Building and Deploying a Microservices using Docker/Kubernetes

By Group – 12 :

Sripada Manvitha – IIT2022018

Shweta Pandey – IIB2022011

Shreya Sinha – IIB2022034

Sushmitha Raj – IIT2022093

Ch. Praghna - IIT2022095

Rishitha Gogu – IIT2022092

Objective:

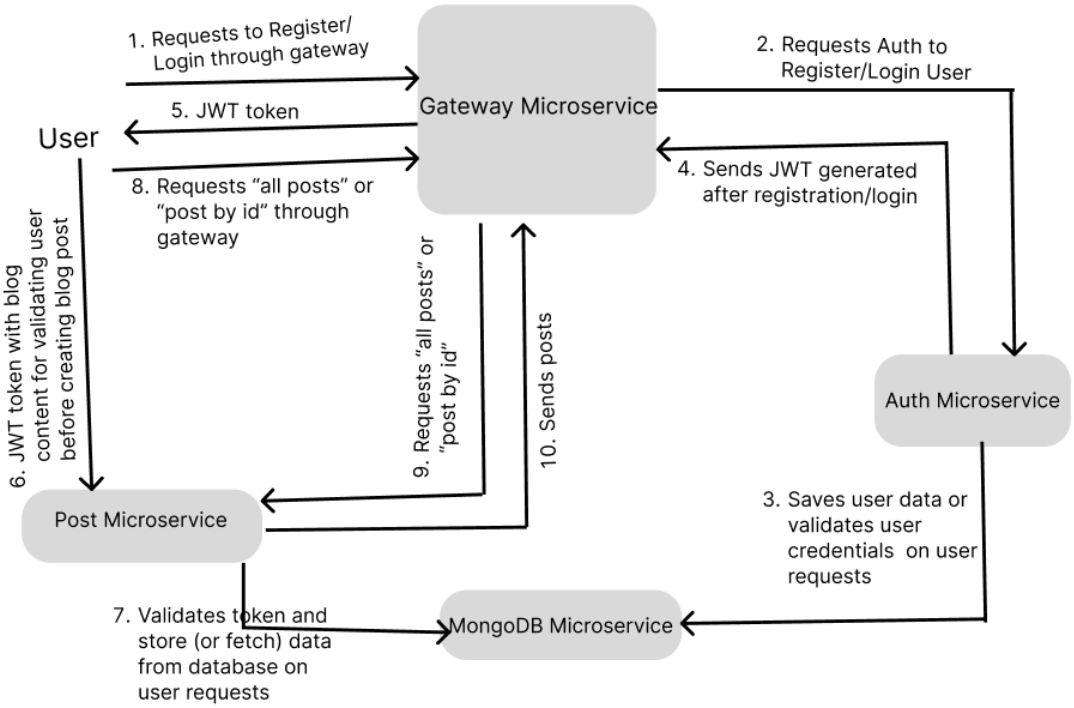
The objective of this project is to design, develop, containerize, and orchestrate a microservices-based application using Docker and Kubernetes. The application consists of multiple independent services communicating via REST APIs. Each service is containerized using Docker, managed collectively with Docker Compose, and deployed on Kubernetes with proper scaling, health checks, and service discovery.

Our project is a microservices-based blog application built with Node.js, Express, MongoDB, Docker, and Kubernetes. It includes modular services like Auth, Post, Gateway and MongoDB database each deployed in its own container and orchestrated via Kubernetes.

Tech Stack

- **Node.js** (Express + Mongoose)
- **MongoDB** (Atlas)
- **Docker** & **Docker Compose**
- **JWT** for auth
- **Axios** for inter-service HTTP calls
- **Kubernetes**

Architecture:



Each microservice has its own:

- Dockerfile
- Kubernetes deployment and service
- Health checks and resource limits

API documentation:

1. Auth Service (/auth)

   - POST /auth/register -> Register new user
   - POST /auth/login -> Login user and return JWT token
   - GET /auth/health/liveness -> Liveness probe
   - GET /auth/health/readiness -> Readiness probe

2. Post Service (/post)

   - Requires Authorization: Bearer <token>
   - POST /post/createPost -> Creates a new blog post

   Request Body:

```json

{

"title": "Post Title",

"content": "Post content...",

"author": "<user_id>"

}
```

- GET /post/getAllPosts -> Returns all blog posts
- GET /post/getPost/:id -> Fetch a specific post by ID

- GET /post/health/liveness -> Liveness probe
- GET /post/health/readiness -> Readiness probe

3. Gateway Service

- GET ${PORT}/health/liveness -> Liveness probe
- GET ${PORT}/health/readiness -> Readiness probe

Deployment Instructions:

1. **Docker Containerization**
    - Each microservice (e.g., Post Service) is containerized using Docker.
    - Dockerfile is created for each service to define the environment and dependencies.
    - **Build the Docker image:**

      docker build -t shweta1902/auth-service ./auth

      docker build -t shweta1902/post-service ./post
    - **Push the image to DockerHub:**

      docker push shweta1902/auth-service

      docker push shweta1902/post-service
2. **Communication Between Microservices**

- Microservices communicate internally using their Kubernetes service names (DNS-based service discovery).
- For example, if gateway-service needs to connect to getAllPosts Service, it uses a command.

  http://post-service:5002/post/getAllPosts
- Environment variables and service names are configured inside Kubernetes manifests.

3. **Kubernetes Deployment**
   - Created a Deployment YAML for each microservice specifying:
     - Container image
     - Ports
     - Resource requests and limits
     - Liveness and readiness probes
   - Create a Service YAML for each microservice to expose it internally (ClusterIP) or externally (NodePort/LoadBalancer).
   - **Command to apply Deployment:**

     kubectl apply -f k8s/auth-deployment.yaml

     kubectl apply -f k8s/post-deployment.yaml

4. **Expose Microservices for External Access**
   - Using NodePort, we access services via a minikube ip which can obtained by running “minikube ip” command.

     http://192.168.49.2:30204/post/getAllPosts

     http://192.168.49.2:30224/post/createPost

     http://192.168.49.2:30204:auth/register

     http://192.168.49.2:30204/auth/login

5. **Verify Deployment**
   - Check pods: kubectl get pods

- Check services: kubectl get services
- Describe pods for resource and probe validation,

  eg : kubectl describe pod auth-deployment-745b9b5d7-bpggb

6. **Health Checks**
   - Validate liveness and readiness endpoints using curl:

     curl http://192.168.49.2:30204/post/health/liveness

     curl http://192.168.49.2:30204/post/health/readiness

Learnings:

1. Learned how **Docker images** are created, built from Dockerfiles, and how containers are isolated yet can communicate through networks.
2. Understood how **Docker Volumes** allow databases to keep their data even after containers stop.
3. Grasped how **Docker Networks** (like blog-network) allow microservices to discover each other without using localhost.
4. Learned the difference between ClusterIP, NodePort, and LoadBalancer service types, and when to use each depending on whether we want internal communication or public access.
5. Learned how to simulate a Kubernetes cluster locally using Minikube and expose services for browser access with minikube service <service-name>.

Challenges Faced:

1. **Networking Confusion:**
   Initially, it was confusing why localhost doesn't work inside Kubernetes Pods and why service names are needed instead.
2. **CrashLoopBackOff Errors:**
   Some Pods went into CrashLoopBackOff due to service discovery issues or environment variables being incorrectly set initially.
3. **Expose Gateway to Browser:**
   It was tricky to expose the gateway service properly using minikube service, understanding why ClusterIP wouldn't work directly for external access.

4. **Deployment vs Docker Compose:**
   Managing deployments and services in Kubernetes is much more detailed compared to Docker Compose — requiring more YAML files and configuration understanding.