**Full Stack Development with MERN**

**1. Introduction**

* **Project Title:** Wise Wallet: Your Budgeting Partner
* **Team Members:** Satvik ,Arun Karthik, Surya Sunanda Meesala , Aattreya K S

**2. Project Overview**

**Purpose:** Wise Wallet is designed to assist users in efficiently managing their finances. The primary goal of the project is to provide a user-friendly platform for tracking expenses and income, offering insightful analytics, and ensuring robust data protection.

**Features:**

**CRUD Operations for Expense and Income:**

* Users can Create new records for expenses and income, providing details such as amount, category, date, and description.
* Users can Read and view their expense and income records in a list format, with options to filter and sort the data.
* Users can Update existing records to correct or modify details as needed.
* Users can Delete records that are no longer relevant, keeping their financial data organized and up-to-date.

**Dashboard to Show Analytics:**

* The dashboard offers visual representations of financial data, such as charts and graphs, to help users understand their spending and income patterns.
* Key metrics displayed include total expenses, total income, balance, and category-wise breakdowns.
* Users can view trends over time, such as monthly spending habits, to make informed budgeting decisions.
* Users can also Filter Transactions based on metrics.

**User Data Protection Using JSON Web Token (JWT):**

Authentication: Users must register and log in to access the application. During this process, a JWT is generated and provided to the user upon successful authentication.

Authorization: The JWT is stored on the client side and sent with each request to protected endpoints, ensuring that only authenticated users can perform CRUD operations and view sensitive data.

Security: JWTs help prevent unauthorized access and ensure that user data is securely protected, maintaining the confidentiality and integrity of financial information.

**State Management Using Redux and React Query:**

**Redux:**

* Global State Management: Redux is used to manage the global state of the application, including user authentication status, expense and income records, and UI state.
* Actions and Reducers: Redux actions and reducers are implemented to handle state changes in a predictable manner, ensuring a consistent state across the application.
* Middleware: Middleware such as Redux Thunk is used for handling asynchronous actions, like API calls, providing a robust solution for side-effect management.

**React Query:**

* Data Fetching: React Query is used to fetch, cache, and update server state, providing an efficient way to manage data fetching from the backend.
* Caching: React Query caches server responses, reducing unnecessary network requests and improving performance.
* Synchronization: React Query ensures that the UI stays in sync with the server state, automatically refetching data when needed, such as when an expense or income record is added or updated.
* Error Handling: React Query provides built-in error handling for network requests, ensuring that the application can gracefully handle issues like network failures or server errors.

**3. Architecture**

 Frontend**:** Built with React, the frontend provides a dynamic and responsive user interface. Redux is used for state management, ensuring consistent behavior across the application. React Router handles navigation and routing.

 Backend**:** The backend is powered by Node.js and Express.js, providing a robust API for the frontend to interact with. Middleware for CORS and JSON parsing ensures secure and efficient data handling.

 Database**:** MongoDB, accessed through Mongoose, serves as the application's database. It stores user profiles, categories, transactions, and other relevant data. The schema is defined using Mongoose models to ensure data integrity.

**4. Setup Instructions**

* **Prerequisites:**

Technology Stack:  
MongoDB ,React ,Node Js ,Express.

External Libraries used:  
express, mongoose, cors, react, redux, axios, yup, react-query

* **Installation**

#### **1. Cloning the Repository**

First, you need to clone the repository to your local machine. Open your terminal and run the following commands:

git clone "https://github.com/GuiltyUpstairs/Personal-Finance-Tracker.git"

cd Personal-Finance-Tracker

This will download the project files and navigate into the project directory.

#### **2. Installing Dependencies**

After cloning the repository, you need to install the necessary dependencies for both the frontend and backend. Navigate to the root directory of the project and run the following commands:

# Navigate to the backend directory

cd backend

# Install backend dependencies

npm install

# Navigate to the frontend directory

cd ../frontend

# Install frontend dependencies

npm install

#### **3. Setting Up Environment Variables**

To configure the application, you need to set up environment variables. Create a .env file in the backend directory with the following content:

# MongoDB connection string

MONGODB\_URI=

# JWT Secret for authentication

JWT\_SECRET=

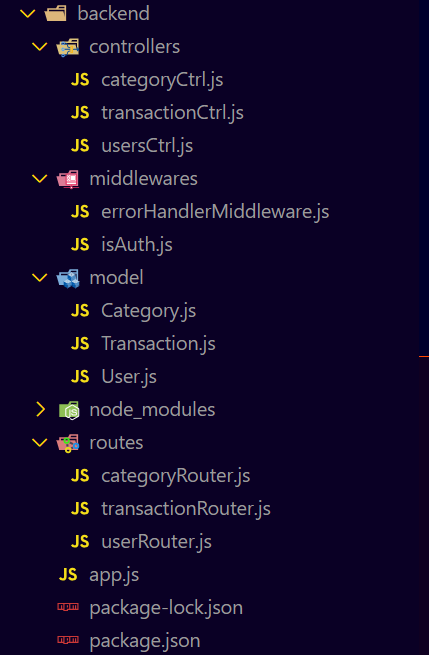
# Server port

PORT=

**5. Folder Structure**

* **Client:**

Project Directory:



**controllers:**

* **categoryCtrl.js**: Manages operations related to categories, such as creating, reading, updating, and deleting categories.
* **transactionCtrl.js**: Handles operations related to transactions, including adding, viewing, updating, and deleting transactions.
* **usersCtrl.js**: Manages user-related operations, such as registration, login, profile updates, and authentication.

**middlewares:**

* **errorHandlerMiddleware.js**: Handles errors that occur in the application, providing a consistent structure for error responses.
* **isAuth.js**: Middleware to check if a user is authenticated. It verifies the presence and validity of a JSON Web Token (JWT) in the request headers, ensuring only authenticated users can access certain routes.

**model:**

* **Category.js**: Defines the schema for the category collection, including fields like name and description.
* **Transaction.js**: Defines the schema for the transaction collection, including fields like amount, category, date, and description.
* **User.js**: Defines the schema for the user collection, including fields like username, email, password, and other relevant user information.

**routes:**

* **categoryRouter.js**: Defines routes for category operations, such as creating, reading, updating, and deleting categories.
* **transactionRouter.js**: Defines routes for transaction operations, including adding, viewing, updating, and deleting transactions.
* **userRouter.js**: Defines routes for user operations, such as registration, login, profile updates, and authentication.

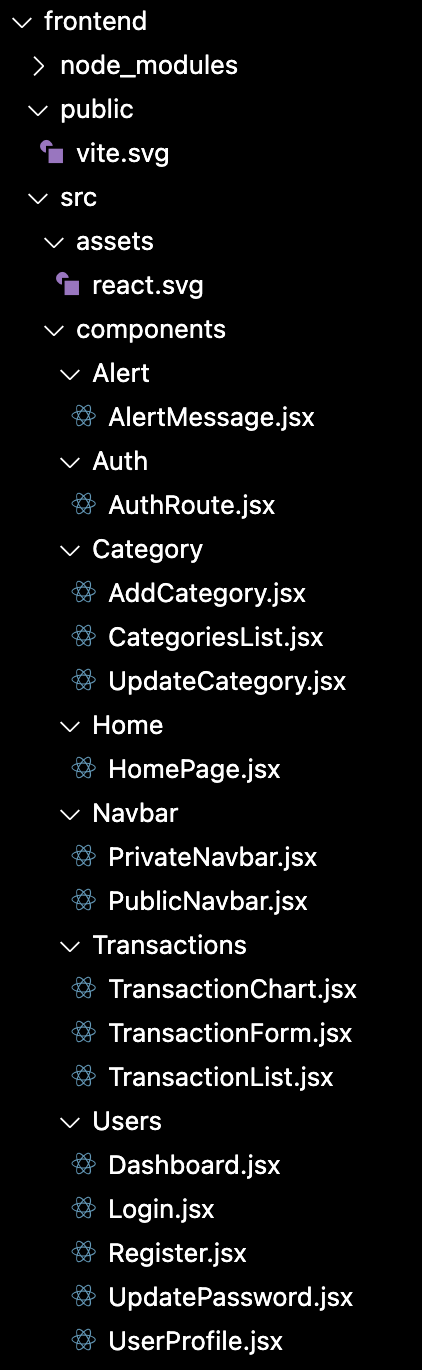
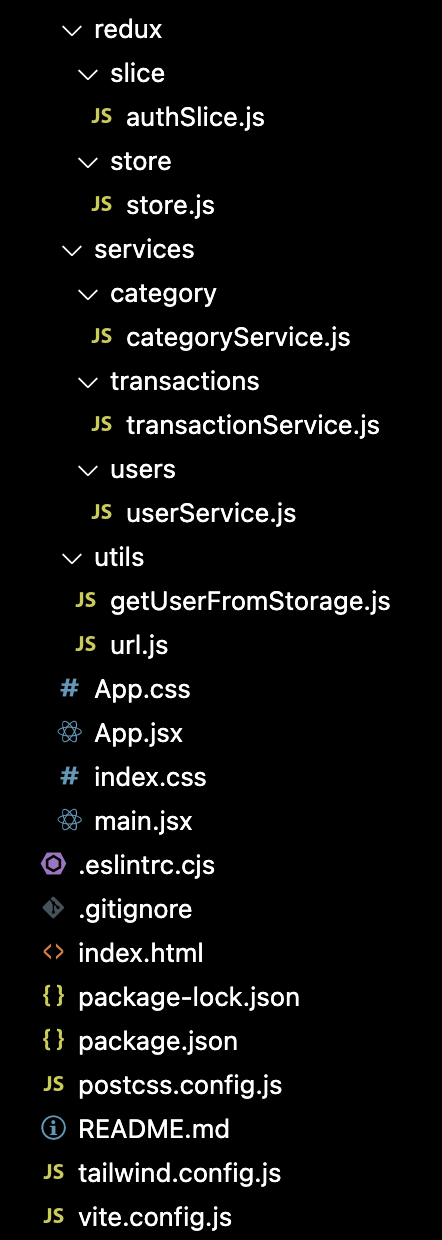
**app.js:**

* **Express Initialization**: Sets up the Express.js server.
* **Middleware Setup**: Configures middleware for CORS, JSON parsing, and logging requests.
* **Route Handling**: Integrates the defined routes for users, categories, and transactions.
* **Error Handling**: Utilizes the error handling middleware to manage errors throughout the application.
* **Server Listening**: Starts the server and listens on a specified port for incoming requests.

**Server:**

**Frontend:**

Project Directory:



**- public:** Contains static files and assets.

**- src:** Main source directory.

**- assets:** Contains asset files like images.

**- components:** Contains reusable UI components.

**- Alert:** Components for displaying alert messages.

**- Auth:** Authentication-related components.

**- Category:** Components for managing categories.

**- Home:** Homepage component.

**- Navbar:** Public and private navigation bars.

**- Transactions:** Components for managing transactions.

**- Users:** User-related components like login, registration, and profile management.

**- redux:** Redux slices and store configuration.

**- slice:** Contains individual slice files for different parts of the state.

**- store:** Configures and combines slices to create the store.

**- services:** Service files for API calls.

**- category:** Category-related API calls.

**- transactions:** Transaction-related API calls.

**- users:** User-related API calls.

**- utils:** Utility functions.

**- getUserFromStorage.js:** Utility function to get user data from local storage.

**- url.js:** Contains URL constants.

**- App.jsx:** Main application component responsible for routing and layout.

**- index.css:** Global styles.

**- main.jsx:** Entry point of the application.

**6. Running the Application**

**Starting the Backend:**

Navigate to the backend directory and start the serve

cd backend

npm start

**Starting the Frontend:**

Open a new terminal window, navigate to the frontend directory, and start the frontend server:

cd frontend

npm start

**7. API Documentation**

**User Endpoints:**

**1. Register User**

* **URL**:http://localhost:8800/api/v1/users/register
* **Method**: POST
* **Description**: Registers a new user.

**2. Login User**

* **URL**: :http://localhost:8800/api/v1/users/login
* **Method**: POST
* **Description**: Authenticates a user and returns a JWT token.

**3. Get User Profile**

* **URL**: http://localhost:8800/api/v1/users/profile
* **Method**: GET
* **Description**: Retrieves the profile of the authenticated user.

**4. Change User Password**

* **URL**: http://localhost:8800/api/v1/users/change-password
* **Method**: PUT
* **Description**: Changes the password of the authenticated user.

**5. Update User Profile**

* **URL**: http://localhost:8800/api/v1/users/update-profile
* **Method**: PUT
* **Description**: Updates the profile information of the authenticated user.

**Category Endpoints**

**1. Create Category**

* **URL**: http://localhost:8800/api/v1/categories/create
* **Method**: POST
* **Description**: Creates a new category.

**2. Get Categories**

* **URL**: http://localhost:8800/api/v1/categories/lists
* **Method**: GET
* **Description**: Retrieves a list of categories.

**3. Update Category**

* **URL**: http://localhost:8800/api/v1/categories/update/:categoryId
* **Method**: PUT
* **Description**: Updates a category by its ID.

**4. Delete Category**

* **URL**: http://localhost:8800/api/v1/categories/delete/:id
* **Method**: DELETE
* **Description**: Deletes a category by its ID.

**Transaction Endpoints**

**1. Create Transaction**

* **URL**: http://localhost:8800/api/v1/transactions/create
* **Method**: POST
* **Description**: Creates a new transaction.

**2. Get Transactions**

* **URL**: http://localhost:8800/api/v1/transactions/lists
* **Method**: GET
* **Description**: Retrieves a list of transactions.

**3. Update Transaction**

* **URL**: http://localhost:8800/api/v1/transactions/update/:id
* **Method**: PUT
* **Description**: Updates a transaction by its ID.

**4. Delete Transaction**

* **URL**: http://localhost:8800/api/v1/transactions/delete/:id
* **Method**: DELETE
* **Description**: Deletes a transaction by its ID.

**8. Authentication**

#### **Method: JSON Web Tokens (JWT)**

The Wise Wallet application uses JSON Web Tokens (JWT) for authentication and authorization. This method ensures secure communication between the client and the server by verifying the identity of users and controlling their access to protected resources.

#### **Workflow:**

1 **User Registration and Login:**

* **Registration:**
  + When a new user registers, they provide their credentials (e.g., username, email, password).
  + The server hashes the password and stores the user details in the database.
  + A JWT token is generated and sent to the client if registration is successful.
* **Login:**
  + An existing user logs in by submitting their email and password.
  + The server verifies the credentials.
  + Upon successful authentication, a JWT token is generated and sent to the client.

2 **JWT Token Generation:**

* Upon successful authentication (either registration or login), the server generates a JWT token.
* This token contains encoded user information and a signature, ensuring its integrity and authenticity.
* The JWT token includes an expiration time to enhance security by limiting the token's lifespan.

3 **Storing the Token on the Client-Side:**

* The client (typically the browser) stores the JWT token, usually in local storage or a cookie.
* This token is sent with each subsequent request to access protected routes.

4 **Sending the Token with Requests:**

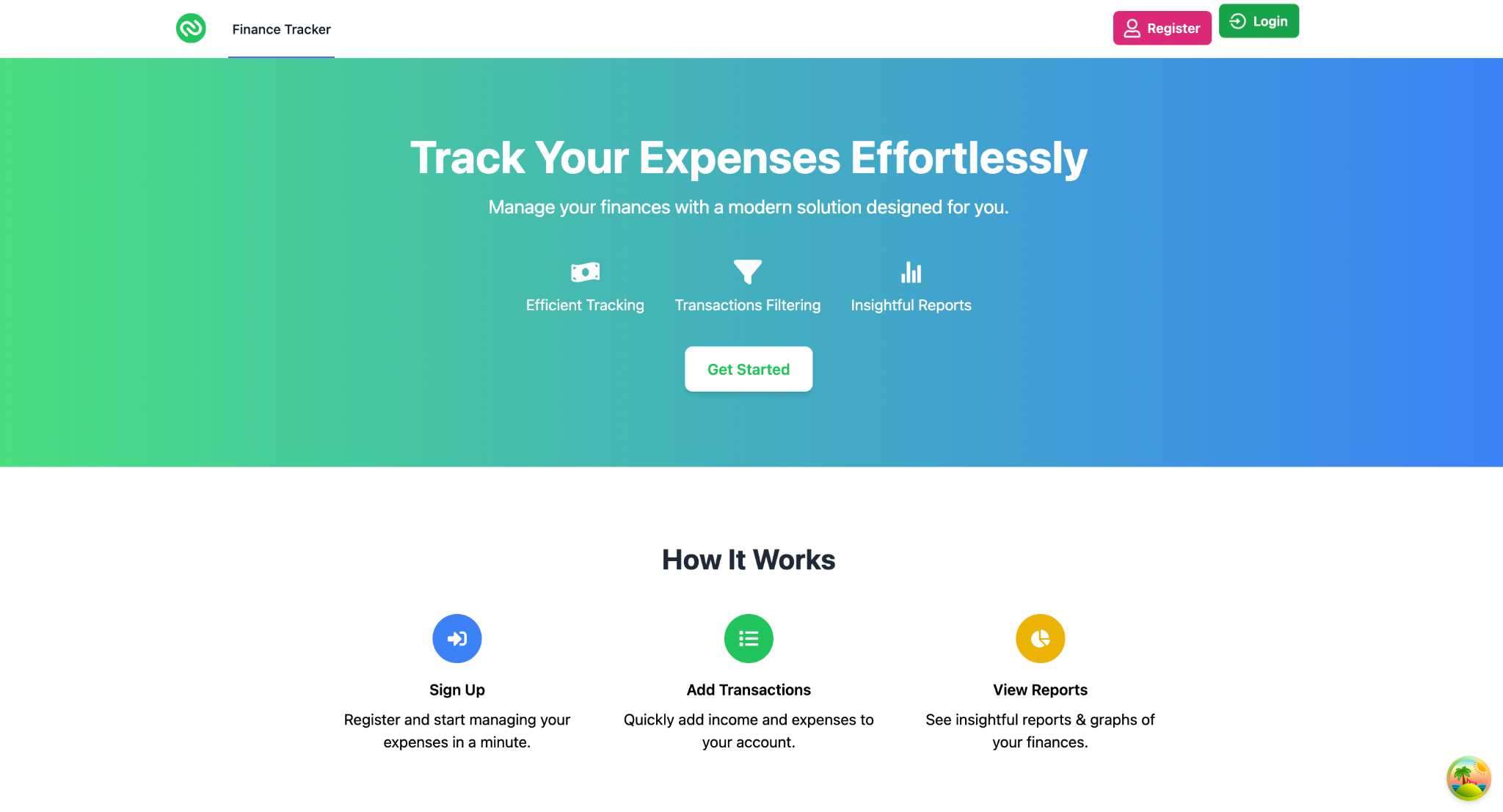
* For requests to protected routes (e.g., adding a transaction, viewing user profile), the client includes the JWT token in the HTTP headers.

5 Token **Verification on the Server-Side:**

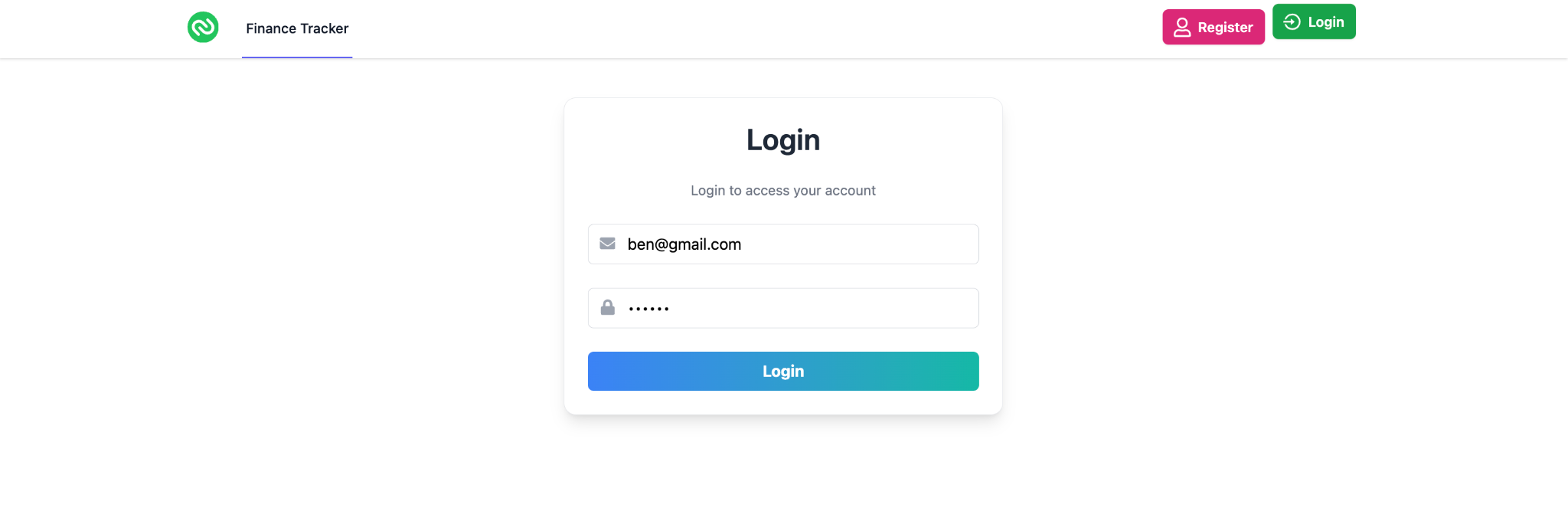
* The server uses middleware to intercept requests to protected routes.
* This middleware extracts the token from the request headers.
* The token is then verified to ensure it is valid and has not expired.
* If the token is valid, the request proceeds, and the user gains access to the protected resource.
* If the token is invalid or expired, the server denies access and returns an appropriate error response.

**9. User Interface**

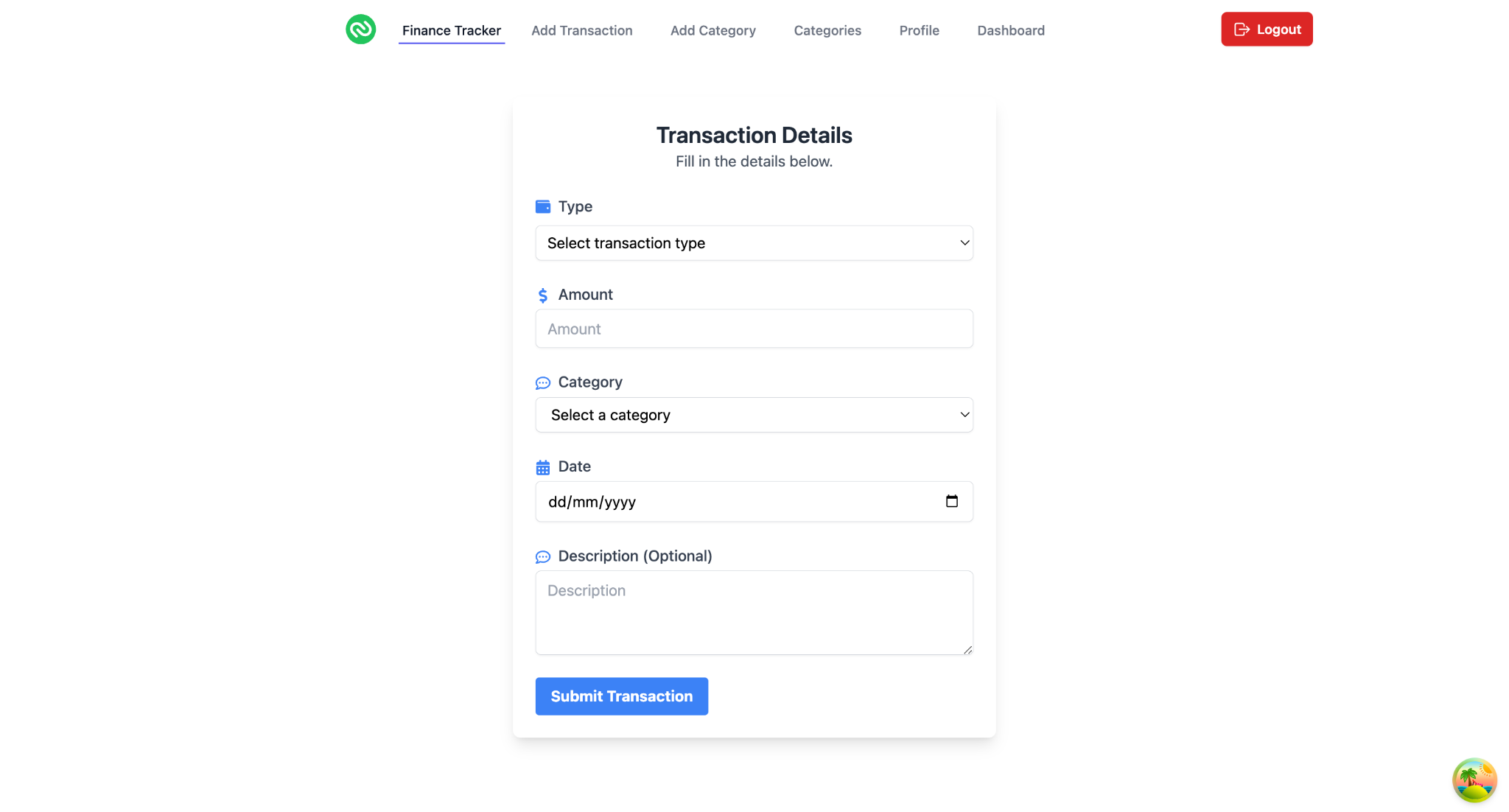
LANDING PAGE:

****

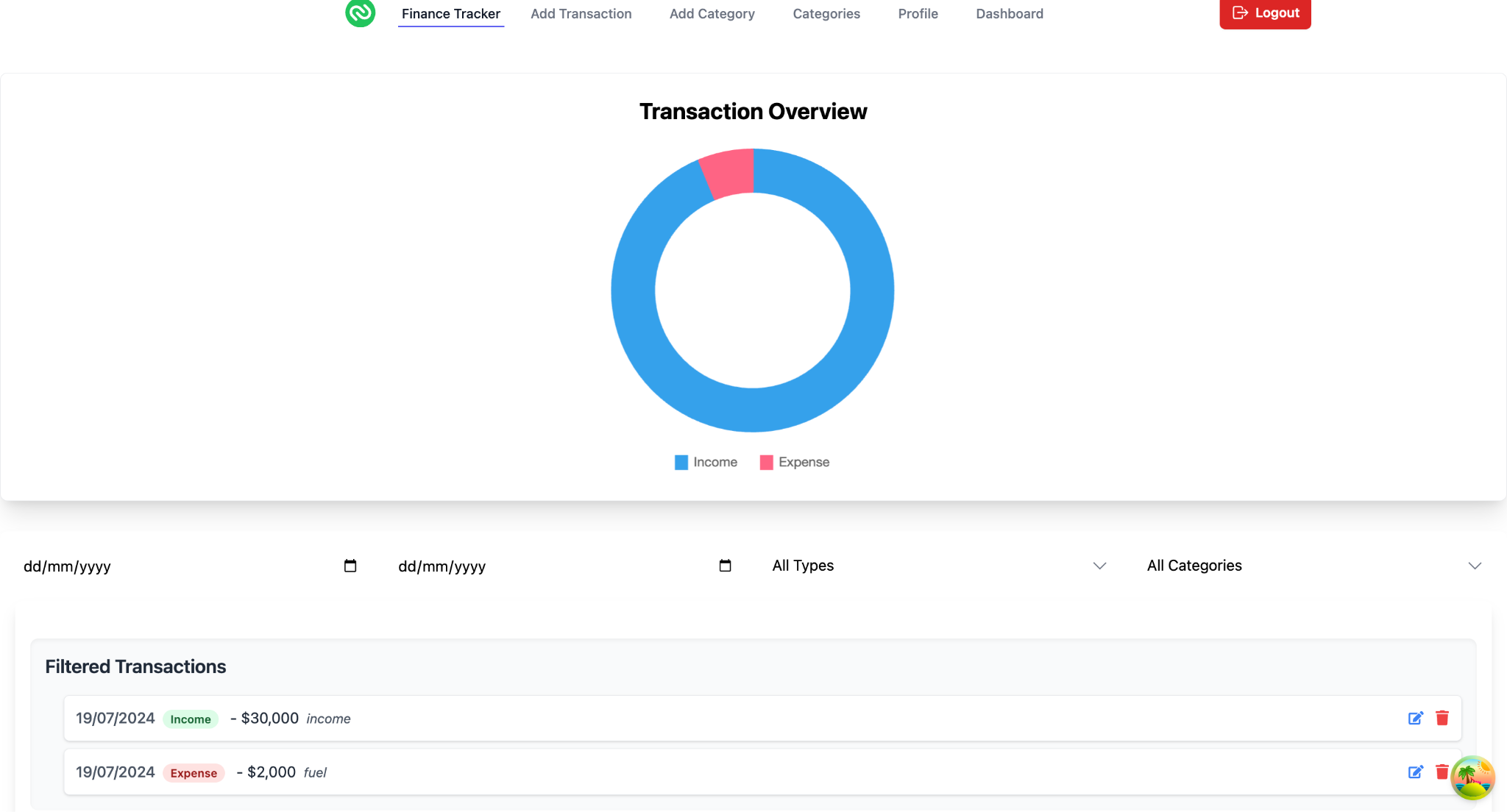
LOGIN PAGE:

****

ADD TRANSACTION PAGE:

****

DASHBOARD PAGE

****

**10. Testing**

#### **Strategy: Manual and Automated Testing**

We employ a combination of manual and automated testing to ensure the reliability and robustness of the Wise Wallet application. This approach covers both frontend and backend components, verifying functionality, performance, and security.

#### **Tools Used:**

* **Jest**: For unit testing of frontend and backend code.
* **Postman**: For testing API endpoints manually.

#### **Description:**

Test cases are designed to cover various scenarios, including user interactions, data validation, and integration between the frontend and backend. The testing process involves checking for expected outcomes, performance benchmarks, and security vulnerabilities.

**11. Demo**

* Provide screenshots or a link to a demo (if available) to showcase the application.

**12. Known Issues**

* Inconsistent display of currency symbols across different components.
* Slow performance on the dashboard with large datasets.

**13. Future Enhancements**

* Implement advanced financial analytics and forecasting features.
* Add multi-currency support.
* Integrate with third-party financial services for automatic transaction import.
* Enhance mobile responsiveness and develop a native mobile application.