CEC Assignment

Building and Deploying Microservices using Docker/Kubernetes

Group Members:

1. IIT2022004 - Vatsal Bhuva
2. IIT2022012 - Keshan Lohani
3. IIT2022035 - Devam Desai
4. IIT2022037 - Darshan Vanjara
5. IIT2022073 - Nitu Sherawat
6. IIB2022015 - Harshit Raj

About the Project

This is a microservices-based e-commerce backend application designed with modularity and scalability in mind. Each service handles a distinct responsibility—user authentication, product management, data storage, and caching. Services communicate over HTTP and utilize Docker for containerized deployment. Redis is used for caching frequently accessed product data, and MongoDB serves as the main database for both user and product data.

Directory Structure

```
ecommerce/
│
├── docker-compose.yml          # Orchestration of all services
├── .gitignore                  # Avoid pushing unnecessary files
├── authentication          # For login, registration, authorization
│   ├── db.js
│   ├── index.js
│   ├── userModel.js
│   ├── Dockerfile
│   ├── package.json
│   └── .env
│
├── product                  # Product service for CRUD operations
│   ├── db.js
│   ├── index.js
│   ├── productModel.js
│   ├── redis.js          # Redis cache service for fast access
│   ├── Dockerfile
│   ├── package.json
│   └── .env
│
└── volumes/
    └── mongo-data/      # Volume for MongoDB data persistence
├── k8s/
│   ├── auth-hpa.yaml       # Scaler for auth-service
│   ├── auth-service.yaml # Deployment and Service for auth
│   ├── ingress.yaml         # nginx ingress
│   ├── mongodb.yaml      # Deployment and Service for db
│   ├── namespace.yaml  # ecommerce namespace definition
```

```
│   ├── product-hpa.yaml  # Scaler for auth-service
│   ├── product-service.yaml # Deployment & Service for products
│   └── redis.yaml     # Deployment & Service for cache
```

Service Breakdown (APIs and Usage)

1. Authentication Service (`auth-service`)

Path: `./authentication`

Port: `3000`

Purpose: Handles user-related operations such as registration, login, JWT-based authentication, and role-based access control.

Endpoints:

- **`POST /register`**: Registers a new user with roles like `user`, `admin`, or `employee`.
- **`POST /login`**: Authenticates a user and returns a JWT
- **`POST /verify`**: Verifies the JWT and returns user info
- **`POST /verifyRole`**: Verifies if the user has the required role(s) to access certain protected resources.

Technologies Used:

- MongoDB (User storage)
- bcrypt (Password hashing)
- JWT (Authentication)

- Express.js (Web framework)

---

2. Product Service (`product-service`)

Path: `./product`
Port: `3001`
Purpose: Manages product-related operations with support for pagination, creation, updating, and deletion. Integrates caching for performance optimization.

Endpoints:

- **`GET /products`**: Retrieves paginated product list, with Redis caching
- **`POST /product`**: Adds a new product (admin or employee only
- **`PUT /update/:id`**: Updates an existing product (admin or employee only
- **`DELETE /delete/:id`**: Deletes a product by ID (admin or employee only).

Middleware:

- **`verifyRole`**: Sends a request to `auth-service` to verify the user's role before allowing protected operations.

Technologies Used:

- MongoDB (Product data storage)

- Redis (Product list caching)
- Axios (Inter-service communication with auth)
- Express.js (Web framework)

---

3. Database Service (`db-service`)

Image: `mongo`
Port: `27017`
Purpose: Provides a centralized MongoDB instance for both `auth-service` and `product-service`.

Data Persistence:

- Uses Docker volume `mongo-data` to persist database files across container restarts.

---

4. Cache Service (`cache-service`)

Image: `redis`
Port: `6379`
Purpose: Caches product listings to reduce MongoDB read load and speed up response time for repeated requests.

Used In:

- `product-service`, specifically in `GET /products` endpoint.

- Caches products fetched for a specific page and a specific limit (like page=1&limit=10). Expiry is set to 60 seconds to avoid the data from becoming very stale.

Docker Services

Docker Compose Overview (`docker-compose.yml`)

Docker Compose is used to define and run multiple services as containers. This file brings together the four main services: `auth-service`, `product-service`, `db-service` (MongoDB), and `cache-service` (Redis). It also sets up shared volumes and handles inter-service dependencies.

Service: `auth-service`

- **Context:** Built from `./authentication` using its Dockerfile
- **Port Mapping:** Host port `3000` maps to container port `3000`
- **Environment:** `.env` file is loaded from the `authentication` directory
- **Volume Mounting:** The host directory `./authentication` is mounted inside the container, allowing live code changes without needing to rebuild the image.
- **Dependency:** Depends on `db-service`, ensuring MongoDB starts first.

Service: `product-service`

- **Context:** Built from ./product.
- **Port Mapping:** Host port 3001 maps to container port 3001.
- **Environment:** Loads variables from ./product/.env.
- **Volume Mounting:** Live mounts the host ./product folder for real-time code updates.
- **Dependencies:** Waits for cache-service (Redis) and db-service (MongoDB) to be ready before starting.

Service: `db-service` (MongoDB)

- **Image:** Uses the official mongo image.
- **Port Mapping:** Exposes default MongoDB port 27017
- **Volume:** Uses a named volume mongo-data to persist data, so that container restarts don't lose the database.

Service: `cache-service` (Redis)

- **Image:** Uses the official redis image.
- **Port Mapping:** Exposes Redis default port 6379.

Kubernetes Setup

1. `namespace.yaml`

Purpose:
Defines a custom namespace `ecommerce` in which all other

resources will be deployed. Namespaces logically isolate Kubernetes resources within the cluster.

---

2. `auth-service.yaml`

Purpose:
Combines the **Deployment** and **Service** for the `auth-service`.

Deployment:

- Runs multiple replicas of the auth microservice (`auth-service` Docker image
- Includes `env` variables (`PORT`, `JWT_SECRET`
- Mounted under the `ecommerce` namespace
- Ensures restart and rescheduling on failure.

Service:

- Exposes the auth pods internally using a ClusterIP service
- Used by other services (like product-service) to reach authentication API by hostname.

---

3. `auth-hpa.yaml`

Purpose:
Horizontal Pod Autoscaler for the `auth-service`.

What it does:

- Automatically scales the number of replicas based on CPU usage.
- Keeps the number of pods between a defined min/max range (e.g., 1–5).
- Requires resource limits to be set in the deployment for autoscaling to work.

---

4. `product-service.yaml`

Purpose:
Defines **Deployment** and **Service** for the product microservice.

🔧 **Deployment:**

- Launches the `product-service` container.
- Injects environment variables like `PORT` and `REDIS_HOST`.
- Handles REST APIs related to product CRUD.

Service:

- Exposes the product pods on a specific port inside the cluster.
- Used by the Ingress controller to reach the product service.

---

5. `product-hpa.yaml`

Purpose:
Scales the number of product-service pods horizontally.

Key Features:

- Target CPU utilization defined (e.g., 80%).
- Min/Max replica count.
- Keeps the product service performant under varying load.

---

6. `mongodb.yaml`

Purpose:
Defines **Deployment** and **Service** for MongoDB.

Deployment:

- Pulls the official `mongo` image.
- Runs a single replica with persistent storage (optional: PVC not included here).
- Starts MongoDB for both `auth` and `product` services to connect.

Service:

- ClusterIP service on port `27017` to allow internal access.

---

7. `redis.yaml`

Purpose:
Defines **Deployment** and **Service** for Redis caching server.

Deployment:

- Run the official `redis` image.
- Provides in-memory data store functionality for the product service.

Service:

- Internal ClusterIP service on port `6379`.

---

8. `ingress.yaml`

Purpose:
Ingress resource for routing external traffic to internal services (`auth-service`, `product-service`).

What it does:

- Maps hostnames and paths (e.g., `/auth`, `/products`) to respective services.
- Uses NGINX Ingress Controller provided by Minikube or Kubernetes.

- Optionally supports TLS for HTTPS.

Example Rules:

- `/auth` → forwards traffic to auth-service on port 3000.
- `/products` → forwards traffic to product-service on port 3001.