**Project Report: Full-Stack MERN Expense Tracker Application**

**Author:** Puli Uday Krishna **Date:** August 31, 2025

**1. Introduction**

This document details the development process of a full-stack MERN (MongoDB, Express.js, React, Node.js) Expense Tracker application. The project is designed to be a modern, responsive, and feature-rich tool for personal finance management. It allows users to track their income and expenses, view a dynamic balance, and manage their transaction history. The application is built with a focus on a clean user interface, robust state management, and is deployed to live servers, making it accessible to anyone with a web browser.

Key features include:

* Full CRUD (Create, Read, Delete) functionality for transactions.
* A persistent backend API connected to a MongoDB Atlas cloud database.
* A dynamic React frontend with global state management via the Context API.
* A modern, responsive UI with a light/dark mode toggle.
* Local storage caching for an enhanced, offline-first user experience.
* Live deployment on Render (backend) and Netlify (frontend).

**2. Technology Stack**

* **Frontend**: React.js, React Hooks (useState, useEffect, useContext, useReducer), Axios
* **Backend**: Node.js, Express.js
* **Database**: MongoDB with Mongoose
* **Development Tools**: VS Code, Git & GitHub, Postman, Nodemon
* **Deployment**: Render (for backend), Netlify (for frontend)

**3. Project Structure**

The project is organized into two main directories, backend and client, to maintain a clean separation of concerns.

expense-tracker/

├── backend/

│ ├── config/

│ │ ├── db.js

│ │ └── config.env

│ ├── controllers/

│ │ └── transactions.js

│ ├── models/

│ │ └── Transaction.js

│ ├── routes/

│ │ └── transactions.js

│ ├── node\_modules/

│ ├── package.json

│ └── server.js

│

└── client/

├── public/

│ └── index.html

├── src/

│ ├── components/

│ │ ├── AddTransaction.js

│ │ ├── Balance.js

│ │ ├── Header.js

│ │ ├── ResetButton.js

│ │ ├── ThemeToggle.js

│ │ └── TransactionList.js

│ ├── context/

│ │ └── GlobalState.js

│ ├── App.css

│ ├── App.js

│ └── index.js

├── node\_modules/

├── package.json

└── .gitignore

**4. Backend Implementation**

The backend is a RESTful API built with Node.js and Express.js, responsible for handling all business logic and database interactions.

**4.1. Server Setup (server.js)**

This is the main entry point for the backend. It initializes the Express server, connects to the database, and mounts the API routes.

JavaScript

const express = require('express');

const dotenv = require('dotenv');

const cors = require('cors');

const connectDB = require('./config/db');

const transactions = require('./routes/transactions');

dotenv.config({ path: './config/config.env' });

connectDB();

const app = express();

app.use(cors());

app.use(express.json());

app.use('/api/v1/transactions', transactions);

const PORT = process.env.PORT || 5000;

app.listen(PORT, console.log(`Server running on port ${PORT}`));

**4.2. Database Connection (config/db.js)**

This module handles the connection to the MongoDB Atlas cloud database using Mongoose.

JavaScript

const mongoose = require('mongoose');

const connectDB = async () => {

try {

const conn = await mongoose.connect(process.env.MONGO\_URI);

console.log(`MongoDB Connected: ${conn.connection.host}`);

} catch (err) {

console.error(`Error: ${err.message}`);

process.exit(1);

}

};

module.exports = connectDB;

**4.3. Data Model (models/Transaction.js)**

This Mongoose schema defines the structure for each transaction document in the database.

JavaScript

const mongoose = require('mongoose');

const TransactionSchema = new mongoose.Schema({

text: {

type: String,

trim: true,

required: [true, 'Please add some text']

},

amount: {

type: Number,

required: [true, 'Please add a positive or negative number']

},

createdAt: {

type: Date,

default: Date.now

}

});

module.exports = mongoose.model('Transaction', TransactionSchema);

**4.4. API Routes and Controllers**

The logic is separated into routes and controllers. The routes define the API endpoints, and the controllers contain the functions that execute when those endpoints are hit.

**routes/transactions.js**

JavaScript

const express = require('express');

const router = express.Router();

const { getTransactions, addTransaction, deleteTransaction, deleteAllTransactions } = require('../controllers/transactions');

router

.route('/')

.get(getTransactions)

.post(addTransaction)

.delete(deleteAllTransactions);

router

.route('/:id')

.delete(deleteTransaction);

module.exports = router;

**controllers/transactions.js**

JavaScript

const Transaction = require('../models/Transaction');

// Get all, add, delete one, and delete all transactions

exports.getTransactions = async (req, res, next) => { /\* ... \*/ }

exports.addTransaction = async (req, res, next) => { /\* ... \*/ }

exports.deleteTransaction = async (req, res, next) => { /\* ... \*/ }

exports.deleteAllTransactions = async (req, res, next) => { /\* ... \*/ }

*(Note: Full controller code is in the project files.)*

**5. Frontend Implementation**

The frontend is a single-page application built with React.

**5.1. Global State Management (context/GlobalState.js)**

To avoid prop-drilling, the application's state (transactions, theme, etc.) is managed globally using React's Context API and the useReducer hook. This file also contains all the axios calls to the backend API.

JavaScript

import React, { createContext, useReducer, useEffect } from 'react';

import axios from 'axios';

const API\_URL = 'https://expense-tracker-app-2h6s.onrender.com/api/v1/transactions';

const AppReducer = (state, action) => { /\* ... \*/ };

const getInitialState = () => { /\* ... \*/ };

const initialState = getInitialState();

export const GlobalContext = createContext(initialState);

export const GlobalProvider = ({ children }) => { /\* ... \*/ };

*(Note: Full GlobalState code is in the project files.)*

**5.2. Main App Component (App.js)**

This component assembles all the other UI components into the final layout and uses the GlobalProvider to make the state available to all children.

JavaScript

import React, { useContext, useEffect } from 'react';

import './App.css';

// ... component imports

import { GlobalProvider, GlobalContext } from './context/GlobalState';

const AppContent = () => { /\* ... \*/ };

function App() {

return (

<GlobalProvider>

<AppContent />

</GlobalProvider>

);

}

export default App;

**5.3. UI Components (components/)**

The UI is broken down into small, reusable components, each with a specific responsibility:

* **Balance.js**: Displays the total balance and handles the show/hide functionality.
* **TransactionList.js**: Fetches and displays the income/expense columns in the "Mini Statement."
* **AddTransaction.js**: Provides the form for adding new transactions.
* **ThemeToggle.js**: The button to switch between light and dark modes.
* **ResetButton.js**: The button to clear all transactions.

**6. Deployment**

The application was deployed to live servers, making it a publicly accessible webpage.

**6.1. Backend Deployment (Render)**

1. The project was pushed to a GitHub repository.
2. A new "Web Service" was created on Render, connected to the GitHub repo.
3. Configuration was set as follows:
   * **Root Directory**: backend
   * **Build Command**: npm install
   * **Start Command**: node server.js
4. The MONGO\_URI was added as an environment variable in the Render dashboard.

**6.2. Frontend Deployment (Netlify)**

1. The frontend code was updated to use the live Render URL for all API calls, and the "proxy" was removed from package.json.
2. A new site was created on Netlify, connected to the same GitHub repo.
3. Build settings were configured:
   * **Base directory**: client
   * **Build command**: npm run build
   * **Publish directory**: client/build

The final live application is available at: [**https://expense-tracker-app-18072003.netlify.app/**](https://expense-tracker-app-18072003.netlify.app/)

**7. Conclusion**

This project successfully demonstrates the creation of a full-stack MERN application, from initial setup to final deployment. It incorporates modern web development practices, including component-based architecture, global state management, responsive design, and a CI/CD workflow through Git, Render, and Netlify.