American University of Armenia

In partial fulfillment of the requirements for the degree of BS in Computer Science

Professor Zaruhi Shahnazaryan

Student Ida Manukyan

Functional Specification Document of Bank System Application

Document Version 1.0.0

20 May

Table of Contents

Introduction

Project Scope

The banking industry is highly competitive, and banks that do not offer a comprehensive and user-friendly application may lose customers to competitors that do. Therefore, a well-designed and well-implemented banking application is crucial for banks to remain competitive.
This document outlines the functional specifications of a banking application that allows users to manage their accounts, conduct cross-currency transfers and access transaction history.

The underlying rationale for this business need is the importance of providing a secure and user-friendly banking application that empowers users to manage their finances efficiently. With the increasing popularity of online banking, users now expect easy-to-use intuitive interfaces, swift and seamless transactions, and secure access to their financial information.
The application provides functionalities for various user types, including bank, customer, and admin users. Each user type is assigned specific permissions and roles, ensuring efficient user profile management, streamlined account administration, and robust transaction analysis. The application consists of four distinct modules, each dedicated to specific functionality.

1. **User Management:** This module manages user profiles and application access, offering features such as user registration, sign-in, permissions, and role management.
2. **Bank Account Management:** This module oversees the management of bank accounts and transactions. It provides functionalities for different customer types (both legal and physical) to register their cards, execute cross-currency transfers, and access transaction history. Bank users review and approve profiles before activating them.

3. **Admin Management:** This module manages the administrative access to the application. It enables the admin to view all cards and accounts and accept, reject, or delete them.
4. **Accounting Analytics:** This module maintains a comprehensive transaction history and performs advanced analyses. It includes features to identify trends in transaction volume over time and conduct other analytical tasks. Additionally, it ensures that sensitive transaction data remains safeguarded against unauthorized access.

Collectively, these four modules deliver a comprehensive and user-friendly banking application that covers the essential requirements of a competitive banking system. The application provides seamless management of user profiles, efficient account administration, and in-depth transaction analysis. Moreover, it operates within a highly secure environment to safeguard sensitive financial information.

Document Scope

This document provides an overview of a bank application, including its architecture, design patterns, third-party integrations, configuration files, and testing approach. The document is intended for developers and stakeholders who are interested in understanding the technical aspects of the application. The document is organized into several sections, each of which provides a brief description of a key aspect of the application. These sections include Introduction, Architecture, Design Patterns, Third-Party Integrations, Configuration Files, Testing, and Conclusion.

References

1. Amazon Web Services: Amazon Web Services. (n.d.). Amazon Web Services (AWS) - Cloud Computing Services. Retrieved from https://aws.amazon.com/
2. Apache Kafka: Apache Kafka. (n.d.). Apache Kafka. Retrieved from https://kafka.apache.org/
3. Clean Code Architecture: Martin, R. C. (2012, August 13). The clean architecture. The Clean Code Blog. Retrieved from https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
4. Microservices Architecture: Richardson, C. (n.d.). Microservices architecture. Retrieved from https://microservices.io/
5. Spring Framework Documentation. (n.d.). Retrieved from https://docs.spring.io/spring-framework/docs/current/reference/html/

Risks and Assumptions

TODO revisit later

System/Solution Overview

Business Processes

In the following section two real life workflows are introduced: one for customer-user and one for bank-user.

Workflow for Customer-User

1. The customer-user logs in to the system using their email and password.
2. The system checks the email and password against the customer_users table to authenticate the user.
3. Upon successful authentication, the customer-user is redirected to the dashboard to view their account details and perform various actions.
4. The customer-user can view their legal and physical accounts and their respective balances by accessing the legal_entity_accounts and physical_entity_accounts tables.
5. The customer-user can also view their physical entity cards and their details by accessing the physical_entity_card table.
6. The customer-user can perform transactions by transferring money between their accounts or to other users. The system checks the balance in the account before allowing the transaction and updates the legal_entity_accounts and physical_entity_accounts tables accordingly.
7. The customer-user can also view their transaction history by accessing the transactions table.
8. If the customer user wants to close their account, the system updates the deleted column of the corresponding account row in the legal_entity_accounts or physical_entity_accounts table.
9. The customer-user logs out of the system.

Workflow for Bank-User

1. The bank user logs in to the system using their email and password.
2. The system checks the email and password against the bank_users table to authenticate the user.
3. Upon successful authentication, the bank user is redirected to the dashboard, where they can view various details and perform various actions.
4. The bank user can view all customer details by accessing the customer_users table.
5. The bank user can view all account details by accessing the legal_entity_accounts and physical_entity_accounts tables.
6. The bank user can view transaction history by accessing the transactions table.
7. The bank user can add, update or delete customers by modifying the corresponding rows in the customer_users table.
8. The bank user can add, update or delete accounts by modifying the corresponding rows in the legal_entity_accounts and physical_entity_accounts tables.
9. The bank user can add or delete physical entity cards by modifying the corresponding rows in the physical_entity_card table.
10. The bank user logs out of the system.

API Contracts

For Bank Users

1. Create Bank User

HTTP Method: POST

URL: /bank-users/add-bank-user

Required Authority: can_add_admin

Request Body:

Content-Type: application/json

Body Parameters:

userDto (BankUserDto): The details of the bank user to be created.

Response:

Content-Type: application/json

Returns: BankUserDto object representing the created bank user.

2. Get All Customers (Retrieve a list of all customer users)

HTTP Method: GET

URL: /bank-users/view-all-customers

Required Authority: can_view_customer

Response:

Content-Type: application/json

Returns: List of CustomerUserDto objects representing all customer users.

Get Customer by ID

3. Retrieve a customer user by their ID.

HTTP Method: GET

URL: /bank-users/customer/{loggedUserId}

Required Path Parameter: loggedUserId (UUID)

Response:

Content-Type: application/json

Returns: CustomerUserDto object representing the customer user with the specified ID.

4. Update Customer (Update an existing customer user).

HTTP Method: PATCH

URL: /bank-users/update-customer/{id}

Required Authority: can_update_customer

Required Path Parameter: id (UUID)

Request Body:

Content-Type: application/json

Body Parameters:

userDto (CustomerUserDto): The updated details of the customer user.

Response:

Content-Type: application/json

Returns: CustomerUserDto object representing the updated customer user.

5. Delete a customer user.

HTTP Method: DELETE

URL: /bank-users/delete-customer/{id}

Required Authority: can_delete_customer

Required Path Parameter:

id (UUID): The ID of the customer user to delete.

Response:

Returns: None

6. Sign In

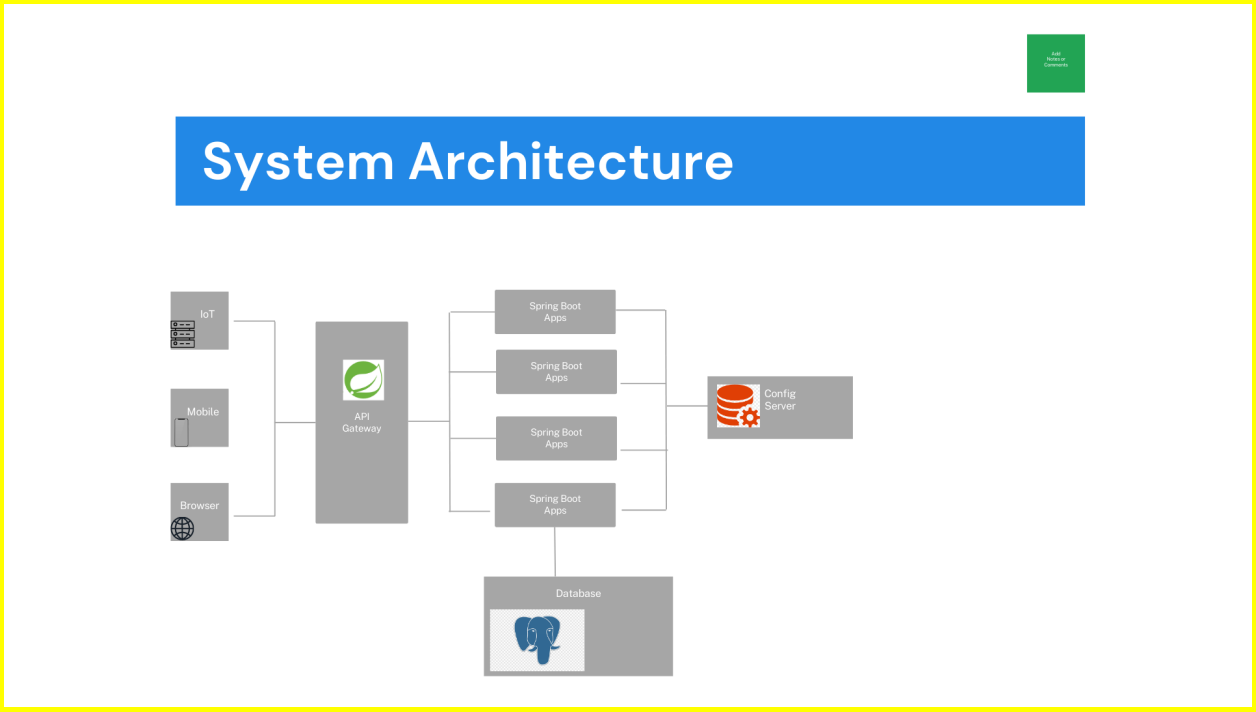
HTTP Method: POST

URL: /bank-users/sign-in

Request Body:

Content-Type: application/json

System architecture



The bank application is designed using a microservices architecture. This means that the application is composed of many small services that are responsible for specific functions. Each microservice is designed to be modular, with a well-defined interface that allows it to communicate with other services. This makes updating and maintaining the system easier, as changes can be made to individual services without affecting the entire system.

Each microservice follows a layered architecture similar to the Spring framework. The persistence layer manages the application's data, storing and retrieving information using a database. The service layer contains the application's business logic, providing a set of functions that other parts of the system can use. Finally, the controller layer manages the communication between the user interface and the service layer.

The layered architecture allows for better separation of concerns, making it easier to manage and test different application parts. In addition, the microservices architecture enables the system to

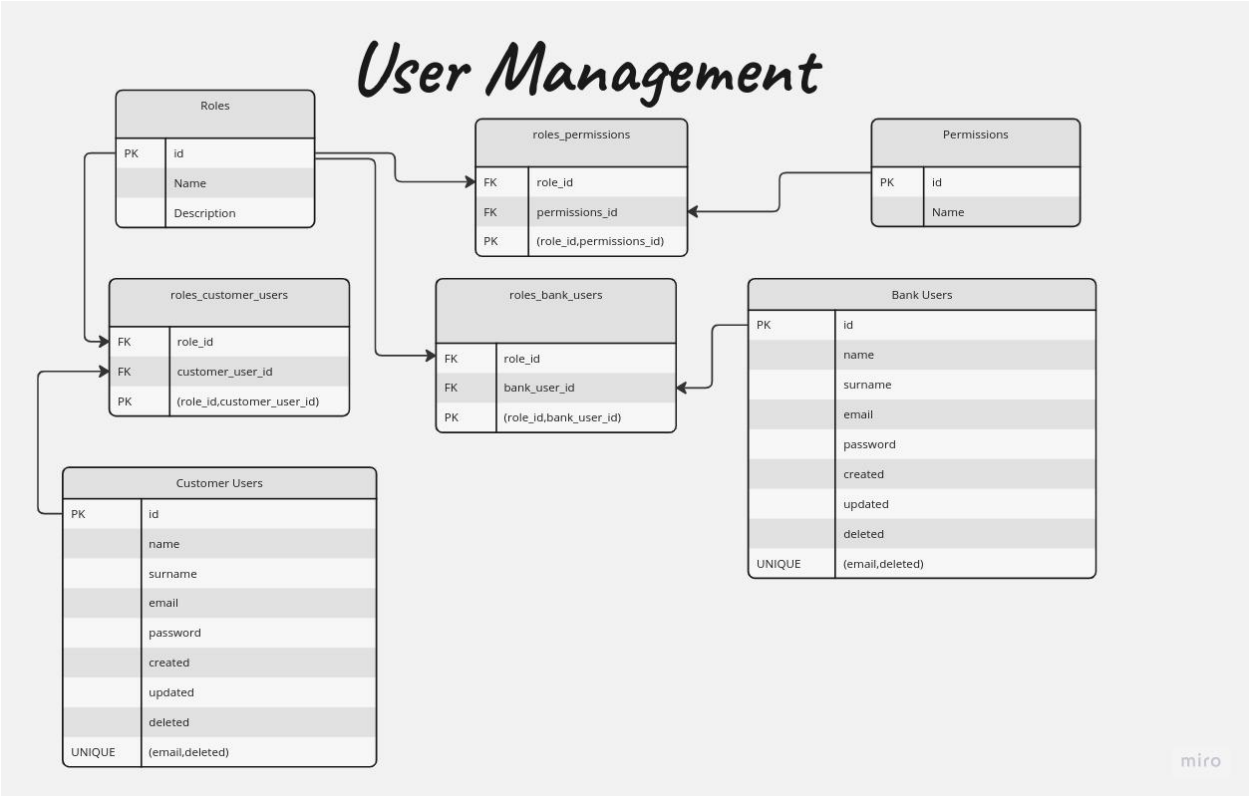
be more scalable and resilient. Each service can be scaled independently based on its workload, and failures in one service do not necessarily affect the entire system.

DB architecture

User Management

This module in the bank application consists of four tables: roles, permissions, customer_users, and bank_users. All tables have a UUID primary key called "id." Three cross tables link the roles and permissions with the customer_users and bank_users tables: roles_customer_users, roles_bank_users, and roles_permissions. These tables have foreign key constraints that ensure that when a user or role is deleted, the corresponding records in the cross tables are also deleted. Additionally, the application has an "admin_user_init" script that inserts some initial values into the roles and permissions tables and connects them via the roles_permissions cross table. This schema allows managing user roles and permissions in a flexible and scalable way while

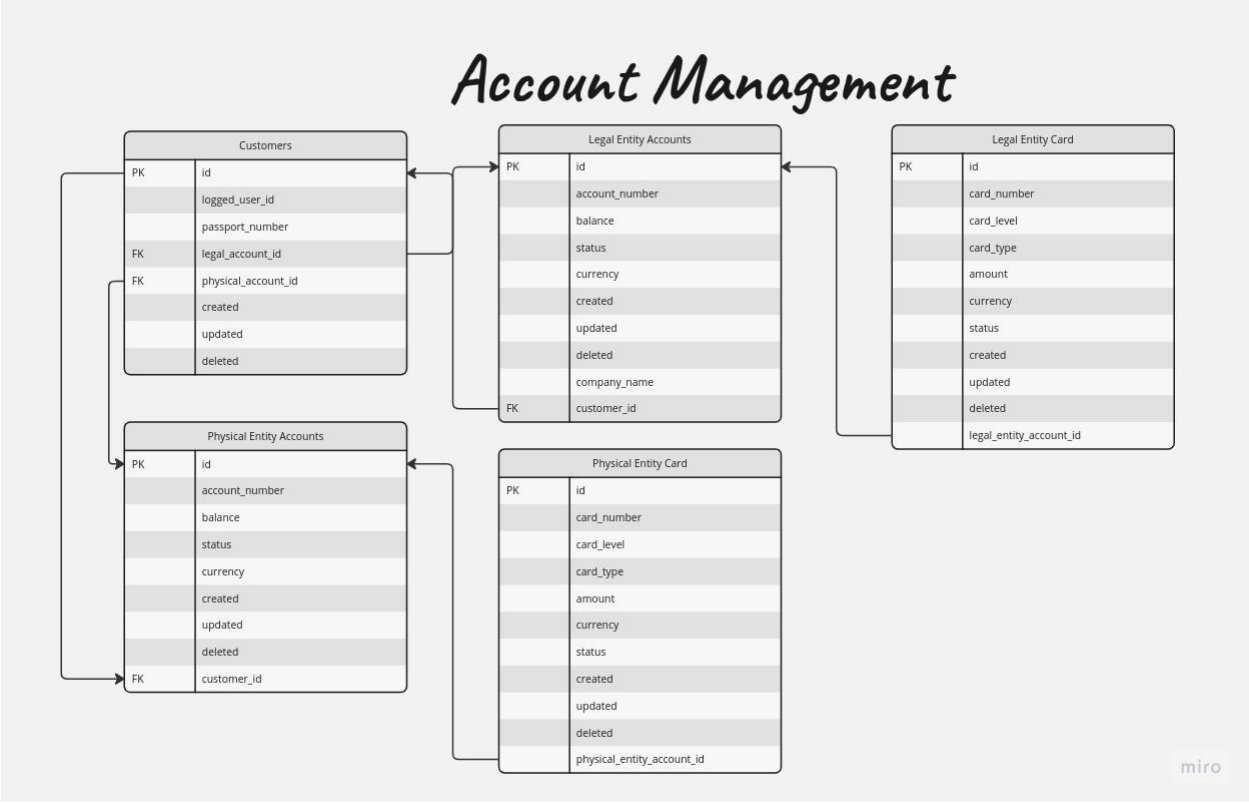
ensuring data integrity and consistency through using foreign key constraints.



Bank Account Management

In this module, each table has a primary key of type UUID and includes columns for created_at, updated_at, and deleted_at timestamps. Foreign key constraints link the tables, such as the legal_entity_customer_fk and physical_entity_customer_fk constraints on the legal_entity_accounts and physical_entity_accounts tables, respectively. These constraints link the accounts to the customer who owns them. In addition, the customer's table has foreign key constraints for its legal and physical account ids, linking them to the corresponding accounts in the legal_entity_accounts and physical_entity_accounts tables. Finally, the physical_entity_card

and legal_entity_card tables have foreign key constraints linking them to the corresponding physical_entity_accounts and legal_entity_accounts tables, respectively.



Messaging

RabbitMQ is an open-source message broker software that allows different applications and services to communicate by exchanging messages. It provides a messaging queue model where producers publish messages to a message broker, and consumers subscribe to specific queues to receive them. RabbitMQ supports multiple messaging protocols such as AMQP, MQTT, STOMP, and HTTP. It also provides advanced features such as message acknowledgment, routing, queuing, and priority. RabbitMQ is widely used in distributed systems, microservices

architectures, and cloud-based applications for reliable and scalable messaging. It helps decouple the producers and consumers, making it easier to scale the application and maintain reliability. In the bank application, RabbitMQ facilitates communication between different microservices, such as account management and transaction services. When a user makes a transaction, the transaction service sends a message to the account management service via RabbitMQ, indicating that the account balance needs to be updated. The account management service receives and processes this message, updating the account balance accordingly.

Design Patterns

Design patterns are solutions to commonly occurring problems in software design that have been tried and tested by experienced software developers. They are a way of capturing and sharing knowledge about effective software design, and they help to ensure that software is reliable, maintainable, and extensible. Design patterns provide a common vocabulary for developers to communicate solutions to problems. They also help to promote best practices, improve code quality, and make code easier to understand and maintain. By using established design patterns, developers can save time and effort, reduce the risk of errors and bugs, and improve the overall quality of the software they are building.

The Proxy Design Pattern is a structural design pattern that provides a surrogate or placeholder for another object to control access to it. In other words, a proxy object acts as an intermediary between the client and the actual object, hiding the details of the underlying object from the client.The Proxy Design Pattern is a powerful tool for controlling access to sensitive operations or data and improving application performance through caching. Its usage in the bank application

can help ensure the security and reliability of the system while also providing a better user experience.

The Proxy Design Pattern is used in the bank application to control access to sensitive operations or data. For example, the application may have certain operations that can only be performed by authorized users or administrators. A proxy object is used to restrict access to these operations, ensuring that only authorized users are able to perform them.Another use case for the Proxy Design Pattern in the bank application is to provide caching of frequently accessed data. A proxy object can be used to cache frequently accessed data, reducing the number of requests that need to be made to the underlying database or system. This can improve performance and reduce latency, especially in high-traffic environments.

Technology Stack

The application is built using Java using the Spring Framework, with microservice architecture and layered architecture. The application uses RabbitMQ for message queuing. The database management system used is PostgreSQL.

Java

Java was chosen for developing the bank application due to its support for modern development practices like functional and reactive programming. Java 11 provides new APIs and language features that can improve the application's performance, scalability, and security. Additionally, Java 11 has several performance improvements, such as a new garbage collector algorithm, that can make the application more efficient.

Spring Boot

Spring Boot simplifies and streamlines development, reducing boilerplate code and enabling greater focus on business logic. With built-in security features for authentication and authorization, Spring Boot is well-suited for sensitive applications like banking. Its microservices-based architecture allows for flexibility and agility, while easy integration with popular technologies like Hibernate and JPA enables comprehensive and modular application development.

Maven

Maven is a build automation tool for managing dependencies, building, and packaging Java applications. It simplifies the build process, reduces the time required to build and deploy the application, and provides a standard build lifecycle. In the bank application project, Maven is used to manage dependencies and automate the build process, increasing the development process's efficiency and productivity.

Flyway

Flyway is a tool that helps manage changes to a database schema and data. It allows for versioning of changes in SQL or Java-based migration scripts, and ensures scripts are executed in the correct order. Features like repeatable migrations, callbacks, and metadata tracking make Flyway more reliable. It integrates with build tools like Maven and Gradle, and with continuous integration and deployment systems. In the banking application, Flyway was used to manage the PostgreSQL database schema and data changes through SQL scripts stored in the application's source code repository.

PostgreSQL

PostgreSQL is a reliable and scalable open-source relational database management system that supports advanced features like transactions, subqueries, and triggers. It was used as the primary database in the banking application and managed using Flyway to ensure synchronization with the application code. The application used Spring Data JPA, which made it easy to interact with the database using Java-based entities and repositories.

Spring Data JPA

Spring Data JPA is a high-level programming model for interacting with relational databases in Spring applications. It uses JPA annotations to map Java entities to database tables and provides standard methods for querying and manipulating data. Spring Data JPA also supports advanced features like pagination and sorting. In the banking application, it was used to interact with the PostgreSQL database, simplifying data access and improving productivity.

Spring Security

Spring Security is a robust and widely used security framework for Java applications. It provides a robust and customizable authentication and authorization mechanism for web and non-web applications. In the bank application project, Spring Security ensured that the application was secure and that only authorized users could access sensitive information. Spring Security provides several features, such as authentication, authorization, and secure communication over HTTPS.

JJWT

JJWT (JSON Web Token) was used for authentication and authorization in the banking application's RESTful API. JJWT is a lightweight library for creating, signing, and parsing JSON Web Tokens, an industry standard for securely transmitting information between parties as a JSON object. With JJWT, a stateless authentication mechanism was implemented that relies on the token passed by the client on each request, enabling an efficient and secure way to protect the application's endpoints and resources.

Spring Cloud Netflix

Spring Cloud Netflix is a toolset for developing scalable and fault-tolerant microservices-based applications. It includes service discovery, client-side load balancing, and distributed tracing. In the bank application project, Spring Cloud Netflix was used to implement service discovery and client-side load balancing with Eureka client, enabling efficient and fault-tolerant architecture and easy management of the different services.

Webflux / WebClient

WebFlux is a Spring Boot framework for building web apps with a non-blocking approach, using an event-driven, non-blocking I/O architecture, making it highly scalable and performant. For the banking app, WebFlux was used to handle high-volume concurrent requests and improve responsiveness. WebClient is a WebFlux component used as a non-blocking, reactive HTTP client to make requests to external APIs like currency conversion. This allows the app to retrieve and process data from these services without blocking the main thread.

MongoDB

In the context of the bank application, MongoDB was used as the database to store transaction data. MongoDB is a NoSQL database that is document-oriented, meaning that data is stored in flexible and dynamic JSON-like documents instead of relational databases' traditional rows and columns format. The use of MongoDB in this project allowed for the storage and retrieval of data in a more efficient and scalable way, as well as providing more flexibility in terms of data modeling. Additionally, MongoDB's ability to handle large amounts of data and its ability to scale horizontally made it a good fit for the transactional nature of the bank application.

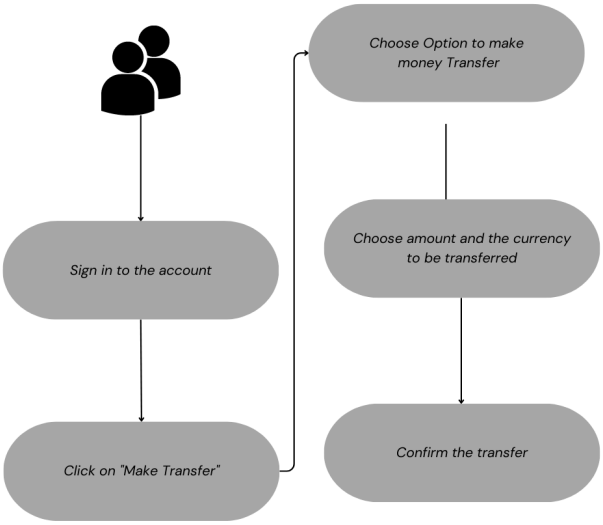
Testing /Junit

JUnit is a widely-used testing framework for Java applications that offers a range of tools for creating and executing automated tests. In the bank application, JUnit was used to ensure the code was functioning correctly and meeting the project's requirements. Its tools, including assertions, test fixtures, and test runners, enabled the creation of tests that could be run in isolation or as part of a larger test suite. The integration of JUnit with other testing tools, such as Mockito, allowed for a more comprehensive testing solution, increasing the reliability and quality of the bank application's code.

User Scenario Coverage

In the following section two real life user scenarios are introduced: one for customer-user and one for bank-user.

Customer User Makes Transaction

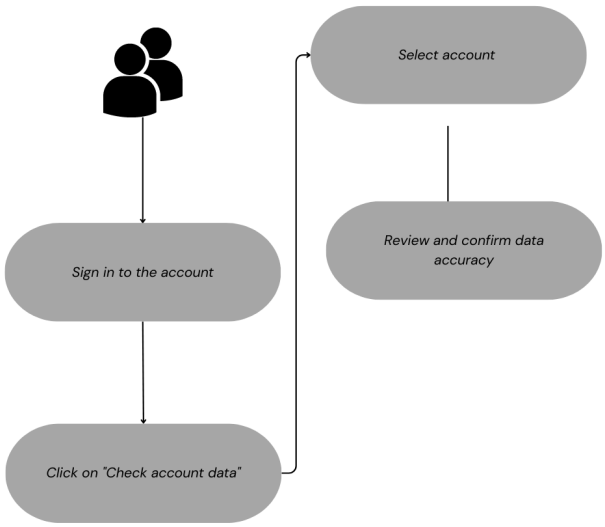


- The customer user logs in to their account and selects the option to make a transfer.
- They choose to transfer money from their physical entity account to their legal entity account.
- The physical entity account has a balance of 1000 USD, and the legal entity account has a balance of 0 EUR.
- The customer-user enters the transfer amount of 500 USD and selects the option to convert the currency to EUR.
- The application automatically converts the amount to EUR using the current exchange rate and displays the converted amount of 428.90 EUR to the customer user.
- The customer-user confirms the transfer, and the application deducts 500 USD from the physical entity account and adds 428.90 EUR to the legal entity account.

- The application updates the transaction history of both accounts and sends a notification to the customer-user confirming the transaction.

Note: The exchange rate used for the currency conversion is being updated regularly to ensure accuracy.

Bank User Reviews Customer's Data



- A bank user logs into the system with their credentials.
- The bank user navigates to the account management module and selects the "Check Account Balance" option.
- The system presents a list of all available accounts.

- The bank user selects an account and provides the required authorization to access the account.
- The system retrieves the account balance and presents it to the bank user.
- The bank user reviews the account balance and confirms that it is accurate.
- The bank user can perform additional actions on the account, such as transferring funds or updating account information.
- Once the bank user is finished, they log out of the system.

System Configurations

System Requirement

Hardware Requirements

Minimum of 8GB RAM allocated to the Java application to ensure optimal performance.

Utilize a quad-core or higher CPU for efficient processing of concurrent user requests.

Network Requirements

Ensure that port 443 is open to allow secure communication over HTTPS. This port should be accessible for incoming and outgoing traffic to facilitate secure transactions and data exchange.

Operating System

The bank management system is implemented in Java Spring Framework and is designed to be platform-independent, ensuring compatibility with various operating systems. The supported

operating systems include but are not limited to Windows, macOS, and Linux.

Database Requirements

Define the compatible database system(s) and versions for the bank management system. Specify any necessary configurations, such as connection details, credentials, and required database privileges.

Software Dependencies

Identify any specific software dependencies or libraries required by the bank management system, such as the Java Development Kit (JDK) version, Spring Boot framework, and any additional third-party libraries.

Server Configuration

Configuration files are essential to any software application as they contain information about how the system should behave and operate. In the case of a banking application, the configuration files include settings for connecting to the database, configuring the user interface, and defining the application's behavior.

The configuration file used in the Spring Framework is the application.properties file. This file contains settings for the Spring Boot application, such as the server port, database connection details, and logging configuration. It uses a simple key-value format to define these properties. Below are described the properties which should be configured before starting the application.

spring.application.name	the name of the application
spring.flyway.schemas	the database schema to be used for Flyway database migrations
spring.datasource.url **spring.datasource.username** **spring.datasource.password**	configure the database connection settings
spring.jpa.properties.hibernate.default_schema	the default schema used for JPA entities
spring.jpa.hibernate.ddl-auto	determines how the database schema is created or updated when Hibernate runs
spring.jpa.show-sql	enables or disables the printing of SQL statements to the console
jwt.secret **jwt.expiration**	used for JSON Web Token (JWT) authentication

```
spring.application.name=bank-user-management-application
spring.flyway.schemas=bank_management_um
spring.datasource.url=jdbc:postgresql://localhost:5432/db_bank_management_um
spring.datasource.username=****
spring.datasource.password=****
spring.jpa.properties.hibernate.default_schema=bank_management_um
spring.jpa.hibernate.ddl-auto=none
spring.jpa.show-sql=true
jwt.secret=test
jwt.expiration=604800

eureka.instance.prefer-ip-address=true
eureka.client.registerWithEureka=true
eureka.client.fetchRegistry=true
eureka.client.service-url.defaultZone=http://localhost:9090/api/v1/discovery/eureka
```

The remaining properties are related to Eureka, a service discovery and registration tool. They configure the client to register with and fetch information from a Eureka server running at http://localhost:9090/api/v1/discovery/eureka.

Testing Strategy

Testing is an important aspect of software development as it ensures that the software works as expected and helps to identify and fix bugs and errors. Unit testing is a type of testing that involves testing individual components or units of code in isolation to ensure that they work correctly. Unit testing helps to catch bugs early in the development process and provides developers with confidence that their code works as intended.

The bank application has undergone rigorous testing through unit tests, which have provided approximately 80% test coverage, ensuring that critical functionalities are thoroughly tested.

This testing process has helped ensure that the application is reliable, robust, and performs as expected. The unit tests for the bank application were implemented using JUnit.

Open Issues

Scalability

Problem: As the user base grows and the application handles more transactions, it may need to be optimized for scalability to ensure it can handle increased traffic and usage without slowing down or crashing. This could involve using distributed systems and load-balancing techniques to improve performance and handle large volumes of data.

To address this issue, there are several possible solutions:

1. Load Balancing: One approach to scaling up the application is implementing load balancing to distribute the load among multiple servers. The load balancer can direct incoming requests to the server with the least load, thus ensuring that the processing load is evenly distributed among the servers.
2. Caching: Caching is another technique that can help improve the application's performance. By caching frequently accessed data, such as customer information and transaction records, the application can reduce the number of database calls, reducing the application's response time.
3. Sharding: Another solution to scaling up the application is to implement sharding. *Sharding* is a technique that involves breaking up the database into smaller, more manageable pieces. Each shard is then stored on a separate server, and data is distributed among the shards based on a predefined key. This approach can help improve the

application's performance by reducing the amount of data that needs to be processed on a single server.

Integration

Problem: The application may need to be integrated with other systems and services to provide a complete banking experience for users. For example, it could be integrated with third-party payment processors, accounting software, or financial management tools.

To address this issue, there are several possible solutions:

1. Implement middleware: Middleware can act as a bridge between the bank application and third-party systems. By implementing middleware, the bank can streamline the integration process and ensure data is accurately and securely transferred between systems.
2. Use pre-built integrations: Some third-party systems offer pre-built integrations with popular banking applications. By leveraging these pre-built integrations, the bank can quickly and easily connect with these systems without requiring extensive custom development work.

Usability

Problem: The application's user interface could be improved to make it more intuitive and user-friendly. This could involve conducting user research to identify pain points and areas for improvement, as well as implementing design changes to make the application more visually appealing and easier to use.

To address this issue, there are several possible solutions:

1. It is important to focus on user-centered design principles and ensure that the application is designed with the needs and preferences of the users in mind. This can be achieved through various techniques such as user testing, feedback collection, and user research.
2. Another solution is to provide clear and concise documentation and help resources to assist users in understanding how to use the application. This can include user guides, FAQs, and tutorials.
3. It is also important to provide a streamlined and intuitive user interface that is easy to navigate and understand. This can be achieved by using clear and consistent design elements, such as icons and labels, and by following established usability guidelines.
4. In addition, providing personalized experiences for users can improve usability. This can be achieved through personalized dashboards, notifications, and alerts that provide users with relevant and timely information.
5. Finally, it is important to regularly review and optimize the application's usability through ongoing user feedback and testing. This will ensure that the application remains user-friendly and effective as it evolves.