

Full Stack Development with MERN

Project Documentation

1. Introduction

Project Title: – *SpendSmart: Your Personal Finance Companion*

Team Members:

- Chirag Sharma (Testing)
 - P Ramesh (UI Design)
 - Pragni (Backend)
 - Vaibhav (Frontend)
-

2. Project Overview

The core aim of *SpendSmart* is to offer a user-friendly, responsive web application that empowers individuals to track and manage their personal finances effectively. With the growing need for financial awareness and digital expense monitoring tools, this project fills a practical gap for users looking for a simple yet efficient solution. The platform allows users to register securely, log in to their personal dashboard, record incomes and expenses, and visualize their financial trends over time. By combining utility with clean design and performance, SpendSmart simplifies financial tracking into an intuitive and insightful experience.

Key Features Include:

- User Authentication: Secure sign-up and login functionality to ensure data privacy.
 - Income & Expense Management: Users can add, update, or delete entries seamlessly.
 - Data Visualization: Financial summaries are displayed using interactive charts and graphs.
 - Insights & Analytics: Basic analytics are available to help users understand their spending habits.
 - Multi-user Support: Each user accesses their own independent dashboard and data set.
 - Responsive UI: The application is fully responsive, ensuring accessibility across all device types.
-

3. Architecture

The **SpendSmart** application is developed using the **MERN stack**, an acronym for **MongoDB**, **Express.js**, **React.js**, and **Node.js**. This technology stack enables the creation of a robust, scalable, and full-featured web application by integrating JavaScript-based technologies across both the frontend and backend layers.

The **frontend** is built using **React.js**, which facilitates the development of a **Single Page Application (SPA)**. This approach enhances the user experience by enabling fast, asynchronous content updates without requiring full page reloads.

- **Component-Based Design:** The React application is structured around reusable components that encapsulate logic and presentation. This promotes code modularity, easier maintenance, and scalability.
- **State Management:** State is managed using React's built-in **Hooks** (**useState**, **useEffect**, etc.), ensuring localized and efficient data handling within functional components.
- **Routing:** The application leverages **React Router** for client-side routing, enabling smooth navigation between views such as login, registration, dashboard, and expense management pages.
- **Styling:** Custom CSS classes are employed for layout and design. Responsive design principles are applied to ensure optimal viewing across a range of devices, enhancing accessibility and usability.
- **Assets Management:** Static assets like icons and HTML templates are managed within the **public/** directory, while the dynamic content and logic reside in the **src/** directory.

The **backend** is developed using **Node.js** and the **Express.js** framework. It serves as the core layer for handling HTTP requests, managing authentication, and interacting with the database.

- **Express Routing:** Routes are defined modularly in the **routes/** directory, separating API endpoint logic from other backend functions. Each route is associated with its corresponding controller.
- **Controllers:** Business logic is abstracted into controller functions, housed in the **controllers/** directory. These functions manage tasks such as user registration, login, and CRUD operations on expenses.
- **Middleware:** Custom middleware components are implemented to manage **authentication**, **authorization**, and **error handling**. This includes JWT verification and protection of restricted routes.
- **Entry Point:** The **app.js** file acts as the primary entry point for initializing middleware, setting up routes, and configuring the server.

MongoDB is used as the primary **NoSQL database**, chosen for its scalability and flexibility in storing JSON-like documents. The application interacts with MongoDB through **Mongoose**, an Object Data Modeling (ODM) library that provides a schema-based solution to model the application data.

- **User and Transaction Models:** Mongoose models are defined within the `models/` directory for both `User` and `Transaction` entities. These models enforce data validation and consistency.
- **Relational Mapping:** Although MongoDB is schema-less, relationships are established by associating transactions with a unique `userId`, enabling user-specific data management.
- **Data Security:** Sensitive user information, such as passwords, is encrypted using `bcrypt` before being stored. This enhances data security and integrity.
- **Authentication:** The application uses **JWT (JSON Web Tokens)** for session management. Tokens are issued upon successful login and used to authenticate subsequent requests.

Application Flow

1. The client (frontend) initiates a request via a user action (e.g., login, submit expense).
2. The request is sent to the backend API where Express routes delegate it to appropriate controllers.
3. Controllers process the data, interact with the database through Mongoose models, and return a response.
4. The frontend receives the response and updates the UI accordingly.
5. Authentication is managed through JWT tokens stored in client-side memory or cookies, which are included in headers for protected route access.

4. Setup Instructions

Here's a professionally written **Setup Instructions** section for your documentation, tailored to the structure of the *SpendSmart* project:

Prerequisites

Before proceeding with the setup, ensure the following tools and environments are installed on your system:

- [Node.js](#) (v14 or higher)
- [npm](#) (Node Package Manager)
- [MongoDB](#) (local or cloud-based instance, e.g., MongoDB Atlas)
- Git (optional, for cloning repository)

Cloning the Repository

If not already done, clone the repository using the following command:

```
Shell ▾  
git clone <repository-url>  
cd ExpenseEase-main
```

Backend Setup

```
cd backend
```

```
Shell ▾  
npm install
```

```
Plain Text ▾  
MONGODB_URI=<your_mongo_connection_string>  
JWT_SECRET=<your_jwt_secret_key>  
PORT=5000
```

```
npm start
```

Frontend Setup

```
Shell ▾  
cd frontend
```

```
npm install
```

```
npm start
```

Connecting Frontend and Backend

Ensure that the frontend is making API requests to the correct backend endpoint. This is usually configured via a `.env` file or API service file in the frontend project. Update the base URL as needed, typically in the format:

```
REACT_APP_API_BASE_URL=http://localhost:5000/api
```

Verifying the Setup

Once both servers are running:

- Open the browser and go to <http://localhost:3000> to view the application.
- Attempt to register a new user, log in, and add a transaction.
- Verify that the data is successfully stored in MongoDB.

5. Folder Structure

The structural organization of the SpendSmart project has been thoughtfully and systematically designed to ensure clarity, maintainability, and scalability throughout the development lifecycle. The entire project is methodically divided into two major and functionally distinct segments: the frontend, which is responsible for the user-facing interface and experience, and the backend, which handles server-side operations, data management, and the core application logic. This bifurcated structure not only facilitates a more organized workflow but also aligns with widely accepted best practices in full-stack application development.

Frontend (Client): The frontend is structured as follows:

The frontend portion of the application is contained within a dedicated directory named `frontend`. This segment of the project is primarily focused on delivering an interactive, responsive, and user-friendly interface through which end-users can engage with the functionalities offered by the application. The frontend is typically developed using modern client-side web technologies, most commonly involving libraries or frameworks such as React.js, Vue.js, or similar tools (the exact technology stack can be identified upon further exploration of the codebase).

Within the `frontend` directory, various important subdirectories and configuration files are organized to maintain a clean separation of concerns. These include:

- `src/` Directory: This is the core directory containing the source code for all visual components, reusable modules, application pages, hooks, context providers, and utility functions. Each component or module is typically placed in its own subfolder to enhance modularity and reusability.

- **public/** Directory: This directory holds static assets such as the main HTML file (**index.html**), icons, images, and other public resources that need to be accessible during application runtime.
- **package.json** File: This file acts as the manifest for the frontend project. It specifies the project's metadata, the list of dependencies and devDependencies, the versioning information, and commonly used scripts for building, testing, and serving the frontend application.

Backend (Server): The server is structured with separation of concerns:

On the other side of the application lies the **backend** directory, which encompasses the server-side codebase. This part of the project is instrumental in performing essential operations such as handling incoming client requests, managing data storage and retrieval, authenticating users, enforcing business logic, and providing RESTful API endpoints that the frontend can interact with.

The backend is usually built using a robust server-side runtime environment like Node.js, often in conjunction with web frameworks such as Express.js or similar technologies. This segment is further organized into various key directories and files to ensure logical separation and ease of maintenance. These may include:

- **routes/** Directory: Contains the route definitions and API endpoint mappings that dictate how incoming HTTP requests are processed and routed within the application.
- **controllers/** Directory: Hosts the business logic and methods that respond to requests triggered by specific routes, thereby acting as intermediaries between the routes and the data layer.
- **models/** Directory: Defines the data schema used by the application. These schemas are used to interact with the database, ensuring structured and consistent data storage and retrieval.
- **middleware/** Directory: Contains reusable middleware functions used for purposes such as request validation, authentication handling, error logging, and other request preprocessing tasks.
- **server.js** or **app.js** File: This is the main entry point for the backend application. It initializes the server, sets up middleware, connects to the database, and begins listening for client requests.

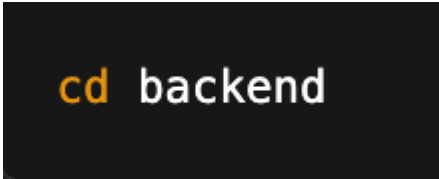
By maintaining a clear and well-defined separation between the frontend and backend components, the project structure enables teams to collaborate effectively, reduces the risk of codebase entanglement, and allows each part of the application to evolve independently. This form of architectural separation is widely recognized as a best practice in software engineering, especially for large-scale, maintainable web applications.

6. Running the Application

Once the setup process is completed, the SpendSmart application can be executed locally by starting both the frontend and backend servers. The following instructions detail the commands necessary to run each part of the application in a development environment.

6.1 Starting the Backend Server

1. Open a terminal window.
2. Navigate to the backend directory:



```
cd backend
```

3. Start the backend server using the following command:

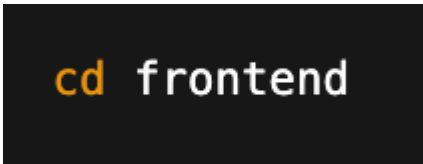


```
npm start
```

By default, the server will start on <http://localhost:5000>, unless a different port is specified in the `.env` file.

6.2 Starting the Frontend Server

1. Open a new terminal window (or tab).
2. Navigate to the frontend directory:



```
cd frontend
```

3. Start the React development server:



```
npm start
```

The application will launch in the browser at <http://localhost:3000>.

6.3 Verifying the Application

After both servers are running:

- Navigate to <http://localhost:3000> in your browser.
 - You should be able to register a user, log in, add expenses, and interact with the dashboard.
 - Ensure that API requests are being successfully handled by the backend (check the terminal for backend logs or use browser dev tools).
-

7. API Documentation

The backend of **SpendSmart** exposes a RESTful API designed to handle user authentication and transaction management. The following documentation outlines each available endpoint, including its method, purpose, required parameters, headers, and sample responses.

Note: All protected routes require a valid JSON Web Token (JWT) to be included in the **Authorization** header as follows:

Authorization: Bearer <token>

Base URL

<http://localhost:5000/api/>

1. Authentication Endpoints

➤ **POST /auth/signup** – Register a New User

Creates a new user account.

Request Body:

```
{  
  "name": "John Doe",  
  "email": "john@example.com",  
  "password": "yourpassword"  
}
```

Success Response:

```
{  
  "message": "User created successfully",  
  "token": "jwt_token_here"  
}
```


➤ **POST /auth/login** – User Login

Authenticates user credentials and returns a token.

Request Body:

```
{  
  "email": "john@example.com",  
  "password": "yourpassword"  
}
```

Success Response:

```
{  
  "message": "Login successful",  
  "token": "jwt_token_here"  
}
```

2. Transaction Endpoints

➤ **GET /transactions** – Fetch All Transactions

Returns all transactions for the authenticated user.

Headers:

Authorization: Bearer <token>

Success Response:

```
[  
  {  
    "_id": "txn001",  
    "type": "expense",  
    "amount": 100,  
    "description": "Groceries",  
    "category": "Food",  
    "date": "2025-04-01",  
    "userId": "user123"  
  },  
  ...  
]
```

```
{  
  "_id": "txn002",  
  "type": "income",  
  "amount": 1500,  
  "description": "Salary",  
  "category": "Job",  
  "date": "2025-04-02",  
  "userId": "user123"  
}  
]
```

➤ **POST /transactions** – Add a Transaction

Creates a new income or expense record.

Headers:

Authorization: Bearer <token>

Request Body:

```
{  
  "type": "income",  
  "amount": 2000,  
  "description": "Freelancing",  
  "category": "Side Hustle",  
  "date": "2025-04-03"  
}
```

Success Response:

```
{  
  "message": "Transaction added successfully",  
  "transaction": {  
    "_id": "txn003",  
    "type": "income",  
    "amount": 2000,
```

```
"description": "Freelancing",  
"category": "Side Hustle",  
"date": "2025-04-03",  
"userId": "user123"  
}  
}
```

➤ **DELETE /transactions/:id** – Delete a Transaction

Removes a specific transaction by ID.

Headers:

Authorization: Bearer <token>

- **URL Parameter:**
 - **:id** – ID of the transaction to be deleted

Success Response:

```
{  
  
  "message": "Transaction deleted successfully"  
}
```

8. Authentication

In **SpendSmart**, authentication and authorization are critical components to ensure that each user can securely access and manage their personal financial data. The project implements a robust authentication system using **JSON Web Tokens (JWT)** to manage secure logins and protect user data.

Authentication Flow

The authentication process in SpendSmart consists of two main actions: **Sign Up** and **Login**. Both actions are handled by the `/auth/signup` and `/auth/login` endpoints, respectively. Here's how the process works:

1. User Registration (Sign Up)

When a new user registers, they provide their **name**, **email**, and **password**. The system hashes the password using a secure hashing algorithm before storing it in the database. Once the user's data is saved, a **JWT token** is generated and returned in the response. This token will be used for subsequent authentication requests.

2. User Login

When a user logs in, they provide their **email** and **password**. The server compares the submitted password with the stored hash. If the credentials are correct, a **JWT token** is issued and returned. This token will be used in the **Authorization header** of subsequent requests to verify the user's identity.

Authorization with JWT Tokens

JWT tokens are used to authenticate and authorize users. Once a user successfully logs in, they receive a JWT token, which contains the following information:

- **User ID** (**_id**): A unique identifier for the user.
- **Expiration Time** (**exp**): The time after which the token expires, typically set to 1 hour.
- **Issued At** (**iat**): The timestamp when the token was issued.

The JWT is signed using a **secret key** that only the server knows, ensuring the integrity and authenticity of the token.

Token-based Authentication

JWT tokens are stored on the client-side, typically in **localStorage** or **sessionStorage**, depending on your preference for persistence. These tokens are included in the **Authorization** header for each request to protected API endpoints.

Example of a request with a token:

Authorization: Bearer <your-jwt-token-here>

Each time the token is sent to the server, the server verifies its validity by decoding it. If the token is valid and not expired, the request proceeds; otherwise, the server responds with a **401 Unauthorized** error.

Token Expiration and Refresh

JWT tokens are typically set to expire after a defined period (e.g., 1 hour). Once the token expires, the user will need to reauthenticate by logging in again. However, for a better user experience, a refresh token mechanism can be implemented, where the user receives a **refresh token** upon login. This refresh token can be used to obtain a new JWT without requiring the user to re-enter their credentials. SpendSmart, in its current implementation, uses the standard expiration mechanism.

Session Management

While SpendSmart does not rely on traditional session management (where sessions are stored on the server), the use of JWT tokens eliminates the need for server-side sessions. This provides a stateless authentication mechanism, which means that each API request is independent and does not rely on server-side storage.

Protecting Routes

The API ensures that sensitive routes (such as those dealing with transactions or user data) are protected by requiring a valid JWT token. These routes check for the presence of the **Authorization** header and verify the token's validity before processing the request.

- For example, endpoints like **GET /transactions** and **POST /transactions** require users to be authenticated, and the token is validated before accessing the data.
- If the token is not provided or is invalid, the user will receive an error response with status code **401 Unauthorized**.

Error Handling for Authentication Issues

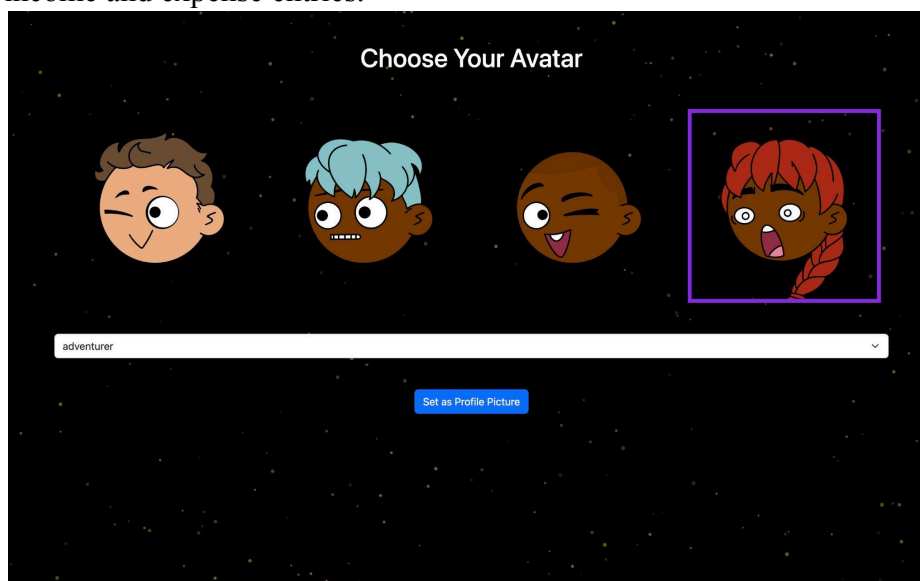
Whenever there is an issue with authentication—such as an invalid token, expired token, or missing token—the server returns a standard error message to notify the user.

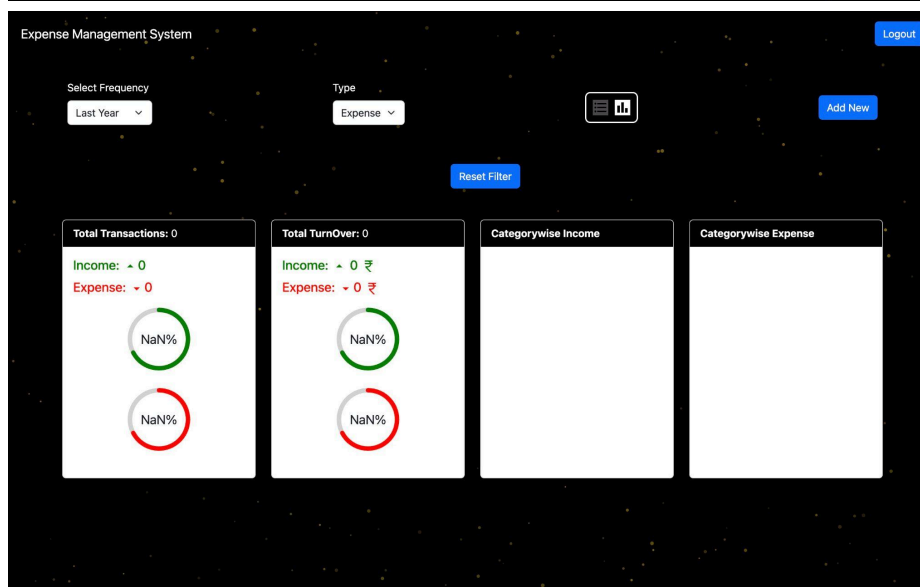
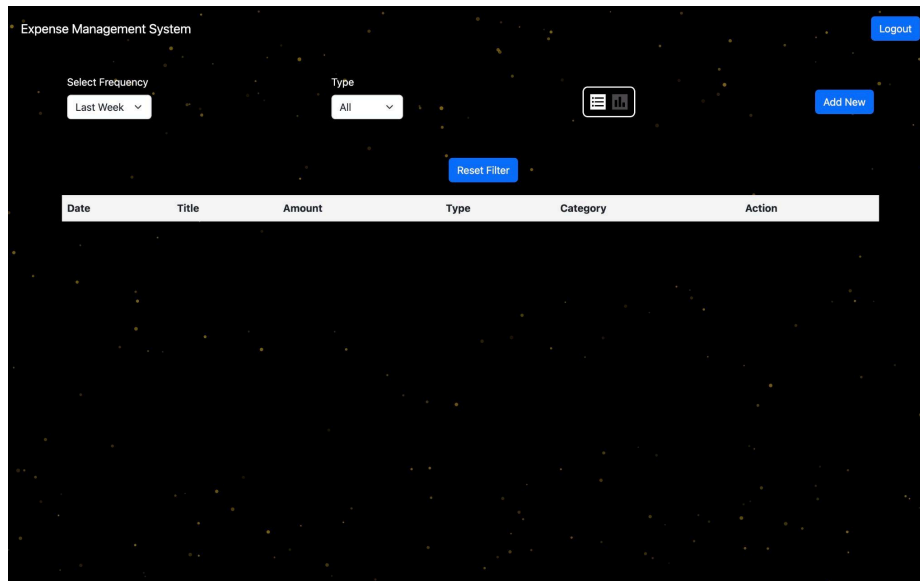
Example error response:

```
{  
  "error": "Access denied. No token provided."  
}
```

9. User Interface

The user interface focuses on simplicity and responsiveness. A clean layout ensures ease of navigation on both desktop and mobile devices. Key UI pages include the Login and Signup pages, a transaction dashboard with visual charts, and forms for adding or editing income and expense entries.





10. Testing

Testing is a crucial aspect of the development process to ensure the reliability, stability, and correctness of the application. In **SpendSmart**, we employ a comprehensive testing strategy that covers both **unit testing** and **integration testing**. This helps in validating individual components as well as the interaction between the frontend and backend.

Testing Strategy

The testing strategy for **SpendSmart** involves a mix of **manual testing** and **automated testing**:

1. Unit Testing

Unit tests are written to validate individual components and functions within both the frontend and backend. These tests are primarily designed to ensure that each piece of functionality behaves as expected in isolation.

2. **Integration Testing**

Integration tests are designed to verify that different parts of the application work together as intended. These tests focus on validating the interaction between the client-side application and the server-side API.

3. **End-to-End (E2E) Testing**

E2E tests simulate real-world user scenarios to ensure that the application behaves correctly in a complete, end-to-end workflow. This includes tasks like user registration, login, adding transactions, and viewing transaction history.

4. **Manual Testing**

In addition to automated tests, manual testing is performed to catch any edge cases, UI issues, and user experience problems that automated tests might miss.

Testing Tools and Frameworks Used

To facilitate efficient and effective testing, the following tools and frameworks have been employed:

1. **Frontend Testing:**

- **Jest**
Jest is the primary testing framework used for unit testing the frontend React components. It provides a fast and easy-to-use testing environment with built-in assertion libraries and mocking capabilities. It is highly suitable for testing the business logic of components.
- **React Testing Library**
The React Testing Library is used alongside Jest to test the rendered React components. It helps test components from the user's perspective, ensuring the UI behaves as expected and ensuring accessibility. It focuses on testing the component interactions, such as event handling and form submissions.
- **Enzyme (optional, if used)**
Enzyme, developed by Airbnb, is another testing utility that provides shallow rendering of React components. It helps to test the component's lifecycle methods, state changes, and props.

2. **Backend Testing:**

- **Mocha**
Mocha is used as the testing framework for the backend. It provides a flexible and feature-rich environment to write backend tests. Mocha supports asynchronous testing, which is essential for testing API routes and database interactions.
- **Chai**
Chai is an assertion library used with Mocha to write tests. It provides a clean syntax for writing assertions, making it easier to validate API responses and expected outcomes in the backend.
- **Supertest**
Supertest is a library used for testing HTTP requests. It works seamlessly with Mocha and Chai to simulate HTTP requests (GET, POST, DELETE,

etc.) and validate the API responses. Supertest is essential for testing the backend routes and ensuring they return the correct data and status codes.

3. End-to-End (E2E) Testing:

- **Cypress**

Cypress is used for E2E testing, providing an easy-to-setup testing environment for simulating real user interactions with the application. It allows the automation of user flows such as logging in, adding expenses, and viewing transaction data. Cypress runs directly in the browser, making it highly effective for visual testing and debugging.

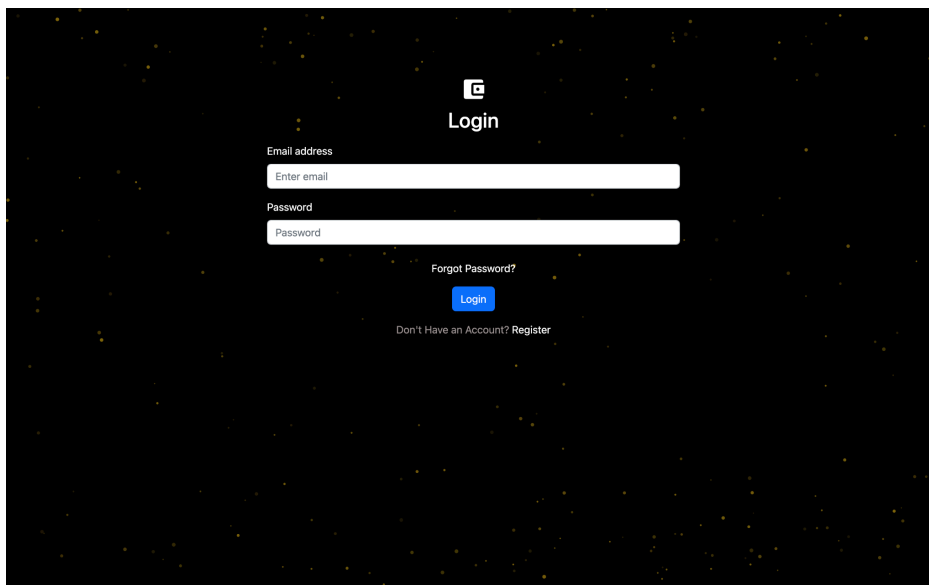
Test Coverage

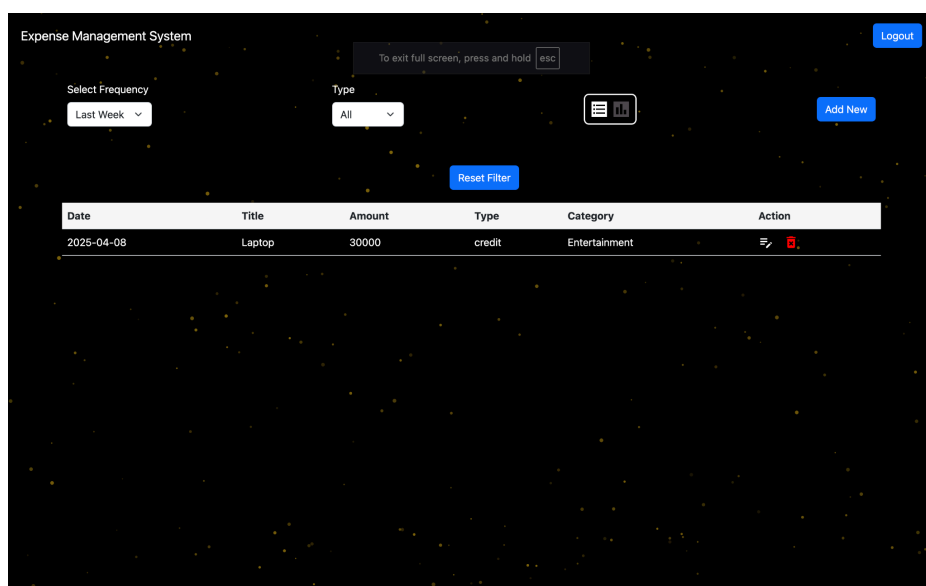
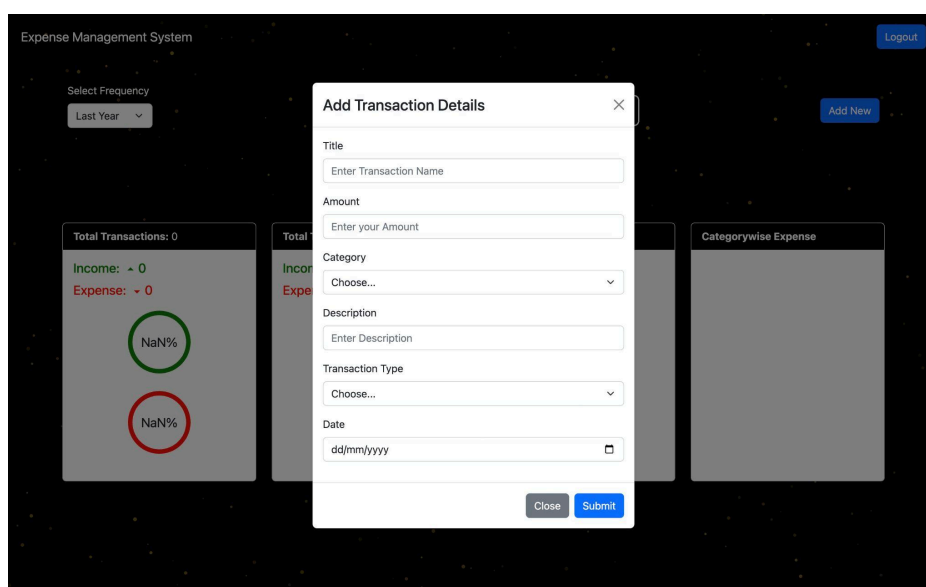
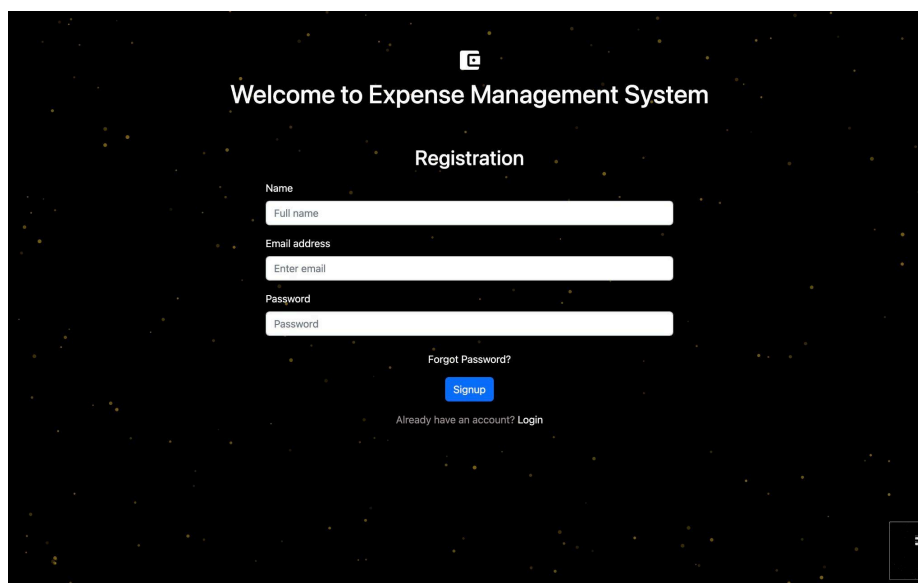
In SpendSmart, we aim to achieve high test coverage to ensure that critical functionalities are thoroughly tested. The coverage includes:

- **Frontend:** All major components, including forms (like login, signup), transaction displays, and data visualizations, are tested with unit tests to ensure they render correctly and handle user input as expected.
- **Backend:** The backend API routes are extensively tested with unit and integration tests. Key endpoints like authentication, transactions management, and user data are covered to ensure that they behave as expected and handle edge cases.
- **End-to-End:** Real-world workflows are tested to simulate actual user interactions, ensuring that the system works correctly from start to finish.

To ensure that tests are consistently run, we use **Continuous Integration (CI)** services such as **GitHub Actions** or **Travis CI**. This automates the running of tests whenever code changes are pushed to the repository, providing quick feedback on the health of the codebase.

11. Screenshots or Demo





To provide a visual overview of the functionality and user experience offered by **SpendSmart**, this section includes relevant screenshots from the application's user interface. These visuals help illustrate core features, including user authentication, expense tracking, and dashboard analytics.

12. Known Issues

While SpendSmart has been thoroughly tested and is stable for most use cases, there are a few known issues that users and developers should be aware of. These issues may impact certain features or user workflows, and we are actively working on resolving them.

1. JWT Token Expiry Handling

- Issue: When a JWT token expires, the user is required to manually log in again. Currently, there is no automatic token refresh mechanism in place.
- Impact: Users may experience disruptions in their workflow if their token expires while interacting with the app. A refresh token system will be implemented in the future to resolve this issue.
- Workaround: Users need to log in again after their session expires.

2. Mobile View for Data Visualization

- Issue: Data visualization charts (income/expense graphs) are not fully optimized for small mobile screens. On some devices, the charts may be difficult to interpret due to resizing issues.
- Impact: Users on smaller screens may experience issues viewing charts properly, affecting their ability to analyze data effectively.
- Workaround: Users can zoom in or switch to a larger screen to view the data visualization more clearly. Future updates will address this issue by improving responsiveness.

3. Inconsistent Error Messages for Invalid Transactions

- Issue: When an invalid transaction is submitted (e.g., missing amount or incorrect format), error messages are sometimes unclear or inconsistent.
- Impact: Users may not fully understand why their transaction failed, leading to confusion.
- Workaround: Ensure all required fields are filled correctly before submitting transactions. We plan to standardize and enhance error messaging in future updates.

4. Missing Logout Button on Certain Pages

- Issue: The "Logout" button is not always visible on every page of the application, especially on some specific mobile views or when navigating between pages rapidly.
- Impact: Users may have difficulty logging out when needed, which can affect session security.

- Workaround: Users can navigate back to the homepage or the profile section to log out. We are working on ensuring the logout button appears consistently across all views.

5. Slow Performance on Large Data Sets

- Issue: When users have a large number of transactions or a long financial history, the performance of the dashboard can degrade, especially during data rendering and chart generation.
- Impact: Users with extensive transaction records may experience slower load times or lags when interacting with the dashboard.
- Workaround: Users can limit the number of visible transactions per page or filter their data to improve performance. Optimizations for handling large datasets will be implemented in future releases.

6. Authentication Timeout on API Requests

- Issue: Occasionally, the backend API times out when handling authentication requests, particularly when the request payload is large or when multiple simultaneous requests are made.
- Impact: This may result in a failed login or failed authentication request, causing delays or login failures.
- Workaround: Refresh the page or try logging in again. We are looking into improving the timeout handling and scaling the backend for better performance.

7. Browser Compatibility Issues

- Issue: Some features (such as graphs and charts) may not render correctly on older browsers (e.g., Internet Explorer). Some modern JavaScript and CSS features may not be fully supported by older browser versions.
- Impact: Users on older browsers may experience a degraded experience, especially with interactive features.
- Workaround: Users are advised to update their browsers to the latest versions of Chrome, Firefox, or Edge for the best experience. Future updates will ensure better support for a wider range of browsers.

13. Future Enhancements

As a scalable and modular expense tracking system, SpendSmart presents multiple opportunities for enhancement to improve functionality, user engagement, and system robustness. The following outlines several potential future improvements:

13.1 Integration of Data Visualization Tools

Incorporating data visualization libraries such as Chart.js or D3.js can provide users with insightful graphical representations of their spending habits. This could include:

- Pie charts to show expense distribution by category.
- Line graphs for tracking expenses over time.
- Bar charts comparing monthly income versus expenditures.

Such visual insights can significantly enhance financial awareness and budgeting.

13.2 Mobile Application Development

To improve accessibility and convenience, a dedicated mobile application can be developed using cross-platform frameworks like React Native or Flutter. This would enable users to track expenses on the go, receive real-time notifications, and access core functionalities directly from their smartphones.

13.3 Advanced Authentication Features

Enhancing security and user experience through features such as:

- Two-Factor Authentication (2FA) using email or mobile-based OTPs.
- OAuth Integration for social logins (Google, Facebook, etc.).
- Biometric Authentication for the mobile app version.

13.4 Budget Planning and Goal Setting

Introducing a module for budget creation and goal tracking would help users plan monthly or yearly budgets and monitor progress towards financial goals (e.g., saving for a vacation, repaying debt). Users could set limits on specific categories and receive alerts upon exceeding them.

13.5 Recurring Transactions and Reminders

Allowing users to configure recurring expenses (like rent or subscriptions) and automated reminders would streamline expense tracking. Calendar integration could also be implemented to display upcoming bills or payments.

13.6 Multi-Currency and Localization Support

To accommodate international users, support for multiple currencies, local time zones, and language localization can be added. This would make the platform more inclusive and globally accessible.

13.7 Export and Import Functionality

Adding support for exporting transaction data to PDF, CSV, or Excel formats would be beneficial for users who need to maintain offline records or integrate with other financial software. Similarly, allowing import of bank statements or transaction data would

streamline onboarding.

13.8 AI-Driven Expense Categorization

Incorporating machine learning algorithms to automatically categorize expenses based on historical data and transaction descriptions can reduce user effort and improve accuracy in financial summaries.

13.9 Admin Dashboard (for Multi-User or Enterprise Use)

If extended to a business or family budgeting context, the application could support:

- Admin dashboards to oversee multiple user accounts.
 - Role-based access control.
 - Team or household-wide reports and analytics.
-