Code Review Report Of Elias Adjetey

Reviewed by: Michael Okyere Adams

Section 1: Microservice Design and Structure

1.1 Executive Summary of Architectural Approach:

- **Overall Assessment:** The project broadly follows a microservices architecture, with services exhibiting varying degrees of independence and cohesion.
- **Number of Services Identified:** Seven(7) distinct microservices were identified.
- **Repository Structure:** The application does **not** utilize a monorepo structure, with each service residing in its own separate directory/repository. A general `pom.xml` file is absent, indicating individual project management for each microservice.

1.2 Service Granularity and Bounded Contexts:

- **Findings:**
  - **Well-Defined Services:** The `OrderService` is well-scoped, handling order lifecycle events within its own distinct domain.
  - **Potential "God Services" / Over-Scoped:** The `AuthServiceImpl` Handles both registration and login logic, including user existence checks, password encoding, role assignment, authentication, and JWT generation. While not currently too large, if more authentication-related features (e.g., password reset, account verification) are added, it could become a "God service." Consider splitting responsibilities if it grows further.

- **Recommendations:** Monitor services like AuthServiceImpl to ensure they do not accumulate too many responsibilities. If authentication, registration, password management, and user profile management grow complex, split them into separate services (e.g., RegistrationService, PasswordService), consider putting duplicated codes like the JwtValidationFilter in a shared library.

1.3 Service Cohesion and Responsibility:

- **Findings:**
  - **High Cohesion:** The User entity demonstrates high cohesion. All its fields and methods directly relate to representing and managing user authentication data (ID, email, password, role). This focused responsibility ensures the class is easy to maintain and understand, aligning well with the principles of high cohesion

1.4 Independence (Deployment and Development):

- **Findings:**
    - **Deployment Independence:** Each service has its own Dockerfile and can be built and deployed independently, indicating good deployment independence.

1.5 Communication Patterns:

- **Findings:**
    - **Synchronous Communication:** No use of `RestTemplate` (or `WebClient`/Feign) for inter-service communication observed.
    - **Asynchronous Communication:** RabitMQ is utilized for event-driven communication, specifically for `NotificationService` to consume `OrderCreatedEvent` from `OrderService`.
    - **Database Sharing:** No direct database sharing observed between distinct services, which is a positive.
- **Recommendations:** Use WebClient to communicate between services like to check if the restaurant passed when making the order actually exists.

1.6 Service Boundaries and API Design:

- **Findings:**
    - **API Consistency:** APIs generally follow RESTful principles with clear resource naming.
    - **Versioning:** APIs use URI versioning (e.g., `/api/v1/orders`).
    - **Documentation:** Swagger/OpenAPI documentation is integrated with most services and is up-to-date, providing clear API contracts.

1.7 Error Handling and Resilience:

- **Findings:**
    - **Circuit Breakers:**Resilience4j is not implemented in any of the services leaving the system to cascading failures.
    - **Retries:** No retry logic is present for external API calls, once failed, the request is not going to retry.
    - **Fallbacks:** Since circuit breakers were not used, Fallbacks were also not defined.
- **Recommendations:** Implement consistent retry policies (with exponential backoff) for all critical synchronous inter-service calls.", "Implement circuit breaker to prevent cascading failures.

Section 2: Use of Spring Boot and Spring Cloud

2.1 General Spring Boot Usage:

- **Findings:**
    - **Project Initialization:** Standard Spring Boot project structure observed
    - **Configuration:** Effective use of `application.yml` for externalizing configuration.
    - **Actuator Endpoints:** Spring Boot Actuator is enabled across services, providing valuable health, metrics, and info endpoints.
    - **Testing:** Absence of unit and integration tests using Spring Boot Test.
- **Recommendations:** Write unit and integration tests to make sure that your application runs they way you want it to run.

2.2 Spring Cloud Component Usage:

2.2.1 Service Discovery (Eureka Server & Client):

- **Findings:**
    - **Eureka Server Setup:** The `Eureka service` is well-configured with appropriate use of the `@EnableEurekaServer` annotation.
    - **Configuration:** The `application.yml` file for Eureka is well-structured, clearly defining the server port and including monitoring metrics (likely via Actuator).
    - **Eureka Client Usage:** Services correctly use `@EnableDiscoveryClient` and register themselves with the Eureka server.

2.2.2 API Gateway (Spring Cloud Gateway):

- **Findings:**
    - **Custom Gateway Filter:** The API Gateway correctly implements a custom Gateway filter for JWT authentication, centralizing security concerns.
    - **Logging:** Both successful and failed authentication attempts are logged, which is beneficial for auditing and debugging.
    - **Public Endpoint Bypassing:** Public endpoints are correctly bypassed by the authentication filter, preventing unnecessary authentication checks.
    - **JWT Validation:** JWT validation correctly uses the secret key from configuration and leverages the `jjwt` library for parsing and validation.
    - **Exception Handling:** Exception handling is present, but current error responses are generic (no response body, only HTTP status code).
- **Recommendations:**
    - Consider adding more detailed error responses for better client feedback, including a consistent JSON error structure (e.g., `{ "timestamp": "...", "status": "...", "error": "...", "message": "...", "path": "..." }`).

Section 3: Security (JWT, RBAC, Resource Ownership)

3.1 Authentication (JWT):

- **Findings:**
  - **JWT Filter Placement:** In the `AUTH Service`'s `SecurityConfig`, the JWT filter is correctly added before the username/password filter, which is the standard practice for stateless JWT authentication.
  - **JWT Configuration:** The `JwtConfig` class uses `@ConfigurationProperties` for type-safe binding of JWT-related configuration properties (e.g., secret key, expiration time), which is a best practice for maintainability.
  - **JWT Validation (JwtService):** The `JwtService` in the `AUTH Service` handles token generation and validation.
  - **Gaps in JwtService:** No explicit exception handling for `invalid` or `expired` tokens is observed within the `validateToken` method of `JwtService`.
  - **Gaps in JwtAuthenticationFilter:** The `JwtAuthenticationFilter` also lacks explicit error handling for invalid tokens when setting the security context, which could lead to generic errors or unhandled exceptions rather than specific feedback.
- **Recommendations:**
  - Implement robust exception handling within `JwtService.validateToken()` to catch `ExpiredJwtException`, `MalformedJwtException`, `SignatureException`, etc., and throw specific, clear exceptions (e.g., `InvalidJwtTokenException`).
  - Enhance `JwtAuthenticationFilter` to catch these specific exceptions thrown by `JwtService` and provide a more informative error response to the client (e.g., "Token expired," "Invalid token signature").

3.2 Authorization (RBAC - Role-Based Access Control):

- **Findings:**
  - **Endpoint Authorization (SecurityConfig):** The `AUTH Service`'s `SecurityConfig` clearly defines endpoint authorization rules:
    - `/api/v1/auth/**` and `/actuator/**` are publicly accessible.
    - `/swagger-ui/**` is restricted to `ADMIN` role.
    - All other endpoints require authenticated access.
  - **Role Mapping:** Roles (`USER`, `ADMIN`, `RESTAURANT_OWNER`) are mapped to Spring Security authorities.
- **Recommendations:** Review the `/swagger-ui/**` access. While `ADMIN` only is secure, consider if `USER` or other roles might need read-only access for API exploration in non-production environments.

3.3 Resource Ownership:

- **Findings:**
    - **`getUserById` in `UserServiceImpl`:** The `getUserById` method in `UserServiceImpl` allows `RESTAURANT_OWNER` users to retrieve details of any user, including those not associated with their specific restaurant. This indicates a potential security vulnerability regarding resource ownership.
    - **Other Services:** The `OrderService` correctly restricts order access to the user who placed the order or an `ADMIN`.
- **Recommendations:**
    - **High Priority:** Modify `UserServiceImpl.getUserById()` to enforce proper resource ownership. A `RESTAURANT_OWNER` should only be able to view users directly associated with their restaurant (e.g., employees, customers who ordered from them), or, if the intent is for global user lookup by admins, clearly differentiate between admin and restaurant owner access.

3.4 General Security Practices:

- **Findings:**
    - **Request Validation:** The `AuthController` uses `@Valid` for request body validation, which is good practice for input sanitization and preventing malformed requests.
    - **Error Responses (AuthController):** `AuthController` returns `201 Created` for successful registration and `200 OK` for login, which is semantically correct. It throws `BadRequestException` for registration errors.
    - **Login Response (AuthService):** The `AuthService`'s login response only returns the token, username, and role.
    - **Logging (AuthService):** Logging is present for user registration attempts.
- **Recommendations:**
    - **Error Responses (AuthController):** While `BadRequestException` is functional, consider using Spring's `ResponseStatusException` for more consistent and flexible error response generation across the application. This allows for direct control over HTTP status codes and custom messages.
    - **Login Response (AuthService):** Enhance the login response from `AuthService` to include a `refresh token` (if refresh tokens are part of the security model) and the `expiration time` of the access token. This allows clients to manage token lifecycles more effectively without hardcoding expiration logic.
    - **Logging (AuthService):** Add logging for *all* login attempts (success and failure) in `AuthService.login()`. This is crucial for security auditing and detecting brute-force attacks.

Section 4: Messaging Integration (RabbitMQ)

4.1 RabbitMQ (Notification Service):

- **Findings:**
  - **`OrderPlacedEventHandler`:** Correctly configured to listen for `OrderPlacedEvent` messages from a specific RabbitMQ queue using `@RabbitListener`.
  - **Event Processing:** Logs the receipt of the event and attempts to send a confirmation email via `EmailService`.
  - **Exception Handling:** The exception handling in `OrderPlacedEventHandler` is broad (`Exception`), which is acceptable for general logging but could be refined to catch more specific error types (e.g., `MessagingException`, `MailSendException`) for more targeted recovery or notification.
  - **`RabbitMQConfig`:**
    - Uses `@Value` to inject queue and exchange names from application properties, promoting external configuration.
    - Sets up a `Jackson2JsonMessageConverter` for JSON serialization/deserialization, ensuring proper message payload handling.
    - Configures a `RabbitTemplate` with the custom message converter, facilitating message sending.
- **Recommendations:**
  - Refine exception handling in `OrderPlacedEventHandler` to catch more specific exceptions to enable more granular error logging or recovery strategies. Consider a Dead Letter Queue (DLQ) strategy for messages that consistently fail processing.

Section 5: Service Discovery and Configuration

5.1 Service Discovery (Eureka):

- **Findings:**
  - **Eureka Server Setup:** The `Eureka service` is well-configured with appropriate use of the `@EnableEurekaServer` annotation.
  - **Configuration:** The `application.yml` file for Eureka is well-structured, clearly defining the server port and including monitoring metrics (likely via Actuator).
  - **Eureka Client Usage:** Services correctly use `@EnableDiscoveryClient` and register themselves with the Eureka server.
- **Recommendations:**

    - Consider enabling peer awareness for Eureka servers in production for high availability.

5.2 Configuration Management (Spring Cloud Config / Local Properties):

- **Findings:**
    - **`application.yml`:** Widely used for service-specific configuration.
    - **Centralized Configuration:** Spring Cloud Config Server is used, externalizing most service configurations.

Section 6: Docker and Deployment Setup

6.1 Dockerization:

- **Findings:**
    - **Dockerfiles:** Each microservice includes a `Dockerfile` for containerization.
    - **Image Optimization:**Dockerfiles use multi-stage builds to create lean production images.
    - **Base Images:** Appropriate base images (e.g., `openjdk:17-jdk-slim`) are used.
- **Recommendations:** Ensure Dockerfiles respect security best practices, such as running as a non-root user.

6.2 Deployment Setup:

- **Findings:**
    - **Orchestration:** [e.g., "Deployment scripts (e.g., `docker-compose.yml`, Kubernetes manifests) are available."]
    - **CI/CD Integration:** [e.g., "Evidence of CI/CD pipelines (e.g., Jenkins, GitLab CI) for building and deploying services."]
- **Recommendations:** [e.g., "Review Kubernetes manifests for readiness/liveness probes, resource limits, and service mesh integration (if applicable)."]

Additional Findings / General Observations

README.md Documentation:

- **Findings:** The `README.md` file for the project is concise and provides an overview.
- **Gaps:** It **lacks a section explaining how to set up the application**, especially concerning the required environment variables. This will hinder new developer onboarding and local development.

- **Recommendations:**
    - Add a detailed "Setup Guide" or "Getting Started" section to the `README.md`.
    - Include clear instructions for setting up the local development environment, listing all necessary environment variables with examples or explanations.

DTO Consistency (Order Service):

- **Findings:** The `Order Service` lacks explicit DTOs for handling responses sent back to clients.
- **Implication:** This can lead to inconsistencies in API responses, expose internal entity structures, and make future API evolution more difficult.
- **Recommendations:**
    - Implement dedicated DTOs (Data Transfer Objects) for all API responses in the `Order Service`. This ensures consistency, allows for data transformation, and decouples the API contract from internal domain models.