1.Introduction

Project Title:

Personal Expense Tracker

Team Members:

TELLAMEKALA NAGA SUCHARITHA	-	TEAM LEADER
UPPU SHANMUKHA SRIVALLI	-	TEAM MEMBER 1
TELAGATHOTI YAHAJIYELU	-	TEAM MEMBER 2
POLAM YADUNANDAN	-	TEAM MEMBER 3
SHAIK SHABNA	-	TEAM MEMBER 4

2. PROJECT OVERVIEW

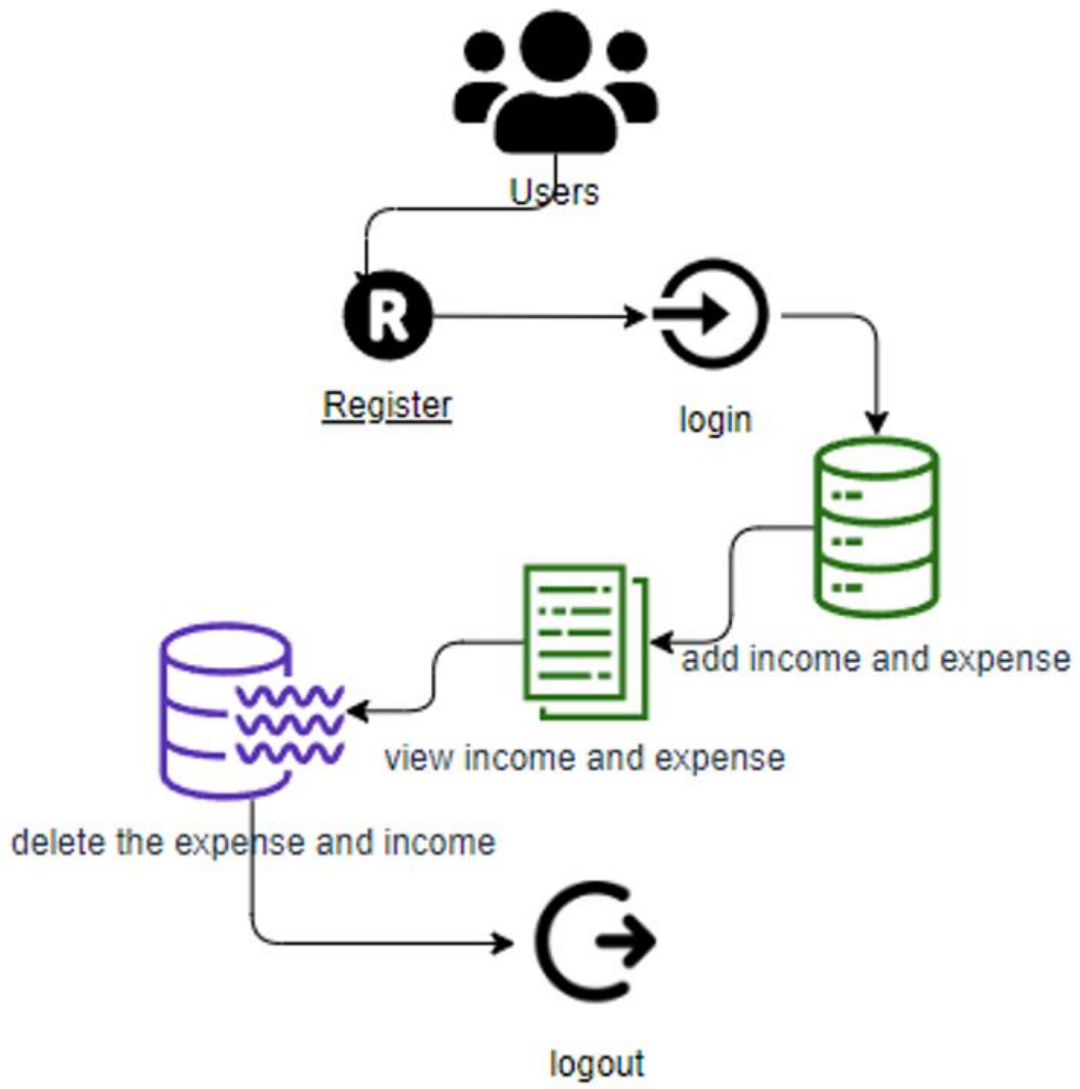
Purpose:

This project shows how to use the entire development process to create a personal expenses tracker application using Full-stack development with MongoDB as a database. The app aims to simplify personal financial management by providing users with a seamless platform where they can effectively track their expenses. Through a user-friendly interface user an access and allocate expenses, view analytics, set budget goals and receive timely reports. The application leverages MongoDB's focused data model to provide powerful data management and scalability. Integration of Front-End Technologies such as HTML, CSS and JavaScript with Back-End Technologies such as Node.js, Express.js and MongoDB increase effort and efficiency. The project not only demonstrates knowledge of development as a whole, but also meet requirements of good financial management in today's world. The personal expenses tracker addresses the contemporary need for streamlined personal finance management to the Dynamics lifestyle of modern users.

Key Features:

- Secure User Authentication
- Easy Expense Management
- Detailed Expense Analysis
- Budget Planning with Notifications
- Comprehensive Functionality
- Enhanced Security Features
- User Friendly Interface
- Seamless Navigation

3. Architecture



Front end:

In expense tracker application, the front-end architecture is built primarily using React, a JavaScript library for building user interfaces. The front-end is responsible for handling the user interactions, making API calls to the back end, and rendering the data dynamically.

1. React Application Structure

The front end will be a React-based single-page application (SPA) that interacts with the backend via APIs. The typical structure includes:

Components: These components represent different parts of the user interface, such as forms, tables, buttons, and dashboards.

Pages/Views: These are full-page components that act as containers for specific views of the app.

Routing: It helps in moving between different views (like /add, /view, /analytics) without reloading the page.

API Integration

The front-end interacts with the back-end API, typically using **Axios** or the native fetch API. The API is used for:

- **Fetching expenses**: Get the list of all expenses from the server (MongoDB) and render them.
- **Adding expenses**: Send data (like name, amount, category, date) to the backend for adding a new expense.
- **Updating expenses**: Modify existing expense details.
- **Deleting expenses**: Remove an expense from the list

2. Error Handling & Notifications

Handling errors (like failed API requests) and providing feedback to the user is essential. Use libraries like **React Toastify** or custom notification components for

1. Displaying success messages when an expense is added.
2. Showing error messages when something goes wrong

UI/UX Design :

For styling and layout, you may use:

- **CSS Modules / SASS / Styled Components**: Depending on your preference for styling. CSS frameworks like Bootstrap or Material-UI can be used to speed up development and create responsive, mobile-friendly layouts.
- **User Interface Elements**:
- **Tables/Lists**: To display a list of expenses.
  - **Forms**: For adding or editing expenses.
  - **Charts**: Use libraries like Chart.js or Recharts to visualize expenses based on categories, monthly expenses, etc.
  - For confirming deletions, displaying form errors, etc.

Back End:

In a full-stack MERN (MongoDB, Express, React, Node.js) expense tracker project, the **back-end architecture** is responsible for handling the business logic, data storage, authentication, and API services. It typically leverages **Node.js** and **Express** for the server-side logic, with **MongoDB** as the database to store expense data.

1. Node.js and Express as the Backend Framework

Node.js is a runtime environment that allows running JavaScript on the server, while **Express.js** is a minimal and flexible web framework that

provides robust features for building web and API applications. Express is used to define routes, handle requests, and manage middleware.

- **Node.js** provides the environment to execute server-side code.
- **Express.js** simplifies the creation of RESTful APIs and middleware management.

2. **API Layer**

The backend in a MERN stack application exposes a set of RESTful APIs to the front end. These APIs perform CRUD (Create, Read, Update, Delete) operations on the expense data and handle any other requests from the client.

HTTP Methods:

- GET: Retrieves resources (e.g., list of expenses).
- POST: Submits data to create a new resource (e.g., add a new expense).
- PUT: Updates a resource (e.g., editing an expense).
- DELETE: Removes a resource (e.g., delete an expense).

Data base (MongoDB):

1.Database Schema

MongoDB, being a **NoSQL** database, does not enforce strict schemas as in relational databases, but it is still good practice to define the data structure using **Mongoose**, an Object Data Modeling (ODM) library for In a **full-stack MERN (MongoDB, Express, React, Node.js) expense tracker** project, the **database schema** and interactions with **MongoDB** are crucial for efficiently storing and retrieving expense data, user data, and any other related information.

MongoDB and Node.js. Mongoose provides schema definitions and validation, allowing us to model and enforce structure on the data.

2.Expense Schema

The **Expense** schema stores individual expense records. Each expense typically contains fields such as amount, category, date, and the user who created it (in case of user-specific expense tracking).

3.User Schema

If you are managing user-specific data, you'll need a **User** schema to store information about users, authentication details, etc.

4.Budget Schema

If the expense tracker includes budgeting features, you could add a **Budget** schema that ties expenses to a monthly or yearly budget

5. Relationships and Referencing

1.User-Expense Relationship: Expenses are associated with users via the userId field in the **Expense** schema. This creates a **one-to-many relationship**, where a single user can have many expenses.

2.Referencing: MongoDB uses references (ObjectId) to connect related documents. In this case, userId in the **Expense** schema is a reference to the **User** model. Mongoose's populate method can be used to retrieve related documents.

3.Setup Instructions

Prerequisites :

To develop a full-stack expense tracker app using React js, Node.js, and MongoDB, there are several prerequisites you should consider. Here are the key prerequisites for developing such an application:

- Node.js and npm
- MongoDB
- Express.js
- React
- HTML,CSS and JavaScript
- Database Connectivity
- Development Environment
- Version control

Installation :

To clone, install dependencies, and set up environment variables for a full-stack MERN (MongoDB, Express, React, Node.js) expenses tracker project, here is a step-by-step guide:

1.Clone the Project from GitHub

- First, clone the project repository from GitHub to your local machine

````
# Open your terminal or command prompt and run:
git clone <repository-url>
````

- Replace <repository-url> with the actual GitHub URL of the project.

````
cd expenses-tracker
````

2.Install Dependencies

a.Server-Side (Backend)
Go to the backend folder and install the necessary Node.js packages

````cd backend # If the backend is in a folder called 'backend'
npm install````

- This command will install all the dependencies listed in the package.json file for the server-side.

**b. Client-Side (Frontend)**
Go to the client-side folder and install the dependencies.

````cd ../frontend  # If the frontend is in a folder called 'frontend'

npm install````

- This will install all the dependencies for the React app

**3.Set Up Environment Variables**

Both backend and frontend environments may require environment variables to be set up**.**

**a. Backend Environment Variables**

1.Create a .env file in the backend directory (if it doesn't already exist).

````touch .env````

2.Add the necessary environment variables. For an expenses tracker, these might include

````MONGO_URI=<your-mongodb-uri>

JWT_SECRET=<your-jwt-secret>

PORT=5000````

- **MONGO_URI:** The connection string to your MongoDB database.
- **JWT_SECRET:** A secret key for JSON Web Token (JWT) authentication**.**
- **PORT:** The port number on which the backend server will run

### b. Frontend Environment Variables

1.Create a .env file in the frontend directory if required.

~~~
````touch .env````
~~~

2.Set up any necessary variables. For example, to connect to the backend API:

~~~
````REACT_APP_API_URL=http://localhost:5000/api````
~~~

**REACT_APP_API_URL:** This should point to your backend's API endpoint

### 3.Run the Backend Server:

Now that the backend dependencies are installed and environment variables are set, start the backend server.

~~~
````npm run dev # or npm start````
~~~

- ➢ Ensure the backend server is running without errors.

### 5. Run the Frontend Server

Go back to the frontend folder and start the React development server.

~~~
````cd ../frontend
npm start````
~~~

- ➢ The frontend should now be running, and you can access it in your browser (typically at http://localhost:3000).

### 6. **Connecting Frontend and Backend**

- ➢ Ensure the API endpoints set in the frontend (in .env or configuration files) point to the backend server (e.g., http://localhost:5000/api). Test the connection by interacting with the app, such as submitting expenses, logging in, etc

# 5.Folder structure

## Client:

### 1. Directory Structure

A well-organized React frontend might have the following directory structure:

```
frontend/
│
├── public/
│   └── index.html
│
├── src/
│   ├── components/
│   │   ├── ExpenseList.js
│   │   ├── ExpenseForm.js
│   │   ├── Header.js
│   │   ├── Footer.js
│   │   └── Dashboard.js
│   │
│   ├── pages/
│   │   ├── Login.js
│   │   ├── Register.js
│   │   └── Home.js
│   │
│   ├── context/
│   │   ├── ExpenseContext.js
│   │   └── AuthContext.js
│   │
│   ├── services/
│   │   └── api.js
│   │
│   ├── hooks/
│   │   └── useExpenses.js
│   │
│   ├── App.js
│   ├── index.js
│   ├── routes.js
│   └── styles/
│      └── main.css
│
└── package.json
```

**2. Main Folders and Files**

Here's what each part of the structure does:

**a. public/**

This folder contains static assets like index.html, where the React app is mounted into the root div. This file rarely needs modification in most projects.

**index.html**: The main HTML file that renders the entire React app.

**b. src/**

The src folder contains all the JavaScript/JSX code for your application, including components, pages, context, hooks, and styles.

**c. components/**

This folder holds reusable components like forms, lists, headers, and footers. These components can be used across multiple pages.

- **ExpenseList.js**: A component that displays the list of all expenses. It fetches data from the backend via context or props and renders each expense.
- **ExpenseForm.js**: A component containing the form to add or edit an expense. It will likely have form fields for things like expense name, amount, category, and date.
- **Header.js**: A component for the application's header, which might include the app's title and navigation links (e.g., Home, Login, Register).
- **Footer.js**: A footer component with any necessary links or information.
- **Dashboard.js**: A high-level component that combines other components like ExpenseList and ExpenseForm and may provide additional stats like total expenses.

**d. pages/**

The pages folder contains top-level components representing different routes in the app.

- **Login.js**: A page for user login with a form for entering email and password.
- **Register.js**: A page for user registration where users can create an account.
- **Home.js**: The homepage, which may show the dashboard of expenses or information about the app.

Each of these pages is mapped to a route (more on that later in routes.js).

**e. context/**

The context/ folder contains the global state management logic. In a MERN project, you might use the Context API to manage states like authentication and expenses.

- **ExpenseContext.js**: This file holds the logic for managing expenses in the global state, including fetching, adding, editing, and deleting expenses.
- **AuthContext.js**: This file handles user authentication, such as login and logout, and holds information about the current user.

**f. services/**
The services folder contains files that make API calls to the backend.

- **api.js**: This file is where you write functions that interact with the backend. It might include functions like getExpenses(), addExpense(), loginUser(), etc. These functions would use fetch or axios to send HTTP requests to the Express backend API.

**g. hooks/**
This folder contains custom React hooks.

- **useExpenses.js**: A custom hook to fetch, create, update, and delete expenses by utilizing the ExpenseContext or calling the API directly.

**h. App.js**
The main component that acts as the root for the entire React app. It usually includes routing and the setup of global state providers (e.g., wrapping the app in ExpenseContextProvider and AuthContextProvider).

**i. index.js**
The entry point for the React application. It renders the <App /> component and mounts it to the DOM. If you are using Redux, React Context, or other state management tools, this is where you would set it up globally.

**j. routes.js**
This file defines the routes for the app. It uses React Router to map different URL paths to specific page components

**3. Component Breakdown**

- **Dashboard**: This will be the central component that aggregates all the necessary data and displays summaries, expense lists, and a form to add expenses.
- **ExpenseList**: This component maps over an array of expenses (fetched from the backend or global state) and renders each one.
- **ExpenseForm**: This form allows users to input details about a new expense (such as name, amount, and category) and either submit or edit an existing expense.
- **Authentication Pages**: Pages like Login.js and Register.js will manage user authentication, allowing users to log in or register.

# Server:

## 1. Directory Structure

A well-organized Node.js backend project might follow this structure

```
backend/
│
├── config/
│   └── db.js
│
├── controllers/
│   ├── expenseController.js
│   └── userController.js
│
├── middleware/
│   ├── authMiddleware.js
│   └── errorMiddleware.js
│
├── models/
│   ├── expenseModel.js
│   └── userModel.js
│
├── routes/
│   ├── expenseRoutes.js
│   └── userRoutes.js
│
├── utils/
```

```
│   └── generateToken.js
│
├── .env
├── server.js
├── package.json
└── README.md
```

**2. Main Folders and Files**

Let’s break down what each part of this structure does:

**a. config/**

This folder stores configuration files, primarily for setting up database connections or other application configurations.

**db.js**: This file contains the logic to connect to MongoDB using the mongoose library.

**b. controllers/**

Controllers handle the business logic and interact with models to process incoming requests and send appropriate responses.

**expenseController.js**: Contains the logic for managing expenses (create, update, delete, and get expenses)

**userController.js**: Contains the logic for handling user-related actions such as registration, login, and getting user profiles.

**c. middleware/**

Middleware functions process requests between the client and server, handling things like authentication, error handling, etc.

**authMiddleware.js**: Verifies the user’s authentication status using JWT.

**errorMiddleware.js**: Custom error handling middleware to catch and handle errors gracefully.

**d. models/**

Models represent the data structure and define how documents are stored in MongoDB. They typically use Mongoose schemas.

**expenseModel.js**: Defines the schema for expense documents in MongoDB.

**userModel.js**: Defines the schema for user documents, including a method for password validation.

**e. routes/**

The routes folder contains route definitions that map URLs to controller functions. Each route is responsible for receiving HTTP requests and passing them to the correct controller.

**expenseRoutes.js**: Routes for handling expenses (fetching, adding, editing, and deleting expenses).

**userRoutes.js**: Routes for handling user-related actions such as login, registration, and getting user profile information.

**f. utils/**

Utility functions that can be used across the backend. In this case, it includes token generation.

**generateToken.js**: Generates a JWT token for  a authenticated users.

# 6.Running the Application

To start both the frontend and backend server you need to run two separate processes in different terminal windows or tabs.

Assuming the standard directory structure:

~~~
````/expenses-tracker
  /client   (React frontend)
  /server   (Node.js/Express backend)````
~~~

1.Starting the Backend (Node.js/Express server) :

Navigate to the backend folder, usually something like /server

~~~
````cd server
npm install   # Install backend dependencies
npm run dev    # Start the backend server (commonly using nodemon for auto-reload)````
~~~

This command starts the backend server on a specific port (often http://localhost:5000). The dev script typically runs nodemon to automatically restart the server on changes. If there is no dev script, you can use:

~~~
````node index.js   # or the entry file of your backend (often index.js or app.js)````
~~~

2. Starting the Frontend (React app)
Navigate to the frontend folder, usually something like /client.

~~~
````cd client
npm install   # Install frontend dependencies
npm start     # Start the React development server````
~~~

- ➢ This will start the frontend React application on a different port (usually http://localhost:3000)
- ➢ Once both servers are running, the frontend will likely make API requests to the backend (Express server). If needed, make sure the API URLs in your React code match the backend server's URL (e.g., http://localhost:5000).

7.API Documentation

In the backend (Node.js/Express) would expose a set of RESTful API endpoints for managing users, authentication, and expenses. Here's an example of how these endpoints might be structured.

Authentication Endpoints:

1.POST/api/auth/register:
Description: Registers a new user.
Request Body:

~~~
````{
  "name": "John Doe",
  "email": "johndoe@example.com",
  "password": "password123"
}````
~~~

Response:

~~~
````{
  "message": "User registered successfully",
  "token": "JWT_TOKEN"
}````
~~~

2.POST /api/auth/login:
Description: Logs in a user and returns a JWT token.
Request Body:

~~~
````{
  "email": "johndoe@example.com",
  "password": "password123"
}````
~~~

Response:

~~~
````{
  "message": "Login successful",
  "token": "JWT_TOKEN"
}````
~~~

3.GET /api/auth/me:
Description: Retrieves details of the currently authenticated user.

Headers:

~~~
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
~~~

Response:

~~~
````{
  "id": "userId",
  "name": "John Doe",
  "email": "johndoe@example.com"
}````
~~~

Expense Endpoints:

3.GET /api/expenses:

Description: Retrieves a list of all expenses for the authenticated user**.**

Headers:

~~~
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
~~~

Response:

~~~
````{
    "id": "expenseId1",
    "description": "Groceries",
    "amount": 50.00,
    "category": "Food",
    "date": "2024-01-15"
  },
  {
    "id": "expenseId2",
    "description": "Electricity Bill",
    "amount": 100.00,
    "category": "Utilities",
    "date": "2024-01-10"
  }````
~~~

5.POST /api/expenses:

Description: Adds a new expense for the authenticated user**.**

Headers:

~~~
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
~~~

Request Body:

~~~
````{
  "description": "Dinner at restaurant",
  "amount": 30.00,
  "category": "Food",
  "date": "2024-01-20"
}````
~~~

Response:

~~~
````{
  "expense": {
    "id": "newExpenseId",
    "amount": 30.00,
    "category": "Food",
    "date": "2024-01-20"
 }````
~~~

6.GET /api/expenses/:id:

Description: Retrieves a specific expense by its ID.

Headers:

~~~
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
~~~

Response:

~~~
````{
  "id": "expenseId1",
  "description": "Groceries",
  "amount": 50.00,
  "category": "Food",
  "date": "2024-01-15"
}````
~~~

7.PUT /api/expenses/:id:

Description: Updates an existing expense by its ID**.**

Headers:

~~~
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
~~~

Request Body (Only include fields to update):

~~~
````{
  "amount": 60.00
}````
~~~

Response:
```
{
  "expense": {
    "id": "expenseId1",
    "amount": 60.00,
    "category": "Food",
    "date": "2024-01-15"
  }
}
```
8.DELETE /api/expenses/:id:
Description: Deletes an existing expense by its ID.
Headers:
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
**Response**:
````{
  "message": "Expense deleted successfully"
}````

# 8.Authentication

**Authentication** and **Authorization** are key components to ensure that only registered users can access their data and perform specific actions.

## 1. Authentication

**Authentication** is the process of verifying a user's identity, typically using credentials like email and password. Once authenticated, a user is issued a JWT (JSON Web Token), which allows the server to recognize the user in future requests.

**How Authentication Works -**

- **Registration (/api/auth/register)**:
    1. **User Sign-up**: When a user signs up, they send their name, email, and password to the backend via a POST request.
    2. **Password Hashing**: The backend uses a library like bcrypt to hash the password before saving it in the database for security purposes.
    3. **User Creation**: After successful validation and hashing, the user's data (name, email, and hashed password) is stored in MongoDB.
    4. **JWT Token Generation**: Once registered, the backend creates a JWT for the user and sends it as part of the response. This token can be used for subsequent authentication (login).

~~~
````const token = jwt.sign({ userId: user._id }, process.env.JWT_SECRET, {
  expiresIn: '1h',
});````
~~~

**Login (/api/auth/login)**:

1. **User Login**: When a user logs in, they provide their email and password.
2. **Password Verification**: The backend checks if the user exists and compares the provided password with the hashed password stored in the database using bcrypt.compare().
3. **JWT Token Issuance**: If the login credentials are valid, the backend generates a JWT token and returns it to the user.

Example response after login:

~~~
````{
  "message": "Login successful",
  "token": "JWT_TOKEN"
}````
~~~

**2. Authorization**

Authorization ensures that a user can only access resources (e.g., expenses, profiles) they are permitted to access. In this project, this is enforced using the JWT token that was issued during authentication.

**How Authorization Works -**

- **JWT Token Storage**:
  - After login, the JWT token is typically stored in the client-side (frontend) in localStorage or sessionStorage (or more securely, in a cookie).
  - The token is included in the Authorization header of every HTTP request that requires authorization (e.g., accessing user’s expenses).

Example request with JWT:

~~~
````{
  "Authorization": "Bearer JWT_TOKEN"
}````
~~~

In the expenses tracker, authorization ensures that a user can only access or modify their own data (i.e., expenses). Here’s how it works:

- **Fetching User's Expenses**: When the user requests to fetch their expenses (GET /api/expenses), the middleware verifies their token, extracts their user ID, and queries the database for expenses associated with that user ID.

~~~
`````// Get all expenses for the authenticated user
router.get('/expenses', authenticateToken, async (req, res) => {
  const expenses = await Expense.find({ userId: req.user });
  res.json(expenses);
});`````
~~~

- **Updating or Deleting an Expense**: For routes like updating (PUT /api/expenses/:id) or deleting (DELETE /api/expenses/:id), the middleware ensures that the expense belongs to the authenticated user by checking the expense's userId field against req.user (the authenticated user's ID).

Example of protecting update logic:

~~~
````router.put('/expenses/:id', authenticateToken, async (req, res) => {
  const expense = await Expense.findById(req.params.id);

  if (!expense || expense.userId.toString() !== req.user) {
    return res.status(403).json({ message: 'Access denied' });
  }

  // Proceed with updating the expense
  expense.description = req.body.description;
  await expense.save();
  res.json(expense);
});`````
~~~

Tokens:

Tokens(especially JSON Web Tokens or JWTs) are commonly used for authentication and session management.

1. JSON Web Tokens (JWT)

JWTs are the most common method for authentication in modern web applications, especially in full-stack MERN projects. They provide a stateless and scalable solution for managing user sessions and verifying their identity.

JWTs are composed of three parts:

1. **Header**: Contains the type of token (JWT) and the signing algorithm (e.g., HS256 for HMAC-SHA256).
2. **Payload**: Contains claims, which are typically user data, such as the user's ID, email, and any other relevant information.
3. **Signature**: A cryptographic signature used to verify that the token has not been tampered with.

Example response after successful login:

~~~
````{
  "message": "Login successful",
  "token": "JWT_TOKEN"
}````
~~~

How JWT is Used in the Expense Tracker Project:

- **Authentication Flow**:
    1. **User Login**: The user logs in by sending their credentials (email and password) to the backend (POST /api/auth/login).
    2. **Token Issuance**: If the login credentials are valid, the backend generates a JWT, signs it using a secret key, and returns it to the user.
    3. **Token Storage**: The frontend stores the JWT (typically in **localStorage** or **sessionStorage**). This token is included in the headers of subsequent API requests.

3.Sessions:

Sessions are a traditional method of handling authentication and user sessions.

How Sessions Work:

- **Session-based Authentication Flow**:
    1. **User Login**: The user logs in by sending their credentials to the backend.
    2. **Session Creation**: If the credentials are valid, the server creates a **session** for the user and stores session data (e.g., user ID) on the server (usually in memory or a database).
    3. **Session ID Storage**: The session ID is stored in a cookie on the client's browser.
    4. **Subsequent Requests**: For each subsequent request, the client sends the session ID in the cookie. The server validates the session ID, retrieves the session data (e.g., user ID), and allows the request to proceed.

Example of session-based authentication:

~~~
````const session = require(,express-session,);
app.use(session({
  secret: 'yourSecretKey',
  resave: false,
  saveUninitialized: false,
  cookie: { secure: true, maxAge: 60000 }
})); `````
~~~

➢ **JWTs** are commonly used for **authentication and authorization** due to their stateless nature and ease of use in single-page applications. They are typically stored on the client side (either in localStorage, sessionStorage.

10.User Interface

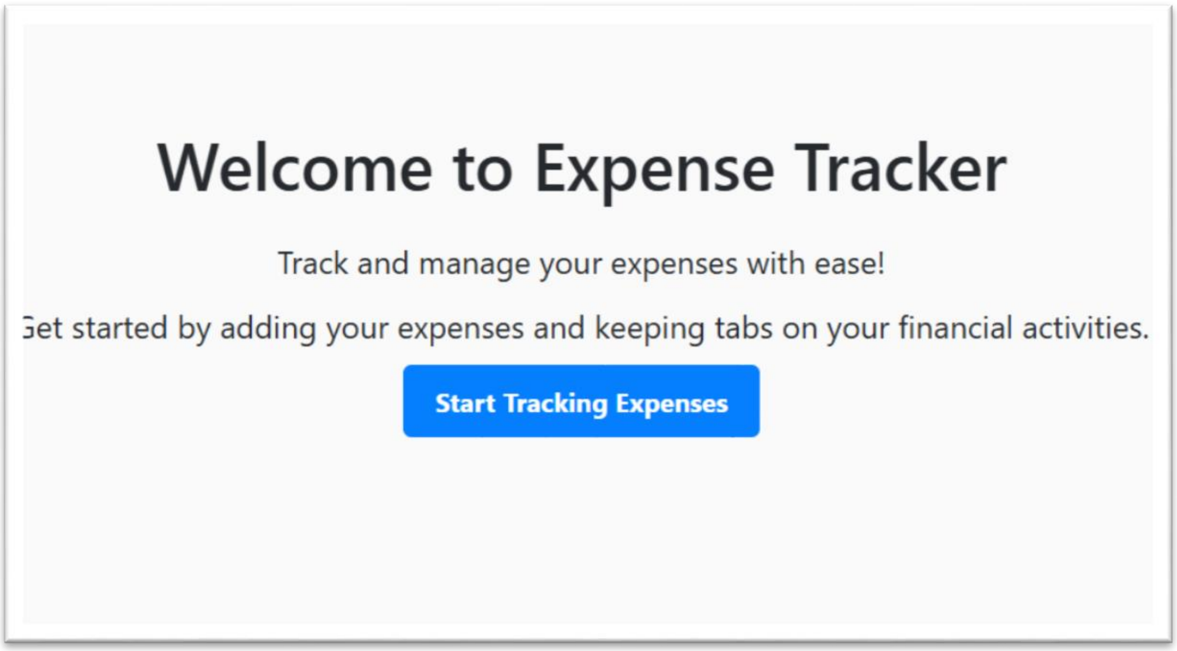

Welcome to Expense Tracker
Track and manage your expenses with ease!
Get started by adding your expenses and keeping tabs on your financial activities.
Start Tracking Expenses

Fig:Landing Page

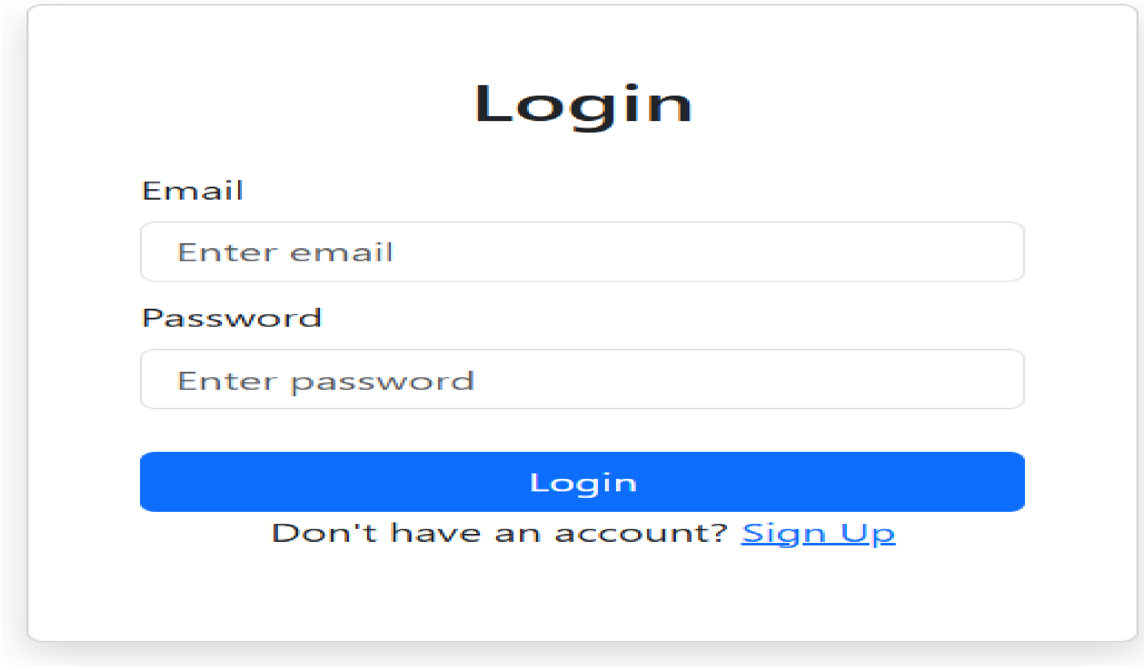


Fig : Sign Up Page

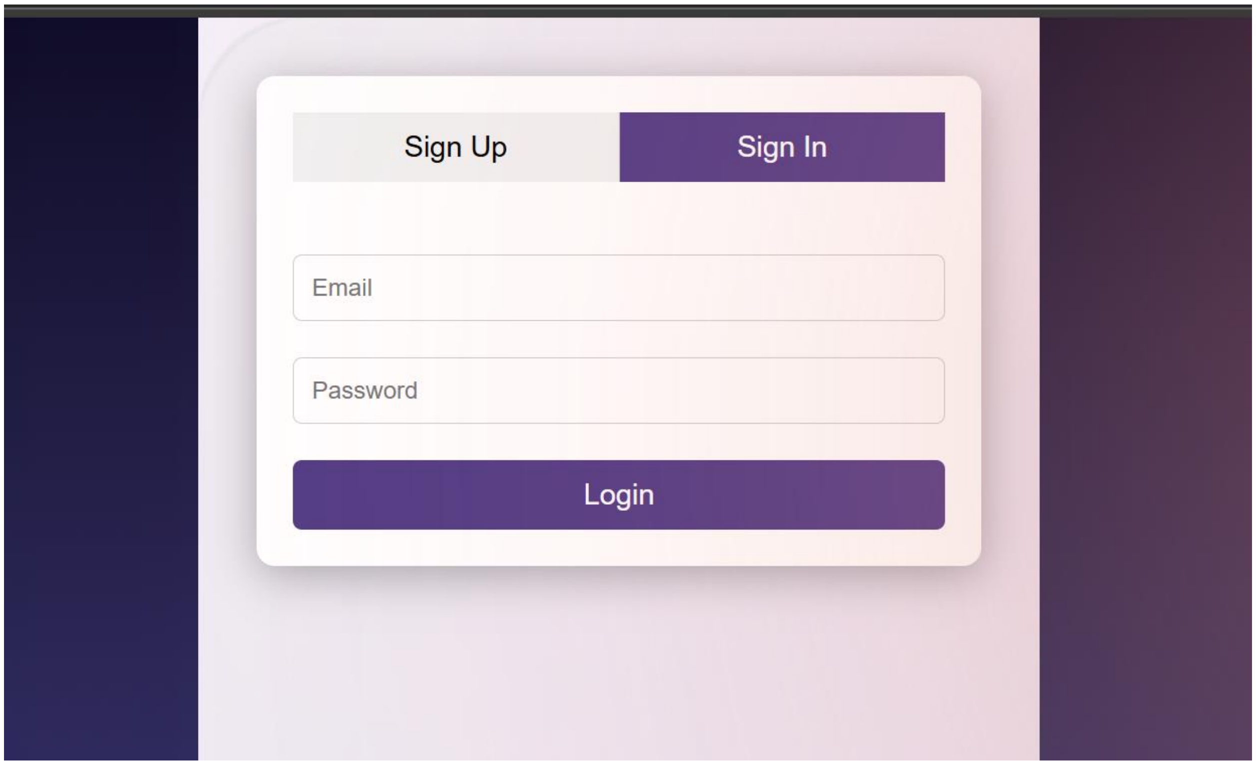
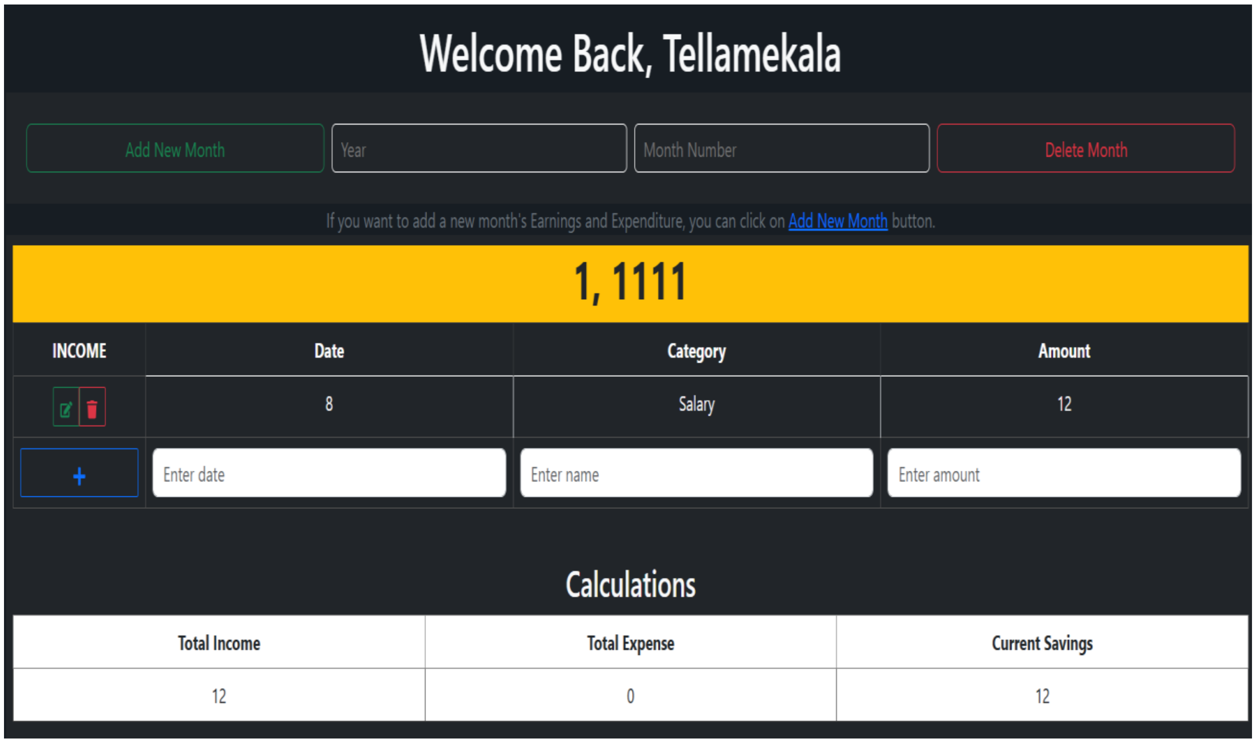


Fig:Sign In Page



If you want to add a new month's Earnings and Expenditure, you can click on Add New Month button.

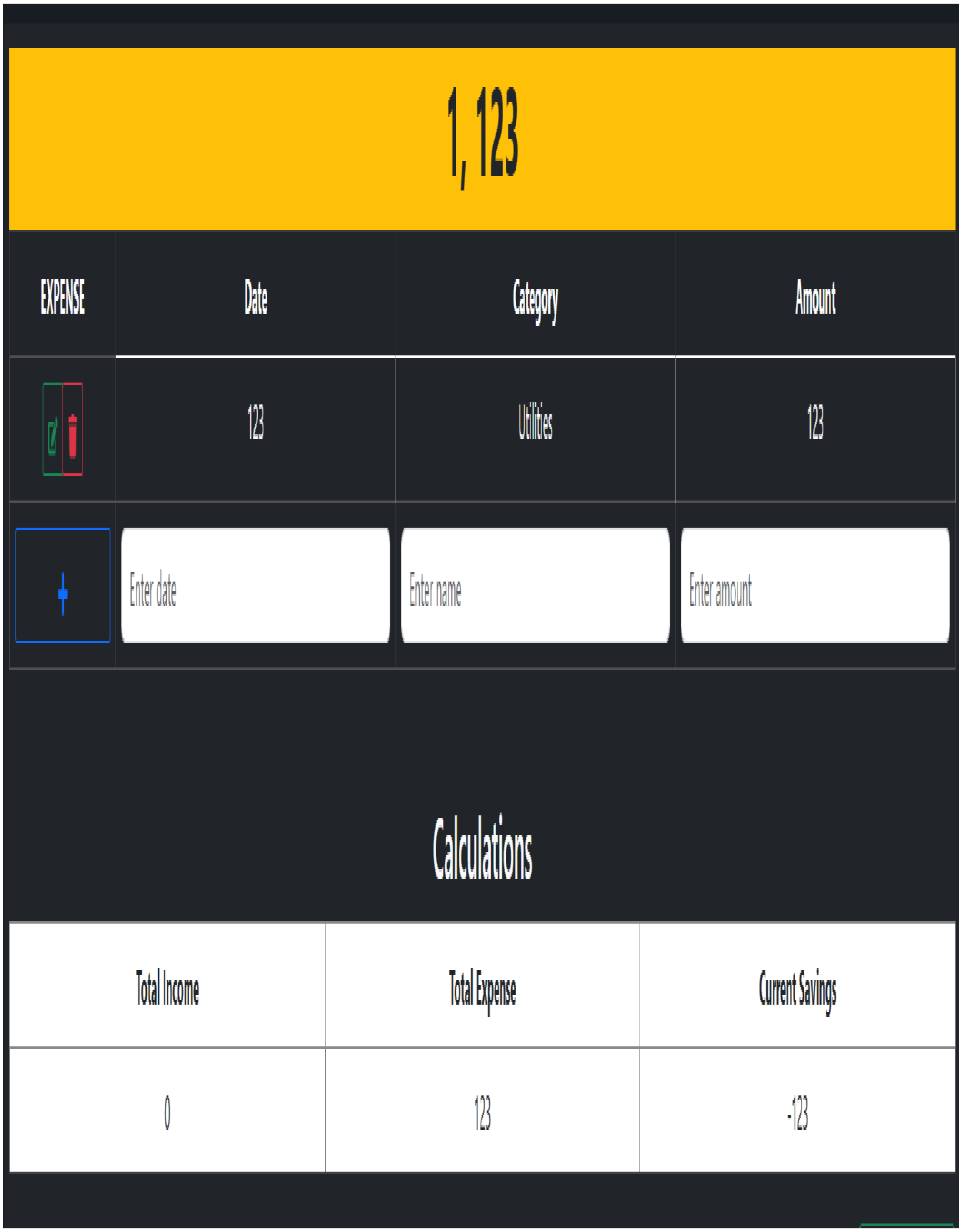
1, 1111

INCOME	Date	Category	Amount
	8	Salary	12
+	Enter date	Enter name	Enter amount

Calculations

Total Income	Total Expense	Current Savings
12	0	12

Fig:Income Management Page



1, 123

EXPENSE	Date	Category	Amount
	123	Utilities	123
+	Enter date	Enter name	Enter amount

Calculations

Total Income	Total Expense	Current Savings
0	123	-123

Fig : Expense Management Page

•

10.Testing

Black Box Testing:

Test Case 1: User Registration

Input:
```json
{
"email":
"testuser@email.com",
"password": "testpassword"
}
```
Expected Output:
```json
{
"message": "User registered successfully"
}
```

Test Case 2: Add Expense :

Input:
```json
{
"amount": 100, "category":
"Food",
"description":
"Lunch"
}
```
Expected Output:
```json
{
"message": "Expense added successfully"
}
```
Black Box Testing focuses on the application's functionality without considering the internal structure of the code. The test cases validate the expected output against the provided input for user registration and adding an expense.

White Box Testing:

Test Case 1: Update User Profile

Input:

```json
{
"email": "updatedtestuser@email.com",
"password": "updatedtestpassword"
}
```

Expected Output:

```json
{
"message": "User profile updated successfully"
}
```

Test Case 2: Edit Expense

Input:

```json
{
"amount": 150, "category":
"Food", "description": "Dinner"
}
```

Expected Output:

```json
{
"message": "Expense edited successfully"
}
```

White Box Testing focuses on the application's internal structure, logic, and code paths. The test cases validate the expected output against the provided input for updating the user profile and editing an expense.

11. Known Issues

Several common bugs or issues can arise. These may be encountered by either developers during the development phase or users when interacting with the app.

1.**Token Expiration and Reauthentication Issues :**

- Issue: If the JWT access token expires, users might be logged out unexpectedly, especially if there's no refresh token mechanism implemented.
  - **Symptoms:** Users may see a sudden logout or error when trying to access protected routes after a certain period (e.g., 1 hour).
  - **Solution:** Implement a refresh token mechanism to automatically refresh the access token without forcing the user to log in again.

2.Inconsistent Data Between Client and Server :

- Issue: Data inconsistencies between the frontend and backend can occur if proper synchronization is not maintained, particularly when using optimistic updates (i.e., updating the UI before the server responds).
  - **Symptoms:** Users may see incorrect balances, duplicate expenses, or deleted items still appearing in the list.
  - **Solution:** Ensure that all changes to the frontend UI are made only after receiving successful responses from the server. Handle errors and rollback changes if necessary.

3.Memory Leaks in React Components :

- **Issue**: Memory leaks can occur in React components, especially when asynchronous operations (such as API calls) are performed in components that unmount before the operation completes.
  - **Symptoms**: Warnings like "Can't perform a React state update on an unmounted component" in the browser console, or memory usage increasing over time.
  - **Solution**: Clean up side effects using useEffect cleanup functions or cancel asynchronous requests when a component unmounts.

4.Slow Performance Due to Inefficient MongoDB Queries

- **Issue**: Inefficient database queries in MongoDB can lead to slow performance, particularly when filtering or sorting large datasets (e.g., expenses over a long period).
  - **Symptoms**: Slow response times when fetching data, especially with large numbers of records.
  - **Solution**: Use indexes on frequently queried fields (e.g., user ID, date) and ensure efficient use of MongoDB's aggregation framework or Mongoose queries

12.Future Enhancements

While the Personal Expenses Tracker web application has been successfully developed and implemented with the essential features and functionalities, there are several areas for further enhancement and improvement to provide users with an even more comprehensive and efficient expense tracking and management system.

Potential Enhancements:
1.User Interface (UI) Enhancements: Enhance the user interface to improve the user experience and make the application more visually appealing and intuitive.

2.Expense Categorization and Analysis: Implement advanced expense categorization and analysis features, such as predictive analysis and personalized recommendations, to provide users with valuable insights and recommendations to optimize their spending habits and manage their expenses more effectively.

3.Budget Planning and Management: Develop advanced budget planning and management features to enable users to create and manage multiple budgets, set budget limits for different expense categories, and receive proactive budget notifications and alerts to help them monitor and control their expenses more effectively.

4.Integration with External Financial Services: Integrate the application with external financial services and platforms, such as online banking and financial planning tools, to enable users to automatically import and synchronize their financial transactions and data, providing them with a more seamless and integrated expense tracking and management experience.

5. Multi-Platform Support: Develop native mobile applications for iOS and Android platforms to provide users with a more convenient and accessible expense tracking and management solution,enabling them to manage their expenses on-the-go and access their financial data and reports anytime, anywhere.