Mini Banking Platform

Overview

Build a simplified banking platform that demonstrates your ability to handle financial transactions, implement proper accounting principles, and create a user-friendly interface. The system should maintain data integrity while providing a smooth user experience.

Time Limit: This assessment is time-boxed to 48 hours from when you begin. Please submit your work within this timeframe regardless of completion status.

Priority: Given the time constraints, prioritize backend correctness and data integrity over frontend polish. A robust, well-architected backend with a simple UI is preferable to a visually polished frontend with incomplete financial logic.

Technical Requirements

Backend

- **Language**: Node.js with NestJS or Go
- **Database**: PostgreSQL
- **Architecture**: RESTful API
- **Authentication**: JWT or session-based (your choice)

Frontend

- **Framework**: React 19+ or Next.js
- **Styling**: Tailwind (+ your choice - simple, clean interface is sufficient)
- **State Management**: Your choice (Context API, Redux, Zustand, etc.)
- **UI Expectations**: Focus on functionality and correct data flow. Polish and animations are nice-to-have but not essential.

Core Requirements

1. Database Design

Implement a **double-entry ledger system** for financial integrity:

- Ledger entries serve as the authoritative audit trail for all transactions
- Each transaction creates balanced entries (sum of amounts must equal zero)
- Account balances should be maintained for performance
- Ensure consistency between ledger entries and account balances

Required Tables (minimum):

- `users` : User information
- `accounts` : User currency accounts with balances (each user has USD and EUR accounts)
- `ledger` : Double-entry bookkeeping records (using positive/negative amounts)
- `transactions` : High-level transaction records for user-facing history

2. User Management

- **User Creation**: Choose one approach:
  - **Option A**: Implement registration endpoint (`POST /auth/register`)
  - **Option B**: Create at least 3 pre-seeded test users on initialization
- **Account Structure**: Each user automatically has:
  - 1 USD account (initial balance: $1000.00)
  - 1 EUR account (initial balance: €500.00)
- **Authentication**: Implement login functionality

3. API Endpoints

Authentication

- `POST /auth/login` - User login
- `POST /auth/register` - User registration (if Option A chosen)
- `GET /auth/me` - Get current user info

Account Operations

- `GET /accounts` - List user's accounts with calculated balances
- `GET /accounts/:id/balance` - Get specific account balance

Transaction Operations

- `POST /transactions/transfer` - Transfer between users (same currency)
- `POST /transactions/exchange` - Currency exchange within user's accounts

Transaction History

- `GET /transactions` - List transactions with filters:
  - `type` : transfer, exchange
  - `page` & `limit` : Pagination

4. Frontend Features

Dashboard Page

- Display current balance for each wallet (USD and EUR)
- Show last 5 transactions
- Forms for:
  - Transfer
  - Exchange

Transaction Pages

- **Transfer Form**
  - Select recipient (dropdown or input)
  - Select currency
  - Enter amount
  - Validation and error display
- **Exchange Form**
  - Select source currency
  - Enter amount
  - Show converted amount (1 USD = 0.92 EUR)
  - Display exchange rate

Transaction History Page

- Table/list of all transactions
- Filter by transaction type
- Pagination controls

Note: Given the time constraints, prioritize functionality over visual design. Ensure forms work correctly, data displays accurately, and errors are handled properly.

Business Requirements

1. **Insufficient Funds**
   - Transfers and exchanges must not exceed available balance
   - System must handle concurrent transaction attempts
2. **Transaction Integrity**
   - All financial operations must maintain data consistency
   - Ledger and balances must remain synchronized
   - Partial transaction completion is not acceptable
3. **Currency Precision**
   - All monetary amounts must maintain 2 decimal place precision
   - No rounding errors should affect user balances
4. **Exchange Operations**
   - Fixed exchange rate: 1 USD = 0.92 EUR
   - Exchange calculations must be transparent to users

Technical Evaluation Criteria

Must Have (70%)

- [ ] Functional double-entry ledger implementation
- [ ] Ledger entries properly maintained as audit trail
- [ ] Account balances kept in sync with ledger
- [ ] All transaction types working correctly
- [ ] Proper handling of concurrent operations
- [ ] Prevention of invalid states (negative balances, etc.)
- [ ] Clean, organized code structure
- [ ] Authentication working
- [ ] Functional user interface with working forms and data display

Should Have (20%)

- [ ] Comprehensive error messages
- [ ] Proper API error handling
- [ ] Loading states in UI

- [ ] Database migrations/seeders (required if using pre-seeded users)
- [ ] Environment configuration (.env)
- [ ] Input validation on both frontend and backend
- [ ] Transaction confirmation before processing

Nice to Have (10%)

- [ ] Unit tests for critical financial logic
- [ ] Balance reconciliation/verification endpoint
- [ ] API documentation (Swagger/OpenAPI)
- [ ] Docker setup
- [ ] Polished UI with good user experience
- [ ] Real-time balance updates (WebSockets)
- [ ] Transaction receipts/details modal
- [ ] Audit log for all operations

Submission Requirements

Deadline: Submit within 48 hours of receiving this assessment.

1. **Code Repository**
    - Push to a public Git repository (GitHub, GitLab, Bitbucket, etc.)
    - Include clear README with:
        - Setup instructions
        - Specify which user management approach was chosen (registration or pre-seeded)
        - Design decisions and trade-offs
        - Known limitations
        - Any incomplete features due to time constraints
2. **Database**
    - Document the double-entry ledger design
    - Explain your approach to maintaining balance consistency
3. **Deployment (Optional)**
    - Frontend: Deploy to Vercel or Netlify
    - Backend: Deploy to Render or alternative (free tier is sufficient)
    - Include deployment URLs in README
    - Ensure database is properly configured on hosting platform

Example Ledger Entries

Transfer $50 from User A to User B:

```
| account        | amount   |
|----------------|----------|
| User A (USD)   | -50.00   |
| User B (USD)   | +50.00   |
```

Exchange $100 to €92 for User A:

```
| account        | amount   |
|----------------|----------|
| User A (USD)   | -100.00  |
| User A (EUR)   | +92.00   |
```

Note: The sum of amounts for each transaction must always equal zero (balanced entries).

AI / LLM Usage

You are allowed (but not required) to use AI tools or LLMs while working on this assessment. However, to help us understand **how** you used these tools, you **must document your prompts**.

Requirements:

- If you use any AI/LLM to:
    - generate non-boilerplate code,
    - design architecture,
    - write tests,
    - draft documentation, or
    - significantly refactor or debug,

then you must provide a record of the prompts and how you used the responses.

- Create a file named `AI_USAGE.md` in the root of your repository containing:

  For each **substantial** AI interaction (not every single autocompletion), add an entry with:

  1. **ID**: A simple label (e.g. `AI-1`, `AI-2`, …).
  2. **Purpose**: What you were trying to achieve *Example*: «Generate initial Nest.js controller for transactions».
  3. **Tool & Model**: Which tool/model you used *Example*: GPT-5.2, Sonnet 4.5, etc.
  4. **Prompt**: The full prompt or message you sent to the AI (You may redact any sensitive or unrelated information if needed.)
  5. **How the response was used**: Brief explanation *Examples*:
     - «Used as-is for initial skeleton, then heavily modified.»
     - «Used only as inspiration; final code written manually.»
     - «Used to debug SQL transaction error; adapted the suggested fix.»

  A single entry can cover a short prompt–reply back-and-forth if they are all part of the same task.

- If you did **not** use any AI/LLM tools, explicitly state that in `AI_USAGE.md`:

```
# AI Usage

I did not use any AI or LLM tools for this assessment.
```

Questions to Consider

- How do you ensure transaction atomicity?
- How do you prevent double-spending?
- How do you maintain consistency between ledger entries and account balances?
- How would you handle decimal precision for different currencies?
- What indexing strategy would you use for the ledger table?
- How would you verify that balances are correctly synchronized?
- How would you scale this system for millions of users?

Important Notes:

- This assessment is strictly time-boxed to 48 hours from your start time
- Given the limited time, we recommend allocating more effort to backend architecture and financial logic
- Ensure core functionality works correctly before adding visual enhancements
- Document any trade-offs or incomplete features due to time constraints
- Focus on correctness, data integrity, and proper error handling as primary goals

Good luck!