⚒️

User and Transactions Relationship Visualization System – Intern Task Specification

Kickoff Date: Today

Deadline: 1 week from today

Project Goal

Build a prototype system to visualize relationships between user accounts using transaction data and shared attributes in a graph database environment. The system should use AWS Neptune or a similar graph database (e.g., Neo4j) for graph storage and querying, inspired by Flagright's FRAML API.

You should generate 100,000 transactions minimum and add that to the database. In the demo environment, we should be able sort, filter and explore these 100,000 Transactions.

Instructions

- Accepted languages & frameworks:
  - TypeScript, JavaScript, Python, Go, Rust (or any language you are comfortable with)
  - AWS Neptune, Neo4j, or any graph database of your choice
- The API endpoints should follow RESTful principles and return appropriate HTTP status codes and responses.
- Ensure meaningful variable and function names, proper indentation, and clean code structure.
- Provide clear instructions in the README file on how to run the server, test the API, and any additional setup requirements.
- Try to keep your commit messages clean and leave comments explaining your logic where necessary.
- Containerize the project for ease of deployment and scalability.
- After completing the assignment, submit the project repository link for review.

1. Data Storage & Graph Database

- Use AWS Neptune or a similar graph database (e.g., Neo4j) for storing user and transaction relationships, or more than one database if required for storage.
- The graph database should represent:
  - **Nodes**: Representing Users and Transactions
  - **Edges**: Representing relationships, including:
    - **User → User**: Shared Attributes or Direct Relationships
    - **User → Transaction**: Participated in a transaction
    - **Transaction → Transaction**: Linked through common attributes like IP or Device ID

2. Backend/API Requirements

- Expose the following API endpoints:
    - `POST /users`: Add or update user information.
    - `POST /transactions`: Add or update transaction details.
    - `GET /users`: List all users.
    - `GET /transactions`: List all transactions.
    - `GET /relationships/user/:id`: Fetch all connections of a user, including direct relationships and transactions.
    - `GET /relationships/transaction/:id`: Fetch all connections of a transaction, including linked users and other transactions.

3. Relationship Detection Logic

- Implement logic to automatically detect relationships and update the graph:
    - **Direct Transaction Links**:
        - *Credit Links*: When a user sends money to another user.
        - *Debit Links*: When a user receives money from another user.
    - **Shared Attribute Links**:
        - *Email Links*: Users sharing the same email addresses.
        - *Phone Links*: Users sharing the same phone numbers.
        - *Address Links*: Users sharing the same physical addresses.
        - *Payment Method Links*: Users sharing the same payment methods.
    - **Business Relationship Links (Optional)**:
        - Parent-Child Links
        - Legal Entity Links
        - Director Links
        - Shareholder Links
    - **Composite Links (Optional)**: Links that combine multiple types of relationships for stronger connection analysis.

4. Frontend Visualization Requirements

- Create a web-based visualization of the graph using a JavaScript graph library (e.g., Cytoscape.js or Vis.js).
- Display:
  - List of Users: Searchable and filterable.
  - List of Transactions: Searchable and filterable.
  - User Connections: Show all user-to-user and user-to-transaction links.
  - Transaction Connections: Show transaction-to-transaction links if common attributes are detected.
  - Edge coloring: Differentiate types of relationships (e.g., shared attribute vs. transaction link).

    An visual example of links could be like

5. Dockerization & Delivery

- Containerize the backend (API + Graph Database) and frontend for easy deployment.
- Provide a README with clear setup instructions, including AWS Neptune setup or Neo4j Docker configuration.
- Include data generation scripts to initialize the database with sample users and transactions for testing.
- Implement a data generation script that:
  - Supports any Docker auto-population on startup or manual triggering through an API endpoint or script.
  - Document approache in the README for flexibility.

6. Example Data (Required for Testing)

- Provide a script or JSON files to populate Neptune (or Neo4j) with sample data:
  - At least 5-10 Users with shared attributes.

    - At least 10-15 Transactions with a mix of direct and indirect links.
    - 3-5 examples of Shared Attributes (e.g., phone, email) that cause user-to-user links.
    - 2-3 Transaction-to-Transaction links based on IP, Device ID, or similar identifiers.
    - Ensure the data can be easily loaded for testing and demo purposes.

7. Bonus Features

- Graph Analytics: Add features like finding the shortest path between two users or transaction clustering.
- Export Features: Allow export of the graph data as CSV or JSON.
- Advanced Search Filters: Enable advanced filtering options in the frontend.

Deliverables:

- Fully functional backend with REST APIs
- Interactive frontend with visualization
- Dockerized environment for easy setup
- Documentation and usage instructions
- Example data for testing
- (Bonus) Graph analytics and export features