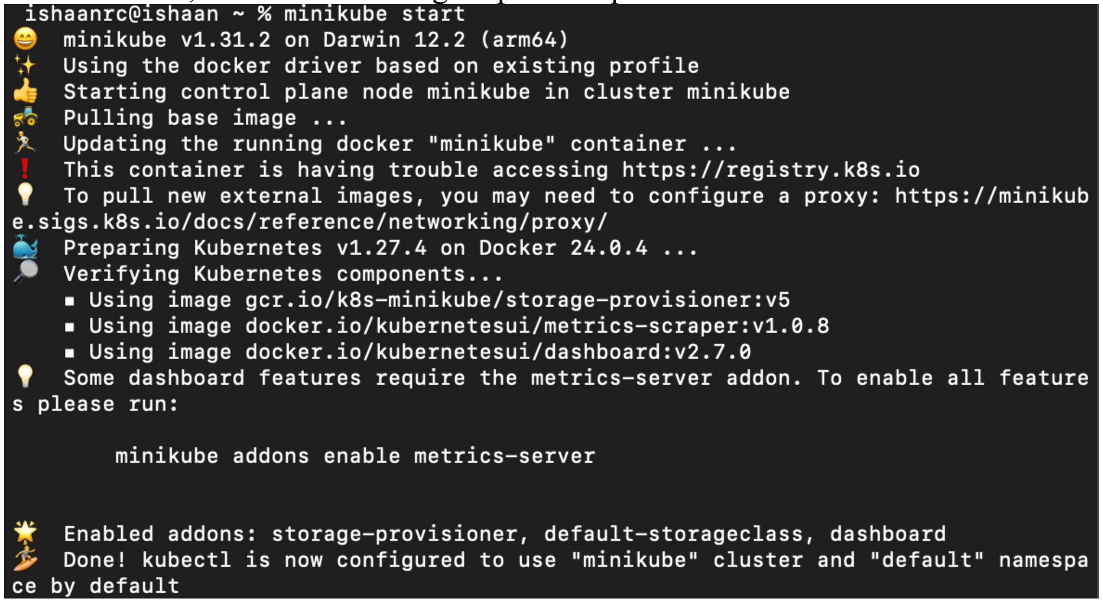
CS 551 : Assignment 5

Part 0: Installing Kubernetes

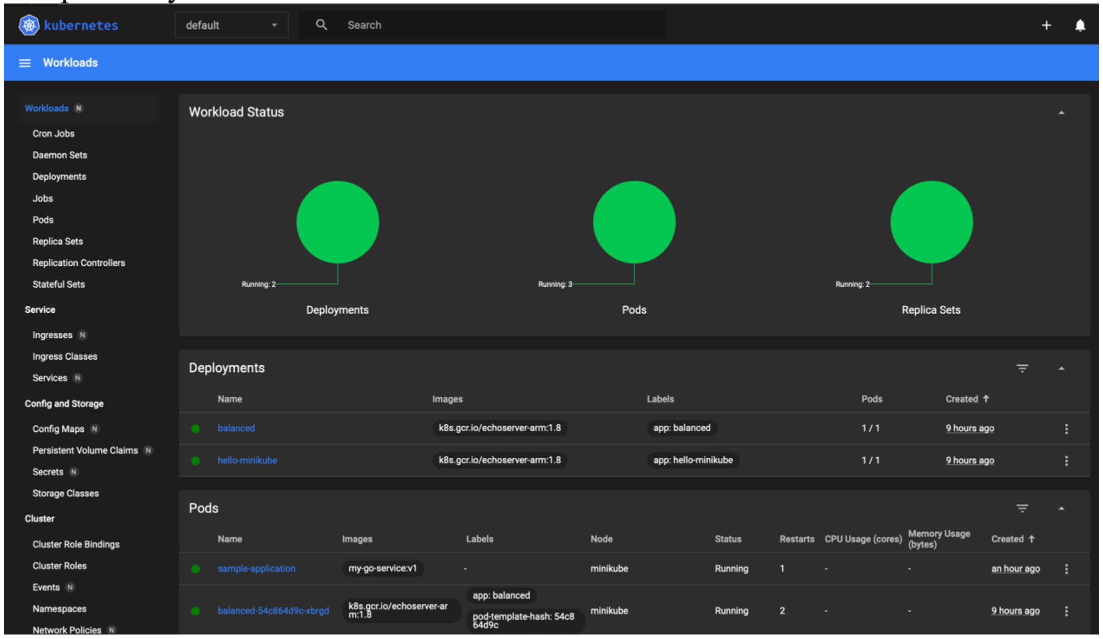
The installation was pretty straightforward. I followed the steps mentioned in https://kubernetes.io/releases/download/

Once installed, I had the following output as expected:



```
 ishaanrc@ishaan ~ % minikube start
😄  minikube v1.31.2 on Darwin 12.2 (arm64)
✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🚜  Pulling base image ...
🏃  Updating the running docker "minikube" container ...
❗  This container is having trouble accessing https://registry.k8s.io
💡  To pull new external images, you may need to configure a proxy: https://minikub
e.sigs.k8s.io/docs/reference/networking/proxy/
🐳  Preparing Kubernetes v1.27.4 on Docker 24.0.4 ...
🔎  Verifying Kubernetes components...
    ▪ Using image gcr.io/k8s-minikube/storage-provisioner:v5
    ▪ Using image docker.io/kubernetesui/metrics-scraper:v1.0.8
    ▪ Using image docker.io/kubernetesui/dashboard:v2.7.0
💡  Some dashboard features require the metrics-server addon. To enable all feature
s please run:

        minikube addons enable metrics-server


🌟  Enabled addons: storage-provisioner, default-storageclass, dashboard
🏄  Done! kubectl is now configured to use "minikube" cluster and "default" namespa
ce by default
```

Sample of my minikube dashboard to make sure installation was successful:



Deployments

	Name	Images	Labels	Pods	Created ↑
●	balanced	k8s.gcr.io/echoserver-arm:1.8	app: balanced	1 / 1	9 hours ago
●	hello-minikube	k8s.gcr.io/echoserver-arm:1.8	app: hello-minikube	1 / 1	9 hours ago

Pods

	Name	Images	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
●	sample-application	my-go-service:v1	-	minikube	Running	1	-	-	an hour ago
●	balanced-54c864d9c-xbrgd	k8s.gcr.io/echoserver-arm:1.8	app: balanced pod-template-hash: 54c864d9c	minikube	Running	2	-	-	9 hours ago

This GUI helps us visualize and understand our deployments, pods, replica sets, etc.

Part 1: Deploy your first K8s service

This section details the process of deploying the first Kubernetes service using the hello-minikube example and an echo server that provides insights into the HTTP requests it receives.

- Image: k8s.gcr.io/echoserver:1.4
- Purpose: The echoserver image serves as a basic HTTP server that returns information about the HTTP request it received. Useful for understanding and debugging Kubernetes deployments.

I followed the steps in the assignment and the only problem I had was since I had a M1 chip I was getting the following error:

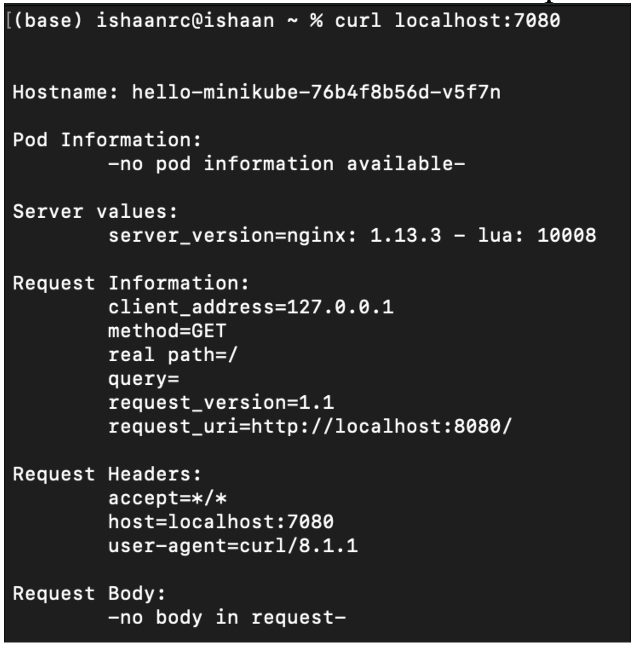
2023/10/28 16:23:26 [error] 1#1: failed to initialize Lua VM in /etc/nginx/nginx.conf:54
nginx: [error] failed to initialize Lua VM in /etc/nginx/nginx.conf:54

The way I solved it was through following the Piazza note for a M1 chip compatible image:

```
kubectl create deployment hello-minikube --image=k8s.gcr.io/echoserver-arm:1.8
```

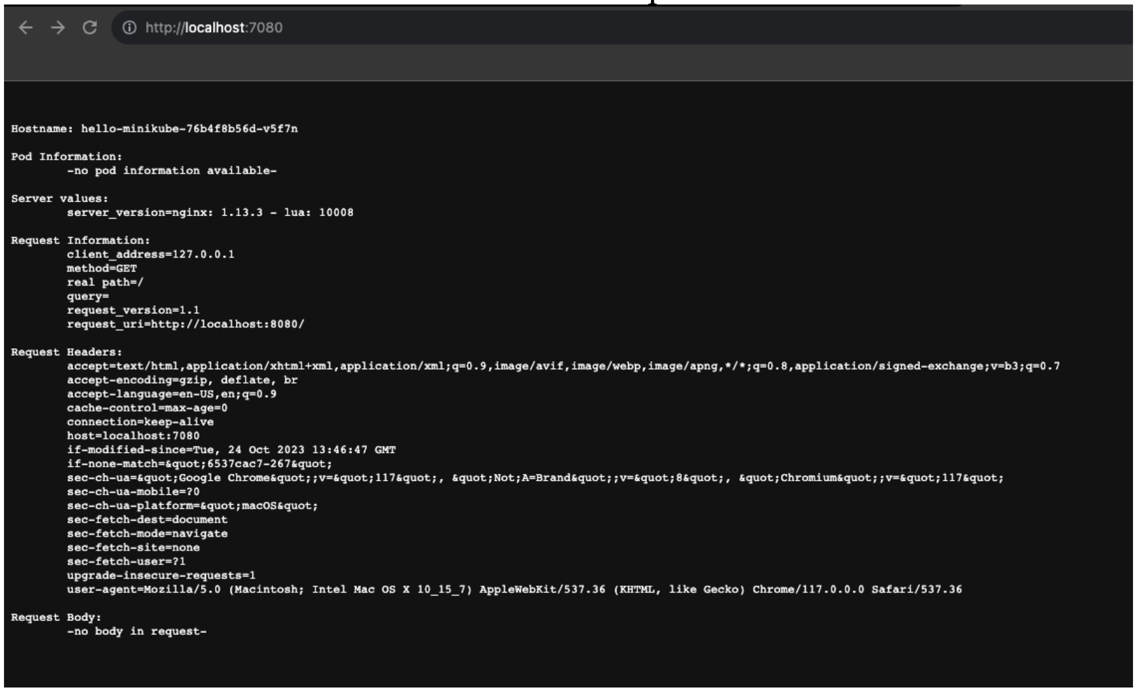
This seemed to have worked seamlessly and I had no problems after this. I used kubectl to deploy my K8 service. Below are the screenshots attached of what I saw and how the client values change based on my requests.

Screenshot of when I curl localhost:7080 after port forwarding:



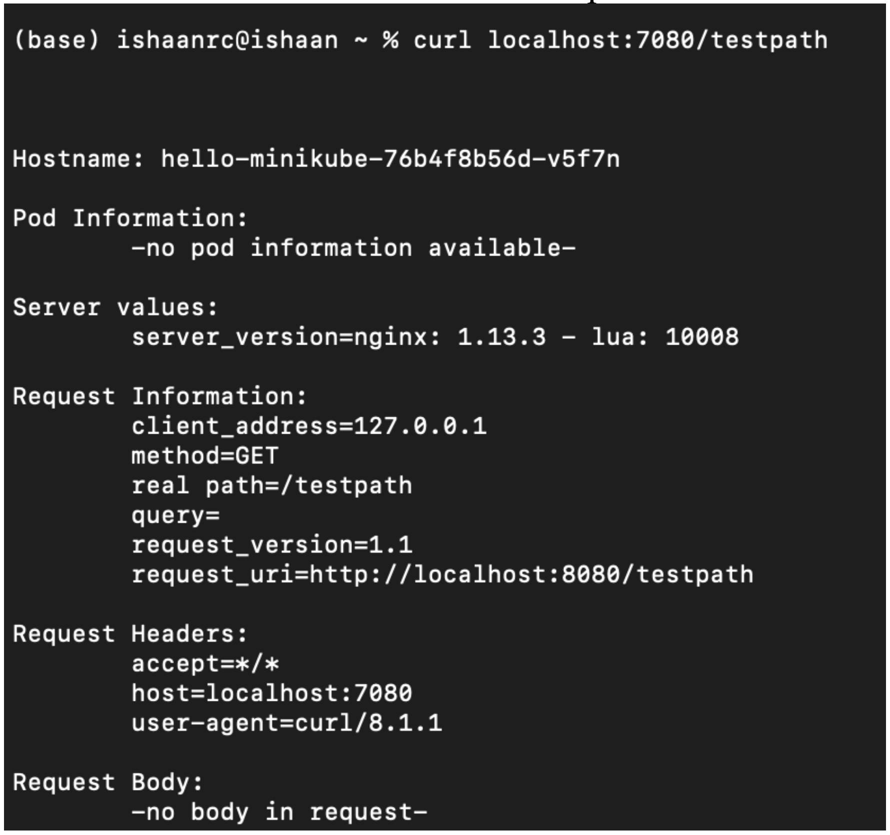
```
(base) ishaanrc@ishaan ~ % curl localhost:7080


Hostname: hello-minikube-76b4f8b56d-v5f7n

Pod Information:
	-no pod information available-

Server values:
	server_version=nginx: 1.13.3 - lua: 10008

Request Information:
	client_address=127.0.0.1
	method=GET
	real path=/
	query=
	request_version=1.1
	request_uri=http://localhost:8080/

Request Headers:
	accept=*/*
	host=localhost:7080
	user-agent=curl/8.1.1

Request Body:
	-no body in request-
```

Screenshot of browser on http://localhost:7080



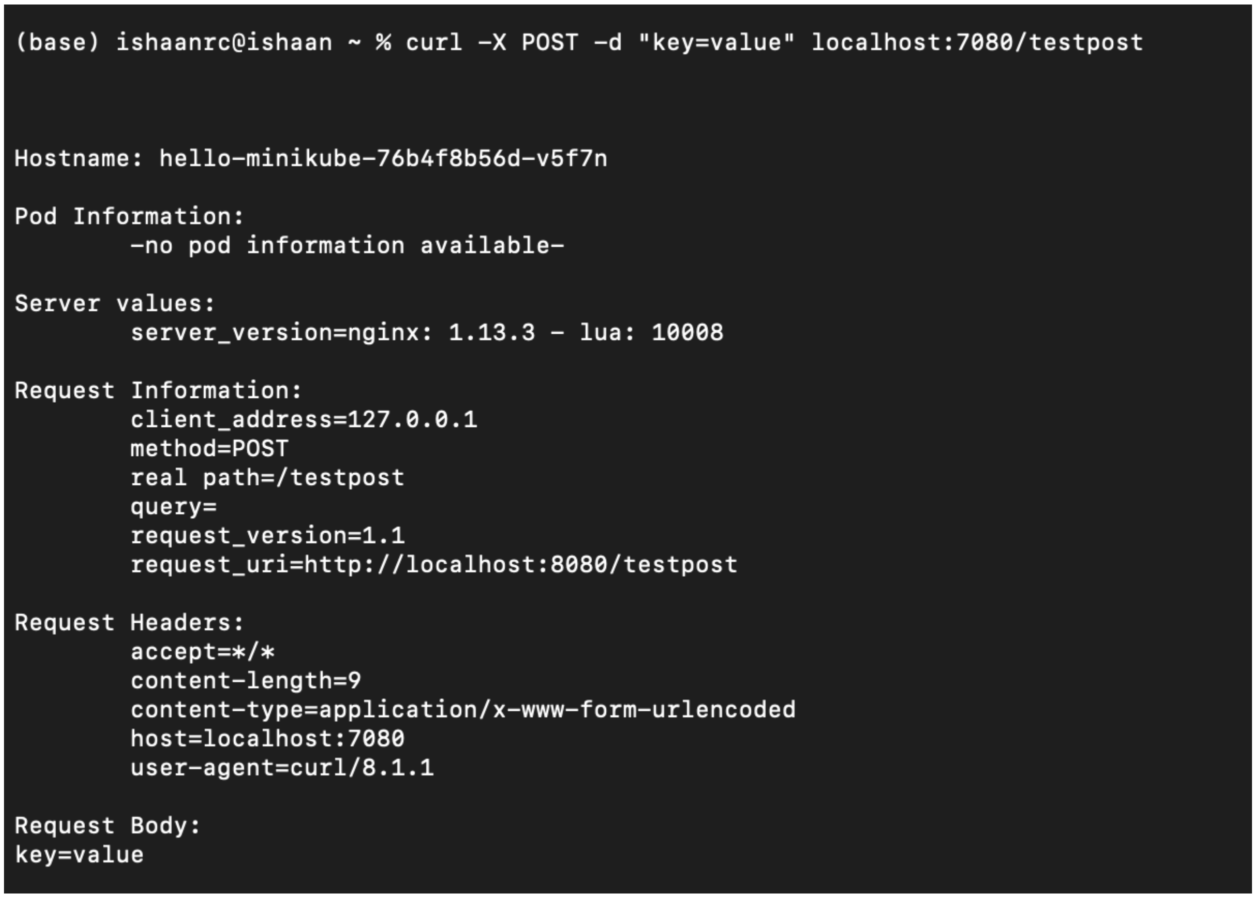
```
Hostname: hello-minikube-76b4f8b56d-v5f7n

Pod Information:
        -no pod information available-

Server values:
        server_version=nginx: 1.13.3 - lua: 10008

Request Information:
        client_address=127.0.0.1
        method=GET
        real path=/
        query=
        request_version=1.1
        request_uri=http://localhost:8080/

Request Headers:
        accept=text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.7
        accept-encoding=gzip, deflate, br
        accept-language=en-US,en;q=0.9
        cache-control=max-age=0
        connection=keep-alive
        host=localhost:7080
        if-modified-since=Tue, 24 Oct 2023 13:46:47 GMT
        if-none-match=&quot;6537cac7-267&quot;
        sec-ch-ua=&quot;Google Chrome&quot;;v=&quot;117&quot;, &quot;Not;A=Brand&quot;;v=&quot;8&quot;, &quot;Chromium&quot;;v=&quot;117&quot;
        sec-ch-ua-mobile=?0
        sec-ch-ua-platform=&quot;macOS&quot;
        sec-fetch-dest=document
        sec-fetch-mode=navigate
        sec-fetch-site=none
        sec-fetch-user=?1
        upgrade-insecure-requests=1
        user-agent=Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/117.0.0.0 Safari/537.36

Request Body:
        -no body in request-
```

Screenshot of when I curl a random path in localhost:



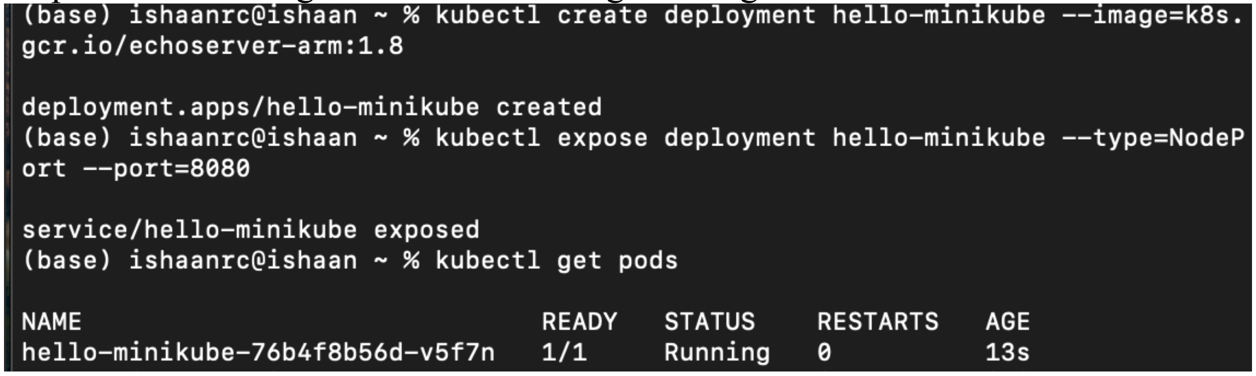
```
(base) ishaanrc@ishaan ~ % curl localhost:7080/testpath



Hostname: hello-minikube-76b4f8b56d-v5f7n

Pod Information:
        -no pod information available-

Server values:
        server_version=nginx: 1.13.3 - lua: 10008

Request Information:
        client_address=127.0.0.1
        method=GET
        real path=/testpath
        query=
        request_version=1.1
        request_uri=http://localhost:8080/testpath

Request Headers:
        accept=*/*
        host=localhost:7080
        user-agent=curl/8.1.1

Request Body:
        -no body in request-
```

Finally, screenshot of curl when I use a POST request and observe the body shown in the BODY section:



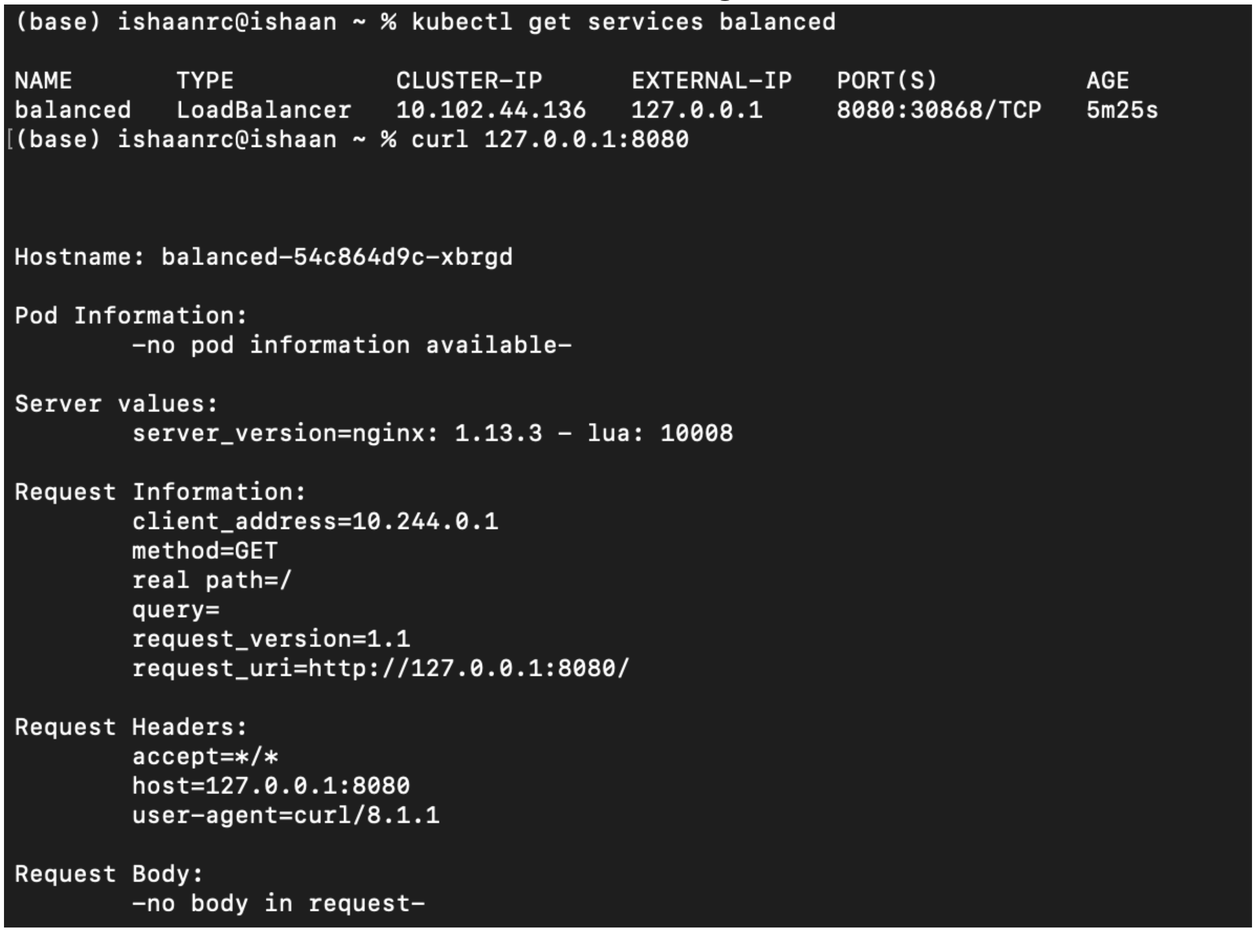
```
(base) ishaanrc@ishaan ~ % curl -X POST -d "key=value" localhost:7080/testpost



Hostname: hello-minikube-76b4f8b56d-v5f7n

Pod Information:
	-no pod information available-

Server values:
	server_version=nginx: 1.13.3 - lua: 10008

Request Information:
	client_address=127.0.0.1
	method=POST
	real path=/testpost
	query=
	request_version=1.1
	request_uri=http://localhost:8080/testpost

Request Headers:
	accept=*/*
	content-length=9
	content-type=application/x-www-form-urlencoded
	host=localhost:7080
	user-agent=curl/8.1.1

Request Body:
key=value
```

Part 2: Add load balancing
(I didn’t face any difficulty in this part)
Steps I followed to get the load balancing working with Minikube tunnel:



```
(base) ishaanrc@ishaan ~ % kubectl create deployment hello-minikube --image=k8s.
gcr.io/echoserver-arm:1.8

deployment.apps/hello-minikube created
(base) ishaanrc@ishaan ~ % kubectl expose deployment hello-minikube --type=NodeP
ort --port=8080

service/hello-minikube exposed
(base) ishaanrc@ishaan ~ % kubectl get pods

NAME                              READY   STATUS    RESTARTS   AGE
hello-minikube-76b4f8b56d-v5f7n   1/1     Running   0          13s
```

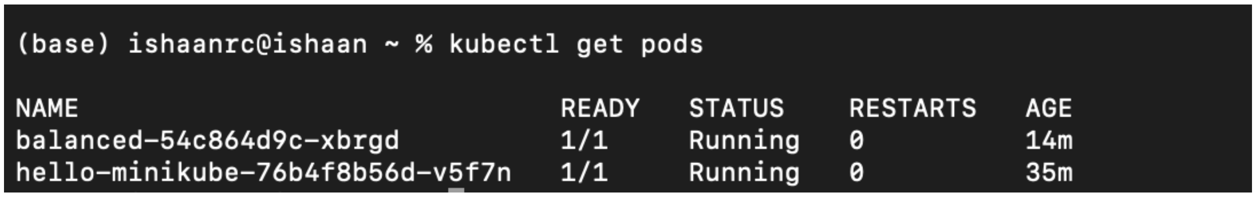
Screenshot of when I ran the command kubectl get services balanced



```
(base) ishaanrc@ishaan ~ % kubectl get services balanced

NAME       TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
balanced   LoadBalancer   10.102.44.136   127.0.0.1     8080:30868/TCP   5m25s
[(base) ishaanrc@ishaan ~ % curl 127.0.0.1:8080



Hostname: balanced-54c864d9c-xbrgd

Pod Information:
        -no pod information available-

Server values:
        server_version=nginx: 1.13.3 - lua: 10008

Request Information:
        client_address=10.244.0.1
        method=GET
        real path=/
        query=
        request_version=1.1
        request_uri=http://127.0.0.1:8080/

Request Headers:
        accept=*/*
        host=127.0.0.1:8080
        user-agent=curl/8.1.1

Request Body:
        -no body in request-
```

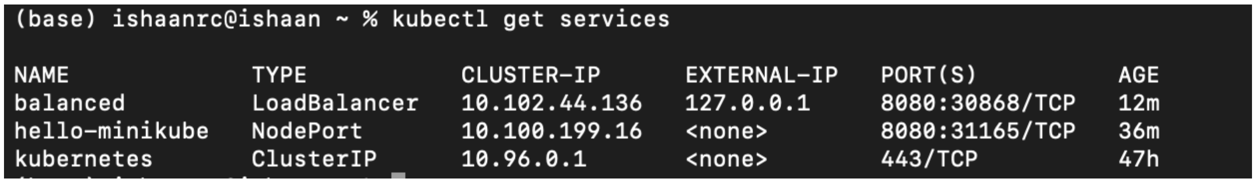
After I got the EXTERNAL-IP address, I ran a few commands to observe my cluster. Below are the following commands I ran:

1) **Command**: kubectl get pods
   Observation: The output shows that there are two pods running: balanced and hello-minikube. The balanced pod has been running for 14 minutes and is in the "Running" state. This means that the application within the pod is active and can serve traffic.



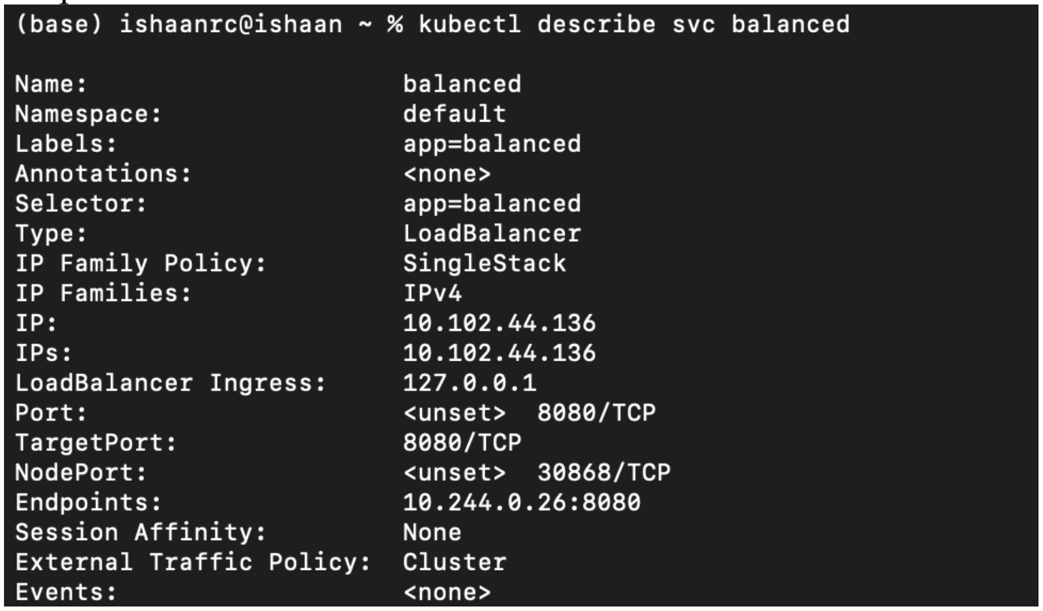
```
(base) ishaanrc@ishaan ~ % kubectl get pods

NAME                              READY   STATUS    RESTARTS   AGE
balanced-54c864d9c-xbrgd          1/1     Running   0          14m
hello-minikube-76b4f8b56d-v5f7n   1/1     Running   0          35m
```

2) **Command**: kubectl get services
Observation: The service named **balanced** is of type **LoadBalancer**. This means that it's set up to distribute incoming network traffic across the pods. The external IP is **127.0.0.1**, which is a loopback address. This indicates that the service is accessible from the local machine.



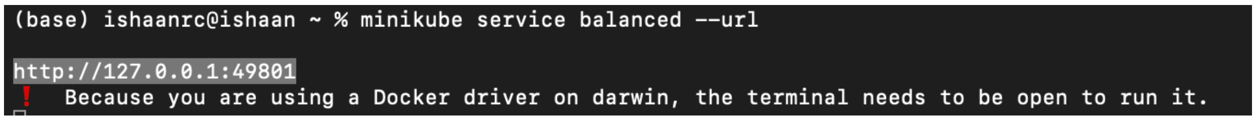
```
(base) ishaanrc@ishaan ~ % kubectl get services

NAME             TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
balanced         LoadBalancer   10.102.44.136   127.0.0.1     8080:30868/TCP   12m
hello-minikube   NodePort       10.100.199.16   <none>        8080:31165/TCP   36m
kubernetes       ClusterIP      10.96.0.1       <none>        443/TCP          47h
```

3) **Command**: kubectl describe svc balanced
Observation: It confirms that the type is LoadBalancer and provides the IP addresses associated with it. The LoadBalancer Ingress is set to 127.0.0.1, indicating where the traffic will be directed. The endpoints section shows the IP and port where the pods behind this service are running and ready to accept traffic.



```
(base) ishaanrc@ishaan ~ % kubectl describe svc balanced

Name:                     balanced
Namespace:                default
Labels:                   app=balanced
Annotations:              <none>
Selector:                 app=balanced
Type:                     LoadBalancer
IP Family Policy:         SingleStack
IP Families:              IPv4
IP:                       10.102.44.136
IPs:                      10.102.44.136
LoadBalancer Ingress:     127.0.0.1
Port:                     <unset>  8080/TCP
TargetPort:               8080/TCP
NodePort:                 <unset>  30868/TCP
Endpoints:                10.244.0.26:8080
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

4) **Command**: minikube service balanced --url
Observation: The output provides the URL where we can access the service. The fact that it specifies a port (**:49801**) indicates that it's a dynamically allocated port for accessing the service on the local machine.



```
(base) ishaanrc@ishaan ~ % minikube service balanced --url

http://127.0.0.1:49801
❗  Because you are using a Docker driver on darwin, the terminal needs to be open to run it.
```

Part 3: Creating your own microservice/backend service with a Minikube cluster. (I didn't face any difficulty in this part)

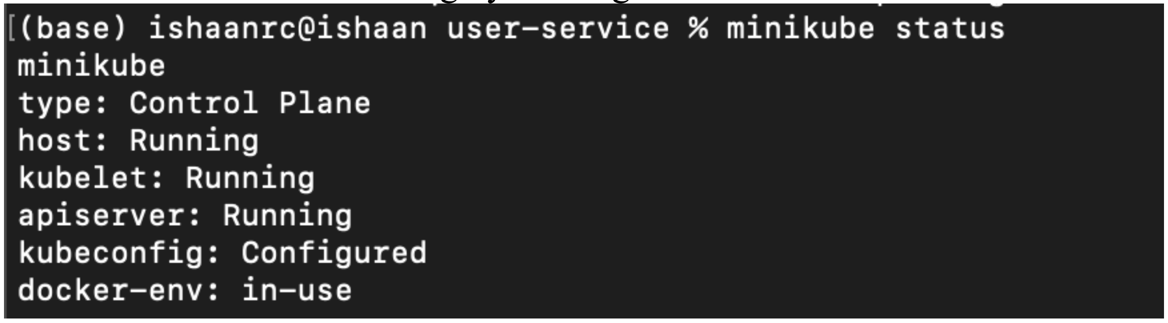
I used the image that was provided in Lab 2, for user microservice.
Cluster Details:

- **Cluster Type:** Minikube
- **Command Used for Creation:** minikube start --nodes 2 -p <cluster name>

I followed the steps given in the assignment handout and made a config.json file:

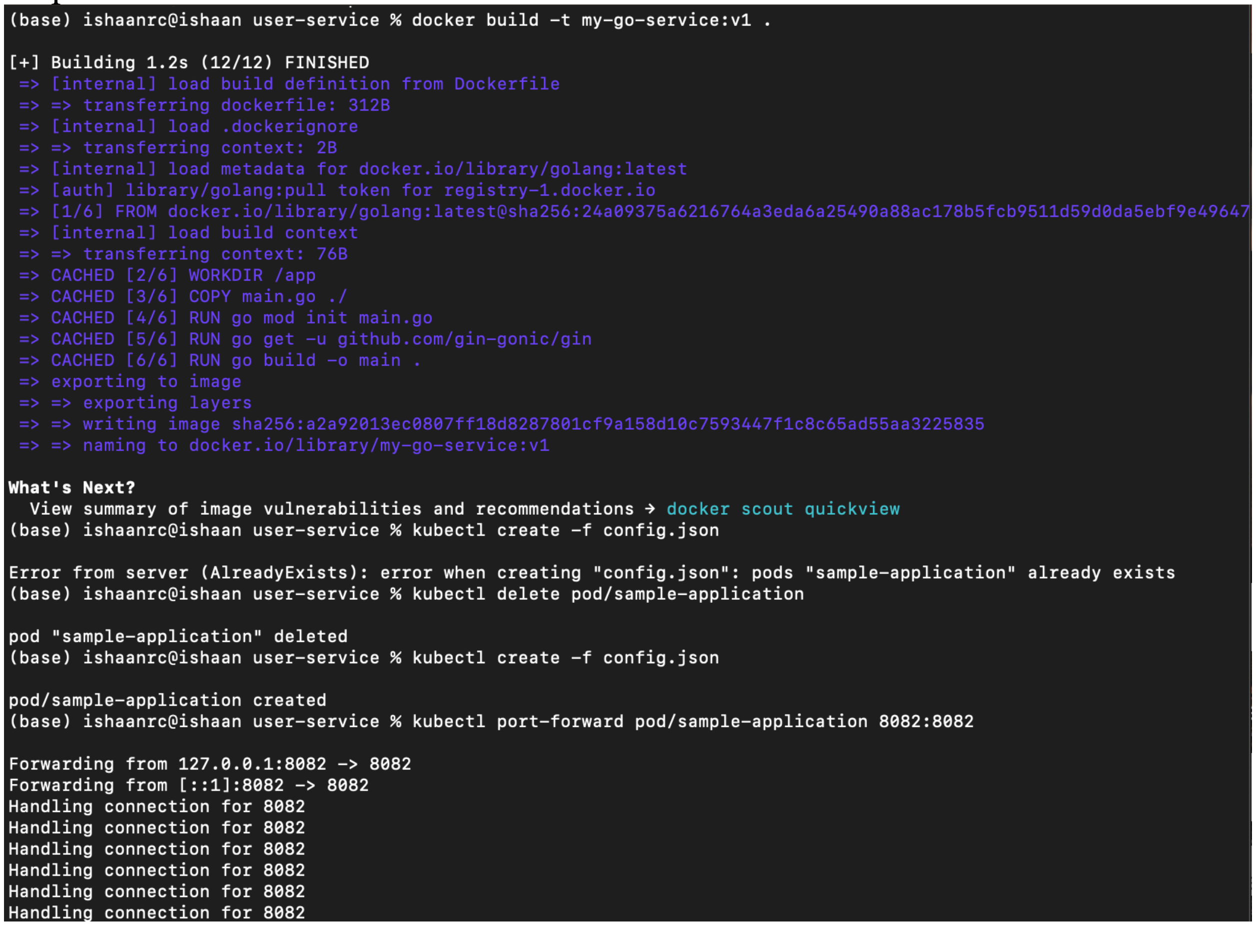
```
{
    "apiVersion": "v1",
    "kind": "Pod",
    "metadata": {
      "name": "sample-application"
    },
    "spec": {
      "containers": [
        {
          "name": "my-go-service",
          "image": "my-go-service:v1",
          "imagePullPolicy": "IfNotPresent",
          "ports": [
            {
              "containerPort": 8082
            }
          ],
          "command": ["./main"]
        }
      ]
    }
  }
```

I used the main.go file and the Dockerfile as submitted in Lab 2(This time on 8082 port)
Then I verified it was working by running the minikube status command:



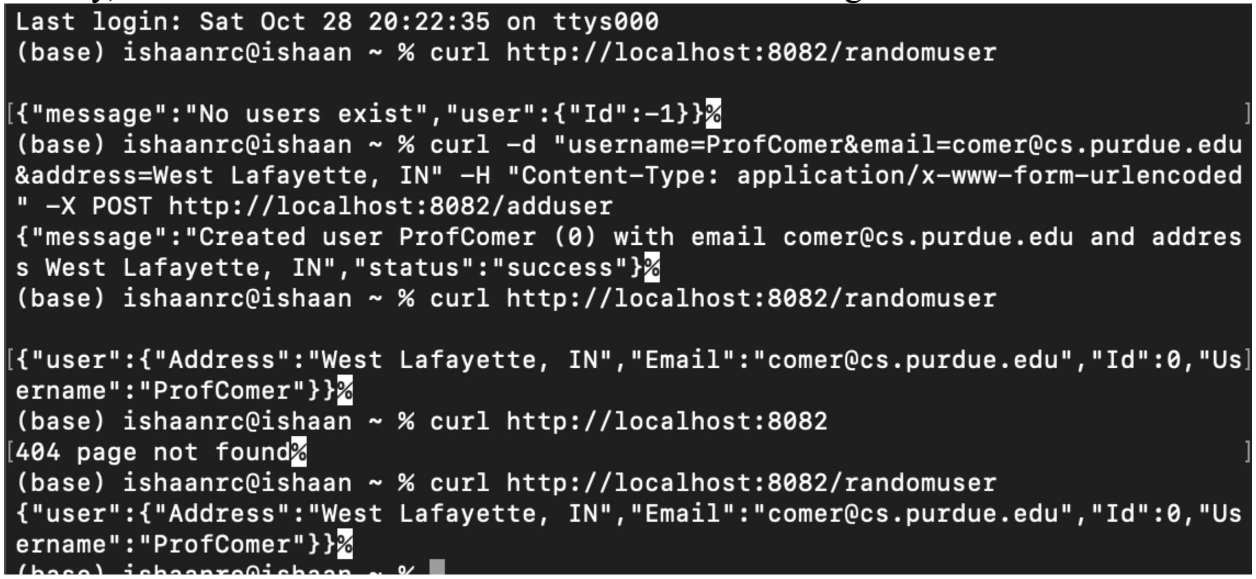
```
[(base) ishaanrc@ishaan user-service % minikube status
minikube
type: Control Plane
host: Running
kubelet: Running
apiserver: Running
kubeconfig: Configured
docker-env: in-use
```

Steps I followed:



```
(base) ishaanrc@ishaan user-service % docker build -t my-go-service:v1 .

[+] Building 1.2s (12/12) FINISHED
 => [internal] load build definition from Dockerfile
 => => transferring dockerfile: 312B
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load metadata for docker.io/library/golang:latest
 => [auth] library/golang:pull token for registry-1.docker.io
 => [1/6] FROM docker.io/library/golang:latest@sha256:24a09375a6216764a3eda6a25490a88ac178b5fcb9511d59d0da5ebf9e49647
 => [internal] load build context
 => => transferring context: 76B
 => CACHED [2/6] WORKDIR /app
 => CACHED [3/6] COPY main.go ./
 => CACHED [4/6] RUN go mod init main.go
 => CACHED [5/6] RUN go get -u github.com/gin-gonic/gin
 => CACHED [6/6] RUN go build -o main .
 => exporting to image
 => => exporting layers
 => => writing image sha256:a2a92013ec0807ff18d8287801cf9a158d10c7593447f1c8c65ad55aa3225835
 => => naming to docker.io/library/my-go-service:v1

What's Next?
  View summary of image vulnerabilities and recommendations → docker scout quickview
(base) ishaanrc@ishaan user-service % kubectl create -f config.json

Error from server (AlreadyExists): error when creating "config.json": pods "sample-application" already exists
(base) ishaanrc@ishaan user-service % kubectl delete pod/sample-application

pod "sample-application" deleted
(base) ishaanrc@ishaan user-service % kubectl create -f config.json

pod/sample-application created
(base) ishaanrc@ishaan user-service % kubectl port-forward pod/sample-application 8082:8082

Forwarding from 127.0.0.1:8082 -> 8082
Forwarding from [::1]:8082 -> 8082
Handling connection for 8082
Handling connection for 8082
Handling connection for 8082
Handling connection for 8082
Handling connection for 8082
Handling connection for 8082
```

Finally, I ran the curl command to make sure it's working:



```
Last login: Sat Oct 28 20:22:35 on ttys000
(base) ishaanrc@ishaan ~ % curl http://localhost:8082/randomuser

{"message":"No users exist","user":{"Id":-1}}%
(base) ishaanrc@ishaan ~ % curl -d "username=ProfComer&email=comer@cs.purdue.edu
&address=West Lafayette, IN" -H "Content-Type: application/x-www-form-urlencoded
" -X POST http://localhost:8082/adduser
{"message":"Created user ProfComer (0) with email comer@cs.purdue.edu and addres
s West Lafayette, IN","status":"success"}%
(base) ishaanrc@ishaan ~ % curl http://localhost:8082/randomuser

{"user":{"Address":"West Lafayette, IN","Email":"comer@cs.purdue.edu","Id":0,"Us
ername":"ProfComer"}}%
(base) ishaanrc@ishaan ~ % curl http://localhost:8082
404 page not found%
(base) ishaanrc@ishaan ~ % curl http://localhost:8082/randomuser
{"user":{"Address":"West Lafayette, IN","Email":"comer@cs.purdue.edu","Id":0,"Us
ername":"ProfComer"}}%
(base) ishaanrc@ishaan ~ %
```

Image details and purpose:

- **Purpose:** The Docker image was designed to run a Go application using the Gin web framework. This application offers endpoints for user management: fetching a random user, adding a new user, and retrieving a user by ID, etc.
- **Port Usage:** The application is designed to listen and respond on port **8082**.
- **Docker Integration with Minikube:** Used **eval $(minikube docker-env)** to align the Docker daemon with the Minikube environment.

Deployment Technique and Details:
For this assignment, I utilized two nodes. I used a Kubernetes configuration file to install the custom Go application within the Kubernetes environment. This file, which can be in.json or.yaml format, acts as a template for Kubernetes to follow in order to establish how the application should be executed within the cluster. The number of replicas, the Docker image to use, the ports to expose, and any startup commands or arguments that should be run are all set inside this configuration.

The command $ sudo kubectl create -f config.json was used for the actual deployment. This tells Kubernetes to generate the resources specified in the config.json file.

Description of the custom service:
The custom Go application, built with the Gin web framework, serves as a user administration microservice. It has features, ranging from a 'Random User Retrieval' feature that provides random user details from its database to a 'Add New User' endpoint that allows for dynamic data entry by accepting key user details. Furthermore, the 'Retrieve User by ID' feature streamlines the process of identifying specific user data amongst a rising database, exemplifying effective user data processing in a compact service.

Once I deployed it on localhost using the steps above, I could access it using the curl command as shown in the screenshots. I can add/retrieve any user as we can see from the screenshot above. And we can also get a random user back.

Part 4: Analysis

1) Kubernetes uses nested layers of virtualization and abstraction in the form of containers, pods, nodes, and clusters. When you deployed the microservices in part 2, how many types of virtualization or abstraction were used (including your AWS VM)?
   There were many nested layers of virtualization and abstraction used in part 2. The first one was through containers when we run the command kubectl create deployment balanced <image name>. Here we use the echoserver image that will run as a container when deployed. In terms of pods, we encapsulate our service in a pod, which is the smallest deployable unit. When we run kubectl create deployment, a pod is created to run the container. This adds another layer. One more layer is from nodes; Minikube creates a single-node Kubernetes cluster within a virtual machine on our local system. So, in the case of our Part 2, the minikube tunnel command and other deployment operations entail interactions with this one node, adding another layer. When we use the command kubectl expose deployment balanced --type=LoadBalancer --port=8080 to expose our deployment, we are creating a Kubernetes service of type LoadBalancer. This is a type of abstraction. Services in Kubernetes allow us to access pods using a consistent IP address and DNS name, regardless of which node they are operating on. A cluster is the whole Kubernetes infrastructure, including one or more nodes, the control plane, and many components that aid in the orchestration of containerized applications. This also adds a layer. So, in all, in part 2, we have achieved virtualization and abstraction through all these layers mentioned above.

2) Why might Kubernetes use configuration json or yaml files to configure a cluster rather than command line arguments?
   A pod configuration might contain metadata, specifications, ports information, environment variables, etc. Capturing all this in command line would not be practical and prone to errors. Json or yaml is a good alternative to this. They are indented well and easily readable in these formats rather than a long command line input. This can also lead to reusability. We can store these templates and use them again as per need. This also in turn allows for version control and rolling back to a configuration if needed. And since Kubernetes uses the late binding approach, json or yaml is more feasible as the template can be defined and instantiated as needed. Lastly, since changes to json or yaml files do not affect running pods and apply only to the new pods, immutability of running resources is easier.