MoMo SMS Data Analysis - Design & Implementation Notes

1. Overview of the Approach

The MoMo SMS Data Analysis project was designed to process, analyze, and visualize mobile money transaction data extracted from SMS messages. The approach follows a clear separation of concerns, consisting of three main components: data extraction and ingestion, backend API development, and frontend visualization.

2. Data Extraction & Parsing Approach

The project begins with raw SMS data exported in XML format (from applications such as SMS Backup & Restore). The parsing logic uses Python’s xml.etree.ElementTree library to read the XML structure, iterate over SMS messages, and extract relevant fields such as sender, timestamp, transaction amount, and transaction type.
The extracted records are processed and categorized into standard transaction types (Received, Payment, Withdrawal, etc.) based on simple keyword pattern matching. The data is then inserted into a SQLite database for persistent storage.

3. Backend API Design (Flask + SQLite)

The backend is built using Flask, exposing RESTful API endpoints to serve data to the frontend. SQLite is chosen as the database to simplify deployment, eliminate the need for external DB servers, and allow file-based storage for easy development.
Flask-CORS is used to handle cross-origin requests between the frontend and backend. The database file (My_Database.db) is automatically created and populated after data ingestion.
The primary API endpoint (/api/messages) returns JSON-formatted transaction records upon frontend requests.

4. Frontend Design (HTML, TailwindCSS, Vanilla JS)

The frontend is built using pure HTML5, TailwindCSS for styling, and Vanilla JavaScript for logic. The visualization components utilize ApexCharts for dynamic and interactive charts.

Axios is used for HTTP requests to the Flask backend. The frontend reads data from the API, computes summary statistics, generates charts, and renders transaction tables.
All frontend code is modularized into components (charts, stats, transaction list, and analytics) to simplify maintainability and scalability.

5. Key Design Decisions

- Simplicity over complexity: SQLite chosen to eliminate need for external DB server and lower entry barrier.

- Lightweight stack: No heavy frontend frameworks; pure HTML/CSS/JS for faster load times and minimal dependencies.

- Clean separation of concerns: Parsing, backend API, and frontend visualization are fully decoupled for easier debugging and future extension.

- Use of XML as raw input format to directly utilize standard SMS export formats available on Android devices.

6. Challenges Encountered

- SMS message inconsistency: Variations in SMS formats from different providers required careful pattern matching and error handling in parsing logic.

- Cross-Origin Resource Sharing (CORS): Ensuring the frontend could reliably communicate with the backend locally across different ports and protocols.

- Port binding differences: Ensuring that frontend API URLs dynamically match the backend URL if server address changes (solution: configurable apiData.js file).

- Parsing edge cases: SMS messages containing partial data or unexpected content required fallback mechanisms during data extraction.

7. Future Improvement Areas

- Dockerization for simplified environment setup and deployment.

- Migration from SQLite to PostgreSQL for larger-scale production use.

- Implementation of user authentication and multi-user support.

- Addition of advanced filters, search, and date-range selectors on the frontend.

- Responsive mobile-first frontend design.