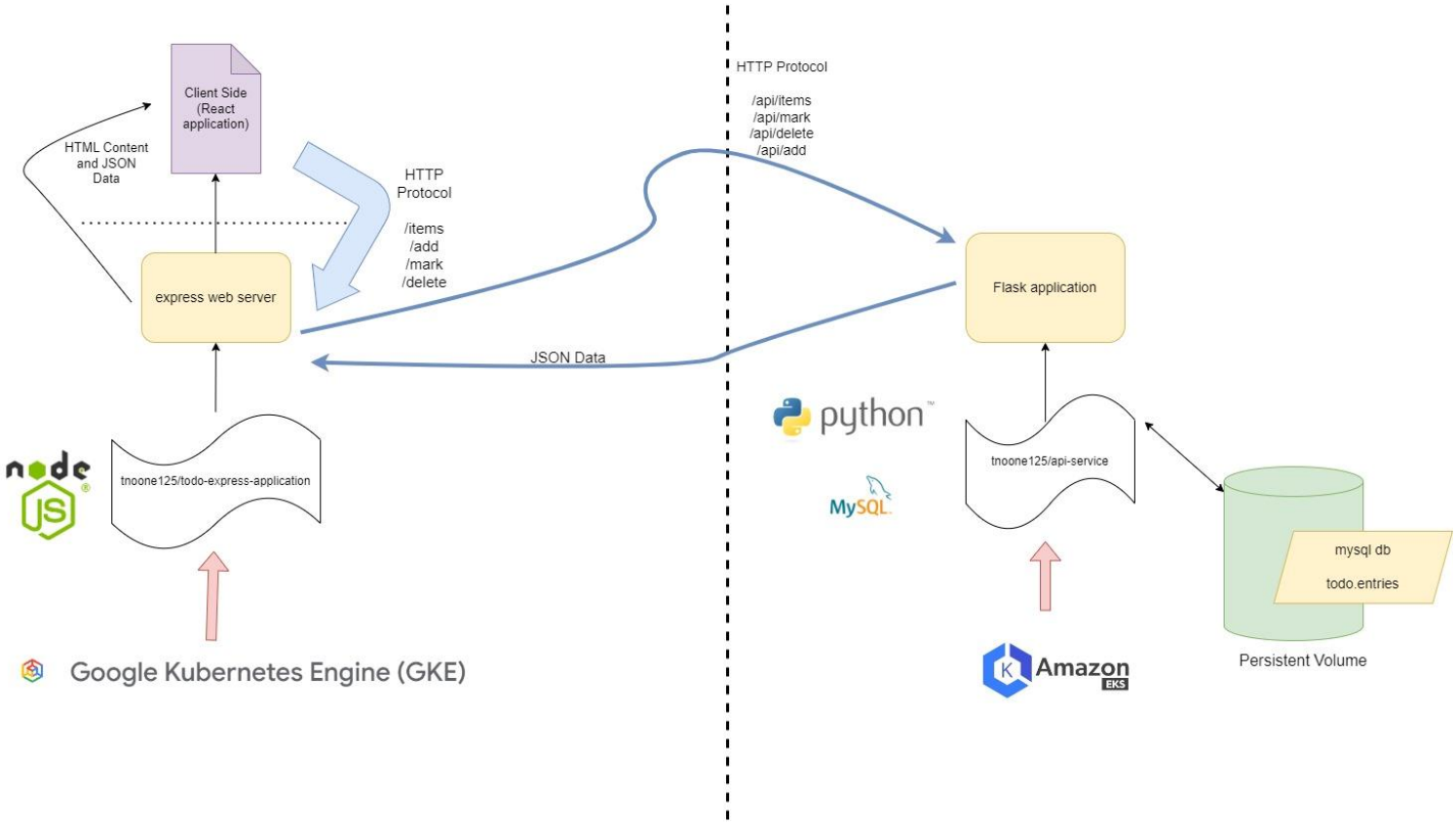
GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

Thomas Noone – CISC 5550 Cloud Computing – Final Project Write Up

Overall Goal

My goal was to make several enhancements to the Todo application. Firstly, since I have extensive experience with React, I wanted to build the front-end application using a framework different than Flask. I also wanted to deploy the API service with a persistent volume rather than using a simple .db file. Finally, I wanted to experiment with a different cloud provider, so I would attempt to deploy the API service to an AWS EKS cluster, while still deploying the web server to a GCloud cluster.



Developing API Service

To begin, I simply copied over my todolist_api.py flask application from Homework 4. The only changes I made were to the app's config to connect via MySQL rather than SQLite3. If I had more time, I would have used some sort of secrets API rather than putting the MySQL db user password directly into my code:

```
from flask import Flask, g, request, jsonify
import mysql.connector

app = Flask(__name__)
app.config.from_object(__name__)

app.config['MYSQL_HOST'] = 'mysql'
app.config['MYSQL_USER'] = 'todouser'
app.config['MYSQL_PASSWORD'] = 'p@ss123'
app.config['MYSQL_DB'] = 'todo'
```

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

To Dockerize this application, I created a Dockerfile based on the python:3.9-slim image (this is smaller and leads to faster deployment time). My RUN commands included installing a mysql client and pip installing flask and mysql-connector-python.

Using Docker compose, I described a container running mysql and is a dependency of the container running the Python Flask application. The mysql container is attached to a volume and I was able to provide an entries.sql script which would run upon the creation of the volume. In my case, this .sql script creates the entries table used by the application.

`docker compose up –build -d` rebuilds the images and runs the containers.

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>docker compose up --build -d
[+] Building 25.2s (13/13) FINISHED                                                                    docker-container:multi-platform-builder
 => [api-service internal] load build definition from Dockerfile                                                                        0.1s
 => => transferring dockerfile: 466B                                                                                                    0.0s
 => [api-service internal] load metadata for docker.io/library/python:3.9-slim                                                          1.4s
 => [api-service internal] load .dockerignore                                                                                           0.0s
 => => transferring context: 2B                                                                                                         0.0s
 => [api-service internal] load build context                                                                                           0.0s
 => => transferring context: 737B                                                                                                       0.0s
 => [api-service 1/7] FROM docker.io/library/python:3.9-slim@sha256:990eb8f30571e0aaabfa1a7deb9fe2f6c1d4163a389004fbf823db6fee7c3642     0.0s
 => => resolve docker.io/library/python:3.9-slim@sha256:990eb8f30571e0aaabfa1a7deb9fe2f6c1d4163a389004fbf823db6fee7c3642                 0.0s
 => CACHED [api-service 2/7] WORKDIR /usr/src/app                                                                                       0.0s
 => CACHED [api-service 3/7] RUN apt-get update     && apt-get upgrade -y     && apt-get install -y gcc default-libmysqlclient-dev pkg-config   0.0s
 => CACHED [api-service 4/7] RUN pip install mysqlclient                                                                                0.0s
 => CACHED [api-service 5/7] RUN pip install flask                                                                                      0.0s
 => CACHED [api-service 6/7] RUN pip install mysql-connector-python                                                                     0.0s
 => [api-service 7/7] COPY . .                                                                                                          0.1s
 => [api-service] exporting to docker image format                                                                                     23.3s
 => => exporting layers                                                                                                                 0.1s
 => => exporting manifest sha256:f0ef4ff79d512d5504cd1f05061896571dde56a38ec4e0d2e44ba154472bea3e                                       0.0s
 => => exporting config sha256:aaba463b54702a3a42b4b130a6ea7c5527ae9b8a2310e729b151d6ab9b5d1a14                                         0.0s
 => => sending tarball                                                                                                                 23.1s
 => [api-service] importing to docker                                                                                                   0.4s
 => => loading layer cd7826d9cd11 1.70kB / 1.70kB                                                                                       0.4s
[+] Running 4/4
 ✔Network api_default          Created                                                                                                  0.4s
 ✔Volume "api_mysql-data"      Created                                                                                                  0.0s
 ✔Container api-mysql-1        Started                                                                                                  4.6s
 ✔Container api-api-service-1  Started                                                                                                  6.3s
```

Now when I run docker ps -a, I see all the running containers, notice the first two. The first is the container for the api-service and the second is for mysql.

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>docker ps -a
CONTAINER ID   IMAGE                          COMMAND                  CREATED          STATUS                      PORTS                                       NAMES
186e0e4606ab   api-api-service                "python todolist_api…"   32 seconds ago   Up 27 seconds               0.0.0.0:5001->5001/tcp                      api-api-service-1
a74ff25400b3   mysql:5.7                      "docker-entrypoint.s…"   32 seconds ago   Up 31 seconds               0.0.0.0:3306->3306/tcp, 33060/tcp           api-mysql-1
abda16451480   moby/buildkit:buildx-stable-1  "buildkitd --allow-i…"   35 hours ago     Up About an hour                                                        buildx_buildkit_multi-platform-builder0
165b0600a1f6   daa33a734682                   "docker-entrypoint.s…"   2 days ago       Exited (255) 30 hours ago   0.0.0.0:5000->5000/tcp                      festive_antonelli
d003157f915f   daa33a734682                   "docker-entrypoint.s…"   2 days ago       Exited (255) 30 hours ago   5000/tcp                                    mystifying_easley
aeb6f4d8a03c   dca57840315a                   "docker-entrypoint.s…"   2 days ago       Exited (255) 30 hours ago   5000/tcp                                    nice_villani

C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>curl http://localhost:5001/api/items
[]
```

I executed curl requests to make sure that the service operated properly:

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

```
thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl -X POST -H "Content-Type: application/json" -d '{"what_to_do": "homework!
!", "due_date": "neva"}' http://localhost:5001/api/add
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    65  100    17  100    48     80    227 --:--:-- --:--:-- --:--:--   309{"
success":true}


thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl http://localhost:5001/api/items
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    60  100    60    0     0   1077      0 --:--:-- --:--:-- --:--:--  1090[{
"due_date":"neva","status":"","what_to_do":"homework!!"}]
```

```
thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl -X POST -H "Content-Type: application/json" -d '{"what_to_do": "project writeup", "due_date": ""}' http://localhost:5001/api/add
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    66  100    17  100    49     97    282 --:--:-- --:--:-- --:--:--   381{"success":true}


thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl -X PUT -H "Content-Type: application/json" -d '{"item": "homework!!"}' http://localhost:5001/api/mark
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    39  100    17  100    22    177    229 --:--:-- --:--:-- --:--:--   410{"success":true}


thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl http://localhost:5001/api/items
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   123  100   123    0     0   2301      0 --:--:-- --:--:-- --:--:--  2277[{"due_date":"neva","status":"done","what_to_do":"homework!!"},{"due_date":"","status":"","what_to_do":"project writeup"}]


thoma@LAPTOP-NPVECTV0 MINGW64 ~
```

Now that my container works locally, I tagged the image and pushed to DockerHub:

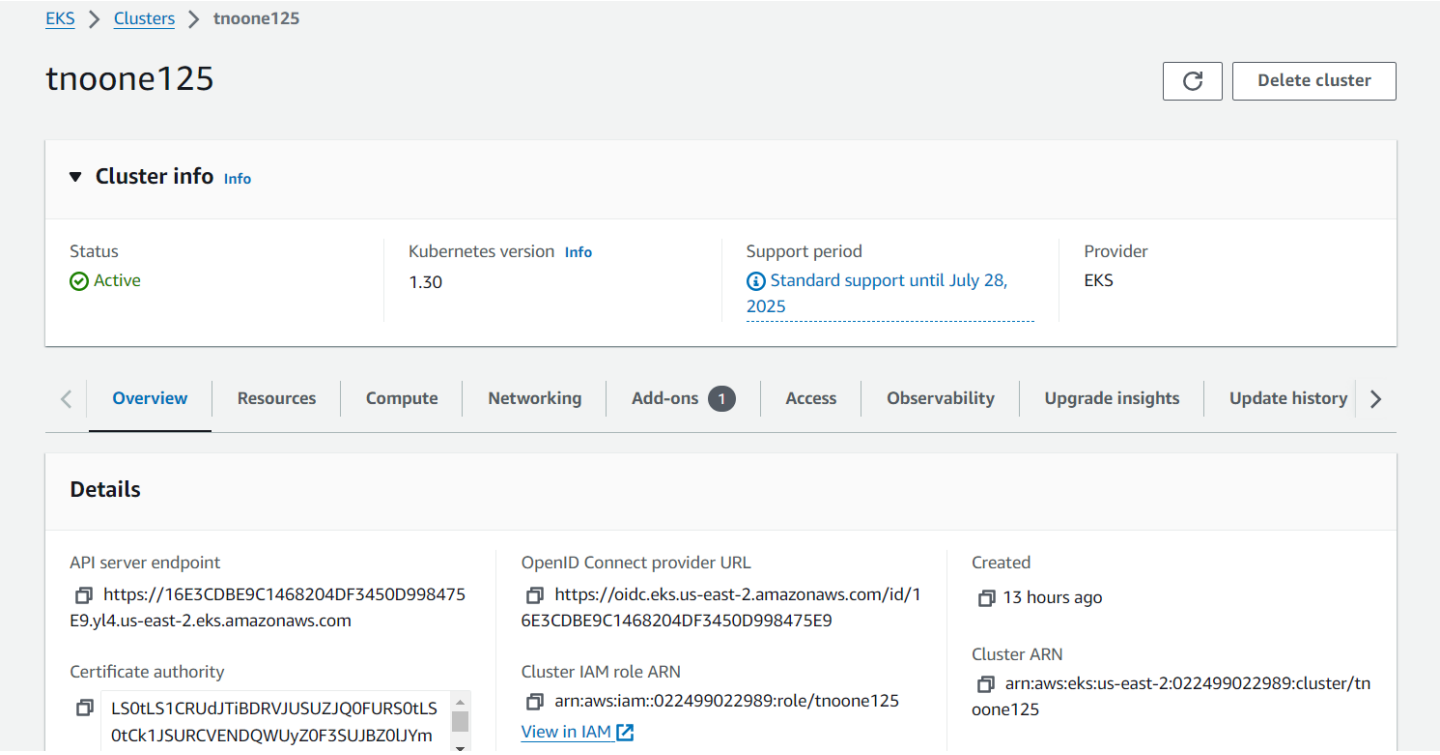
```
docker tag api-api-service:latest tnoone125/api-service:latest

docker push tnoone125/api-service:latest
```

Deploy to Amazon Web Services – EKS with kubectl

I now wanted to experiment with a different cloud provider, so I created an AWS EKS Cluster. In my opinion, the dashboard is cleaner than the Google Cloud console:

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project



Here is me creating a cluster via the command line:

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws>eksctl create cluster --name cisc5550-api-service --region us-east-2 --version 1.30
2024-08-05 00:32:18 [ℹ]  eksctl version 0.188.0
2024-08-05 00:32:18 [ℹ]  using region us-east-2
2024-08-05 00:32:18 [ℹ]  setting availability zones to [us-east-2c us-east-2b us-east-2a]
2024-08-05 00:32:18 [ℹ]  subnets for us-east-2c - public:192.168.0.0/19 private:192.168.96.0/19
2024-08-05 00:32:18 [ℹ]  subnets for us-east-2b - public:192.168.32.0/19 private:192.168.128.0/19
2024-08-05 00:32:18 [ℹ]  subnets for us-east-2a - public:192.168.64.0/19 private:192.168.160.0/19
2024-08-05 00:32:18 [ℹ]  nodegroup "ng-3b1258d8" will use "" [AmazonLinux2/1.30]
2024-08-05 00:32:18 [ℹ]  using Kubernetes version 1.30
2024-08-05 00:32:18 [ℹ]  creating EKS cluster "cisc5550-api-service" in "us-east-2" region with managed nodes
2024-08-05 00:32:18 [ℹ]  will create 2 separate CloudFormation stacks for cluster itself and the initial managed nodegroup
2024-08-05 00:32:18 [ℹ]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --region=us-east-2 --cluster=cisc5550-api-service
'
2024-08-05 00:32:18 [ℹ]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for cluster "cisc5550-api-service" in "us-east-2"
2024-08-05 00:32:18 [ℹ]  CloudWatch logging will not be enabled for cluster "cisc5550-api-service" in "us-east-2"
```

My goal is to use kubectl commands to deploy this API service to my cisc5550-api-service cluster.

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

I found a very cool CLI tool called “kompose" (https://kubernetes.io/docs/tasks/configure-pod-container/translate-compose-kubernetes/) which translates docker-compose.yml files to appropriate deployment/service.yaml files that can be applied in kubectl commands.

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kompose convert
WARN Failed to check if the directory is empty: readdir C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api\entries.sql: The system cannot find the path specified.
WARN Skip file in path C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api\entries.sql
WARN File don't exist or failed to check if the directory is empty: CreateFile :/var/lib/mysql: The filename, directory name, or volume label syntax is incorrect.
INFO Kubernetes file "api-service-service.yaml" created
INFO Kubernetes file "mysql-service.yaml" created
INFO Kubernetes file "api-service-deployment.yaml" created
INFO Kubernetes file "mysql-deployment.yaml" created
INFO Kubernetes file "mysql-data-persistentvolumeclaim.yaml" created
```

Because I want the storage to last even when pods go down, I want to deploy a **persistent volume**. A persistent volume claim (PVC) which will request storage resources from the cluster. In this claim I specify that only one pod can write to storage at a time.

Checkout mysql-data-persistentvolumeclaim.yaml to see the full description of the PVC. You cannot create PV/PVC with plain kubectl command line, the .yaml file is necessary.

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kubectl apply -f mysql-data-persistentvolume.yaml
persistentvolume/mysql-data created

C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kubectl apply -f mysql-data-persistentvolumeclaim.yaml
persistentvolumeclaim/mysql-data created
```

Note that the .yaml files generated by kompose were not perfect – I needed to add `storageClassName: manual` to both the PV and PVC yamls, and once I did and re-applied them, the persistent volume was “bound” to my PVC.

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kubectl apply -f mysql-data-persistentvolumeclaim.yaml
persistentvolumeclaim/mysql-data created

C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kubectl describe pvc mysql-data
Name:          mysql-data
Namespace:     default
StorageClass:  manual
Status:        Bound
Volume:        mysql-data
Labels:        io.kompose.service=mysql-data
Annotations:   pv.kubernetes.io/bind-completed: yes
               pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      100Mi
Access Modes:  RWO
VolumeMode:    Filesystem
Used By:       mysql-f9db5d6d9-prsjr
Events:        <none>
```

Next, I ran `kubectl apply -f mysql-deployment.yaml` and `kubectl apply -f mysql-service.yaml`

This created the container which is running mysql, not the Python application. The kompose tool stipulated that this container should be mounted to the `mysql-data` persistent volume – it put this in the deployment.yaml:

```
volumeMounts:
            - mountPath: /var/lib/mysql
              name: mysql-data
      restartPolicy: Always
      volumes:
        - name: mysql-data
          persistentVolumeClaim:
            claimName: mysql-data
```

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

The mysql-deployment.yaml file included the database and user I wanted (todo / todouser) – but I still needed to create the entries table. So before starting the api-service container, I exec'd into the mysql pod:

```
kubectl get pods

kubectl exec mysql-f9db5d6d9-prsjr -it bash


bash-4.2#mysql -u root -proot

>USE todo;

>CREATE TABLE IF NOT EXISTS entries (what_to_do VARCHAR(100) NOT NULL, due_date VARCHAR(100) DEFAULT '', STATUS VARCHAR(100) DEFAULT '');
```

Even though this particular pod `mysql-f9db5d6d9-prsjr` is not persistent, the *volume* is persistent, so I only need to run this SQL command once.

Now it was time to deploy the Python application in the api-service container. Note that the .yaml file generated by compose did not have the proper image tag, so I had to manually edit it. I also had to add

```
type: LoadBalancer
```

to the specification for the service in order to expose the deployment with an external IP.

This also could easily have been done without .yaml files and simply used:

```
kubectl create deployment api-service --image=tnoone125/api-service:latest --port=5001

kubectl expose deployment api-service --type="Load Balancer"

kubectl get service api-service
```

Finally the backend service is completely up and running:

```
C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kubectl get services
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)          AGE
api-service   LoadBalancer   10.100.128.119   a2ae5f2eed9384523901bed8fc1dc923-1564554210.us-east-2.elb.amazonaws.com   5001:32059/TCP   8m7s
kubernetes    ClusterIP      10.100.0.1       <none>                                                                     443/TCP          96m
mysql         ClusterIP      10.100.66.55     <none>                                                                     3306/TCP         63m

C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\api>kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
api-service-65b687568b-4wsl5   1/1     Running   0          7m31s
mysql-f9db5d6d9-prsjr          1/1     Running   0          66m
```

Note that the external IP for the api-service is within the amazonaws domain. I needed to create a firewall rule to allow HTTP traffic to my cluster. First I needed to obtain the security group ID for my cluster:

```
aws ec2 describe-security-groups --filters "Name=tag:aws:eks:cluster-name,Values=cisc5550-api-service"
```

and then I could create the rule:

```
aws ec2 authorize-security-group-ingress --group-id sg-0c525d1a3b450d2b4 --protocol tcp --port 5001 --cidr 0.0.0.0/0
```

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

I tested out the API service through curl –

```
thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl http://a2ae5f2eed9384523901bed8fc1dc923-1564554210.us-east-2.elb.amazonaws.com:5001/api/items
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100     3  100     3    0     0     23      0 --:--:-- --:--:-- --:--:--    23[]


thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl -X POST -H "Content-Type: application/json" -d '{"what_to_do": "front-end", "due_date": "aug 6"}' http://a2ae5f2eed9384523901bed8fc1dc923-1564554210.us-east-2.elb.amazonaws.com:5001/api/add
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    65  100    17  100    48    125    353 --:--:-- --:--:-- --:--:--   481{"success":true}
```

Before moving onto the front-end application, I also needed to test the **persistence** of my storage. Via curl I added an item to the database, but I needed to test that the item remained in the database once a new pod started. In Kubernetes, we generally do not restart pods, we *replace* them. So I ran the following command:

```
kubectl rollout restart deployment api-service
```

This particular command does not lead to any service downtime because it pro-actively starts up a new pod before shutting the old one down. Once again, with curl, I verified that my items were still persisted in the DB:

```
thoma@LAPTOP-NPVECTV0 MINGW64 ~
$ curl http://a2ae5f2eed9384523901bed8fc1dc923-1564554210.us-east-2.elb.amazonaws.com:5001/api/items
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    60  100    60    0     0    397      0 --:--:-- --:--:-- --:--:--   402[{"due_date":"aug 6","status":"","what_to_do":"front-end"}]
```

Developing the Web Server / Front-End Application

There is a handy script one can run to begin developing a ReactJS application:

```
npx create-react-app todo-app
```

It automatically generates a public/ directory with an index.html file and an App.js serving up basic JSX component. JSX is HTML-like mark-up and describes what you want emitted from the browser.

The client-side React application is not enough, though. I also spun up an Express web server which can serve up the client-side webpage and communicate with the API service via HTTP protocol.

The endpoints for the Express web server are structured like so:

```
app.post("/add", async (req, res) => {
        const response = await axios.post(`http://${todo_ip}:5001/api/add`, {
                what_to_do: req.body.what_to_do,
                due_date: req.body.due_date
        });
        …
});
app.put("/mark/:task", async (req, res) => {
        const { task } = req.params;
        const decodedTask = decodeURIComponent(task);

        try {
                const response = await axios.put(`http://${todo_ip}:5001/api/mark`, {
                item: decodedTask,
                });
        } …
});
…
```

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

```
// SERVE UP THE HTML - create a Node.JS stream
// Documentation: https://react.dev/reference/react-dom/server/renderToPipeableStream#rendertopipeablestream
app.get('*', (req, res) => {
    const indexHtmlPath = path.join(buildPath, 'index.html');
    const indexHtml = fs.readFileSync(indexHtmlPath, 'utf8');
    const { pipe, abort: _abort } = ReactDOMServer.renderToPipeableStream(
      <>
          <StaticRouter location={req.url}>
              <App />
          </StaticRouter>
      </>,
      {
        onShellReady() {
          res.statusCode = 200;
          res.setHeader("Content-type", "text/html");
          res.send(indexHtml);
        },
                …
      }
    );
    pipe(res);
  });

app.listen(5000, () => {
  console.log("App is running on http://localhost:5000");
});
```

As for the actual client-side React, I maintained “stateful” representations of the items to display, whether the “add” form should be displayed, and whether we should show a loader icon (if still fetching items). None of these stateful items persist – when the user X’s out the application, this information is gone. I will call out important code snippets:

```
const [entries, setEntries] = useState([]);
const [showEntryForm, setShowEntryForm] = useState(false);
const [showLoader, setShowLoader] = useState(true);

// Upon React Component Mounting, fetch the items from the server
// This just occurs once.
useEffect(() => {
    const fetchItems = async () => {
      try {
        const response = await fetch('/items');
        const data = await response.json();
        setEntries(data.items);
        setShowLoader(false);
      } catch (error) {
        console.error('Error fetching data:', error);
      }
    };

    fetchItems();
}, []);
```

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

This particular way of calling useEffect will cause my client-side application to call the "/items" endpoint from the Express-server immediately upon mounting to the Browser. It will add those items to the local state and stop showing the Loader. For those familiar with older versions of React, this is like `componentDidMount`.

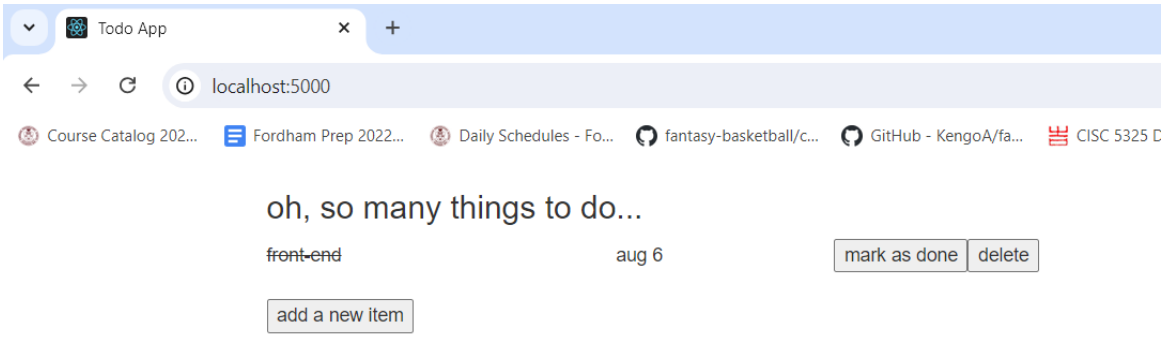
Here are local helper functions for generating the table rows and an example of calling the local endpoint for deletion:

```
const deleteRow = async (i) => {
    const task = entries[i].what_to_do;
    setEntries(entries.slice(0, i).concat(entries.slice(i+1)));
    const response = await fetch(`/delete/${encodeURIComponent(task)}`, {
      method: 'PUT'
    });

    if (!response.ok) {
      console.error(`Error occurred when deleting ${task.what_to_do}.`);
    } else {
      console.log(`${task.what_to_do} deleted.`);
    }
  }

  const getRowFromEntry = (entry, id) => {
    const addStrikethrough = entry.status === 'done';
    const className = addStrikethrough ? 'done' : 'not_done';
    return (
      <tr key={id}>
        <td className={className} style={{
            ...addStrikethrough ? { textDecoration: 'line-through' } : {}
          }}>{entry.what_to_do}</td>
        <td>{entry.due_date}</td>
        <td>
          <button onClick={() => markRowAsDone(id)}>mark as done</button>
          <button onClick={() => deleteRow(id)}>delete</button>
        </td>
      </tr>
    );
```

I confirmed locally that the client-side application works – and immediately saw the one entry I had persisted through the API service via curl!



GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

Dockerizing the Web Application

Like Homework 4, I created a Dockerfile, but I based the image on `node:14`. Upon starting the container we must run `npm install` and `npm run build` to build the React portion of the application. Finally, the container starts with `node index/server.js` and the express server will listen on port 5000. I used the same trick as the professor to pass the TODO API IP as a build argument (in my case, it is the amazonaws hostname).

I noticed that the docker build process took longer than the ubuntu image from Homework 4, and that is because npm install takes a while and more files must be copied. I added `a .dockerignore` file to make sure that `node_modules/` and `build/` were not copied into the image – those are automatically generated from the npm install and `npm run build` processes. Notice the timestamp on the RUN npm install:

```
=> [internal] load build definition from Dockerfile                                                    0.1s
=> => transferring dockerfile: 335B                                                                    0.1s
=> [internal] load metadata for docker.io/library/node:14                                              1.4s
=> [auth] library/node:pull token for registry-1.docker.io                                             0.0s
=> [internal] load .dockerignore                                                                       0.0s
=> => transferring context: 86B                                                                        0.0s
=> [internal] load build context                                                                       0.4s
=> => transferring context: 258.50kB                                                                   0.4s
=> [1/6] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa  0.0s
=> CACHED [2/6] WORKDIR /usr/src/app                                                                   0.0s
=> [3/6] COPY package.json ./                                                                          0.2s
=> [4/6] RUN npm install                                                                             109.3s
=> [5/6] COPY . .                                                                                      0.2s
=> [6/6] RUN npm run build                                                                            26.3s
=> exporting to image                                                                                 27.4s
=> => exporting layers                                                                                27.3s
=> => writing image sha256:dca57840315a48dd81fb87e8f464f9bb1dc35b77e9e7d082ef8b99b52e80cfa2            0.0s
=> => naming to docker.io/tnoone125/todo-web-application:latest                                        0.0s

What's next:
```

After pushing to DockerHub, I could test the container locally with:

```
docker run -d -p 5000:5000 tnoone125/todo-express-application:latest
```

Deploying to Kubernetes – Gcloud

I followed simple gcloud commands (like Homework 4) to deploy this web application to a Google Cloud cluster.

```
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> gcloud services enable container.googleapis.com
Operation "operations/acf.p2-215002388658-cecbcfc9-2df5-41d9-90db-d2514b767127" finished successfully.
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> gcloud container clusters create cisc5550-final-pr
oject --zone=us-east1-b
Default change: VPC-native is the default mode during cluster creation for versions greater than 1.21.0-gke.1500. To create advanced r
outes based clusters, please pass the `--no-enable-ip-alias` flag
Note: The Kubelet readonly port (10255) is now deprecated. Please update your workloads to use the recommended alternatives. See https
://cloud.google.com/kubernetes-engine/docs/how-to/disable-kubelet-readonly-port for ways to check usage and for migration instructions
.
Note: Your Pod address range (`--cluster-ipv4-cidr`) can accommodate at most 1008 node(s).
Creating cluster cisc5550-final-project in us-east1-b... Cluster is being configured.../
```

After the cluster was created, I ran:

```
kubectl create deployment todo-webapp –image=tnoone125/todo-express-application:latest --port=5000
kubectl expose deployment –type=LoadBalancer
gcloud compute firewall-rules create allow-http-5000 --allow tcp:5000
gcloud compute firewall-rules create allow-http-80 --allow tcp:80
kubectl get services
```

GitHub Repository: https://github.com/tnoone125/cloudcomputing-final-project

```
kubeconfig entry generated for cisc5550-final-project.
NAME                    LOCATION    MASTER_VERSION      MASTER_IP        MACHINE_TYPE  NODE_VERSION        NUM_NODES  STATUS
cisc5550-final-project  us-east1-b  1.29.6-gke.1254000  104.196.159.162  e2-medium     1.29.6-gke.1254000  3          RUNNING
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> kubectl create deployment todo-webapp --image=tnoone125/todo-express-a
pplication:latest --port=5000
deployment.apps/todo-webapp created
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> kubectl expose deployment todo-webapp --type=LoadBalancer
service/todo-webapp exposed
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> kubectl get service todo-webapp
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
todo-webapp   LoadBalancer   34.118.232.153   <pending>     5000:30330/TCP   18s
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> gcloud compute firewall-rules create allow-http-5000 --allow tcp:5000
Creating firewall...-Created [https://www.googleapis.com/compute/v1/projects/cisc5550-final-project/global/firewalls/allow-http-5000].
Creating firewall...done.
NAME             NETWORK  DIRECTION  PRIORITY  ALLOW     DENY  DISABLED
allow-http-5000  default  INGRESS    1000      tcp:5000        False
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> gcloud compute firewall-rules create allow-http-80 --allow tcp:80
Creating firewall...-Created [https://www.googleapis.com/compute/v1/projects/cisc5550-final-project/global/firewalls/allow-http-80].
Creating firewall...done.
NAME          NETWORK  DIRECTION  PRIORITY  ALLOW   DENY  DISABLED
allow-http-80  default  INGRESS    1000      tcp:80        False
PS C:\Users\thoma\OneDrive\Documents\CISC 5500\cisc5550-final-project-aws\todo-app> kubectl get service todo-webapp
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP    PORT(S)          AGE
todo-webapp   LoadBalancer   34.118.232.153   34.73.124.70   5000:30330/TCP   86s
```

And finally – I could reach the remote web application, which is properly communicating with the API service!