Deployment in kubernetes:

Design and Implement a Microservices-Based Application with Four Distinct Services

This step is covered by the following:

Microservices Design:

- The four microservices are:
  - **Authentication Service**: Handles user authentication (JWT, login).
  - **Digital Asset Service**: Manages digital assets (images).
  - **Marketplace Service**: Manages product listings, pricing, orders, etc.
  - **Payment Service**: Manages payment processing.
  - **AMQP service**: For Message queue
  - **Mongodb service**: For databases

Core Principles Focused On:

- **Independent Deployment**: Each service is designed to be independently deployable. Kubernetes allows us to update one service without affecting the others.
- **Scalability**: Kubernetes can scale each service independently depending on demand. For example, if the **Marketplace Service** has more traffic, we can scale it independently from the others.

Containerization with Docker:

- A **two-stage Dockerfile** for each microservice is provided, with a **build stage** (for installing dependencies and preparing the app) and a **production stage** (which contains only the necessary code and dependencies for a smaller production-ready image).
- This ensures the services are lightweight and optimized for production.

Kubernetes Deployment:

- Kubernetes deployment and service configuration files are provided for each microservice.

- Each microservice is deployed as a separate Kubernetes Deployment with its own service, allowing it to scale independently.

Microservices Design

Let's clarify the roles of each microservice in our architecture:

Authentication Service: Handles user authentication (e.g., login, JWT tokens).

Digital Asset Service: Manages digital assets images.

Marketplace Service: Manages the listing of products, pricing, orders, and search functionality.

Payment Service: Handles payment processing (integration with payment gateways).

Each of these services will be Dockerized and deployed as **independent services** in a **Kubernetes cluster**, focusing on the core principles of **independent deployment** and **scalability**.

Dockerizing the Microservices

We'll create a **multi-stage Dockerfile** for each microservice, focusing on separating the build environment (to install dependencies) and the production environment (which is smaller and optimized).

Let's start with a sample Dockerfile for one of the services (e.g., the **Authentication Service**), and then the same approach can be applied to the other services.

Dockerfile for Authentication Service

```
# Stage 1: Build
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Stage 2: Production
FROM node:18-alpine
WORKDIR /app
COPY --from=build /app /app
ENV PORT=3012
ENV NODE_ENV=production
EXPOSE 2012
CMD ["npm", "start"]
```

Explanation of the Dockerfile:

Stage 1: Build:

- We use node:18-alpine to keep the image small.
- We set the working directory to /app and copy over package.json files to install dependencies using npm install.
- After that, we copy the entire app code into the container.

Stage 2: Production:

- We use node:18-alpine again to keep the image small for the production environment.
- The built app is copied over from the first stage.
- We set environment variables such as NODE_ENV to production and expose the port that the service will run on (3001 for the Authentication service).
- The container will run the application using "npm start".

Dockerizing the Other Services

We can use the same structure for the other services, just changing the service-specific details (port number, environment variables, etc.).

Digital Asset Service Dockerfile:

```
# Stage 1: Build
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Stage 2: Production
FROM node:18-alpine
WORKDIR /app
COPY --from=build /app /app
ENV PORT=3011
ENV NODE_ENV=production
EXPOSE 3011
CMD ["npm", "start"]
```

Marketplace Service Dockerfile:

```
# Stage 1: Build
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Stage 2: Production
FROM node:18-alpine
WORKDIR /app
COPY --from=build /app /app
ENV PORT=3013
ENV NODE_ENV=production
EXPOSE 3013
CMD ["npm", "start"]
```

Payment Service Dockerfile:

```
# Stage 1: Build
FROM node:18-alpine AS build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .

# Stage 2: Production
FROM node:18-alpine
WORKDIR /app
COPY --from=build /app /app
ENV PORT=3012
ENV NODE_ENV=production
EXPOSE 2012
CMD ["npm", "start"]
```

Building and pushing the Docker Images

We need to **build the Docker images** for each service and **push them to a Docker registry** (e.g., Docker Hub, AWS ECR, Google Container Registry).

Build Docker Images:

```
# Build Authentication Service

docker build -t shoib/authentication:latest

# Build Digital Asset Service

docker build -t shoib/digital-assets:latest .

# Build Marketplace Service
docker build -t shoib/marketplace:latest .
# Build Payment Service
docker build -t shoib/payment:latest .
```

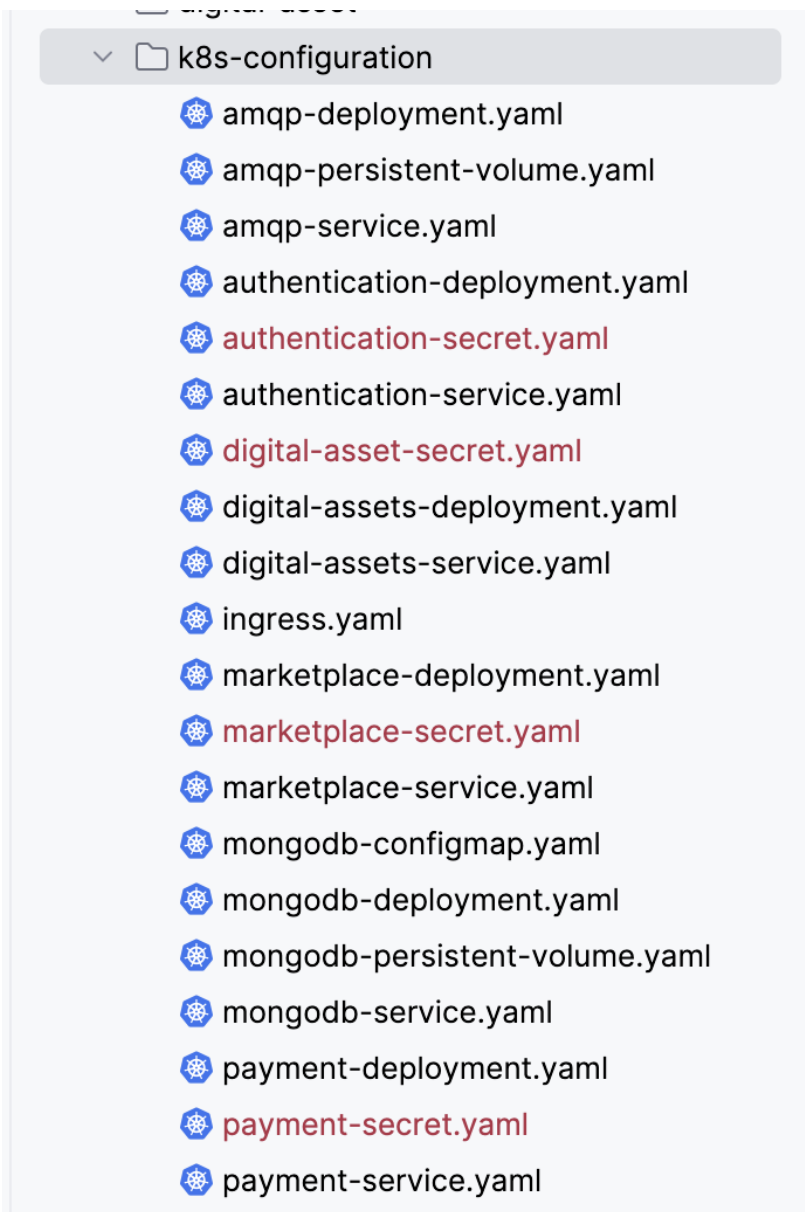
Push Docker Images:

After building the images, I have push them to my registry:

```
# Push Authentication Service

docker push shoib/authentication:latest

# Push Digital Asset Service

docker push shoib/digital-assets:latest

# Push Marketplace Service

docker push shoib/marketplace:latest

# Push Payment Service

docker push shoib/payment:latest
```

Kubernetes Deployment

Once the Docker images are built and pushed, We can deploy the microservices in a **Kubernetes cluster**.



k8s-configuration
- amqp-deployment.yaml
- amqp-persistent-volume.yaml
- amqp-service.yaml
- authentication-deployment.yaml
- authentication-secret.yaml
- authentication-service.yaml
- digital-asset-secret.yaml
- digital-assets-deployment.yaml
- digital-assets-service.yaml
- ingress.yaml
- marketplace-deployment.yaml
- marketplace-secret.yaml
- marketplace-service.yaml
- mongodb-configmap.yaml
- mongodb-deployment.yaml
- mongodb-persistent-volume.yaml
- mongodb-service.yaml
- payment-deployment.yaml
- payment-secret.yaml
- payment-service.yaml

Img 2.1 kubenetes files screenshots please refer below link for details

for reference Refer : *https://github.com/Mtech-Assignment/scalable-services/tree/main/k8s-configuration*

Deploy All Microservices to Kubernetes:

We can create similar deployment and service YAML files for the other three services (Digital Asset, Marketplace, and Payment) by modifying the ports, images, and names accordingly.

Once the YAML files are ready, we can deploy them using the kubectl apply command:

Scaling and Management with Kubernetes

Kubernetes makes it easy to scale services. For example, if the **Authentication Service** is under heavy load, we can scale it independently:

```
kubectl scale deployment authentication --replicas=5
```

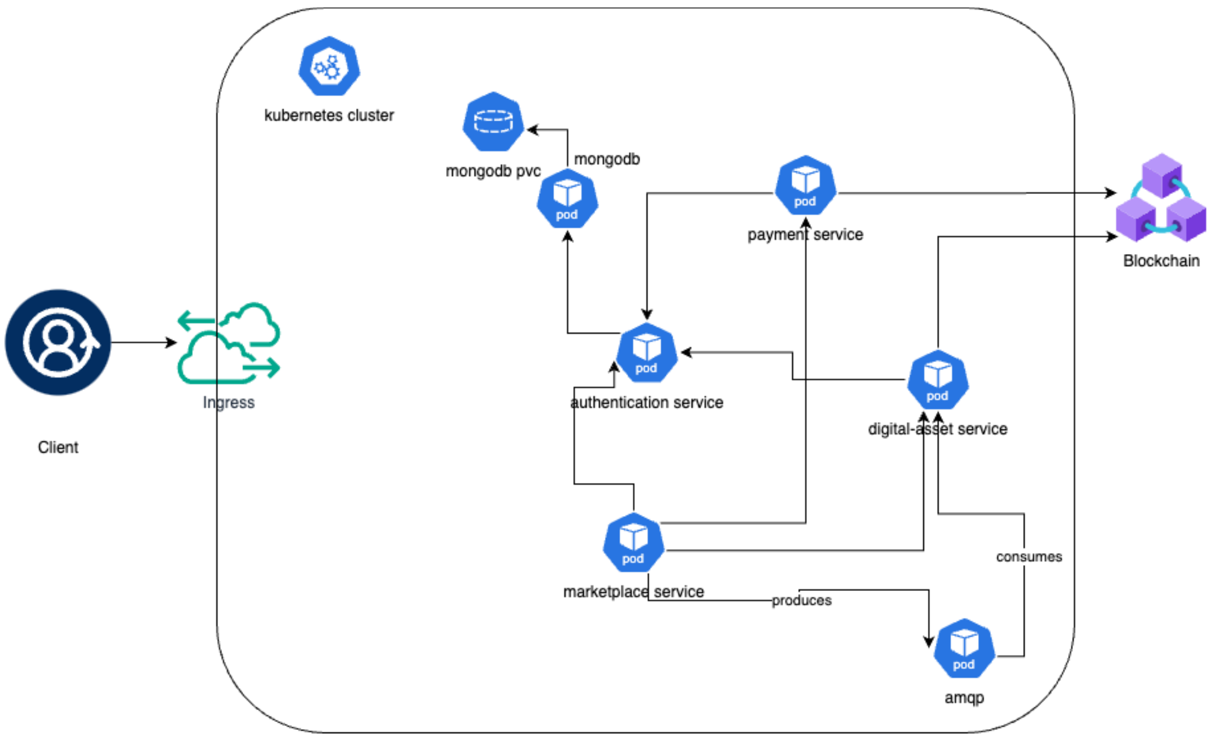
We can also update the deployment to use a new image:

```
kubectl set image deployment/authentication
authentication=shoib/authentication:latest
```

Ingress (For External and Service to Service communication)

Ingress will act as gateway to Kubernetes cluster

Ingress in a Kubernetes cluster is an API object that manages external access to services, typically HTTP and HTTPS. It acts as a reverse proxy, routing incoming traffic from outside the cluster to appropriate services inside the cluster based on defined rules. Ingress allows for path-based or host-based routing, enabling the use of a single external IP address to expose multiple services. It also supports features like SSL termination, load balancing, and URL rewriting. Typically, Ingress controllers like Nginx is deployed to handle the traffic routing as specified by Ingress resources. This simplifies service exposure and management, reducing the need for multiple LoadBalancers or NodePorts, making it cost-efficient and scalable for large applications. Ingress is essential for managing complex routing scenarios and providing centralized traffic control within a Kubernetes ecosystem.



Img 2.2 Application complete Architecture with ingress

As we can see client will call the API ingress will be the main component to connect with both client and services. The kubernetes cluster will be connected to ingress. That means all the services will be interacting with ingress whichever is needed. Here we can see **payment service, authentication service, marketplace service and digital-assets services** will be open for ingress but **mongodb and amqp services** will not be open to the ingress.

Conclusion

Microservices Architecture: We've designed and deployed four Express-based microservices, with each microservice containerized using a two-stage Dockerfile for build and production.

Kubernetes Deployment: Each service was deployed using Kubernetes, making them independently scalable and manageable.

Docker for Automation: Docker simplifies the deployment process by ensuring that each microservice is packaged with all dependencies, and Kubernetes takes care of deployment, scaling, and fault tolerance.

This solution demonstrates both independent deployment and scalability while leveraging Docker to streamline the process of building, testing, and deploying our microservices.

Demonstrate How Docker Can Be Used to Simplify and Automate the Deployment Process

This step is covered by the following:

Dockerization of the Microservices:

- **Dockerfiles** are provided for each of the four microservices, demonstrating the two-stage build process.
- Docker allows the services to be packaged with all their dependencies and configurations, ensuring that they run the same way in development, staging, and production.

Automated Build and Push:

- The **docker build** commands show how each microservice is containerized by building Docker images for each service.
- The **docker push** commands show how these images are pushed to a container registry (e.g., Docker Hub), making it easy to deploy them in Kubernetes or any cloud platform.

Kubernetes Deployment Automation:

- **Kubernetes YAML files** for each service's Deployment and Service configurations are provided. This enables us to deploy each microservice to a Kubernetes cluster.
- Using **kubectl apply**, we can automate the deployment process to Kubernetes.

CI/CD Automation (Optional):

- The example includes a **GitHub Actions** workflow, which automates the entire pipeline: building Docker images, pushing them to Docker Hub, and deploying the services to a Kubernetes cluster. This is an example of how Docker and Kubernetes can work together in a continuous integration/continuous deployment (CI/CD) setup.

.env files conversion to Secrets using script:

For this we have made a script that will change all .env files into Secrets
https://github.com/Mtech-Assignment/scalable-services/blob/main/scripts_for_env
also we have written. Scripts to automatically run all applications
https://github.com/Mtech-Assignment/scalable-services/blob/main/apply-k8s.sh