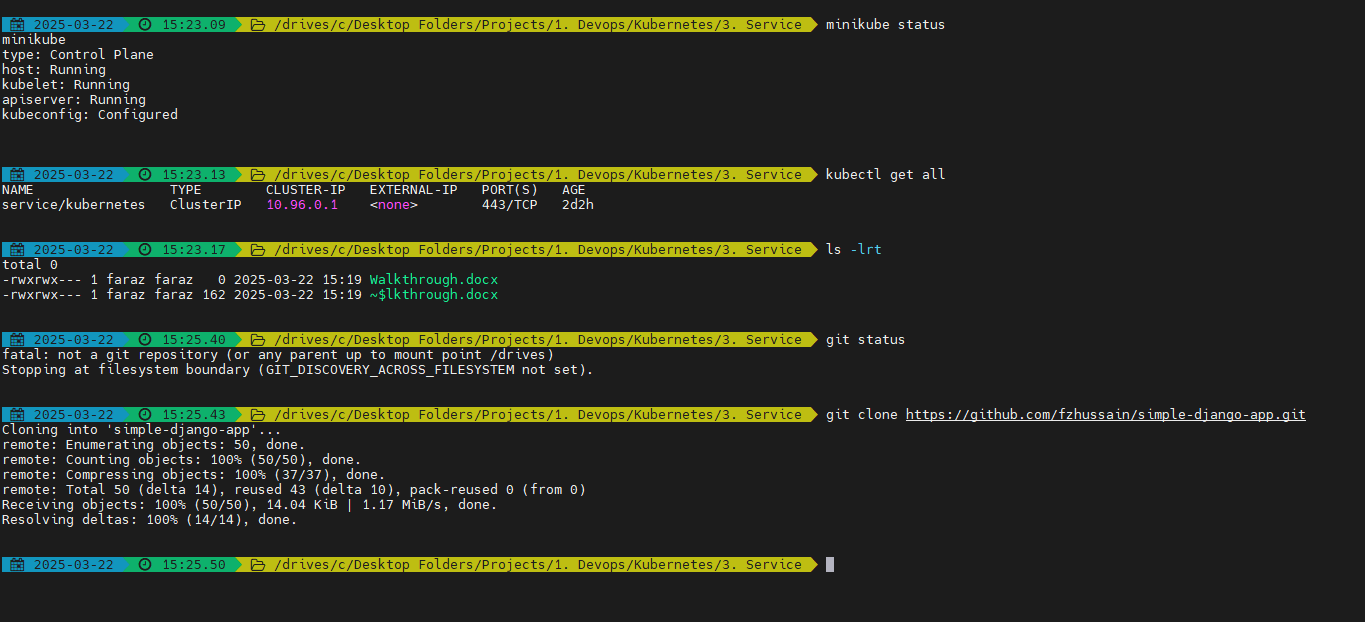
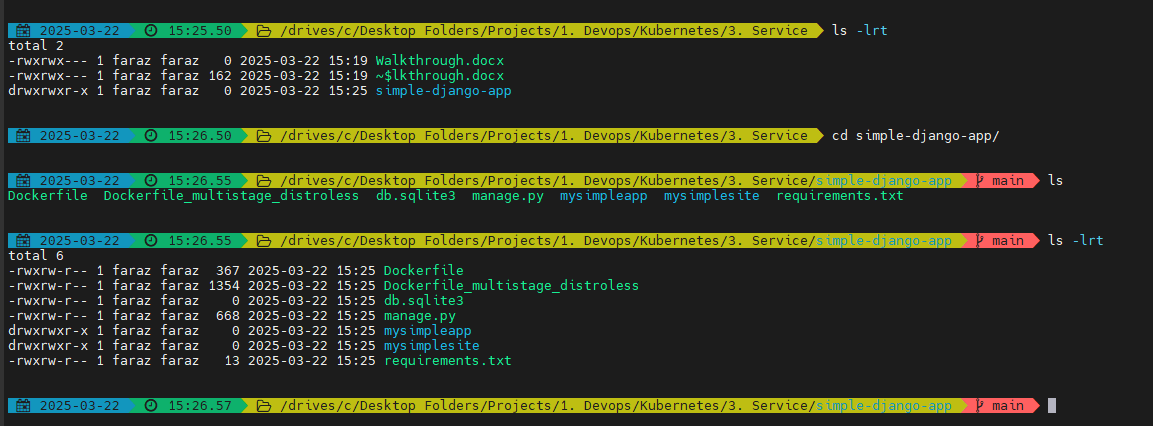
Make sure you start the minikube cluster using:

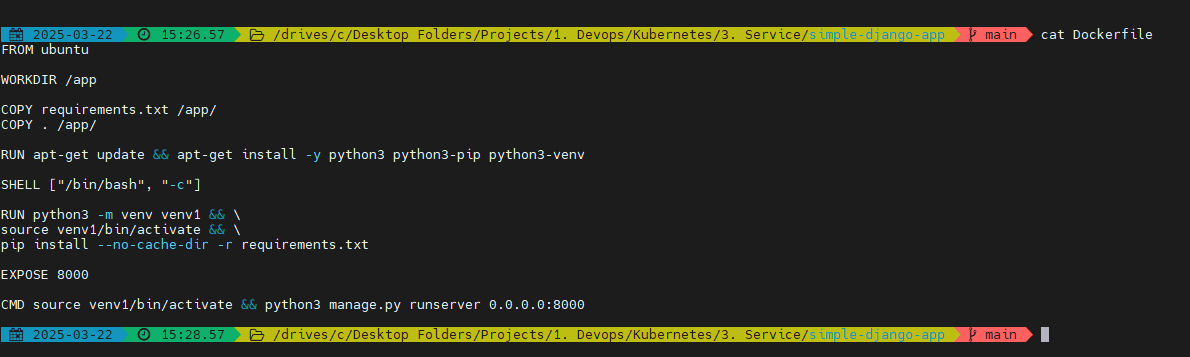
* minikube start

Clone the repository using:

* git clone https://github.com/fzhussain/simple-django-app.git

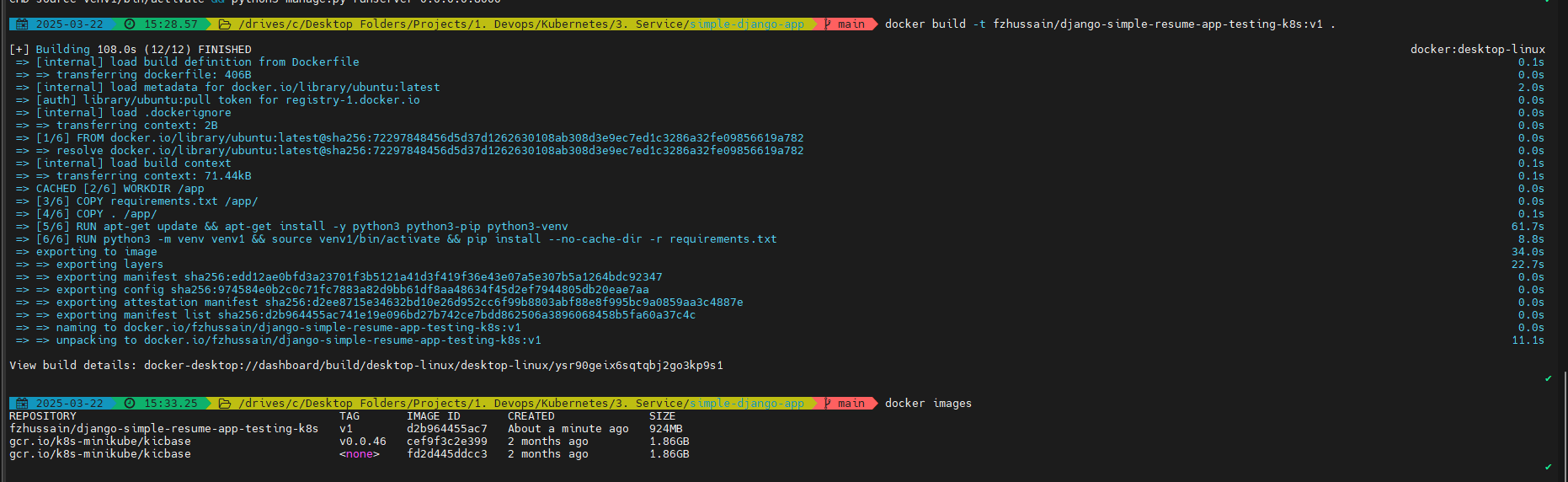






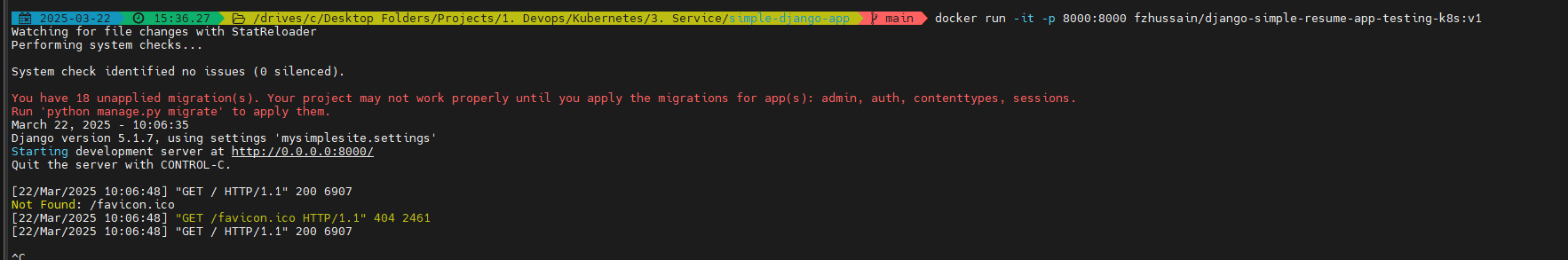
Now build the image:

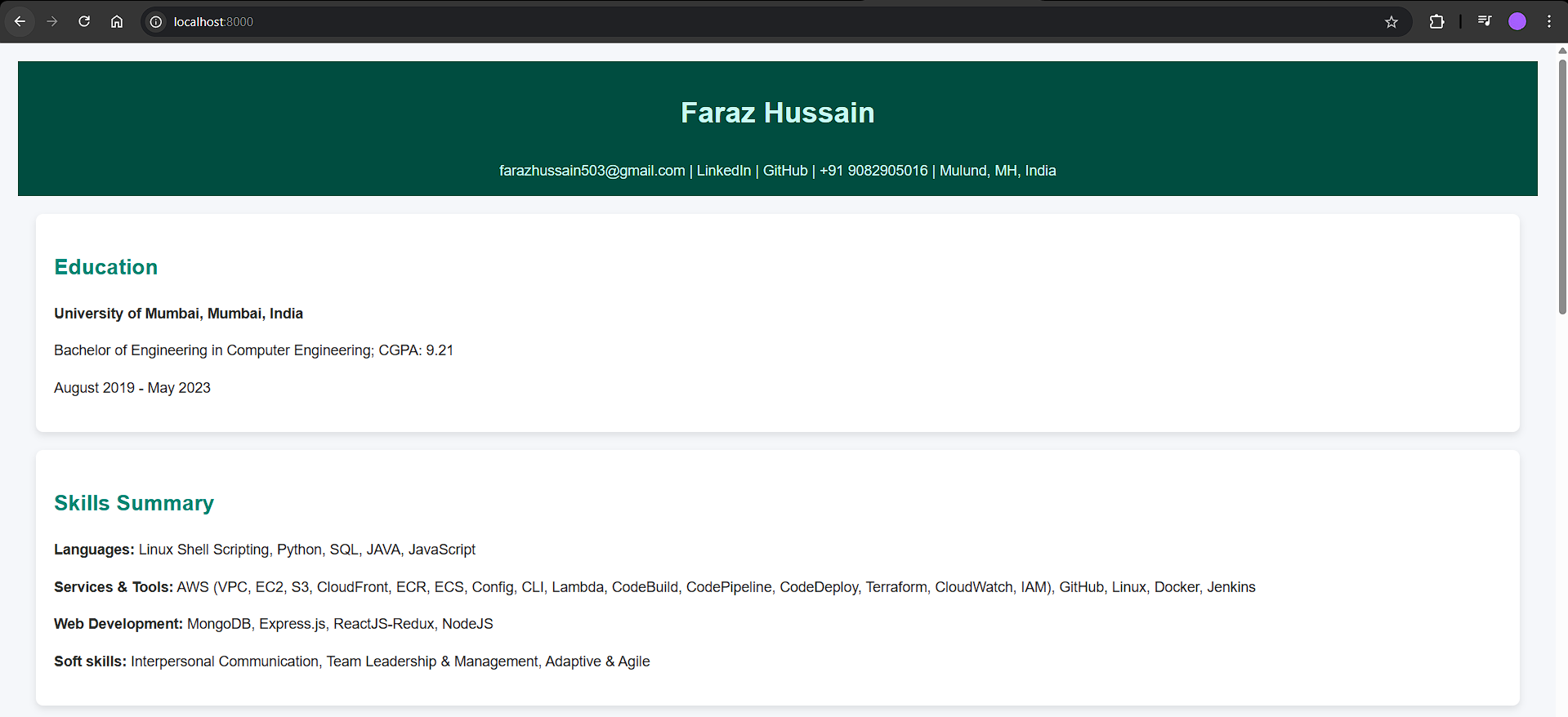
* docker build -t fzhussain/django-simple-resume-app-testing-k8s:v1 .



Run the container and verify if the build image is running successfully:

* docker run -it -p 8000:8000 fzhussain/django-simple-resume-app-testing-k8s:v1





Now we will create K8s deployment:

Go to: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

Use the simple example:

apiVersion: apps/v1

kind: Deployment

metadata:

name: faraz-django-app

labels:

app: faraz-django-app-label

spec:

replicas: 2

selector:

matchLabels:

app: faraz-django-app-label

template:

metadata:

labels:

app: faraz-django-app-label

spec:

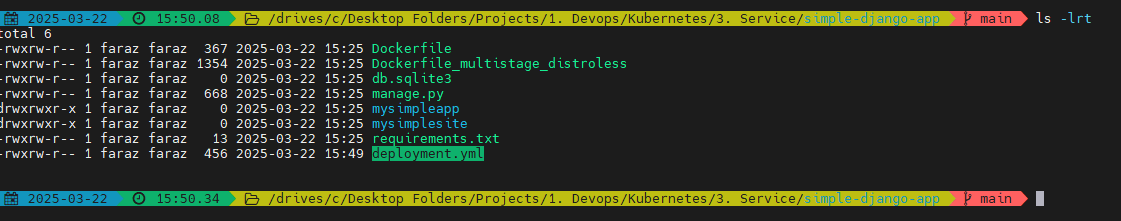
containers:

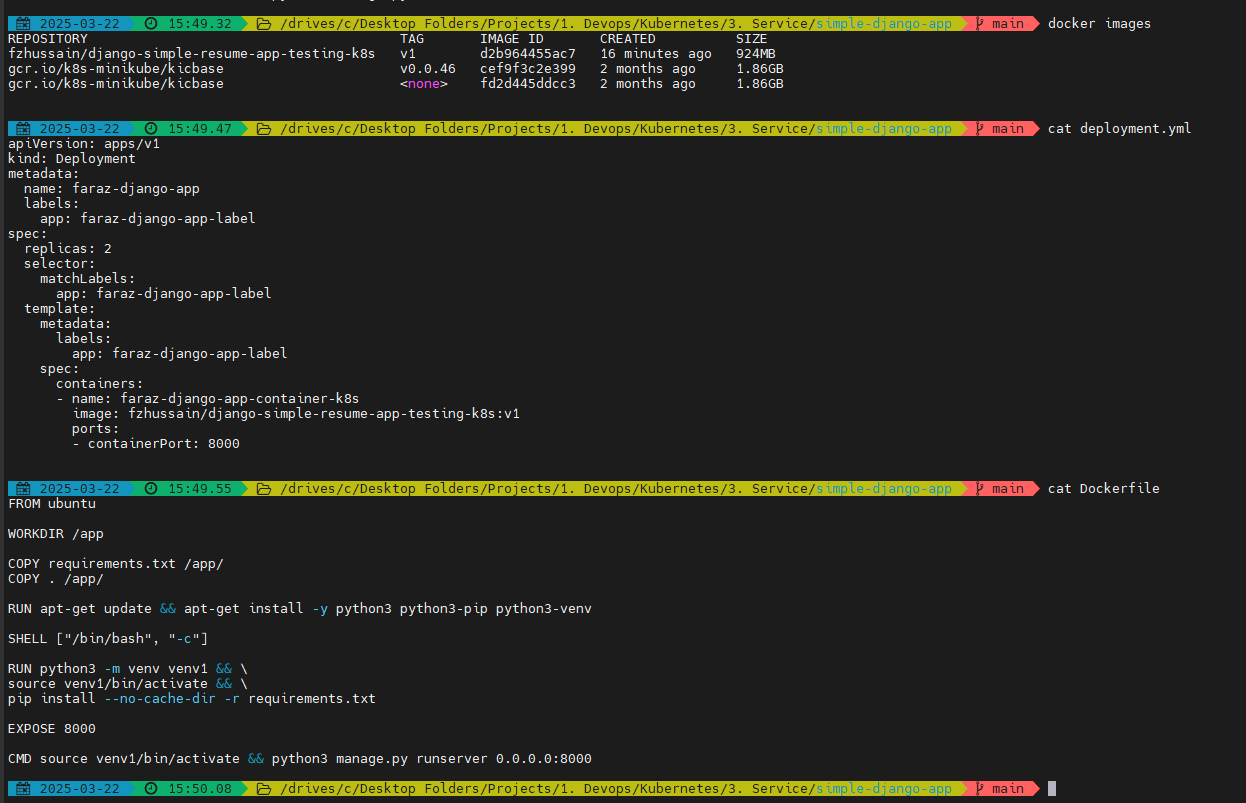
- name: faraz-django-app-container-k8s

image: fzhussain/django-simple-resume-app-testing-k8s:v1

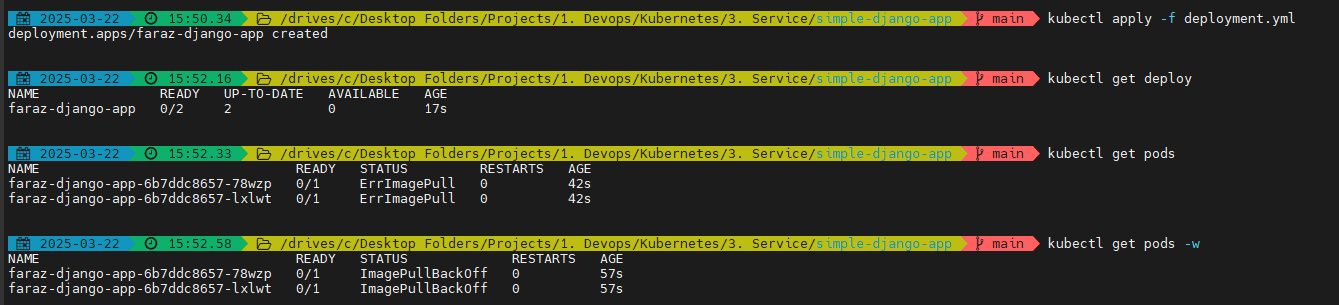
ports:

- containerPort: 8000

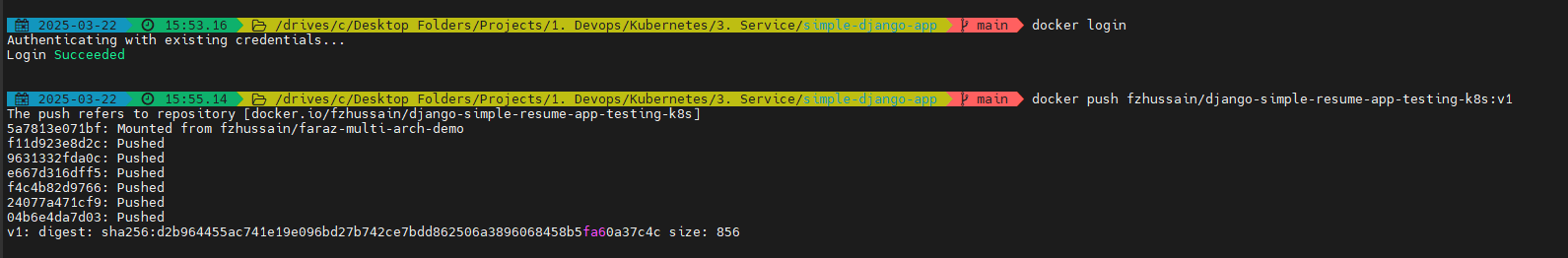




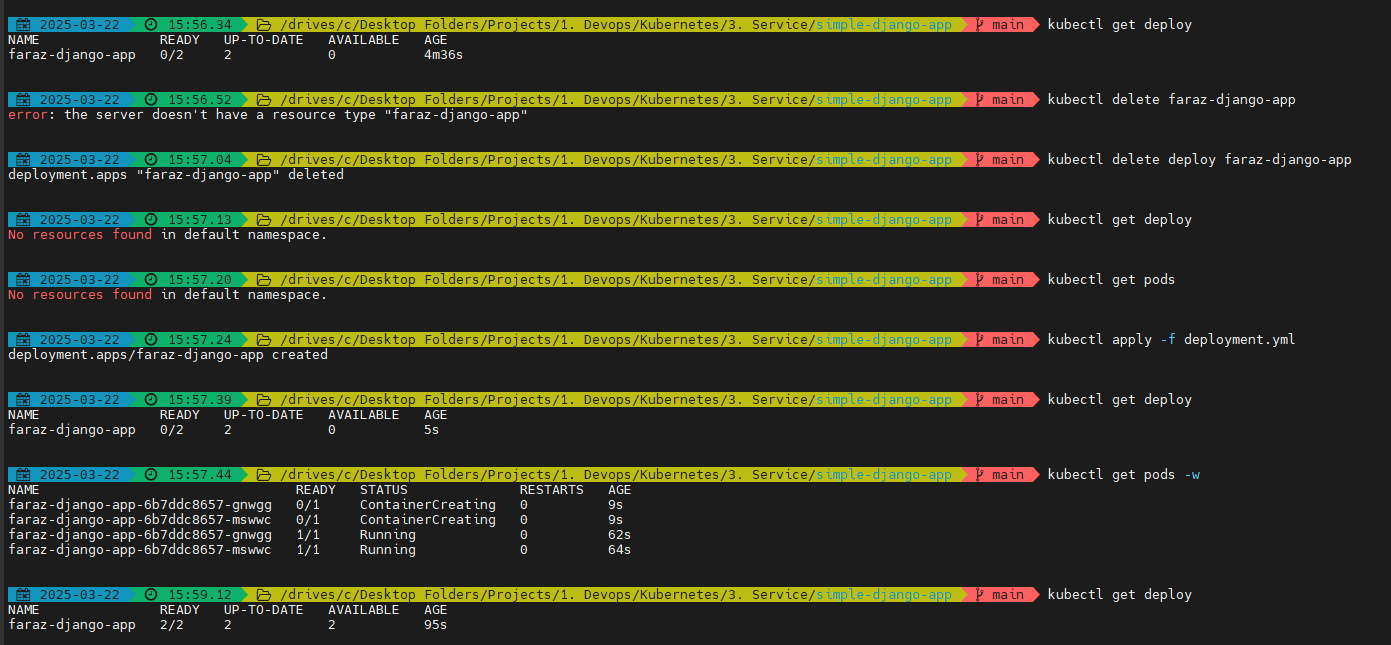
Now apply the deployment

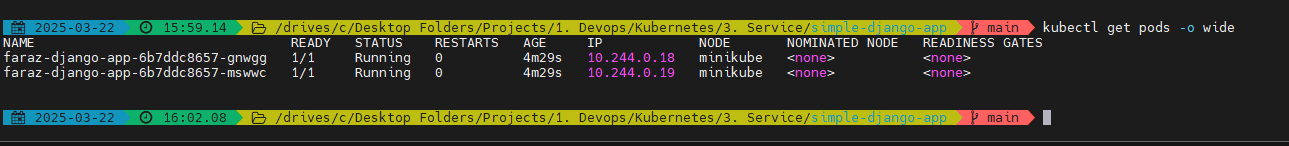


Here, we may get error on Image pull. It can be solved by simply pushing the image to registry



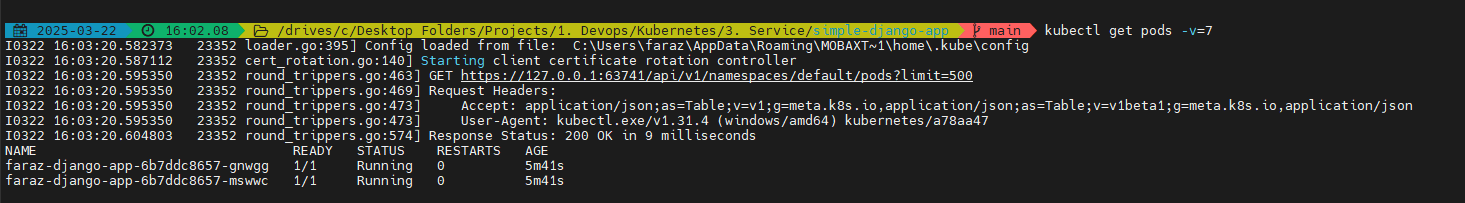
Now the pods are up and running:



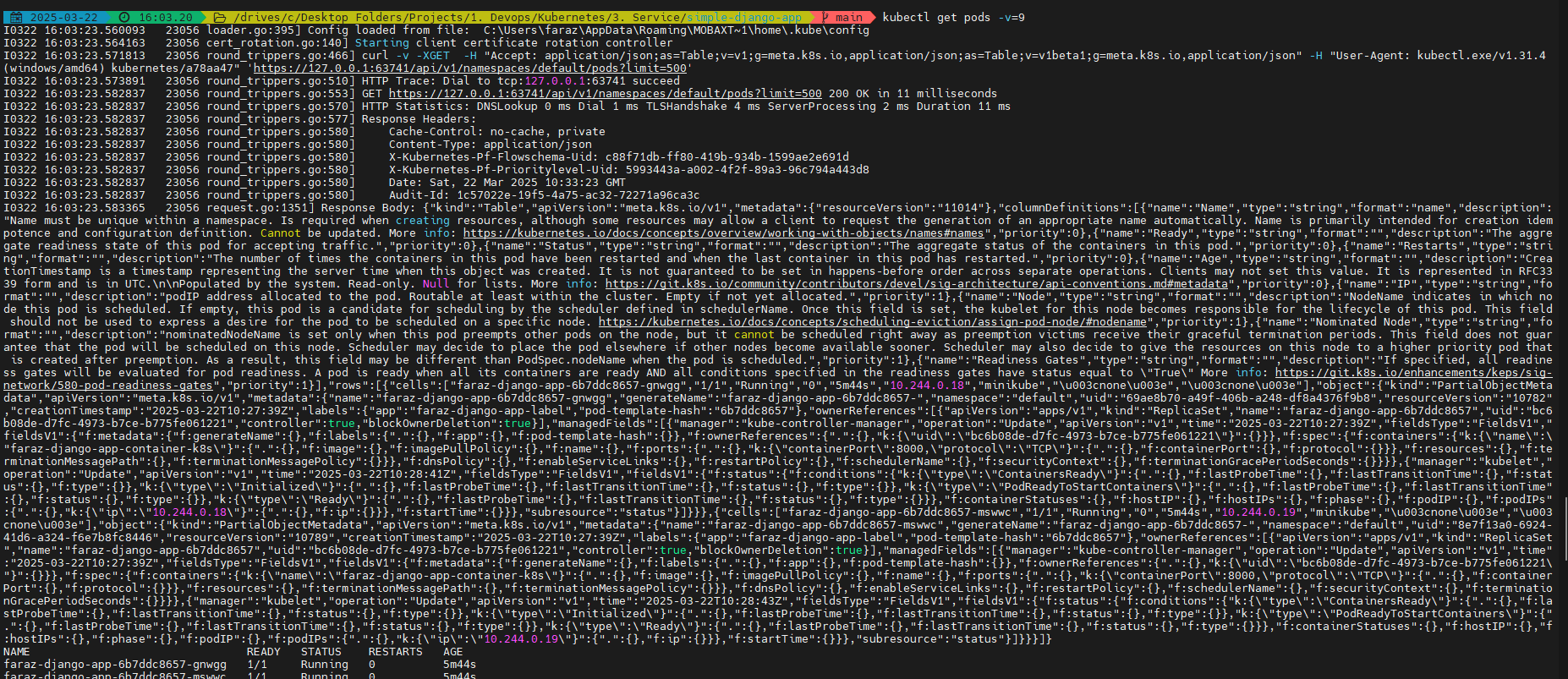


If you are curious to know how kubectl is talking to API server:

* kubectl get pods -v=7 –o json



* kubectl get pods -v=9

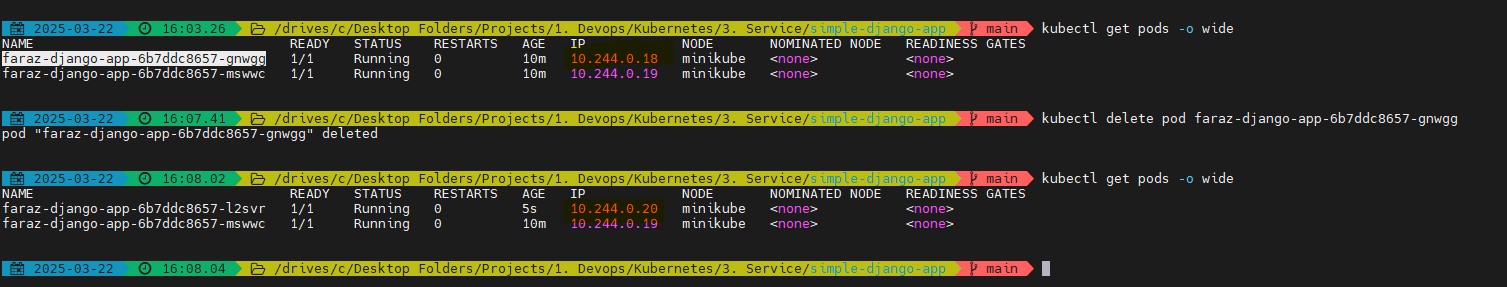


The maximum verbosity level for kubectl is -v=10. At this level, kubectl will log extremely detailed information, including internal processing logic and API server communications. This level is typically used for debugging very specific or complex issues, as it produces a significant amount of output.

The command kubectl get pods -v=7 retrieves information about the pods in your Kubernetes cluster, but the -v=7 flag adds verbose output for debugging purposes.

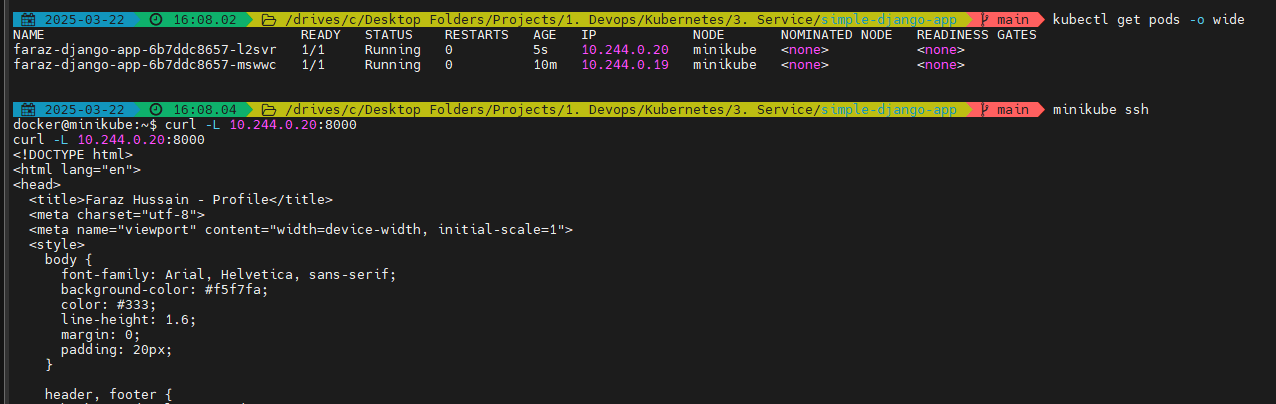
This higher verbosity level (-v=7) enables detailed logging, which can help you understand what is happening during the execution of the kubectl command.

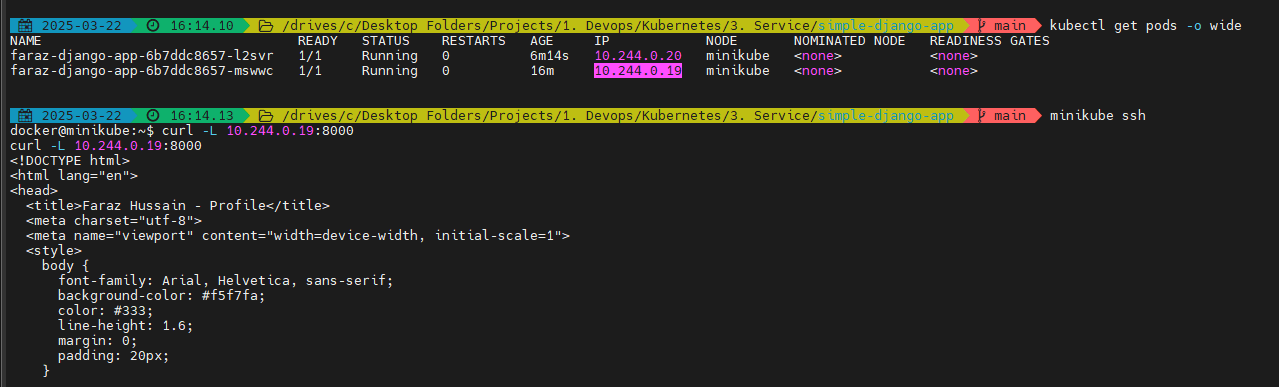
Now if you check the problem with deployment is that if we delete the pod or the pod crashes due to some reason. The IP address will change as K8s does dynamic allocation of IP addresses:



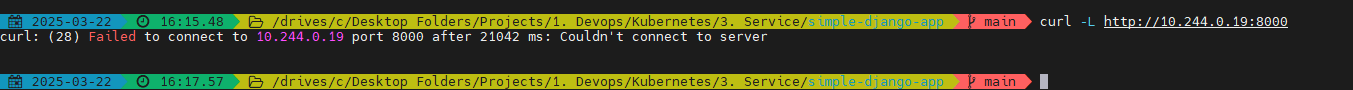
Hence, we require a service discovery mechanism. Hence, K8s doesn’t uses IP addresses for Service Discovery but uses Labels and Selectors

You can always CURL and check the application traffic:





We wont be able to access the application outside the cluster because K8s by default uses cluster IP service



But the people within your organization & end users won’t be able to get this application as it is scoped within your cluster only. To solve this, we can use Services of NodePort (expose worker node IP addresses) or Load Balancer (Public IP address) mode.

Now we will create a service of type NodePort:

<https://kubernetes.io/docs/concepts/services-networking/service/#type-nodeport>

Create service.yml

apiVersion: v1

kind: Service

metadata:

name: faraz-django-app-service

spec:

type: NodePort

selector:

# Service will be looking at pods using selectors

# Always pick from the template > metadata > label > app which is for Pods as we might have different labels for deployments

app: faraz-django-app-label

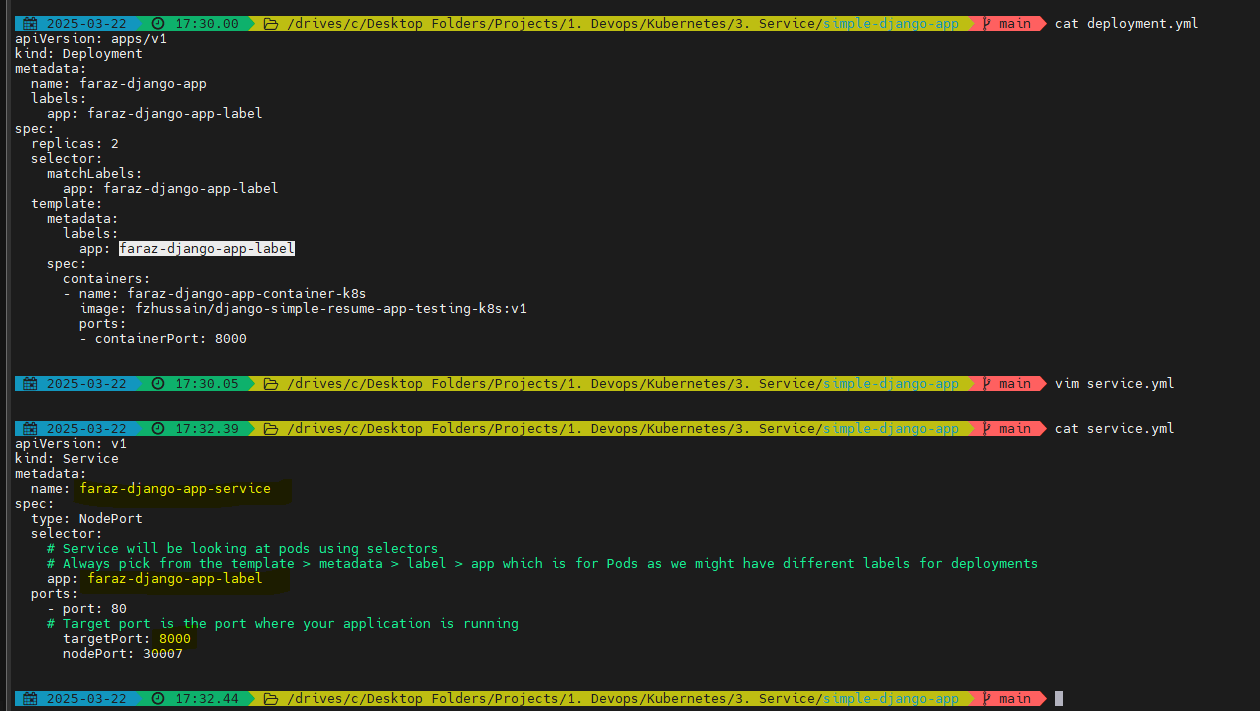
ports:

- port: 80

# Target port is the port where your application is running

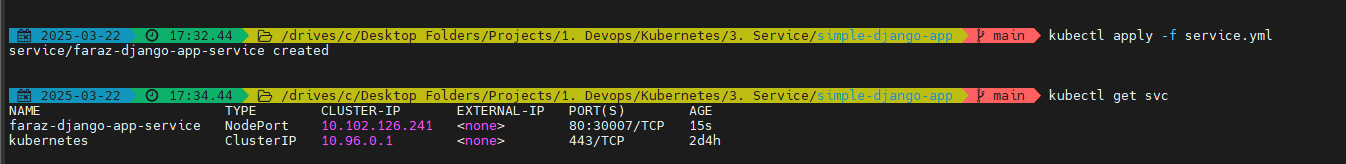
targetPort: 8000

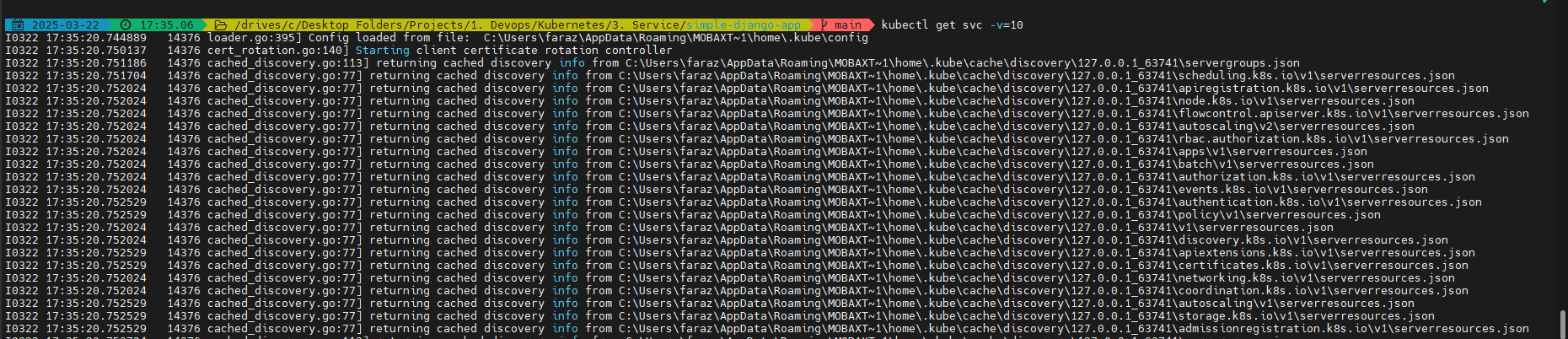
nodePort: 30007



Now apply the changes:

* kubectl apply -f service.yml

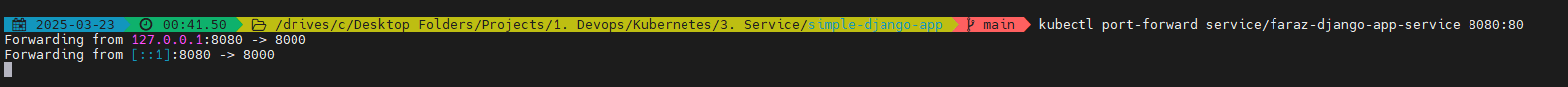




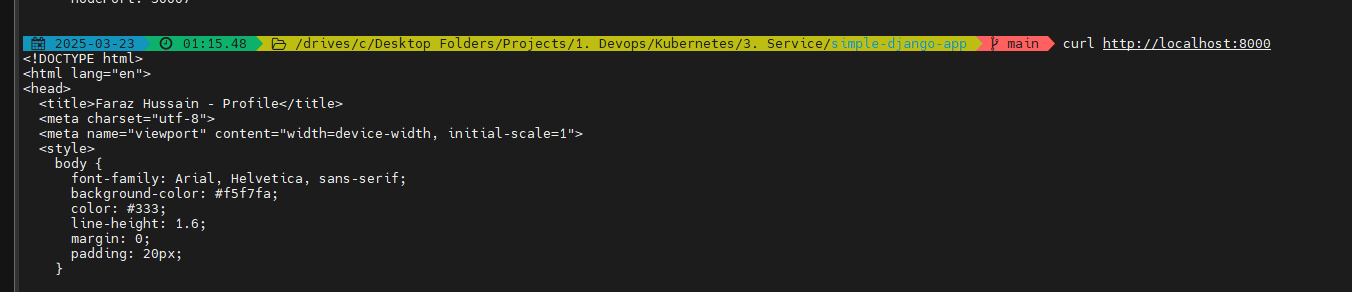
If using docker driver:

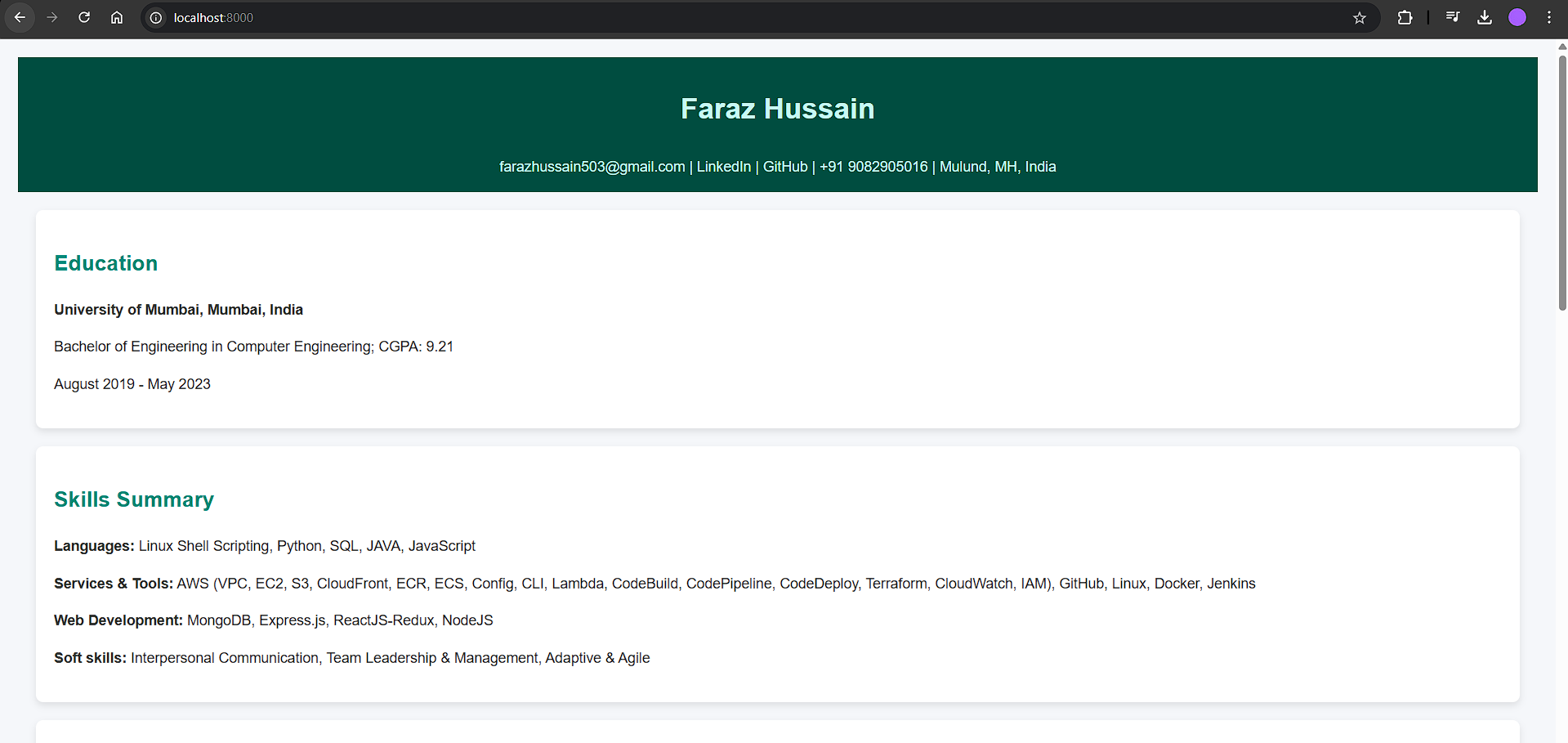
Now forward the port:

* kubectl port-forward service/faraz-django-app-service 8000:80



I will be able to view it on: <http://localhost:8000/>





**1. Why http://192.168.49.2:30007 Doesn't Work**

**Minikube Networking with Docker Driver:** When you use the Docker driver for Minikube, the NodePort (192.168.49.2:30007) is bound to the Minikube virtual environment (VM). Your host machine (Windows) cannot directly access this internal VM address unless additional networking is configured.

**Network Isolation:** The VM isolates its IP (192.168.49.2), which is why your host machine cannot connect to it. By default, Docker doesn't allow direct communication between the host system and the containerized cluster unless specific network bridges are set up.

**2. Why http://localhost:8000 Works**

When you run the kubectl port-forward command, you explicitly map a local port (8000 on your host machine) to the service's port (80 within the Kubernetes cluster). Here's why this works:

**Direct Tunnel:** Port-forwarding bypasses the networking limitations of Minikube and creates a direct tunnel from your local machine (localhost) to the Kubernetes service.

**No Dependence on NodePort:** Unlike NodePort, port-forwarding doesn't involve the Minikube VM or its internal network. Instead, it opens a direct connection from your host to the cluster.

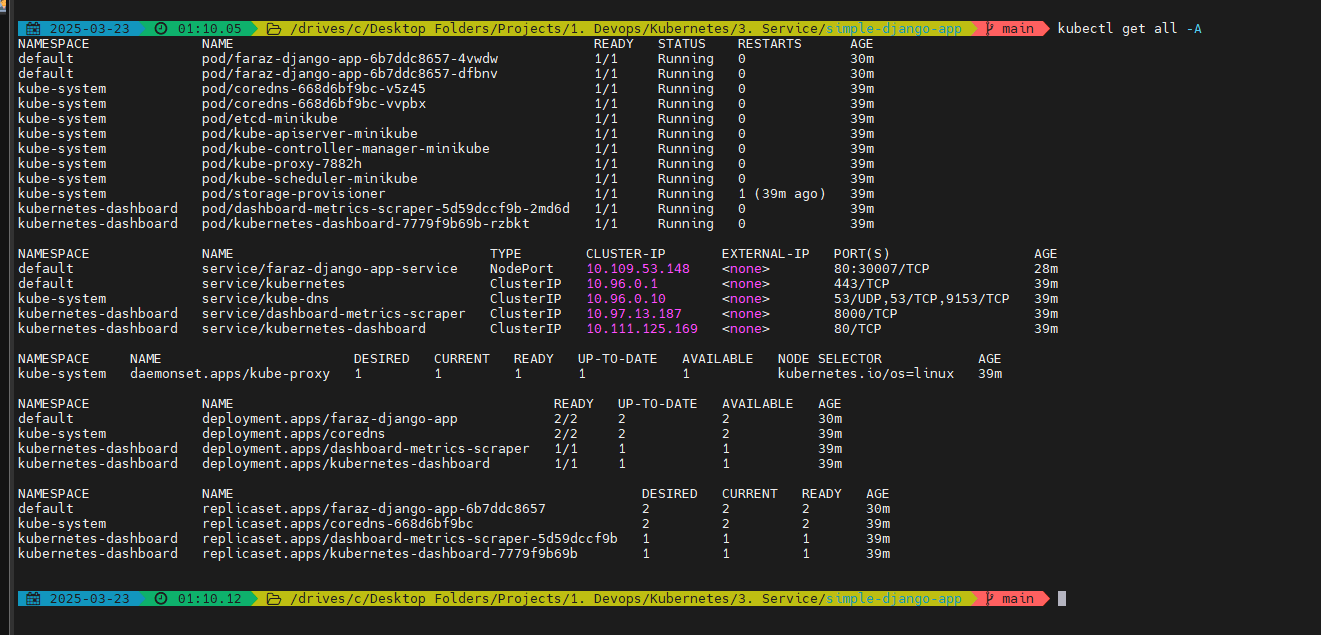
**How to Access the NodePort**

**Enable Host-Machine Networking:**

You'd need to configure Minikube with a driver that supports seamless host-to-cluster communication, such as --driver=hyperv or --driver=virtualbox (if available on your system).

Alternatively, you can set up port-forwarding on Docker to enable traffic between the VM and your host machine, but this can be complex.

**Stick with Port-Forwarding:** Since kubectl port-forward works and gives you access via localhost, it's often the easiest and most reliable solution, especially for local development.



**containerPort (8000):**

This is the port exposed inside the container where your application is running.

In your deployment file (deployment.yml), the containerPort: 8000 tells Kubernetes that the application within the container listens on port 8000.

It's internal to the container and not accessible directly from outside unless explicitly exposed through a Service.

**targetPort (8000):**

This is the port inside the pod that the Service routes traffic to.

In your service.yml, the targetPort: 8000 tells Kubernetes to forward incoming traffic to port 8000 of the container (which corresponds to the containerPort).

Think of it as the Service's endpoint within the Pod.

**port (80):**

This is the port exposed by the Service for external communication within the cluster.

When other applications or services in the cluster want to communicate with your app, they do so via this port. In this case, the Service listens on port 80.

**nodePort (30007):**

This is the external port on the cluster's node (e.g., your Minikube node) that maps to the Service's port.

It allows access to your application from outside the cluster. For example, if your Node has an IP address (192.168.49.2), accessing 192.168.49.2:30007 will forward traffic to the Service (port: 80) and then to the application (targetPort: 8000).

**Significance of Each Port**

**containerPort:** Internal container-level port where your app is running.

**targetPort:** Connects the Service to the Pod's internal container application.

**port:** The entry point for traffic within the Kubernetes cluster.

**nodePort:** Allows access to the application from outside the Kubernetes cluster.

**How They Work Together**

**Here’s the flow:**

Traffic arrives at nodePort (30007 on 192.168.49.2).

The Service listens on port (80) and forwards the traffic to the appropriate targetPort (8000).

The application receives the traffic on containerPort (8000).

This layered approach provides flexibility, scalability, and security for managing traffic flow.

**When you use kubectl port-forward, it creates a direct connection between your local machine and the Kubernetes Service, bypassing the complexities of cluster networking.**

Local Port Binding:

The kubectl port-forward command binds a local port on your machine (in this case, 8000) and redirects any traffic received on that port to the Kubernetes Service.

Connection to the Service:

The Service in Kubernetes listens on its defined port (80 in your case). The port-forwarding mechanism intercepts traffic on localhost:8000 and forwards it to the Service's port: 80 within the cluster.

Service Routing:

The Service then routes the traffic to its associated Pods via the targetPort (8000). This is the port where your application is running inside the container, as defined in the deployment.

Application Receives Traffic:

The application running on the container (bound to containerPort: 8000) receives the forwarded traffic, processes it, and responds. The response follows the same path back to your local machine.

Flow Example

If you run:

kubectl port-forward service/faraz-django-app-service 8000:80

Your browser or curl: Sends a request to http://localhost:8000.

Port-forwarding: Redirects the request to the Service on port: 80 in the cluster.

Service: Forwards the request to the targetPort: 8000 of the Pods associated with it.

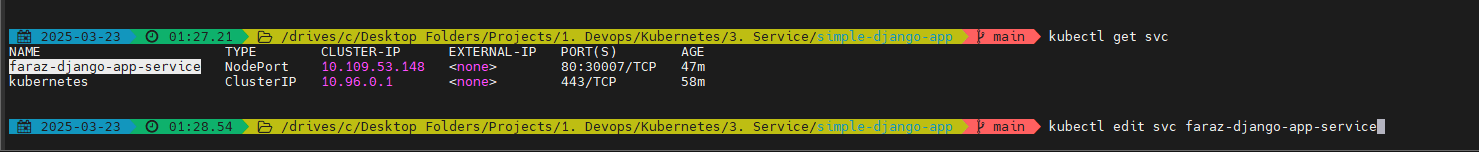
Pod: The application running inside the container (listening on containerPort: 8000) processes the request and sends a response.

Why It Works Without Networking Issues

Unlike NodePort, kubectl port-forward directly tunnels the traffic from your local machine to the cluster, bypassing networking isolation or firewall restrictions. This is why you can access your application via localhost:8000 even when 192.168.49.2:30007 isn't accessible.

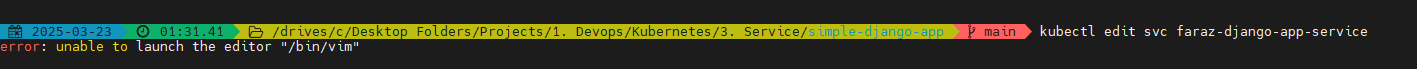
Now if we want to edit to Loadbalancer type of service:

* kubectl edit svc faraz-django-app-service

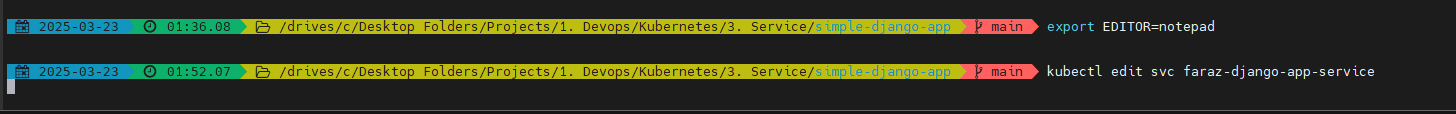


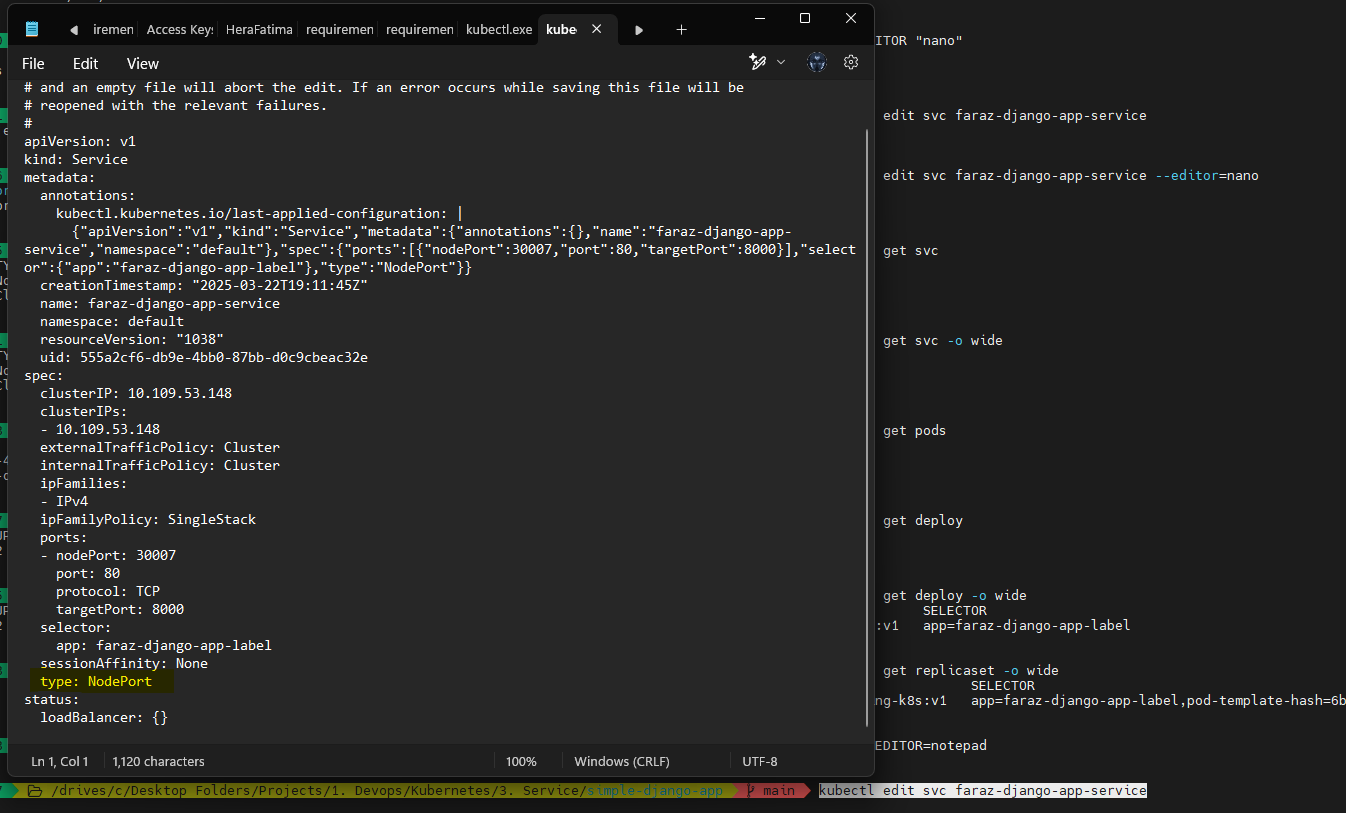
And change the type: Loadbalancer

If facing errors like:

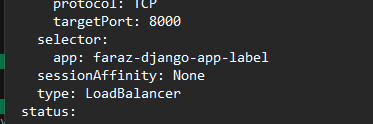


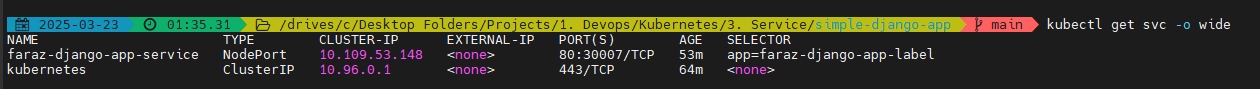
* export EDITOR=notepad
* kubectl edit svc faraz-django-app-service



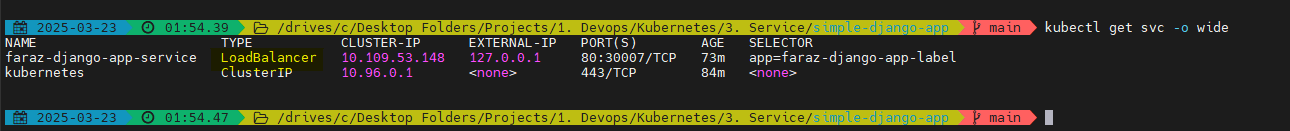


Change it to:



Earlier: 

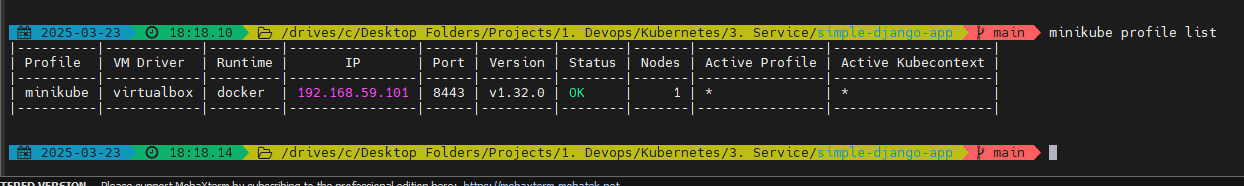
Now:

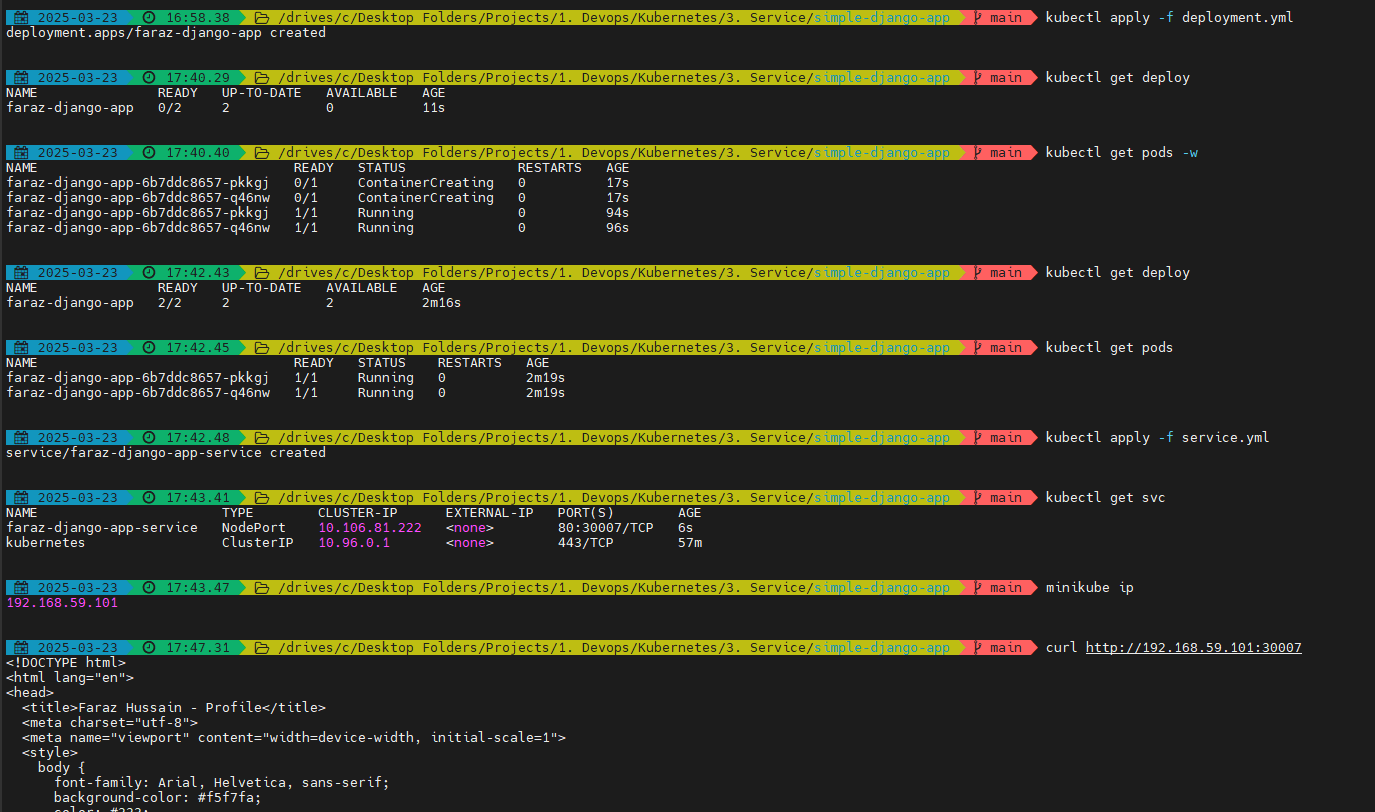


If we are on EC2, the Cloud Control manager will generate External IP for you

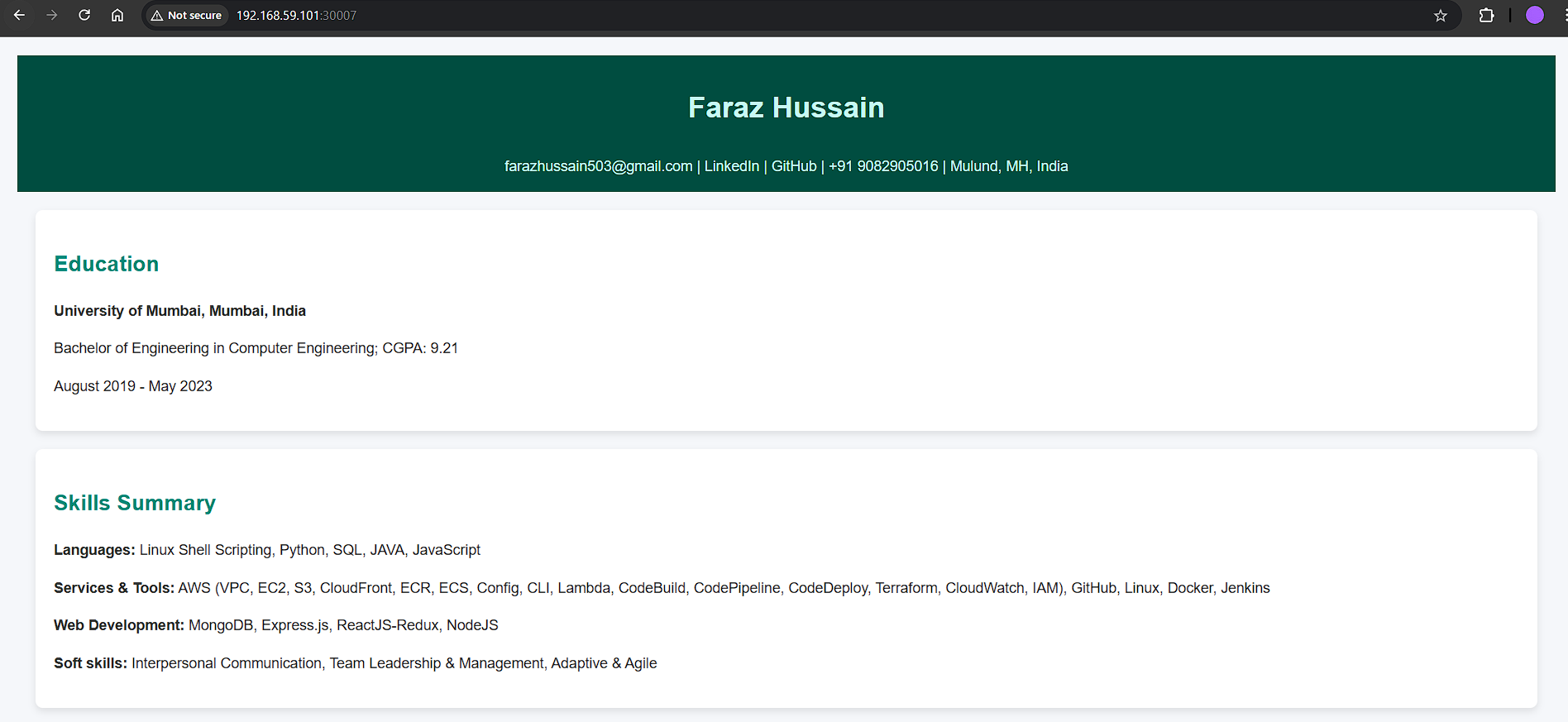
For exposing in minikube using metallb: <https://kubebyexample.com/learning-paths/metallb/install>

If using virtualbox driver:



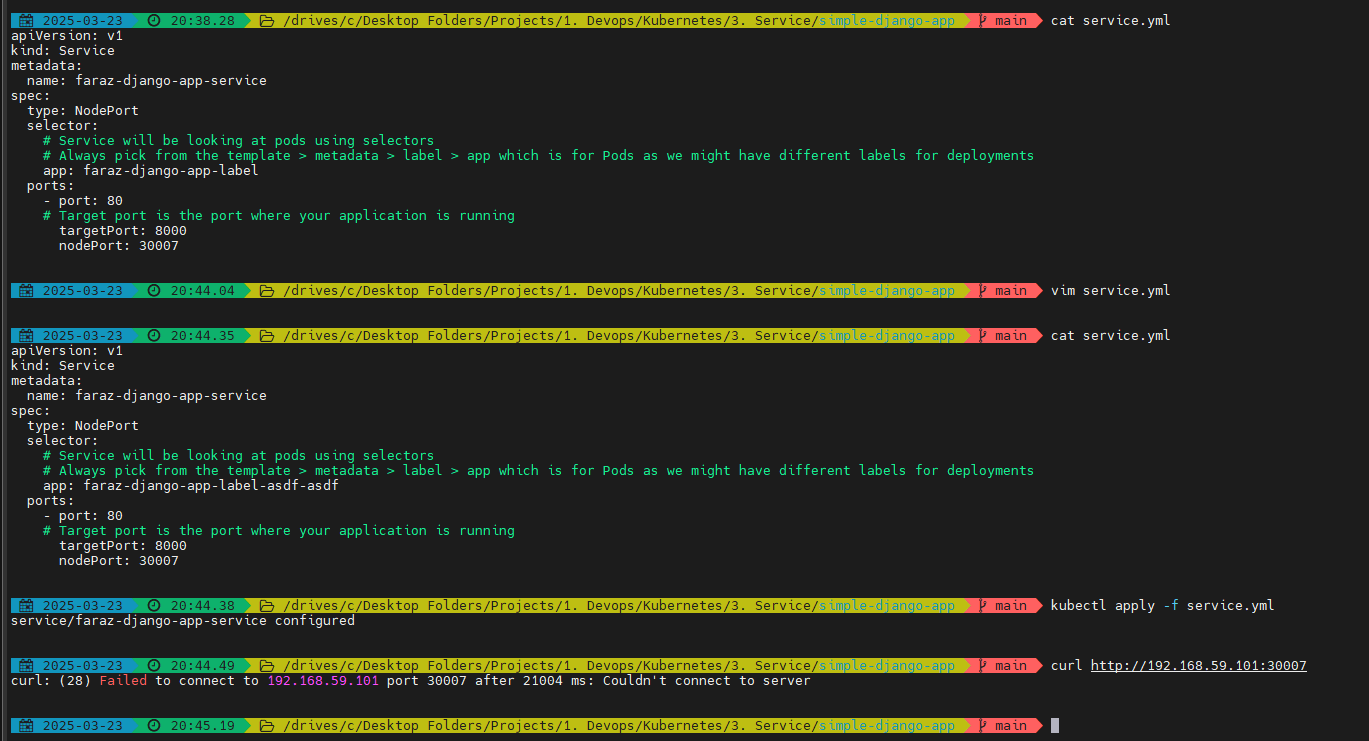


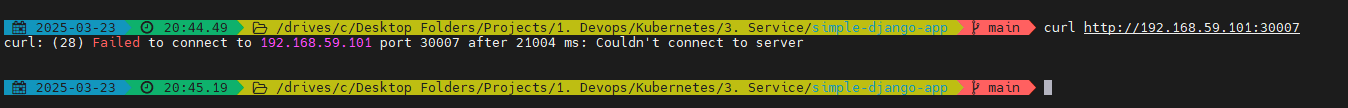
* curl <http://192.168.59.101:30007>

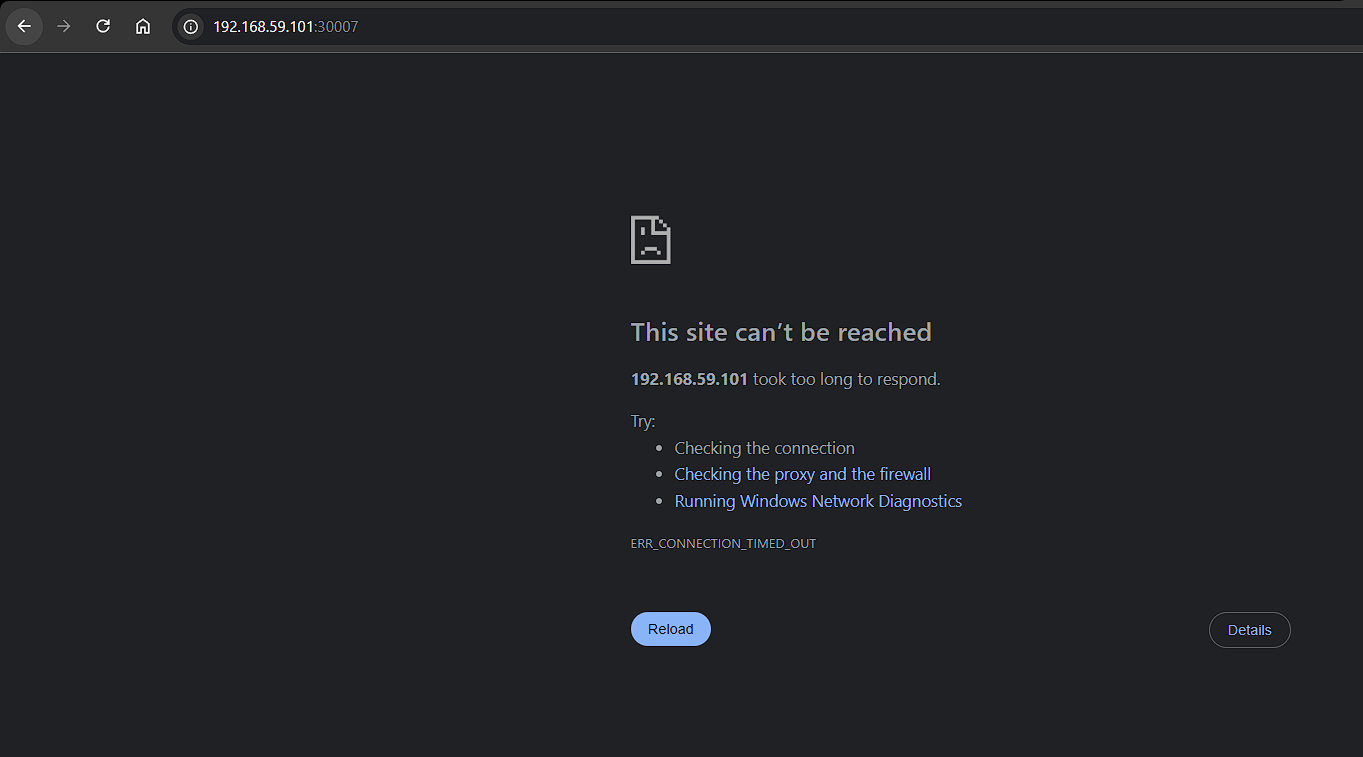


To test service discovery:

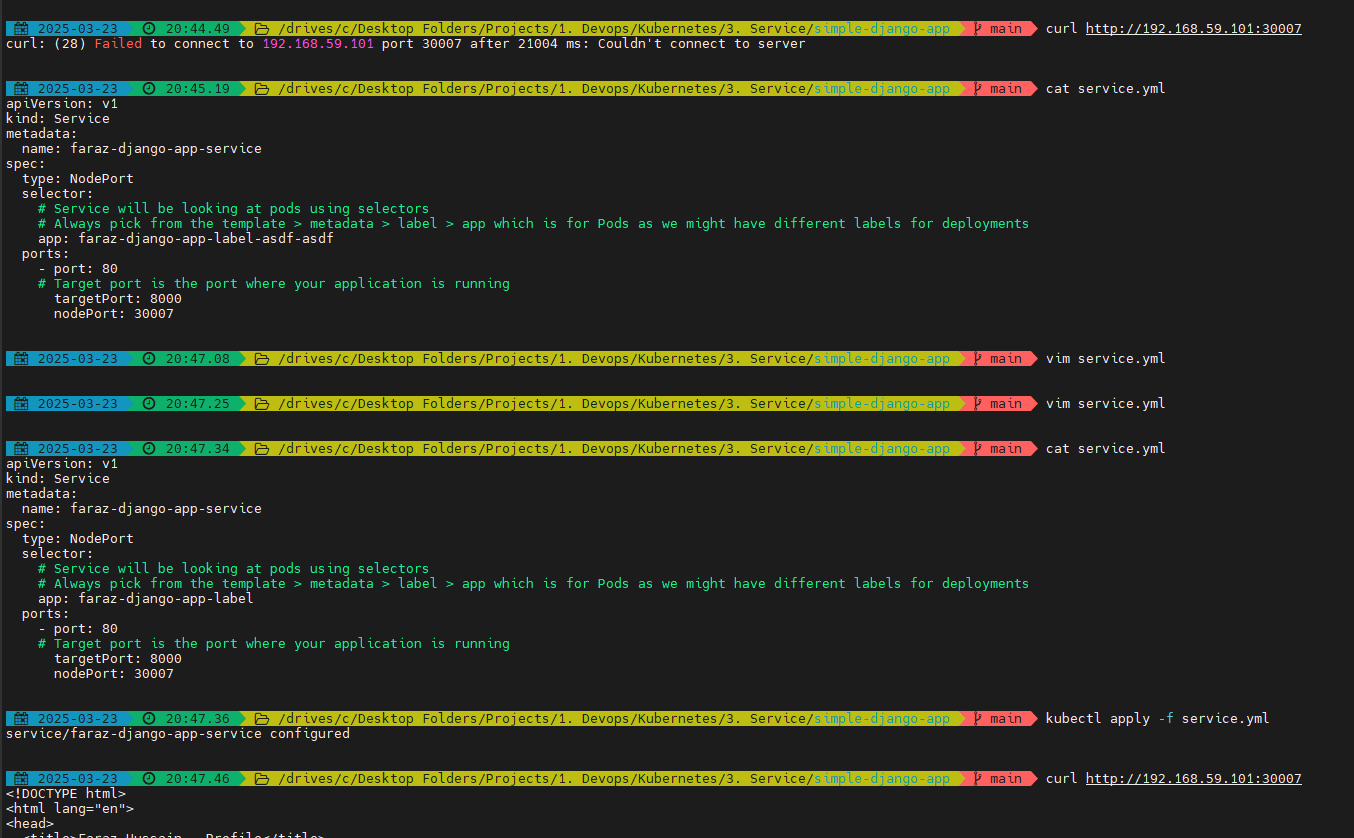
Make changes to your service selector:

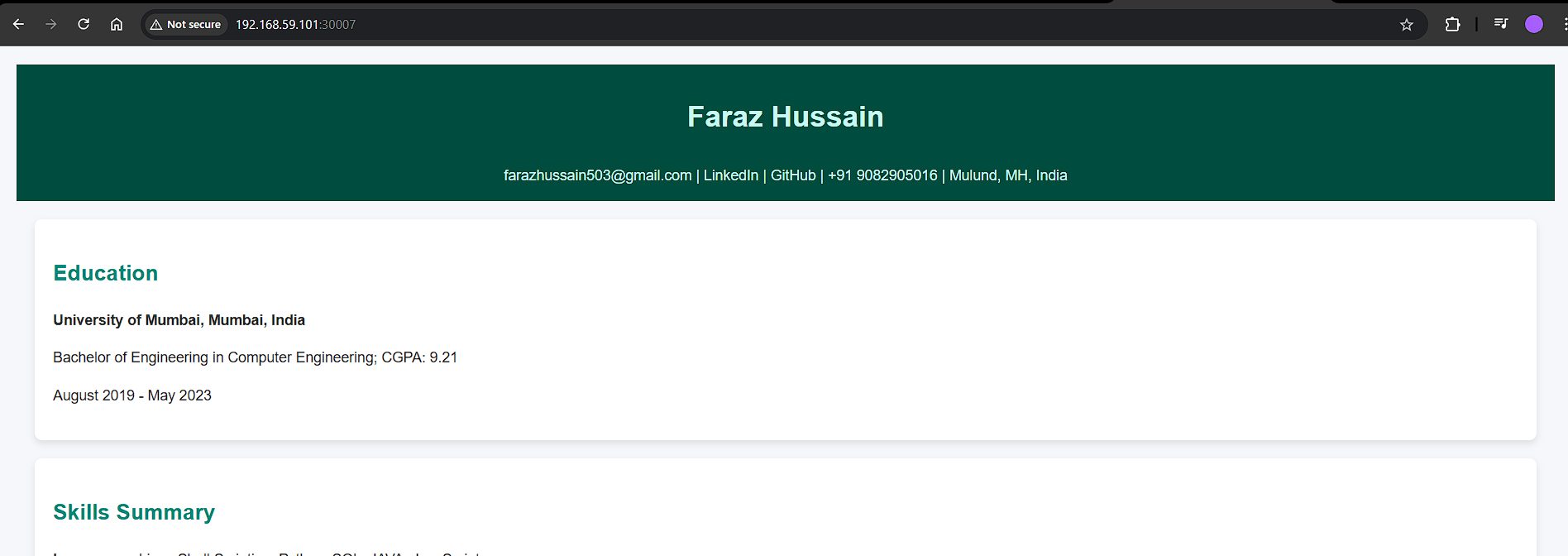






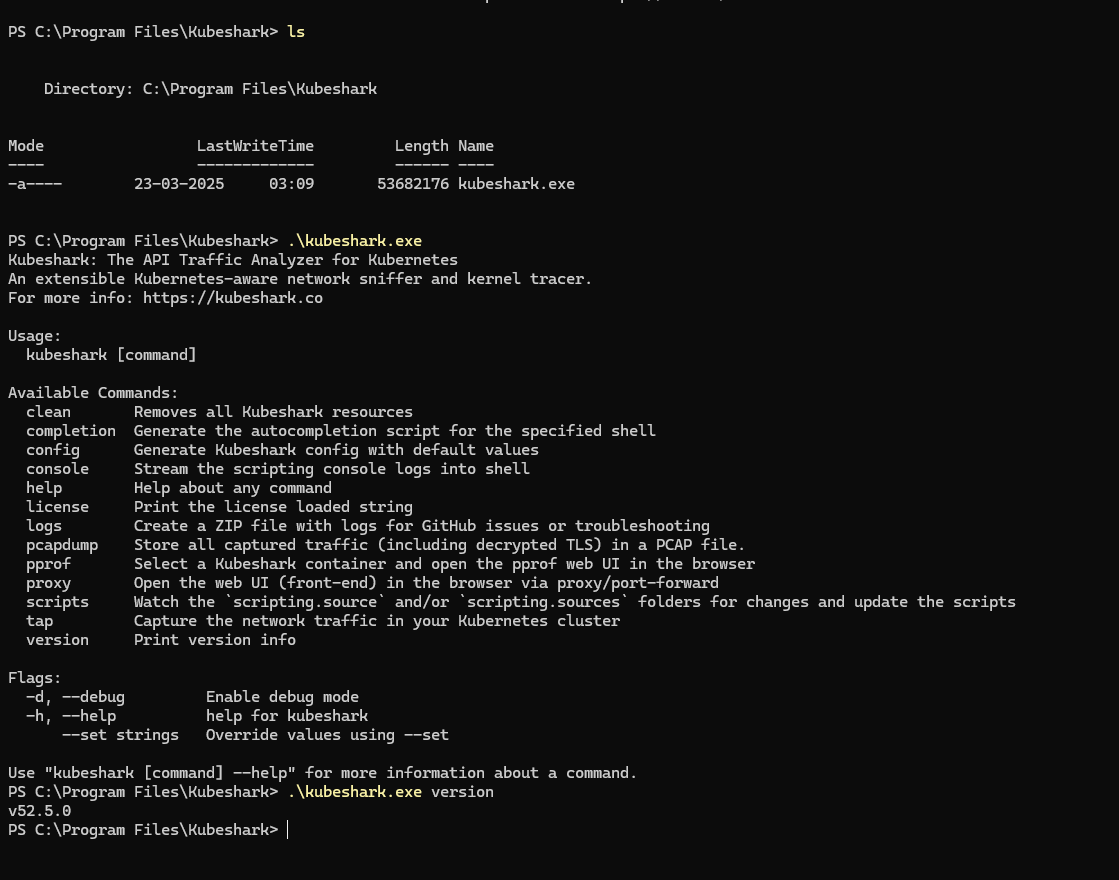
Now apply the proper selector:



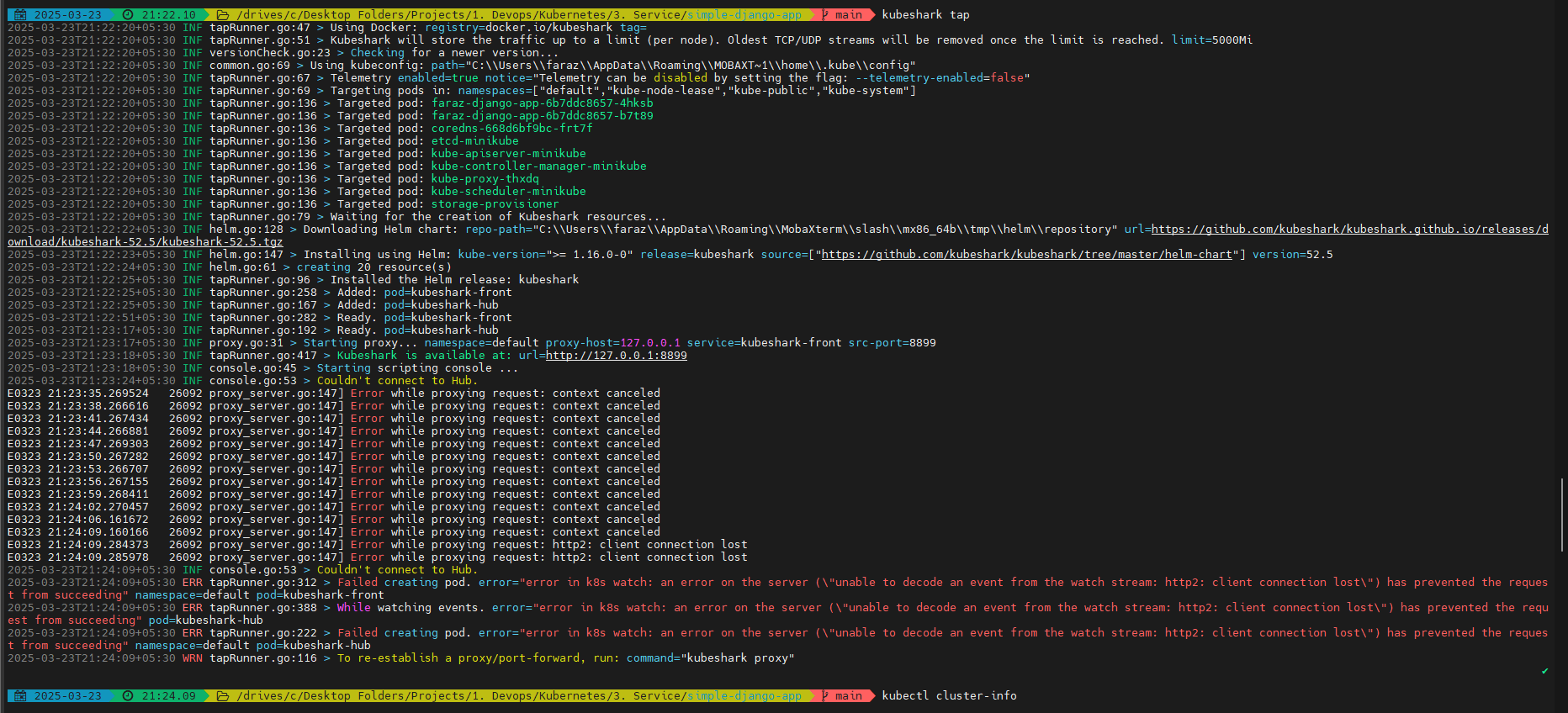


Install Kubeshark:

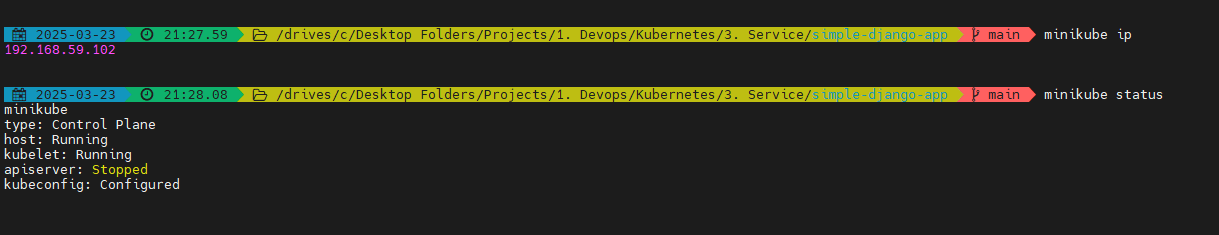
curl -LO <https://github.com/kubeshark/kubeshark/releases/download/v52.5.0/kubeshark.exe>



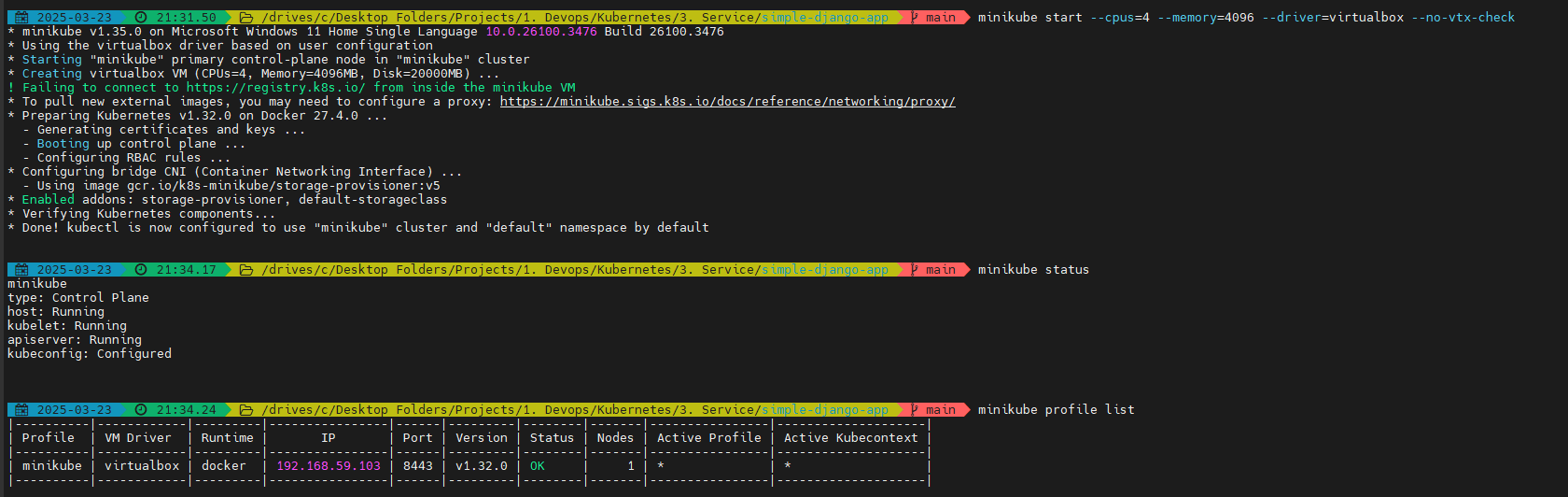
Ran into these issues on virtual box:

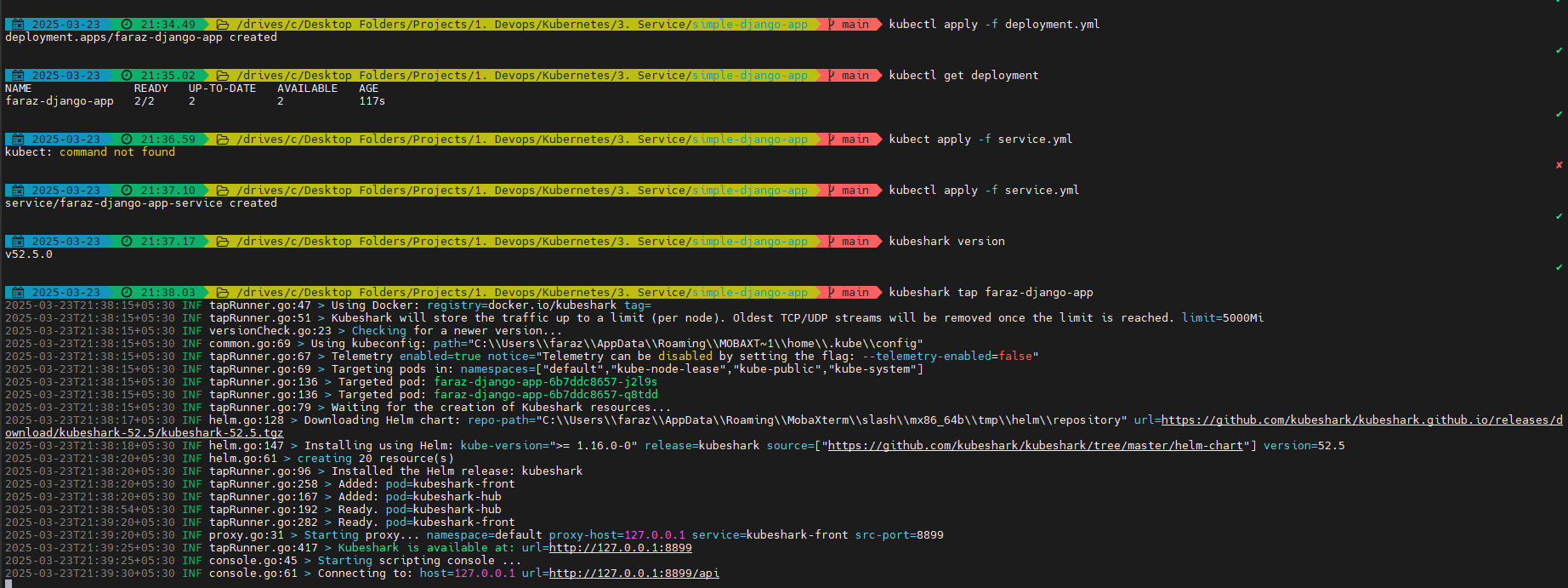


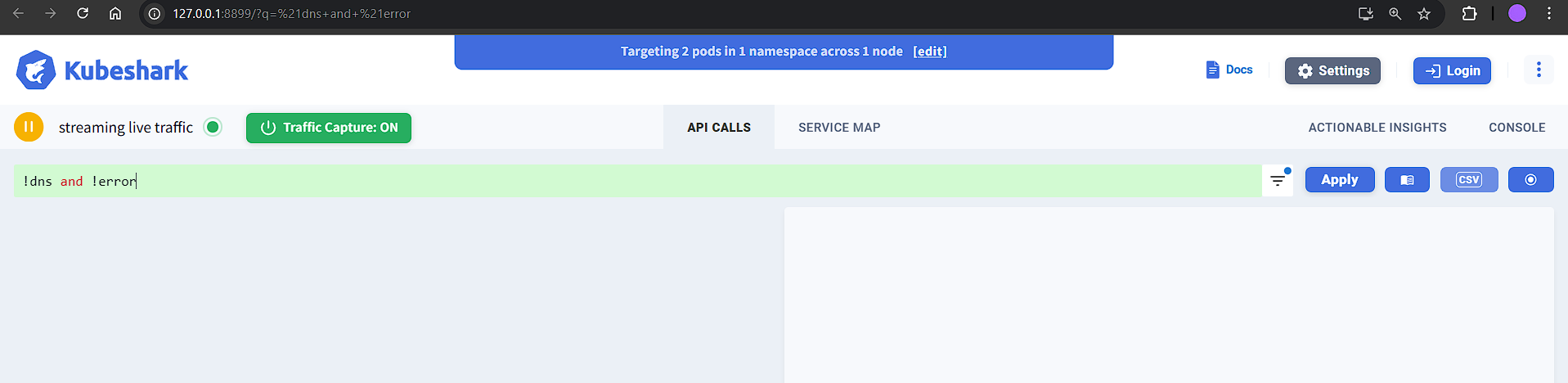
The API server was stopping several times:



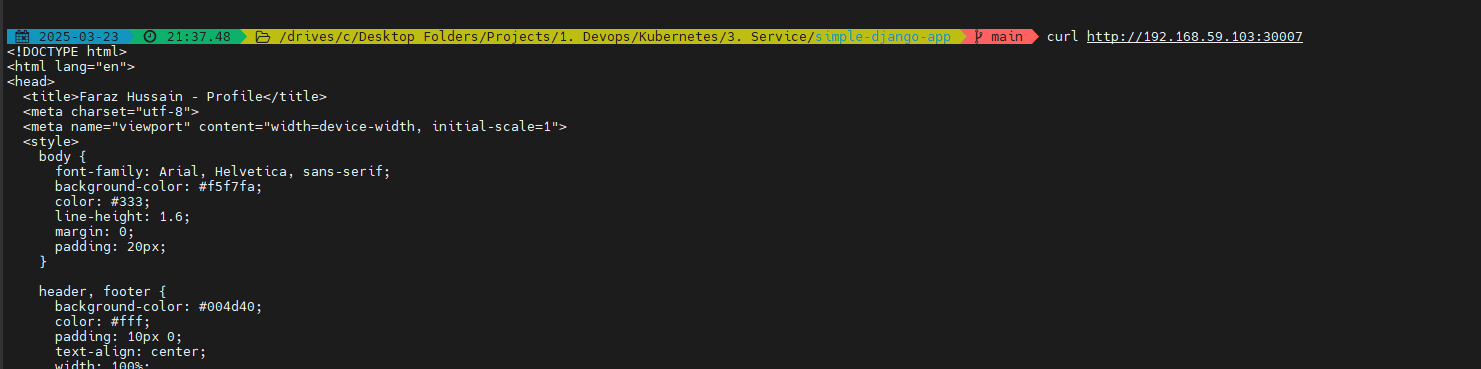
* minikube delete
* minikube start --cpus=4 --memory=4096 --driver=virtualbox --no-vtx-check



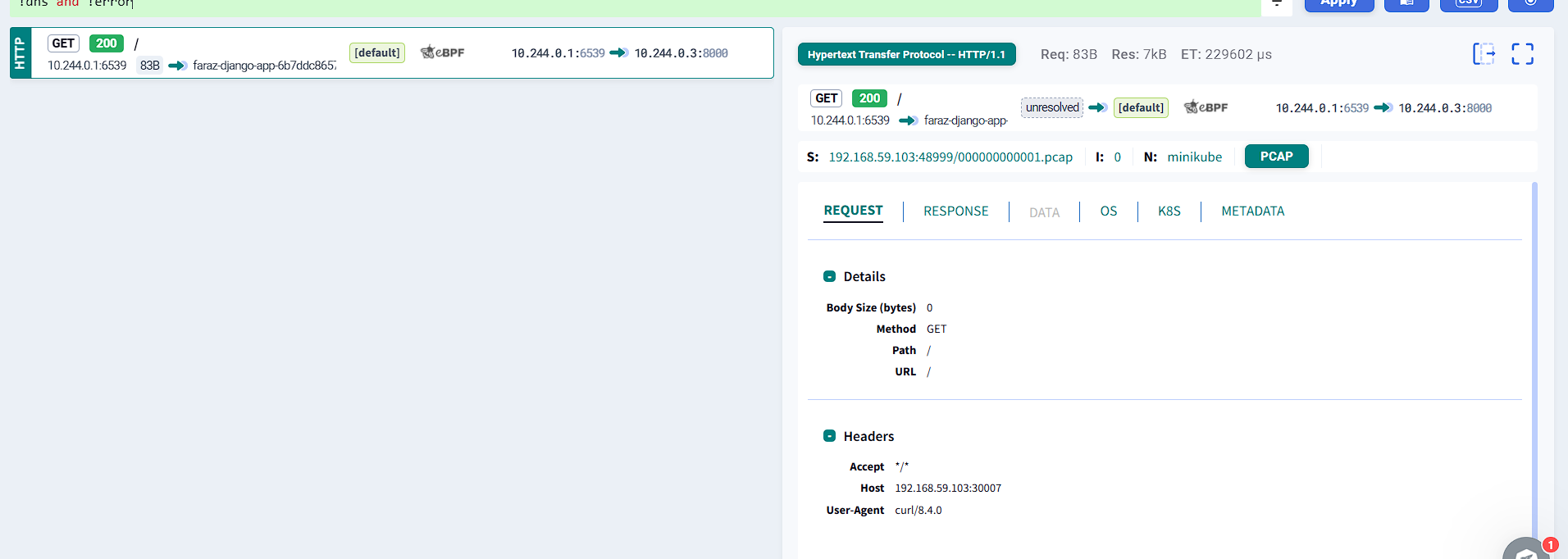




Now if I do:

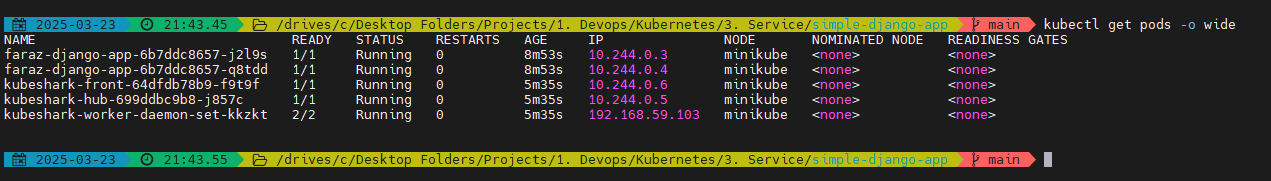


I will be able to see:



We can see that the traffic was routed to:

* faraz-django-app-6b7ddc8657-j2l9s -> 10.244.0.3:8000



The second request was routed to:

* faraz-django-app-6b7ddc8657-q8tdd -> 10.244.0.4:8000

