CS4442: Computation & Architecture

Cloud Computing Project - Spring Semester 2024

Student: Oluwajomiloju Olajitan: 23373326

Project Description

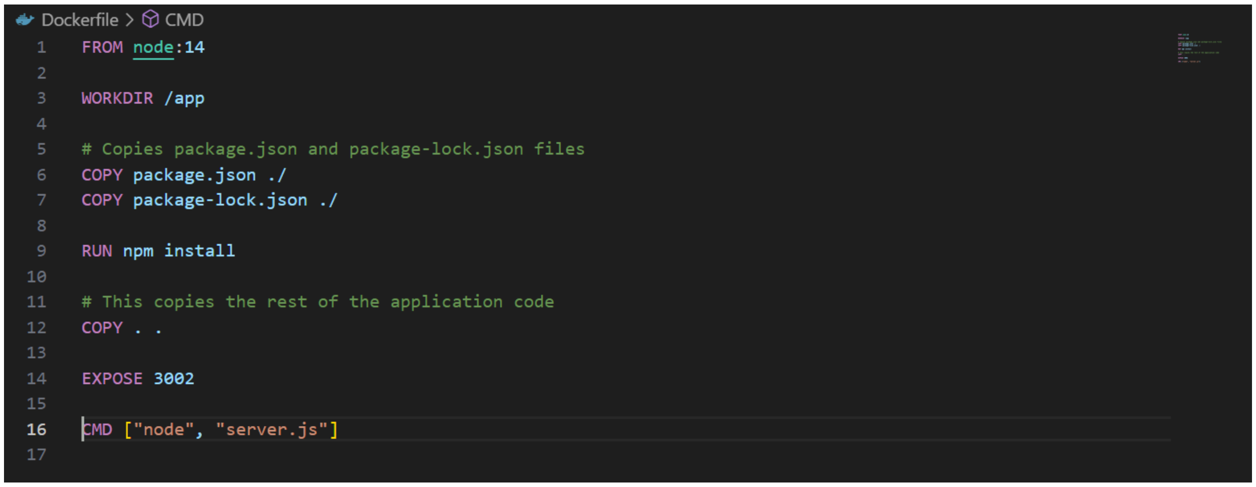
Upon conducting rigorous research into websites that were not only relevant, but beneficial to the task at hand, I decided to deploy an open-source website from the repository https://github.com/Sowwyz/sowwyz-drawing. The repository contained a drawing website without a database implemented or a dockerized image.

There are several reasons why I picked this website but the main cause was because during this project, I wanted to fully dockerize a website that I have an interest in, being art. I also chose this website as I have a history using various sites from this repository for my own personal uses. Lastly, I wanted to choose a website that had not implemented any aws features, a docker image, a running kubernetes cluster, or helm charts. With all these factors, I ended up making this decision.

There are many benefits to me choosing to containerize this website. The app is currently designed as a temporary canvas where users can draw anything, switch colours and just reset their image. WIth a system like this in place, I believe users should be able to interact with a database to store their images and be able to view all works of art they have made in the past. This website is also open to future expansion, as with the introduction of aws features, you may view the gallery of all users once granted permission. By using containers and managing them with Kubernetes, you can easily scale your application to meet changing demands. This improves both its availability and performance. Even if an individual container malfunctions, Kubernetes can automatically restart it, ensuring your application remains accessible to users.

1. Local Deployment Using Docker

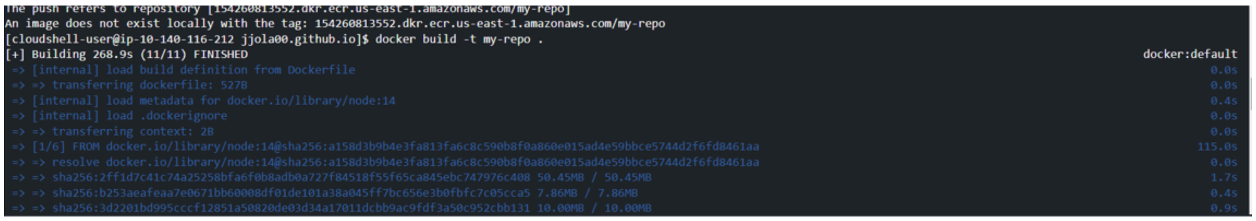
The first order of operation was to create a docker file.



```
Dockerfile > CMD
FROM node:14

WORKDIR /app

# Copies package.json and package-lock.json files
COPY package.json ./
COPY package-lock.json ./

RUN npm install

# This copies the rest of the application code
COPY . .

EXPOSE 3002

CMD ["node", "server.js"]

```

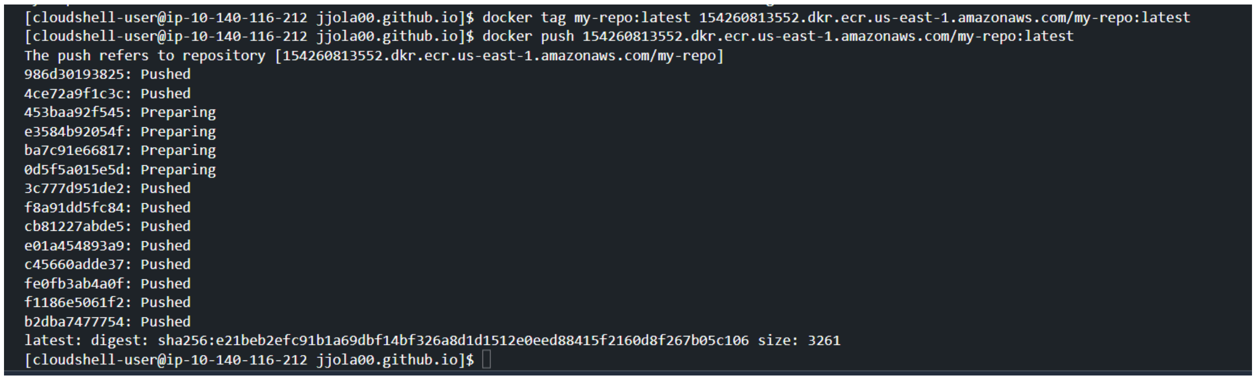
Fig 1.1: Dockerfile Configuration

After defining my docker file, I then built my image locally and pushed it to ECR on aws by following the operations defined.



```
The push refers to repository [154260813552.dkr.ecr.us-east-1.amazonaws.com/my-repo]
An image does not exist locally with the tag: 154260813552.dkr.ecr.us-east-1.amazonaws.com/my-repo
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ docker build -t my-repo .
[+] Building 268.9s (11/11) FINISHED                                                                        docker:default
 => [internal] load build definition from Dockerfile                                                                  0.0s
 => => transferring dockerfile: 527B                                                                                  0.0s
 => [internal] load metadata for docker.io/library/node:14                                                            0.4s
 => [internal] load .dockerignore                                                                                     0.0s
 => => transferring context: 2B                                                                                       0.0s
 => [1/6] FROM docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa    115.0s
 => => resolve docker.io/library/node:14@sha256:a158d3b9b4e3fa813fa6c8c590b8f0a860e015ad4e59bbce5744d2f6fd8461aa      0.0s
 => => sha256:2ff1d7c41c74a25258bfa6f0b8adb0a727f84518f55f65ca845ebc747976c408 50.45MB / 50.45MB                      1.7s
 => => sha256:b253aeafeaa7e0671bb60008df01de101a38a045ff7bc656e3b0fbfc7c05cca5 7.86MB / 7.86MB                        0.4s
 => => sha256:3d2201bd995cccf12851a50820de03d34a17011dcbb9ac9fdf3a50c952cbb131 10.00MB / 10.00MB                      0.9s
```

Fig 1.2: Docker Image building

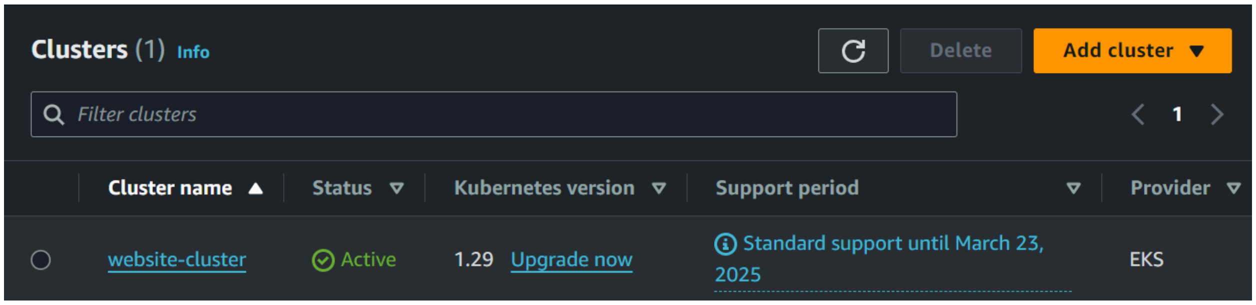


```
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ docker tag my-repo:latest 154260813552.dkr.ecr.us-east-1.amazonaws.com/my-repo:latest
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ docker push 154260813552.dkr.ecr.us-east-1.amazonaws.com/my-repo:latest
The push refers to repository [154260813552.dkr.ecr.us-east-1.amazonaws.com/my-repo]
986d30193825: Pushed
4ce72a9f1c3c: Pushed
453baa92f545: Preparing
e3584b92054f: Preparing
ba7c91e66817: Preparing
0d5f5a015e5d: Preparing
3c777d951de2: Pushed
f8a91dd5fc84: Pushed
cb81227abde5: Pushed
e01a454893a9: Pushed
c45660adde37: Pushed
fe0fb3ab4a0f: Pushed
f1186e5061f2: Pushed
b2dba7477754: Pushed
latest: digest: sha256:e21beb2efc91b1a69dbf14bf326a8d1d1512e0eed88415f2160d8f267b05c106 size: 3261
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$
```

Fig 1.3: Image pushed

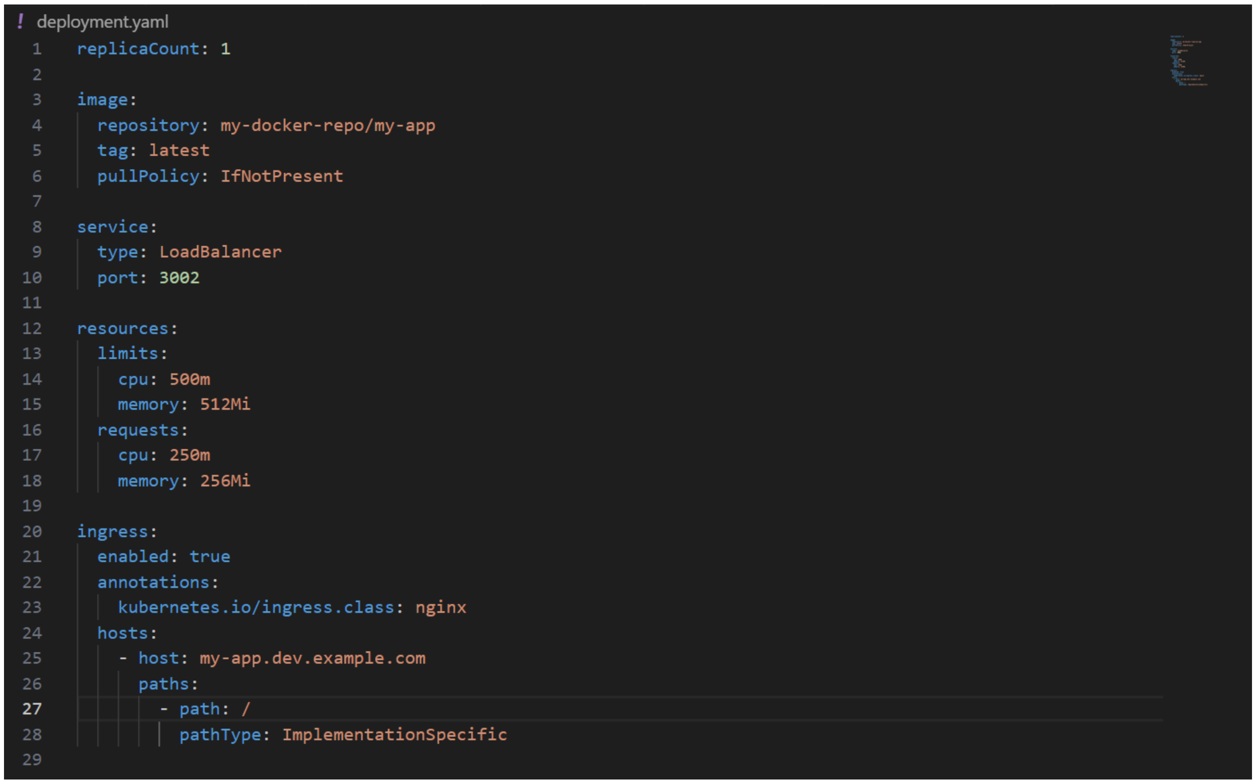
2. Set up Kubernetes Cluster

I first decided to set up my kubernetes cluster first before helm charts because it was an area I have some experience in and thought it would be more appropriate for this reason. I created my cluster on the aws website using EKS.



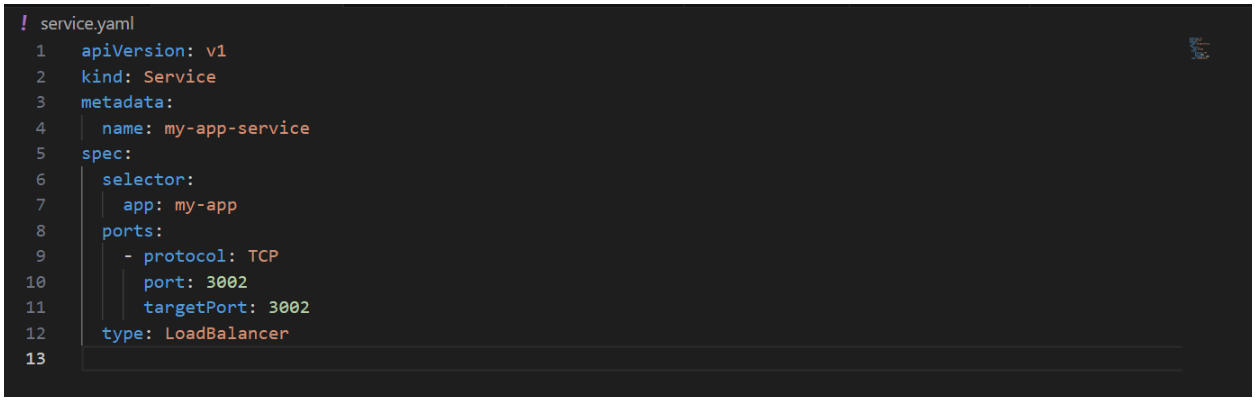
	Cluster name	Status	Kubernetes version	Support period	Provider
○	website-cluster	Active	1.29 Upgrade now	Standard support until March 23, 2025	EKS

Fig 2.1: EKS Cluster Setup



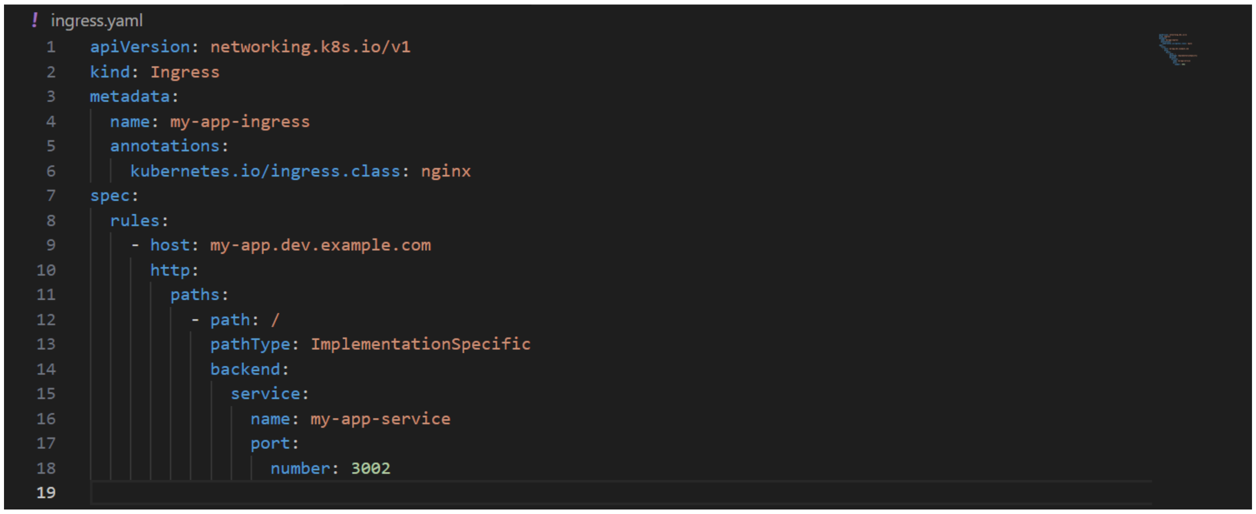
```yaml
deployment.yaml
replicaCount: 1

image:
  repository: my-docker-repo/my-app
  tag: latest
  pullPolicy: IfNotPresent

service:
  type: LoadBalancer
  port: 3002

resources:
  limits:
    cpu: 500m
    memory: 512Mi
  requests:
    cpu: 250m
    memory: 256Mi

ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
  hosts:
    - host: my-app.dev.example.com
      paths:
        - path: /
          pathType: ImplementationSpecific

```

Fig 2.2 : deployment.yaml



```yaml
service.yaml
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 3002
      targetPort: 3002
  type: LoadBalancer

```

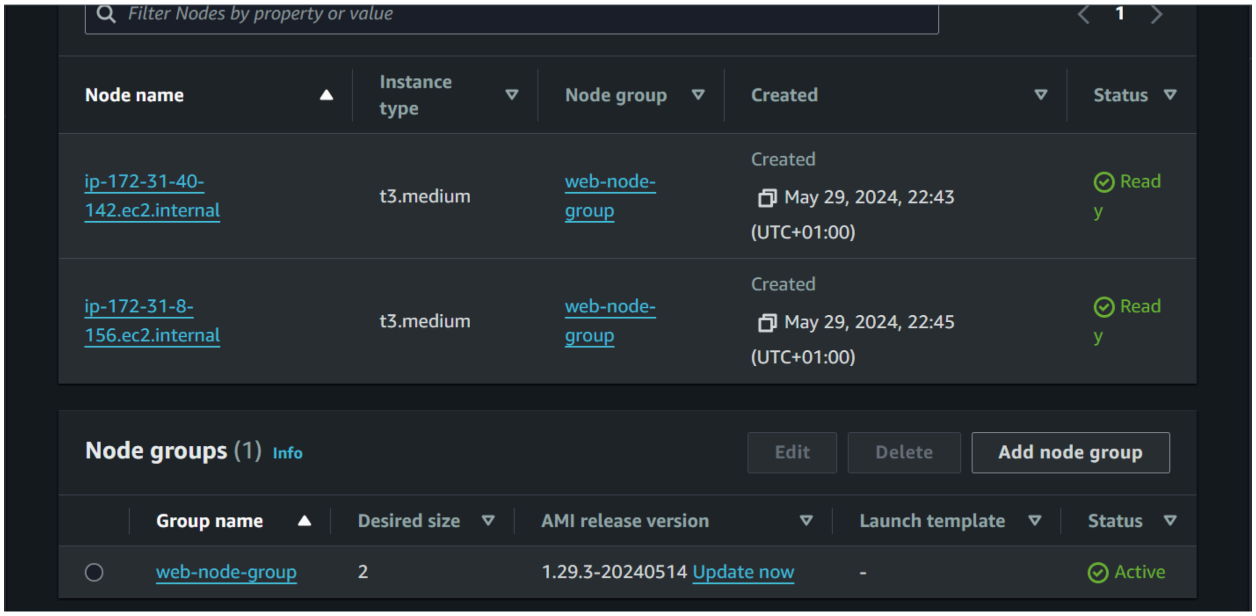
Fig 2.3: service.yaml



```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-app-ingress
  annotations:
    kubernetes.io/ingress.class: nginx
spec:
  rules:
    - host: my-app.dev.example.com
      http:
        paths:
          - path: /
            pathType: ImplementationSpecific
            backend:
              service:
                name: my-app-service
                port:
                  number: 3002

```

Fig 2.4: ingress.yaml

After deployment, I verified the application's accessibility using the kubectl get services command. Initially, the application wasn't reachable. To fix the issue, I used kubectl get pods and kubectl get nodes to inspect the pods and nodes in the cluster. I discovered that there weren't any running nodes available. To address this, I created a node group.

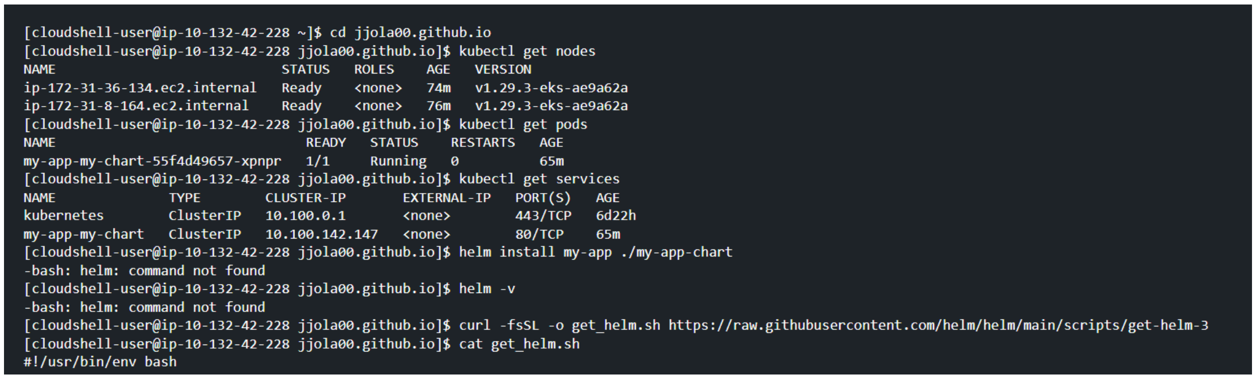


Node name	Instance type	Node group	Created	Status
ip-172-31-40-142.ec2.internal	t3.medium	web-node-group	Created May 29, 2024, 22:43 (UTC+01:00)	Ready
ip-172-31-8-156.ec2.internal	t3.medium	web-node-group	Created May 29, 2024, 22:45 (UTC+01:00)	Ready

Node groups (1) Info

Group name	Desired size	AMI release version	Launch template	Status
web-node-group	2	1.29.3-20240514 Update now	-	Active

Fig 2.5: Nodes running

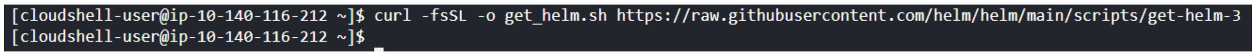


```
[cloudshell-user@ip-10-132-42-228 ~]$ cd jjola00.github.io
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ kubectl get nodes
NAME                              STATUS   ROLES    AGE   VERSION
ip-172-31-36-134.ec2.internal   Ready    <none>   74m   v1.29.3-eks-ae9a62a
ip-172-31-8-164.ec2.internal    Ready    <none>   76m   v1.29.3-eks-ae9a62a
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ kubectl get pods
NAME                              READY   STATUS    RESTARTS   AGE
my-app-my-chart-55f4d49657-xpnpr   1/1     Running   0          65m
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ kubectl get services
NAME              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)   AGE
kubernetes        ClusterIP   10.100.0.1       <none>        443/TCP   6d22h
my-app-my-chart   ClusterIP   10.100.142.147   <none>        80/TCP    65m
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ helm install my-app ./my-app-chart
-bash: helm: command not found
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ helm -v
-bash: helm: command not found
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
[cloudshell-user@ip-10-132-42-228 jjola00.github.io]$ cat get_helm.sh
#!/usr/bin/env bash
```

Fig 2.6: List of kubectl services

3. Setting up Helm

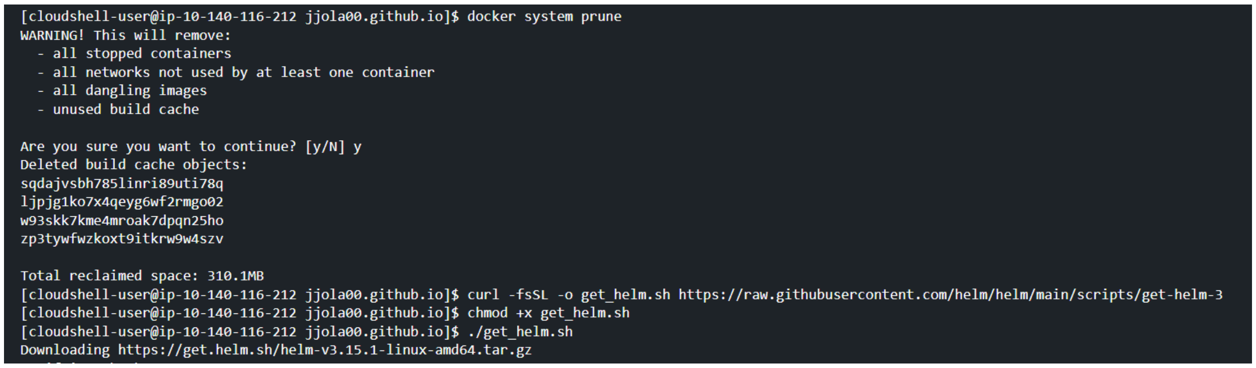
The first step I took was to install Helm.



```
[cloudshell-user@ip-10-140-116-212 ~]$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
[cloudshell-user@ip-10-140-116-212 ~]$
```

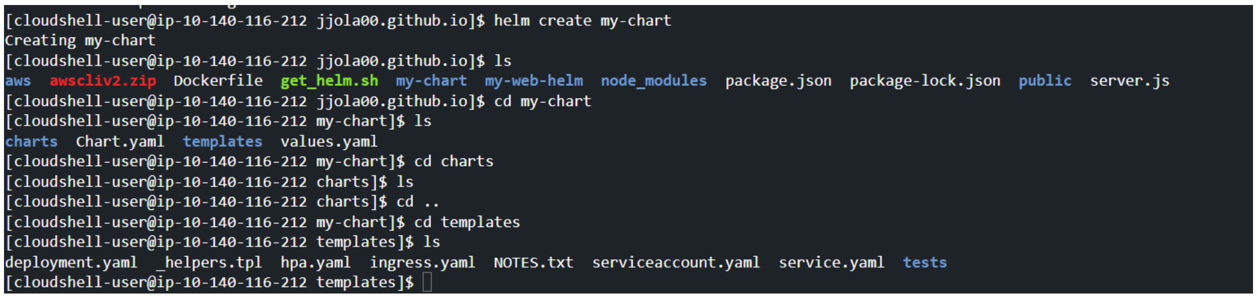
Fig 3.1: First attempt of Helm Install

At first I ran into an issue where I could not install Helm due to storage,so I freed up space in aws first. After this, I reran the code to install helm and tested the version.



```
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ docker system prune
WARNING! This will remove:
  - all stopped containers
  - all networks not used by at least one container
  - all dangling images
  - unused build cache

Are you sure you want to continue? [y/N] y
Deleted build cache objects:
sqdajvsbh785linri89uti78q
ljpjg1ko7x4qeyg6wf2rmgo02
w93skk7kme4mroak7dpqn25ho
zp3tywfwzkoxt9itkrw9w4szv

Total reclaimed space: 310.1MB
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ chmod +x get_helm.sh
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ ./get_helm.sh
Downloading https://get.helm.sh/helm-v3.15.1-linux-amd64.tar.gz
```

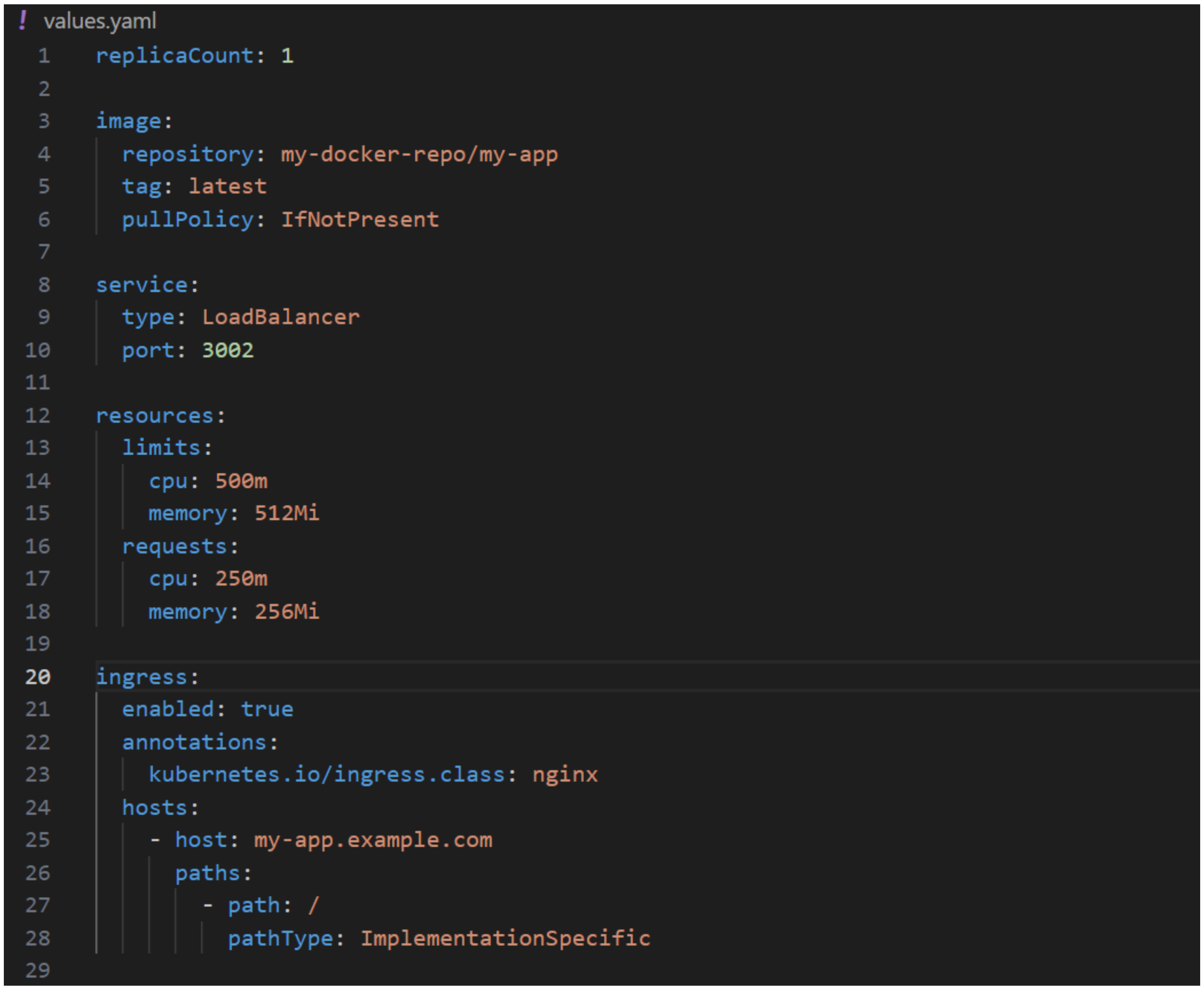
Fig 3.2: Re running installation once storage freed



```
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ helm create my-chart
Creating my-chart
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ ls
aws  awscliv2.zip  Dockerfile  get_helm.sh  my-chart  my-web-helm  node_modules  package.json  package-lock.json  public  server.js
[cloudshell-user@ip-10-140-116-212 jjola00.github.io]$ cd my-chart
[cloudshell-user@ip-10-140-116-212 my-chart]$ ls
charts  Chart.yaml  templates  values.yaml
[cloudshell-user@ip-10-140-116-212 my-chart]$ cd charts
[cloudshell-user@ip-10-140-116-212 charts]$ ls
[cloudshell-user@ip-10-140-116-212 charts]$ cd ..
[cloudshell-user@ip-10-140-116-212 my-chart]$ cd templates
[cloudshell-user@ip-10-140-116-212 templates]$ ls
deployment.yaml  _helpers.tpl  hpa.yaml  ingress.yaml  NOTES.txt  serviceaccount.yaml  service.yaml  tests
[cloudshell-user@ip-10-140-116-212 templates]$
```

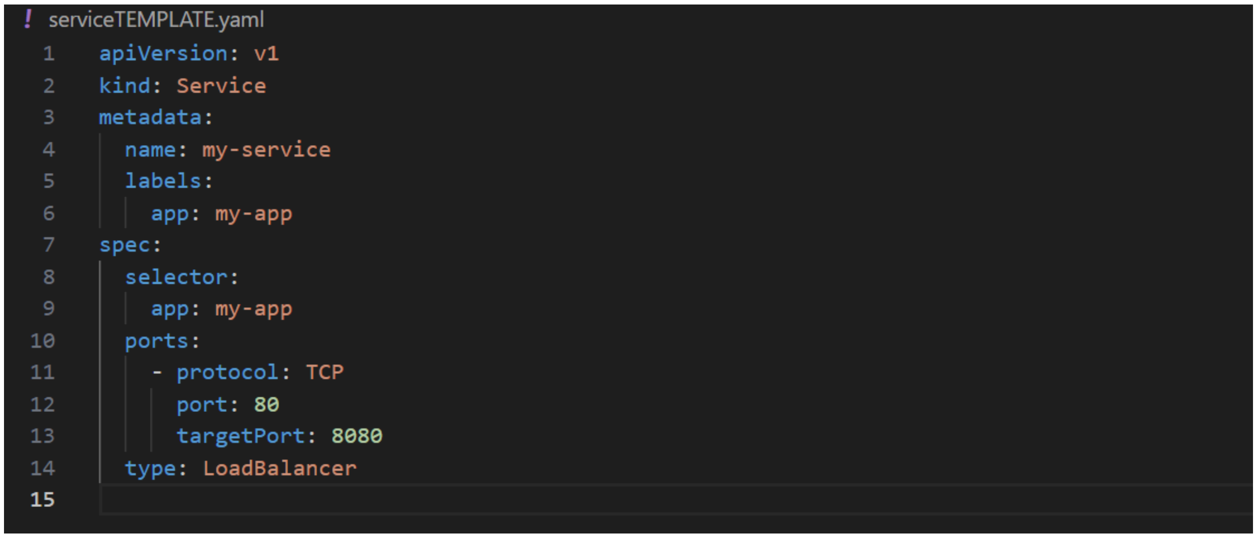
Fig 3.3: Project setup after installing helm

After setting up and testing helm, i decided to update some of my yaml files:


values.yaml

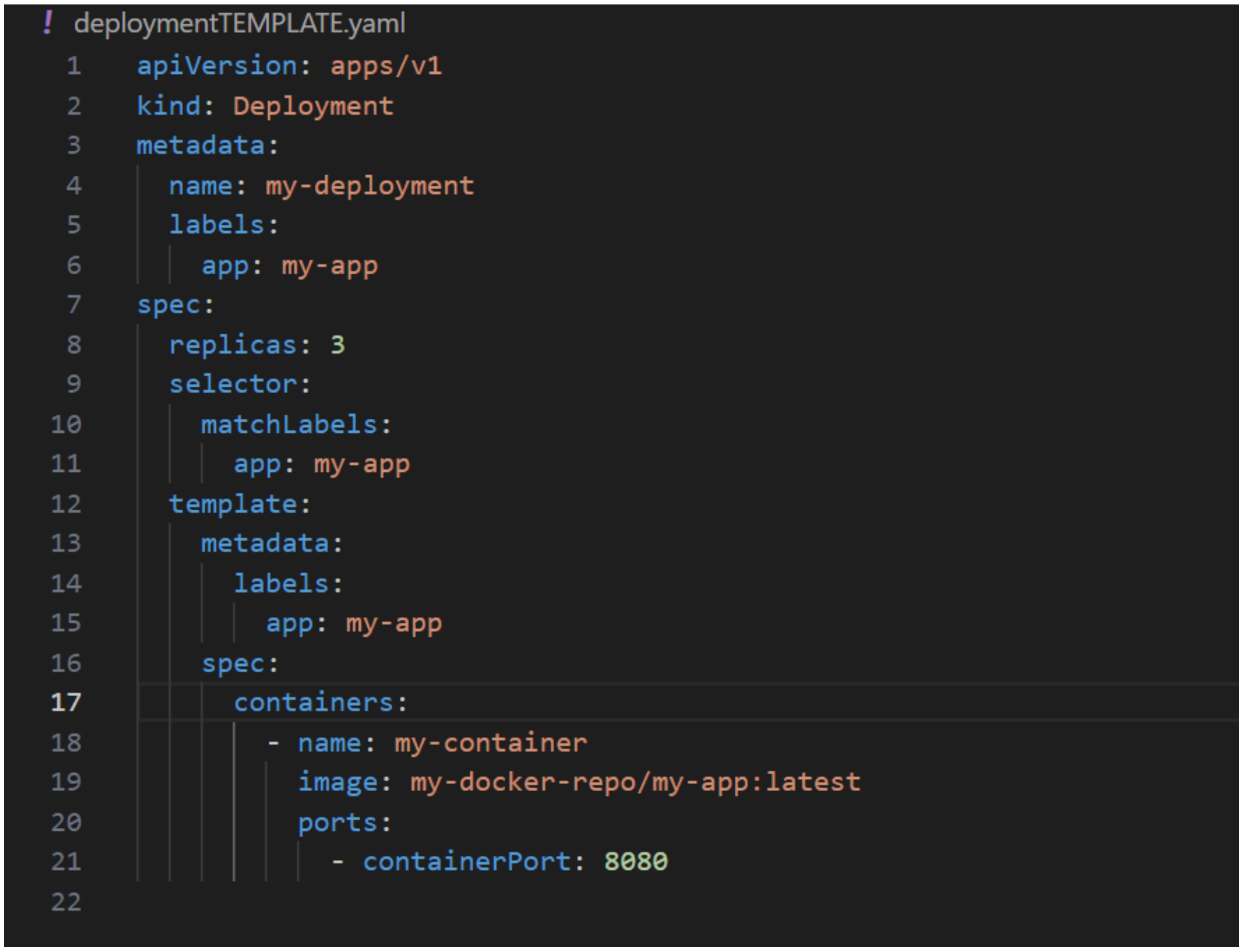
```yaml
replicaCount: 1

image:
  repository: my-docker-repo/my-app
  tag: latest
  pullPolicy: IfNotPresent

service:
  type: LoadBalancer
  port: 3002

resources:
  limits:
    cpu: 500m
    memory: 512Mi
  requests:
    cpu: 250m
    memory: 256Mi

ingress:
  enabled: true
  annotations:
    kubernetes.io/ingress.class: nginx
  hosts:
    - host: my-app.example.com
      paths:
        - path: /
          pathType: ImplementationSpecific

```

Fig 3.4: values.yaml


serviceTEMPLATE.yaml

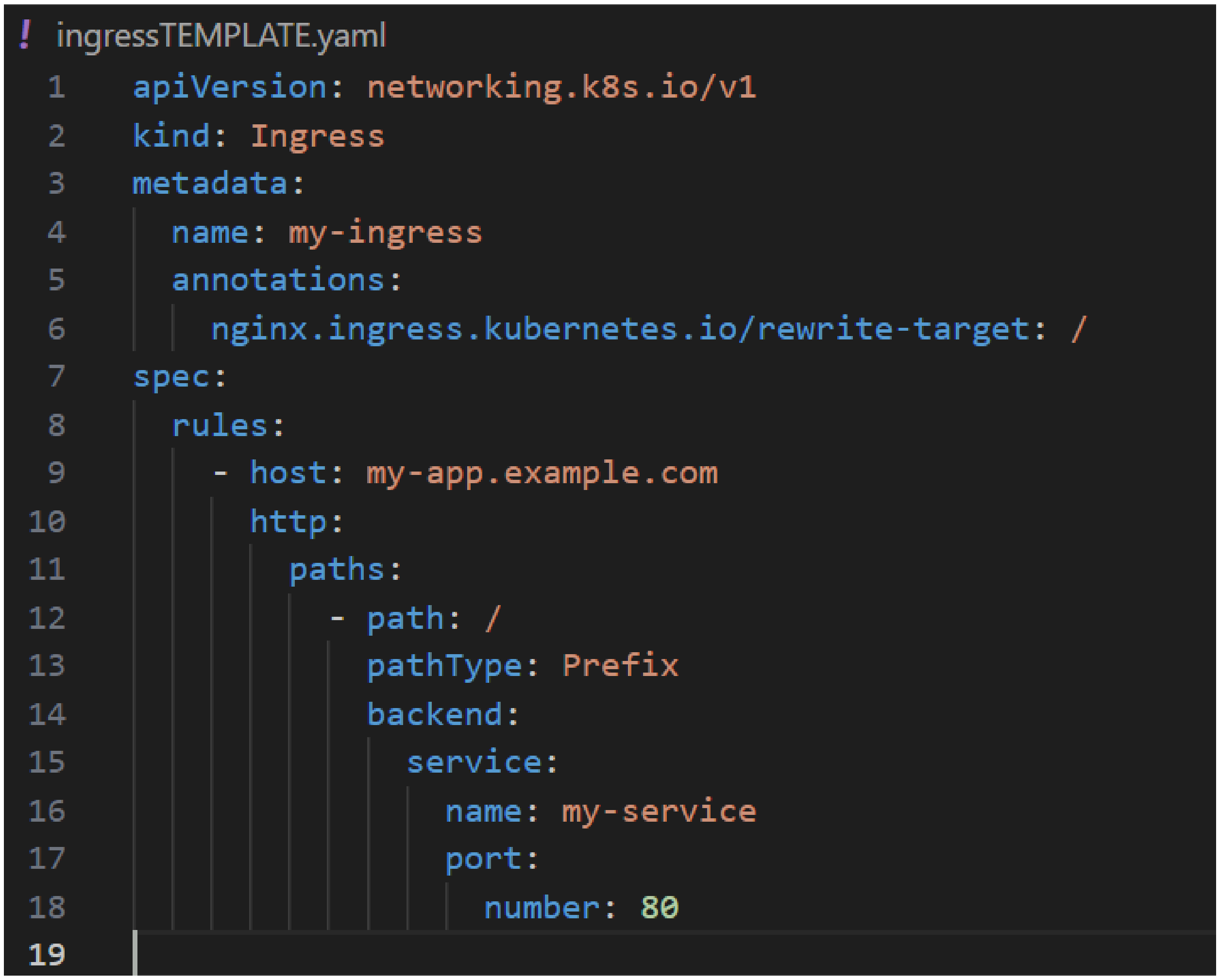
```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
  labels:
    app: my-app
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: LoadBalancer

```

Fig 3.5: serviceTEMPLATES.yaml



```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: my-docker-repo/my-app:latest
          ports:
            - containerPort: 8080
```

Fig 3.6: deploymentTEMPLATES.yaml

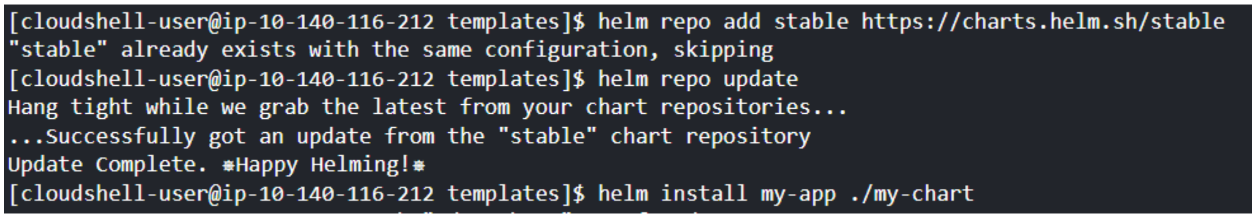


ingressTEMPLATE.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
    - host: my-app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: my-service
                port:
                  number: 80

```

Fig 3.7: ingresTEMPLATES.yaml

Finally, I deployed my web chart using helm

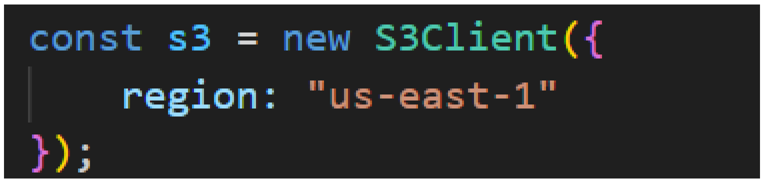


```
[cloudshell-user@ip-10-140-116-212 templates]$ helm repo add stable https://charts.helm.sh/stable
"stable" already exists with the same configuration, skipping
[cloudshell-user@ip-10-140-116-212 templates]$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "stable" chart repository
Update Complete. ⎈Happy Helming!⎈
[cloudshell-user@ip-10-140-116-212 templates]$ helm install my-app ./my-chart
```

Fig 3.8: Helm web chart deployment

4. AWS SERVICE 1: S3 Buckets

The first service I successfully implemented was an S3 bucket to store images.



```
const s3 = new S3Client({
    region: "us-east-1"
});
```

Fig 4.1: Initialising bucket

Once the user draws anything on the website, they have a prompt to save it. This is then saved in buckets on the AWS.

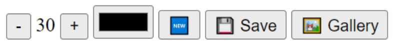
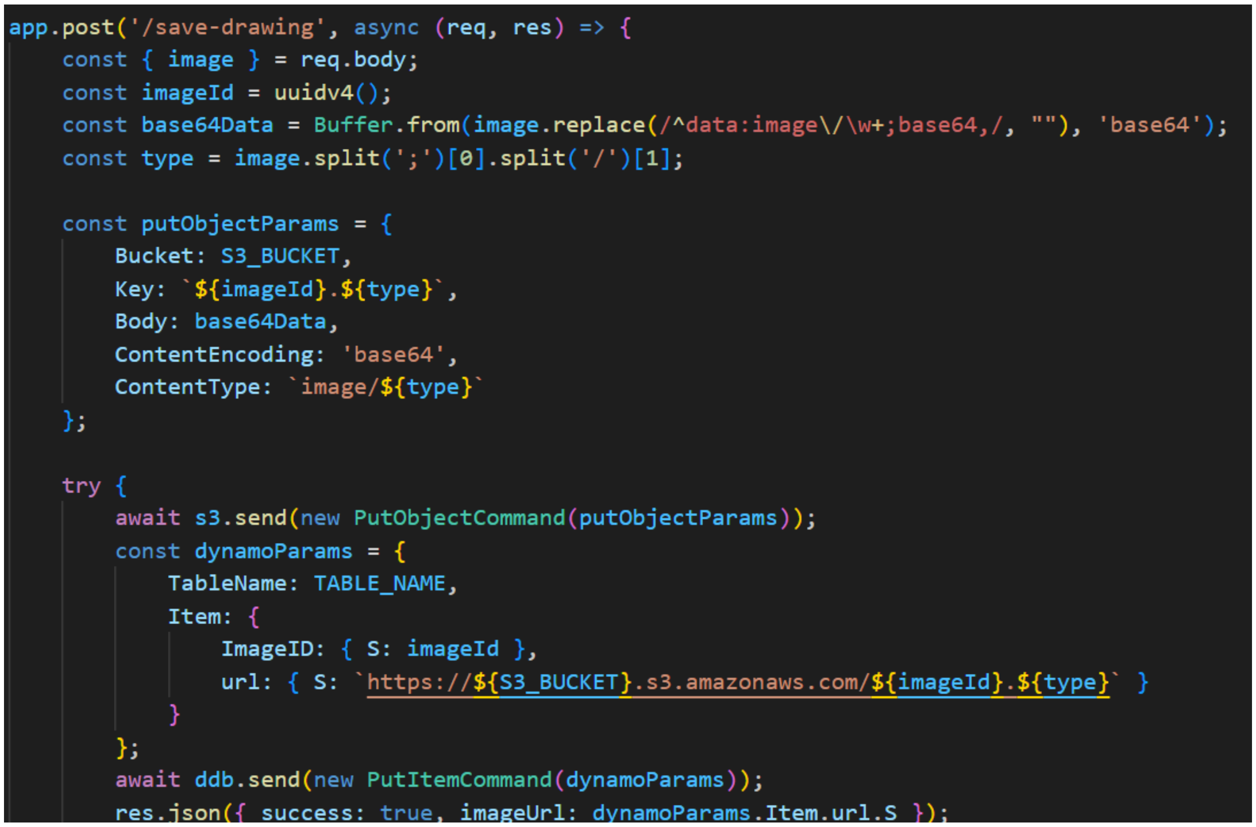
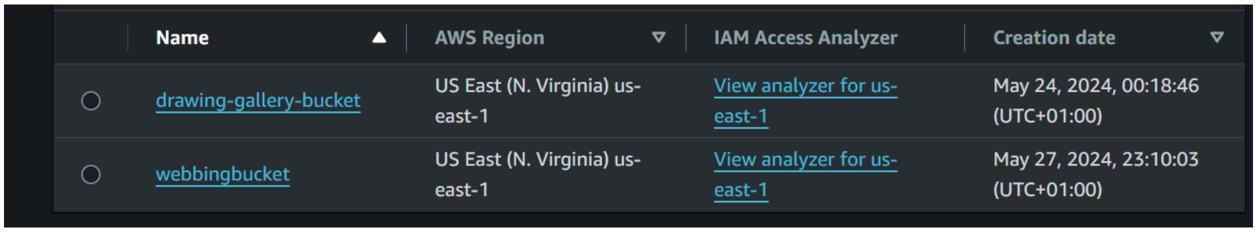


Fig 4.2: Website prompt to Save image



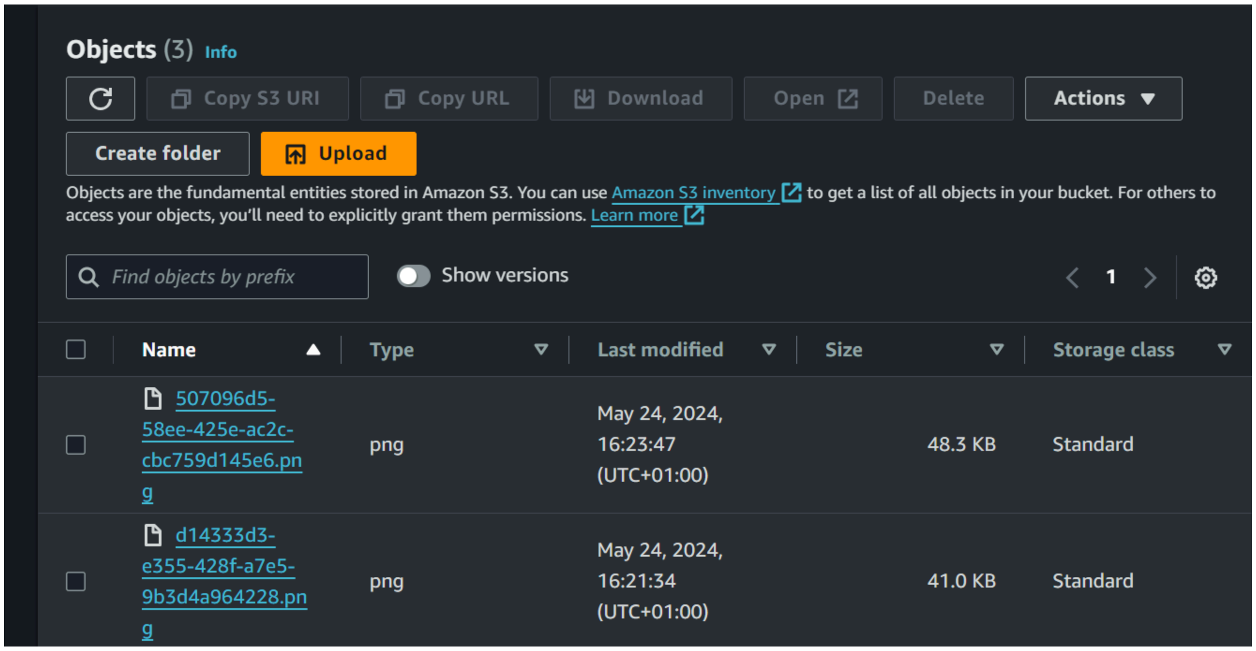
```
app.post('/save-drawing', async (req, res) => {
    const { image } = req.body;
    const imageId = uuidv4();
    const base64Data = Buffer.from(image.replace(/^data:image\/\w+;base64,/, ""), 'base64');
    const type = image.split(';')[0].split('/')[1];

    const putObjectParams = {
        Bucket: S3_BUCKET,
        Key: `${imageId}.${type}`,
        Body: base64Data,
        ContentEncoding: 'base64',
        ContentType: `image/${type}`
    };

    try {
        await s3.send(new PutObjectCommand(putObjectParams));
        const dynamoParams = {
            TableName: TABLE_NAME,
            Item: {
                ImageID: { S: imageId },
                url: { S: `https://${S3_BUCKET}.s3.amazonaws.com/${imageId}.${type}` }
            }
        };
        await ddb.send(new PutItemCommand(dynamoParams));
        res.json({ success: true, imageUrl: dynamoParams.Item.url.S });
```

Fig 4.3: Back end logic to connect buckets



	Name	AWS Region	IAM Access Analyzer	Creation date
○	drawing-gallery-bucket	US East (N. Virginia) us-east-1	View analyzer for us-east-1	May 24, 2024, 00:18:46 (UTC+01:00)
○	webbingbucket	US East (N. Virginia) us-east-1	View analyzer for us-east-1	May 27, 2024, 23:10:03 (UTC+01:00)

Fig 4.4: Sample Buckets. Drawing-gallery-buckets stores the images



Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

	Name	Type	Last modified	Size	Storage class
☐	507096d5-58ee-425e-ac2c-cbc759d145e6.png	png	May 24, 2024, 16:23:47 (UTC+01:00)	48.3 KB	Standard
☐	d14333d3-e355-428f-a7e5-9b3d4a964228.png	png	May 24, 2024, 16:21:34 (UTC+01:00)	41.0 KB	Standard

Fig 4.5: Images being stored in the cloud

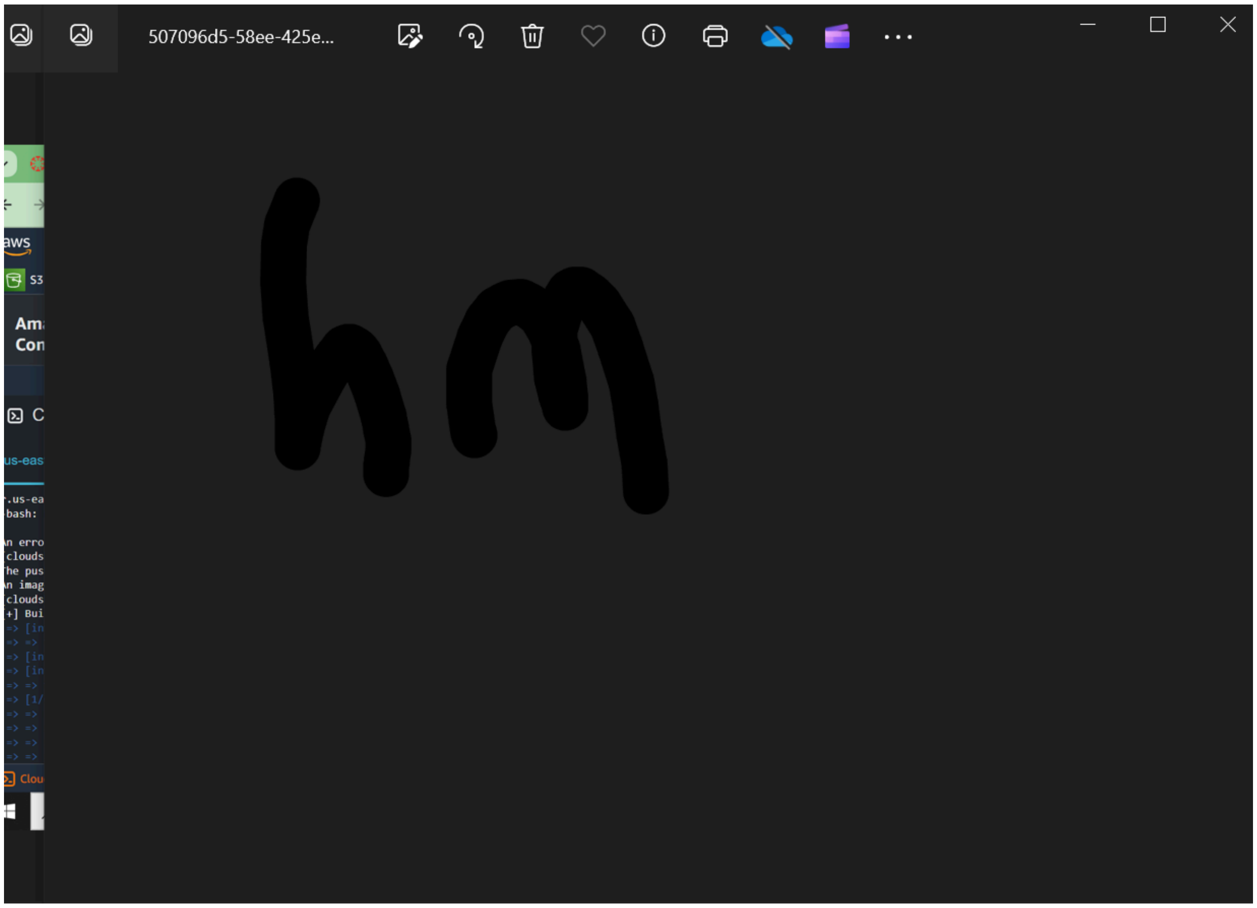
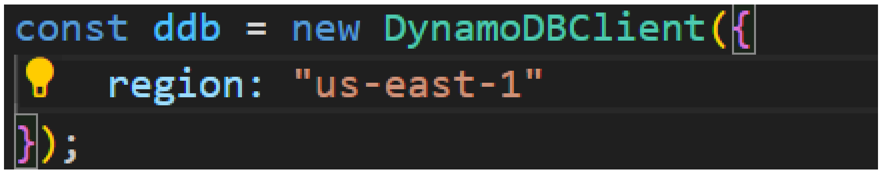


Fig 4.6: Proof of image

5. AWS SERVICE 2: DynamoDB

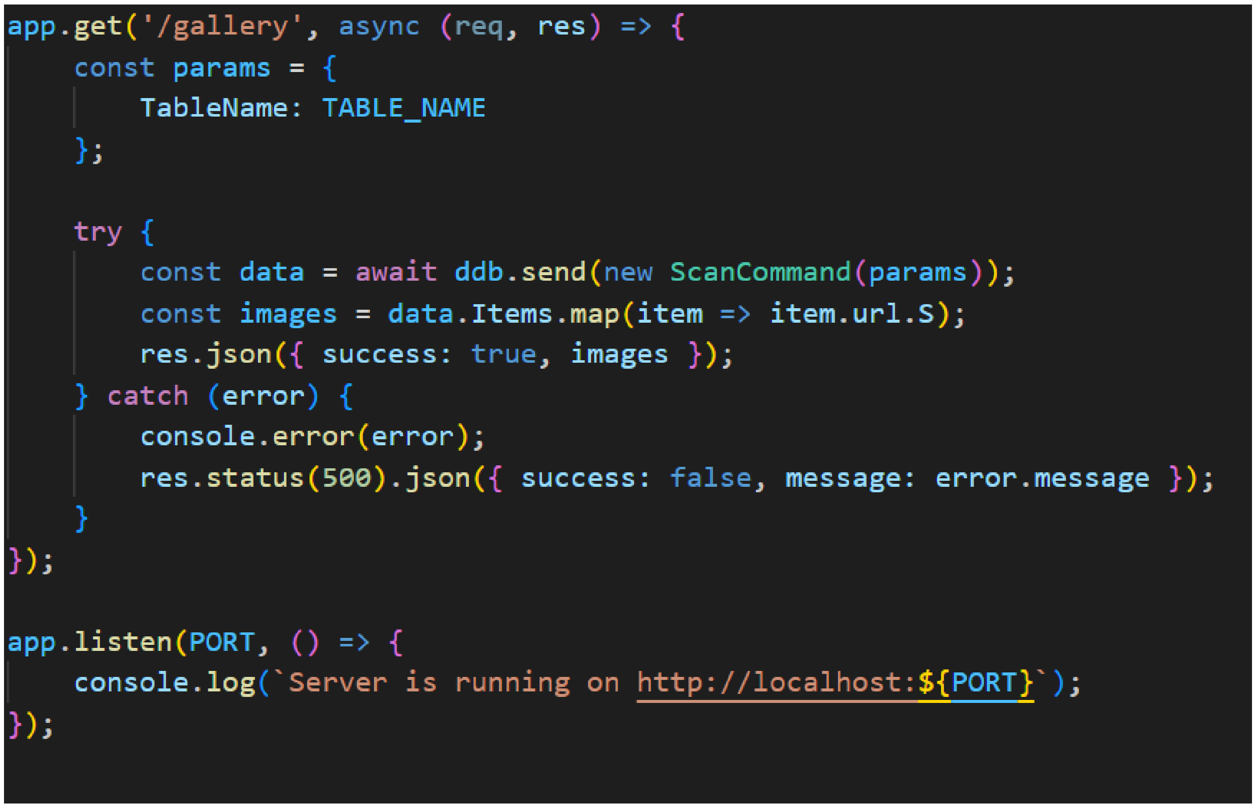
After storing the buckets in the AWS, I also wanted users to be able to see their images. To accomplish this, I implemented a database using dynamoDB.



```
const ddb = new DynamoDBClient({
    region: "us-east-1"
});
```

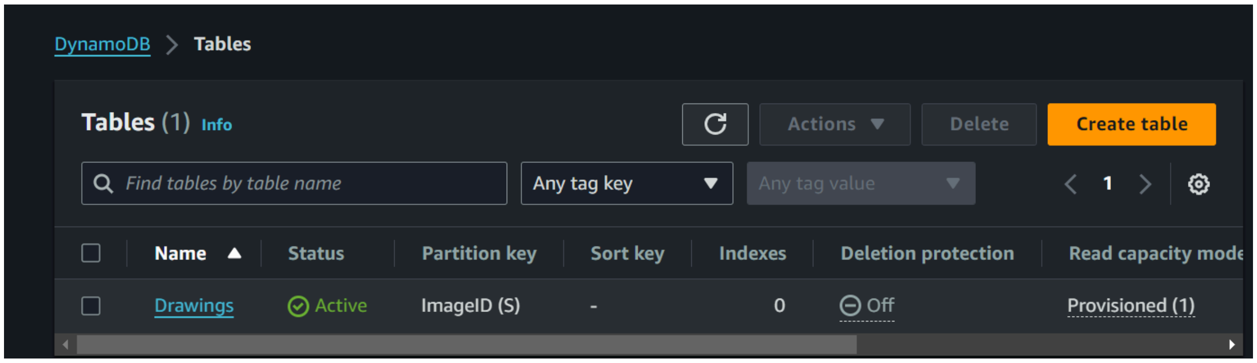
Fig 5.1: Initialising DynamoDB

Once the user selects the Gallery button, A list of the picture of the images, its creator and time created will be returned.



```
app.get('/gallery', async (req, res) => {
    const params = {
        TableName: TABLE_NAME
    };

    try {
        const data = await ddb.send(new ScanCommand(params));
        const images = data.Items.map(item => item.url.S);
        res.json({ success: true, images });
    } catch (error) {
        console.error(error);
        res.status(500).json({ success: false, message: error.message });
    }
});

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```

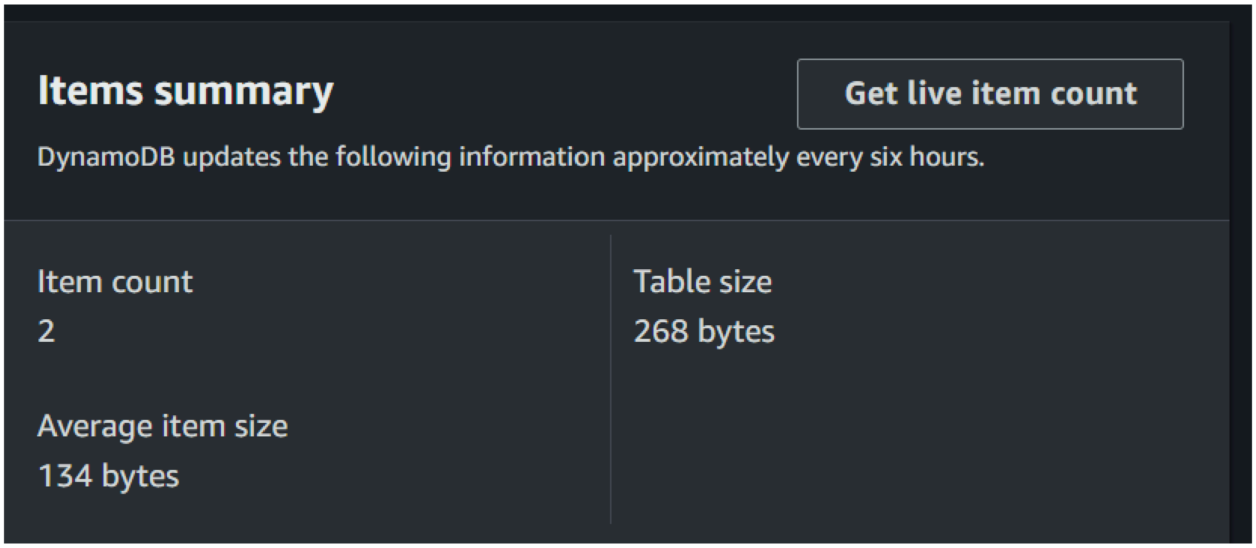
Fig 5.2: Backend logic


DynamoDB > Tables

Tables (1) Info

	Name	Status	Partition key	Sort key	Indexes	Deletion protection	Read capacity mode
☐	Drawings	Active	ImageID (S)	-	0	Off	Provisioned (1)

Fig 5.3: Proof of drawing table


Items summary

DynamoDB updates the following information approximately every six hours.

Item count
2

Table size
268 bytes

Average item size
134 bytes

Fig 5.4: Drawing count

6. Reflection

I personally found this project to be one of the most challenging of the Immersive Software Engineering syllabus so far, even after fully completing the AWS online learners course. I felt lost during various stages of the project and felt a severe lack of online resources to turn to when a roadblock is hit. However, upon completing this project I can say with utmost certainty that my software skills in relation to AWS and cloud computing has vastly improved.

One of the most important lessons I learnt during this project was the importance of going into a massive amount of work such as this with a pre-planned schedule to never veer off track and maintain tidiness and cohesiveness. The distribution of work along the 3 week period was not a linear line. The line of work and rest fluctuated due to lack of work on some days and too much work the next day. I found implementing a schedule during the latter days of the project put me at a great advantage as it allowed me to cohesively wrap up my project and tidy up any errors.

In conclusion, this project was a challenging but rewarding learning experience. It exposed knowledge gaps and emphasised the importance of continuous learning and adaptation. By analysing the challenges and my achievements, I've gained valuable insights that will propel my future projects and professional development.