A

PROJECT ON

PRESONAL FINANCE MANAGER WITH MERN TECHNOLOGY

BY

DIKSHA BALIRAM BHOSALE

Table of Contents

1. Introduction

1.1 Project Overview

The Personal Finance Manager is a web application built using the MERN stack (MongoDB, Express.js, React.js, Node.js) that helps users track and manage their financial activities. It allows users to record income and expenses, categorize transactions, set budgets, and analyze financial trends through interactive charts and reports. The system provides a user-friendly interface and ensures data security with authentication mechanisms.

1.2 Objective

The primary objectives of the Personal Finance Manager are:

To provide users with an efficient tool for managing personal finances.

To enable tracking of income, expenses, and financial goals.

To visualize financial data using graphs and reports.

To enhance security through user authentication and authorization.

To offer a responsive and seamless user experience across devices.

1.3 Scope

The project aims to:

Allow users to register and log in securely.

Enable addition, modification, and deletion of financial transactions.

Categorize transactions into predefined or custom categories.

Display monthly and yearly financial summaries with insights.

Provide budgeting and goal-setting functionalities.

Implement data security measures, including password hashing and JWT-based authentication.

Support future integrations, such as bank API connections and AI-driven financial recommendations.

2. Technologies Used

The Personal Finance Manager is built using the MERN stack (MongoDB, Express.js, React.js, Node.js) for a full-stack web development experience. The following technologies are used:

2.1 Frontend Technologies

React.js – For building a dynamic and responsive user interface.

React Router – For client-side navigation.

Redux (Optional) – For state management (if required for complex state handling).

Material-UI / Tailwind CSS / Bootstrap – For UI components and styling.

Chart.js / Recharts – For data visualization (charts and graphs).

2.2 Backend Technologies

Node.js – A JavaScript runtime for backend development.

Express.js – A lightweight framework for handling API requests and routing.

JWT (JSON Web Token) – For user authentication and authorization.

bcrypt.js – For password hashing and security.

dotenv – For environment variable management.

2.3 Database Technologies

MongoDB – A NoSQL database for storing user and transaction data.

Mongoose – An ODM (Object Data Modeling) library for MongoDB to manage database operations efficiently.

2.4 Authentication & Security

JWT (JSON Web Token) – For secure user authentication.

bcrypt.js – For encrypting user passwords.

CORS (Cross-Origin Resource Sharing) – To manage secure API access from different domains.

Helmet.js – For securing HTTP headers in Express.js applications.

2.5 Deployment & DevOps

Frontend Deployment: Vercel / Netlify.

Backend Deployment: Render / Heroku.

Database Hosting: MongoDB Atlas (Cloud-based MongoDB).

Version Control: Git & GitHub for collaboration and code management.

3. System Desig

3.1 Architecture

The Personal Finance Manager follows a client-server architecture, where:

1. Frontend (React.js):

Manages user interactions.

Sends API requests to the backend.

Displays financial data using charts and tables.

2. Backend (Node.js + Express.js):

Handles authentication and business logic.

Processes requests from the frontend.

Communicates with the database.

3. Database (MongoDB):

Stores user data, transactions, and financial goals.

Uses Mongoose ORM for efficient data handling.

4. Authentication (JWT + bcrypt.js):

Secures user sessions with JWT tokens.

Hashes passwords using bcrypt.js for security.

3.1.1 System Architecture Diagram

[React.js (Frontend)] <--> [Express.js + Node.js (Backend)] <--> [MongoDB (Database)]

Each component communicates via RESTful APIs, ensuring modularity and scalability.

3.2 Database Schema

3.2.1 Users Collection

3.2.2 Transactions Collection

3.2.3 Budgets Collection

3.3 API Endpoints

3.3.1 User Authentication APIs

POST /api/auth/register – Register a new user.

POST /api/auth/login – Authenticate and return JWT token.

GET /api/auth/user – Get logged-in user details.

3.3.2 Transactions APIs

POST /api/transactions – Add a new transaction.

GET /api/transactions – Get all transactions of a user.

PUT /api/transactions/:id – Update a transaction.

DELETE /api/transactions/:id – Delete a transaction.

3.3.3 Budget & Goals APIs

POST /api/budgets – Set a budget for a category.

GET /api/budgets – Retrieve budgets.

PUT /api/budgets/:id – Update a budget.

DELETE /api/budgets/:id – Delete a budget.

3.4 Data Flow Diagram (DFD)

1. User Registers/Login → Sends request to backend → Backend verifies credentials → Generates JWT → Returns token to frontend.

2. User Adds Transaction → Frontend sends data → Backend validates & stores in MongoDB.

3. User Views Reports → Frontend requests data → Backend fetches transactions → Returns summarized data.

4. Features & Functionalities

4.1 User Authentication & Management

✅ User Registration & Login – Secure authentication using JWT & bcrypt.js.

✅ Role-Based Access Control (RBAC) (Optional) – Admin vs. Regular User roles.

✅ Profile Management – Users can update their profile details.

✅ Password Encryption – Uses bcrypt.js for enhanced security.

4.2 Financial Transactions (Income & Expenses)

✅ Add Transactions – Users can add income and expenses with details like category, amount, and date.

✅ Edit & Delete Transactions – Users can modify or remove incorrect transactions.

✅ Categorization – Transactions are classified into categories like Food, Rent, Travel, Salary, etc.

✅ Search & Filter – Users can filter transactions by date range, category, or type (Income/Expense).

4.3 Data Visualization & Analytics

✅ Dashboard Overview – Displays total income, expenses, and savings at a glance.

✅ Expense Breakdown Chart – Visualizes spending in pie charts/bar graphs (using Chart.js / Recharts).

✅ Monthly & Yearly Reports – Provides detailed spending analysis over time.

✅ Comparison Insights – Compare spending patterns month-over-month.

4.4 Budgeting & Goal Setting

✅ Set Monthly Budgets – Users can define budgets for categories like Food, Shopping, etc.

✅ Budget Alerts – Notifies users when spending exceeds a preset limit.

✅ Goal Tracker – Users can set financial goals (e.g., Save $1000 in 3 months) and track progress.

✅ Expense Forecasting (Future Scope) – AI-based predictions for future spending trends.

4.5 Additional Features

✅ Dark Mode Support – For an enhanced user experience.

✅ Multi-Currency Support – Users can select their preferred currency.

✅ Export Data – Users can download transaction history as CSV/PDF.

✅ Multi-Device Compatibility – Fully responsive design for web & mobile.

4.6 Security Features

✅ JWT Authentication – Ensures secure access to user data.

✅ Data Encryption – Uses bcrypt.js for password protection.

✅ CORS & Helmet.js – Protects against cross-site scripting (XSS) and other security vulnerabilities.

✅ Two-Factor Authentication (Future Scope) – Additional security for user accounts.

5. Implementation Details

5.1 Frontend Development (React.js)

The frontend is built using React.js, ensuring a dynamic and responsive user interface.

5.1.1 UI Components

Login & Signup Pages → Handles authentication.

Dashboard → Displays financial summaries and charts.

Transaction Page → Allows users to add, edit, and delete transactions.

Budget & Goals Page → Helps users manage their financial goals.

5.1.2 State Management

React Context API / Redux (Optional) → Manages global state for authentication and transactions.

React Query / Axios → Handles API requests efficiently.

5.1.3 API Integration

Axios is used to connect the frontend to the backend APIs.

Implements JWT authentication by storing the token in localStorage or HTTP-only cookies.

5.2 Backend Development (Node.js + Express.js)

The backend handles user authentication, transactions, and data processing.

5.2.1 Server Setup

Express.js is used to create the server.

CORS middleware is enabled for cross-origin access.

dotenv is used to manage environment variables.

5.2.2 Authentication System

JWT (JSON Web Token) is used for secure authentication.

bcrypt.js encrypts passwords before storing them in the database.

5.2.3 RESTful API Endpoints

The backend exposes RESTful APIs to communicate with the frontend:

User Authentication APIs

POST /api/auth/register → Register a new user.

POST /api/auth/login → Authenticate and return a JWT token.

Transactions APIs

POST /api/transactions → Add a new transaction.

GET /api/transactions → Fetch all transactions of a user.

PUT /api/transactions/:id → Edit a transaction.

DELETE /api/transactions/:id → Delete a transaction.

Budget & Goal APIs

POST /api/budgets → Set a new budget.

GET /api/budgets → Retrieve budget data.

5.3 Database Management (MongoDB + Mongoose)

The database is designed using MongoDB with Mongoose as the ODM (Object Data Modeling) library.

5.3.1 Database Schema

User Schema

```
const userSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  createdAt: { type: Date, default:
Date.now }
});
```

Transaction Schema

```
const transactionSchema = new
mongoose.Schema({
  userId: { type:
mongoose.Schema.Types.ObjectId, ref:
'User' },
  type: { type: String, enum: ['Income',
'Expense'] },
  category: String,
  amount: Number,
  date: { type: Date, default:
Date.now }
});
```

Budget Schema

```
const budgetSchema = new
mongoose.Schema({
  userId: { type:
mongoose.Schema.Types.ObjectId, ref:
'User' },
  category: String,
  limit: Number,
  createdAt: { type: Date, default:
Date.now }
});
```

5.4 Deployment

5.4.1 Frontend Deployment

Hosted on Vercel or Netlify for fast and scalable deployment.

Uses CI/CD pipelines for automatic updates.

5.4.2 Backend Deployment

Hosted on Render / Heroku for seamless backend execution.

MongoDB Atlas is used for cloud database hosting.

5.4.3 Environment Variables

Sensitive credentials (JWT secret, database URI) are stored in a .env file and never hardcoded in the codebase.

6. Testing & Deployment

6.1 Testing Strategies

6.1.1 Unit Testing

✅ Frontend (React.js) → Tested using Jest & React Testing Library

✅ Backend (Node.js & Express.js) → Tested using Jest & Supertest

✅ Database (MongoDB) → Tested with MongoDB Memory Server for isolated database tests

Example:

```
test('should create a new transaction',
async () => {
  const res = await request(app)
    .post('/api/transactions')
    .send({ type: 'Expense', category:
'Food', amount: 50 });
  expect(res.statusCode).toBe(201);
});
```

6.1.2 Integration Testing

✅ Ensures smooth communication between frontend, backend, and database

✅ Uses Postman for API testing

✅ End-to-end (E2E) testing using Cypress

Example:

User Flow Test:

1. User logs in → Auth token is generated

2. User adds a transaction → Data is saved in the database

3. User views reports → Correct data is displayed

6.1.3 Performance & Security Testing

✅ Load Testing → Simulates multiple users accessing the system using K6

✅ Security Testing → Prevents SQL Injection, XSS, CSRF attacks using OWASP ZAP

Example Load Test (Using K6):

```
import http from 'k6/http';
export default function () {

http.get('https://your-api-url.com/api/t
ransactions');
}
```

6.2 Deployment Process

6.2.1 Frontend Deployment (React.js)

✅ Hosted on Vercel / Netlify for fast deployment

✅ Steps:

1. Push the latest code to GitHub

2. Connect GitHub repository to Vercel / Netlify

3. Enable CI/CD pipeline for automatic deployments

6.2.2 Backend Deployment (Node.js + Express.js)

✅ Hosted on Render / Heroku for cloud-based backend execution

✅ Uses GitHub Actions for continuous integration

✅ Steps:

1. Push backend code to GitHub

2. Deploy to Render / Heroku

3. Set environment variables (.env) in the cloud environment

6.2.3 Database Deployment (MongoDB Atlas)

✅ MongoDB Atlas is used for cloud storage

✅ Connected securely using MongoDB URI in .env

Example Connection:

```
mongoose.connect(process.env.MONGO_URI,
{
  useNewUrlParser: true,
  useUnifiedTopology: true
});
```

6.2.4 Domain & Hosting

✅ Custom Domain Setup → Using Vercel / Netlify custom domains

✅ SSL Security → Ensured via HTTPS encryption

6.3 Continuous Integration & Deployment (CI/CD)

✅ GitHub Actions automates testing & deployment

✅ Code is automatically deployed after passing tests

Example CI/CD Pipeline in GitHub Actions:

```
name: Deploy Backend
on:
  push:
    branches:
      - main
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2
      - name: Install dependencies
        run: npm install
      - name: Run tests
        run: npm test
      - name: Deploy to Heroku
        run: git push heroku main
```

7. Challenges & Solutions

7.1 Technical Challenges & Solutions

1. Secure User Authentication & Authorization

Challenge: Preventing unauthorized access to sensitive financial data.

Solution:

✅ Implemented JWT (JSON Web Token) for secure authentication.

✅ Used bcrypt.js to hash passwords before storing them in the database.

✅ Implemented role-based access control (RBAC) to restrict access based on user roles.

2. Managing Large Data Efficiently

Challenge: As users store more transactions, database queries slow down.

Solution:

✅ Used MongoDB indexes to speed up database queries.

✅ Implemented pagination when fetching transactions to load data efficiently.

Example:

```
const transactions = await
Transaction.find({ userId })
  .skip(page * limit)
  .limit(limit);
```

3. Handling API Performance & Load Balancing

Challenge: Slow API responses due to multiple database queries.

Solution:

✅ Optimized API responses by caching frequent queries using Redis (future scope).

✅ Used load balancing techniques by deploying backend instances on Render/Heroku.

4. Ensuring Data Security & Preventing Attacks

Challenge: Preventing XSS (Cross-Site Scripting) and CSRF (Cross-Site Request Forgery) attacks.

Solution:

✅ Used Helmet.js to secure HTTP headers.

✅ Enabled CORS policies to prevent unauthorized API requests.

✅ Implemented input validation using Joi/Yup to prevent SQL Injection.

Example:

```
const schema = Joi.object({
  email:
Joi.string().email().required(),
  password:
Joi.string().min(6).required()
});
```

5. Data Visualization for Reports

Challenge: Displaying financial reports dynamically with real-time updates.

Solution:

✅ Used Chart.js / Recharts to visualize income and expenses dynamically.

✅ Implemented real-time updates using React Context API / Redux.

6. Cross-Browser & Mobile Responsiveness Issues

Challenge: UI was inconsistent on different browsers and mobile devices.

Solution:

✅ Used CSS Flexbox & Grid for responsive design.

✅ Performed cross-browser testing on Chrome, Firefox, Safari, Edge.

7. Deployment & Environment Configuration Issues

Challenge: Issues with CORS errors and API failures after deployment.

Solution:

✅ Configured CORS settings in the backend to allow frontend requests.

✅ Used environment variables (.env) for managing API keys securely.

Example:

```
app.use(cors({ origin:
"https://yourfrontend.com", credentials:
true }));
```

7.2 Performance Optimization Strategies

✅ Minimized API Calls – Reduced redundant API requests using React Query.

✅ Lazy Loading – Loaded components and charts only when needed.

✅ Optimized Database Queries – Used Mongoose aggregate functions for faster data retrieval.

✅ Enabled GZIP Compression – Reduced response size for better API performance.

8. Future Scope

8.1 Advanced Features

1. AI-Powered Expense Prediction

- ✅ Machine Learning (ML) Model to analyze past transactions and predict future expenses.
- ✅ Helps users plan their budgets proactively.
- ✅ Uses TensorFlow.js / Scikit-learn for AI-driven predictions.

2. Smart Expense Categorization

- ✅ AI-driven auto-categorization of expenses using Natural Language Processing (NLP).
- ✅ Suggests categories based on transaction descriptions.
- ✅ Example: "McDonald's" → Automatically classified as "Food & Dining".

3. Multi-Currency Support & Forex Updates

- ✅ Users can choose their preferred currency.
- ✅ Real-time exchange rates integration using Forex API.

4. Integration with Bank Accounts & UPI

- ✅ Directly fetches bank transactions using Plaid API / Open Banking APIs.
- ✅ Allows UPI payments & tracking for better financial automation.

5. Voice Command-Based Expense Entry

- ✅ Users can add transactions using voice commands.
- ✅ Uses Google Voice API / Web Speech API.
- ✅ Example: "Add ₹500 to Groceries" → Auto-updates the database.

8.2 Security & Privacy Enhancements

6. Two-Factor Authentication (2FA)

✅ Adds an extra layer of security using OTP or Authenticator Apps.

7. Blockchain-Based Financial Security (Future Scope)

✅ Blockchain ledger for immutable financial records.

✅ Ensures tamper-proof transaction history.

8.3 UI/UX Improvements

8. Dark Mode & Custom Themes

✅ Users can personalize the app with different themes & color schemes.

9. PWA (Progressive Web App) Support

✅ Users can install the app on mobile & desktop without downloading from an app store.

✅ Works offline with service workers.

8.4 Scalability & Deployment Enhancements

10. Microservices Architecture

✅ Breaks the system into independent services for better scalability.

✅ Example: Separate services for Authentication, Transactions, Reports, AI Predictions.

11. Multi-User & Family Finance Tracking

✅ Allows multiple users in one account (e.g., family members).

✅ Shared budgets & expenses for better financial planning.

9.Code and explanation

1. Backend (Node.js + Express.js)

Setup & Installation

1. Create a new backend folder:

```
mkdir finance-manager-backend && cd
finance-manager-backend
```

2. Initialize a Node.js project:

```
npm init -y
```

3. Install dependencies:

```
npm install express mongoose cors
dotenv bcryptjs jsonwebtoken
body-parser
```

4. Install nodemon for auto-reloading (optional):

```
npm install -g nodemon
```

1.1 Backend - Server Setup (server.js)

```
const express = require("express");
const mongoose = require("mongoose");
const dotenv = require("dotenv");
const cors = require("cors");

dotenv.config();

const app = express();
app.use(express.json());
app.use(cors());

// Import Routes
const authRoutes = require("./routes/
authRoutes");
const transactionRoutes = require("./
routes/transactionRoutes");

// Use Routes
app.use("/api/auth", authRoutes);
app.use("/api/transactions",
transactionRoutes);

// Connect to MongoDB
mongoose.connect(process.env.MONGO_URI,
{
  useNewUrlParser: true,
  useUnifiedTopology: true,
}).then(() => console.log("MongoDB
Connected"))
  .catch(err => console.log(err));

const PORT = process.env.PORT || 5000;
app.listen(PORT, () =>
console.log(`Server running on port $
{PORT}`));
```

1.2 Backend - User Model (models/User.js)

```
const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
  name: String,
  email: { type: String, unique: true },
  password: String,
  createdAt: { type: Date, default:
Date.now },
});

module.exports = mongoose.model("User",
UserSchema);
```

1.3 Backend - Transaction Model (models/Transaction.js)

```
const mongoose = require("mongoose");

const TransactionSchema = new
mongoose.Schema({
  userId: { type:
mongoose.Schema.Types.ObjectId, ref:
"User" },
  type: { type: String, enum: ["Income",
"Expense"], required: true },
  category: String,
  amount: Number,
  date: { type: Date, default:
Date.now },
});

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

1.4 Backend - Authentication Routes (routes/authRoutes.js)

```
const express = require("express");
const Transaction = require("../models/
Transaction");
const router = express.Router();

// Add a new transaction
router.post("/", async (req, res) => {
  const { userId, type, category,
amount } = req.body;
  const newTransaction = new
Transaction({ userId, type, category,
amount });

  await newTransaction.save();
  res.json({ message: "Transaction added
successfully" });
});

// Get all transactions
router.get("/:userId", async (req, res)
=> {
  const transactions = await
Transaction.find({ userId:
req.params.userId });
  res.json(transactions);
});

module.exports = router;
```

1.5 Backend - Transaction Routes

```
const express = require("express");
const Transaction = require("../models/
Transaction");
const router = express.Router();

// Add a new transaction
router.post("/", async (req, res) => {
  const { userId, type, category,
amount } = req.body;
  const newTransaction = new
Transaction({ userId, type, category,
amount });

  await newTransaction.save();
  res.json({ message: "Transaction added
successfully" });
});

// Get all transactions
router.get("/:userId", async (req, res)
=> {
  const transactions = await
Transaction.find({ userId:
req.params.userId });
  res.json(transactions);
});

module.exports = router;
```

2. Frontend (React.js)

Setup & Installation

1. Create a new React app:

```
npx create-react-app
finance-manager-frontend
```

2. Install dependencies:

```
npm install axios react-router-dom
```

2.1 Frontend - Authentication (components/Auth.js)

```
import { useState } from "react";
import axios from "axios";

const Auth = () => {
  const [email, setEmail] =
useState("");
  const [password, setPassword] =
useState("");

  const handleLogin = async () => {
    const res = await
axios.post("http://localhost:5000/api
/auth/login", { email, password });
    localStorage.setItem("token",
res.data.token);
  };

  return (
    <div>
      <h2>Login</h2>
      <input type="email"
placeholder="Email" onChange={(e) =>
setEmail(e.target.value)} />
      <input type="password"
placeholder="Password" onChange={(e) =>
setPassword(e.target.value)} />
      <button onClick={handleLogin}
>Login</button>
    </div>
  );
};

export default Auth;
```

2.2 Frontend - Dashboard (components/Dashboard.js)

```
import { useEffect, useState } from
"react";
import axios from "axios";

const Dashboard = () => {
  const [transactions, setTransactions]
= useState([]);
  const userId = "USER_ID_HERE"; //
Replace with logged-in user's ID

  useEffect(() => {
    const fetchTransactions = async ()
=> {
      const res = await
axios.get(`http://localhost:5000/api
/transactions/${userId}`);
      setTransactions(res.data);
    };
    fetchTransactions();
  }, []);

  return (
    <div>
      <h2>Dashboard</h2>
      <ul>
        {transactions.map((t) => (
          <li key={t._id}>{t.category}:
${t.amount}</li>
        ))}
      </ul>
    </div>
  );
};

export default Dashboard;
```

3. Deployment

Backend (Render / Heroku)

1. Deploy backend on Render:

```
git push render main
```

2. Set up MongoDB Atlas and update .env:

```
MONGO_URI=your-mongodb-uri
JWT_SECRET=your-secret-key
```

Frontend (Vercel / Netlify)

1. Build the project:

```
npm run build
```

2. Deploy using Vercel:

```
vercel deploy
```

10.conclusion

The Personal Finance Manager project, built using the MERN technology stack, provides a robust, user-friendly solution for efficiently managing individual financial activities. This application integrates key features such as income and expense tracking, budget planning, and data visualization through interactive charts, ensuring that users can gain comprehensive insights into their financial health. Leveraging the power of React for an intuitive front-end interface, MongoDB for secure and scalable data management, and Express and Node.js for a seamless backend infrastructure, this project demonstrates the significant advantages of adopting modern web technologies. While challenges such as implementing data security and ensuring performance optimization were encountered, these were successfully addressed through best practices and innovative problem-solving approaches. The result is a powerful platform that not only meets its goals of simplifying personal finance management but also holds immense potential for future enhancements like AI-driven financial advice, integration with banking systems, and mobile app development. In conclusion, this project stands as a testament to how technology can empower individuals to take control of their finances with ease and efficiency, making financial planning accessible and impactful.

11.References

https://www.google.com/

https://chatgpt.com/

https://gemini.google.com

https://duckduckgo.com/?q=DuckDuckGo+AI+Chat&ia=chat&duckai=1

https://copilot.microsoft.com/