|  |
| --- |
|  |

|  |
| --- |
| Codigo Test |

# Objective

The goals of the test:

1. Able to create backend RESTful APIs using the Java, JPA and Spring Framework.
2. Able to implement caching mechanisms like Redis for performance optimization.
3. Able to secure APIs using authentication mechanisms (JWT or Bearer tokens).
4. Able to implement token expiration and refresh processes.
5. Capable of handling concurrent requests efficiently.
6. Able to design and document a functional flow diagram for the system.
7. Able to design a database relational diagram and schema.
8. Able to demonstrate knowledge of microservices architecture and its implementation.
9. Able to demonstrate knowledge of cloud system design diagrams.
10. Capable of implementing scheduler tasks or any different logics for promo code generation.
11. Able to create a user-friendly CMS UI for eVoucher management using Angular or React.
12. Able to follow best practices in code quality, modularity, scalability, and security.
13. Able to deliver well-documented and structured code through GitHub.

## Quantitative Analysis

1. Users can generate multiple promo codes for a single voucher.
2. Voucher creation frequency is relatively low (around three creations per day), but each voucher can have up to 100 promo codes.
3. Based on these assumptions, an estimated 300 promo codes may be generated daily.
4. The system is expected to handle 10⁵ users daily accessing the website.
5. Approximately 50% of users will view discount promos as part of marketing.

## Project Details

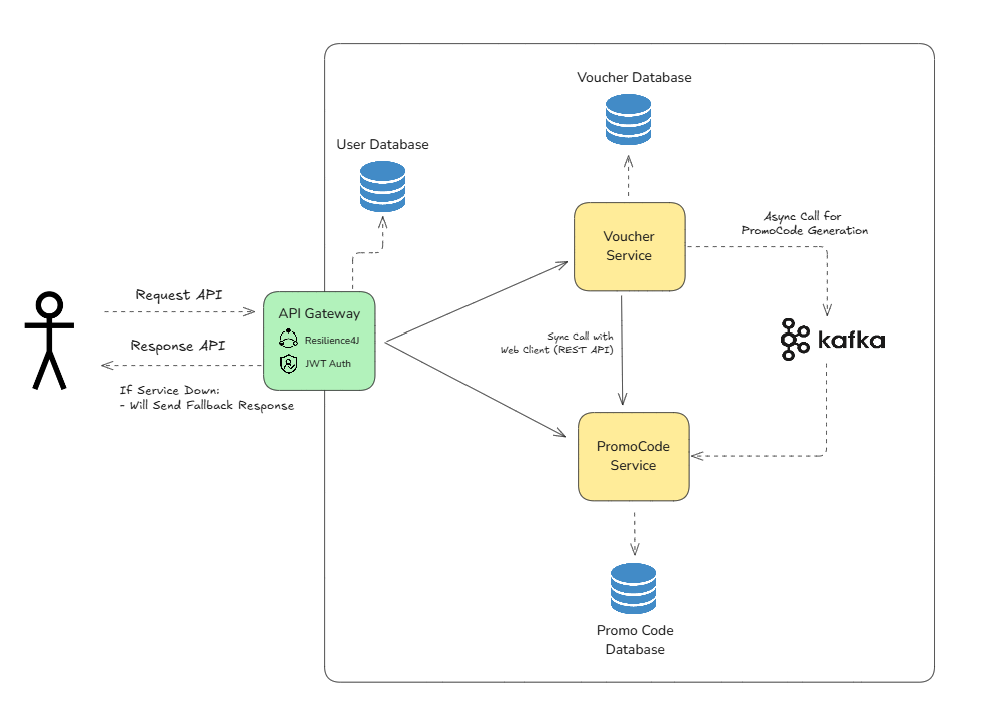


Figure 1. Microservice Diagram

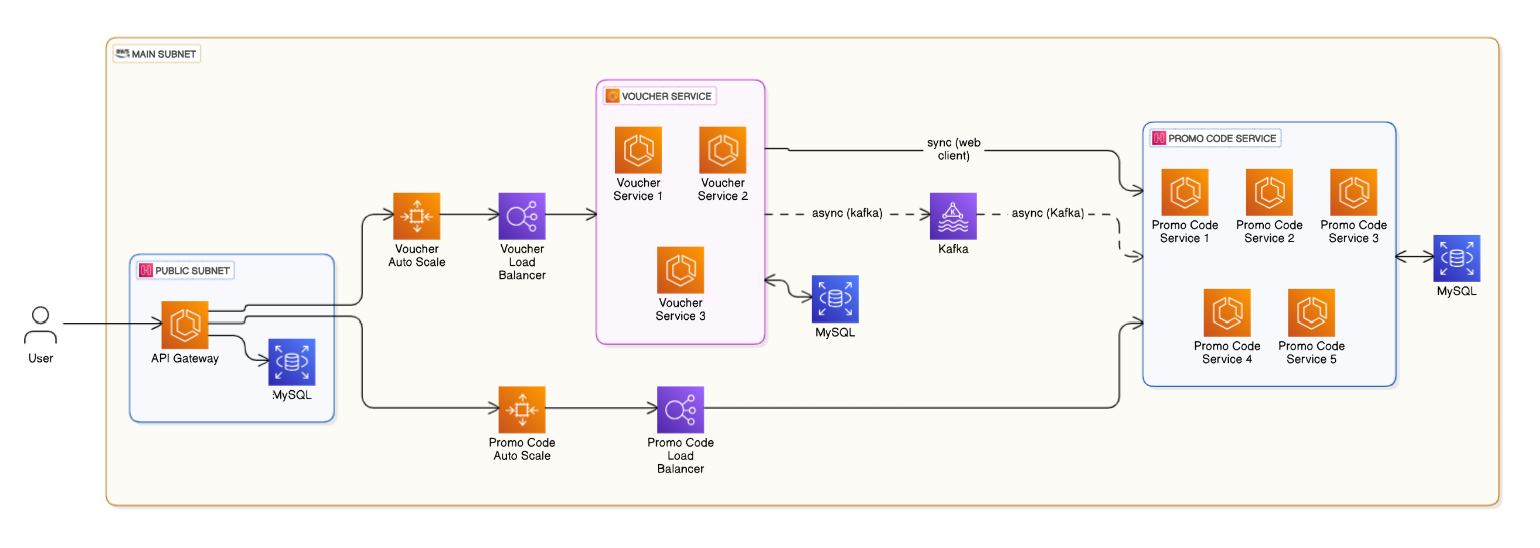


Figure 2. Cloud Diagram

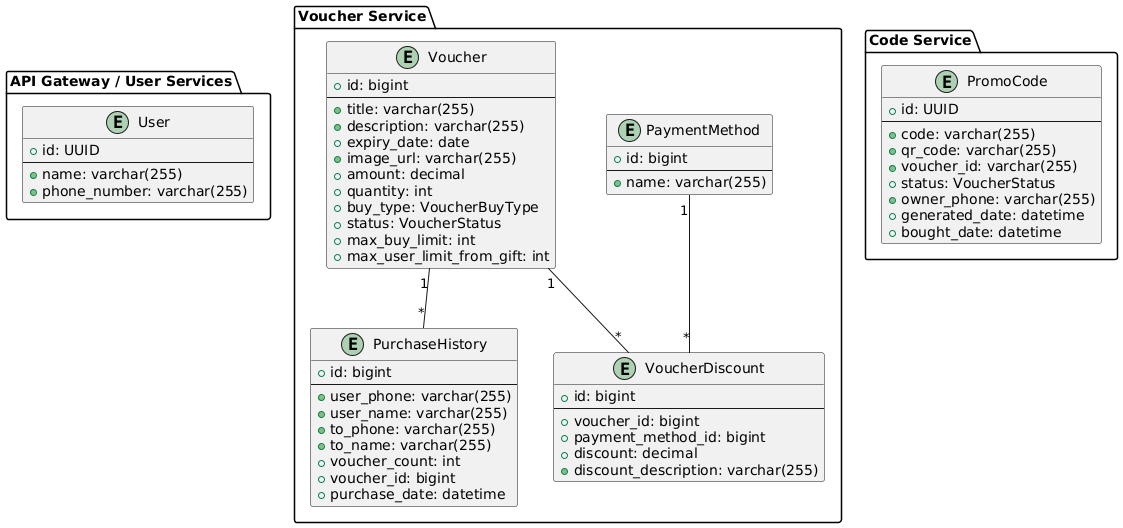


Figure 3. ERD (Entity Relationship Diagram)

## Purchase Functional Diagram

In this implementation, we will create new logic beyond the existing test logic for generating purchases. The test requirements specify that QR codes and promo codes will be generated asynchronously using a scheduler. However, this approach could negatively impact the user experience, as users would not see their purchased promo codes immediately.

To address this, instead of generating promo codes and QR codes asynchronously, we will pre-generate QR codes and promo codes when vouchers are created. This solution will enhance the user experience by ensuring that the codes are available immediately upon purchase.

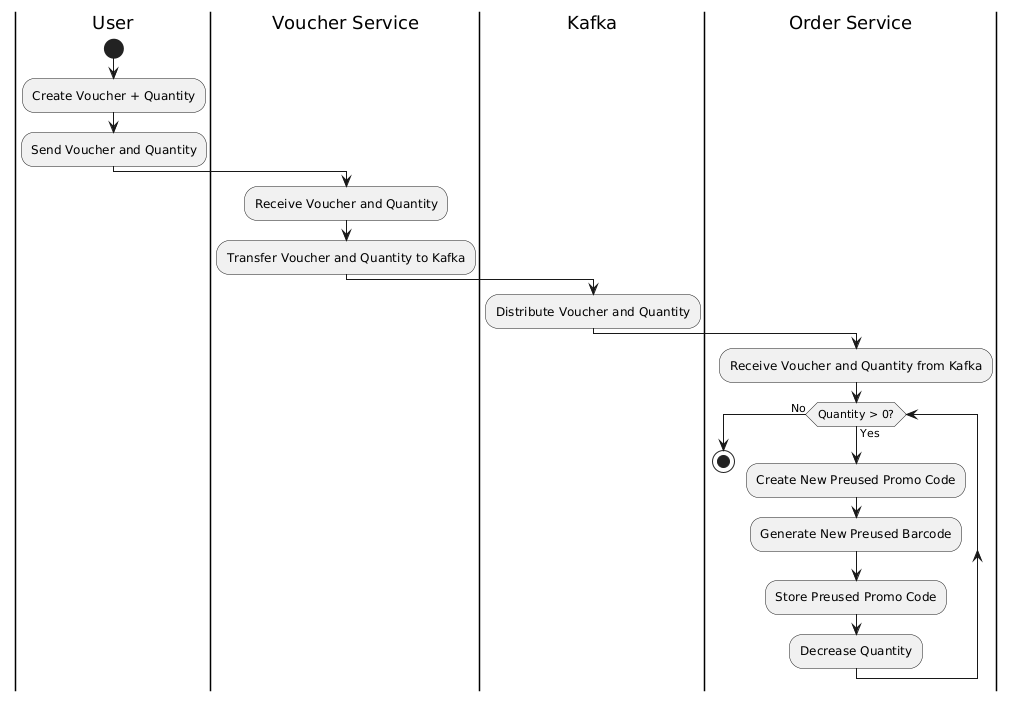


Figure 4. Create Voucher Flow Diagram

In this diagram, shows whenever a user or admin creates a voucher, a pre-generated promo code and barcode are automatically created to streamline the process.

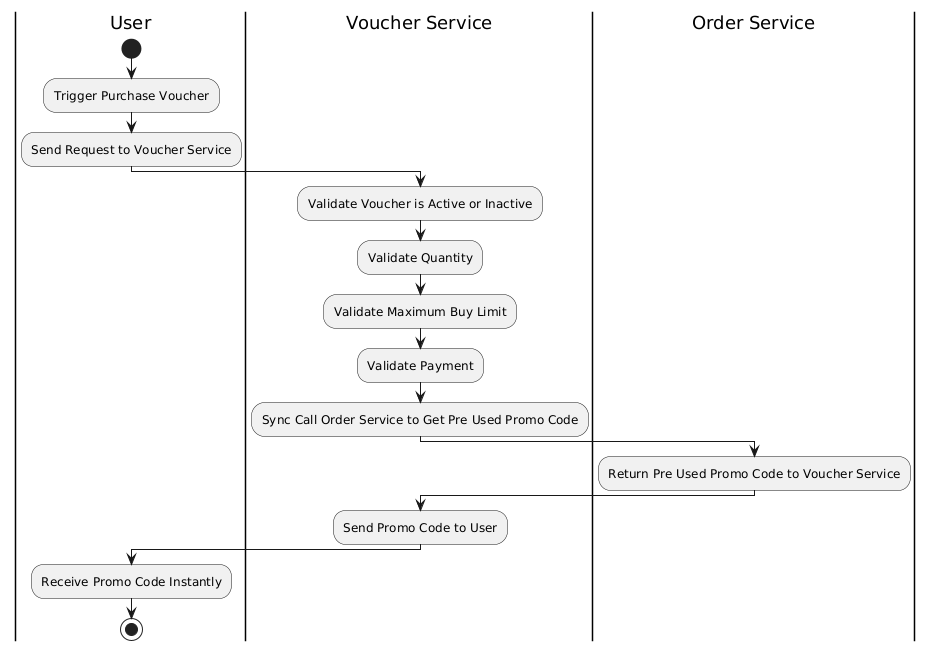


Figure 5. Purchase Voucher Flow Diagram

When users purchase a voucher, the system will take only a second to retrieve the pre-generated promo code. As a result, users will instantly see the promo code and QR barcode.

## Authentication Functional Diagram

Authentication Diagram Flow:

## PlantUML Diagram

Figure 6. Authentication Flow Diagram

From the above diagram, we can see that all the APIs referenced in the process must be authenticated at the API Gateway. If a user is not authenticated, they will be unable to access the APIs, except for the authentication API. This ensures that no unauthenticated requests are directly routed to the services, maintaining security and access control.

## 

## Payment Logics

Payment Diagram Flow:

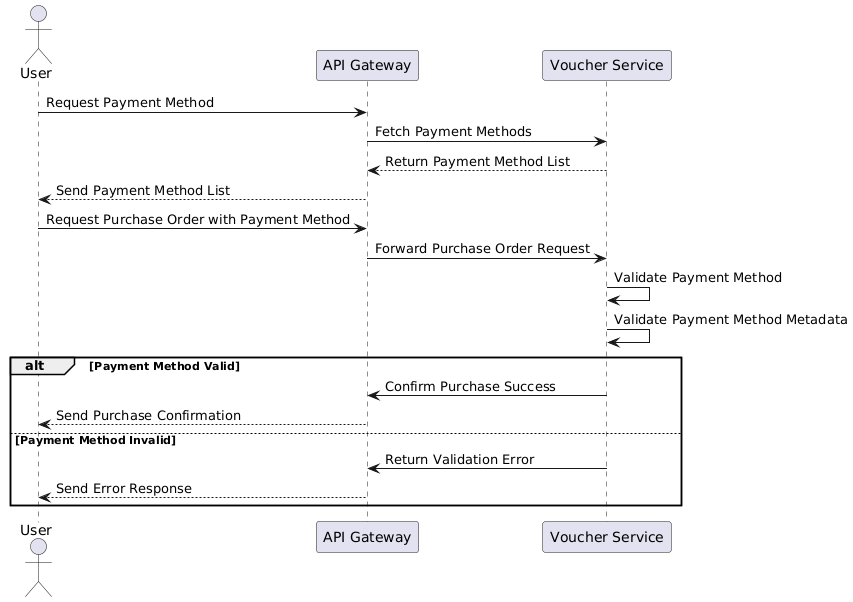


Figure 7. Payment Logics

In a real scenario, before users access this API or make a purchase, they will be referred to a third-party service to complete the payment. After completing the payment, the system will receive the payment metadata. When users proceed with a purchase, their request will include this metadata. The purpose of the backend is to validate the metadata provided by the user by verifying it with the third-party service. If the metadata is confirmed to be correct, the purchase can proceed.

## Circuit Breaker Implementation

Using resilience, we’re implementing a **circuit breaker** pattern in our system. **Circuit breaker** helps manage failures in a distributed system, ensuring that the system can gracefully handle failure scenarios instead of allowing the failure to propagate.

In our Voucher and Promo Code Management systems, implementing the circuit breaker ensures that if any downstream service, such as the database, experiences failures, the rest of the system remains unaffected, thereby maintaining a smooth user experience and improving system stability.

## Reactify Implementation

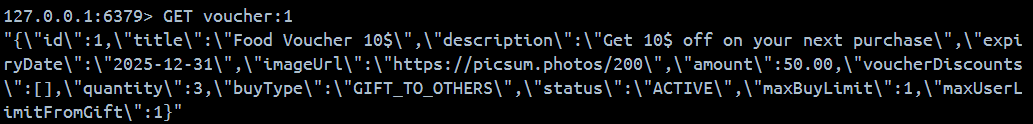
For the **Voucher Service**, we are implementing a **reactive programming model** to handle the concurrency requirements effectively. Reactive programming allows us to build scalable and responsive systems that can handle a large number of concurrent requests without being constrained by traditional blocking architectures.

We need to use a reactive approach because the **Voucher Service** requires high concurrency to efficiently manage voucher creation, promo code generation, and purchase processes. A traditional blocking architecture, such as one using **JDBC** and **JPA**, does not natively support non-blocking I/O, which can lead to performance bottlenecks when dealing with a large number of concurrent users or requests.

Instead, we are using **R2DBC (Reactive Relational Database Connectivity)** for database operations in the Voucher Service. R2DBC is designed to work seamlessly in a reactive environment, enabling us to perform non-blocking database operations while leveraging the benefits of reactive streams.

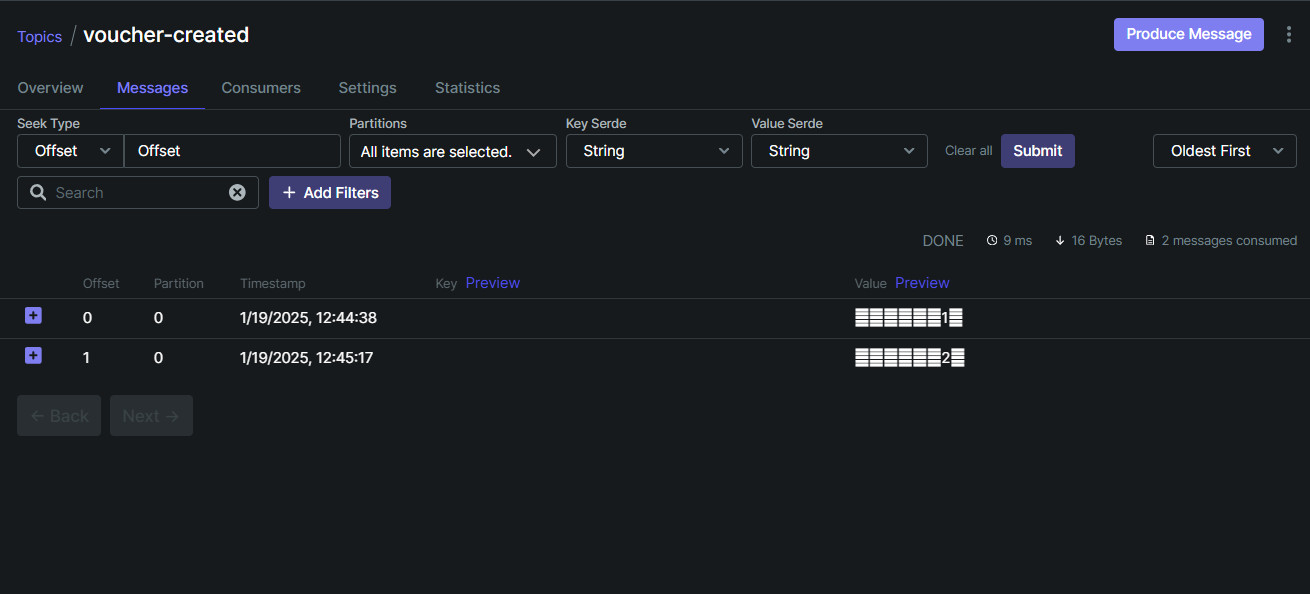
## Redis Implementation

Based on the assumption, we need to implement a faster way to get the discount of each voucher, so redis will be implemented on Voucher Service.



## Kafka Implementation

Kafka used for asynchronous call, this scenario will be triggered when voucher was created and order service must picked the messages, to generate promo code.



## Result

Here’s the result of the backend projects:

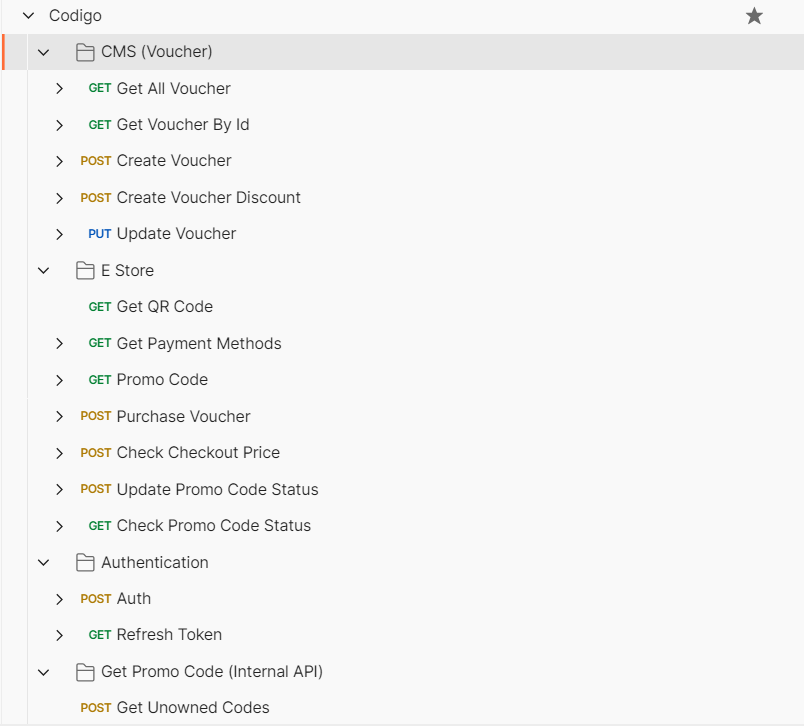


Figure 8. Result List

Total 15 API was created for this implementation

## Further Improvement

Due to the time limitations of the project, there is still room for improvement:

1. The generation of pre-used promo codes can be optimized to occur only when the number of available promo codes reaches a certain threshold. For example, if a voucher has a quantity of 1,000, it is unnecessary to generate all 1,000 promo codes at once. Instead, we can initially generate a batch of 100 promo codes and implement logic to monitor the count. Ifthe available promo codes fall below a certain threshold (e.g., 100), additional promo codes can be generated dynamically. This approach helps to prevent database overloading and optimizes resource usage on the server.