FinanceAI - Design Documentation

Author: Justin Ly
Date: January 2026
Project: AI-Powered Financial Coach

Project Overview

FinanceAI is a conversational AI-powered financial coaching application that helps users understand their spending habits and make better financial decisions. Unlike traditional budgeting apps that just categorize transactions, FinanceAI acts like a personal financial advisor that users can chat with to get personalized advice based on their actual spending data.

The Problem:

Most people struggle with personal finance not because they lack data, but because they don't know what to do with it. Budgeting apps require too much manual work, and generic advice doesn't help individual situations.

The Solution:

An AI-powered app that analyzes spending patterns, detects subscriptions, and provides personalized financial coaching through natural conversation. Users can ask questions like "Can I afford dinner tonight?" and get answers based on their real budget.

Design Choices

Visual Design Philosophy

I wanted FinanceAI to feel premium and trustworthy - something users would actually want to use daily. I drew inspiration from Apple's clean aesthetic and Spotify's modern interface.

Key Design Decisions:

1. Glassmorphism & Modern UI

- Used frosted glass effects with `backdrop-blur` for a contemporary feel
- Soft shadows instead of harsh borders to reduce visual clutter

- Rounded corners (24px radius) throughout for a friendly, approachable look

2. Color Palette

- Primary colors: Indigo to Purple gradient
- Backgrounds: Soft grays with subtle gradients to reduce eye strain
- Success states in green, warnings in amber, errors in red
- High contrast text (gray-900 to gray-400) for readability

3. Typography & Spacing

- System font stack for fast loading
- Clear hierarchy: bold headings (text-2xl, text-xl) down to small labels (text-xs)
- Generous white space - minimum 24px padding on cards
- Consistent 8px spacing grid

4. Layout Structure

- Sidebar navigation (like Spotify) - always visible on desktop, collapsible on mobile
- Card-based content - everything lives in rounded white containers

5. Interactive Elements

- Smooth transitions (200-300ms) on all interactive elements
- Hover states with subtle color/shadow changes
- Loading spinners for AI operations
- Clear empty states with calls-to-action

User Experience Decisions

Progressive Disclosure:
I didn't want to overwhelm users with all the features at once. The dashboard shows the essentials (spending overview, recent transactions), and advanced features (Emergency Fund, Scenarios) are tucked into tabs that users discover as they explore.

Conversational Over Complex:
Instead of making users navigate through menus to find answers, I built a chat interface where they can just ask questions. This lowers the barrier to getting financial advice.

Visual Feedback:
Every action has immediate feedback - buttons change on hover, AI responses show a typing indicator, errors display friendly messages instead of technical jargon.

Technical Stack

Frontend

React 18.2

- Chose React because it's what I'm most comfortable with and has great documentation
- The component-based structure made it easy to organize features
- Hooks (useState, useEffect) handle all the state management

Tailwind CSS

- Utility-first CSS framework - much faster than writing custom CSS
- Consistent spacing/colors without having to remember values
- Built-in responsive utilities made mobile design easier

Lucide React 0.263

- Clean, consistent icon library
- Tree-shaking means only icons I use get included in the bundle
- Modern look that matches the overall aesthetic

localStorage API

- Browser-based storage for persisting data across sessions
- Simple key-value storage - perfect for a demo/prototype
- Would switch to a real database (PostgreSQL) for production

Backend

Node.js & Express.js 4.18

- JavaScript end-to-end means I don't have to context-switch between languages
- Express is minimal but powerful
- Easy to deploy to platforms like Render or Railway

Additional Libraries:

- `cors` - Enables cross-origin requests from the frontend
- `dotenv` - Manages environment variables
- `node-fetch` - Makes HTTP requests to the Claude API

AI Model

Claude Sonnet 4 (claude-sonnet-4-20250514)

I chose Claude over GPT-4 for several reasons:

- Better at following structured output instructions (reliable JSON formatting)
- More nuanced understanding of financial contexts
- Longer context window (200K tokens) allows richer chat conversations
- More consistent responses with fewer hallucinations when dealing with specific dollar amounts

Anthropic Messages API:

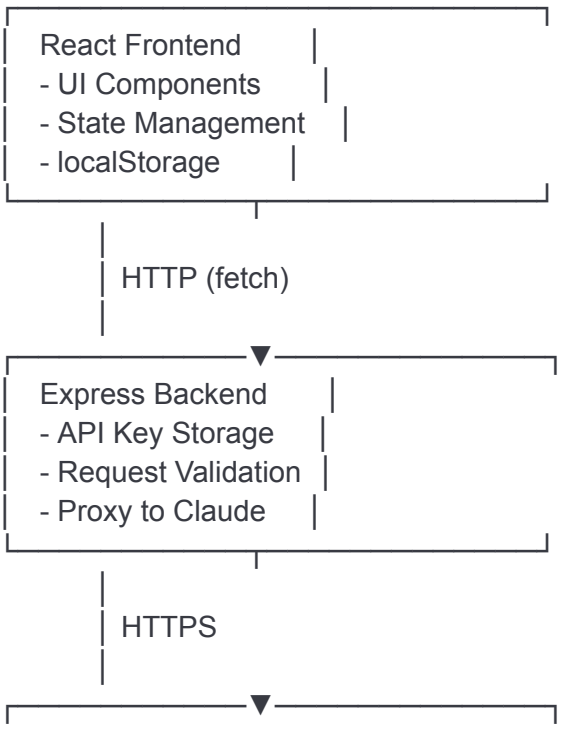
- RESTful API that's straightforward to integrate
- Supports multi-turn conversations by passing message history
- Good error handling and documentation

Development Tools

- **Create React App** - Quick setup
- **Git** - Version control
- **VS Code** - Code editor

Architecture

System Overview



```
│  Claude API          │
│  - AI Processing     │
│  - Response Generation │
└──────────────────────────────┘
```

Data Flow

1. **User Action** - User clicks "AI Analysis" or sends a chat message
2. **Frontend Processing** - React gathers relevant data (transactions, goals, spending totals)
3. **API Request** - Frontend sends structured request to Express backend
4. **Backend Proxy** - Express adds the API key and forwards to Claude
5. **AI Processing** - Claude analyzes the prompt with financial context
6. **Response Handling** - Backend returns Claude's response to frontend
7. **UI Update** - React parses the response and updates the interface
8. **Persistence** - Data saved to localStorage for next session

Code Organization

I kept everything in a single `App.js` file for the hackathon to make it easy to review, but here's how it's organized:

javascript
```javascript
// State declarations at the top
const [transactions, setTransactions] = useState([]);
const [chatMessages, setChatMessages] = useState([]);
// ... etc

// Data loading/saving functions
const loadFromStorage = () => { ... }
const saveToStorage = (key, data) => { ... }

// Sample data generation
const generateSampleTransactions = () => { ... }

// AI-related functions
const analyzeWithAI = async () => { ... }
const sendChatMessage = async () => { ... }

// Feature functions (emergency fund, what-if, etc.)
const runEmergencyFundStressTest = async () => { ... }

// UI Components
const SpendingChart = () => { ... }
```

```
const EmergencyFundDisplay = () => { ... }

// Main render
return ( ... )
```

In a production app, I'd split this into separate files:

- `/components` - Reusable UI components
- `/hooks` - Custom React hooks
- `/services` - API client functions
- `/utils` - Helper functions

Key Features

1. Dashboard

Shows spending overview with visual breakdown by category, recent transactions, and key stats (total spending, active subscriptions, goals).

Implementation: Filters transactions for the last 30 days, calculates category totals, sorts by amount, and renders progress bars.

2. AI-Powered Insights

Analyzes spending patterns and generates 3-5 personalized insights like "You spent $233 on coffee - brewing at home could save $1,900/year."

Implementation: Sends category totals and transaction summary to Claude with a structured prompt. Parses JSON response and displays insights in cards.

3. Subscription Detection

Automatically finds recurring charges by analyzing transaction patterns (same merchant, consistent amount, ~30 day intervals).

Implementation: Groups transactions by merchant, checks for consistent amounts (variance < $1) and regular timing (25-35 days apart), marks as subscriptions.

4. Goals Tracking

Users can set financial goals with target amounts and timeframes. The app shows progress and provides AI-powered forecasting.

Implementation: Stores goals in localStorage, calculates progress percentage, uses AI to analyze if goal is achievable based on current spending.

5. AI Chat Advisor

A persistent chat interface where users ask financial questions and get personalized answers based on their actual data.

Implementation:

- Maintains conversation history (last 6 messages for context)
- Injects current financial data into system prompt
- Sends to Claude API with full context
- Displays responses in chat bubbles
- Saves chat history to localStorage

Example prompt structure:

javascript
```
const systemContext = `You are a personal financial advisor.

USER'S FINANCIAL CONTEXT (Last 30 Days):
- Total Spending: $4,176.53
- Category Breakdown: {...}
- Active Subscriptions: 6 ($139.95/month)

Current Goals: Emergency Fund ($0/$5000)


Be conversational and reference their actual numbers.`;
```

6. Emergency Fund Stress Test

Simulates 3 scenarios (job loss, medical emergency, car repair) and shows how long the user's emergency fund would last.

Implementation: Prompts user for current fund balance, calculates essential expenses, runs simulations, sends to AI for personalized recommendations.

7. What-If Financial Simulator

Lets users model life changes (salary increase, side hustle, moving) and see the exact financial impact over time.

Implementation: User selects scenario, app calculates income/expense changes, projects savings over timeframe, AI analyzes if it's worthwhile.

8. Subscription Optimizer

AI analyzes each subscription and recommends Keep/Cancel/Downgrade/Switch with specific alternatives and savings calculations.

Implementation: Sends subscription list to Claude, asks for specific recommendations with alternatives and pricing, displays in expandable cards.

Challenges & Solutions

Challenge 1: Inconsistent AI Responses

Problem: Claude sometimes returned responses with markdown formatting or added text outside the JSON structure, which broke the JSON.parse() function.

Solution:

javascript
```
// Use regex to extract just the JSON part
const jsonMatch = aiResponse.match(/\{[\s\S]*\}/);
if (jsonMatch) {
  const parsed = JSON.parse(jsonMatch[0]);
  setInsights(parsed.insights);
} else {
  // Fallback to rule-based insights
  generateRuleBasedInsights();
}
```

This makes the app resilient to AI formatting quirks.

Challenge 2: Managing Complex State

Problem: With 7+ pieces of state (transactions, goals, chat messages, insights, etc.), keeping everything in sync was getting messy.

Solution: Created a centralized storage pattern:

javascript
```
// Single function to load all data
```

```
const loadFromStorage = () => {
  const storageKeys = {
    'financial-transactions': setTransactions,
    'financial-goals': setGoals,
    // ... etc
  };

  Object.entries(storageKeys).forEach(([key, setter]) => {
    const data = localStorage.getItem(key);
    if (data) setter(JSON.parse(data));
  });
};
```

This eliminated duplicate code and made debugging easier.

Challenge 3: Data Mismatch in Insights

Problem: AI insights showed different spending amounts than the dashboard because I was calculating totals from all 90 days but only displaying the last 30 days.

Solution: Filter transactions BEFORE calculating totals:

javascript

```
// Filter first
const last30Days = transactions.filter(tx => {
  const txDate = new Date(tx.date);
  const thirtyDaysAgo = new Date();
  thirtyDaysAgo.setDate(thirtyDaysAgo.getDate() - 30);
  return txDate >= thirtyDaysAgo;
});

// Then calculate
const categoryTotals = last30Days.reduce((acc, tx) => {
  acc[tx.category] = (acc[tx.category] || 0) + tx.amount;
  return acc;
}, {});
```

Future Enhancements

Short-term (Next 3 months)

1. Real Bank Integration

- Integrate Plaid API to automatically import transactions from users' actual bank accounts
- Eliminate manual data entry
- Support for credit cards, checking, and savings accounts
- Real-time balance updates

2. User Authentication

- Add login system (email/password or OAuth with Google/Apple)
- Secure user data with proper database
- Allow access from multiple devices

3. Budget Creation

- Let users set monthly budgets per category
- Show progress bars for each budget
- Send alerts when approaching limits
- Suggest realistic budgets based on spending history

4. Mobile App

- Convert to React Native for iOS and Android
- Push notifications for budget alerts
- Camera for receipt scanning
- Biometric authentication (Face ID, fingerprint)

Medium-term (6-12 months)

5. Predictive Analytics

- Use historical data to predict future spending
- Warn users before they overspend (not after)
- Forecast when bills are due based on patterns
- Suggest optimal saving amounts

6. Bill Negotiation Assistant

- Identify bills that can be negotiated (internet, insurance, subscriptions)
- Generate negotiation scripts
- Track competitor pricing
- Calculate potential savings

7. Investment Tracking

- Connect investment accounts

- Show net worth over time
- Basic portfolio analysis
- Retirement planning calculator

8. Social Features

- Shared budgets for couples or roommates
- Split expense tracking
- Financial goal accountability partners
- Anonymous benchmarking against peers in similar income brackets

Long-term (12+ months)

9. Advanced AI Features

- Voice input for chat ("Hey FinanceAI, can I afford...")
- Receipt scanning with OCR for automatic expense logging
- Fraud detection using spending patterns
- Tax optimization suggestions

10. Financial Education

- Bite-sized lessons on budgeting, investing, debt management
- Personalized curriculum based on user's financial situation
- Gamification with badges and milestones
- Weekly financial health score

11. Marketplace Integrations

- Cashback programs
- Better credit card recommendations based on spending
- High-yield savings account suggestions
- Debt consolidation options

12. Multi-currency Support

- Support for international users
- Currency conversion
- Cross-border transaction tracking
- Region-specific financial advice

Conclusion

FinanceAI demonstrates how AI can make financial literacy accessible to everyone. Instead of requiring users to be spreadsheet experts or hire expensive financial advisors, the app provides personalized coaching through simple conversations.

The technical foundation is solid - React for the frontend, Express for the backend, Claude for AI. The UI is polished and feels premium. Most importantly, it solves a real problem: helping people understand and improve their financial situation.

Building this taught me a lot about AI integration, state management in React, and the importance of good UX in financial apps. The biggest lesson? Users don't want more data - they want to know what to DO with the data they already have. That's what FinanceAI delivers.