PIMONO
SOFTWARE DESIGN L.L.C

Technical Assignment: "Mini Wallet" Application (High-Performance Version)

Introduction & Goal

We would like you to develop a simplified digital wallet application that allows users to transfer money to each other, similar to the projects we work on at our company. The primary goal of this project is to understand your Full Stack (Laravel & Vue.js) skills, your code quality, and your problem-solving approach, especially when dealing with **high-traffic, large-scale financial systems**.

Technology Stack

- **Backend:** Laravel (The latest stable version is preferred)
- **Frontend:** Vue.js (The latest stable version is preferred; Composition API is highly encouraged)
- **Database:** MySQL or PostgreSQL
- **Real-time:** Pusher

Project Requirements

1. Backend - Laravel API

The database should have at least two tables: **`users`** and **`transactions`**.

- **`users` table:** In addition to the standard Laravel columns, it must include a **`balance`** column (decimal type) to store the user's current funds.
- **`transactions` table:** It should record every transfer, including details like `sender_id`, `receiver_id`, `amount`, `commission_fee`, `timestamp`, etc.

Required API Endpoints:

1. **`GET /api/transactions`**: Returns the transaction history (both incoming and outgoing) and the current balance for the authenticated user.
2. **`POST /api/transactions`**: Executes a new money transfer. The request body will contain the `receiver_id` and `amount`.

Business Logic & Performance Expectations:

- **High Concurrency:** The system must be designed to handle **hundreds of transfers per second**. This requires you to develop a solution that prevents issues like **race conditions** and handles potential **database locking** during balance updates.
- **Large-Scale Data:** You must assume that the `transactions` table will eventually contain **millions of rows**. Therefore, your method for calculating or storing a user's balance must be designed with this scalability in mind. Calculating the balance by querying the entire `transactions` table on-the-fly is not an acceptable solution.
- **Commission Calculation:** For every successful transfer, a commission of **1.5%** of the transferred amount must be charged. This commission should be debited from the sender. For example, if User A sends 100.00 units to User B:
  - A total of **101.50** units will be debited from User A's balance.
  - **100.00** units will be credited to User B's balance.
- **Data Integrity:** The money transfer process must be atomic. If any part of the process fails (e.g., debiting the sender, crediting the receiver, or creating the transaction record), the entire operation must be rolled back, and no changes should be permanently saved to the database. Data consistency is paramount.
- **Validation:** All incoming requests must be thoroughly validated (e.g., does the `receiver_id` exist, is the `amount` a positive number, is the sender's balance sufficient, etc.).

2. Real-time Updates (Mandatory)

- After a successful transfer, an event must be broadcast via **Pusher** using Laravel's broadcasting system.
- This event must be sent to both the sender and the receiver.
- The frontend must listen for this event and **instantly update** the relevant users' transaction lists and balances without requiring a page refresh.

3. Frontend - Vue.js Interface

A simple and functional user interface is expected.

- **Transfer Page:**
  - A form to initiate a new transfer, containing:
    - An input field for the recipient's user **ID**.
    - An input field for the amount to be sent.
    - A "Send" button.
- **Transaction History & Balance:**
  - A section that lists all of the user's past transactions, populated by the `/api/transactions` endpoint.
  - A section that displays the user's current balance.
  - This entire section must be automatically updated by the real-time events received from Pusher.

Bonus Points (Optional)

- **Frontend Enhancements:** Implementing user experience improvements such as real-time form validation, currency formatting, or user feedback messages.

Evaluation Criteria

- **Scalable Balance Management:** Your approach to calculating and updating balances in a way that remains performant under high traffic and with millions of transaction records.
- **Correct Real-Time Integration:** Proper implementation of event broadcasting via Pusher and handling these events on the frontend to update the UI instantly.
- **Code Quality:** Clean, readable, maintainable, and well-structured code that adheres to common standards (e.g., PSR-12).
- **Problem-Solving:** Your approach to critical challenges, particularly regarding high concurrency and data integrity.
- **Security:** Implementation of basic security measures (e.g., validation, authorization).
- **Git Usage:** A clean and understandable commit history with meaningful commit messages.

Submission

- Please upload your project to a Git provider (e.g., GitHub, GitLab). Once your work is complete, please email the repository link to **info@pimono.ae**.
- The root of the project must contain a **`README.md`** file with clear instructions on how to set up and run the application.

We wish you the best of luck!