SHZ

Group Project Report

Student Name: Yanhao Sun **Student Number**: 19205434

Student Name: Zhao Tong **Student Number**: 23203139

Student Name: Jiayu He **Student Number**: 20205655

Code Branch: SHZ_DS_Final_Code

1. Synopsis

The system we intend to create is a bank system. This system allows users to register and login the system, as well as deposit and withdraw money, and buy several financial products provided by the bank. The majority of technologies we used are MoM and REST. Basically, our system involves six microservices, which can be grouped to three large services: User Service, Basic Service, and Financial Service. The main functions of User Service implemented are user identity and access management, i.e. the logic that handles the interaction with the database during registration and login and the selection of subsequent operations and access. The main function of User Service implemented is account management, i.e. enable access to money and save user information such as transaction records. Financial Service includes four microservices, (the overall function of this part is similar to Quoco), where brokers achieve and the other two parts of the interaction, bond, fund and stock for the three financial products system respectively.

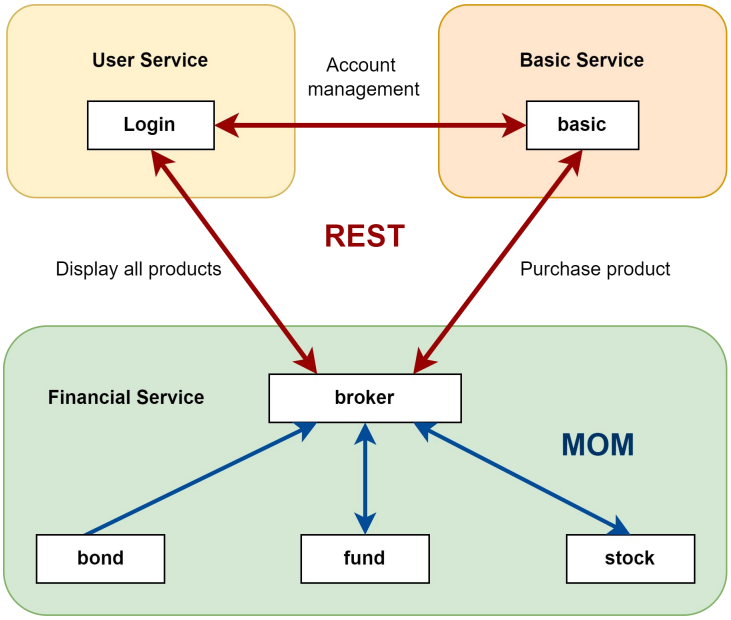

Figure 1. Integration of Technologies in This System

As shown in figure 1, our approach to integrating the two technologies involves using REST for interactions between three components, while Message-Oriented Middleware (MOM) is employed for communication among the four microservices within the financial products segment. Therefore, the broker serves as the convergence point for interactions between MOM and REST.

The detailed functions that each service implemented are shown below:

User Service:

- Login: log in to a user account with username and password.
- Register: register for a new account, user can set their username and password.

- User Interface: choose the service users want to access, including check account (check the balance, credit score, transaction history and financial product holding), logement, withdrawal, getting financial product recommended investment and buying a financial product.

Basic Service:

- Deposit and withdrawal
- Check balance and credit score
- Check transactions
- Check financial purchase records

Financial Service:

- Finance service involves three financial products: bond, stock and fund.
- Investment Recommendation: each financial product service will give a recommended investment to the client based on their age, credit score and balance.
- Buying Product: buy a product with customized price.

2. Technology Stack

Tech 1: Representational State Transfer (REST)

What it is used for:

This technology is used for user service and basic service. For example, the register function in User Service sends an HTTP POST request to access and modify the database through the path *localhost:8080/user/register*. The check balance function in Basic Service sends an HTTP GET request to access and get value from the database through the path *localhost:8081/balance*. We also used this technology for the communication between the three services. For example, User Service sends an HTTP POST request to Financial Service to ask for the recommended investment through the path *localhost:8082/products*.

Why we choose to use it:

REST utilizes standard HTTP methods (GET, POST, etc.) and data formats (such as JSON or XML). This consistency simplifies the development and understanding of complex banking APIs. In our distributed system, the statelessness of REST offers significant advantages in handling large-scale concurrent requests, maintaining high availability and compatibility, and simplifying regulatory compliance. This approach is particularly suited to highly regulated and dynamically changing banking business requirements.

The specific requirements in the bank system and corresponding benefits of using REST are shown below:

1. Simplified Server and Client Interaction

   The banking system needs to efficiently handle thousands of customer requests, which may originate from various endpoints such as mobile apps, web interfaces, and ATMs. The statelessness of REST simplifies the interaction between servers and clients. Specifically, each request is self-contained with all necessary information, and the server does not need to retain the state from previous requests. This simplifies server design and operation, as there is no need for additional logic to manage and maintain state across requests.

2. The Requirement of Scalability in the Banking System

   The Banking system needs an ability to accommodate increasing numbers of users and transaction volumes during peak periods. REST enables systems to scale more easily, as adding additional servers to handle requests does not require synchronizing user sessions or state information. The stateless nature of RESTful APIs allows for load balancing across multiple servers without concerns about data consistency among users.

3. Reliability and Fault Tolerance

   The Banking system requires very high reliability and fault tolerance to ensure continuous availability of services, even in the event of component failures. The request model of REST means that each request can be processed independently, and the failure of one request does not affect others. If a server fails, clients can resend the same request to another server without losing information or causing inconsistencies in state.

Tech 2: Message-Oriented Middleware (MoM)

What it is used for:

This technology is used for the interaction inside the financial service. For example, when the User Service sends a request to check the provided financial products, the broker broadcasts the user's information via topic. Each product and its investment amount are then retrieved from a queue and packaged before being returned to the User service. Similarly, when a user wants to purchase a financial product, the broker determines which product is being purchased and communicates this purchasing message to the specific product service through a designated queue.

Why we choose to use it:

In the Financial Product Management Microservice architecture, leveraging Message-Oriented Middleware (MOM) offers advantages in handling asynchronous communication, decoupling services, and ensuring system flexibility and scalability. The following highlights specific requirements of these microservices within the financial management system and the advantages of MOM:

1. Asynchronous Communication

   Financial product management microservices need to handle complex data analysis, investment strategy recommendations, and responses to market changes, which may require varying processing times. Immediate responses are not always feasible or necessary. MOM allows these services to exchange messages asynchronously. For example, a service can send a message to a queue, and another service can retrieve it when ready to process. This approach means services can operate at their own pace without being blocked by waiting for responses from other services.

2. Service Decoupling

   Each financial product management microservice needs to be developed and deployed independently to quickly respond to market changes or changes in business requirements. By using MOM, services do not need to communicate directly with each other; instead, they interact through message queues or topics. This decoupling ensures that changes, upgrades, or failures of individual services do not directly impact other services, thereby enhancing the overall robustness and flexibility of the system.

3. Fault Isolation and Recovery

   When there is a failure in the financial product management service or network issue, the system needs to continue accessing other services and quickly recover from the failure. With MOM's buffering capabilities, even if a service is temporarily unable to process messages, the messages can wait in the queue until the service is restored. This ensures that information is not lost and allows the system to quickly return to normal operation once the service is recovered.

3. System Overview

3.1 Main Component of the System and Their Functions

3.1.1 User Service

Database:

The user service involves a database with one table: *users* to store users' basic information.

table: *users*			
user_id	user_name	age	password

To connect the REST service to the database, this work leverages Spring Data JPA to create a repository interface extending JpaRepository<User, Integer> that provides CRUD (create, read, update and delete) operations on the database. Within the service functions, by injecting an instance of this interface, we can use its methods in REST controllers or service classes to perform database operations such as querying users, saving users, updating users, etc. Spring Data JPA automatically translates method names into corresponding SQL queries, allowing seamless database operations in the application.

Services and Functions:

This service involves a backend and a frontend service.

The backend service takes the responsibility of login authentication and registration, which involves the following functions:

loginService(String userName, String password): check if can find the user by the input username and password in the *user* table. If it can, the password is correct, otherwise the user may input a wrong password or it does not exist.

registerService(User user): check if the user already exists in the *user* table. If not, insert a new user to the *user* table, otherwise make no change.

The frontend service first takes the responsibility of registration for new user and login authentication. When a user asks for a registration, the frontend service sends HTTP POST requests with a new username and password via localhost:8080/user/register to call the registerService function. When a user asks for a login, the frontend service sends a HTTP POST request with username and password via localhost:8080/user/login to call the loginService function.

The frontend service takes the responsibility of user interface and connection with the Basic Service and Finance Service. Users can choose the service they want to take, such as deposit, withdraw and buy a finance product. The frontend service shows the user if the operation is successful. This is implemented by sending HTTP requests to the other two services, which is explained in 3.2.

3.1.2 Basic Service

Database:

The basic service involves a database with three tables: *users* to store users' balance and credit score, *transactions* to store users' transactions history, and *finance_transactions* to store users' financial product purchasing history.

table: *users*		
user_id	balance	credit

table: transaction*s*			
transaction_id	user_id	type	amount

table: finance_transaction*s*		
transaction_id	user_id	product_name

This service uses the same method as the one used in User Service to connect REST service to the database.

Service and Functions:

This service takes the responsibility of account management such as logement, withdrawal, transaction checking, etc., which involves the following basic functions:

-register(int user_id): add a new item in the *user* table.

deposit(int user_id, double amount): update the *user* table with the money and credit score after deposit and insert a transaction history in the *transactions* table. In this function, each deposit will add 1% of the number of deposits to the user's credit score. Notice that since the credit score is an integer number, deposit an amount under 100 cannot get a credit score.

withdraw(int user_id, double amount):check if the balance is more than the money to withdraw. If it is enough, update the *user* table with the money after withdrawal and insert a transaction history in the *transactions* table. Otherwise, make no change.

addFinanceHistory(Finance history): insert a financial product purchasing record to the *finance_transactions* table

checkBalance(int user_id): get the value of balance from the *user* table with user id

checkCredit(int user_id): get the value of credit score from the *user* table user id

checkTransactions(int user_id): get a list that store the transactions of the user from the *transactions* table

checkFinances(int user_id):get a list that store the financial product purchasing history of the user from the *finance_transaction* table

IfExistUser(int user_id): a helper function, which iterate the user table to find if the input user id is in the *user* table (i.e. if the user is recorded in this service)

This service listens to and gets HTTP requests from User Service and makes corresponding operations and responses based on the requests and parameters sent from User Service. It is also a 'hub' of the financial product purchasing, taking the responsibility of deducting money and confirming the purchasing. These are explained in section 3.2.

3.1.3 Financial Service

Database:

The financial service involves three databases, each with a table to store the users who bought the corresponding product.

database: *bond / fund / stock*

table: *bond_purchases / fund_purchases / stock_purchases*			
purchase_id	user_id	purchase_time	investment

This service uses JDBC（Java Database Connectivity）to connect the service and database. It uses the JDBC API from the Java standard library to establish a connection with the database for performing database operations within the application. Using this approach, a database connection can be made within the application using the DatabaseUtil.getConnection() method, enabling the connection between the service and the database.

Service and Functions:

Three microservices are involved in this system to generate recommended investment:

- Bond: for users over 55 years of age or has a credit score over 750, recommend an investment of 40% balance. Otherwise, recommend an investment of 20% balance.
- Fund: 30% balance for most case
- Stock: for users under 40 and has a balance more than 20000, recommend an investment of 50% balance. Otherwise, recommend an investment of 30% balance.

The program establishes a connection (Connection) with the ActiveMQ message broker using the ActiveMQConnectionFactory class. A broker gets the client information such as age, credit score and balance, broadcasting these messages to the three microservices (product service). Each product service will receive this client information. The three product services calculate based on the standards shown above. After calculating the recommended investments, each product service sends its recommendation to a designated queue associated with the broker. This queue holds the recommendations from all three product services. The broker retrieves the recommended investments from the queue. It collects and processes the recommendations received from the Bond, Fund, and Stock services. Once all recommendations are collected by the broker, the broker sends the compiled list of investment recommendations to the User Service. The inner service is implemented with MoM and the communication with other two services is implemented with REST. The process of communication with User Service is explained in section 3.2.

The product services also allow clients to buy the product based on their preferences. Clients are allowed to choose which type of financial product and customize the price they want to buy. When a client buys a certain product, a history will be added to the corresponding database.

3.2 Interaction Between Services

The basic structure of the system is shown in Figure 2.

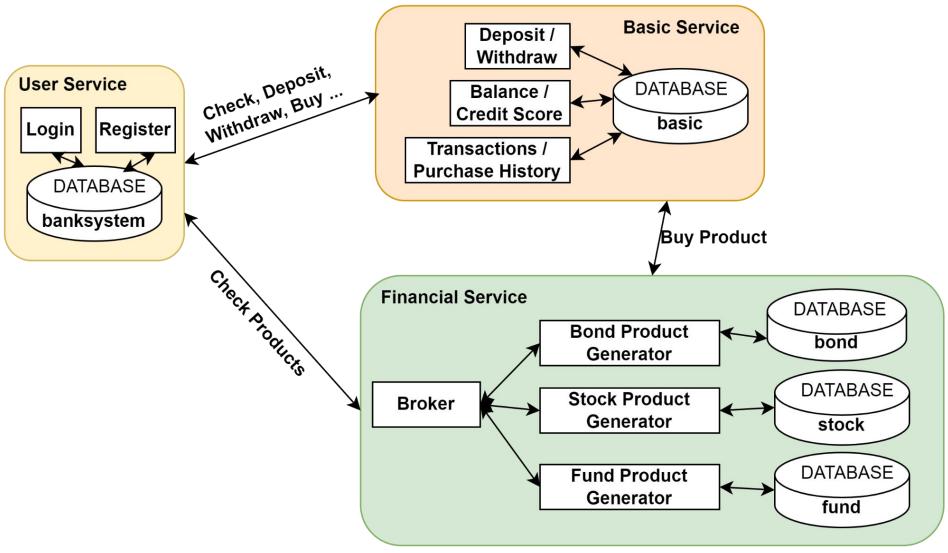

Figure 2. Architecture of the Bank System

After login which is explained in section 3.1.1, User Service shows the user interface. Notice that when a user asks for registration, User Service sends a register information to Basic Service at the same time by sending an HTTP POST via *localhost:8081/register*. When Basic Service receives the request, it calls the register function. This is because a new account is needed to be added in the users table in Basic Service database at the same time of a new user adding in the user table in User Service.

Users can select which service they are going to take. After selecting, User Service sends requests to Basic Service and Financial Service to call the functions in the two remote services, and the two services send a response back to show the operation result. The requests and responses transferring are implemented with REST.

To deposit a certain amount of money, User Service sends a HTTP POST request through *localhost:8081/deposit* with the user's ID and the amount of money intended to save to Basic Service. When Basic Service receives the request, it calls the deposit function inside its service. Basic Service responds to User Service with an Ack. The first element of the Ack is a ClientInfo data structure, which is shown in snippet 1. The ClientInfo is a shared data structure between the three services used to transform information. In this case, Basic User responds to User Service with a ClinetInfo<userID, balance, creditScore>. The second element of the Ack is a boolean type to show if the depositing is successful or not. When getting the Ack, User Service first checks the second element, if it is true, to get the balance from the data structure. In this case, it can get the updated balance and credit score. Otherwise, prints failed.

```
public class ClientInfo implements Serializable{
    public ClientInfo() {}
    //some other variable declaration here
    public Product product = null;
    public String name = null;
    public int age;
    public int id;
    public int creditScore = 0;
    public double balance = 0.0;
}
```

snippet 1. ClientInfo Data Structure

Similarly, to withdraw a certain amount of money, User Service sends a HTTP POST request through *localhost:8081/withdraw* with the user's ID and the amount of money intended to withdraw to Basic Service. Basic Service calls the withdraw function and responds to User Service with an Ack. The first element of the Ack is a ClientInfo data structure, which includes the updated balance. The following steps are the same as the previous one.

To check and balance and credit score , User Service sends a HTTP GET request through *localhost:8081/balance/{userID}* to Basic Service, and Basic Service calls the response to User Service with an Ack. The first element of the Ack is a ClientInfo data structure, which includes the requested user's balance and credit score. The following steps are the same as the previous one.

To get the transaction history, User Service sends a HTTP GET request through *localhost:8081/transactions/{userID}* to Basic Service, and Basic Service calls the checkTransactions function and responds to an Ack. The first element of the Ack is a list that stores all of the transactions. The Transaction data structure is also shared between the three services, which includes a String type value showing the type of transaction (In or Out), and a double type value showing the amount of transaction. Following processes are the same as the previous one.

Similarly, the financial transaction history can be obtained by sending a HTTP GET request through *localhost:8081/finances/{userID}*. Basic Service responds to User Service with an Ack with the first element of a list that stores all of the product names, which is a String type variable. Following processes are the same as the checking of the previous one.

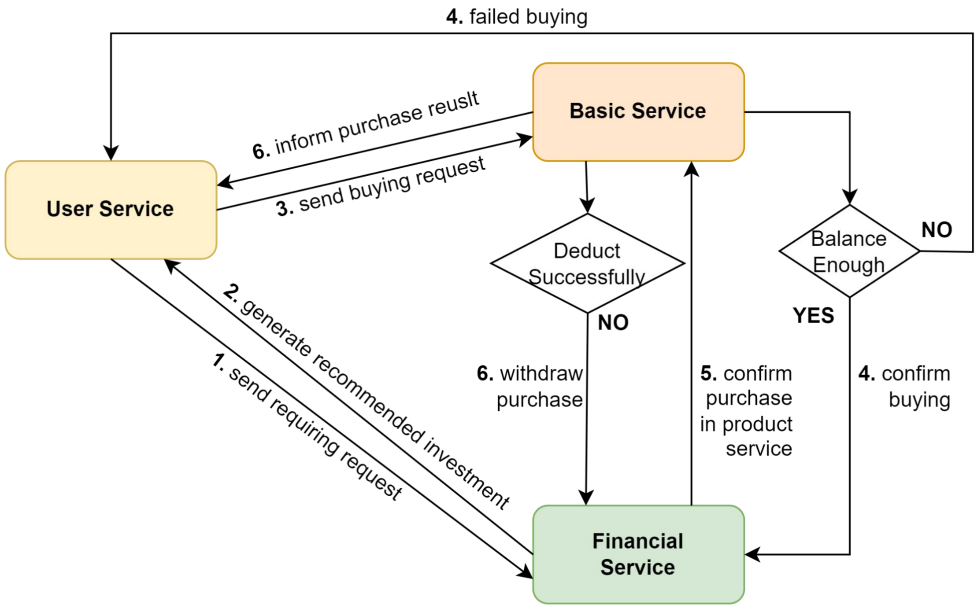

Figure 3. Flow Chart of Buying Financial Product

Figure 3 shows the process of buying a financial product. User Service first sends a HTTP POST request to the Financial Service through *localhost:8082*/products with the ClientInfo as the input RequestBody. The broker receives the request, generates a list of recommended investments of the three financial product producers (the generation process is inside the financial service), and sends the list back to User Service as the response. Then the user can choose which financial product he or she wants to buy and enter the purchasing price.

After that, User Service sends the user and product information (both the information are stored in the ClientInfo) with the intended price to Basic Service through l*ocalhost:8081/finances/* with a HTTP GET request. Basic Service then determines if the balance is sufficient to buy the requested price of the financial product. If sufficient, send a HTTP POST request to Financial Service through *localhost:8082/purchases* to confirm the purchase. Otherwise, send rejection to the User Service.

After receiving the buying confirmation request, Financial Service completes the operation inside and responds to Basic Service with a String type purchasing result (purchase successful, purchase failed, etc.). If the “purchase successful” is sent back, Basic Service deducts corresponding money and response to User Service with a PurchaseResult data structure, which includes the result String and the product information. Specifically, if the deduce operation in Basic Service failed, Basic Service will send a HTTP POST to Financial Service through *localhost:8082/failure*, then Financial Service will withdraw the corresponding purchasing record.

3.3 Scalability and Fault Tolerance

3.3.1 Scalability

Our system allows a large number of users to register. When the number of users increases, the ability of horizontal scaling of our system can be achieved by adding more instances of user services. For example, using a load balancer to distribute user requests across multiple instances of user services, thereby improving system throughput and concurrent processing capability.

We also used ActiveMQ, which is an asynchronous communication and message queues to handle interactions between services that can reduce coupling between services, enhance system scalability, and improve performance. Message queues enable buffering and throttling of requests and responses between different services, making the system more resilient.

Moreover, each service can be independently deployed and scaled because they communicate with each other via APIs. This decoupled design allows each service to scale horizontally independently as needed, without impacting the performance and availability of other services.

3.3.2 Fault Tolerance

In the implementation of login, unexist username and wrong password will both lead to a failed login, which ensures the correct access to the user account, assuring the security of the user and account. Moreover, in the implementation of registration, registering by the same user name will be rejected, avoiding the waste of resources in the database. In the implementation of withdrawal, withdrawing more money than deposit will be denied.

In the implementation of buying a financial product, step 4, 5 and 6, as shown in Figure 2, perform a three-way handshake type acknowledgement. Basic Service first sends a buying request without deducting money, while the money is deducted after receiving the acknowledgement from Financial Service. This is used to ensure that the financial service receives the buying request and adds a purchasing history to the corresponding database, avoiding the situation of deducing the client's money but not recording the user's purchase. Similarly, if the Basic Service deduces money unsuccessfully, it will send a withdrawing request to Financial Service to notify the service to cancel the purchasing record adding, which avoids the situation of adding a purchasing history but failing to deduce money. If the Financial Service does not receive a withdrawing request, it will confirm the purchase. This three way hand-shake type acknowledgement ensures the synchronization between the Financial Service database and the Basic Service database. What's more, chargeback failures are in the minority, so we chose to send an extra cancellation request on failure rather than a request on success to minimise wasted resources.

4. Contributions

- **Yanhao Sun**: User Service and video recording
- **Jiayu He**: Basic Service and report
- **Zhao Tong**: Financial Service and dockerization

5. Reflections

Challenges and Solution:

1. Challenges of the interaction between distributed systems

The biggest challenge we encountered during the design was the purchase process for financial products. This required interaction between three services. This is because different services have different user information. For example, the generation of recommended investments for bond products requires the user's age and credit score, while the age information is maintained by the user service and the credit score information is maintained by the basic service. In addition, payment should also be completed in the basic service, and before deduction, basic service should interact with the financial product to ensure that the purchase is also successful in the financial product's microservice in order to ensure the consistency of things between microservices and system security. This is a complex interaction process, which must ensure that the information in the three service databases is updated at the same time.

Our solution incorporates REST and MOM technologies. The flowchart shown in Figure1 and Figure 3 virtualises the purchasing process. The user service uses REST technology when it initiates a purchase request to the basic service, the basic service also uses REST technology when it initiates a purchase request to the financial product, and MOM technology is used for the interaction between the financial services internally. Payment is one of the core steps in the process, so we use the basic service as a 'hub' to determine whether the user can purchase the product. We also found that the purchase information in the financial service and the basic service must be consistent during the purchase process, however the update of both is not simultaneous but sequential, which can lead to security issues. Therefore we designed a three-way handshake protocol between the two services during the purchase process to ensure the consistency of information, which has been explained in section 3.2.

2. Problems in the containerisation process

The microservice containers are created before the database container is initialised, resulting in the system not running correctly. Each microservice needs to have its own corresponding database and the connection to the database before the start of the service is necessary. The build order of multiple databases and multiple services in the containerisation process is random, so there will always be the issue that the database has not been initialised and the service has already tried to connect resulting in an error.

Our solution is to add a delay and health check when creating the database, and add a dependency when creating the microservice container, i.e., the microservice must wait for its corresponding database container to be created and pass the health check before it can be created.

What would have done differently if could start again:

If we could start again, we would first identify the interfaces for each interactive function, defining the requested and returning data types and structures. In the interaction of User Service and Basic Service, we first built and tested the service with the service private data structure instead of defining a shared data structure used for transferring. Therefore, although both of the services could run with a satisfied performance individually, we wasted a lot of time on fixing the interface to implement the communication and transferring between the two services. Therefore, we realized that predefining the interface and shared data structure is important in designing a distributed system.

Learning from the technologies used in this work:

Through the use of REST, we learned how to design and implement web services that adhere to REST principles, including resource identification, state transfer, and operations, and understood how to utilize common HTTP methods (GET, POST, PUT, DELETE, etc.) to manipulate resources and enable CRUD operations. We understood one of the core concepts of REST, which is state transfer, achieved through HTTP methods to manipulate and transfer resource state, reflecting a state-driven design philosophy. We also grasped the principle of statelessness in REST, where each request contains sufficient information and the server does not need to maintain client state, aiding in scalability and simplifying development and maintenance.

One of the weaknesses of REST is the lack of standardization. REST does not have explicit standards, leading to certain variations between implementations, which may result in inconsistencies and uncertainties. For example, during the testing of the communication between User Service and Basic Service, Basic Service responded to a list of Transaction data structures but User Service got a LinkedHashMap type data. And we needed some extra steps to get the list from the LinkedHashMap, which made the program a little bit complex. What's more, REST has limited support for security, requiring additional security measures such as HTTPS and OAuth to protect aspects like data transmission and identity authentication.

The benefits of REST is that it is based on the HTTP protocol, using a simple resource and state transfer model that is easy to understand and learn, reducing development complexity. Its Loose Coupling and Scalability provided by the statelessness and resource-oriented design of REST makes interactions between components more flexible and facilitates horizontal scalability. For example, Financial Service and User Service used different technology to implement the inner architecture, but they can interact with each other with REST, enabling the communication between different components.

Through the use of MOM, we learned how to utilize message middleware to implement an asynchronous communication model, where message sending and receiving are independent, decoupling message producers and consumers, studied how to configure message middleware for routing and forwarding messages, ensuring messages reach their intended applications or services. We also learned how message middleware ensures message reliability and persistence through mechanisms like message persistence, ensuring messages remain intact even after system failures or restarts.

One of the weaknesses of MOM is configuring and managing message middleware can be complex, especially when dealing with high concurrency, large-scale messaging, and distributed systems, requiring consideration of various performance and reliability issues. What's more, Some message middleware may

not guarantee strict message ordering, especially in distributed environments, potentially leading to message reordering. For example, when User Service requests for the recommended investment from the three financial product services, Financial Service may send a product list with the id order of 3, 1, 2, etc. This has no impact on the functional implementation, but leads to some imperfections. What's more, deploying a single instance of message middleware can become a system's single point of failure, necessitating high availability and fault tolerance measures.

The benefits of MOM is that message middleware facilitates easy system scalability and distributed deployment, supporting collaboration and communication among multiple application instances. For example, MOM can simply implement the distributed system of three microservices (bond service, fund service and stock service) in Financial Service. Message middleware can be used to implement event-driven architectures, supporting real-time notifications and event processing, enhancing system responsiveness and real-time capabilities. For example, the broker in Financial Service can get the client's information and generate the recommended investment in real time.