California State University, Sacramento
College of Engineering and Computer Science
CSC 131-05: Computer Software Engineering

SOFTWARE DESIGN & IMPLEMENTATION

(Design & Implementation Phase - Waterfall Methodology)

for

PROJECT BRAVO

Prepared By: Sparsh Saraiya, Ronald Salas

Table Of Contents

SACRAMENTO STATE

CSC 131-05: Computer Software Engineering

Project Bravo: Software Design & Implementation Document

Chapter 1: Introduction

1.1 Purpose, Scope, & Objective:

This document serves as a summary of the Expense Tracker design and how it was created. It describes how the system is set up, how each of the primary components is structured, and what technologies were applied to develop it. This document links the application to the SRS requirements and to the completed working application. There is an overview of the architecture, core modules, database approach, and user interfaces, all within the scope of this document. To avoid documenting the same thing, the UML diagrams will be included in a separate modeling document. Therefore, this document mainly contains essential details regarding how the application was created and how it is built. **This document matches both the design & implementation phase of the Waterfall methodology.**

Chapter 2: Software Design

2.1 Architectural Design:

The Expense Tracker follows a client/server architecture that employs a React front-end paired with a Supabase back-end. The React front-end is a single-page application developed using the TypeScript language in conjunction with Vite to build the app. The front-end is responsible for capturing user interactions, validating input, and displaying information. The back-end component consists of hosted PostgreSQL databases and APIs provided by Supabase that authenticate and store user data. Three separate layers make up this application in a logical fashion: The **Presentation Layer** contains React components and pages which provide the interface to the user. Examples would include forms, tables, charts, and navigation, The **Application Logic Layer** contains all of the functions, hooks, and event handlers that control how the application works. This includes functions to manage application state, process user input, access and update the application's data, etc, & The **Data Layer** consists of all the queries and tables within the Supabase database to hold and retrieve user account details, categories, and records of all transactions. The client-server architecture enables the front-end of the application to remain separate from the physical database as all data interactions occur solely through the Supabase client library.

2.2 Database Design:

The data for the Expense Tracker is stored in a Postgres-compatible database hosted on Supabase. A logical model of the data has been built around the transactions table, which captures all income/expense entries in the Expense Tracker application. Each transaction entity contains several fields, including a unique identifier, user reference, date the transaction occurred, transaction type (either income or expense), transaction category, transaction amount, and an optional description of the transaction. The expense tracker application has additional tables available to support users and categories so that the user can associate an entry with a specific user and categorize expenses into groups (Food, Rent, Entertainment, etc.). The database configuration and migration files are located in the supabase folder of the Expense Tracker project, and this folder can be utilized to recreate or migrate a database schema within a Supabase environment. The React front end communicates with the PostgreSQL database via the Supabase JavaScript client, which allows the application to send queries and mutations to the database securely.

2.3 Interface Design:

The application's user interface is made up of several pages and components developed in React that correspond to certain core features of the app: First, an "Add Entry" page allows users to enter information about their income or expenses, set a category for their entry, and store this information. Next, there is a "Categorized Expenses" page that lists expenses by category for quick identification. Thirdly, the third page of the app is a “Month Review” where users can see an overview of their income and expense totals along with balance and Chart graphics generated by Recharts. An “Admin” desktop view is available for administrators which allows them to manage their user’s data and provide more comprehensive summary views of user activity. All these pages are connected through React Router. Each of these pages will have similar UI design principles based on Radix components and Tailwind CSS so they will be user-friendly and responsive across all platforms.

2.4 Component Design:

A reusable architecture of react components and modules separate function areas clearly: a **transaction component layer** where income or expense entries can be created, edited, deleted and listed; a **data access layer** that wraps supabase calls to give functions/hooks to load/update transaction data/user information; A **reporting component layer** that aggregates data to create summary reports passed to charting components for visualisation; layout/UI components (such as navigation bars, forms, cards, dialogue and tables), made from Tailwind CSS and Radix UI, are reused across many screens; Using a component-based approach to designing this application gives you the advantage of building on top of what you have already created rather than starting from scratch, as each feature you want to add, change or remove is independent of the rest of the system, so the maintenance and extension of your app becomes easy.

Chapter 3: Implementation Summary

3.1 Tools & Technologies Used:

The Expense Tracker application is built on React 18 and TypeScript, utilizing Vite for quick builds and hot reloading. To create a modern and consistent interface, Tailwind CSS provides a utility-first styling system, while Radix UI provides accessible, low-level UI components. On the server side, the Expense Tracker leverages Supabase, which provides back-end service functionality with a PostgreSQL database and interface for accessing data and authentication. Other libraries used in the development process include React Router for navigation, Recharts for creating charts and graphs, and React Hook Form for managing forms. To support ongoing collaboration and control, GitHub is used for version control and collaborative efforts.

3.2 Coding & Configuration Overview:

The application's code is organized inside the “src” directory, which contains folders for pages, components, and data utility functions. Each feature, including adding entries, viewing summaries, and admin functionality, is implemented as either a React route or a component. The application uses reusable UI elements throughout, including buttons, input fields, cards, and layouts, to achieve a uniform design and minimize duplication. Files used by Vite, Tailwind, and PostCSS to configure the build, style system, and assets. TypeScript's option to check for types across components provides a greater level of confidence in the correctness of code. The Supabase client is

instantiated via environment variables, which enables the app to use the appropriate backend project and keep credentials secure without hardcoding them into the source code.

3.3 Data Integration & Backend:

The React Components utilize the Supabase JavaScript client for the integration of Data to connect the Front end and the Backend. The components can perform functions such as adding new transactions or querying existing records. Once the component receives the result of the function call, it can then set that into its State and either Render the result as a Table, List, or some other method of Visualisation. The Supabase backend has an API that provides authenticated access to the PostgreSQL Database that is managed by Supabase. The supabase folder of the project contains the configuration to define the Backend Schema for maintaining the Supabase Back end. This means that the project does not require the creation of an additional Custom Server or anything else in order to have a Fully Functional Backend for Persisted Local Data and a Secure Method for Performing Operations on the database; It also means that the system will be able to Scale and/or Evolve when the database schema and APIs change without having to change anything within the existing front end structure of the project.