Assignment (Expected Time to Complete - 24 hours Or 3 Days)

Building a Professional E-Commerce Platform with Spring Boot

Objective:

The goal of this assignment is to develop an **e-commerce platform** using **Spring Boot**. The platform will include the core functionality of managing products, processing orders, and securing the application with proper authentication and authorization. This assignment is designed to simulate a **real-world project** and test your ability to build scalable, modular, and secure microservices.

What You Will Build:

You will develop a **multi-module Spring Boot application** consisting of three microservices:

1. **Product Service**: Manages the product catalog, including features like adding products, updating product details, and retrieving product lists with pagination and sorting.
2. **Order Service**: Manages the ordering process by integrating with the Product Service to fetch product details and update stock upon order placement.
3. **User Service**: Provides authentication and authorization, using JWT-based security to restrict access based on user roles (ADMIN and USER).

Each service will be deployed independently and communicate with one another using **REST APIs** and **messaging queues** (RabbitMQ or Kafka).

Core Features:

1. **Product Management**:
    - Add, view, update, and delete products.
    - Implement input validation, pagination, and sorting.
    - Protect endpoints to allow only authorized access.
2. **Order Processing**:
    - Place an order by fetching product details from the Product Service.
    - Deduct stock from products when an order is placed.
    - Use messaging to ensure consistency between services.
3. **User Authentication and Authorization**:
    - Implement role-based access control using JWT.

    - Secure APIs to restrict certain actions to ADMIN users only.
4. **Microservices Architecture**:
    - Modular and decoupled services with clear responsibilities.
    - Communication between services via REST APIs and messaging queues.
5. **Deployment**:
    - Use **Docker Compose** to containerize and deploy all services.
    - Provide a unified deployment setup for seamless integration.

How to Approach the Assignment:

This assignment is divided into the following parts:

1. **Part 1: Product Service**:
    - Build and test all product-related functionality.
    - Focus on input validation, pagination, and sorting.
2. **Part 2: User Service**:
    - Implement JWT-based authentication and authorization.
    - Secure endpoints to allow role-based access control.
3. **Part 3: Order Service**:
    - Integrate with Product Service to fetch product details and update stock during order placement.
    - Use RabbitMQ or Kafka for messaging.
4. **Part 4: Deployment**:
    - Write a `docker-compose.yml` file to containerize and deploy all services.
    - Document the deployment steps in the README.

Each part will include detailed tasks, instructions, and expected inputs/outputs for the APIs.

Grading Breakdown

Topic	Percentage Weight	Description
1. Spring Boot Basics	10%	- Proper project structure. - Configuration using `application.properties`. - Dependency setup (Spring Web, Data JPA, etc.).
2. RESTful API Development	20%	- Correct implementation of CRUD operations. - Proper usage of HTTP methods and status codes. - Pagination and sorting in Product Service.

3. Data Access with Spring JPA	15%	- Correct use of Spring Data JPA for database operations. - Proper entity mapping and relationships. - Validations using annotations like `@NotNull` and `@Positive`.
4. Spring Boot Security	20%	- Implementation of JWT-based authentication. - Role-based access control (`ADMIN`, `USER`). - Secure endpoints with proper error handling (`401`, `403`).
5. Spring Boot Messaging	15%	- Integration with RabbitMQ or Kafka. - Publishing and consuming messages. - Synchronization of services using messaging.
6. Spring Boot Configuration	10%	- Use of profiles for environment-specific configurations. - Proper externalization of secrets (e.g., JWT keys).
7. Testing	5%	- Basic API testing using tools like Postman. - Unit tests or integration tests for key components (if applicable).
8. Deployment and Docker	5%	- Containerization of services using Docker. - Correct use of `docker-compose` for deployment. - Environment variable management.

Part 1: Product Service

Objective:

The **Product Service** is responsible for managing the product catalog. You will build a standalone Spring Boot service that supports CRUD operations for products, input validation, pagination, and sorting. This service will form the foundation for the e-commerce platform.

Tasks:

1. **Set Up the Service**:

- Create a Maven project for `product-service`.
- Add the following dependencies:
  - `Spring Web`
  - `Spring Data JPA`
  - `H2 Database`
  - `Spring Boot DevTools`
- Configure the `application.properties` file for an in-memory H2 database.

Example `application.properties`:

```
spring.datasource.url=jdbc:h2:mem:productdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.jpa.show-sql=true
```

2. **Define the Product Entity**:
   - Fields:
     - `id` (auto-generated, primary key)
     - `name` (non-null, unique)
     - `price` (non-negative)
     - `category` (enum: `ELECTRONICS`, `FASHION`, `GROCERY`, etc.)
     - `stock` (non-negative)
   - Use validation annotations like `@NotNull`, `@Positive`, and `@Size` where applicable.
3. **Create CRUD APIs**:
   - Add, view, update, and delete products.
   - Include pagination and sorting in the retrieval APIs.
4. **Organize the Code**:
   - Separate the application into layers:
     - **Controller**: Handles HTTP requests.
     - **Service**: Contains business logic.
     - **Repository**: Handles database interaction.
5. **Test the APIs**:
   - Use Postman or cURL to test each API.
   - Ensure proper status codes and error handling.

APIs with Input and Output:

1. Add Product

- **Endpoint**: `POST /products`
- **Description**: Adds a new product to the catalog.

Input:

```
{
    "name": "Laptop",
    "price": 1200.00,
    "category": "ELECTRONICS",
    "stock": 50
}
```

- **Output (Success)**:
    - Status: `201 Created`

```
{
    "id": 1,
    "name": "Laptop",
    "price": 1200.00,
    "category": "ELECTRONICS",
    "stock": 50
  }
```

- **Output (Failure)**:
    - Status: `400 Bad Request`

```
{
    "error": "Validation failed: Name must not be null, Price must be non-negative."
  }
```

2. Fetch All Products

- **Endpoint**: `GET /products`
- **Description**: Retrieves a paginated and sorted list of all products.
- **Input** (Query Parameters):
    - `page` (optional): Default is `0`.
    - `size` (optional): Default is `10`.
    - `sort` (optional): Format `{field,order}` (e.g., `price,asc`).
- **Output**:
    - Status: `200 OK`


```
{
    "products": [
      {
        "id": 1,
        "name": "Laptop",
        "price": 1200.00,
        "category": "ELECTRONICS",
        "stock": 50
      },
      {
        "id": 2,
        "name": "Phone",
        "price": 800.00,
        "category": "ELECTRONICS",
        "stock": 100
      }
    ],
    "page": 0,
    "size": 10,
    "totalElements": 2,
    "totalPages": 1
  }
```


3. Fetch Product by ID

- **Endpoint**: `GET /products/{id}`
- **Description**: Fetches product details by its ID.
- **Input**: `{id}` (Path Parameter)
- **Output (Success)**:
    - Status: `200 OK`

```
{
    "id": 1,
    "name": "Laptop",
    "price": 1200.00,
    "category": "ELECTRONICS",
    "stock": 50
  }
```


- **Output (Failure)**:
    - Status: `404 Not Found`


```
{
    "error": "Product with ID 10 not found."
  }
```


4. Update Product

- **Endpoint**: `PUT /products/{id}`
- **Description**: Updates the details of an existing product.
- **Input**:
    - `{id}` (Path Parameter)

Request Body:


```
{
    "name": "Laptop Pro",
    "price": 1500.00,
    "category": "ELECTRONICS",
    "stock": 30
  }
```


- **Output (Success)**:

- Status: `200 OK`

```
{
    "id": 1,
    "name": "Laptop Pro",
    "price": 1500.00,
    "category": "ELECTRONICS",
    "stock": 30
  }
```

- **Output (Failure)**:
  - Status: `404 Not Found`

Status: `400 Bad Request`

```
{
    "error": "Validation failed: Stock must be non-negative."
  }
```

5. Delete Product

- **Endpoint**: `DELETE /products/{id}`
- **Description**: Deletes a product from the catalog.
- **Input**: `{id}` (Path Parameter)
- **Output (Success)**:
  - Status: `204 No Content`
- **Output (Failure)**:
  - Status: `404 Not Found`

```
{
    "error": "Product with ID 10 not found."
  }
```

Part 2: User Service

Objective:

The **User Service** manages user authentication and authorization. It uses **JWT-based security** to restrict access to APIs based on user roles (`ADMIN` and `USER`), ensuring only authorized users can perform specific actions.

Tasks:

1. **Set Up the User Service**:
    - Create a new Maven project named `user-service`.
    - Add dependencies:
        - Spring Web
        - Spring Security
        - JWT Library (e.g., `jjwt` or another suitable library).
    - Configure the `application.properties` to include:
        - A secret key for signing JWTs.
        - Token expiration duration (e.g., 1 hour).
2. **Define Roles and Users**:
    - Use an **in-memory user store** for this assignment.
        - `admin` (Role: `ADMIN`, Password: `admin123`)
        - `user` (Role: `USER`, Password: `user123`)
3. **Implement Login Functionality**:
    - Create a `POST /login` endpoint that:
        - Accepts username and password in the request.
        - Validates the credentials.
        - Generates and returns a JWT token on successful authentication.
4. **Secure the APIs**:
    - Configure Spring Security to:
        - Allow unauthenticated access to `/login`.
        - Require authentication for all other APIs.
    - Use **role-based access control**:
        - `ADMIN` can add, update, and delete products.
        - `USER` can only view products.
5. **Validate JWT Tokens**:
    - Intercept API requests to:
        - Extract the JWT token from the `Authorization` header.
        - Validate the token and extract the user's details.

- Restrict access if the token is invalid or the user does not have the required role.

APIs with Input and Output:

1. Login API

- **Endpoint**: `POST /login`
- **Description**: Authenticates the user and returns a JWT token.

Input:

```
{
  "username": "admin",
  "password": "admin123"
}
```

- **Output (Success)**:
  - Status: `200 OK`

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}
```

- **Output (Failure)**:
  - Status: `401 Unauthorized`

```
{
  "error": "Invalid username or password."
}
```

---

2. Secured APIs

- **Description**: All other APIs (e.g., `/products`, `/orders`) are secured and require a valid JWT token in the `Authorization` header.

Authorization Header:
makefile
CopyEdit
`Authorization: Bearer <token>`

- **Unauthorized Access**:
  - Status: `401 Unauthorized`

```
{
  "error": "Token is missing or invalid."
}
```

- **Forbidden Access (Role-Based Restriction)**:
  - Status: `403 Forbidden`

```
{
```

```
  "error": "Access denied. Insufficient permissions."
}
```

Role-Based Access Control:

- Role assignments determine access:
  - `ADMIN`:
    - Can access all APIs (add, update, delete, and view products).
  - `USER`:
    - Limited to read-only access (e.g., `GET /products`).
- This ensures APIs are secure and actions are restricted to authorized roles.

Part 3: Order Service

Objective:

The **Order Service** manages the ordering process by integrating with the Product Service. It handles operations such as placing an order, fetching order details, and ensuring stock is updated in the Product Service. This service will communicate with the Product Service via **REST APIs** and use **messaging queues** to synchronize stock updates.

Tasks:

1. **Set Up the Order Service**:
   - Create a Maven project named `order-service`.
   - Add dependencies:
     - Spring Web
     - Spring Data JPA
     - H2 Database
     - Spring Boot DevTools
     - RabbitMQ or Kafka (for messaging).

    - Configure `application.properties` to set up an H2 database for order records.
2. **Define the Order Entity**:
    - Fields:
        - `orderId`: Auto-generated primary key.
        - `productId`: Foreign key referring to a product in the Product Service.
        - `quantity`: Number of items ordered.
        - `totalPrice`: Calculated as `quantity × product price`.
        - `orderDate`: Timestamp of the order.
    - Ensure proper validation for fields like `quantity` (non-negative).
3. **Integrate with Product Service**:
    - Use REST APIs from the Product Service to:
        - Fetch product details during order placement.
        - Update stock when an order is placed.
    - Handle errors (e.g., product not found, insufficient stock) gracefully.
4. **Implement Messaging**:
    - Publish an event to a messaging queue (RabbitMQ or Kafka) when an order is placed.
    - Consume the event in the Product Service to synchronize stock updates.
5. **Create APIs**:
    - Place an order.
    - Fetch all orders.
    - Fetch order by ID.
6. **Test the APIs**:
    - Use Postman or similar tools to:
        - Place orders.
        - Fetch order details.
        - Verify stock updates in the Product Service.

APIs with Input and Output:

1. Place Order

- **Endpoint**: `POST /orders`
- **Description**: Places a new order and updates the stock in the Product Service.

Input:

```
{
  "productId": 1,
  "quantity": 2
}
```

- **Output (Success)**:
    - Status: `201 Created`

```
{
  "orderId": 1,
  "productId": 1,
  "quantity": 2,
  "totalPrice": 2400.00,
  "orderDate": "2025-01-24T12:00:00Z",
  "status": "Order Placed"
}
```

- **Output (Failure)**:
    - Status: `404 Not Found` (if product does not exist in Product Service).

```
{
  "error": "Product with ID 10 not found."
}
```

- Status: `400 Bad Request` (if insufficient stock).

```
{
  "error": "Insufficient stock for product ID 1."
}
```

---

2. Fetch All Orders

- **Endpoint**: `GET /orders`
- **Description**: Retrieves a list of all orders.
- **Output**:
  - Status: `200 OK`

```
[
  {
    "orderId": 1,
    "productId": 1,
    "quantity": 2,
```

```
    "totalPrice": 2400.00,
    "orderDate": "2025-01-24T12:00:00Z",
    "status": "Order Placed"
  },
  {
    "orderId": 2,
    "productId": 2,
    "quantity": 1,
    "totalPrice": 800.00,
    "orderDate": "2025-01-24T14:00:00Z",
    "status": "Order Placed"
  }
]
```


---

3. Fetch Order by ID

- **Endpoint**: `GET /orders/{id}`
- **Description**: Retrieves details of an order by its ID.
- **Input**: `{id}` (Path Parameter)
- **Output (Success)**:
    - Status: `200 OK`

```
{
  "orderId": 1,
  "productId": 1,
  "quantity": 2,
  "totalPrice": 2400.00,
  "orderDate": "2025-01-24T12:00:00Z",
  "status": "Order Placed"
}
```

- **Output (Failure)**:
    - Status: `404 Not Found`

```
{
  "error": "Order with ID 10 not found."
}
```

-

Integration Details:

1. **Communication with Product Service**:
    - Use REST APIs to:
        - Fetch product details (e.g., `GET /products/{id}`).

        - Update stock after an order is placed.
2. **Messaging**:
    - Publish an event (e.g., `OrderPlacedEvent`) when an order is successfully created.
    - The event should include:
        - `orderId`, `productId`, `quantity`, `status`.

Example Event:

```
{
  "orderId": 1,
  "productId": 1,
  "quantity": 2,
  "status": "Order Placed"
}
```

    - Consume the event in the Product Service to update stock.

Part 4: Deployment

Objective:

Deploy the e-commerce platform with all three microservices (`product-service`, `order-service`, and `user-service`) using **Docker Compose**. This part will ensure that the services run independently but communicate seamlessly in a containerized environment.

Tasks:

1. **Containerize Each Service**:
    - Create a `Dockerfile` for each service to build a Docker image.
    - Add the necessary dependencies to the image and ensure the services start correctly.
2. **Set Up a `docker-compose.yml` File**:
    - Use **Docker Compose** to:
        - Run all three services in separate containers.
        - Set up RabbitMQ/Kafka as the messaging queue.
        - Define network communication between the services.
3. **Environment Variables**:
    - Use `environment` properties in the `docker-compose.yml` file to configure:
        - Database connections.
        - JWT secret keys.
        - Service URLs (e.g., `PRODUCT_SERVICE_URL`).
4. **Volume Management**:
    - Use Docker volumes to persist data for services like RabbitMQ and databases.
5. **Test the Deployment**:
    - Use `docker-compose up` to start the containers.
    - Verify that all services are running and communicating properly.
    - Test APIs using Postman.

Deliverables:

1. **Dockerfile** for Each Service:

Example for `product-service`:

```
FROM openjdk:17-jdk-slim
ARG JAR_FILE=target/product-service-0.0.1-SNAPSHOT.jar
COPY ${JAR_FILE} app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

    - Repeat for `order-service` and `user-service`.
2. **docker-compose.yml**:
    - Define services for `product-service`, `order-service`, and `user-service`.
    - Include RabbitMQ/Kafka for messaging.

Example:

```yaml
version: "3.8"
services:
  product-service:
    build:
      context: ./product-service
    ports:
      - "8081:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:h2:mem:productdb
      - JWT_SECRET_KEY=yourSecretKey
    depends_on:
      - rabbitmq

  order-service:
    build:
      context: ./order-service
    ports:
      - "8082:8080"
    environment:
      - PRODUCT_SERVICE_URL=http://product-service:8081
      - RABBITMQ_HOST=rabbitmq
    depends_on:
      - product-service
      - rabbitmq

  user-service:
    build:
      context: ./user-service
    ports:
      - "8083:8080"
    environment:
      - JWT_SECRET_KEY=yourSecretKey
    depends_on:
      - rabbitmq

  rabbitmq:
    image: rabbitmq:management
    ports:
      - "15672:15672"
```

```
      - "5672:5672"
```

3. **Database Configuration**:
    - Ensure each service uses an H2 or in-memory database for simplicity, or use Docker volumes to persist data if needed.
4. **Networking**:
    - Use the default Docker Compose network to enable communication between services.

Testing Steps:

1. **Build and Run Services**:

Navigate to the project directory and run:

```
docker-compose up --build
```

2. **Verify Services**:
    - Use the following URLs to verify services:
        - `product-service`: `http://localhost:8081`
        - `order-service`: `http://localhost:8082`
        - `user-service`: `http://localhost:8083`
    - Access RabbitMQ management console: `http://localhost:15672`.
3. **Test APIs**:
    - Use Postman to test all APIs:
        - Product Service APIs (CRUD operations).
        - User Service APIs (login and access control).
        - Order Service APIs (placing and fetching orders).
    - Verify that:
        - Orders placed in `order-service` update stock in `product-service`.
        - JWT authentication works for secured APIs.
4. **Messaging Verification**:
    - Log in to RabbitMQ management console:
        - Username: `guest`
        - Password: `guest`
    - Verify message queues are created and events are processed.