Approach to Building the Personal Finance Assistant

Problem Statement

The Personal Finance Assistant is a full-stack web application designed to help users track, manage, and understand their financial activities. The core requirements include creating and listing income/expense transactions, visualizing spending habits through graphs, and extracting expenses from uploaded receipts (images/PDFs). Bonus features include supporting PDF transaction history uploads, pagination for transaction listings, and multi-user functionality. The application must use a suitable data model, separate frontend and backend APIs, persist data in a database, and provide comprehensive documentation, adhering to high code quality standards.

Approach

1. Requirement Analysis

The project requirements were broken down into core and bonus features:

- **Core Features**:
  - Create income/expense entries via a web app.
  - List transactions within a time range.
  - Display graphs (e.g., expenses by category or date).
  - Extract expenses from uploaded receipts (images/PDFs).
- **Bonus Features**:
  - Upload and parse tabular PDF transaction histories.
  - Support pagination for transaction listings.
  - Enable multiple users with authentication.
- **Technical Constraints**:
  - Separate frontend and backend with API communication.
  - Persist data in a database.
  - Follow code quality guidelines (clean code, modularity, error handling, documentation).

2. Technology Stack Selection

To meet the requirements efficiently, the following technologies were chosen:

- **Frontend**: React for a dynamic, component-based UI; React Router for navigation; Chart.js for data visualization; Axios for API calls.
- **Backend**: Node.js with Express for a lightweight, scalable API server; Mongoose for MongoDB interaction; JWT for authentication; Multer for file uploads; Swagger for API documentation.
- **Database**: MongoDB for its flexibility with JSON-like documents and scalability.
- **File Handling**: Multer for handling file uploads, with assumed external libraries (e.g., Tesseract.js or pdf.js) for receipt/PDF parsing (not shown in provided files).

3. Data Model Design

The database schema was designed to support the required functionality and relationships:

- **User**: Stores user details (`username`, `email`, `passwordHash`, `createdAt`) for multi-user support.
- **Transaction**: Represents income/expense entries (`title`, `amount`, `date`, `category`, `description`, `type`, `userId`, `createdAt`) with a reference to the user.
- **Receipt**: Stores uploaded receipt data (`transactionId`, `filePath`, `extractedText`, `uploadedAt`) linked to transactions.
- **Relationships**:
  - One `User` to many `Transactions` (via `userId`).
  - One `Transaction` to many `Receipts` (via `transactionId`).

4. Backend Implementation

The backend was structured as a RESTful API with modular components:

- **Models** (`User.js`, `Transaction.js`, `Receipt.js`): Defined Mongoose schemas for data persistence and relationships.
- **Controllers** (`userController.js`, `transactionController.js`): Handled business logic for user authentication and transaction management.
- **Routes** (`userRouter.js`, `transactionRoutes.js`): Defined API endpoints (e.g., `/users/register`, `/transactions/create`).
- **Middleware**:
  - `authMiddleWare.js`: Implemented JWT authentication to secure routes.
  - `uploadMiddleWare.js`: Used Multer to handle file uploads for receipts and PDFs.
- **API Documentation** (`swagger.yaml`, `swaggerConfig.js`): Used OpenAPI 3.0 to document all endpoints, schemas, and security schemes, served via Swagger UI at `/api-docs`.
- **Main Server** (`index.js`): Set up Express, MongoDB connection, CORS, and Swagger for API documentation.
- **API Endpoints**:
  - User: Register (`POST /users/register`), Login (`POST /users/login`), Get Transactions (`GET /users/:id/transactions`).
  - Transaction: Create (`POST /transactions/create`), List (`GET /transactions`), Delete (`DELETE /transactions/:id`), Summary (`GET /transactions/summary`), Receipt Upload (`POST /transactions/receipt-upload`), PDF Upload (`POST /transactions/pdf-upload`).

5. Frontend Implementation

The frontend was built as a single-page application (SPA) using React:

- **Components**:
  - `Navbar.jsx`: Provided navigation with authentication-aware links (login/register or logout).

    - `TransactionForm.jsx`: Form for creating transactions with validation (e.g., positive amount).
    - `TransactionList.jsx`: Table for listing transactions with filtering (date range, type) and pagination.
    - `SummaryChart.jsx`: Pie and Bar charts for income vs. expense visualization using Chart.js.
    - `FileUpload.jsx`: File upload interface for receipts and PDFs, dynamically selecting the appropriate API.
    - `Home.jsx`, `Dashboard.jsx`, `Transactions.jsx`, `Upload.jsx`, `Login.jsx`, `Register.jsx`: Pages for different functionalities.
- **Routing** (`App.js`): Used React Router with private routes to protect authenticated pages.
- **API Integration** (`api.js`): Axios-based API client with token-based authentication for secure API calls.

6. Key Features Implementation

- **Transaction Creation**: `TransactionForm.jsx` sends data to `POST /transactions/create`, validated and stored in MongoDB.
- **Transaction Listing with Pagination**: `TransactionList.jsx` fetches data from `GET /transactions` with query parameters (`start`, `end`, `type`, `page`, `limit`) and implements client-side pagination.
- **Data Visualization**: `SummaryChart.jsx` fetches summary data from `GET /transactions/summary` and renders charts using Chart.js.
- **Receipt/PDF Upload**: `FileUpload.jsx` uploads files to `POST /transactions/receipt-upload` or `POST /transactions/pdf-upload`, with results displayed in `Upload.jsx`.
- **Multi-User Support**: `userController.js` and `authMiddleWare.js` handle user registration, login, and route protection using JWT.
- **API Documentation**: `swagger.yaml` and `swaggerConfig.js` provide a comprehensive OpenAPI specification, accessible via Swagger UI.

7. Code Quality Adherence

- **Clean Code**: Used meaningful names (e.g., `createTransaction`, `formData`) and consistent formatting.
- **Modularity**: Separated concerns into controllers, models, middleware, and components.
- **Error Handling**: Implemented try-catch in controllers and error states in React components.
- **Documentation**: Integrated Swagger for API documentation (`swagger.yaml`, `swaggerConfig.js`) and provided a README for setup and usage instructions.
- **Scalability**: Used MongoDB for flexible data storage and Express for a lightweight API server.

8. Challenges and Solutions

- **Receipt Extraction**: Assumed a service (`transactionService.js`) for parsing images/PDFs, integrated with Multer for file handling.

- **Pagination**: Implemented server-side pagination in `transactionController.js` and client-side controls in `TransactionList.jsx`.
- **Authentication**: Used JWT to secure routes and associate transactions with users.
- **Visualization**: Chose Chart.js for responsive, easy-to-implement charts, addressing the graphing requirement.
- **API Documentation**: Used Swagger to document all endpoints, ensuring developer-friendly access to API details.

Conclusion

The Personal Finance Assistant was built as a robust, user-friendly application meeting all core and bonus requirements. The modular design, secure authentication, intuitive UI, and comprehensive Swagger documentation ensure scalability and maintainability. The project demonstrates proficiency in full-stack development, API design, data visualization, and API documentation, with opportunities for further enhancement in testing and advanced features.