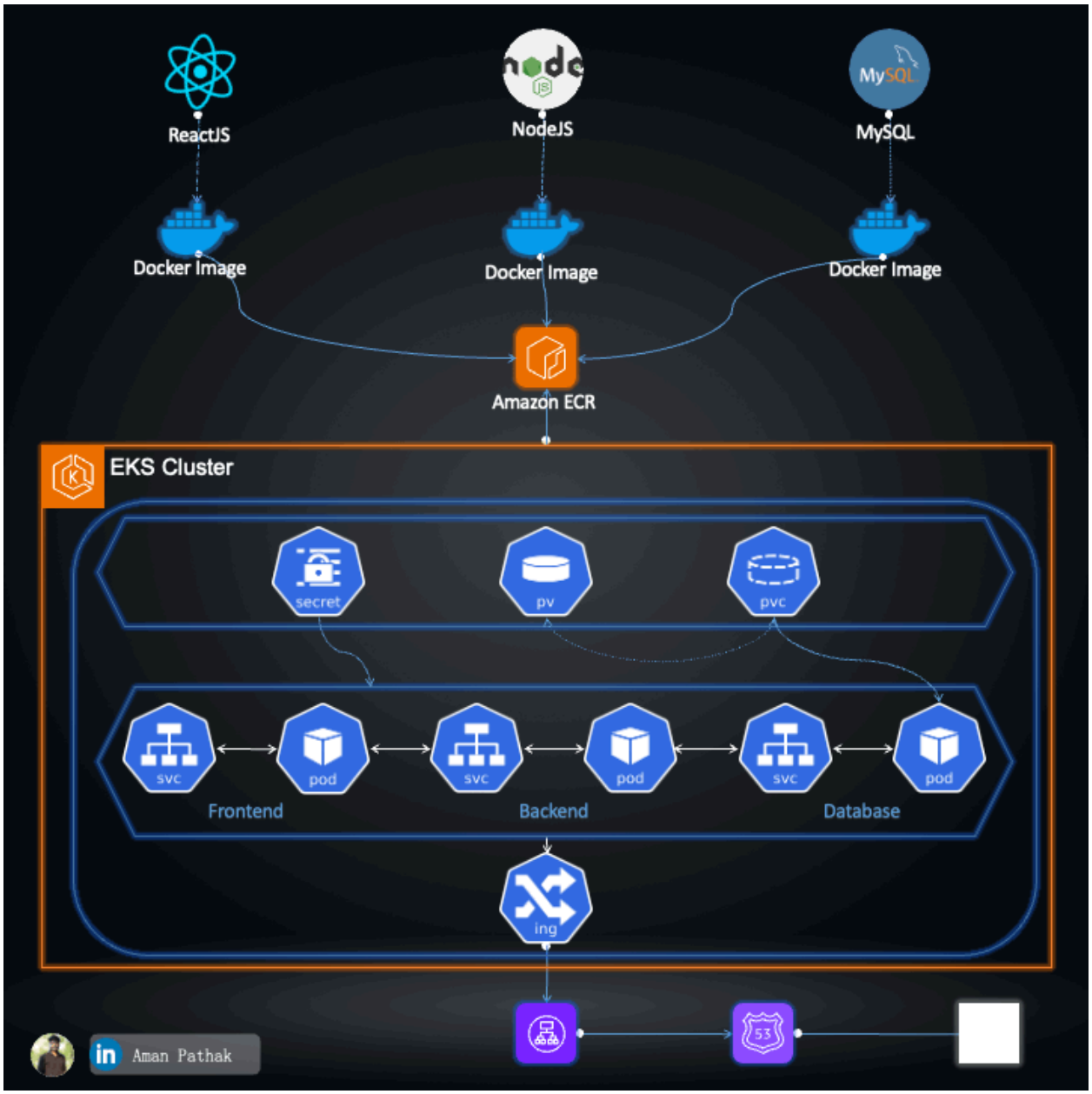
How to Containerize and Deploy a Three-Tier Application on AWS EKS with Kubernetes



Introduction

Deploying a three-tier application on a Kubernetes Cluster might seem challenging, but once you understand the basics, it becomes much easier. In this guide, we'll walk through the steps to containerize your applications with Docker and deploy them on a Kubernetes (EKS) Cluster using Kubernetes manifests. By the end, you'll have a clear roadmap to deploying similar applications in the future.

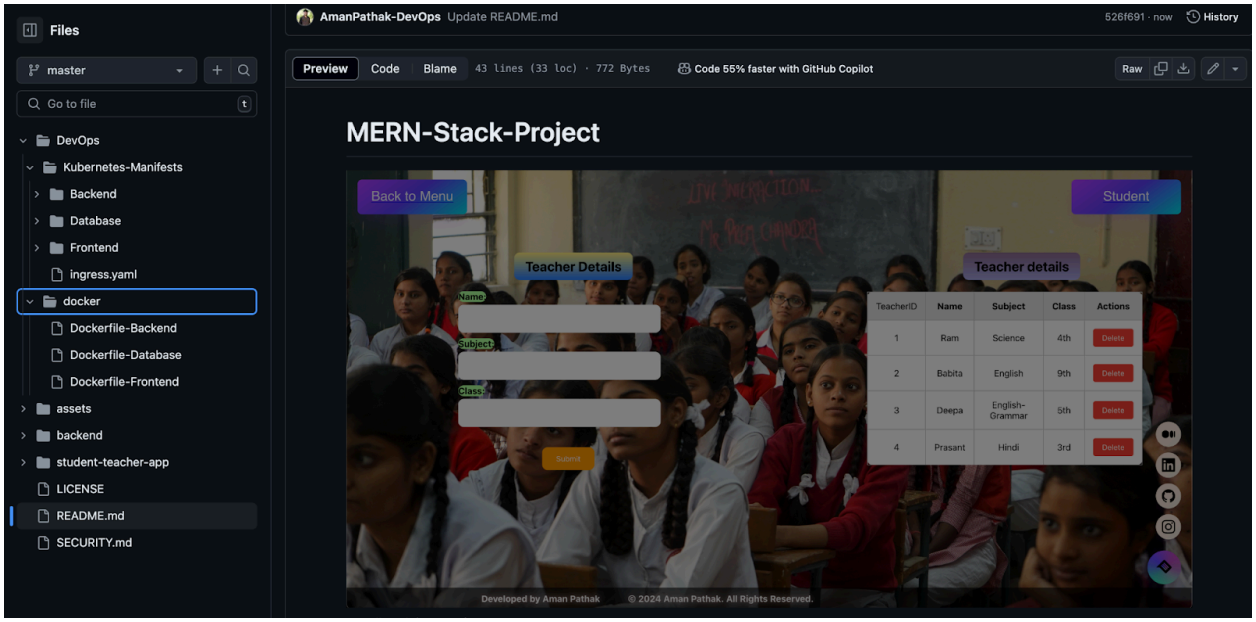
Prerequisites

Before we start, make sure you have the following:

- **Docker Knowledge**: Familiarity with creating Dockerfiles and managing images.
- **Kubernetes Basics**: Understanding of pods, services, and deployments.
- **AWS Account**: Required for setting up EKS.
- **Command Line Tools**: Ensure Docker, kubectl, and AWS CLI are installed and configured.

Before deploying a three-tier application on Kubernetes Cluster, we need to understand the directory structure. So, you won't be confused while doing a follow-along.

There are three categories of all the directories shown below in the snippet.
The first one is Source Code where the directory name is backend and student-teacher-app.
The second and third are the same but have different use cases that come under DevOps.
In the DevOps directory, we have docker and Kubernetes Manifests. As the name suggests, initially we are going to containerize the application then, we will deploy it on the Kubernetes(EKS) Cluster with the help of Kubernetes Manifests.



Hope you understand the structure of the directory.
Let me tell you one thing before going ahead with this article. My main focus is to teach you how to deploy three-tier applications where frontend, backend, and database communicate with each other by creating Dockerfiles from scratch and Kubernetes manifests. Once you understand how to do this, you can easily deploy any application where you just need to ask a few things from your developer. One more thing, there are tons of features we can add to our deployment files in Kubernetes but our main aim is to deploy our application and it should store the data, delete the data, and retrieve the data accordingly. If it's working, it means we deployed our application perfectly.

Containerize the Applications

Database Dockerfile

We are going to start with the Database application. Generally, Organizations use some managed cloud services for it such as RDS, and DocumentDB. But in our case, we need to understand how to work with unmanaged services.

Our database docker file is straightforward as we did not need to provide more instructions. We have provided the base image, port exposes, volume to persist the data, and CMD instruction to run the MySQL server.
Here is one more important observation: we can directly deploy the Kubernetes deployment file with the same base image name but again for understanding purposes. Once you understand the process, you can find an effective and efficient way.

```
FROM mysql:latest

# Expose the MySQL port
EXPOSE 3306

# Define a named volume for MySQL data
VOLUME /var/lib/mysql

# Command to run the MySQL server
CMD ["mysqld"]
```

Once our dockerfile is ready. We can create the docker image and push it to our docker hub repository.
Follow the below command to build the image and push it to the docker hub.
To push the docker image on the docker hub, you need to run the **docker login** command which needs the Personal Access Token(PAT) of the docker hub.

```
docker build -t <your_dockerhub_username>/mysql-image:<tag> .
docker push <your_dockerhub_username>/mysql-image:<tag>
```

Backend Dockerfile

Now, we are to work with the Backend Dockerfile. As our code is in nodejs, we will be going to use node as our base image in the Dockerfile.
We need to install dependencies and build our code inside the container. For that, we will be going to use a separate work directory name app. After that, we are going to copy our packages.json file inside the work director. Then, we will install the dependencies by running the npm install command.
Once the dependencies have been installed, we need to copy our source code which is server.js in the current scenario.

After copying the server.js file, we will run our backend application through CMD.

```
# Use the official Node.js image as a base image
FROM node:14

# Set the working directory inside the container
WORKDIR /app

# Copy package.json and package-lock.json to the working directory
COPY package*.json ./

# Install dependencies
RUN npm install

# Copy the rest of the application code to the working directory
COPY . .

# Command to run your application
CMD ["node", "server.js"]
```

Once our dockerfile is ready. We can create the docker image and push it to our docker hub repository.
Follow the below command to build the image and push it to the docker hub.

```
docker build -t <your_dockerhub_username>/backend-image:<tag> .
docker push <your_dockerhub_username>/backend-image:<tag>
```

Frontend Dockerfile

We are going to build this docker file as a multi-stage. The main reason is to decrease the size of the docker image. Generally, we use this in real-time projects. So you should also work on it.

In the frontend docker file, we will be going to use general commands that are used to build a normal react application. If you are seeing a few additional things, such as ENV and npm update. This is because my application was quite old and it needs new packages with new dependencies.
Once our first stage is completed, We will copy the build folder content inside the new stage to reduce the size due to dependencies installed in the node_modules folder in the first stage.

After copying the build folder, we can run our application through the npm start command.

```
# Stage 1: Build the application using Node.js
FROM node:21-alpine3.17 as build
WORKDIR /app
COPY . .
ENV NODE_OPTIONS=--openssl-legacy-provider
RUN npm install
RUN npm update
RUN npm run build

# Stage 2: Run the application
FROM node:21-alpine3.17
WORKDIR /app
COPY --from=build /app /app
ENV NODE_OPTIONS=--openssl-legacy-provider
CMD ["npm", "start"]
```

Once our dockerfile is ready. We can create the docker image and push it to our docker hub repository.
Follow the below command to build the image and push it to the docker hub.

```
docker build -t <your_dockerhub_username>/frontend-image:<tag> .
docker push <your_dockerhub_username>/frontend-image:<tag>
```

Here, we have containerized our application.

Deploying Containerized Applications with Kubernetes Manifests

Before creating any yaml or manifest file, I would like to explain a few things that we are going to use to deploy our three-tier application such as **deployment** for all three applications. We need to connect all three applications which will be done by Kubernetes **Service** object. We need persistent storage, for that, we will

be going to use PersistentVolument and Persistent VolumeClaim. So, If the pods get deleted due to some issue. Our data won't be lost. Also, we will use Kubernetes Secrets to follow the best practices where our backend application needs database credentials, etc. You can use AWS SSM, Vaults, etc tools to keep your credentials more secure.

Database Kubernetes Manifest

I will be going to provide the manifest file along with a detailed explanation.

namespace- This file will create a new namespace as it's good practice to deploy your application in the isolated namespace to avoid any confusion.

```
apiVersion: v1
kind: Namespace
metadata:
  name: mern
  labels:
    name: mern
```

Pv- This file will create a persistent volume where our data will reside even after our pods get deleted. But this won't work without PVC.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: db-pv
  namespace: mern
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: standard
  hostPath:
    path: /mnt/data
```

Pvc- This file will create a persistent volume claim that claims the storage by a pod from a particular persistent volume(PV).

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: db-pvc
  namespace: mern
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: standard
```

Secrets- This file will create a secret for our application. Now the secrets are related to the database including the database name, database password, and database host(Kubernetes service). This will be used by both the Backend and Database deployment pods.

```
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
  namespace: mern
type: Opaque
data:
  host: ZGF0YWJhc2U= #database
  user: cm9vdA== #root
  password: bXlzcWwxMjM= #mysql123
  database: c2Nob29s #school
```

Deployment- This file will create a deployment where we have to create replicas for our database pods. Few things you need to observe such as the image name that we have pushed earlier in this article. Also, we have attached secrets and pvc to our database deployment.

```
apiVersion: apps/v1
kind: Deployment
```

```yaml
metadata:
  name: database
  namespace: mern
  labels:
    role: database
    env: dev
spec:
  replicas: 1
  selector:
    matchLabels:
      role: database
  template:
    metadata:
      labels:
        role: database
    spec:
      containers:
      - name: database
        image: avian19/mysql-mern:2
        imagePullPolicy: Always
        ports:
        - containerPort: 3306
        env:
        - name: MYSQL_ROOT_PASSWORD
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: password
        - name: MYSQL_DATABASE
          valueFrom:
            secretKeyRef:
              name: db-credentials
              key: database
        volumeMounts:
        - name: db-storage
          mountPath: /var/lib/mysql
      volumes:
      - name: db-storage
        persistentVolumeClaim:
          claimName: db-pvc
```

Service- This file will create a service where that will help to allow the connection to the database pods from any other pods. In our case, we need to make a communication between the backend and the database.

```
apiVersion: v1
kind: Service
metadata:
  name: database
  namespace: mern
spec:
  ports:
  - port: 3306
    protocol: TCP
  type: ClusterIP
  selector:
    role: database
```

Backend Kubernetes Manifest

Deployment- This file will create a deployment where we have to create replicas for our backend pods. Few things you need to observe such as the image name that we have pushed earlier in this article. Also, we have attached secrets to our backend deployment.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: mrn
  labels:
    role: backend
    env: dev
spec:
  replicas: 1
  selector:
    matchLabels:
      role: backend
  template:
    metadata:
      labels:
```

```
      role: backend
  spec:
    containers:
    - name: backend
      image: avian19/backend:1
      imagePullPolicy: Always
      ports:
      - containerPort: 3500
      livenessProbe:
        httpGet:
          path: backend/
          port: 3500
        initialDelaySeconds: 3
        periodSeconds: 10
      env:
      - name: host
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: host
      - name: user
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: user
      - name: password
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: password
      - name: database
        valueFrom:
          secretKeyRef:
            name: db-credentials
            key: database
```

Service- This file will create a service where that will help to allow the connection to the backend pods from any other pods. In our case, we need to make a communication between both the database and the frontend pods.
The Backend pod will be running on port 3500 which is targetPort. But we want to access our backend application on port 80. So we have configured that thing as well. This will help us in our Ingress part.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: mern
spec:
  ports:
  - name: backend-port
    port: 80
    protocol: TCP
    targetPort: 3500
  selector:
    role: backend
```

How backend is going to connect with database pods?

Yes, It's a service but something we need to provide to the backend pods. So, it will find a way to communicate with database pods.
So, where we have provided that.
If you observe it, we have created secrets in the Database Kubernetes Manifests file where we have provided hosts as ZGF0YWJhc2U= #database. So, let me tell you, the database(encoded value ZGF0YWJhc2U=) is the service that we have created in the Database Kubernetes Manifests file. So, this database service will help the backend pod to communicate with the database pod.

Frontend Kubernetes Manifest

Deployment- This file will create a deployment where we have to create replicas for our frontend pods. Few things you need to observe such as the image name that we have pushed earlier in this article. Also, we have added the REACT_APP_API_BASE_URL env variable to our front-end deployment to connect with backend pods.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
```

```
  namespace: mern
  labels:
    role: frontend
    env: dev
spec:
  replicas: 1
  selector:
    matchLabels:
      role: frontend
  template:
    metadata:
      labels:
        role: frontend
    spec:
      containers:
      - name: frontend
        image: avian19/frontend-mern:8
        imagePullPolicy: Always
        ports:
        - containerPort: 3000
        env:
        - name: REACT_APP_API_BASE_URL
          value: "http://<your-domain-name>/backend"
        - name: NODE_OPTIONS
          value: "--openssl-legacy-provider"
```

Service- This file will create a service where that will help to allow the connection to the frontend pods from any other pods. In our case, we need to make communication between both the backend and the frontend pods.

```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  namespace: mern
spec:
  ports:
  - port: 3000
    protocol: TCP
  type: ClusterIP
  selector:
```

```
    role: frontend
```

How frontend is going to connect with backend pods?

Yes, It's a service but something we need to provide to the backend pods. So, it will find a way to communicate with backend pods.
So, where we have provided that.
If you observe it, we have provided env REACT_APP_API_BASE_URL in the Backend Kubernetes Manifests file. So, let me tell you, the backend is the service that we have created in the Backend Kubernetes Manifests file. So, this backend service will help the backend pod to communicate with the backend pod.
But this time you see the domain name along with the service name. Why?
You can work in that way also. But, here we will try a new approach where we will use a fully qualified domain name with a backend service name.

How are we doing this?

As we are going to deploy our application outside of the cluster we will use Load Balancer where the Ingress controller will be there to help us. So, we are going to expose our backend application outside of the cluster(not best practices).

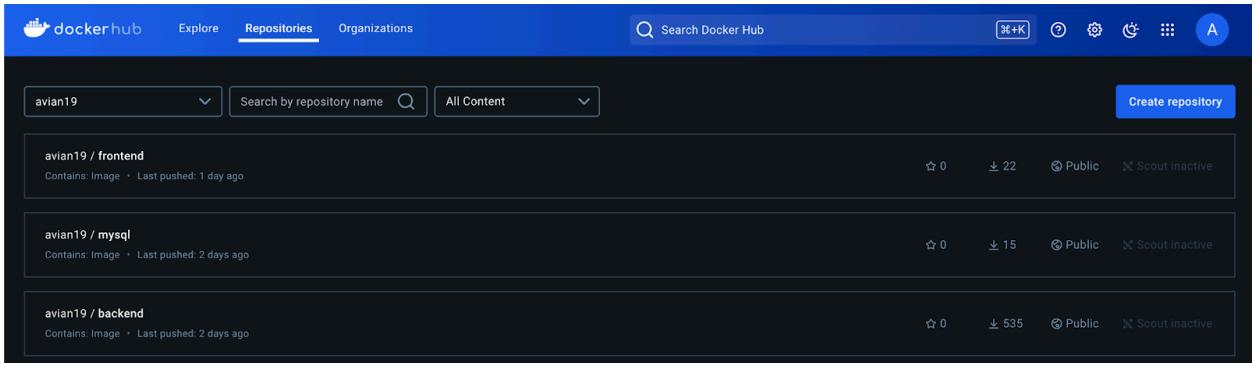
Ingress Kubernetes Manifest

There is only one file that will expose our application outside of the cluster. We are deploying frontend and backend through ingress. So, you can observe that on the default path(/) frontend will be deployed and on the /backend path backend will be deployed. For your information, we are deploying an **internet-facing** load balancer which is used to deploy applications on the internet whereas there is another type of load balancer known as **internal** which helps us to deploy backend applications. But for now, we are deploying both frontend and backend applications on an internet-facing load balancer.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
```

```
  name: mern-alb-dev
  namespace: mern
  annotations:
    kubernetes.io/ingress.class: alb
    alb.ingress.kubernetes.io/listen-ports: '[{"HTTP": 80}]'
    alb.ingress.kubernetes.io/scheme: internet-facing
    alb.ingress.kubernetes.io/target-type: ip
    alb.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: alb
  rules:
    - host: <your-domain-name>
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: frontend
                port:
                  number: 3000
          - path: /backend
            pathType: Prefix
            backend:
              service:
                name: backend
                port:
                  number: 80
```

Now, theory is enough. Let's do a hands-on and deploy our application within a few minutes.
These three are our docker images which have been pushed to the docker hub



All three pods are running

```
NAME                                READY   STATUS    RESTARTS   AGE
backend-78458987b6-wkq9b            1/1     Running   0          88s
database-b7cd6d5c6-mhv2v            1/1     Running   0          69m
frontend-5d98bcfb8c-dgf42           1/1     Running   0          51m
```

All three services are present

```
NAME         TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
backend      ClusterIP   172.20.222.44    <none>        80/TCP     81m
database     ClusterIP   172.20.221.214   <none>        3306/TCP   81m
frontend     ClusterIP   172.20.26.175    <none>        3000/TCP   80m
```

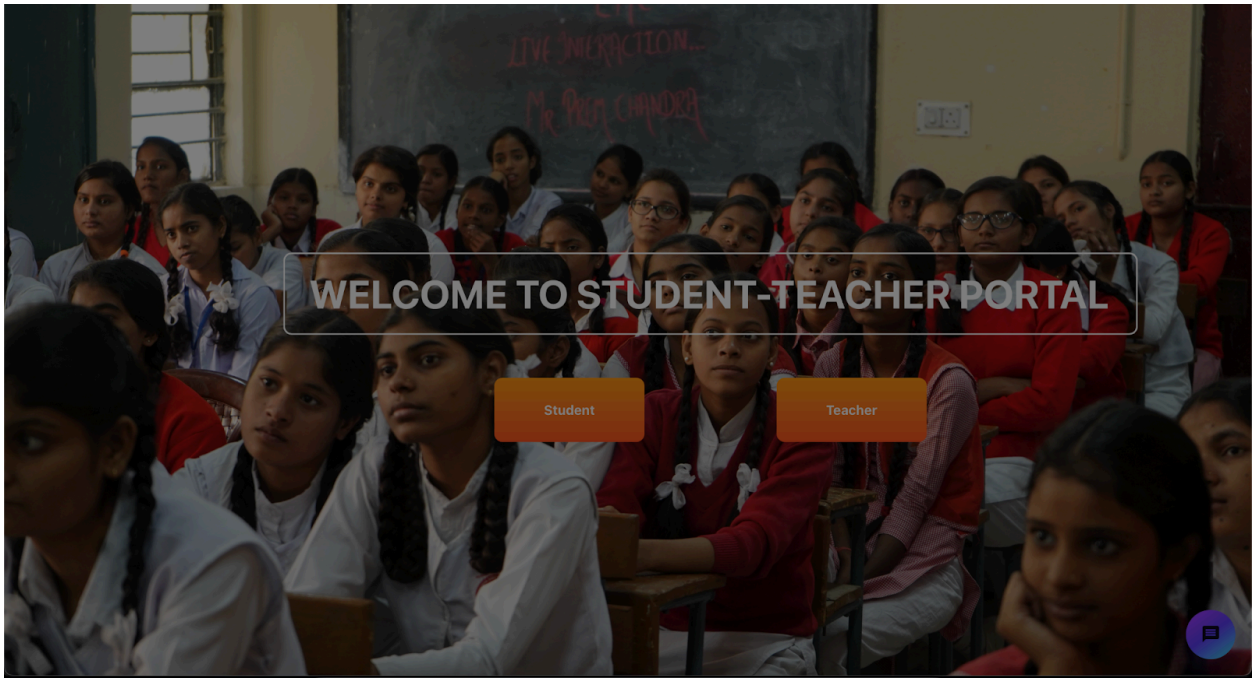
Ingress is reconciled successfully

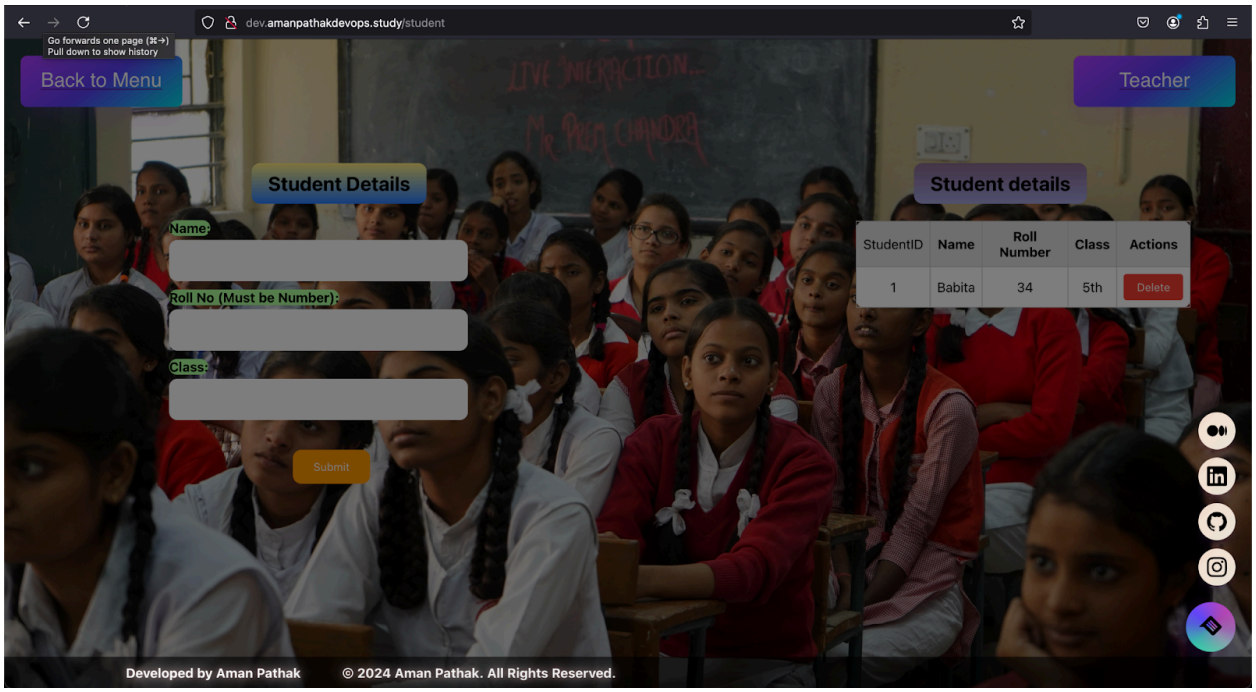
```
NAME          CLASS   HOSTS                       ADDRESS                                                                  PORTS   AGE
mern-alb-dev  alb     dev.amanpathakdevops.study  k8s-mern-mernalbd-d8fadc7152-129541313.ap-south-1.elb.amazonaws.com  80      80m
```

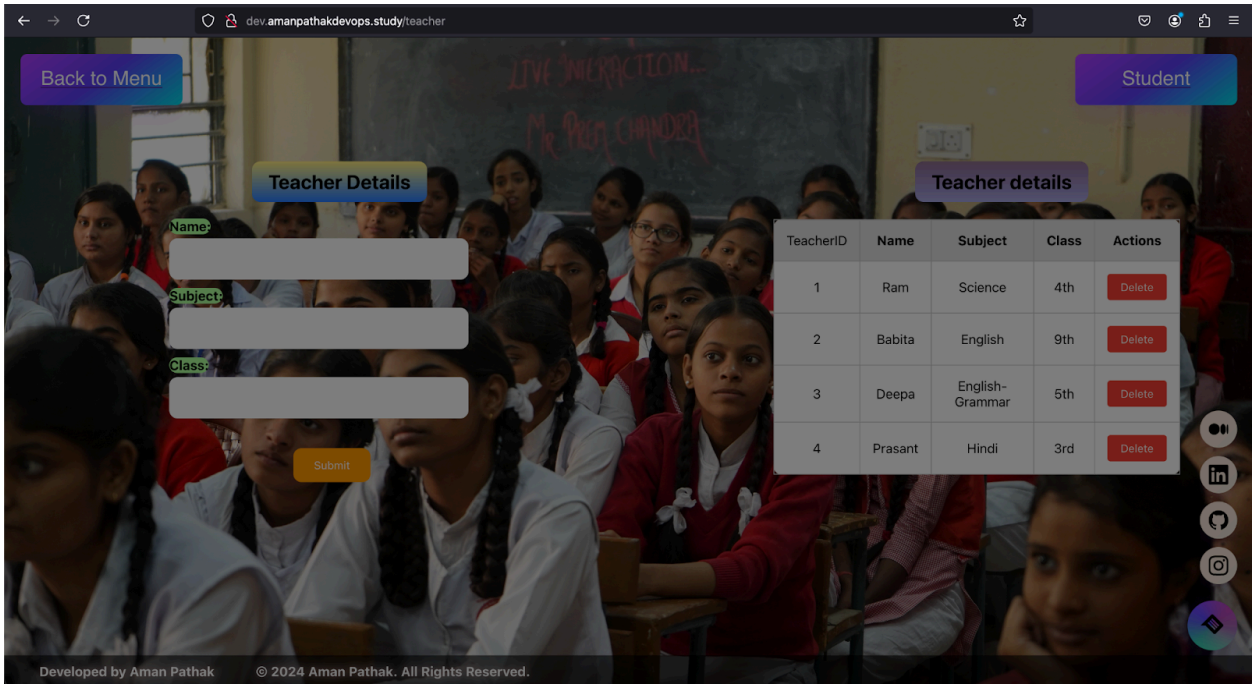
You can run a command to check all the resources in our mern namespace

```
NAME                                 READY   STATUS    RESTARTS   AGE
pod/backend-78458987b6-wkq9b         1/1     Running   0          4m24s
pod/database-b7cd6d5c6-mhv2v         1/1     Running   0          72m
pod/frontend-5d98bcfb8c-dgf42        1/1     Running   0          54m

NAME                 TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
service/backend      ClusterIP   172.20.222.44    <none>        80/TCP     82m
service/database     ClusterIP   172.20.221.214   <none>        3306/TCP   83m
service/frontend     ClusterIP   172.20.26.175    <none>        3000/TCP   81m

NAME                         READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend      1/1     1            1           81m
deployment.apps/database     1/1     1            1           83m
deployment.apps/frontend     1/1     1            1           71m

NAME                                    DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-78458987b6      1         1         1       81m
replicaset.apps/database-b7cd6d5c6      1         1         1       83m
replicaset.apps/frontend-5d98bcfb8c     1         1         1       71m
```

You can hit your domain provided in the ingress on your favorite browser. But make sure you add the record in your domain provider and it should be populated.
The application has been deployed


WELCOME TO STUDENT-TEACHER PORTAL
Student
Teacher


dev.amanpathakdevops.study/student
Back to Menu
Teacher
Student Details
Name:
Roll No (Must be Number):
Class:
Submit
Student details
StudentID
Name
Roll Number
Class
Actions
1
Babita
34
5th
Delete
Developed by Aman Pathak
© 2024 Aman Pathak. All Rights Reserved.



Conclusion

Deploying a three-tier application on Kubernetes is more straightforward than it seems. With this guide, you've learned to containerize your applications and deploy them on an EKS cluster. This foundational knowledge will help you confidently tackle more complex deployments as you progress.

Repository for this Project on GitHub-
https://github.com/AmanPathak-DevOps/MERN-Stack-Project

Follow for more

Join Discord Community: https://lnkd.in/dsEdxpst

Follow on GitHub: https://github.com/AmanPathak-DevOps/

Follow on LinkedIn- https://www.linkedin.com/in/aman-devops/