Architecture and Infrastructure

The project demonstrates a strong understanding of microservice architecture. Core infrastructure components including the **Config Server**, **Discovery Server**, and **API Gateway** are well-configured and showcase a competent grasp of service orchestration and distributed system design.

The services are logically separated, and the integration of a **Git-backed Config Server** reflects a mature approach to centralized configuration management. Additionally, the use of **RabbitMQ** for asynchronous event-driven communication, particularly in the order-service, is commendable and fully meets the architectural requirement.

Identified Issues and Recommended Improvements

1. Dependency Injection Practice

- **Issue:** Field injection (`@Autowired`) is used across multiple services, including `order-service` and `restaurant-service`.

- **Recommendation:** Refactor to **constructor injection** to enhance testability, ensure immutability of dependencies, and improve clarity regarding required components.

2. Security Flaws

- **Hardcoded Secrets:**

    - **Issue:** The **JWT secret** is stored in plaintext in the `application.yml` file, which resides in a public repository.

    - **Recommendation:** Encrypt all sensitive values using the config server's `/encrypt` endpoint and store the ciphertext in the configuration files.

- **Admin Registration Endpoint:**

    - **Issue:** The `auth-service` exposes a public endpoint (`/api/v1/auth/register-admin`) that allows arbitrary creation of administrative accounts.

    - **Recommendation:** Restrict access to this endpoint to authenticated administrators or remove it entirely to eliminate the security risk.

- **Database Credentials:**
    - **Issue:** Database usernames and passwords are exposed in plain text within the public configuration repository.
    - **Recommendation:** Encrypt all sensitive configuration data using the config server's encryption capabilities.

3. Resilience and Fault Tolerance

- **Issue:** The system lacks a **circuit breaker** mechanism, making it vulnerable to cascading failures during service outages.
- **Recommendation:** Integrate **Resilience4j** and implement `@CircuitBreaker` annotations with fallback methods on all inter-service calls (e.g., from `order-service` to `restaurant-service`).

4. Transaction Management

- **Issue:** Several service methods that perform write operations to the database are not transactional.
- **Recommendation:** Annotate such methods with `@Transactional` to maintain data consistency and atomicity.

5. Input Validation

- **Issue:** There is insufficient input validation on DTOs, which could allow malformed or malicious data to propagate through the system.
- **Recommendation:** Apply validation annotations (e.g., `@NotBlank`, `@Email`) on DTO fields and enable validation in controllers using the `@Valid` annotation.

Strengths

- **Service Layer Abstraction:** The consistent use of **Data Transfer Objects (DTOs)** in the API layer is a highlight of the implementation, promoting clean separation of

concerns and encapsulation.

- **Event-Driven Design:** The RabbitMQ-based message queuing between services enhances decoupling and system scalability.

- **Documentation:** The project is well-documented, which greatly improves readability, maintainability, and onboarding for future contributors.

Conclusion

Overall, the project is a strong implementation of a microservices-based system. Despite a few critical flaws primarily in security and resilience the foundation is well-structured and shows a clear grasp of modern distributed system principles. By addressing the identified shortcomings, this project can be elevated to a production-grade application. Excellent progress and a commendable effort throughout.