**Budget Tracker Web Application**

**System Architecture Document**

**1. Introduction**

This document outlines the system architecture for the Budget Tracker Web Application developed using the MERN (MongoDB, Express.js, EJS, Node.js) stack. The architecture reflects a modular, scalable, and secure structure suited for real-time personal finance tracking, enabling users to manage income, expenses, and budgets efficiently.

**2. Architectural Overview**

The application follows a **three-tier architecture** consisting of:

* **Frontend Layer** – Built with EJS (current), handles UI rendering and interactions.
* **Backend Layer** – Developed using Express.js and Node.js, handling business logic and RESTful APIs.
* **Database Layer** – MongoDB Atlas stores persistent data including transactions, user info, and recurring flags.

**3. System Components**

**3.1 Frontend (Client-side)**

* **Technology**: HTML5, CSS3 (Tailwind or Bootstrap), Vanilla JS or React.js (EJS in current setup)
* **Responsibilities**:
  + Display dashboards, charts, and forms and recurring/pinned status icons.
  + Send API requests to backend (e.g., add/edit transactions).
  + Validate recurring logic (e.g., show next scheduled date) and highlight pinned entries
  + Handle user interactions and provide visual feedback.
  + Implement client-side validation and routing.

**3.2 Backend (Server-side)**

* **Technology**: Node.js with Express.js
* **Responsibilities**:
  + Define and manage RESTful APIs.
  + Authenticate users using JWT and bcrypt.
  + Handle business logic such as budgeting limits, transaction categorization.
  + Logic for managing recurring transactions:
* Recognize recurring and recurringType (e.g., monthly)
* Automatically duplicate entries on scheduled basis (future cron job support)
* Logic for isPinned flag:
* Mark/highlight pinned transactions
  + Communicate with the MongoDB database.
  + Serve static content and rendered templates.

**3.3 Database (MongoDB Atlas)**

* **Collections**:
  + Users: Stores user credentials (hashed passwords using bcrypt), account information, and profile data.
  + Income: Stores income records linked to users, including source, amount, date, icon and recurring: Boolean, recurringType: String, isPinned: Boolean
  + Expense: Stores expense records linked to users, including category, amount, date, icon, optional notes and recurring: Boolean, recurringType: String, isPinned: Boolean

**3.4 Authentication & Security**

* **JWT**: Used for session management.
* **Bcrypt**: Password hashing and secure storage.
* **HTTPS**: Ensures secure data transmission.
* **Input Validation**: Prevents injection attacks.

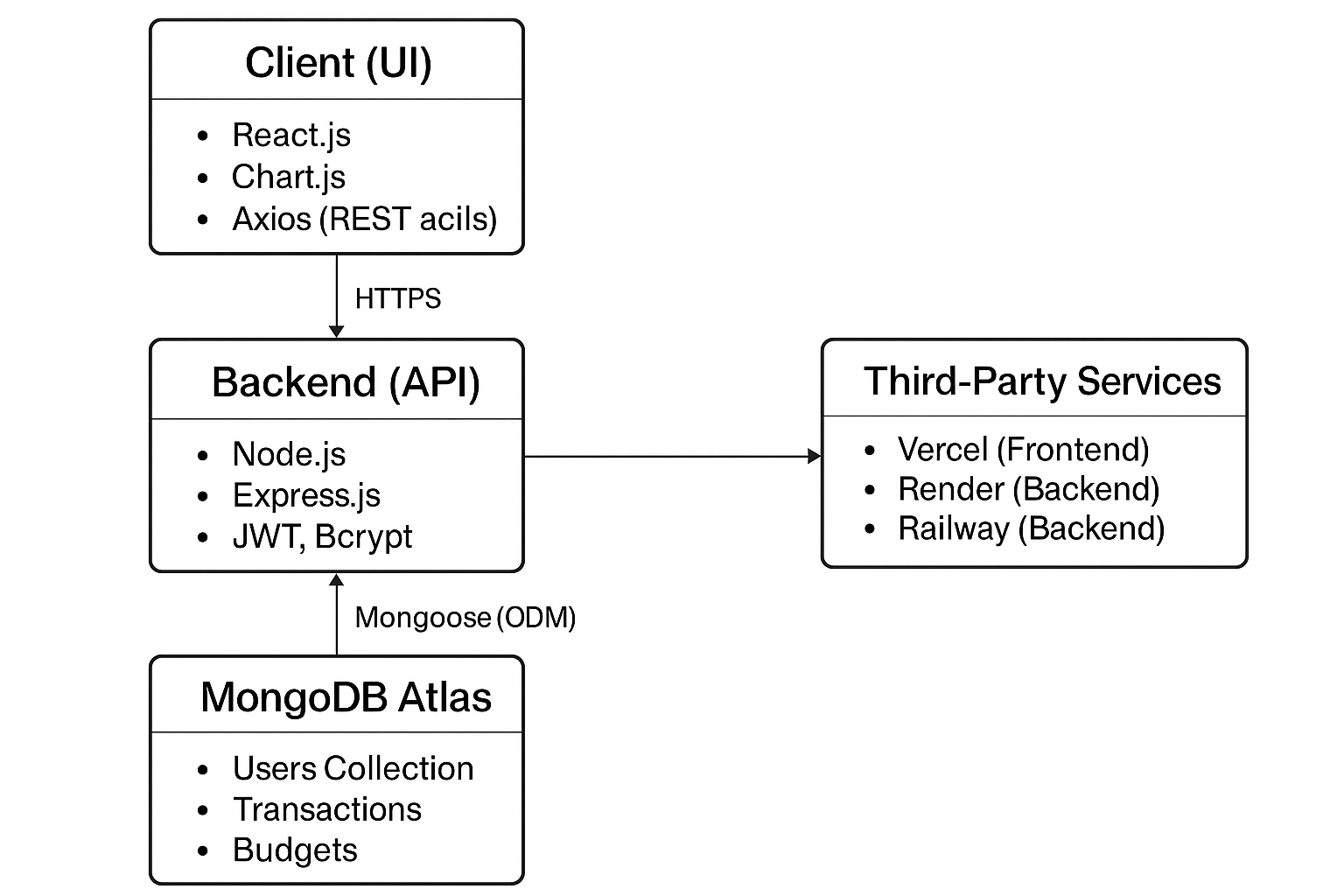
**4. Data Flow Description**

1. **User Interaction**: User accesses the web app via a browser.
2. **API Requests**: User actions (e.g., adding a transaction) trigger REST API calls.
3. **Server Processing**: Express.js routes handle the request, apply logic, and interact with the database.
4. **Database Transactions**: MongoDB stores or retrieves data.
5. **Response**: The server responds with success/failure and data payload, updating the frontend accordingly.
6. **Visualization**: Charts and budgets are rendered using Chart.js on the client side.

**5. Deployment Architecture**

* **Frontend Deployment**: Hosted on platforms like Vercel or Netlify.
* **Backend Deployment**: Hosted using Render, AWS, or Heroku.
* **Database Hosting**: MongoDB Atlas with secure connection string and access controls.

**6. Diagram: System Architecture**



**7. Non-Functional Requirements**

* **Performance**: API response within 500ms.
* **Scalability**: Modular design for microservices integration in the future.
* **Security**: Data encryption, validation, and secure authentication.
* **Reliability**: MongoDB replica set ensures data availability.
* **Responsiveness**: Works across devices and browsers.

**8. Future Enhancements**

* Admin dashboard for user monitoring.
* Export to PDF/CSV.
* Multi-currency support.
* Recurring transactions.
* Integration with payment APIs.

**9. Conclusion**

This architecture ensures the system is secure, scalable, and user-friendly. It supports real-time interaction with robust backend services and a clean UI layer, optimized for performance and maintainability.