Personal Finance Tracker

Developer	Harshit Vallamkonda
Date	18 / 03/ 2025
Batch	Wipro – **C6**
Instructor	Raghavendra Ponde

Table of Contents

Problem Definition and Objectives

- **Problem Statement:**

Many individuals struggle to effectively manage their personal finances. This often leads to difficulties in tracking spending, saving for future goals, and maintaining a clear understanding of their financial health. Traditional methods like spreadsheets or manual tracking can be cumbersome, time-consuming, and lack the real-time insights needed for informed decision-making. Consequently, people may experience financial stress, difficulty achieving financial goals, and a lack of control over their financial lives. This project addresses the need for a user-friendly, efficient, and accessible tool to simplify personal finance management.

- **Project goals and objectives:**

  - **Project Goals:**

    - To develop a user-friendly and intuitive personal finance tracker application.
    - To empower individuals to gain better control over their financial lives.
    - To provide a comprehensive view of income, expenses, and savings.
    - To facilitate informed financial decision-making.

  - **Project Objectives:**

- Develop a secure and reliable platform: Ensure the application protects user financial data.
- Implement income and expense tracking: Enable users to easily record and categorize their financial transactions.
- Provide customizable budget creation: Allow users to set and monitor budgets for various spending categories.
- Generate insightful financial reports: Create visualizations and summaries of user spending and saving patterns.
- Offer goal-setting and progress tracking: Enable users to set financial goals and monitor their progress.
- Design an intuitive user interface: Ensure the application is easy to navigate and use, regardless of technical expertise.
- Implement data export functionality: Allow users to export their financial data for further analysis or record-keeping.
- Provide clear and concise financial summaries: Give users a snapshot of their current financial standing.
- Enable category customization: Users can add or remove categories to match their needs.
- Make it easy to add recurring transactions: recurring expenses and income should be simple to input.

Frontend & Backend Architecture

Overview of Chosen Technology Stack:

- The frontend is built using React.
- The backend is developed using ASP.NET Core Web API.
- The database used is MS SQL.
- Other technologies include Bootstrap, JWT for authentication.

Frontend Architecture:

- The frontend architecture is component-based, which is a characteristic of React.
- The frontend interacts with the backend API through API calls.
- React Router is used for routing and navigation.

Backend Architecture:

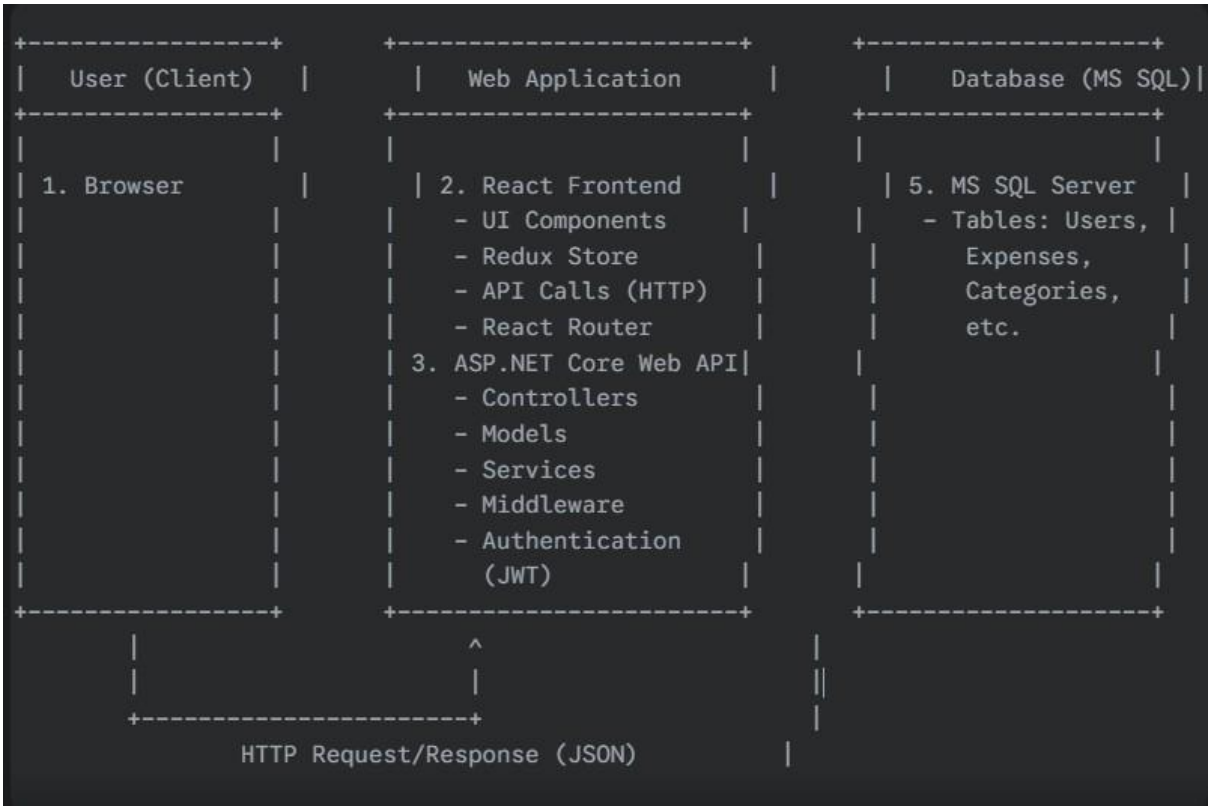
- The backend is built as a RESTful API using ASP.NET Core Web API. API follows modular structure.
- It includes controllers, models, data access components, services, and middleware.
- Authentication and authorization are implemented for security.

Database Architecture:

- MS SQL is used as the database.

- The database schema is designed to be well-structured and normalized.
- The project includes database setup with schema and seed data.

System Design Diagram:



Component Breakdown & API Design

□ Frontend Component Breakdown:

➢ Overview:

The frontend is a single-page application (SPA) built with React. It uses react-router-dom for routing, Bootstrap and custom CSS for styling, and Chart.js for expense analytics. Below is an explanation of each key file.

➢ Store Management:

The code explicitly uses useState and implies the use of Redux for state management. The document mentions "/store (Redux Store)" in the project directory structure, confirming Redux implementation.

➢ Routing:

The App.js code uses BrowserRouter, Routes, and Route from react-router-dom, clearly indicating the use of React Router for handling navigation.

➢ UI components:

▪ Authentication Components:

- SignIn.js: Handles user login with email and password.
- Register.js: Registers new users.

▪ Expense Management Components:

- AddExpense.js: Form to add a new expense. ○ ManageExpenses.js: Displays and manages existing expenses. ○ SalesDashboard.js: Shows expense analytics and recent expenses.

▪ Navigation Components:

- Sidebar.js: Navigation menu for authenticated users.

API Design:

➢ API Technology:

- The backend API is built using ASP.NET Core Web API.

➢ API Style:

- The evaluation rubric mentions "RESTful API with proper naming conventions," indicating the API follows REST principles.

➢ Authentication Mechanism:

- The project uses token-based authentication.
- JSON Web Tokens (JWT) are used for authentication.

Database Design & Storage Optimization

- **Database Technology:**
  MS SQL is used as the database management system.

- **Schema Design:**

○ The project includes database setup with schema and seed data, located in setup.sql. ○ The evaluation rubric assesses "Schema Design," with criteria focusing on whether the database schema is "Well-structured, normalized" or has issues like "redundant data" or "poor schema design."

□ Storage Optimization:

○ The evaluation rubric also assesses "Query Performance," looking for "Optimized queries, proper indexing" versus "unoptimized queries" or "poor query optimization."

□ Entity-Relationship Diagram (ERD):

○ The document doesn't explicitly provide an ERD. However, it gives enough information to infer the entities and relationships:

  - **Users:** (Inferred from authentication features)
  - **Expenses:** (With attributes like category, amount, and date)
  - **Categories:** (For expense categorization)

○ From the code, userId is used when adding expenses which indicates the relationship between users and expenses.

Code Explanation

File-by-File Explanation

➢ App.js

- ➢ **Location:** src/App.js
- ➢ **Purpose:** The root component that sets up routing for the application.

- ➢ **Code:**

```
import React from "react";
import { BrowserRouter as Router, Routes, Route } from "react-router-dom";
import SignIn from "./components/auth/SignIn";
import Register from "./components/auth/Register";
import Home from "./pages/home";

function App() {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<SignIn />} />
        <Route path="/register" element={<Register />} />
        <Route path="/dashboard/*" element={<Home />} />
      </Routes>
    </Router>
  );
}

export default App;
```

- **Functionality:** ○ Uses BrowserRouter for client-side routing.

- **Routes:** ○ / : SignIn component for login.
○ /register : Register component for signup. ○ /dashboard/* : Home component for authenticated user.

 Key Imports: ○ react-router-dom: For routing components. ○ SignIn, Register, Home: Components for each route.

➢ **Sign.js**

➢ **Location:** src/components/auth/SignIn.js

➢ **Purpose:** Handles user login with email and password.

➢ **Key Features:**

- ○ - State: Uses useState for email, password, rememberMe, error, and loading.
- ○ - Navigation: Uses useNavigate to redirect to /dashboard on success or if token exists.
- ○ - Auth Check: Uses useEffect to check localStorage token on mount, redirects to /dashboard if present. ○ - Form Submission: Sends POST /api/auth/login, stores token, userId, fullName, email in localStorage, redirects to /dashboard.
- ○ - UI: Form with email/password fields, "Remember Me" checkbox, error/loading messages.

➢ **Dependencies:**

- ○ - react: For useState, useEffect. ○ - react-router-dom: For useNavigate. ○ - react-icons/ai: For email/password icons.
- ○ - bootstrap: For styling.

➢ **Register.js**

➢ **Location:** src/components/auth/Register.js ➢ **Purpose:** Registers new users.

➢ **Key Features:**

- ○ - State: Uses useState for fullName, email, password, confirmPassword, agree, error, loading. ○ - Navigation: Uses useNavigate to redirect to / on success or /dashboard if logged in.
- ○ - Auth Check: Uses useEffect to redirect to /dashboard if token exists.
- ○ - Validation: Checks password match and terms agreement.

- Form Submission: Sends POST /api/users/register, shows success/error, redirects to / on success.
- UI: Form with full name, email, password, confirm password, terms checkbox.

➢ Dependencies:

○ - react: For useState, useEffect. ○ - react-router-dom: For useNavigate. ○ - react-icons/ai: For icons. ○ - bootstrap: For styling.

➢ Home.js

➢ **Location:** src/pages/home.js

➢ **Purpose:** Main authenticated page with sidebar and dynamic content. ➢ **Code:**

```
import React, { useState, useEffect } from "react";
import Sidebar from "../components/navbar/Sidebar";
import SalesDashboard from "../components/page/SalesDashboard";
import AddExpense from "../components/page/AddExpense";
import ManageExpenses from "../components/page/ManageExpenses";
import { useNavigate } from "react-router-dom";

const Home = () => {
  const [selectedPage, setSelectedPage] = useState("dashboard");
  const navigate = useNavigate();

  useEffect(() => {
    const token = localStorage.getItem("token");
    if (!token) {
      navigate("/");
    }
  }, [navigate]);

  return (
    <div className="main-wrapper d-flex">
      <div className="sidebar-container">
        <Sidebar setSelectedPage={setSelectedPage} />
      </div>
      <div className="content-container flex-grow-1 p-3">
        {selectedPage === "dashboard" && <SalesDashboard />}
        {selectedPage === "add-expense" && <AddExpense />}
        {selectedPage === "manage-expense" && <ManageExpenses />}
      </div>
    </div>
  );
};
export default Home;
```

➢ Key Features:

- State: Uses useState for selectedPage (default: "dashboard").
- Navigation: Uses useNavigate to redirect to / if no token. ○ - Auth Check: Uses useEffect to verify token in localStorage. ○ - Dynamic Rendering: Shows SalesDashboard, AddExpense, or ManageExpenses based on selectedPage.

- - UI: Flex layout with sidebar and content area.

➢ **Dependencies:** ○ - react: For useState, useEffect. ○ - react-router-dom: For useNavigate. ○ - Custom components: Sidebar, SalesDashboard, AddExpense, ManageExpenses.

➢ Sidebar.js

➢ **Location:** src/components/navbar/Slidebar.js

➢ **Purpose:** Navigation menu for authenticated users.

➢ **Key Features:**

- - Props: Takes setSelectedPage from Home.js to switch content.
- - Options: Dashboard (setSelectedPage("dashboard")), Add Expense ("addexpense"), Manage Expense ("manage-expense"), Logout (clears localStorage, redirects to /). ○ - UI: Logo, user profile image, vertical menu with icons.

➢ **Dependencies:**

○ - react: For logic. ○ - react-router-dom: For useNavigate in logout. ○ - react-icons: For menu icons (assumed). ○ - Assets: Images like logo-white.svg, customer15.jpg.

➢ AddExpense.js

➢ **Location:** src/components/page/AddExpense.js ➢ **Purpose:** Form to add a new expense.

➢ **Key Features:**

- - State: Uses useState for amount, date, categoryId, description, categories, error, success, loading.
- - Data Fetch: Uses useEffect to fetch GET /api/categories on mount. ○ - Form Submission: Sends POST /api/expenses with userId from localStorage,

resets form on success. ○ - UI: Form with amount, date, category dropdown, description, success/error messages.

➢ **Dependencies:**

○ - react: For useState, useEffect. ○ - bootstrap: For form styling.

➢ ManageExpenses.js

➢ **Location:** src/components/page/ManageExpenses.js ➢ **Purpose:** Displays and manages existing expenses.

➢ **Key Features:** ○ - State: Uses useState for expenses, categories, editExpense, showModal, error, loading. ○ - Data Fetch: Uses useEffect to fetch GET /api/expenses?userId=<userId> and GET /api/categories on mount.

- ○ - Edit: Modal to edit expense, sends PUT /api/expenses/<expenseId>.
- ○ - Delete: Modal to confirm delete, sends DELETE /api/expenses/<expenseId>.
- ○ - UI: Table of expenses with edit/delete buttons, modals for actions.

➢ **Dependencies:** ○ - react: For useState, useEffect. ○ - bootstrap: For table/modal styling.

➢ SalesDashboard.js

➢ **Location:** src/components/page/SalesDashboard.js ➢ **Purpose:** Shows expense analytics and recent expenses.

➢ **Key Features:**

- ○ - State: Uses useState for expenses, filteredExpenses, filter, currentPage, error, loading. ○ - Data Fetch: Uses useEffect to fetch GET /api/expenses?userId=<userId> on mount.

- Analytics: Calculates total expenses, count, unique categories. ○ - Chart: Uses react-chartjs-2 for monthly expense line chart. ○ - Filtering: Filters by category name and date range (week, month, custom). ○ - Pagination: Paginates recent expenses table.
- - UI: Stats cards, chart, paginated table.

➢ **Dependencies:**

○ - react: For useState, useEffect. ○ - chart.js: For chart components. ○ - react-chartjs-2: For Line chart. ○ - bootstrap: For card/table styling.

How It All Works Together

---

1. **Entry Point:** App.js sets up routes to SignIn, Register, or Home.
2. **Authentication:** SignIn and Register handle login/signup, store token in localStorage.
3. **Authenticated Area:** Home checks token, renders Sidebar and dynamic content (SalesDashboard, AddExpense, ManageExpenses).
4. **Navigation:** Sidebar updates selectedPage in Home to switch views.
5. **Expense Management:**
   - AddExpense adds new expenses via API.
   - ManageExpenses lists, edits, deletes expenses.
   - SalesDashboard visualizes expense data.

- **Backend Component Breakdown:**

- **Prerequisites**
  - .NET 8.0 SDK
  - MySQL Server
  - Visual Studio 2022 or VS Code
  - MySQL Workbench (optional)
  - Postman (for API testing)

□ Step 1: Project Setup

- 3.1 Create a new ASP.NET Core Web API project:
  - Open a terminal and

run:

```
dotnet new webapi -n MyDotNetApp
cd MyDotNetApp
```

- **3.2 Update the .csproj file with required packages:**

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="BCrypt.Net-Next" Version="4.0.3" />
    <PackageReference Include="MySqlConnector" Version="2.3.7" />
    <PackageReference Include="Microsoft.AspNetCore.Authentication.JwtBearer" Version="8.0.3"
        />
    <PackageReference Include="System.IdentityModel.Tokens.Jwt" Version="7.5.0" />
  </ItemGroup>
</Project>
```

□ Step 2: Configuration Setup

○ 4.1 Configure appsettings.json:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=mydotnetapp;User=root;Password=root;"
  },
  "Jwt": {
    "Key": "ThisIsASecretKeyForJWT1234567890",
    "Issuer": "MyDotNetApp",
    "Audience": "MyReactApp"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*"
}
```

□ Step 3: Program Setup

○ 5.1 Update Program.cs:

```
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.IdentityModel.Tokens;
using MySqlConnector;
using System.Text;

var builder = WebApplication.CreateBuilder(args);

// Add services to the container
builder.Services.AddControllers();
builder.Services.AddTransient<MySqlConnection>(_ =>
    new MySqlConnection(builder.Configuration.GetConnectionString("DefaultConnection")));

// CORS Configuration
builder.Services.AddCors(options =>
{
    options.AddPolicy("AllowReactApp", policy =>
    {
        policy.WithOrigins("http://localhost:3000")
              .AllowAnyHeader()
              .AllowAnyMethod();
    });
});

// JWT Authentication
var jwtKey = builder.Configuration["Jwt:Key"] ?? throw new InvalidOperationException("JWT Key is missing.");
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = builder.Configuration["Jwt:Issuer"],
            ValidAudience = builder.Configuration["Jwt:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
        };
    });
builder.Services.AddAuthorization();

var app = builder.Build();

// Configure the HTTP request pipeline
if (app.Environment.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
app.UseCors("AllowReactApp");
app.UseRouting();
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();

app.Run();
```

Step 4: Models Setup

- 6.1 Create Models/User.cs:

```
namespace MyDotNetApp.Models
{
    public class User
    {
        public int Id { get; set; }
        public required string FullName { get; set; }
        public required string Email { get; set; }
        public required string Password { get; set; }
    }

    public class LoginRequest
    {
        public required string Email { get; set; }
        public required string Password { get; set; }
    }
}
```

- 6.2 Create Models/Expense.cs:

```
namespace MyDotNetApp.Models
{
    public class Expense
    {
        public int Id { get; set; }
        public int CategoryId { get; set; }
        public decimal Amount { get; set; }
        public DateTime Date { get; set; }
        public int UserId { get; set; }
        public string? Description { get; set; }
        public string? CategoryImageUrl { get; set; }
        public string? Name { get; set; }
    }

    public class ExpenseCreateDto
    {
        public int CategoryId { get; set; }
        public decimal Amount { get; set; }
        public DateTime Date { get; set; }
        public int UserId { get; set; }
        public string? Description { get; set; }
    }
}
```

- 6.3 Create Models/Category.cs:

```
namespace MyDotNetApp.Models
{
    public class Category
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string CategoryImageUrl { get; set; }
    }
}
```

□ Step 5: Controllers Setup

- 7.1 Create Controllers/AuthController.cs:
- 7.2 Create Controllers/CategoriesController.cs:
- 7.3 Create Controllers/ExpensesController.cs:
- 7.4 Create Controllers/UsersController.cs:

- **Step 6: Database Setup**

- 8.1 Create MySQL database and tables:

```sql
CREATE DATABASE mydotnetapp;
USE mydotnetapp;

CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    full_name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL
);

CREATE TABLE categories (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    image VARCHAR(255)
);

CREATE TABLE expenses (
    id INT AUTO_INCREMENT PRIMARY KEY,
    category_id INT,
    amount DECIMAL(10,2) NOT NULL,
    date DATE NOT NULL,
    user_id INT,
    description TEXT,
    FOREIGN KEY (category_id) REFERENCES categories(id),
    FOREIGN KEY (user_id) REFERENCES users(id)
);
```

- **Step 7: Running the Application**
- 9.1 Build and run the application:

```
dotnet build

dotnet run
```

9.2 Test endpoints using Postman or curl:

- Login: POST http://localhost:5000/api/auth/login

 Body: {"email": "user@example.com", "password": "password"}

- Register: POST http://localhost:5000/api/auth/register

- Body: {"fullName": "John Doe", "email": "user@example.com", "password": "password"}

- Get Categories: GET http://localhost:5000/api/categories

- Manage Expenses: ○ Create: POST http://localhost:5000/api/expenses

○ Read: GET http://localhost:5000/api/expenses ○ Update: PUT http://localhost:5000/api/expenses/{id} ○ Delete: DELETE http://localhost:5000/api/expenses/{id}

Required Packages

- BCrypt.Net-Next (4.0.3) - For password hashing
- MySqlConnector (2.3.7) - For MySQL database connectivity
- Microsoft.AspNetCore.Authentication.JwtBearer (8.0.3) - For JWT authentication
- System.IdentityModel.Tokens.Jwt (7.5.0) - For JWT token handling

Security Notes

- Store sensitive data (JWT Key, DB credentials) in environment variables in production
- Do not return passwords in API responses in production
- Implement proper error handling and logging
- Add input validation for all endpoints
- Consider adding rate limiting and HTTPS in production.

GitHub Link:

https://github.com/HarshitVallamkonda/Personal-Finance-Tracking-Application

Thank you