SOFTWARE DESIGN SPECIFICATION

SOFTWARE DESIGN SPECIFICATION
Project: Personal Finance and Budgeting Application – Group 4: Jacob G, Nick M, Anthony P

1.0 Introduction

This Software Design Specification details the **data**, **architecture**, **interface**, and **component-level design** for the Personal Finance and Budgeting Application. The application's purpose is to enable users to record expenses, set budgets, and view financial summaries securely.

1.1 Goals and Objectives

- **Goal**: Provide a secure, user-friendly solution for personal financial management.
- **Objectives**:
    1. **Organized Data Handling**: Store user, expense, and budget information in a structured manner.
    2. **Clear Architecture**: Separate concerns between front end, back end, and database.
    3. **Scalability & Maintainability**: Employ design principles that allow the system to be extended (adding features like CSV import or external bank integration).

1.2 Statement of Scope

- **Core Functions**:
    - User registration and authentication.
    - Expense CRUD (Create, Read, Update, Delete).
    - Budget CRUD tied to categories.
    - Summaries and analytics (monthly or category-based).
- **Inputs**: Expense attributes (amount, category, date, note), budget settings (category, limit), user details (email, password).
- **Outputs**:
    - Lists of expenses.
    - Budget usage info.
    - Visual summaries (charts, tables).

1.3 Software Context

- **Domain**: Personal finance management.
- **Integration**: Potential for CSV import or external banking APIs.
- **Hosting**: Could be deployed on a cloud service (AWS, Render, Heroku) with a free-tier database (e.g., MongoDB Atlas).
- **Stakeholders**: Primary users (individuals/students who want to track spending), potential administrator (optional).

1.4 Major Constraints

- **Time**: Must be completed within the semester's timeframe.
- **Technical**: Relying on Node.js/TypeScript (or similar) and a NoSQL database like MongoDB.
- **Security**: Must ensure safe storage of user credentials and financial data (JWT, password hashing).
- **Budget**: Uses free-tier or minimal-cost hosting and database services.

2.0 Data Design

2.1 Data Structures

- **User**:

```
interface User {
  _id: string;
  name: string;
  email: string;
  password: string; // stored securely (hashed)
  createdAt: Date;
  updatedAt: Date;
}
```

- **Expense**:

```
interface Expense {
  _id: string;
  userId: string; // references User._id
  amount: number;
  category: string;
  date: Date;
  note?: string;
  createdAt: Date;
  updatedAt: Date;
}
```

- **Budget**:

```
interface Budget {
  _id: string;
  userId: string; // references User._id
  category: string;
  limit: number;
  createdAt: Date;
  updatedAt: Date;
}
```

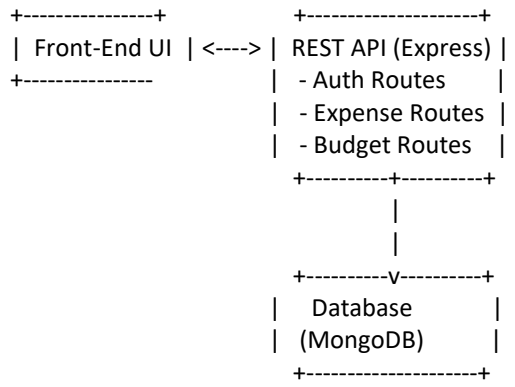
2.2 Database Description

- **MongoDB** (example):
  - **Collections**: users, expenses, budgets.
  - **Indexes**:
    - users.email should be unique to prevent duplicate registration.
    - Potential indexing on expenses.userId for faster queries.
- **Relationships**:
  - **One User → Many Expenses**
  - **One User → Many Budgets**
  - No direct relationship between Budget and Expense beyond sharing the `userId` and `category` strings.

3.0 Architectural and Component-Level Design

3.1 Architecture Diagrams

3.1.1 Logical View



Key Points:

- **Front End** communicates with **REST API** (Express, Node.js) to submit or fetch data.
- The **API** handles authentication (JWT) and interacts with the **MongoDB** database.

3.1.2 Process View

1. **User** → **Login** Request → **Auth Controller** → checks credentials → returns JWT.
2. **User** → **Expense** or **Budget** request → **Expense/Budget Controller** → CRUD operations → **Database**.

3.1.3 Physical View

- **Server** can be hosted on a cloud platform (AWS EC2, Render, Heroku).
- **Database** instance is managed by a service like MongoDB Atlas or a self-hosted DB.
- **Client** is typically the user's web browser or mobile app.

3.2 Description for Components

3.2.1 Component: Authentication Controller

- **Interface Description**
  - **Inputs**: User registration (name, email, password), Login (email, password).
  - **Outputs**: Confirmation message (registration), JWT token (login).
  - **Exceptions**: Invalid credentials, email already in use.
- **Static Models**
  - Typically includes a **User** model instance with validations.
- **Dynamic Models**
  - **Sequence**: POST /register or POST /login → verify data → hash/check password → respond with success or error.

3.2.2 Component: Expense Controller

- **Interface Description**
  - **Inputs**: Expense data (amount, category, date, note), user authentication (JWT).
  - **Outputs**: Expense object(s) with status messages.
  - **Exceptions**: Unauthorized access (no token), invalid data.
- **Static Models**
  - Uses **Expense** schema in the database.
- **Dynamic Models**
  - **Sequence**: POST /expense → validate JWT → create expense → save to DB → return new expense.
  - **Sequence**: GET /expense → validate JWT → fetch expenses by userId → return list.

3.2.3 Component: Budget Controller

- **Interface Description**
  - **Inputs**: Budget details (category, limit), user authentication (JWT).
  - **Outputs**: Budget record(s) with success/failure messages.
  - **Exceptions**: Unauthorized access, invalid category/limit.
- **Static Models**
  - Uses **Budget** schema.
- **Dynamic Models**
  - **Sequence**: POST /budget → check if budget for category exists → create or update → return record.
  - **Sequence**: GET /budget → retrieve budgets by userId.

3.3 External Interface Description

- **HTTP/HTTPS**: The API listens on a configured port (5000) and expects JSON payloads.
- **Database Connection**: The server uses a MongoDB client or ORM/ODM (Mongoose) to manage data.
- **Optional**: External APIs for advanced features (bank account integration, CSV import).

4.0 User Interface Design

4.1 Description of the User Interface

- **Login/Register** Screen: Simple forms for email/password (+ name for register).
- **Dashboard**:
    - **Expense List**: Table of expenses (amount, date, category).
    - **Add Expense Form**: Quick form to add new expenses.
    - **Budget Overview**: Summarize budgets and how much has been spent in each.
- **Budget Management** Screen:
    - **Budget List**: Table of category vs. limit.
    - **Add/Update Budget Form**: Input for category and limit.
- **Visual Summaries** (optional/advanced):
    - Pie chart for spending categories, bar/line chart for monthly totals, etc.

4.2 Interface Design Rules

- **Consistency**: Use consistent color scheme and layout for forms, tables, and buttons.
- **Responsiveness**: Ensure screens scale for desktop, tablet, and mobile.
- **Usability**: Clear labels, input validation, error messages in red.
- **Security**: Always use HTTPS in production, do not show plain-text passwords, clearly label logout buttons.

5.0 Restrictions, Limitations, and Constraints

- **Scalability**: Large user adoption may require load balancing or optimization for database queries.
- **Time Constraints**: Some features like advanced alerts or external integration may be postponed.
- **Free-Tier Services**: Could limit the database size, concurrency, or performance metrics.
- **User Data Sensitivity**: Must ensure best practices for storing PII (email, password).

6.0 Appendices

6.1 Requirements Traceability Matrix

Requirement	Design Component	Implementation
R1: User Registration	Auth Controller	POST /auth/register
R2: User Login	Auth Controller	POST /auth/login
R3: Expense Management	Expense Controller, Expense DB	POST /expense, GET /expense
R4: Budget Management	Budget Controller, Budget DB	POST /budget, GET /budget
R5: Summaries/Analytics	Expense + Budget Controllers	Potential front-end chart

6.2 Implementation Issues

- **Database Migrations**: If schema changes, data migration scripts may be needed.
- **Testing Strategy**: Unit tests for controllers, integration tests for entire user flows, load testing if feasible.
- **Deployment**: Must coordinate environment variables (DB URI, JWT secret) across dev, test, and production.