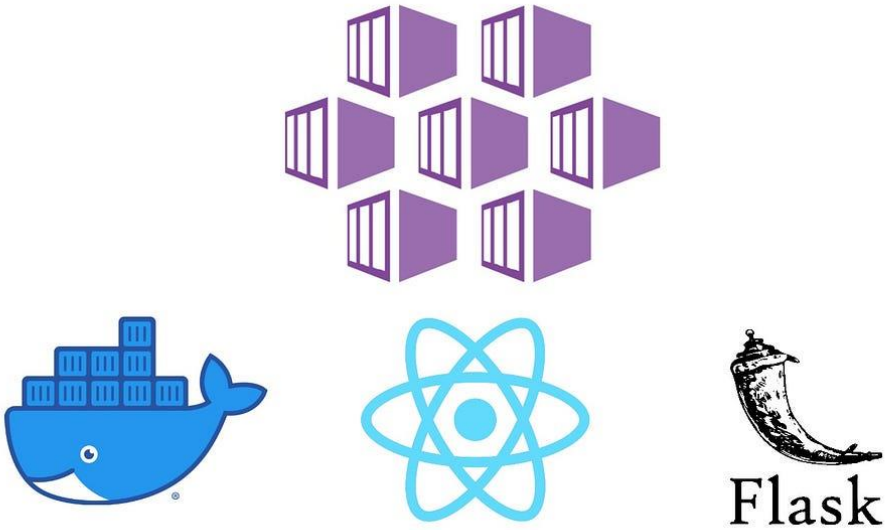
Deploy a Full Stack Web App to Azure Kubernetes Service with Dockerhub Images

- **Ismail Shaikh (Senior Cloud Engineer)**



Introduction

In this blog i will walk you through the process of getting a full-stack application up and running on AKS. Our sample calculator app has a separated React frontend and Flask backend. Both of them are built through **Docker** and pushed to Docker Hub but first we will start with basic understanding

What is Kubernetes?

Kubernetes is an open source container orchestrator that automates many tasks involved in deploying, managing, and scaling containerized applications.

What is Azure Kubernetes Service?

Azure Kubernetes Service is a managed container orchestration service based on the open source Kubernetes system, which is available on the Microsoft Azure public cloud. Using AKS simplifies the process of running Kubernetes on Azure without needing to install or maintain your own Kubernetes control plane. An organization can use AKS to handle critical functionality such as deploying, scaling and managing Docker containers and container-based applications.It provides a hosted Kubernetes cluster that you can deploy your microservices to.

Pre-Requisiste :

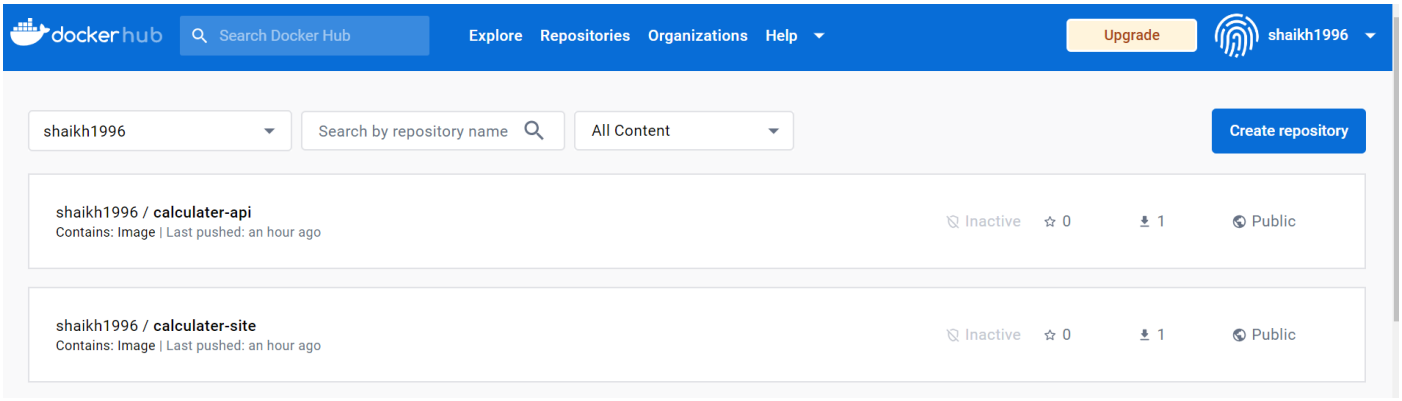
- Azure Kubernetes Service
- NGINX Ingress Controller
- Azure CLI

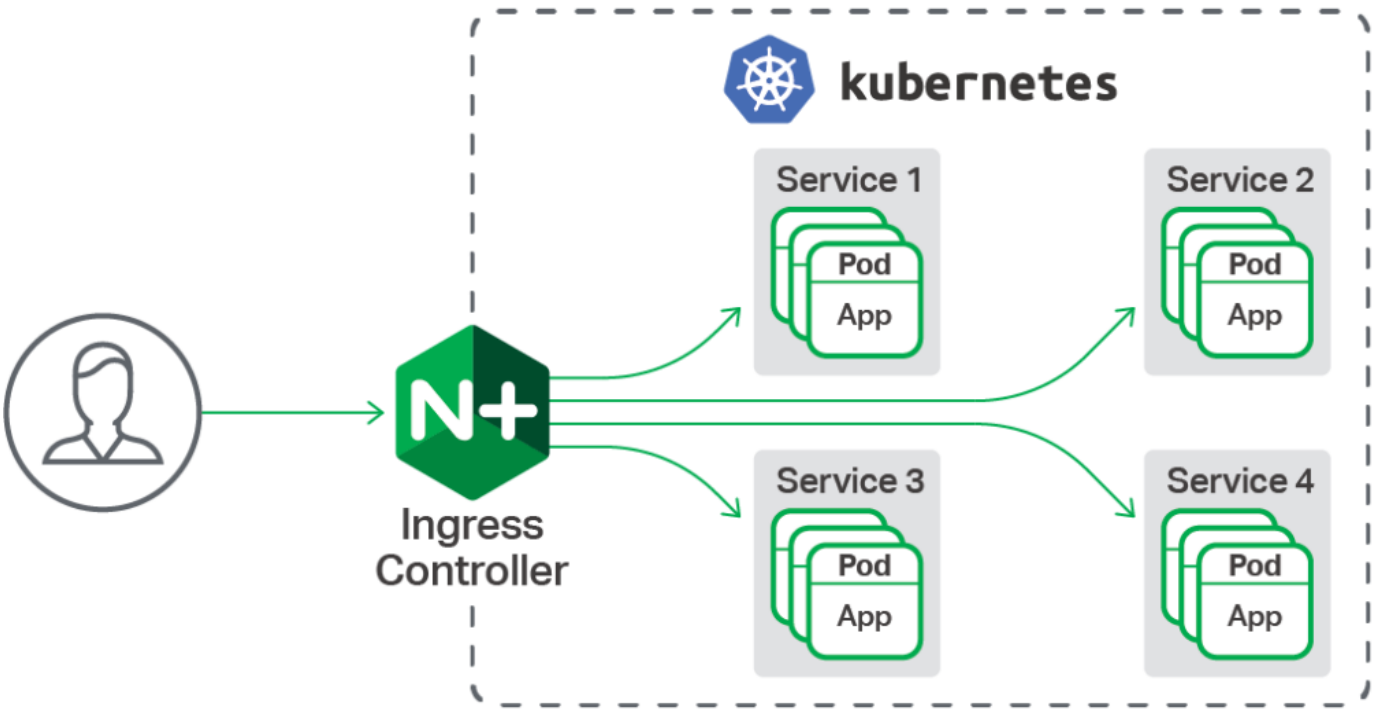
- Kubectl, Helm
- React frontend, Flask backend

Overview

We will create two Kubernetes deployments, one for the React frontend and the other for the Flask API. Two Kubernetes services will also be created for us to access the deployed application.

After having both frontend and backend running on AKS cluster, we will create an **ingress resource** to route traffic to each application. By using an ingress controller and ingress rules, a single IP address can be used to route traffic to multiple services in a Kubernetes cluster.

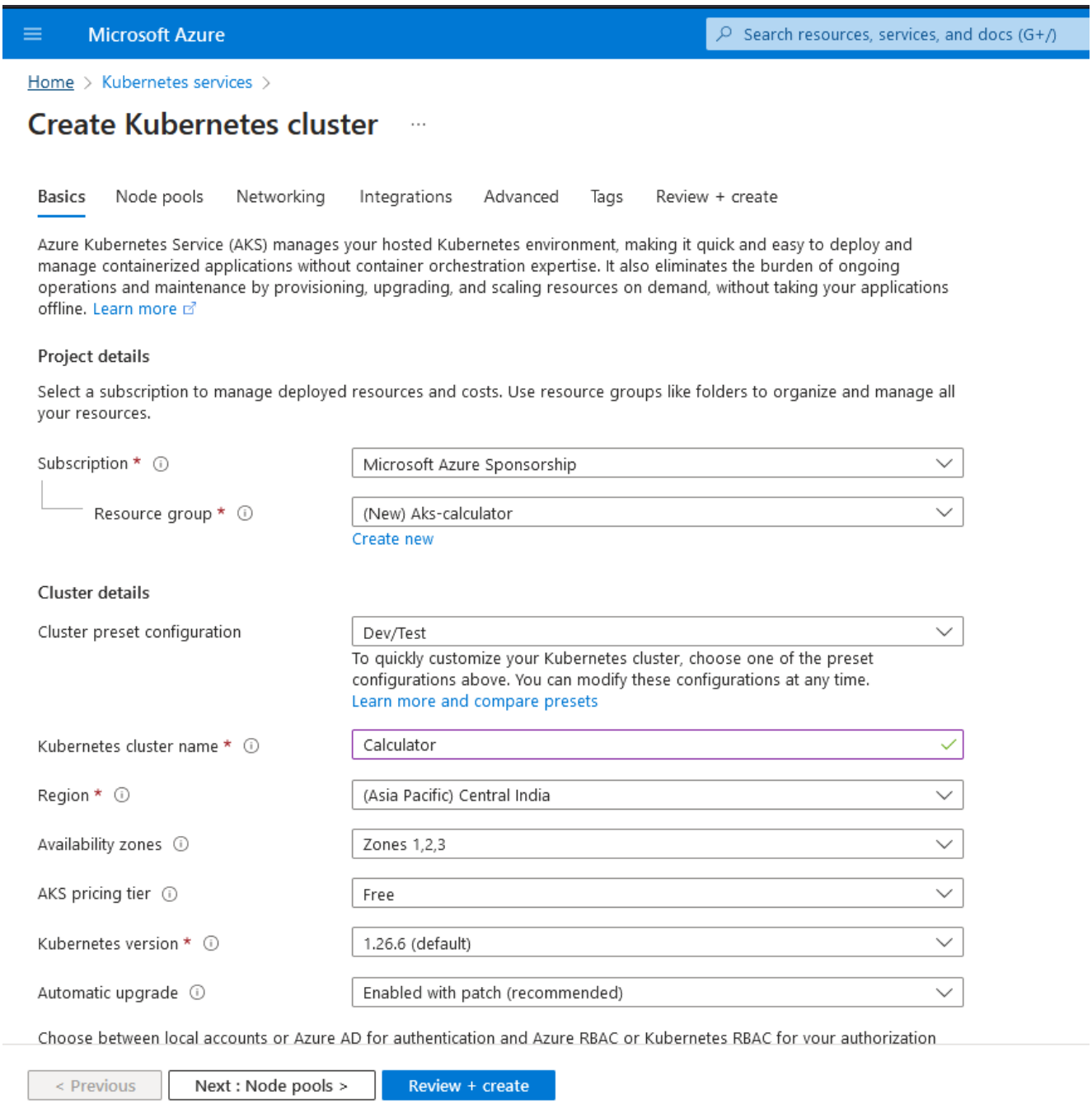




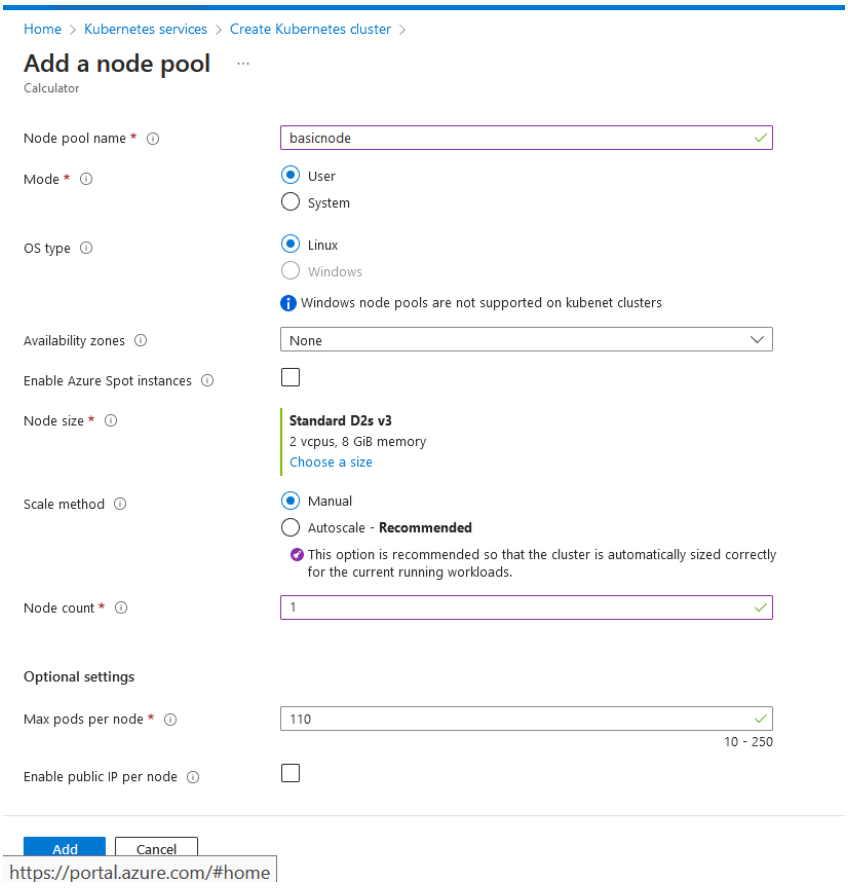
Getting Started

Step 1:- Create AKS Cluster

The first thing we have to do is create an AKS cluster. After creating a resource group in your preferred region, we can create an AKS cluster with a similar method. Personally, I like to create it with the UI, which is a pretty straightforward approach with the friendly **Azure Portal** Interface. Or you can also create it with Azure CLI following the Microsoft Docs.



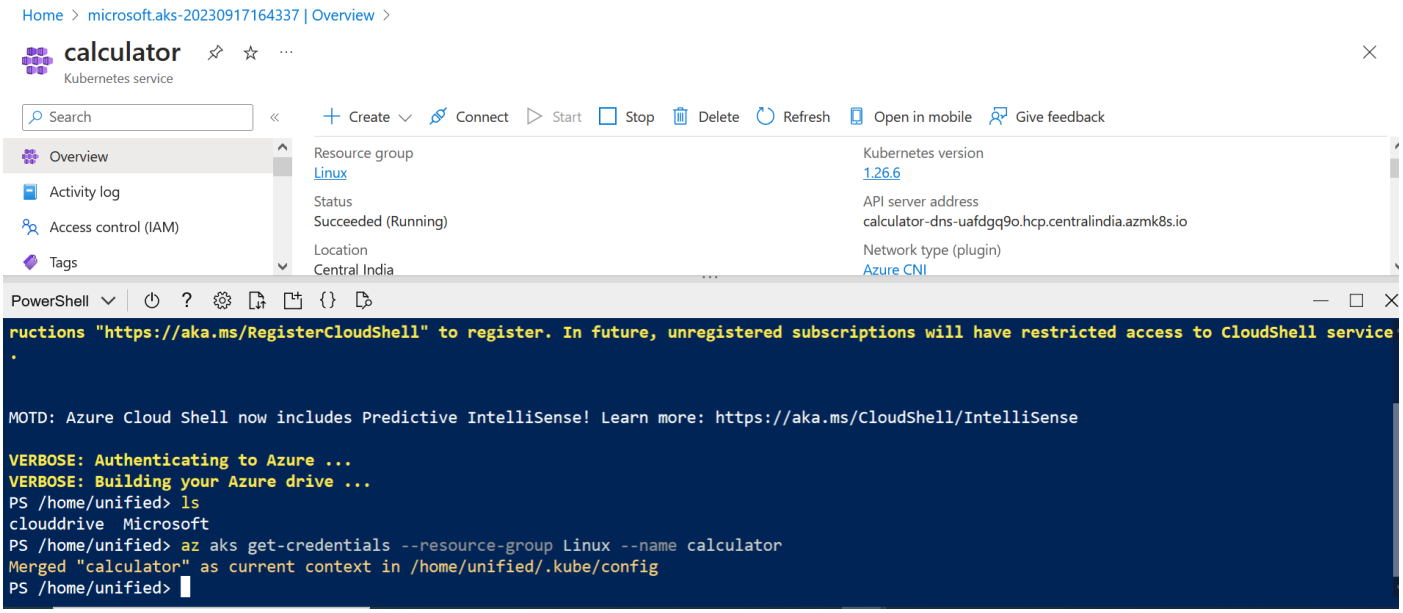
In the next step we will create out own customize node pole.



Step 2:- Connect to Cluster

We will use **kubectl** to manage the Kubernetes cluster. Run the command below in **Azure CLI** Powershell Mode to configure kubectl and connect to the cluster we previously created.

```
az aks get-credentials --resource-group YourResourceGroup --name Calculator
```



Step 3:- Create an NGINX Ingress Controller

```
# Create a K8s namespace for the ingress resources
kubectl create namespace ingress-calc


# Add the ingress-nginx repository
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx


# Use Helm to deploy an NGINX ingress controller
helm install nginx-ingress ingress-nginx/ingress-nginx \
     --namespace ingress-calc \
     --set controller.replicaCount=2 \
     --set controller.nodeSelector."beta\.kubernetes\.io/os"=linux \
     --set defaultBackend.nodeSelector."beta\.kubernetes\.io/os"=linux
```

Step 4:- Run the Application

- Application is deployed by applying a **YAML** file to the cluster. The two deployments and each corresponding service are created using the YAML file below.

- In this example, we create this file by typing `code calculator.yaml` on Azure CLI. Paste the manifest below and save it. Please note that on line 17 and line 54, we are pulling the prebuilt image from Docker Hub. Feel free to use your own image.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: api
  template:
    metadata:
      labels:
        app: api
    spec:
      containers:
      - name: api
        image: shaikh1996/calculator_api:latest
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: api
spec:
  ports:
  - port: 80
  selector:
    app: api
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: website
```

```yaml
spec:
  replicas: 3
  selector:
    matchLabels:
      app: website
  template:
    metadata:
      labels:
        app: website
    spec:
      containers:
      - name: website
        image: shaikh1996/calculator_site:latest
        resources:
          requests:
            cpu: 100m
            memory: 128Mi
          limits:
            cpu: 250m
            memory: 256Mi
        ports:
        - containerPort: 3000
---
apiVersion: v1
kind: Service
metadata:
  name: website
spec:
  ports:
  - port: 3000
  selector:
    app: website
```

- Run the frontend and backend in the namespace we created using `kubectl apply`

```
kubectl apply -f calculator.yaml --namespace ingress-calc
```

```
PS /home/unified> kubectl apply -f calculator.yaml --namespace ingress-calc
deployment.apps/api created
service/api created
deployment.apps/website created
service/website created
PS /home/unified>
```

Step 6:- Create an Ingress Route

Both the frontend and backend are now running on the Kubernetes Cluster. Now we create an ingress resource to configure the rules that route traffic to our website and API. Run `code ingress.yaml`

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: calculator-ingress
  namespace: ingress-calc
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/ssl-redirect: "false"
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/rewrite-target: /$1
spec:
  rules:
  - http:
      paths:
      - path: /?(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: website
            port:
              number: 3000
```

```
      - path: /api/?(.*)
        pathType: ImplementationSpecific
        backend:
          service:
            name: api
            port:
              number: 80
```

We route the traffic from the root URL to our website service. Similarly, we route the traffic from `/api` to our API service.

- Apply this command to create the ingress resource

```
kubectl apply -f ingress.yaml
```

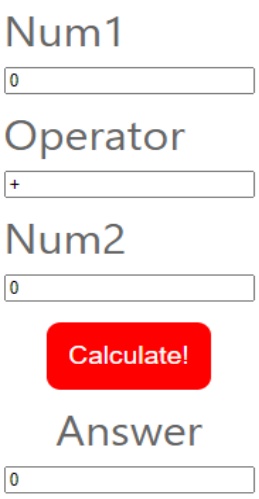
- The Kubernetes load balancer service is created for the NGINX ingress controller, we can access our app with the assigned dynamic public IP address.

```
kubectl --namespace ingress-calc get services -o wide -w nginx-ingress-ingress-nginx-controller
```

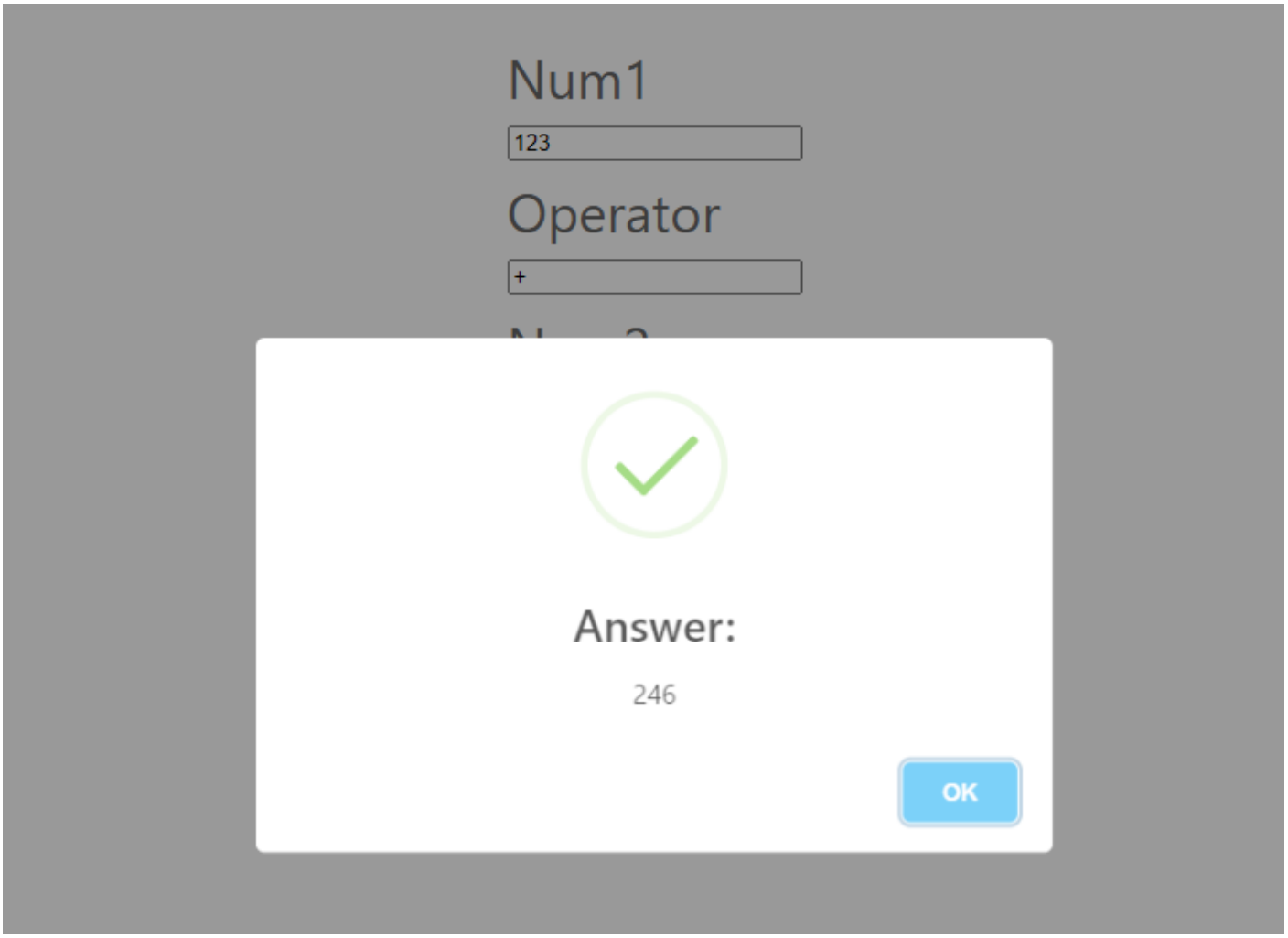
- Open up your browser go to the external ip address and boom.

```
PowerShell
PS /home/unified> kubectl apply -f ingress.yaml
ingress.networking.k8s.io/calculator-ingress created
PS /home/unified> kubectl --namespace ingress-calc get services -o wide -w nginx-ingress-ingress-nginx-controller
NAME                                     TYPE           CLUSTER-IP     EXTERNAL-IP     PORT(S)                      AGE   SELECTOR
nginx-ingress-ingress-nginx-controller   LoadBalancer   10.0.223.20    40.80.77.145    80:30972/TCP,443:30653/TCP   46m   app.kubernetes.io/compone
nt=controller,app.kubernetes.io/instance=nginx-ingress,app.kubernetes.io/name=ingress-nginx
```

We can reach the calculator UI because our traffic is routed from the root URL to the website page by NGINX ingress controller.



- Once we press the calculate button, it will send a HTTP POST request to `/api`. Similarly, the traffic to `/api`is routed to the API service by the NGINX ingress controller. The answer is calculated by the backend API. Responded answer is taken by the frontend website and used to generate a pop up on the site.



Challenges Faced :-

1. Configuration complexity when setting up AKS clusters and managing multiple deployments.
2. Ensuring seamless communication between frontend and backend services.
3. Handling and configuring the NGINX Ingress Controller for proper routing.
4. Setting up AKS clusters and managing deployments presented configuration complexities.
5. Coordinating communication between frontend and backend services required careful configuration.

Business Benefits :-

1. Increased operational efficiency through containerization and automated deployment.
2. Enhanced scalability and flexibility in managing application workloads.
3. Improved resource utilization and cost optimization with Kubernetes orchestration.
4. Streamlined traffic routing and load balancing for a seamless user experience
5. Containerization and automation improved operational efficiency and scalability.
6. Kubernetes orchestration enhanced resource utilization and cost optimization.
7. NGINX Ingress Controller streamlined traffic routing, ensuring a seamless user experience