InstaBudget Code Structure

This document explains how our codebase is organized. We tried to keep things organized by feature so it's easier to find different parts.

Architecture Overview

We're using Next.js with the App Router. The code is split into a few main parts:

- **Pages** (app/) - These are the actual pages/routes
- **Components** (components/) - Reusable UI components
- **Business Logic** (lib/) - Providers, utilities, data management
- **Backend Functions** (supabase-backend-edge-fn/) - Serverless functions
    - Note: The Supabase Edge Functions and database setup were implemented by us directly within the Supabase platform. The edge function files included in the code repository are provided for reference only, since the functions are deployed, managed, and executed through Supabase rather than run directly from the application code repo.

Complete File Structure

Here's the full file tree:

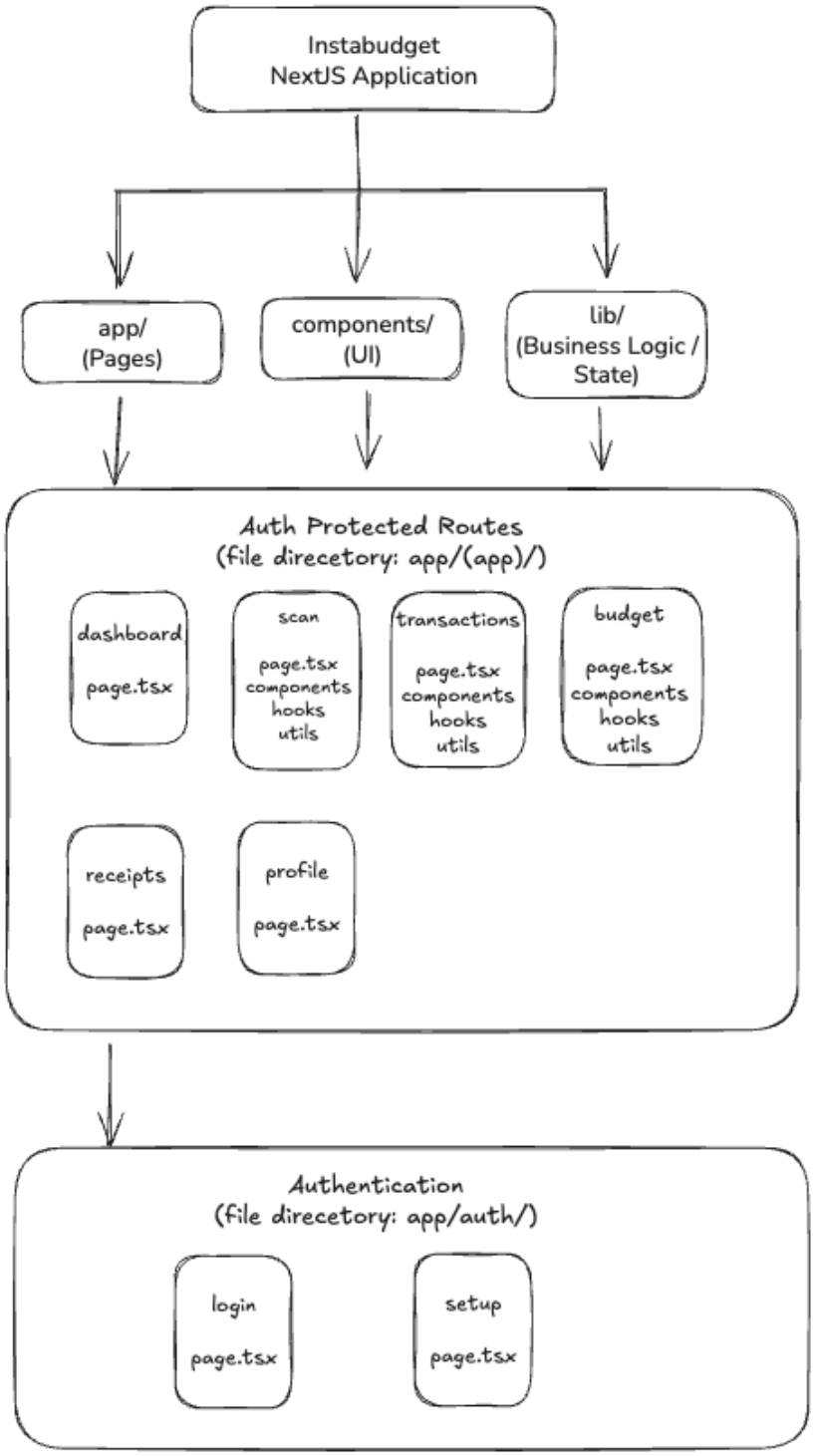
```
instab/
│
├── app/                          # Next.js App Router (file-based routing)
│   │
│   ├── layout.tsx                # Root layout with providers
│   ├── page.tsx                  # Home page (redirects to profile)
│   ├── globals.css               # Global styles
│   │
│   ├── (app)/                    # Protected routes (need to be logged in)
│   │   ├── layout.tsx            # App shell with navigation
│   │   │
│   │   ├── dashboard/            # Dashboard page
│   │   │   ├── page.tsx          # Main dashboard view
│   │   │   └── hooks/            # Dashboard-specific hooks
│   │   │
│   │   ├── scan/                 # Transaction input page
│   │   │   ├── page.tsx          # Main scan page with tabs
│   │   │   ├── types.ts          # TypeScript types
│   │   │   ├── constants.ts      # Constants and config
│   │   │   ├── utils.ts          # Utility functions
│   │   │   ├── components/       # Scan-specific components
│   │   │   │   ├── scan-header.tsx
│   │   │   │   ├── scan-tabs.tsx
│   │   │   │   ├── scan-tab-content.tsx  # Receipt scanning UI
│   │   │   │   ├── voice-tab-content.tsx # Voice recording UI
```

```
│   │   │   ├── manual-tab-content.tsx # Manual entry UI
│   │   │   ├── transaction-form.tsx   # Shared transaction form
│   │   │   ├── voice-recorder.tsx     # Voice recording controls
│   │   │   └── auto-save-success-card.tsx
│   │   └── hooks/                  # Scan-specific hooks
│   │       ├── use-edge-function.ts  # Calls Supabase edge functions
│   │       ├── use-voice-recording.ts # Voice recording logic
│   │       ├── use-scan-state.ts     # Scan page state management
│   │       └── use-transaction-save.ts # Transaction saving logic
│   │
│   ├── transactions/               # Transaction management page
│   │   ├── page.tsx                # Transaction list page
│   │   ├── types.ts                # Transaction types
│   │   ├── constants.ts            # Category labels, etc.
│   │   ├── utils.ts                # Date/currency formatting
│   │   ├── components/             # Transaction UI components
│   │   │   ├── transactions-list.tsx
│   │   │   ├── transaction-item.tsx
│   │   │   ├── transaction-edit-dialog.tsx
│   │   │   ├── transactions-header.tsx
│   │   │   ├── transactions-filters.tsx
│   │   │   └── transactions-stats.tsx
│   │   └── hooks/                  # Transaction hooks
│   │       ├── use-transaction-edit.ts
│   │       ├── use-transactions-filters.ts
│   │       └── use-transactions-stats.ts
│   │
│   ├── budget/                     # Budget configuration page
│   │   ├── page.tsx                # Budget setup page
│   │   ├── types.ts                # Budget types
│   │   ├── utils.ts                # Budget calculations
│   │   ├── components/             # Budget UI components
│   │   │   ├── budget-header.tsx
│   │   │   ├── budget-snapshot-card.tsx
│   │   │   ├── budget-save-button.tsx
│   │   │   └── loading-state.tsx
│   │   └── hooks/                  # Budget hooks
│   │       ├── use-budget-state.ts   # Budget state management
│   │       ├── use-budget-categories.ts # Category management
│   │       ├── use-budget-calculations.ts # Spending calculations
│   │       └── use-budget-save.ts     # Save budget to database
│   │
│   ├── receipts/                   # Receipt gallery page
│   │   ├── page.tsx                # Receipt list page
│   │   ├── types.ts                # Receipt types
│   │   ├── utils.ts                # Receipt utilities
│   │   ├── components/             # Receipt UI components
│   │   │   ├── receipts-grid.tsx
│   │   │   ├── receipt-card.tsx
│   │   │   ├── receipt-detail-dialog.tsx
│   │   │   ├── receipts-header.tsx
```

```
        │   │   │   │   ├── empty-receipts-state.tsx
        │   │   │   │   └── loading-state.tsx
        │   │   │   └── hooks/
        │   │   │       └── use-receipts.ts      # Receipt data fetching
        │   │   │
        │   │   └── profile/                  # User profile page
        │   │       ├── page.tsx              # Profile settings page
        │   │       ├── types.ts              # Profile types
        │   │       ├── components/           # Profile UI components
        │   │       │   ├── profile-header.tsx
        │   │       │   ├── profile-form.tsx
        │   │       │   └── account-actions.tsx
        │   │       └── hooks/
        │   │           └── use-profile-form.ts    # Profile form logic
        │   │
        │   └── auth/                         # Authentication routes
        │       ├── page.tsx                  # Login page
        │       └── setup/                    # First-time setup
        │           └── page.tsx              # Profile creation
        │
        ├── components/                       # Reusable React components
        │   ├── app-shell.tsx                 # Main app layout with navigation
        │   ├── theme-provider.tsx            # Dark/light theme provider
        │   └── ui/                           # shadcn/ui component library
        │       ├── button.tsx
        │       ├── card.tsx
        │       ├── input.tsx
        │       ├── dialog.tsx
        │       ├── budget_cycle_card.tsx         # Budget cycle configuration
        │       ├── budget_progress_card.tsx      # Budget progress visualization
        │       ├── category_limits_card.tsx      # Category limit management
        │       ├── transaction_history_card.tsx  # Transaction history display
        │       └── llm_insight_card.tsx          # AI insights display
        │
        ├── lib/                              # Core business logic & utilities
        │   ├── supabase.ts                   # Supabase client initialization
        │   ├── date-utils.ts                 # Date formatting (local timezone)
        │   ├── utils.ts                      # General utilities
        │   │
        │   ├── auth-provider.tsx             # Authentication context provider
        │   │                                 # - User session management
        │   │                                 # - Profile data
        │   │                                 # - Login/logout functions
        │   │
        │   └── user-data-provider.tsx        # User data context provider
        │       │                             # - Transactions
        │       │                             # - Budget categories
        │       │                             # - Receipts
        │       │                             # - AI insights
        │       ├── types.ts                  # Shared TypeScript types
        │       ├── hooks/                    # Data fetching hooks
```

```
        ├── use-transactions.ts       # Transaction CRUD operations
        ├── use-budget-categories.ts  # Budget category CRUD
        ├── use-receipts.ts           # Receipt fetching
        └── use-ai-insights.ts        # AI insight generation
    └── utils/
        └── calculate-category-spent.ts # Spending calculations

├── hooks/                            # Global custom hooks
    ├── use-auth-guard.ts             # Route protection
    ├── use-toast.ts                  # Toast notifications
    └── use-budget-limit-toasts.ts    # Budget limit alerts

├── supabase-backend-edge-fn/         # Supabase Edge Functions (Deno)
    ├── LLM_receipt.ts                # Receipt OCR & extraction
                                      # - Uses Google Gemini API
                                      # - Extracts transaction data
                                      # - Auto-categorizes spending

    ├── ai_audio_scan.ts              # Voice transaction processing
                                      # - Uses OpenAI Whisper (transcription)
                                      # - Uses Google Gemini (extraction)
                                      # - Creates transaction from audio

    ├── ai_insight.ts                 # AI spending insights
                                      # - Generates summaries
                                      # - Provides financial tips

    ├── budget-categories.ts          # Budget category operations
    └── upsert-profile.ts             # Profile creation/update

├── public/                           # Static assets
    └── *.svg                         # Icons and images

├── docs/                             # Documentation
    ├── ER Diagram/                   # Database schema diagrams
    ├── Tech Plan/                    # Technical planning docs
    └── UML Diagrams/                 # UML diagrams

└── Configuration Files
    ├── package.json                  # Dependencies & scripts
    ├── tsconfig.json                 # TypeScript configuration
    ├── next.config.ts                # Next.js configuration
    ├── tailwind.config.js            # Tailwind CSS config
    ├── components.json               # shadcn/ui configuration
    └── eslint.config.mjs             # ESLint configuration
```

Visual Code Structure



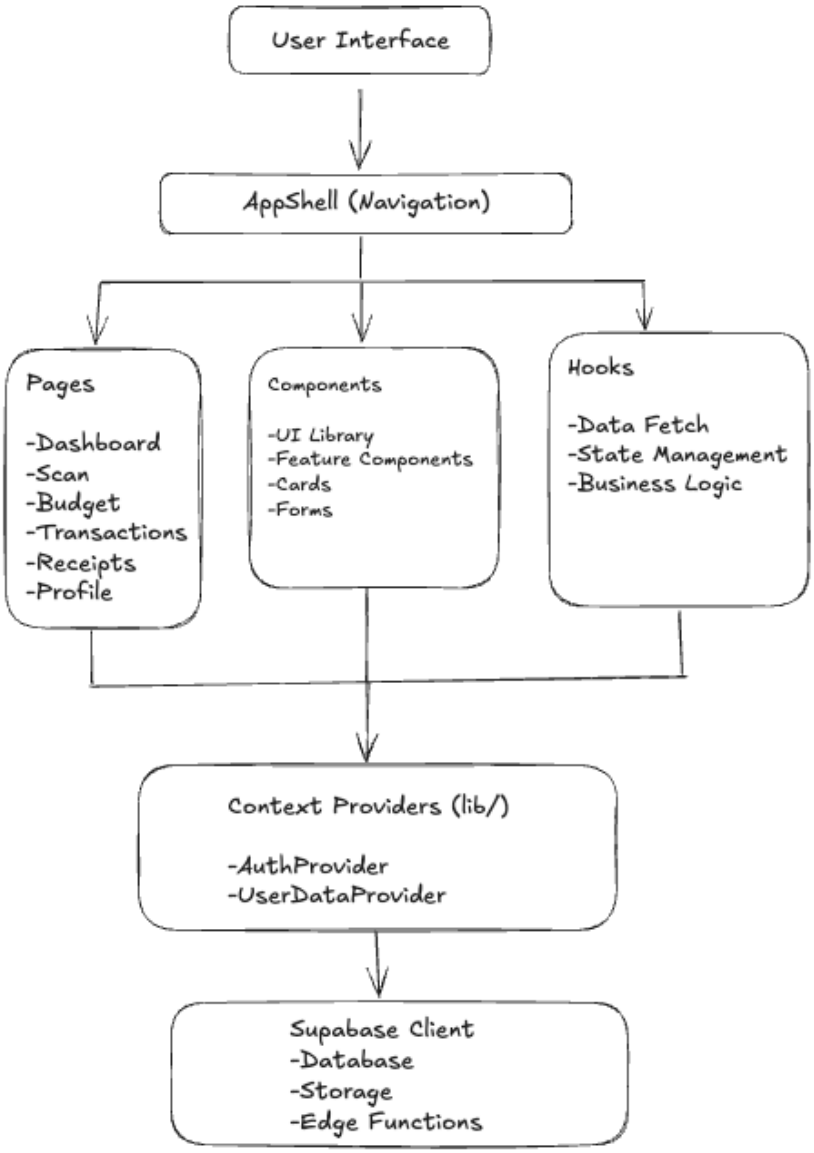
This diagram shows how the InstaBudget codebase is organized at a high level. The application is split into pages (routing), reusable UI components, and shared business logic. Protected routes are grouped under the main app folder, with each feature (dashboard, scan, budget, transactions,

etc.) organized in its own directory. Authentication pages are kept separate to clearly distinguish login/setup from the main application.

The top section shows the high-level folders of the application (pages, UI components, and shared logic). These feed into the protected routes below, where each feature page (dashboard, scan, transactions, budget, etc.) uses those shared components and logic to function.

Component Architecture

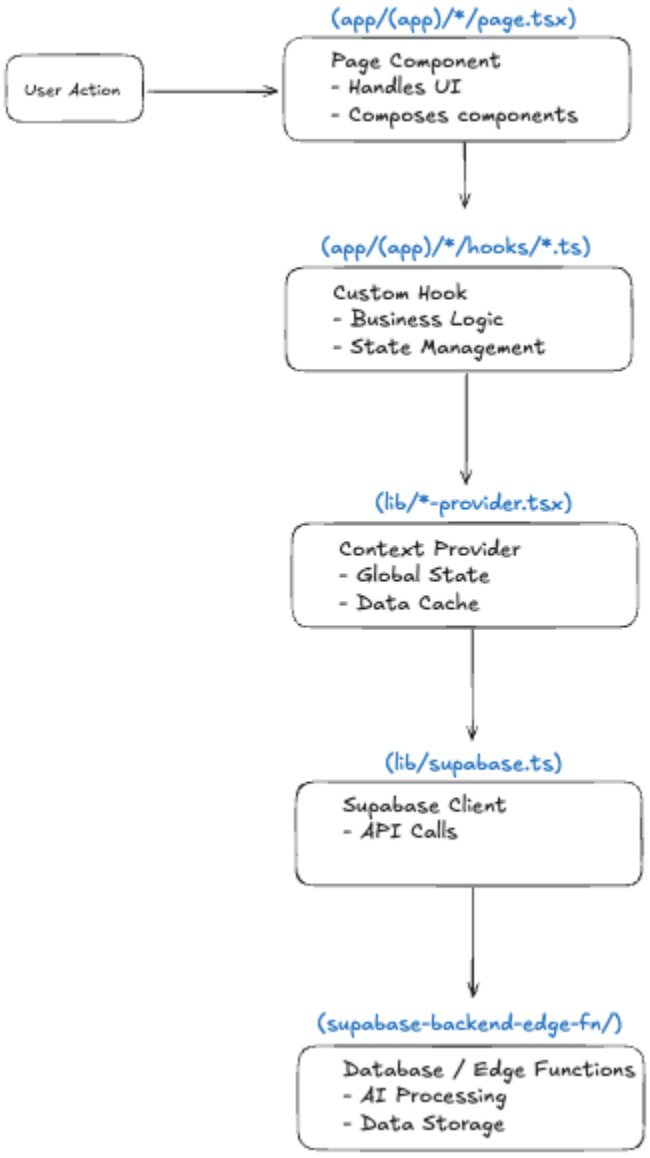
This shows how components are organized:



This diagram shows how the user interface is structured from top to bottom. Pages display the UI, components handle visual elements, hooks manage logic and data, and context providers connect everything to the Supabase backend.

Data Flow

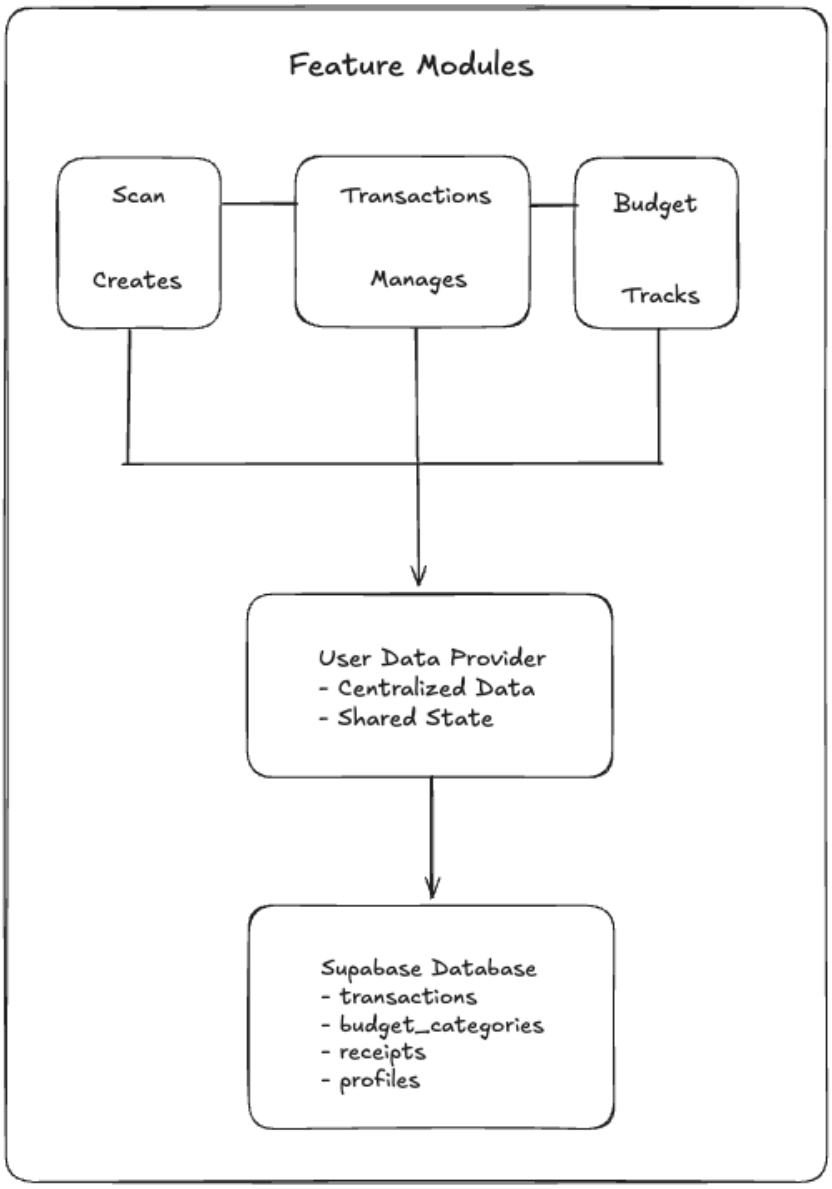
This is how data flows through the app when a user does something:



So essentially, user clicks something → page component handles it → hook does the logic → context provider manages state → Supabase client talks to database/edge functions.

Module Dependencies

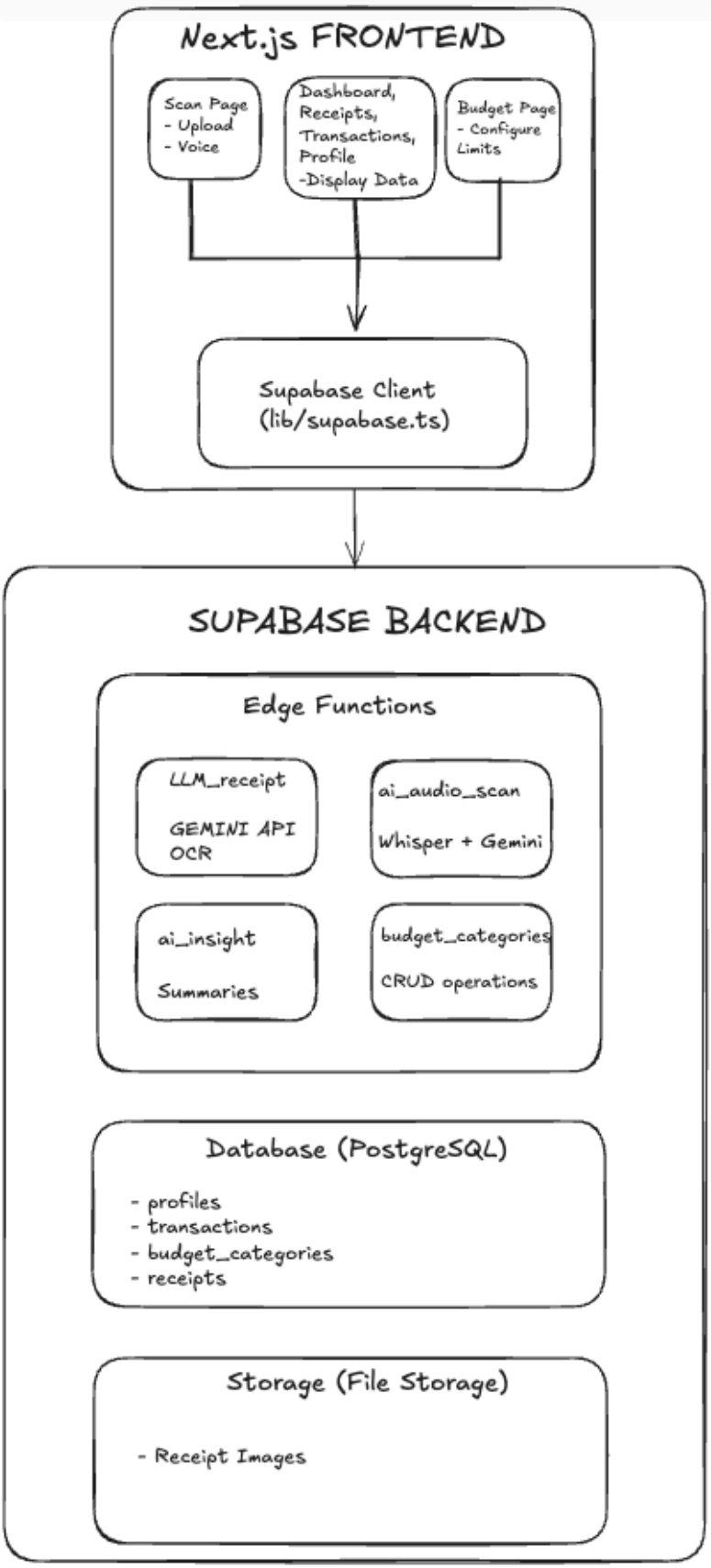
Here's how the different features connect:



All the features (scan, transactions, budget) share data through the UserDataProvider (this is a React context global state), which connects to the Supabase database.

Backend Integration

How the frontend connects to the backend:


Next.js FRONTEND
Scan Page
- Upload
- Voice
Dashboard,
Receipts,
Transactions,
Profile
-Display Data
Budget Page
- Configure
Limits
Supabase Client
(lib/supabase.ts)
SUPABASE BACKEND
Edge Functions
LLM_receipt
GEMINI API
OCR
ai_audio_scan
Whisper + Gemini
ai_insight
Summaries
budget_categories
CRUD operations
Database (PostgreSQL)
- profiles
- transactions
- budget_categories
- receipts
Storage (File Storage)
- Receipt Images

The frontend pages all use the Supabase client to talk to edge functions (for AI data and user data) and the database (for storing data).

How We Organized Things

1. Feature-Based Organization

We organized by feature so everything related to one thing is in the same place. Each feature folder has:

- page.tsx - The main page component
- components/ - UI components just for that feature
- hooks/ - Business logic hooks
- types.ts - TypeScript types
- utils.ts - Helper functions

This makes it easier to find things and understand what goes with what.

2. Separation of Concerns

We tried to keep things separated:

Pages (app/) - These handle routing and putting components together

- They don't have much logic, mostly just composition
- They use hooks for business logic
- They use components for UI

Components (components/) - Reusable UI pieces

- Focused on UI, not much state
- Get data through props

Business Logic (lib/) - Data management and utilities

- Context providers for global state (auth, user data)
- Custom hooks for fetching data
- Utility functions

Backend Functions (supabase-backend-edge-fn/) - Serverless functions

- Handle AI processing (receipt scanning, voice, insights)
- Database operations
- External API calls (Gemini, Whisper)

3. Data Flow

When a user does something, here's what happens:

1. User action triggers something in a page component
2. Page component calls a custom hook
3. Hook uses context provider to get/update data
4. Context provider uses Supabase client
5. Supabase client talks to database or edge functions

4. Component Hierarchy

The app shell wraps everything and provides navigation. Then each page has its own components, and we use shared UI components from the components/ui/ folder.

Design Patterns We Used

1. Feature Module Pattern

Each feature is self-contained in its own folder:

```
feature/
├── page.tsx          # Entry point
├── components/       # Feature UI
├── hooks/            # Feature logic
├── types.ts          # Feature types
└── utils.ts          # Feature utilities
```

2. Provider Pattern

We use React Context for global state:

```
lib/
├── auth-provider.tsx       # Authentication state
└── user-data-provider.tsx  # User data state
```

3. Hook Pattern

We put business logic in custom hooks so it's reusable:

```
hooks/
├── use-[feature]-[action].ts
└── Custom hooks for reusable logic
```

Important Files

Core Infrastructure

- **lib/auth-provider.tsx** - Manages user authentication and profile. We use React Context for this.

- **lib/user-data-provider.tsx** - Manages all user data (transactions, categories, receipts, insights). Also uses Context.
- **lib/date-utils.ts** - Date utilities. We had to make these because we were having timezone issues with dates showing the wrong day.
- **components/app-shell.tsx** - Main layout with navigation bar

Key Feature Modules

Scan Module (app/(app)/scan/)

- Handles three ways to add transactions: receipt scan, voice recording, manual entry
- Uses edge functions for AI processing
- Manages the whole transaction creation workflow

Budget Module (app/(app)/budget/)

- Lets users set up budget cycles
- Manage category limits
- Track spending progress

Transaction Module (app/(app)/transactions/)

- Shows list of all transactions
- Can filter and sort
- Can edit/delete transactions
- Shows statistics

Dashboard Module (app/(app)/dashboard/)

- Shows summary of transactions and budget category spending limits in one place
- Shows AI insights on how the user is currently spending

Receipts Module (app/(app)/receipts/)

- Shows stored photos of users' receipts

File Naming Conventions

We tried to be consistent with naming:

- **Pages**: page.tsx (Next.js convention)
- **Components**: kebab-case.tsx (like budget-header.tsx)
- **Hooks**: use-kebab-case.ts (like use-budget-state.ts)

Summary

Our codebase is organized by feature, which makes it easier to:

- Find code related to a specific feature
- Understand how things are connected
- Add new features without breaking existing ones
- Maintain and update code

The structure is:

1. **Pages** handle routing and composition
2. **Components** provide reusable UI
3. **Hooks** encapsulate business logic
4. **Providers** manage global state
5. **Edge Functions** handle server-side processing (AI data)
6. **Utilities** provide shared functionality

This keeps things organized and makes the codebase easier to work with.