ByteBites Microservices Platform - Code Review Report

Executive Summary

The ByteBites project is well-structured and module providing separation of concerns. The architecture demonstrates good understanding of microservices patterns, with proper separation of concerns, security implementation, and event-driven communication. However, there are several areas that i think could be improve to make the project more robust such as containerization, and monitoring on some services

Overall Assessment

Strengths:

- Clean microservices architecture with proper service boundaries and separation of concerns.
- The implementation of JWT and OAuth2 was comprehensive and well implemented for better security.
- Good use of Spring Cloud components such as the spring gateway,spring eureka, and config server.
- For Event-driven communication, RabbitMQ was used which was implemented well.
- Proper resource ownership and RBAC implementation was made allowing restaurant owners to only access restaurants their restaurants related materails and customers only see their orders and stuff.

Areas for Improvement:

- Missing Docker containerization: The project lacked a Docker containerization which is crucial for the deployment and robustness of the use of the application
- Inconsistent Spring Boot versions across services: The version of dependencies in the project weren't consistent for different services on the project.
- Limited error handling and validation: Though there was error handling and validation on some services and inputs, the validation wasn't consistent and some use
- Missing monitoring and observability for some services: Some services s
- No distributed tracing implementation

---

1. Microservice Design and Structure

Strengths

Service Decomposition:

Well-defined service boundaries (Auth, Restaurant, Order, Discovery) Each service has a clear, single responsibility with proper separation of business logic across services The project isn't a "distributed monolith." Each service has a clear job, which is the 1st rule of microservices. This makes it easier to manage, scale, and update individual parts of the application without breaking everything

Service Independence:

Services can be developed and deployed independently as each service has its own database.

Issues and Recommendations

1.1 Inconsistent Service Registration

```
 # restaurant-service/application.yml - Line 15-19
#  cloud:
#    config:
#      discovery:
#        enabled: true
#        service-id: config-server
```

Restaurant service has commented-out config server configuration, meaning it is using local profile configuration.

Recommendation: Standardize configuration across all services to maintain consistency over the services for easy deployment and avoid issues that might results when shipping to production or other deployment environment.

1.2 Missing Service Health Checks No health check endpoints implemented for restaurant service and authservice.

Recommendation: Add Spring Boot Actuator to all services for health monitoring especially restaurant and auth service.

---

2. Spring Boot and Spring Cloud Usage

Strengths

Spring Cloud Components:

- Proper use of Eureka for service registry and discovery
- Spring Cloud Gateway for API routing and authenication.
- Spring Cloud Stream for messaging

Spring Boot Features:

- Good use of Spring Data JPA, proper security configuration and Lombok for reducing boilerplate

Issues and Recommendations

2.1 Version Inconsistencies

```
<!-- api-gateway/pom.xml - Line 6 -->
<version>3.3.13</version>
       <!-- Other services use 3.5.3 -->
```

- API Gateway uses Spring Boot 3.3.13 while other services use 3.5.3
- **Recommendation:** Standardize Spring Boot version across all services will reduces issues of inconsistent independence conflict.

2.2 Missing Resilience Patterns

- Only order-service has Resilience4j circuit breaker for resilence
- **Recommendation:** Implement circuit breakers for all service-service communication to ensure there are not cascading failure of services.

2.3 Incomplete Spring Cloud Stream Configuration

```
# order-service/application.yml - Lines 18-25
spring:
  cloud:
    stream:
      bindings:
        destination: order-events
        contentType: application/json
```

- Stream configuration is inconsistent and incomplete over some services
- **Recommendation:** Complete the Spring Cloud Stream configuration for proper event handling

---

3. Security Implementation

Strengths

JWT Implementation: Proper RSA key-based JWT signing, JWT validation in API Gateway and Role-based access control (RBAC)

OAuth2 Authorization Server: Well-implemented OAuth2 authorization server in auth-service with proper JWT endpoint for token validation Secure token generation and validation for Oauth2 users

Resource Ownership: Proper implementation of resource ownership checks. Users can only access their own data

Issues and Recommendations

3.1 Hardcoded RSA Keys

```
# auth-service/application.yml - Lines 25-26
jwt:
  rsa:
    private-key: MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQC87F/mNwChRG7XJZ7B13tmMWuowSPUAlGefCSFAsNEFimiq4LFlto7KMQJLRNCLHo
```

RSA keys are hardcoded in the application.yml which is not secured

Recommendation: Using environment variables (.env) or external secret management will be a better way of maintaining privacy and integrity.

3.2 Missing Input Validation Some endpoints do not have input validation, such as in the authcontroller of the authservice.

Recommendation: Add comprehensive input validation using Bean Validation for the request coming in.

---

4. Messaging Integration

Strengths

Event-Driven Architecture:

Proper use of Spring Cloud Stream with RabbitMQ that implements an orderPlacedEvent, decoupling services and promoting asynchronous communication patterns

Issues and Recommendations

4.1 Incomplete Event Handling

```
// order-service/src/main/java/com/bytebites/orderservice/service/OrderService.java - Line 45
streamBridge.send("orderPlacedOutput", new OrderPlacedEvent(this, savedOrder.getId(), savedOrder.getCustomerId(), savedOrder.getRest
```

Events are published but no consumer is implemented to consume them on the notification service.

Recommendation: Implement event consumers for notification service

4.2 Missing Error Handling for Messaging

No dead letter queue configuration and retry mechanisms for failed message processing were implemented to better handle situations of failure.

Recommendation: Implement proper error handling and retry logic for robust messaging.

4.3 Inconsistent Messaging Dependencies

```
<!-- order-service/pom.xml - Lines 45-65 -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-stream-binder-rabbit</artifactId>
</dependency>
<dependency>
<groupId>org.apache.kafka</groupId>
<artifactId>kafka-streams</artifactId>
</dependency>
```

Both RabbitMQ and Kafka dependencies are included but only rabbi

Recommendation: Choose one messaging platform and remove unused dependencies

---

5. Service Discovery and Configuration

Strengths

Eureka Implementation:

Proper service registration and discovery which allow all services to register with the Eureka server with load balancing through service names

---

6. Docker and Deployment Setup

Critical Missing Components

6.1 No Docker Implementation

No Dockerfiles for individual services.A missing docker-compose.yml for local development and container orchestration setup

Recommendations:

1. Create Dockerfiles for each service
2. Implement docker-compose.yml for local development
3. Implement multi-stage builds for optimized images to reduce size and fast deplomyent

---

7. Data Management and Persistence

Strengths

Database Design:

Entities were well defined and relationships properly implemented. Use of JPA annotations with automatic generation of UUID and auto-increment IDs appropriately used

Issues and Recommendations

7.1 Development-Only Database

```
# All services use H2 in-memory database
datasource:
  url: jdbc:h2:mem:authdb
```

All services use H2 in-memory database for storage and management of data for the services

Recommendation: Implement PostgreSQL or MySQL for production

7.2 Missing Database Migrations No Flyway or Liquibase migrations for proper management of database schema changes.

Recommendation: Implement database migration tool like fly way to better manage database schema changes by validation.

Recommendation: Use records for dtos for clean, immutable design with less boilerplate data carriers.

---

8. Testing and Quality Assurance

Areas for Improvement

8.1 Limited Test Coverage

- Only basic test classes exist, no integration tests or end-to-end tests

Recommendations:

1. Add comprehensive unit tests for each service
2. Implement integration tests with TestContainers

---

9. Security Vulnerabilities

Identified Issues

10.1 Exposed H2 Console

```
# auth-service/application.yml - Lines 20-22
h2:
  console:
    enabled: true
    path: /h2-console
```

H2 console is enabled in production configuration though it's the only implemented database been used.

Recommendation: Disable in h2 console for production profile or secure with authentication

10.2 Missing Rate Limiting

No rate limiting on API endpoints means endpoints can be overloaded with request.**Recommendation:** Implement rate limiting with Spring Cloud Gateway

10.3 No CORS Configuration

No CORS policy defined for external interactions.

Recommendation: Implement proper CORS configuration for integration.

Recommendations

High Priority (Production Blockers)

1. **Implement Docker containerization**
2. **Add comprehensive error handling**
3. **Add health checks and monitoring**
4. **Fix version inconsistencies**
5. **Add comprehensive testing**
6. **Implement proper database migrations**
7. **Implement Resilience by adding circuit breakers everywhere they are needed**