**CP3061 – DevOps and Microservices**

**Mini Project**

**Infrastructure as Code Using Terraform and Kubernetes**

**Done by:**

Gowtham A (2024207036)

M.E CSE

1. **Abstract**

In the rapidly evolving landscape of cloud computing and DevOps, the ability to automate infrastructure provisioning and application deployment has become a cornerstone of modern software development. This project exemplifies the principles of Infrastructure as Code (IaC) by leveraging Terraform, Kubernetes, and Docker to deploy a scalable, resilient, and fully automated full-stack web application. The system comprises a frontend built with HTML, CSS, and JavaScript, hosted on an Nginx server, and a backend REST API developed using Node.js and Express.js to manage a dynamic product catalog.

By codifying infrastructure requirements into declarative Terraform configurations, this project eliminates manual intervention, ensuring reproducibility across environments. Kubernetes orchestrates containerized workloads, enabling self-healing, load balancing, and horizontal scaling, while Docker guarantees environment consistency by packaging applications into portable containers. The integration of these tools demonstrates a seamless transition from development to production, mimicking real-world DevOps workflows.

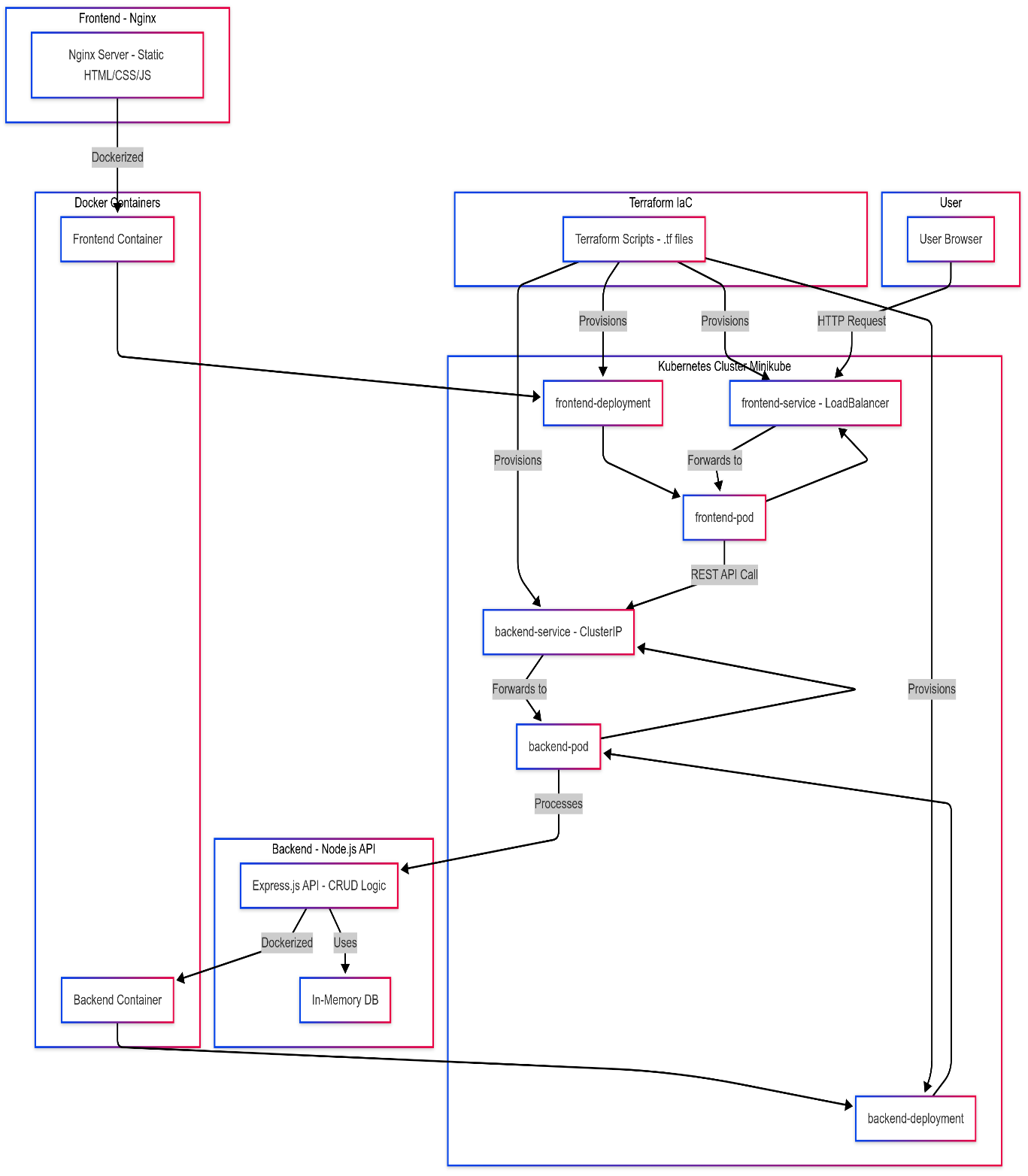
This work not only highlights the technical implementation of IaC but also underscores its broader implications: reducing human error, accelerating deployment cycles, and fostering collaboration between development and operations teams. By bridging the gap between theoretical concepts and practical application, the project serves as a pedagogical tool for understanding cloud-native architectures and the transformative potential of automation in modern software engineering.

1. **Introduction**

The rise of cloud computing and microservices has revolutionized how applications are developed, deployed, and scaled. However, traditional infrastructure management—reliant on manual configuration and ad-hoc scripting—remains fraught with challenges, including inconsistency across environments, scalability bottlenecks, and deployment failures. These inefficiencies are exacerbated in complex systems where even minor misconfigurations can lead to downtime or security vulnerabilities.

Infrastructure as Code (IaC) emerges as a paradigm shift, enabling developers to define and manage infrastructure using machine-readable configuration files. This approach ensures version control, auditability, and repeatability, aligning infrastructure management with software development best practices. This project embodies these principles by automating the deployment of a full-stack application using Terraform for infrastructure provisioning, Kubernetes for orchestration, and Docker for containerization.

1. **Architecture Diagram**



1. **Requirements and Supported Systems**

**4.1 Hardware Requirements**

* CPU: 2+ cores (supports virtualization).
* RAM: 4+ GB.
* Storage: 20+ GB.

**4.2 Software Requirements**

* Terraform - Version 1.5+
* Kubernetes Cluster- Version 1.27+
* Docker -Version 24.0+
* Node.js -Version 18.0+
* Minikube -Version 1.35+

**4.3 Supported Systems**

* Operating Systems: Windows 10+, Linux (Ubuntu 22.04+), macOS Ventura+.
* Kubernetes Distributions: Minikube, Docker Desktop, AWS EKS, Google GKE.

1. **Tools Used** 
   1. **Software Tools**

|  |  |
| --- | --- |
| **Tool** | **Role** |
| Terraform | IaC for Kubernetes resource provisioning. |
| Kubernetes | Orchestrates containerized workloads. |
| Docker | Builds and manages application containers. |
| Node.js | Backend API runtime. |
| Nginx | Frontend web server. |
| Minikube | Local Kubernetes cluster |

* 1. **Hardware**
* Local Machine: Intel i5/Ryzen 5 or equivalent.
* Cloud VM (Optional): AWS EC2, Google Compute Engine.
* Storage (SSD) for optimal Minikube performance.

1. **Implementation**

**Step 1:** Set Up Kubernetes Cluster

1. **Start Minikube**
   * minikube start --driver=docker
2. **Verify Cluster**
   * kubectl get nodes

**Step 2:** Build and deploy applications

1. **Build Docker Images**

* Use Minikube’s Docker daemon

minikube docker-env | Invoke-Expression

* Frontend

docker build -t frontend:latest ./frontend

* Backend

docker build -t backend:latest ./backend

1. **Apply Terraform Configurations**

cd terraform

terraform init

terraform apply -auto-approve

**Step 3:** Configure Networking

* 1. **Verify Services**

kubectl get svc

Ensure frontend-service (LoadBalancer) and backend-service (ClusterIP) are running

* 1. **Access Frontend**

minikube service frontend-service --url

**Step 4:** Validate Deployment

1. **Check Pods:**

kubectl get pods -l app=frontend

kubectl get pods -l app=backend

1. **Test API Connectivity:**

# Forward backend port

kubectl port-forward svc/backend-service 3000:3000

1. **Input and Output**
   1. **Input**

* Terraform Files:
  + main.tf (Kubernetes deployments/services definitions).
* Dockerfiles:
  + frontend/Dockerfile, backend/Dockerfile.
* Application Code:
  + Frontend: index.html, script.js, style.css.
  + Backend: index.js, package.json.
  1. **Output**
* **Deployed Application**:

minikube service frontend-service --url

Output: http://127.0.0.1:30734

* **Backend API Test**:

curl http://localhost:3000/products

Output: [{"id":1,"name":"Laptop","price":999}]

* **Self-Healing Test:**

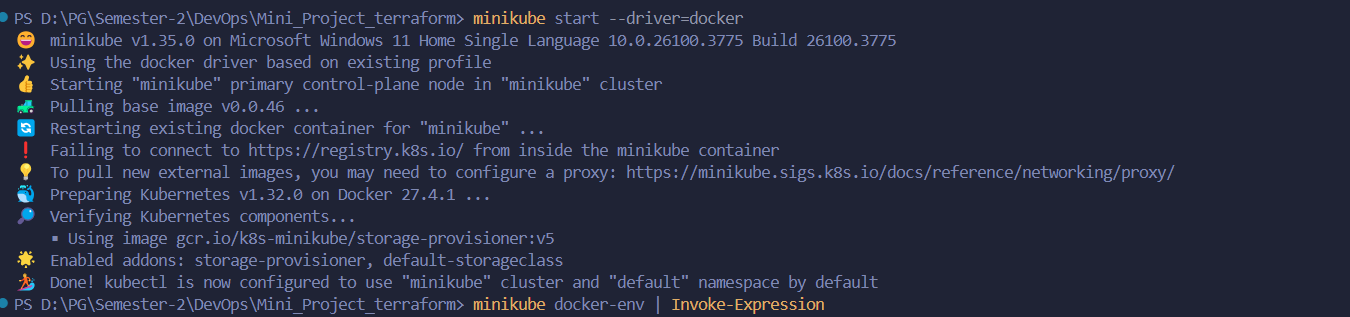
kubectl delete pod frontend-75849fb77b-rgkt9

kubectl get pods –watch

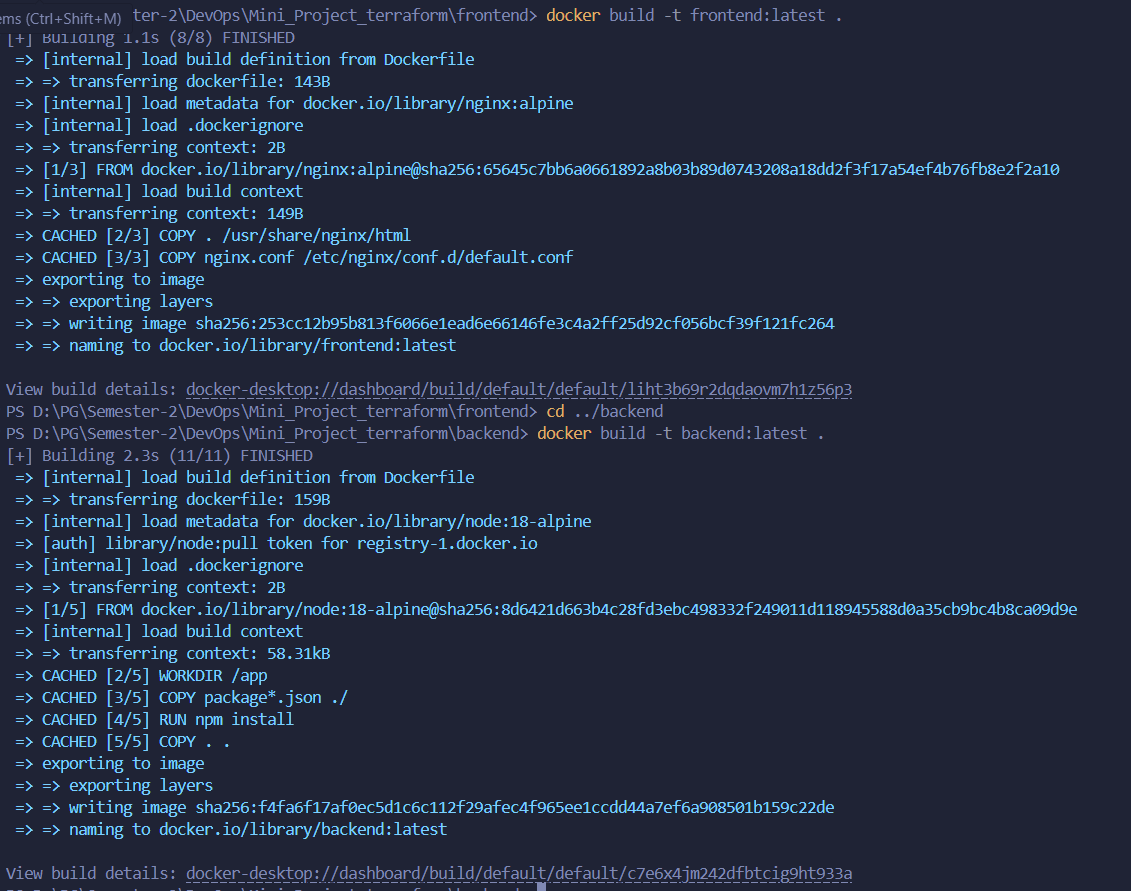
Deleting the frontend pod to see whether it is recreating the pod in case of a system crash or network error.

**Screenshots:**

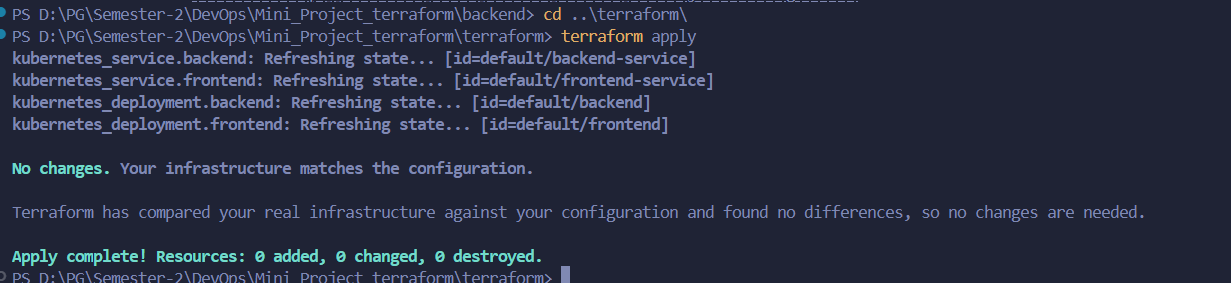
* Starting and initializing the created MiniKube cluster:



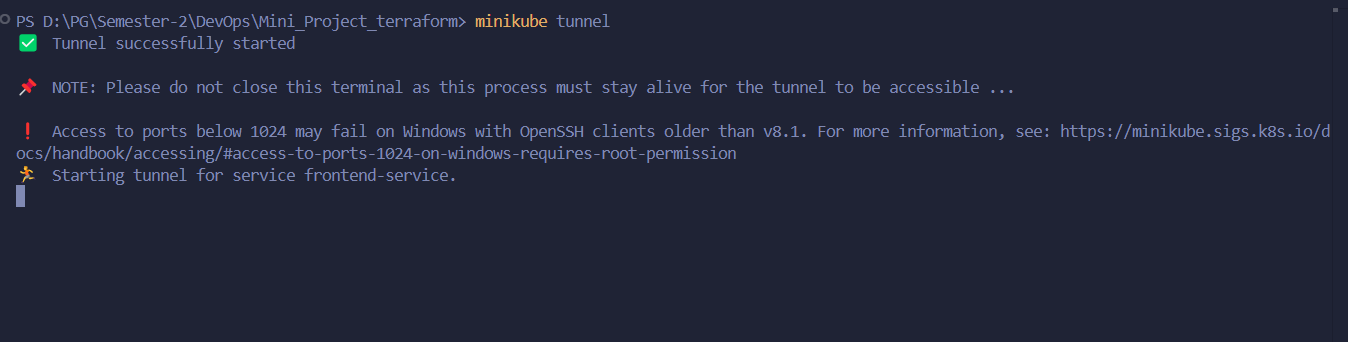
* Updation of recent code into the Dockerfile



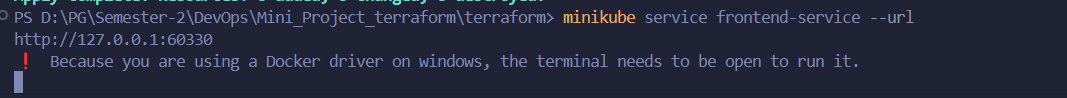
* Terraform Apply



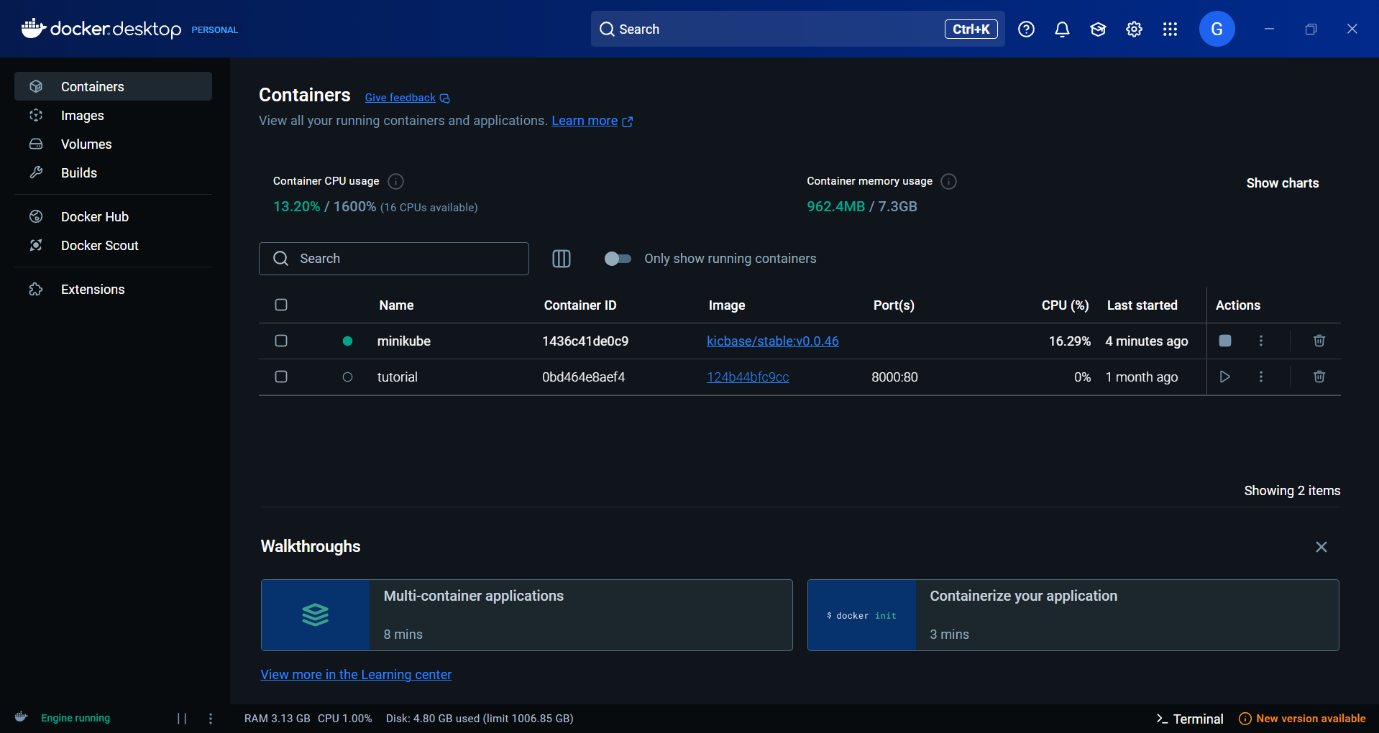
* Starting MiniKube tunnel



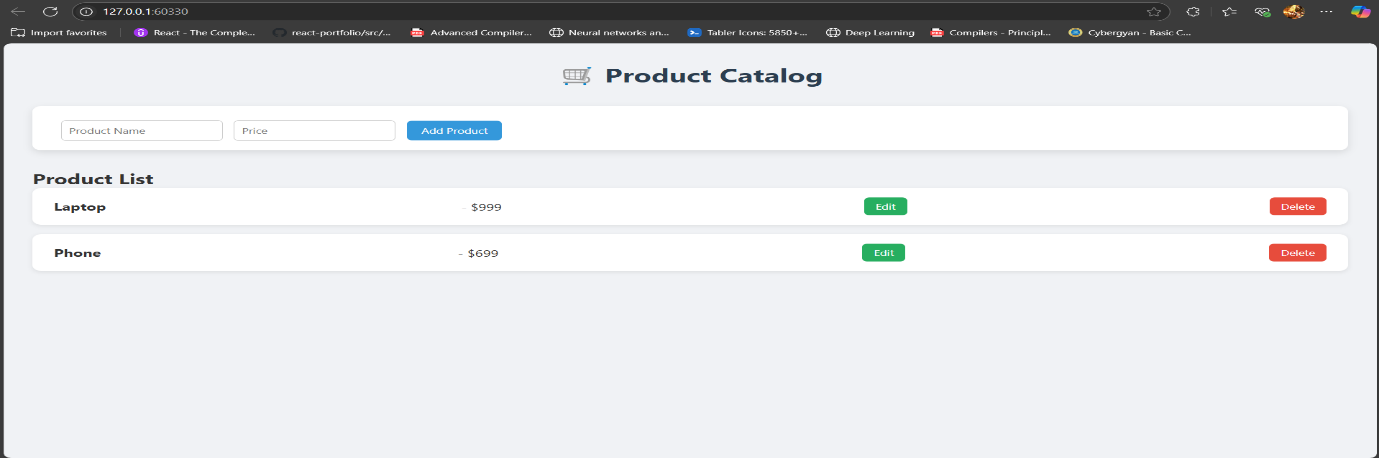
* The web app service



* Minikube instance running on the Docker Desktop



* Web App – Product Catalog -Performs CRUD



1. **Applications**
2. **Automated Deployment of Scalable Applications**

* Use case: Companies can deploy applications across multiple environments (dev, staging, production) with just a few commands.
* Example: E-commerce platforms can auto-scale their services during high traffic (like festive sales) without manual intervention.

1. **Multi-Cloud and Hybrid Cloud Management**

* Use case: Organizations using AWS, Azure, or GCP can manage resources from all providers using a single Terraform codebase.
* Example: A company can deploy its frontend on Azure and backend on AWS, all orchestrated via Kubernetes.

1. **Disaster Recovery and Infrastructure Replication**

* Use case: Quickly replicate infrastructure in a different region or cloud in case of failure or for backup.
* Example: Banking and financial services can restore full infrastructure in a different region to maintain businesscontinuity.

1. **Resource Efficiency and Cost Optimization**

* Use case: Kubernetes autoscaling ensures resources are used only when needed, and Terraform helps destroy unused resources to cut cloud bills.
* Example: Streaming platforms scale up during peak hours and scale down during off-hours to save cloud costs.

1. **References**
2. HashiCorp. (2023). *Terraform Documentation*.  
   <https://developer.hashicorp.com/terraform/docs>
3. Kubernetes. (2023). *Kubernetes Documentation*.  
   <https://kubernetes.io/docs>
4. Docker. (2023). *Docker Documentation*.  
   <https://docs.docker.com>
5. Minikube. (2023). *Getting Started*.  
   <https://minikube.sigs.k8s.io/docs/start/>