Personal Budget Spreadsheet → App Conversion Guide

Overview

This document outlines how to convert the Personal Budget spreadsheet into a web/mobile application. The spreadsheet is designed with this migration in mind, using clean data structures and formulas that can be translated to code.

Spreadsheet Structure

Current Sheets & Their Purpose

1. **Dashboard** - Overview and KPIs (Main app home screen)
2. **Income** - Income sources (Input form)
3. **Expenses** - Budget by category (Input form)
4. **Savings & Goals** - Savings targets and progress (Goal tracking screen)
5. **Monthly Tracker** - Historical expense tracking (Transaction log + analytics)

Database Schema (for App)

Tables Needed

sql

```sql
-- Users
CREATE TABLE users (
  user_id UUID PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  name VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Income Sources
CREATE TABLE income_sources (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(user_id),
  source_name VARCHAR(100),
  amount DECIMAL(10,2),
  frequency VARCHAR(20), -- 'monthly', 'weekly', 'biweekly'
  is_active BOOLEAN DEFAULT true,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Expense Categories
CREATE TABLE expense_categories (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(user_id),
  category_name VARCHAR(100),
  parent_category VARCHAR(100), -- 'Housing', 'Transportation', etc.
  budgeted_amount DECIMAL(10,2),
  is_active BOOLEAN DEFAULT true
);

-- Transactions
CREATE TABLE transactions (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(user_id),
  transaction_date DATE NOT NULL,
  category_id UUID REFERENCES expense_categories(id),
  amount DECIMAL(10,2),
  description TEXT,
  type VARCHAR(20), -- 'income', 'expense'
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Savings Goals
CREATE TABLE savings_goals (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(user_id),
  user_id UUID REFERENCES users(user_id),
  goal_name VARCHAR(100),
  target_amount DECIMAL(10,2),
  current_amount DECIMAL(10,2),
  deadline DATE,
  priority INTEGER,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Monthly Budgets (snapshot per month)
CREATE TABLE monthly_budgets (
  id UUID PRIMARY KEY,
  user_id UUID REFERENCES users(user_id),
  month DATE NOT NULL,
  category_id UUID REFERENCES expense_categories(id),
  budgeted_amount DECIMAL(10,2),
  actual_amount DECIMAL(10,2),
  UNIQUE(user_id, month, category_id)
);
```

App Architecture

Technology Stack Options

Option 1: Web App (Full Stack)

- **Frontend**: React + TypeScript
- **Backend**: Node.js + Express or Python + FastAPI
- **Database**: PostgreSQL
- **Hosting**: Vercel (frontend) + Render/Railway (backend)
- **Auth**: Clerk or Auth0

Option 2: Mobile App (Native)

- **iOS**: Swift + SwiftUI
- **Android**: Kotlin + Jetpack Compose
- **Backend**: Firebase or Supabase
- **Storage**: Cloud Firestore

Option 3: Cross-Platform

- **Framework**: React Native or Flutter
- **Backend**: Supabase (Backend-as-a-Service)
- **Auth**: Supabase Auth
- **Real-time**: Supabase Realtime

Recommended: Progressive Web App (PWA)

Best balance of reach, cost, and features:

```
Tech Stack:
- Frontend: Next.js 14 + React + TypeScript + Tailwind CSS
- State: Zustand or Redux Toolkit
- Backend: Next.js API Routes or tRPC
- Database: PostgreSQL (via Supabase)
- Auth: NextAuth.js
- Charts: Recharts or Chart.js
- Forms: React Hook Form + Zod
- Hosting: Vercel
```

Feature Mapping: Spreadsheet → App

Phase 1: MVP (Core Features)

Spreadsheet Feature	App Feature	Implementation
Dashboard summary	Home screen	Real-time calculations from DB
Income entry	Income form	CRUD operations
Expense budget	Budget setup	Category management
Monthly tracker	Transaction log	Add/edit/delete transactions
Basic calculations	Automatic totals	Backend calculations

Phase 2: Enhanced Features

Feature	Description	Tech
Bank Integration	Connect to Plaid API	Plaid SDK
Receipt Scanning	OCR for expenses	Google Vision API
Notifications	Budget alerts	Push notifications
Reports	PDF/Excel exports	jsPDF or ExcelJS
Multi-currency	Support multiple currencies	Exchange rate API
Recurring Transactions	Auto-add monthly bills	Cron jobs

Phase 3: Advanced Features

Feature	Description	Tech
AI Insights	Spending predictions	TensorFlow.js
Bill Splitting	Share expenses	Multi-user logic
Investment Tracking	Portfolio integration	Stock API
Tax Export	Generate tax reports	Custom reports
Family Accounts	Multi-user budgets	Role-based access
Dark Mode	Theme switching	CSS variables

Formula Translation

Spreadsheet Formula → App Code

Example 1: Total Income

excel

```excel
Spreadsheet: =SUM(B6:B13)
```

javascript

```javascript
// App Code
const totalIncome = incomeSources.reduce((sum, source) => {
  return sum + parseFloat(source.amount || 0);
}, 0);
```

Example 2: Savings Rate

```excel
Spreadsheet: =IF(B6=0,0,B8/B6)
```

```javascript
// App Code
const savingsRate = totalIncome === 0
 ? 0
 : netIncome / totalIncome;
```

Example 3: Budget vs Actual

```excel
Spreadsheet: =B17-B18
```

```javascript
// App Code
const budgetDifference = budgetedAmount - actualAmount;
const percentOfBudget = budgetedAmount === 0
 ? 0
 : (actualAmount / budgetedAmount) * 100;
```

App Screens Breakdown

1. Dashboard Screen

Data to Display:

- Total Income (current month)
- Total Expenses (current month)
- Net Income
- Savings Rate
- Budget vs Actual by category (bar chart)
- Spending trend (line chart, last 6 months)

API Endpoints:

```
GET /api/dashboard?month=2026-01
Response: {
  totalIncome: 5000,
  totalExpenses: 3500,
  netIncome: 1500,
  savingsRate: 0.30,
  categories: [...]
}
```

2. Transactions Screen

Features:

- Add new transaction
- Filter by date range, category, type
- Search transactions
- Edit/delete transaction
- Bulk import (CSV)

API Endpoints:

```
GET /api/transactions?start_date=2026-01-01&end_date=2026-01-31
POST /api/transactions
PUT /api/transactions/:id
DELETE /api/transactions/:id
```

3. Budget Screen

Features:

- Set category budgets
- View budget templates
- Copy from previous month
- Set recurring budgets

API Endpoints:

```
GET /api/budgets?month=2026-01
POST /api/budgets
PUT /api/budgets/:category_id
```

4. Goals Screen

Features:

- Create savings goal
- Track progress (progress bar)
- Set deadline
- Allocate monthly savings
- Mark goal as complete

API Endpoints:

```
GET /api/goals
POST /api/goals
PUT /api/goals/:id
DELETE /api/goals/:id
```

5. Reports Screen

Features:

- Income vs Expenses (line chart)
- Category breakdown (pie chart)
- Monthly comparison (bar chart)
- Year-over-year trends
- Export to PDF/Excel

API Endpoints:

```
GET /api/reports/monthly?year=2026
GET /api/reports/category?start_date=2026-01-01&end_date=2026-12-31
GET /api/reports/export?format=pdf&month=2026-01
```

Development Phases

Phase 1: Foundation (Week 1-2)

- [ ] Set up project structure
- [ ] Create database schema
- [ ] Implement authentication
- [ ] Build basic UI components
- [ ] Create API routes for CRUD operations

Phase 2: Core Features (Week 3-4)

- [ ] Income management
- [ ] Expense budget setup
- [ ] Transaction entry
- [ ] Dashboard calculations
- [ ] Basic data validation

Phase 3: Data Visualization (Week 5-6)

- [ ] Dashboard charts
- [ ] Category breakdowns
- [ ] Trend analysis
- [ ] Budget vs actual visualization
- [ ] Mobile responsiveness

Phase 4: Advanced Features (Week 7-8)

- [ ] Recurring transactions
- [ ] Savings goals tracking
- [ ] Report generation
- [ ] Data export (CSV/PDF)
- [ ] Settings and preferences

Phase 5: Polish & Deploy (Week 9-10)

- [ ] Error handling
- [ ] Loading states
- [ ] Offline support (PWA)
- [ ] Performance optimization
- [ ] Testing (unit + integration)
- [ ] Deploy to production

Sample Code Snippets

React Component: Budget Card

typescript

```typescript
// components/BudgetCard.tsx
interface BudgetCardProps {
  category: string;
  budgeted: number;
  actual: number;
  color: string;
}

export function BudgetCard({ category, budgeted, actual, color }: BudgetCardProps) {
  const difference = budgeted - actual;
  const percentage = budgeted === 0 ? 0 : (actual / budgeted) * 100;
  const isOverBudget = actual > budgeted;

  return (
    <div className="border rounded-lg p-4 shadow-sm">
      <div className="flex justify-between items-center mb-2">
        <h3 className="font-semibold">{category}</h3>
        <span className={`text-sm ${isOverBudget ? 'text-red-600' : 'text-green-600'}`}>
          {isOverBudget ? '+' : ''}{formatCurrency(difference)}
        </span>
      </div>

      <div className="space-y-2">
        <div className="flex justify-between text-sm">
          <span>Budgeted:</span>
          <span className="font-medium">{formatCurrency(budgeted)}</span>
        </div>
        <div className="flex justify-between text-sm">
          <span>Actual:</span>
          <span className="font-medium">{formatCurrency(actual)}</span>
        </div>

        {/* Progress bar */}
        <div className="w-full bg-gray-200 rounded-full h-2.5">
          <div
            className={`h-2.5 rounded-full ${
              isOverBudget ? 'bg-red-500' : 'bg-green-500'
            }`}
            style={{ width: `${Math.min(percentage, 100)}%` }}
          />
        </div>

        <div className="text-right text-xs text-gray-500">
          {percentage.toFixed(1)}% of budget
        </div>
      </div>
    </div>
  );
}
```

API Route: Dashboard Data

typescript

```typescript
// app/api/dashboard/route.ts
import { NextRequest, NextResponse } from 'next/server';
import { getServerSession } from 'next-auth';
import { prisma } from '@/lib/prisma';

export async function GET(request: NextRequest) {
 const session = await getServerSession();
 if (!session?.user?.id) {
  return NextResponse.json({ error: 'Unauthorized' }, { status: 401 });
 }

 const searchParams = request.nextUrl.searchParams;
 const month = searchParams.get('month') || new Date().toISOString().slice(0, 7);

 // Get total income for month
 const income = await prisma.transaction.aggregate({
  where: {
   userId: session.user.id,
   type: 'income',
   transactionDate: {
    gte: new Date(`${month}-01`),
    lt: new Date(`${month}-01`).setMonth(new Date(`${month}-01`).getMonth() + 1)
   }
  },
  _sum: { amount: true }
 });

 // Get total expenses for month
 const expenses = await prisma.transaction.aggregate({
  where: {
   userId: session.user.id,
   type: 'expense',
   transactionDate: {
    gte: new Date(`${month}-01`),
    lt: new Date(`${month}-01`).setMonth(new Date(`${month}-01`).getMonth() + 1)
   }
  },
  _sum: { amount: true }
 });

 const totalIncome = income._sum.amount || 0;
 const totalExpenses = expenses._sum.amount || 0;
 const netIncome = totalIncome - totalExpenses;
 const savingsRate = totalIncome === 0 ? 0 : netIncome / totalIncome;
```

```
  // Get expenses by category
  const categoryExpenses = await prisma.transaction.groupBy({
    by: ['categoryId'],
    where: {
      userId: session.user.id,
      type: 'expense',
      transactionDate: {
        gte: new Date(`${month}-01`),
        lt: new Date(`${month}-01`).setMonth(new Date(`${month}-01`).getMonth() + 1)
      }
    },
    _sum: { amount: true }
  });

  return NextResponse.json({
    totalIncome,
    totalExpenses,
    netIncome,
    savingsRate,
    categoryExpenses
  });
}
```

Database Query Hook

typescript

```typescript
// hooks/useTransactions.ts
import { useQuery, useMutation, useQueryClient } from '@tanstack/react-query';

export function useTransactions(startDate: string, endDate: string) {
  return useQuery({
    queryKey: ['transactions', startDate, endDate],
    queryFn: async () => {
      const response = await fetch(
        `/api/transactions?start_date=${startDate}&end_date=${endDate}`
      );
      if (!response.ok) throw new Error('Failed to fetch transactions');
      return response.json();
    }
  });
}

export function useAddTransaction() {
  const queryClient = useQueryClient();

  return useMutation({
    mutationFn: async (transaction: NewTransaction) => {
      const response = await fetch('/api/transactions', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify(transaction)
      });
      if (!response.ok) throw new Error('Failed to add transaction');
      return response.json();
    },
    onSuccess: () => {
      queryClient.invalidateQueries({ queryKey: ['transactions'] });
      queryClient.invalidateQueries({ queryKey: ['dashboard'] });
    }
  });
}
```

Data Migration Strategy

Step 1: Export Spreadsheet Data

python

```python
import pandas as pd

# Read spreadsheet
income_df = pd.read_excel('personal_budget.xlsx', sheet_name='Income')
expenses_df = pd.read_excel('personal_budget.xlsx', sheet_name='Expenses')
tracker_df = pd.read_excel('personal_budget.xlsx', sheet_name='Monthly Tracker')

# Convert to JSON for import
income_data = income_df.to_dict('records')
expenses_data = expenses_df.to_dict('records')
```

Step 2: Import to Database

typescript

```typescript
// scripts/import-from-excel.ts
import { prisma } from './lib/prisma';
import * as fs from 'fs';

async function importData(userId: string) {
  const data = JSON.parse(fs.readFileSync('budget-data.json', 'utf-8'));

  // Import income sources
  for (const income of data.income) {
    await prisma.incomeSource.create({
      data: {
        userId,
        sourceName: income.source,
        amount: income.amount,
        frequency: 'monthly'
      }
    });
  }

  // Import expense categories
  for (const expense of data.expenses) {
    await prisma.expenseCategory.create({
      data: {
        userId,
        categoryName: expense.category,
        budgetedAmount: expense.budget
      }
    });
  }
}
```

Testing Strategy

Unit Tests

typescript

```typescript
// __tests__/calculations.test.ts
describe('Budget Calculations', () => {
  it('calculates savings rate correctly', () => {
    const income = 5000;
    const expenses = 3500;
    const savingsRate = (income - expenses) / income;
    expect(savingsRate).toBe(0.30);
  });

  it('handles zero income', () => {
    const income = 0;
    const expenses = 100;
    const savingsRate = income === 0 ? 0 : (income - expenses) / income;
    expect(savingsRate).toBe(0);
  });
});
```

Integration Tests

typescript

```typescript
// __tests__/api/transactions.test.ts
describe('POST /api/transactions', () => {
  it('creates a new transaction', async () => {
    const response = await fetch('/api/transactions', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({
        date: '2026-01-15',
        amount: 50.00,
        category: 'Food',
        description: 'Groceries'
      })
    });

    expect(response.status).toBe(201);
    const data = await response.json();
    expect(data.amount).toBe(50.00);
  });
});
```

Deployment Checklist

- [ ] Environment variables configured
- [ ] Database migrations run
- [ ] API rate limiting implemented
- [ ] Error tracking (Sentry)
- [ ] Analytics (Plausible/PostHog)
- [ ] SSL certificate
- [ ] CDN for static assets
- [ ] Backup strategy
- [ ] Monitoring (Uptime)
- [ ] Performance testing

Cost Estimate

Hosting (Monthly)

- Vercel (Frontend): $0 - $20
- Supabase (Database): $0 - $25
- Storage: $0 - $5
- **Total**: $0 - $50/month

Services

- Domain: $12/year
- Email (SendGrid): $0 - $15/month
- Monitoring: $0 (free tier)

Development Time

- Solo developer: 8-10 weeks
- Small team (2-3): 4-6 weeks

Resources

Learning

- Next.js Tutorial
- React Query Docs
- Supabase Quickstart
- Tailwind CSS

Tools

- shadcn/ui - UI components
- Recharts - Charts
- date-fns - Date utilities
- Zod - Validation

Deployment

- Vercel Deployment
- Supabase Deployment

Next Steps: Start with Phase 1 (Foundation) and iterate based on user feedback!