Student Portal Application Deployment Guide

📌 Project Objective

To deploy a two-tier web application consisting of a Flask frontend and PostgreSQL database backend on Kubernetes using Minikube, with the database hosted on AWS RDS. This project demonstrates containerization, orchestration, secrets management, and cloud integration skills.

Prerequisites

- Docker installed on your local machine
- AWS CLI installed and configured
- kubectl installed
- Kubernetes cluster (Minikube or EKS)
- AWS account with appropriate permissions
- Student Portal application source code with Dockerfile

Create Dockerfile

Create a Dockerfile in your application root directory with the necessary configuration for your Flask application.

Build Docker Image

```
docker build -t student-portal:1.0 .
```

Configure AWS CLI

```
aws configure
```

Enter your AWS Access Key ID, Secret Access Key, default region (us-east-1), and output format.

Create ECR Repository

```
aws ecr create-repository --repository-name student-portal --region us-east-1
```

Note: Save the repository URI from the output

Format: `<account-id>.dkr.ecr.<region>.amazonaws.com/student-portal`

Authenticate Docker to ECR

```
aws ecr get-login-password --region us-east-1 | docker login --username AWS
--password-stdin <account-id>.dkr.ecr.us-east-1.amazonaws.com
```

Tag the Docker Image

```
docker tag student-portal:1.0
<account-id>.dkr.ecr.us-east-1.amazonaws.com/student-portal:1.0
```

Push Image to ECR

```
docker push <account-id>.dkr.ecr.us-east-1.amazonaws.com/student-portal:1.0
```

Create RDS PostgreSQL Database

Navigate to AWS RDS Console and create a new database:

- Click "Create database"
- Choose PostgreSQL engine
- Configure instance settings:
  - DB instance identifier: `student-portal-db`
  - Master username: `postgres` (or your preferred username)
  - Master password: Create a strong password
  - DB instance class: Select based on your needs (e.g., db.t3.micro for testing)
  - Storage: Configure as needed
  - VPC and Security Group: Ensure your Kubernetes cluster can access the database
- Note down the endpoint URL after creation

Database connection string format:

```
postgresql://<username>:<password>@<rds-endpoint>:<port>/<database-name>
```

Step 9

Store Database Connection in AWS Secrets Manager

```
aws secretsmanager create-secret \
  --name db/student-portal/db_link \
  --description "Database connection string for Student Portal" \
  --secret-string
'{"db_link":"postgresql://username:password@endpoint:5432/dbname"}' \

  --region us-east-1
```

Important: Replace the connection string with your actual RDS details.Step 10

Create IAM Policy for Secrets Manager Access

Create a file named `secrets-policy.json`:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "secretsmanager:GetSecretValue",
        "secretsmanager:DescribeSecret"
      ],
      "Resource":
"arn:aws:secretsmanager:us-east-1:<account-id>:secret:db/student-portal/db_link
*"
    }
  ]


}
```

Create the policy:

```
aws iam create-policy \
  --policy-name StudentPortalSecretsReadPolicy \


  --policy-document file://secrets-policy.json
```

Create IAM User

```
aws iam create-user --user-name student-portal-secrets-reader
```

Attach Policy to IAM User

```
aws iam attach-user-policy \
  --user-name student-portal-secrets-reader \
```

```
  --policy-arn
arn:aws:iam::<account-id>:policy/StudentPortalSecretsReadPolicy
```

Create Access Keys for IAM User

```
aws iam create-access-key --user-name student-portal-secrets-reader
```

Important: Save the AccessKeyId and SecretAccessKey from the output securely.Step 14

Create Kubernetes Namespace

```
kubectl create namespace student-portal
```

Create Docker Registry Secret for ECR

```
kubectl create secret docker-registry ecr-registry-secret \
  --docker-server=<account-id>.dkr.ecr.us-east-1.amazonaws.com \
  --docker-username=AWS \
  --docker-password=$(aws ecr get-login-password --region us-east-1) \
```

```
  --namespace=student-portal
```

Note: ECR tokens expire after 12 hours. For production, consider using IAM roles for service accounts (IRSA).Step 16

Create Kubernetes Secret for AWS Credentials

```
kubectl create secret generic aws-credentials \
  --from-literal=AWS_ACCESS_KEY_ID=<your-access-key-id> \
  --from-literal=AWS_SECRET_ACCESS_KEY=<your-secret-access-key> \

  --namespace=student-portal
```

Create Deployment YAML File

Create a file named `deployment.yaml`:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: student-portal
  namespace: student-portal
  labels:
    app: student-portal
spec:
  replicas: 2
  selector:
    matchLabels:
      app: student-portal
  template:
    metadata:
      labels:
        app: student-portal
    spec:
      # Init container to fetch secrets from AWS Secrets Manager
      initContainers:
      - name: fetch-secrets
        image: amazon/aws-cli:latest
        env:
        - name: AWS_ACCESS_KEY_ID
```

```
          valueFrom:
            secretKeyRef:
              name: aws-credentials
              key: AWS_ACCESS_KEY_ID
        - name: AWS_SECRET_ACCESS_KEY
          valueFrom:
            secretKeyRef:
              name: aws-credentials
              key: AWS_SECRET_ACCESS_KEY
        - name: AWS_DEFAULT_REGION
          value: "us-east-1"
        command: ["/bin/sh"]
        args:
          - -c
          - |
            #!/bin/sh
            set -e

            echo "Fetching secret from AWS Secrets Manager..."

            # Fetch the secret value
            DB_LINK=$(aws secretsmanager get-secret-value \
              --secret-id db/student-portal/db_link \
              --region us-east-1 \
              --query SecretString \
              --output text | sed 's/.*"db_link":"\([^"]*\)".*/\1/')

            mkdir -p /secrets

            # Write to shared volume
            echo "DB_LINK=${DB_LINK}" > /secrets/db_link.env

            echo "Secret fetched and saved successfully to volume from init
container"

        # Volume mount to share secrets
        volumeMounts:
        - name: secrets-volume
          mountPath: /secrets

      # Image pull secret for ECR
      imagePullSecrets:
      - name: ecr-registry-secret

      # Main application container
      containers:
      - name: flask
        image: <account-id>.dkr.ecr.us-east-1.amazonaws.com/student-portal:1.0
        imagePullPolicy: Always
        ports:
        - containerPort: 8000
        command: ["/bin/sh"]
        args:
          - -c
          - |
            #!/bin/sh
            # Source the environment file
            set -a
            . /secrets/db_link.env
            set +a

            # Verify the variable is set
            echo "DB_LINK environment variable is loaded: ${DB_LINK}"

            # Start your Flask application
            exec python run.py

        # Volume mount to access secrets from init container
        volumeMounts:
        - name: secrets-volume
          mountPath: /secrets
          readOnly: true

      # Volume to share secrets between init container and main container
      volumes:
      - name: secrets-volume
        emptyDir:
          medium: Memory
```

Important: Replace `<account-id>` with your actual AWS account ID in the image URL.Step 18

Create Service YAML File

Create a file named `service.yaml`:

```
apiVersion: v1
kind: Service
```

```
metadata:
  name: student-portal
  namespace: student-portal
spec:
  selector:
    app: student-portal
  ports:
    - protocol: TCP
      port: 8080          # Service port
      targetPort: 8000   # Container port
  type: ClusterIP
```
Step 19

Apply the Deployment

```
kubectl apply -f deployment.yaml
```

Apply the Service

```
kubectl apply -f service.yaml
```

Verify Pod Status

```
kubectl get pods -n student-portal
```

Wait until all pods are in `Running` state.

Check Deployment Logs

Check init container logs:

```
kubectl logs <pod-name> -n student-portal -c fetch-secrets
```

Check application container logs:

```
kubectl logs <pod-name> -n student-portal -c flask
```

If there are issues:

```
kubectl describe pod <pod-name> -n student-portal
```

Access the Application Using Port Forward

```
kubectl port-forward service/student-portal 8080:8080 -n student-portal
```

Open your browser and navigate to:

```
http://localhost:8080
```

Test Database Connectivity (Optional)

Create a debug pod to test database connectivity:

```
kubectl run postgres-debug \
  --image=postgres:15 \
  --rm -it \
  --namespace=student-portal \
```

```
  --command -- bash
```

Inside the debug pod, connect to the database:

```
psql "postgresql://username:password@rds-endpoint:5432/dbname"
```

Note: Replace with your actual database credentials and endpoint.

Troubleshooting Commands

Check all resources in namespace:

```
kubectl get all -n student-portal
```

View detailed pod information:

```
kubectl describe pod <pod-name> -n student-portal
```

View events in namespace:

```
kubectl get events -n student-portal --sort-by='.lastTimestamp'
```

Clean Up Resources (When Done)

Delete Kubernetes resources:

```
kubectl delete namespace student-portal
```

Delete AWS ECR repository:

```
aws ecr delete-repository --repository-name student-portal --force --region
us-east-1
```

Delete AWS Secrets Manager secret:

```
aws secretsmanager delete-secret --secret-id db/student-portal/db_link
--force-delete-without-recovery --region us-east-1
```

Delete IAM resources:

```
aws iam detach-user-policy --user-name student-portal-secrets-reader
--policy-arn arn:aws:iam::<account-id>:policy/StudentPortalSecretsReadPolicy
aws iam delete-access-key --user-name student-portal-secrets-reader
--access-key-id <access-key-id>
aws iam delete-user --user-name student-portal-secrets-reader
```

```
aws iam delete-policy --policy-arn
arn:aws:iam::<account-id>:policy/StudentPortalSecretsReadPolicy
```

Delete RDS instance:

```
aws rds delete-db-instance --db-instance-identifier student-portal-db
--skip-final-snapshot
```

Important Notes

1. **Security:** Store all credentials securely. Never commit secrets to version control.

2. **ECR Token Expiry:** ECR authentication tokens expire after 12 hours. For production, use IAM Roles for Service Accounts (IRSA) on EKS.
3. **RDS Security:** Ensure RDS security groups allow inbound traffic from your Kubernetes cluster nodes.
4. **Cost Management:** Remember to delete resources when not in use to avoid unnecessary AWS charges.
5. **Production Considerations:**
   - Use IRSA instead of IAM user credentials
   - Implement automated ECR token renewal
   - Use AWS RDS Proxy for better connection management
   - Enable RDS encryption at rest and in transit
   - Set up proper monitoring and logging

Student Portal Application Deployment Guide

Steps Summary:

1. Create Dockerfile for the student-portal app.
2. Build the image using Dockerfile.
3. Create the ECR repo to store the image.
4. Tag the image based on the ECR repo name.
5. Authenticate the system to AWS using aws cli "aws configure"
6. Authenticate the system to ECR using aws cli "aws ecr get-login-password & docker login".
7. Push the image to ECR repo
8. Create kubernetes secret of type docker-registry, with ecr login password (using kubectl).
9. Create postgres db in RDS.
10. Create the db_link in aws secret manager.
11. Create deployment YAML file with the below details:
    a. Create init container to fetch secrets from AWS Secrets Manager. Use image: amazon/aws-cli
    b. Create an IAM user in aws with custom policy attached for reading the aws secret manager.
    c. Create kubernetes secret of type Opaque, with access key and secret access key of that user.
    d. Provide that access key and secret access key as ENV variable to the init container, for authentication.

   e. Inside init container, execute aws cli "aws secretsmanager get-secret-value" to fetch the db_link
   f. Extract the db_link value, and store it in a file inside the volume mount (Common volume for init conatiner and main container).
   g. Create the main container by using ECR image and provide "imagePullSecret" for docker-registry secret.
   h. Inside the main container, read the db_link file from volume mount and set it as ENV variable for the container and override the CMD.
12. Create service YAML file for the above deployment, with type ClusterIP.
13. Use "kubectl apply" to create all the resources in the required namespace.
14. Use kubectl port-forward to access the minikube's cluster ip service from the local machine itself.
15. Inorder to test the db connectivity separately, create a debug pod using image:postgres, and use the psql command inside the pod.