**Comprehensive Project Development Prompt: Family Budget Management Application**

**Project Title:** Family Budget Management Web Application - Continued Development & Frontend Implementation

**I. Project Overview & Goal:**

You are tasked with continuing the development of a comprehensive Family Budget Management web application. The primary goal is to build a feature-rich, user-friendly, and robust platform that empowers users to manage their finances effectively. This includes tracking income/expenses, handling complex transactions (recurring, installments), managing detailed budgets across multiple profiles, and gaining financial insights.

**The application will be bilingual (Hebrew - RTL primary, English - LTR secondary).**

A significant portion of the backend API infrastructure is already in place. This phase will heavily focus on **completing and refining the backend APIs as needed, and extensively developing the React-based frontend** to bring all specified functionalities to life.

**II. Core Reference Document:**

* **"Comprehensive System Specification Document - Family Budget Management Application (Version 1.0)"**: This document (to be provided separately by the project owner) is your **primary source of truth**. It details all functional requirements, UX/UI principles, data models (Entities), existing and planned API endpoints, technology stack, and future roadmap items. **Adherence to this document is crucial.**

**III. Current Project Status (Key Highlights):**

* **Backend (Node.js, Express, TypeScript, TypeORM, PostgreSQL):**
  + **Entities:** All core entities (User, Category (with userId, name\_he, name\_en), Subcategory (with name\_he, name\_en), Income, Expense (with expenseType, parentId, isProcessed), UserSettings, RecurringExpenseDefinition, InstallmentTransaction, BudgetProfile, Budget) are defined.
  + **Database:** PostgreSQL schema is up-to-date with all entities via TypeORM migrations. Soft delete (deletedAt) is implemented for most data entities.
  + **Authentication:** JWT-based authentication (/api/auth/signup, /api/auth/login, /api/auth/profile) is functional. auth.middleware.ts protects routes.
  + **Core APIs Implemented (CRUD or near-CRUD for instances):**
    - Categories: GET /api/categories (supports type filtering, returns global & user-specific).
    - User Settings: GET/PUT /api/user-settings.
    - Incomes: Full CRUD for single income instances.
    - Expenses (Unified POST): POST /api/expenses handles creation of single expenses, and also the creation of RecurringExpenseDefinition + its instances, and InstallmentTransaction + its instances. Other CRUD operations for individual Expense instances are also in place.
    - Budget Profiles: Full CRUD via POST/GET/PUT/DELETE /api/budget-profiles.
    - Budget Allocations: Upsert (POST), GET (with spending calculation), DELETE via /api/budgets.
    - Dashboard Data: GET /api/dashboard/summary, /api/dashboard/recent-transactions, /api/dashboard/expense-distribution are implemented.
  + **Data Seeding:** Scripts seed.ts (basic categories) and seed-advanced.ts (comprehensive dummy data) exist.
  + **Development Environment:** Dockerized with Docker Compose. Hot-reloading for backend development is functional.
* **Frontend (React, Vite, Tailwind CSS):**
  + **Project Setup:** Vite project initialized. Key dependencies (react-router-dom, axios, @tanstack/react-query, recharts, date-fns, react-icons, react-hook-form, tailwindcss) are installed.
  + **Core Structure:** Basic folder structure (pages, components, contexts, hooks, api) is in place.
  + **Styling:** Tailwind CSS is configured, including tailwind.config.js (with custom theme colors/fonts) and postcss.config.js. RTL support via dir="rtl" on <html> and Tailwind's RTL modifiers.
  + **Authentication Context (AuthContext.jsx):** Implemented for managing user state, token, login/logout functions, and JWT persistence in localStorage. Includes apiClient.js (Axios instance) with baseURL and JWT request interceptor.
  + **Routing (App.jsx):** Basic routing with react-router-dom for public (Login, Signup) and protected routes. ProtectedRoute.jsx and MainLayout.jsx (with Sidebar and Header components) are implemented.
  + **Initial Pages/Components:**
    - LoginPage.jsx, SignupPage.jsx: Functional forms connecting to auth APIs.
    - DashboardPage.jsx: Structure with widgets in place. Connected to dashboard APIs (useDashboardSummary, useRecentTransactions, useExpenseDistribution) and displays fetched data, including charts and period navigation.
    - AddTransactionModal.jsx: Implemented with react-hook-form for adding/editing single incomes/expenses, and for initiating recurring/installment expenses.
    - TransactionsListPage.jsx: Basic structure and data fetching implemented. Displays transactions and has stubs/basic implementation for edit/delete actions.

**IV. Key Development Tasks (Continuing from current state):**

**A. Backend Refinement & Completion (TypeScript, TypeORM, Express):**

1. **API for Recurring/Installment Definitions (Ensure Full CRUD and Logic):**
   * **RecurringExpenseDefinition (/api/recurring-definitions):**
     + Verify/Complete PUT /api/recurring-definitions/:id: Crucially, when a definition is updated (e.g., amount, frequency, dates), all future, unprocessed Expense instances linked to it **must be soft-deleted and regenerated** according to the new definition. Ensure nextDueDate and isActive are correctly updated.
     + Verify/Complete DELETE /api/recurring-definitions/:id: Ensure it soft-deletes the definition AND all its future, unprocessed Expense instances atomically (use transaction).
   * **InstallmentTransaction (/api/installment-transactions):**
     + Verify/Complete PUT /api/installment-transactions/:id: Allow updates to non-financial fields (description, paymentMethod, isCompleted). Changing core financial details (totalAmount, numberOfInstallments) post-creation should ideally involve deleting and recreating the plan.
     + Verify/Complete DELETE /api/installment-transactions/:id: Ensure it soft-deletes the transaction AND all its future, unprocessed Expense payment instances atomically.
2. **(If not fully robust) Income - Recurring Definitions:**
   * Implement RecurringIncomeDefinition entity (similar to expenses).
   * Implement API endpoints (/api/recurring-income-definitions) for CRUD operations, including generation of future Income instances.
3. **Refine Budget API (/api/budgets):**
   * Ensure GET endpoint robustly calculates spentAmount when year and month (and budgetProfileId) are provided.
   * Consider edge cases for POST (upsert) if multiple users might (in a future scenario) share a budget profile (though current model links profile to one user).
4. **Error Handling & Logging:** Review and enhance error handling for consistency. Ensure meaningful logs for debugging.
5. **Security Review:** Double-check input validations and authorization logic on all sensitive endpoints.
6. **(Optional Refactor) Service Layer:** Consider refactoring complex business logic from route handlers into dedicated service classes/modules for better organization and testability (e.g., ExpenseService, BudgetService).
7. **Scheduler/Cron Job (Initial Implementation - if not done):**
   * While instances are pre-generated, implement a basic daily job that:
     + Updates Expense.isProcessed to true for instances whose date has passed.
     + Updates RecurringExpenseDefinition.nextDueDate and InstallmentTransaction.isCompleted based on processed instances.
     + (Future: This job would also handle ongoing generation for "indefinite" recurring items).

**B. Frontend Implementation & Completion (React, Tailwind CSS, React Query, etc.):**

1. **Finalize Dashboard (DashboardPage.jsx - Ref: Spec Sec 4.6):**
   * Ensure all three widgets (Summary, Recent Transactions, Expense Distribution) are fully functional, robustly displaying data from their respective hooks, and handling loading/error/empty states gracefully.
   * Verify period selection (month/year navigation) correctly updates data for Summary and ExpenseDistribution widgets.
   * Ensure "Refresh All" button works as expected.
2. **Complete AddTransactionModal.jsx (Ref: Spec Sec 4.3, 4.4):**
   * **Full Edit Functionality:** When transactionToEdit prop is passed, the modal must correctly pre-fill *all* relevant fields for the given transaction type (single income/expense). For recurring\_instance or installment\_instance, editing should focus on fields of the *instance* (e.g., actual amount if different from planned, date, description, isProcessed status). Editing the parent *definition* is handled elsewhere.
   * **Submission Logic:** Ensure onSubmit correctly calls PUT /api/incomes/:id or PUT /api/expenses/:id when in edit mode, with the appropriate payload.
   * **Recurring/Installment Creation:** Verify data for new definitions (frequency, totalAmount, etc.) is correctly captured and sent via the unified POST /api/expenses.
   * Ensure form reset and modal close on successful save/update, and React Query invalidations trigger UI updates.
3. **Complete TransactionsListPage.jsx (Ref: Spec Sec 4.7):**
   * **Display:** Show all transactions (incomes and expenses, including planned expenses visually marked with isProcessed status and expenseType).
   * **Filtering:** Implement UI and logic for all defined filters (date range, type, isProcessed status, search term). Ensure React Query queryKey updates to refetch with server-side filters if API supports it, or apply client-side.
   * **Actions:**
     + **Edit:** Fully functional (opens AddTransactionModal pre-filled).
     + **Soft Delete:** Functional.
     + **Restore:** Implement restoreMutation and connect to UI.
   * Handle pagination if the list can become very long (requires backend API support).
4. **Implement UserSettingsPage.jsx (Ref: Spec Sec 4.9):**
   * Fetch and display current monthlyBudgetGoal and defaultCurrency.
   * Allow user to update monthlyBudgetGoal. Save via PUT /api/user-settings.
   * Ensure changes are reflected in the Dashboard's budget summary.
5. **Implement User Onboarding Wizard (Ref: Spec Sec 4.8):**
   * Conditional rendering after first login.
   * **Step 1 (Income):** Form for defining regular monthly income.
     + API: Call backend to create RecurringIncomeDefinition(s) or simplified income records.
   * **Step 2 (Fixed Expenses):** Form for defining fixed monthly expenses.
     + API: Call POST /api/expenses with expenseType:'recurring' to create RecurringExpenseDefinition(s).
   * **Step 3 (Variable Budgets):**
     + Calculate and display discretionary income.
     + List expense subcategories. Allow user to input budget amounts for the current month and active/default BudgetProfile.
     + API: Call POST /api/budgets for each allocation.
   * Ensure smooth navigation between steps and proper data saving.
6. **Implement Budget Management Screens (Ref: Spec Sec 4.5):**
   * **Budget Profiles Screen:**
     + List (GET /api/budget-profiles), Create (POST), Edit (PUT), Delete (DELETE), Activate (PUT) budget profiles.
   * **Monthly Allocation Screen (for a selected profile & month):**
     + UI for month/year selection.
     + Fetch and display budget allocations (GET /api/budgets) including allocatedAmount and spentAmount.
     + Allow editing allocatedAmount (calls POST /api/budgets - upsert).
     + Visual indicators for budget utilization per subcategory.
7. **Language & RTL:**
   * Ensure all new UI components and text are in Hebrew and correctly support RTL layout with Tailwind CSS.
   * Plan for how category names (fetched from API) will be displayed based on a future language toggle (for now, display name\_he).
8. **General UI/UX Refinements:**
   * Consistent styling, error handling, loading states, and empty states across all new screens and components.
   * Ensure responsiveness on all device sizes.

**V. Testing & Quality Assurance:**

* **Backend:** Implement unit and integration tests for services and API endpoints.
* **Frontend:** Implement component tests and potentially end-to-end tests for key user flows.
* Thorough manual testing of all functionalities on different devices and browsers.

**VI. Documentation:**

* Update the "Comprehensive System Specification Document" as new decisions are made or features evolve.
* Maintain clear JSDoc/TSDoc comments in the code.

This prompt provides a detailed continuation plan. The AI/Developer should prioritize tasks based on dependencies (e.g., frontend components often depend on backend APIs being ready). Regular communication and iterative development are encouraged.

**(הוסף את הסעיף הבא למסמך המאוחד, למשל אחרי סעיף "Technology Stack & Architecture" או כנספח)**

**X. Project Directory Structure**

This section outlines the recommended directory structure for both the backend and frontend applications. This structure promotes modularity, separation of concerns, and maintainability.

**X.1. Overall Project Root Structure (Illustrative)**

family-budget-app/

├── backend/ # Backend Node.js/Express application

├── frontend/ # Frontend React/Vite application

├── .git/ # Git repository data

├── .gitignore # Specifies intentionally untracked files

├── docker-compose.yml # Docker Compose configuration for all services

└── README.md # Project overview and setup instructions

**X.2. Backend Application Structure (backend/)**

backend/

├── dist/ # Output directory for compiled JavaScript (from TypeScript)

├── node\_modules/ # NPM project dependencies

├── src/ # TypeScript source code

│ ├── @types/ # Custom type definitions (if any, e.g., for Express Request)

│ ├── api/ # (Future) If API versioning or grouping is needed beyond routes

│ ├── config/ # Application configuration files (e.g., constants, specialized configs)

│ ├── data-source.ts # TypeORM DataSource configuration and initialization

│ ├── entity/ # TypeORM Entities (Database table models)

│ │ ├── User.ts

│ │ ├── Category.ts

│ │ ├── Subcategory.ts

│ │ ├── Income.ts

│ │ ├── Expense.ts

│ │ ├── UserSettings.ts

│ │ ├── RecurringExpenseDefinition.ts

│ │ ├── InstallmentTransaction.ts

│ │ ├── BudgetProfile.ts

│ │ ├── Budget.ts

│ │ └── index.ts # Exports ALL\_ENTITIES array for DataSource

│ ├── middleware/ # Express middleware functions

│ │ └── auth.middleware.ts # JWT authentication and user attachment

│ │ └── errorHandler.middleware.ts # (Future) Centralized error handling

│ ├── migration/ # TypeORM migration files (auto-generated & potentially edited)

│ ├── routes/ # Express route definitions (Controllers layer)

│ │ ├── index.ts # Main router aggregating all feature routes

│ │ ├── auth.routes.ts

│ │ ├── category.routes.ts

│ │ ├── income.routes.ts

│ │ ├── expense.routes.ts

│ │ ├── userSettings.routes.ts

│ │ ├── budgetProfile.routes.ts

│ │ ├── budget.routes.ts

│ │ └── dashboard.routes.ts

│ ├── services/ # Business logic layer (to be refactored into)

│ │ ├── auth.service.ts

│ │ ├── transaction.service.ts (for income/expense logic)

│ │ ├── budget.service.ts (for budget profile & allocation logic)

│ │ └── user.service.ts

│ ├── utils/ # Utility functions and helpers

│ │ ├── responseBuilders.ts (Example: for formatting API responses)

│ │ └── date.utils.ts (Example: for common date operations)

│ ├── index.ts # Main application entry point (Express server setup, DB init)

│ ├── seed.ts # Seeder for basic/global categories

│ └── seed-advanced.ts # Seeder for comprehensive mock data across all entities

├── .env # Environment variables (DB credentials, JWT\_SECRET, ports) - NOT in Git

├── .eslintignore # ESLint ignore patterns

├── .eslintrc.js # ESLint configuration

├── .gitignore # Git ignore patterns for backend

├── .prettierrc.json # Prettier configuration

├── Dockerfile # Docker build instructions for the backend service

├── nodemon.json # (Optional) Nodemon configuration

├── package-lock.json

├── package.json # NPM project manifest, scripts, dependencies

└── tsconfig.json # TypeScript compiler options

IGNORE\_WHEN\_COPYING\_START

content\_copy download

Use code [with caution](https://support.google.com/legal/answer/13505487).

IGNORE\_WHEN\_COPYING\_END

**X.3. Frontend Application Structure (frontend/)**

frontend/

├── dist/ # Output directory for Vite build (static assets)

├── node\_modules/ # NPM project dependencies

├── public/ # Static assets (e.g., index.html, favicon, images)

│ ├── index.html

│ └── vite.svg # (Example, replace with actual assets)

├── src/ # React application source code (TypeScript or JavaScript)

│ ├── api/ # API communication layer

│ │ └── apiClient.js # Configured Axios instance with interceptors

│ ├── assets/ # Static assets used by components (images, svgs, fonts not from CDN)

│ ├── components/ # Reusable UI components

│ │ ├── auth/ # Authentication-specific components (e.g., LoginForm, SignupForm)

│ │ ├── common/ # Generic, widely used components (e.g., Button, InputField, Spinner)

│ │ ├── dashboard/ # Dashboard-specific widget components

│ │ │ ├── SummaryWidget.jsx

│ │ │ ├── RecentTransactionsWidget.jsx

│ │ │ └── ExpensesChartWidget.jsx

│ │ ├── layout/ # Layout components

│ │ │ ├── MainLayout.jsx

│ │ │ ├── Header.jsx

│ │ │ └── Sidebar.jsx

│ │ ├── transactions/ # Components related to transaction management

│ │ │ ├── AddTransactionModal.jsx

│ │ │ └── TransactionListItem.jsx

│ │ ├── budget/ # Components for budget management

│ │ └── ui/ # Low-level UI primitives (e.g., Modal, Card, ProgressBar)

│ ├── contexts/ # React Context API providers

│ │ └── AuthContext.jsx

│ │ └── SettingsContext.jsx # (Future for user UI preferences)

│ ├── hooks/ # Custom React Hooks

│ │ ├── useAuth.js # Shortcut for useContext(AuthContext)

│ │ └── useDashboardData.jsx # Hooks for fetching dashboard-specific data via React Query

│ ├── pages/ # Top-level route components (screens)

│ │ ├── LoginPage.jsx

│ │ ├── SignupPage.jsx

│ │ ├── DashboardPage.jsx

│ │ ├── TransactionsListPage.jsx

│ │ ├── UserSettingsPage.jsx

│ │ ├── BudgetSetupPage.jsx # (For Onboarding & Budget Management)

│ │ └── NotFoundPage.jsx # (404 Error Page)

│ ├── services/ # (Optional) Client-side business logic or complex data transformations

│ ├── styles/ # Global styles, Tailwind base imports

│ │ └── index.css # Main CSS file with @tailwind directives

│ ├── utils/ # Utility functions (formatting, validation helpers, etc.)

│ │ └── formatters.js # e.g., formatCurrency, formatDate

│ ├── App.jsx # Root application component, defines routes

│ └── main.jsx # Entry point of the React application, renders App

├── .env # Frontend environment variables (e.g., VITE\_BACKEND\_API\_URL) - NOT in Git

├── .eslintignore

├── .eslintrc.cjs # ESLint configuration (may be .js or .json)

├── .gitignore

├── .prettierrc.json

├── index.html # Main HTML file (moved to public/ by Vite default)

├──postcss.config.js # PostCSS configuration (for Tailwind CSS)

├── package-lock.json

├── package.json

├── tailwind.config.js # Tailwind CSS configuration

└── vite.config.js # Vite build tool configuration

**Key Considerations for Directory Structure:**

* **Backend services/ layer:** This is marked as "to be refactored into." Currently, much of the business logic might reside in the route handlers (routes/). Moving this to a dedicated service layer improves separation of concerns and testability as the application grows.
* **Frontend components/ organization:** The sub-directory structure within components/ (e.g., auth/, dashboard/, ui/) helps in organizing components by feature or type. common/ or shared/ is often used for truly generic components.
* **Naming Conventions:** Consistent naming for files and folders (e.g., PascalCase for React components, camelCase for utility files) is recommended.
* **Scalability:** This structure is designed to be scalable. As new features are added, new directories and files can be logically placed within this framework.
* **Flexibility:** This is a recommended structure. Developers can adapt it based on specific project needs or team preferences, but consistency is key.