DEMO CREDIT WALLET SERVICE OVERVIEW

SYSTEM OVERVIEW

The Demo Credit Wallet Service is designed as a comprehensive financial transaction system aimed at facilitating seamless digital transactions for users. Its primary purpose is to provide a secure and efficient platform where users can create and manage their own digital wallets, allowing for easy funding and tracking of transactions.

One of the key features of the Demo Credit Wallet Service is user account creation. This feature enables users to set up their accounts quickly and securely, ensuring that personal information is protected. Users can easily register by providing minimal information, which is then validated to prevent unauthorized access. Once registered, users can log in to their accounts using secure credentials.

The wallet funding capabilities are another vital aspect of the service. Users can deposit funds into their wallets using various methods, including bank transfers and credit card payments. This flexibility allows users to choose their preferred funding source, making it convenient for them to add money to their wallets whenever necessary.

Additionally, the service offers transaction history tracking, enabling users to monitor their financial activities. Every transaction is logged, providing users with a detailed overview of their spending patterns and balances. This transparency is crucial for users to manage their finances effectively and ensures that they have access to all necessary information at their fingertips.

The underlying technology stack for the Demo Credit Wallet Service comprises modern and robust components, including Node.js for server-side development, TypeScript for type-safe coding, and MySQL as the relational database management system. Furthermore, JSON Web Tokens (JWT) are employed for authentication, providing a secure method for verifying user identities and protecting sensitive information throughout the transaction process. This combination of technologies ensures a reliable and scalable service that meets the demands of its users.

ARCHITECTURE

The Demo Credit Wallet Service utilizes a clean architecture pattern to ensure separation of concerns, testability, and maintainability throughout the application. The directory structure is organized to reflect the various layers of the architecture, which enhances readability and supports the scalability of the system. Below is an overview of the directory structure:

```
/src
    /controllers
    /services
    /repositories
    /models
    /routes
    /middlewares
    /utils
    /config
```

- **controllers**: This layer handles incoming requests and directs them to the appropriate services.
- **services**: Contains business logic, orchestrating operations between controllers and repositories.
- **repositories**: An abstraction layer that communicates with the database, implementing data access patterns.
- **models**: Represents the data structures used within the application.
- **routes**: Defines the API endpoints and their corresponding controller functions.
- **middlewares**: Contains functions that intercept requests for tasks such as authentication and logging.
- **utils**: Houses utility functions that can be reused across different parts of the application.
- **config**: Manages configuration settings for the application.

In addition to the clean architecture, several design patterns are implemented within the system to promote best practices:

1. **Repository Pattern**: This pattern isolates data access logic, allowing for easier testing and maintenance. It provides a collection-like interface for accessing domain objects.

2. **Service Layer Pattern:** This pattern encapsulates the business logic, making it reusable and easier to manage. It acts as a bridge between the controllers and repositories.

3. **Dependency Injection:** This technique allows for the decoupling of the components of the application, making it easier to manage dependencies and facilitating unit testing.

4. **Factory Pattern:** Utilized for creating objects without specifying the exact class of the object that will be created, improving flexibility and scalability.

5. **Observer Pattern:** This pattern is used for establishing a subscription model, allowing one part of the system to notify other parts about changes in state, which is particularly useful for real-time features like transaction updates.

Through the combination of clean architecture and these design patterns, the Demo Credit Wallet Service achieves a robust and maintainable codebase that is well-suited for future enhancements and modifications.

DATABASE DESIGN

The entity-relationship diagram for the database schema of the Demo Credit Wallet Service comprises three primary entities: USERS, WALLETS, and TRANSACTIONS. Each of these entities has specific attributes, with designated primary keys (PK), unique keys (UK), foreign keys (FK), and data types tailored to their requirements.

USERS ENTITY

- **Attributes:**
  - `user_id` : UUID (PK)
  - `username` : STRING (UK)
  - `email` : STRING (UK)
  - `password_hash` : STRING
  - `created_at` : TIMESTAMP
  - `updated_at` : TIMESTAMP
- **Description:** The USERS entity holds information about each user registered in the system. The `user_id` serves as the unique identifier for each user, while `username` and `email` must be unique across the database. Timestamps track account creation and updates.

WALLETS ENTITY

- **Attributes:**
  - `wallet_id` : UUID (PK)
  - `user_id` : UUID (FK referencing USERS.user_id)
  - `balance` : DECIMAL
  - `currency` : ENUM('USD', 'EUR', 'GBP', 'JPY')
  - `created_at` : TIMESTAMP
  - `updated_at` : TIMESTAMP
- **Description:** The WALLETS entity represents each user's digital wallet. The `wallet_id` uniquely identifies each wallet, while `user_id` links the wallet to a specific user. The `balance` holds the current amount of money in the wallet, and `currency` designates the type of currency being used. Timestamps provide history on wallet creation and modifications.

TRANSACTIONS ENTITY

- **Attributes:**
  - `transaction_id` : UUID (PK)
  - `wallet_id` : UUID (FK referencing WALLETS.wallet_id)
  - `amount` : DECIMAL
  - `transaction_type` : ENUM('DEPOSIT', 'WITHDRAWAL', 'TRANSFER')
  - `timestamp` : TIMESTAMP
  - `metadata` : JSONB
- **Description:** The TRANSACTIONS entity records each financial transaction performed by users. The `transaction_id` serves as a unique identifier for each transaction, while `wallet_id` connects the transaction to the relevant wallet. The `amount` indicates the transaction value, and `transaction_type` specifies the nature of the transaction. The `metadata` field is used to store additional information about the transaction in a flexible JSON format.

This schema design effectively captures the core functionalities of the Demo Credit Wallet Service, facilitating efficient data management and retrieval for user interactions.

API DESIGN

The Demo Credit Wallet Service provides a comprehensive set of API endpoints categorized primarily into authentication and wallet operations. The following sections outline the available endpoints, their purposes, and example request methods for clarity.

AUTHENTICATION ENDPOINTS

1. User Registration
    - **Endpoint:** `/api/register`
    - **Method:** `POST`
    - **Description:** This endpoint allows new users to register by submitting their username, email, and password. The service validates the input and creates a new user account.
    - **Example Request:**

    ```
    {
      "username": "john_doe",
      "email": "john@example.com",
      "password": "securePassword123"
    }
    ```

2. User Login
    - **Endpoint:** `/api/login`
    - **Method:** `POST`
    - **Description:** This endpoint authenticates users by verifying their email and password. Upon successful authentication, a JSON Web Token (JWT) is returned for session management.
    - **Example Request:**

    ```
    {
      "email": "john@example.com",
      "password": "securePassword123"
    }
    ```

WALLET OPERATIONS ENDPOINTS

1. Fund Wallet
    - **Endpoint:** `/api/wallet/fund`
    - **Method:** `POST`
    - **Description:** This endpoint allows users to add funds to their digital wallet. Users must include the amount and payment method in the request.
    - **Example Request:**

```
{
  "amount": 100.00,
  "payment_method": "credit_card"
}
```

2. Transfer Funds
    - **Endpoint:** `/api/wallet/transfer`
    - **Method:** `POST`
    - **Description:** This endpoint facilitates the transfer of funds between user wallets. The request must contain the recipient's wallet ID and the transfer amount.
    - **Example Request:**

```
{
  "recipient_wallet_id": "wallet-12345",
  "amount": 50.00
}
```

3. Withdraw Funds
    - **Endpoint:** `/api/wallet/withdraw`
    - **Method:** `POST`
    - **Description:** This endpoint allows users to withdraw funds from their wallet to an external account. The request should specify the amount to withdraw.
    - **Example Request:**

```
{
  "amount": 30.00
}
```

4. Check Balance
   - **Endpoint:** `/api/wallet/balance`
   - **Method:** `GET`
   - **Description:** This endpoint retrieves the current balance of the authenticated user's wallet.
   - **Example Request:** (No body required)

These API endpoints collectively enable users to manage their accounts and perform essential wallet operations effectively. Each endpoint is designed with user security and ease of use in mind, ensuring a reliable transaction experience.

IMPLEMENTATION DETAILS

The implementation of transaction management in the Demo Credit Wallet Service relies on KnexJS, a SQL query builder for Node.js that simplifies database interactions while ensuring compliance with ACID (Atomicity, Consistency, Isolation, Durability) principles. This is crucial for maintaining the integrity of financial transactions, particularly when transferring funds between wallets.

TRANSACTION MANAGEMENT WITH KNEXJS

To facilitate a fund transfer between two wallets, the process must ensure that either both operations succeed or neither does. This can be achieved using KnexJS's transaction handling features. Below is an example method that illustrates how to transfer funds:

```
const transferFunds = async (senderWalletId,
recipientWalletId, amount) => {
  return await knex.transaction(async (trx) => {
    // Check if sender has sufficient balance
    const senderWallet = await trx('WALLETS')
      .select('balance')
      .where('wallet_id', senderWalletId)
      .first();

    if (senderWallet.balance < amount) {
      throw new Error('Insufficient funds');
    }

    // Deduct amount from sender's wallet
    await trx('WALLETS')
      .where('wallet_id', senderWalletId)
      .update({ balance: senderWallet.balance -
amount });

    // Add amount to recipient's wallet
    const recipientWallet = await trx('WALLETS')
      .select('balance')
      .where('wallet_id', recipientWalletId)
      .first();

    await trx('WALLETS')
      .where('wallet_id', recipientWalletId)
      .update({ balance: recipientWallet.balance +
amount });

    // Log the transaction
    await trx('TRANSACTIONS').insert({
      wallet_id: senderWalletId,
      amount: -amount,
      transaction_type: 'TRANSFER',
      timestamp: new Date(),
    });

    await trx('TRANSACTIONS').insert({
      wallet_id: recipientWalletId,
      amount: amount,
      transaction_type: 'TRANSFER',
      timestamp: new Date(),
    });
  });
};
```

In this example, the transferFunds function executes a series of operations within a transaction. It first checks the sender's balance, deducts the amount from the sender's wallet, and adds it to the recipient's wallet. If any operation fails, the transaction is rolled back, ensuring that the database remains consistent.

INTEGRATION WITH ADJUTOR KARMA API

To enhance security, the Demo Credit Wallet Service integrates with Lendsqr's Adjutor Karma API for blacklist verification. Prior to executing a fund transfer, the service queries the Adjutor Karma API to confirm that neither wallet involved in the transaction is blacklisted. This step is essential to prevent transactions that may involve fraudulent or risky accounts.

The integration can be implemented as follows:

```
const isBlacklisted = async (walletId) => {
  const response = await axios.get(`https://adjutor-
karmapi.com/blacklist/${walletId}`);
  return response.data.is_blacklisted;
};

const transferFundsWithBlacklistCheck = async
(senderWalletId, recipientWalletId, amount) => {
  const isSenderBlacklisted = await
isBlacklisted(senderWalletId);
  const isRecipientBlacklisted = await
isBlacklisted(recipientWalletId);

  if (isSenderBlacklisted || isRecipientBlacklisted) {
    throw new Error('One of the wallets is blacklisted');
  }

  return transferFunds(senderWalletId, recipientWalletId,
amount);
};
```

This method first checks both wallets against the blacklist before proceeding with the fund transfer, ensuring compliance with regulatory requirements and enhancing the overall security of the transaction process.

SECURITY MEASURES

The Demo Credit Wallet Service employs a multi-faceted approach to security, ensuring the integrity and confidentiality of user data while safeguarding transactions. Key security measures include robust authentication procedures, data protection strategies, and transaction security features.

AUTHENTICATION PROCEDURES

At the heart of the security framework is the implementation of JSON Web Token (JWT)-based authentication. Upon successful login, users receive a JWT that is used to verify their identity for subsequent requests. This token-based system enhances security by eliminating the need to send sensitive credentials with every request, thereby minimizing the risk of interception. Additionally, passwords are securely hashed using industry-standard algorithms like bcrypt, ensuring that even if the database is compromised, user passwords remain protected.

DATA PROTECTION STRATEGIES

Data protection is paramount in any financial service. The Demo Credit Wallet Service employs encryption protocols to secure sensitive data at rest and in transit. All data transmitted between the client and server is encrypted using TLS (Transport Layer Security), preventing eavesdropping and man-in-the-middle attacks. Furthermore, sensitive user data stored in the database, such as email addresses and hashed passwords, is subject to stringent access controls, allowing only authorized personnel to interact with the data.

TRANSACTION SECURITY FEATURES

To ensure transaction security, the service incorporates input validation and audit logging mechanisms. Input validation checks user inputs to prevent common vulnerabilities, such as SQL injection and cross-site scripting (XSS). This step is critical in maintaining the integrity of the application and protecting against malicious attacks.

Audit logging is another vital aspect of security, as it tracks all user activities related to account management and financial transactions. This logging not only aids in monitoring suspicious activities but also provides an audit trail for compliance with regulatory standards.

By integrating these comprehensive security measures, the Demo Credit Wallet Service safeguards user accounts and transactions, fostering trust and confidence in the platform.

TESTING STRATEGY

The testing strategy for the Demo Credit Wallet Service is centered around ensuring the reliability and correctness of the application's functionalities, particularly within the WalletService using Jest as the testing framework. Jest provides a robust environment for conducting unit tests, allowing developers to verify that each function behaves as expected.

UNIT TESTING THE WALLETSERVICE

A critical component of the WalletService is the `transfer()` method, which facilitates fund transfers between user wallets. Ensuring that this method works accurately is essential for maintaining the integrity of financial transactions. Below are examples of unit tests that address both successful fund transfers and scenarios where insufficient funds are present.

Test for Successful Fund Transfer

The first test checks whether the transfer function correctly executes a fund transfer when the sender has a sufficient balance.

```
const WalletService = require('./WalletService');
const { createMockWallet } = require('./mocks');

test('should transfer funds successfully', async () => {
  const senderWallet = createMockWallet({ balance:
100 });
  const recipientWallet = createMockWallet({ balance:
50 });

  const walletService = new WalletService();
  await walletService.transfer(senderWallet.id,
recipientWallet.id, 50);

  expect(senderWallet.balance).toBe(50);
  expect(recipientWallet.balance).toBe(100);
});
```

In this example, the test setup creates mock wallet instances, simulating a sender with a balance of 100 and a recipient with 50. The `transfer()` method is called, and assertions are made to confirm the updated balances.

Test for Insufficient Funds

The second test verifies that the `transfer()` method properly handles attempts to transfer funds when the sender's balance is insufficient.

```
test('should throw error when sender has insufficient
funds', async () => {
  const senderWallet = createMockWallet({ balance: 30 });
  const recipientWallet = createMockWallet({ balance:
50 });

  const walletService = new WalletService();

  await expect(walletService.transfer(senderWallet.id,
recipientWallet.id, 50))
      .rejects
      .toThrow('Insufficient funds');
});
```

This test simulates a scenario where the sender has only 30 in their wallet while attempting to transfer 50. The expectation is that the method will throw an error, thus ensuring that the application prevents unauthorized transactions.

COMPREHENSIVE TESTING APPROACH

By employing these unit tests, the Demo Credit Wallet Service can ensure that critical functionalities like fund transfers are thoroughly vetted against both expected and unexpected behaviors. This comprehensive testing approach, combined with continuous integration practices, reinforces the overall reliability and quality of the application.

SETUP AND DEPLOYMENT

To successfully set up and deploy the Demo Credit Wallet Service, there are several prerequisites and steps that need to be followed. This section outlines the required software versions, installation steps, and necessary environment variables for the application.

PREREQUISITES

Before starting the setup process, ensure that the following software is installed on your system:

- **Node.js**: Version 14.x or later
- **MySQL**: Version 5.7 or later
- **npm**: Version 6.x or later (comes with Node.js)
- **Knex.js**: To manage database migrations and queries

INSTALLATION STEPS

1. **Clone the Repository** Begin by cloning the application repository from GitHub:

   ```
   git clone https://github.com/yourusername/demo-
   credit-wallet-service.git
   cd demo-credit-wallet-service
   ```

2. **Install Dependencies** Once inside the project directory, install the required dependencies using npm:

   ```
   npm install
   ```

3. **Run Migrations** After installing the dependencies, set up the database by running the migrations:

   ```
   npx knex migrate:latest --env development
   ```

4. **Start the Development Server** Now, you can start the development server to run the application locally:

```
npm run dev
```

5. **Run Tests** It's important to verify that everything is set up correctly by running the tests:

```
npm test
```

REQUIRED ENVIRONMENT VARIABLES

To configure the application correctly, you need to set the following environment variables in a `.env` file located in the root of your project:

- **DATABASE_URL**: The connection string for your MySQL database.
    - Example: `mysql://user:password@localhost:3306/database_name`
- **JWT_SECRET**: A secret key used for signing JSON Web Tokens.
    - Example: `your_jwt_secret_key`
- **NODE_ENV**: The environment in which the application is running (e.g., development, production).
    - Example: `development`
- **PORT**: The port on which the server will run.
    - Example: `3000`

Setting up these environment variables ensures that the application can run smoothly with the necessary configurations.

CONCLUSION

The Demo Credit Wallet Service stands as a production-ready solution that effectively addresses the needs of users seeking a secure and efficient digital wallet experience. Throughout its development, the service has adhered to best practices in software engineering, ensuring a high standard of code quality and maintainability. The incorporation of a clean architecture and design patterns such as the Repository and Service Layer patterns has facilitated a robust codebase that is both scalable and easy to manage.

Security is paramount in financial transaction processing, and the Demo Credit Wallet Service has been meticulously designed to protect user data and transaction integrity. By implementing advanced authentication mechanisms, such as JSON Web Tokens, alongside encryption protocols for data at rest and in transit, the service ensures that user interactions are safeguarded against unauthorized access and potential threats.

Moreover, the transaction management system leverages KnexJS to uphold ACID principles, guaranteeing that all financial transactions are processed reliably and consistently. This attention to detail in the design and architecture of the service reinforces its capability to handle secure financial transactions seamlessly.

In summary, the Demo Credit Wallet Service not only fulfills its intended purpose but also sets a benchmark in the digital wallet landscape by delivering a secure, robust, and user-friendly platform for managing financial transactions.