Technical Assessment

Backend Architecture and Design

Name: Okoro Jeremiah

Email: okorojeremiah91@gmail.com

Phone Number: 08101558612

1. API Structure

- **User Registration and Authentication**:
    - **Endpoints**:
        - /users/register (POST): Creates a new user account.
        - /users/login (POST): Authenticates a user and returns a secure token, such as JWT.
        - /users/logout (POST): Invalidates the current user's token, effectively logging them out.
    - **HTTP Methods and Request/Response Formats**:
        - POST requests will utilize JSON in the request body to send user data (e.g., username, password, email).
        - Successful responses will return JSON with relevant data, such as user ID and token, along with appropriate HTTP status codes (e.g., 200 OK, 201 Created).
        - Error responses will include error messages and corresponding HTTP status codes (e.g., 400 Bad Request, 401 Unauthorized).
- **User Profile Management**:
    - **Endpoints**:
        - /users/me (GET): Retrieves the current user's profile information.
        - /users/me (PUT): Updates the current user's profile information.
        - /users/me/password (PUT): Allows the current user to change their password.
    - **HTTP Methods and Request/Response Formats**:
        - Similar to user registration/authentication, data exchange will be done via JSON, with appropriate HTTP status codes provided for successful operations and error handling.

2. Scalability and Data Consistency

- **Load Balancing**: To ensure the API can handle high traffic, I'll employ load balancers like **Nginx** or **AWS ELB**, combined with horizontal scaling to distribute requests across multiple servers.
- **API Rate Limiting**: Rate limiting will be essential for preventing abuse. I'd implement a system like **Redis** to track and limit the number of requests per user, ensuring fair usage.
- **Statelessness**: Keeping the API stateless will ensure each request is independent, making it easier to scale across different server instances. No session state will be maintained between requests, with all required information being passed in the token.
- **Data Consistency**: For data consistency, I'll implement database transactions or optimistic locking mechanisms. This will ensure that concurrent updates (e.g., to user profiles) are handled efficiently without conflicts.

3. Pagination

- **Handling Large Datasets**:
  To handle large datasets like user lists and for scalability, I'll implement cursor-based pagination, which is more efficient for handling real-time data and ensures consistency in paginated data retrieval.

4. Efficient Data Loading Techniques

- **Database Indexing:** I'll ensure proper indexing on frequently queried fields (e.g., username, email) to drastically speed up query performance and data retrieval.
- **In-Memory Caching:** I'll use in-memory caches (like Redis) to store frequently accessed paginated data. This reduces the number of database queries, especially for data that doesn't change often or is accessed frequently.
    - **Example:** I'll cache the results of the first few pages of a list that is commonly accessed (e.g., top 100 transactions).
- **Per-User Cache:** For user-specific data (e.g., their transaction history), I'll cache the results of previous queries to avoid redundant database access.
- **Lazy Loading:** For related data (e.g., user's transaction history), I'll implement lazy loading, ensuring that related data is only fetched when explicitly requested by the client. This will reduce the initial load time and improve overall performance.
    - **Example:** For a list of users, I'll avoid loading users' complete profiles with each paginated response unless the client requests that information.

Final Thoughts:

This approach to designing the API is tailored to ensure scalability, efficiency, and maintainability. My focus is on creating a clean, well-documented API that's easy for frontend developers to consume. By prioritizing the use of industry-standard practices and tools, the API will be robust, scalable, and maintainable, handling high traffic while ensuring a responsive user experience. Optimizing for

scalability early on is essential as the user base grows, and the use of advanced techniques like cursor-based pagination and lazy loading, as described above, will ensure smooth and efficient operations as the system scales.