**A black background with a black square

AI-generated content may be incorrect.ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

Факултет Приложна математика и информатика

**ДИПЛОМНА РАБОТА**

Тема: **“Приложение за управление на лични финанси интегриращо базиран на изкуствен интелект асистент”**

**Дипломант**: Васил Ганев

**Факултетен номер**: 471221010

**Научен ръководител:**  доц. д-р Златко Захариев

**Специалност**: Информатика и софтуерни науки

**ОКС**: Бакалавър

СОФИЯ 2024

**Съдържание**

[**Въведение**……………………………...……………………………………....5](#Въведение)

[**Глава 1: Цел на проекта**…………….,,.……………………………………..6](#Цел_на_проекта)

[1.1 Към какъв тип потребители е насочен проекта?.............................6](#Към_какъв_тип_потребители_е_насочен)

[1.2 Защо потребителите да ползват FinEdge, вместо конкуренцията?](#Защо_потребителите_да_ползват_FinEdge)

[1.3 Какво прави FinEdge по-добрият избор?..........................................7](#Какво_прави_FinEdge_подобрият_избор)

[**Глава 2: Технологии и среди за разработка**…………...…………………..8](#Технологии_и_среди_за_разработка)

[2.1 Използвани Технологии…………………………………………….8](#Използвани_Технологии)

[2.1.1 C#.............................................................................................8](#C_шарп)

[2.1.2 NET Core Web API………………………………………….8](#NET_Core_Web_API)

[2.1.3 Entity Framework……………………………………………8](#Entity_Framework)

[2.1.4 JWT (JSON Web Token)…………………………………….8](#JWT)

[2.1.5 Swashbuckle…………………………………………………9](#Swashbuckle)

[2.1.6 Hangfire……………………………………………………...9](#Hangfire)

[2.1.7 OpenAI Developer API……………………………………...9](#OpenAI_Developer_API)

[2.1.**8 Vite…………………………………………………………..9**](#Vite)

[2.1.9 TypeScript……………………………………………………9](#TypeScript)

[2.1.10 React………………………………………………………10](#React)

[2.1.11 React Router DOM………………………………………..10](#React_Router_DOM)

[2.1.12 TanStack Query (React Query)……………………………10](#TanStack_Query)

[2.1.13 KendoReact……………………………………………….10](#KendoReact)

[2.1.4 Tailwind…………………………………………………….11](#Tailwind)

[**2.2 Среди за разработка**…..…………………………………………..11](#Среди_за_разработка)

[2.2.1 Visual Studio………………………………………………..11](#Visual_Studio)

[2.2.2 Visual Studio Code…………………………………………11](#Visual_Studio_Code)

[2.2.3 SQL Server Management Studio (SSMS)……………….…11](#SQL_Server_Management)

[2.2.4 Node.js……………………………………………………...11](#Node_js)

[**Глава 3: Анализ на изискванията към проекта………………………...12**](#Анализ_на_изискванията_към_проекта)

[3.1 Функционални изисквания към проекта…………………………12](#Функционални_изисквания_към_проекта)

[3.1.1 Публична част……………………………………………..12](#Публична_част)

[3.2 Нефункционални изисквания към проекта………………………12](#Нефункционални_изисквания_към_проекта)

[3.2.1 Сигурност………………………………………………….12](#Сигурност)

[3.2.1 Производителност………………………………………...12](#Производителност)

[3.2.1 Поддръжка............................................................................13](#Поддръжка)

[3.2.1 Употреба и достъп...............................................................13](#Употреба_и_достъп)

[3.2.1 Съвместимост.......................................................................13](#Съвместимост)

[3.3 Структура на базата данни...............................................................13](#Структура_на_базата_данни)

[3.4 Архитектура на проекта...................................................................16](#Архитектура_на_проекта)

[3.5 Интеграцията с OpenAI API и LLM (Large Language Model).......16](#Интеграцията_с_OpenAI_API)

[**Глава 4: Техническа реализация…………………………………………..**18](#Техническа_реализация)

[4.1 Структура на проекта……………………………………………...18](#Структура_на_проекта)

[4.1.1 FinEdgeFrontend…………………………………………...18](#FinEdgeFrontend)

[4.1.2 FinEdgeBackend……………………………………………20](#FinEdgeBackend)

[4.1.3 FinEdgeTests……………………………………………….21](#FinEdgeTests)

[4.2 Създаване на потребители и потребителски профили…………..22](#Създаване_на_потребители_и_потребителски)

[4.3 Създаване на Акаунти (сметки)…………………………………...23](#Създаване_на_акаунти_сметки)

[4.3.1 Backend имплементация………………………………….23](#AccountController)

[4.3.2 Frontend имплементация………………………………….25](#GetAccounts_tsx)

[4.4 Създаване на Категории…………………………………………...27](#Създаване_на_категории)

[4.5 Създаване на Транзакции………………………………………….27](#Създаване_на_транзакции)

[4.5.1 Backend имплементация………………………………….27](#TransactionController_cs)

[4.5.2 Frontend имплементация………………………………….29](#GetTransactions_sx)

[4.6 Интеракция с AI асистеснт………………………………………..31](#Интеграция_на_AI_асистент_в_проекта)

[4.6.1 Backend имплементация………………………………….31](#FinancialRecommendationController_cs)

[4.6.2 Frontend имплементация………………………………… 33](#CreateFinancialRecommendations_tsx)

[**Глава 5: Ръководство и използване**……………………………………….38](#Ръководство_за_използване)

[**Заключение**…………………………………………………………………..39](#Заключение)

[**Декларация за авторство**…………………………………………………...40](#ДЕКЛАРАЦИЯ)

[**Използвана литература**……………………………………………………..41](#ИЗПОЛЗВАНА_ЛИТЕРАТУРА)

**Въведение**

Необходимостта от ефективно управление на личните финанси става все по-осезаема, тъй като разходите и приходите на хората стават по-динамични и разнообразни. С навлизането на технологиите в ежедневието, дигиталните решения предлагат удобен начин за следене и анализиране на финансовото състояние. В този контекст софтуерните приложения за управление на личните финанси играят ключова роля в изграждането на по-добра финансова дисциплина и планиране.

Настоящият проект се фокусира върху разработката на приложение за следене на личните финанси, базирано на **React** за клиентската част и **.NET Core Web API** за сървърната логика. Към него е интегрират AI асистент, който помага допълнително на потребителя с разпределянето на финансите си, и му дава съвети базирани на финансовите му навици и методология. Основната цел на приложението е да предостави удобен и интуитивен интерфейс, който позволява на потребителите да следят своите приходи и разходи, да анализират финансовите си навици и да планират бюджета си ефективно.

Проектът включва изграждането на архитектурата на системата, разработването на основните функционалности, като категоризация на разходите, визуализация на финансови данни и генериране на отчети, както и интеграция с база данни за сигурно съхранение на информацията. Освен това ще бъдат разгледани и внедрени добри практики за потребителско изживяване и оптимизация на производителността.

**Глава 1: Цел на проекта**

**Към какъв тип потребители е насочен проекта?**

FinEdge е насочен към хората, които искат да управляват по-ефективно своите лични финанси, като следят приходите и разходите си, създават бюджети и планират бъдещи финансови цели. Приложението предоставя удобен интерфейс и автоматизирани инструменти, които помагат на потребителите да анализират финансовите си навици, да намалят ненужните разходи и да подобрят финансовата си стабилност.

Основната цел на проекта е да създаде интуитивна и достъпна платформа, която комбинира гъвкавост, сигурност и модерни технологии за ефективно управление на личните финанси.

Интегрираният AI асистент допълнително служи за анализ на финансовите навици, предоставяне на персонализирани съвети и помощ при създаването на бюджети и финансови планове. Той може да идентифицира модели на разходи, да предлага оптимизации за спестяване и да отговаря на въпроси, свързани с управлението на личните финанси, спрямо **методологията** на потребителя. Също така може да предупреждава за потенциални финансови рискове и да предлага алтернативни стратегии за постигане на дългосрочни финансови цели.

**Защо потребителите да ползват FinEdge, вместо конкуренцията?**

FinEdge предлага модерно и интуитивно решение за управление на личните финанси, като комбинира удобен интерфейс, мощни аналитични инструменти и AI асистент. В сравнение с популярните приложения като 1Money, Spendee и Monefy, FinEdge предоставя по-широк набор от функции и адресира някои от основните им недостатъци:

* **Spendee** – Приложението предлага разнообразни функционалности, но не е user-friendly за нови потребители и може да бъде трудно за настройка. Освен това, липсва персонализиран AI асистент, който да помага с анализи и финансови съвети.
* **Monefy** - Има опростен интерфейс, но липсва разширена аналитика и възможност за детайлно бюджетиране. Също така, няма AI асистент, който да предоставя персонализирани препоръки или да помага с управлението на разходите.
* **1Money** – Приложението беше една от най-добрите алтернативи, но от ноември 2024 е премахнато от Google Play и App Store, което прави достъпът до него ограничен. Освен това, **няма AI асистент**, който да помага с оптимизацията на личните финанси.

**Какво прави FinEdge по-добрият избор?**

* **Интуитивен интерфейс**, подходящ както за начинаещи, така и за напреднали потребители.
* **AI асистент**, който анализира разходите и предлага персонализирани финансови съвети.
* **Мощни аналитични инструменти** – диаграми, статистики и детайлни отчети за финансите.
* **Десктоп версия**, която позволява удобен достъп и управление на финансите на по-голям екран.
* **Сигурност и гъвкавост**, с интеграция на модерни технологии за защита на потребителските данни.

FinEdge не само улеснява управлението на личните финанси, но също така предоставя интелигентна подкрепа, която липсва при конкурентите.

**Целева аудитория**

Крайните потребители на FinEdge са хора, които редовно управляват личните си финанси и търсят удобство, бързина и лесен достъп до финансовата си информация.

Те могат бързо и ефективно да следят своите приходи, разходи и бюджети, като това е гарантирано чрез интуитивния интерфейс и интегрирания AI асистент, който предоставя персонализирани анализи и препоръки.

**Глава 2: Технологии и среди за разработка на приложението**

**Използвани Технологии**

**C#**

C# е обектно ориентиран език за програмиране, разработен от Microsoft като част от софтуерната платформа .NET.

**.NET Core Web API**

**.NET Core Web API** е фреймворк за създаване на уеб услуги (RESTful APIs), който позволява разработката на мащабируеми и високопроизводителни бaкенд приложения. Той предоставя инструменти за обработка на HTTP заявки, управление на бази данни, сигурност и автентикация, което го прави подходящ за модерни уеб и мобилни приложения.

**Entity Framework**

Entity Framework (EF) е ORM (Object-Relational Mapper) за .NET, който улеснява работата с бази данни, като позволява да се използват C# обекти вместо SQL заявки. Той автоматично преобразува обектите в таблици и обратно, което ускорява разработката и намалява риска от грешки при работа с данни.

**JWT (JSON Web Token)**

JSON Web Token е предложен интернет стандарт за създаване на данни с незадължителен подпис и/или незадължително криптиране, чийто полезен товар (payload) съдържа JSON, който предявява известен брой претенции. Токените се подписват или с помощта на частна тайна, или с публичен/личен ключ.

**Swashbuckle**

**Swashbuckle** е библиотека за **автоматично генериране на Swagger документация** за ASP.NET Core уеб API-та. Позволява създаване на интерактивна документация, чрез която API-тата могат лесно да бъдат тествани.

**Hangfire**

Hangfire е библиотека за изпълнение на фонови задачи (background jobs) в .NET. Тя позволява лесно планиране, отложено изпълнение и периодично стартиране на задачи.

**OpenAI Developer API**

Този пакет осигурява достъп до **OpenAI API**, което позволява използване на модели като GPT-4.o за обработка и анализ на данни, автоматизация и други.

**Vite**

Vite е локален сървър за разработка на фронт енд частта на приложения и използван за шаблони на проекти на React, Vue, Svelte и други.

**TypeScript**

TypeScript е език за програмиране на високо ниво с отворен код, разработен от Microsoft, който добавя статично въвеждане с незадължителни анотации за тип към JavaScript. Той е предназначен за разработване на големи приложения и се транспилира в JavaScript.

**React**

React (известен също като React.js или ReactJS) е безплатна библиотека с отворен код, която има за цел да направи изграждането на потребителски интерфейси, базирани на компоненти, по-"безпроблемно". Поддържа се от Meta и общност от отделни разработчици и компании.

**React Router DOM**

React Router е библиотека за **маршрутизация в React**, която позволява създаване на навигация между различни страници в приложението.

**TanStack Query (React Query)**

TanStack Query (известен преди като React Query) е “липсващата” библиотека за извличане на данни за уеб приложения. Библиотеката прави извличането, кеширането, синхронизирането и актуализирането на състоянието на сървъра във уеб приложения по-лесно.

**Axios**

Axios е базиран на обещания (promises) HTTP клиент за node.js и браузъра. Той е изоморфен (може да работи в браузъра и nodejs със същата кодова база). От страна на сървъра той използва http модул node.js, докато на клиента (браузър) използва XMLHttpRequests.

**KendoReact**

KendoReact е UI библиотека от корпоративен клас със 120+ безплатни и премиум компонента, проектирана и създадена за разработване на бизнес приложения с React.

**Tailwind**

Tailwind CSS е CSS рамка (framework) с отворен код. Тя създава списък от "полезни" CSS класове, които могат да се използват за стилизиране на всеки елемент чрез смесване и съпоставяне.

**Среди за разработка**

**Visual Studio**

Visual Studio е интегрирана среда за разработка (IDE), разработена от Microsoft. Използва платформи за разработка на софтуер на Microsoft, включително Windows API, Windows Forms, Windows Presentation Foundation (WPF), Microsoft Store, Microsoft Silverlight и други.

**Visual Studio Code**

S Code е модерна, лека и мощна интегрирана среда за разработка , създадена от Microsoft. Средата за разработка е налична за всички операционни системи - Linux, Windows и macOS.

**SQL Server Management Studio (SSMS)**

**SQL Server Management Studio**(**SSMS**) е софтуерно приложение, което се използва за конфигуриране, управление и администриране на всички компоненти в SQL Server.

**Node.js**

Node.js е междуплатформена среда за изпълнение на JavaScript с отворен код, която може да работи на Windows, Linux, Unix, macOS и др.

**Глава 3: Анализ на изискванията към проекта**

**Функционални изисквания към проекта**

**Публична част**

Чрез публичната част, клиентите трябва да имат възможност за:

* Създаване/Влизане в своя профил и редакция на данните в него
* Създаване/Редакция/Филтриране на Акаунти/Категории/Транзакци
* Визуализация на съсдадените Акаунти/Категории/Транзакции
* Интеракция с AI асистент (AIPrompt), който да дава персонализирани съвети към потребителя, спрямо неговите данни, навици за харчене и избрана финансова методология
* Да не се затруднява с използването на приложението (приложението трябва да има разбираем UI/UX)
* Да може да ползва приложението и на телефона си (приложението трябва да е mobile responsive)

**Нефункционални изисквания към проекта**

**Сигурност**

Системата трябва да предоставя надеждни методи за удостоверяване и контрол на достъпа за потребители. Данни като пароли следва да се съхраняват във вид на хеширани стойности.

**Производителност**

Бързото зареждане на страниците и ефективната обработка на данни, дори при високо натоварване, осигуряват качествено и приятно потребителско изживяване.

**Поддръжка**

Кодът и архитектурата на платформата трябва да бъдат ясно документирани и структурирани, с цел да се улесни поддръжката и бъдещите актуализации от страна на разработчиците.

**Употреба и достъп**

Потребителският интерфейс трябва да бъде интуитивен и удобен за работа, с ясно разположени функции и лесна навигация. Приложението следва да бъде оптимизирано за коректна работа на различни устройства и браузъри.

**Съвместимост**

Системата трябва да поддържа лесна интеграция с външни модули/библиотеки/API като React Router DOM, KendoReact и OpenAI API. Освен това, тя трябва да позволява лесен начин за разработчика да обновява вечедобавените пакети, за да може винаги да ползва най-актуалните им версии.

**Структура на базата данни**

Изборът на **Microsoft SQL Server**, управляван чрез **SQL Server Management Studio (SSMS)**, пред други релационни бази данни като MySQL или PostgreSQL, може да бъде обоснован с няколко ключови предимства, които го правят особено подходящ за бизнес и академични приложения, изискващи висока производителност, сигурност и интеграция с .NET екосистемата.

SQL Server предлага **пълна поддръжка на ACID (Atomicity, Consistency, Isolation, Durability) транзакции**, което гарантира надеждността и целостта на данните – изключително важно при финансови приложения. Също така, платформата осигурява вградени механизми за сигурност, мащабируемост и удобство при работа със сложни заявки, тригери, съхранени процедури и изгледи.

Допълнително предимство е тясната интеграция с .NET и Entity Framework, което улеснява изграждането на бакенд логиката и осигурява безпроблемна комуникация между приложението и базата данни. SQL Server Management Studio предоставя интуитивен интерфейс за разработка, тестване и администриране на бази данни, което ускорява процеса на разработка и поддръжка на FinEdge.

На фигура 1 са показани връзките между таблиците на приложението:

A computer screen shot of a computer

AI-generated content may be incorrect.

**Фигура 1. Връзки между таблиците от базата данни на приложението**

В основата на приложението стои релационна база данни, която съхранява информация за потребителите и техните финансови дейности. Основната таблица Users съдържа основните данни за потребителите, включително име, имейл, парола, общ баланс и мметодология. Всяко действие и обект в системата е свързан с конкретен потребител чрез UserID.

Таблицата Accounts (Акаунти) съхранява информация за различните сметки на потребителя – банкови, парични, инвестиционни и др. Към всяка сметка могат да бъдат асоциирани множество Transactions (Транзакции), които описват конкретни финансови операции – разходи, приходи, повтарящи се плащания и др. Всяка транзакция е свързана едновременно със сметка и категория.

Categories (Категории) дефинират различните типове разходи и приходи (например "Храна", "Заплата", "Транспорт"), а към тях могат да бъдат създавани по-финни подразделения в таблицата Subcategories (Събкатегории). За всяка категория се поддържат месечни обобщения чрез CategorySnapshots, които позволяват ретроспективен анализ на бюджета.

Notifications (Нотификации) са предназначени за известяване на потребителя – например за важни събития или препоръки.

Финансови препоръки (FinancialRecommendations) съхранява съвети и анализи, предоставени от AI асистента, като всеки запис съдържа заглавие и отговор на конкретна заявка.

За сигурността на аутентикацията е включена таблицата RefreshTokens (Рефреш токени), където се пазят активните токени на потребителите, с информация дали са оттеглени и докога са валидни.

Цялостната структура е организирана с ясно дефинирани зависимости и външни ключове, което осигурява стабилна, надеждна и разширяема основа за управлението на личните финанси в приложението FinEdge.

**Архитектура на проекта**

При първоначалното проектиране на приложение, от изключително значение е да се вземе под внимание дали се предвиждат интеграции с външни системи. Винаги е полезно да съществува документация, която описва тези взаимовръзки. На фигура 2 е представена контекстната архитектура на приложението.

A diagram of software components

AI-generated content may be incorrect.

**Фигура 2. Контекстна** архитектура на приложението

**// dobavi komponentna diagrama**

**Интеграцията с OpenAI API и LLM (Large Language Model)**

За да бъде успешно интегрирана услугата на OpenAI, трябва да бъдат разгледани основните й елементи и как те взаимодействат помежду си в детайли. Това включва конфигурация на API клиент, обработка на заявки, управление и отговори. На фигура 3 е показан процеса при създаване създаване на AI генерирани препоръки.

A diagram of a software application

AI-generated content may be incorrect.

**Фигура 2.** Процес при създаване създаване на AI генерирани препоръки

Стъпки за създаване на AI генерирани препоръки

1. Клиентът изпраща заявка с въведен prompt и избрана дата.
2. Сървърът получава заявката и взима данните на текущия потребител.
3. Тези данни се сериализират и се комбинират с допълнителни инструкции.
4. Сървърът изпраща съобщението към OpenAI API.
5. LLM връща отговор – кратка финансова препоръка.
6. Сървърът записва препоръката в базата и изпраща отговора обратно към клиента.

**Глава 4: Техническа реализация**

**Структура на проекта**

Приложението се състои от няколко главни директории:

* **FinEdgeFrontend**
* **FinEdgeBackend**
* **FinEdgeTests**

**FinEdgeFrontend**

Тази директория съдържа няколко различни папки и файлове, които се отнасят за фронтенд частта на приложението.

**node\_modules -** Тази папка съдържа всички зависимости на проекта, които са инсталирани чрез npm или yarn. Това включва библиотеки като React, React Router, KendoReact и други, които проектът използва.

**src -** Това е основната папка, където се намира целият фронтенд код на проекта. Тук се съдържат компонентите на React, логиката за рендериране на страниците, и други важни части от приложението. Вътре в src има подкатегории:

* **Components -** Тази папка съдържа отделни React компоненти, които могат да бъдат използвани на различни места в приложението. Всеки компонент е TypeScript файл, който позволява използването на JSX синтаксис (TSX) за визуализация на част от интерфейса.
* **Hooks -** В тази папка се намират custom React hooks. Те са функции, които позволяват да използваме React State и други функционалности в компонентите без употребата на класове. Основно куките са създадени за взимане на информацията от сървъра
* **Pages** - В тази папка се съдържат компонентите, които съответстват на различни страници в приложението.
* **Utils** - Тази папка съдържа помощни функции, endpoints за връзка със сървъра и TypeScript типове, които се използват в различни части на приложението. Те могат да включват функции за форматиране на данни, валидиране на входни данни, манипулиране с дати и т.н. Пример: formatDate.js, validateInput.js.

Останалата часто от файловете за автоматично генерирани от Vite и служат основно за конфигуриране на клиентската част на проекта

A screenshot of a computer program

AI-generated content may be incorrect.

**FinEdgeBackend**

Структурата на папките и файловете е специфична за сървърната част на приложението. Всеки компонент има своя роля в архитектурата на приложението и е организиран по начин, който улеснява управлението и поддръжката на кода.

**Controllers -** Папката съдържа класовете, които обработват HTTP заявките, постъпващи от клиента (браузър/мобилно приложение). Всеки контролер съответства на определен ресурс или група ресурси и съдържа методи, които се наричат "Action Methods". Тези методи обработват различни HTTP методи като GET, POST, PUT, DELETE и др.

**Data -** Папката съдържа класове, които се отнасят до достъпа до данни и комуникацията с базата данни (DbContext).

**DTOs (Data Transfer Objects) -** Папката съдържа класове, които се използват за прехвърляне на данни между слоевете на приложението (например между контролери и услуги). DTO класовете не трябва да съдържат бизнес логика, а само данни, които трябва да се предадат.

**Enums -** Папката съдържа изброими типове (enums), които дефинират набор от свързани стойности. Те се използват за представяне на фиксирани стойности или състояния, които имат ограничен набор.

**Interfaces -** Папката съдържа интерфейси, които дефинират контрактите за взаимодействие между различните слоеве на приложението. Интерфейсите се използват за отделяне на абстракцията от конкретната имплементация.

**Migrations -** Папката съдържа файловете, които управляват миграциите на базата данни при използване на Entity Framework Core. Миграциите се използват за актуализиране на структурата на базата данни (създаване на таблици, промени в колоните и т.н.) и за синхронизиране на модела с реалната база данни.

**Models -** Папката съдържа класовете, които дефинират структурираните данни, използвани в приложението, като същинските "модели" на базата данни, които ще бъдат съхранени в нея.

**Services -** Папката съдържа класове, които реализират бизнес логиката на приложението. Тези услуги обикновено се използват от контролерите за извършване на сложни операции и обработка на данни. Те могат да използват репозиториите за достъп до базата данни и да предоставят логика, която не е свързана директно с представянето (например в контролерите).

A screenshot of a computer

AI-generated content may be incorrect.

**FinEdgeTests**

Този проект е създаден с цел юнит тестване на различните контролери от FinEdgeBackend, и по-специално е насочен към проверка на логиката, която се изпълнява в тях, без да се включват реални зависимости като бази данни или външни услуги.

Използван е FakeItEasy – библиотека за създаване на фалшиви обекти, която прави лесно мокването на зависимости.

A screenshot of a computer

AI-generated content may be incorrect.**Създаване на потребители и потребителски профили**

За да могат потребителите да се регистрират и виждат своите профили е необходимо първо да се мсъздаде инстанция на User класа. На фигура 4 е показана последователността на действия, които се извършват като потребител се регистрира.

A screenshot of a computer screen

AI-generated content may be incorrect.

Фигура 4. Последователност от действия за регистрацията на нов потребител

Може да разгледате пълния код как е имплементирана регистрацията на следните линкове:

* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeFrontend/src/Components/HomePage/Cards/UserProfile.tsx#L8> - Frontend
* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeBackend/Controllers/AuthenticationController.cs#L12> - Backend

**Създаване на акаунти (сметки)**

След регистрацията, потребителите могат да си създават свои сметки (акаунти), за които си задават име, текущ баланс, тип и цвят. След създаването му се появява “карта” с информацията за акаунта. Потребителят по всяко време може да ъпдейтне/изтрие вечесъздаден акаунт.

**AccountController.cs**

**[HttpPost]**

**[Route("create")]**

**public async Task<IActionResult> CreateAccount([FromBody] AccountDTO accountDto)**

**{**

**User currentUser = await \_userService.GetCurrentUserAsync();**

**if (!\_accountService.Validate(accountDto))**

**{**

**...**

**return BadRequest();**

**}**

**Account account = await \_accountService.GetAccountForCurrentUserByNameAsync(accountDto.Name!, currentUser);**

**if (account is not null)**

**{**

**...**

**return BadRequest();**

**}**

**currentUser.TotalBalance += accountDto.Balance;**

**await \_accountService.CreateAccountAsync(new Account()**

**{**

**UserID = currentUser.ID,**

**User = currentUser,**

**Name = accountDto.Name,**

**Balance = accountDto.Balance,**

**Currency = accountDto.Currency,**

**Color = accountDto.Color,**

**DateCreated = DateTime.Now,**

**AccountType = (AccountType)Enum.Parse(typeof(AccountType), accountDto.AccountType!),**

**});**

**await \_notificationService.CreateNotificationAsync(new Notification()**

**{**

**Title = $"Successfully created Account '{accountDto!.Name}'!",**

**Description = $"Your account '{accountDto!.Name}' has been successfully created and is now ready to use.",**

**NotificationType = NotificationType.Success,**

**IsRead = false,**

**User = currentUser,**

**UserID = currentUser.ID**

**});**

**return Ok(account);**

**}**

Методът CreateAccount позволява на потребителя да създаде нова финансова сметка (Account). Първо се извлича текущият потребител чрез ***\_userService.GetCurrentUserAsync()***, след което се валидират въведените от него данни.

Ако данните не са коректни (напр. липсва име или балансът е отрицателен), се създава уведомление за грешка и се връща BadRequest().

След това се проверява дали вече съществува сметка със същото име за този потребител. Ако има, отново се изпраща уведомление за грешка и заявката се прекратява.

При успешно преминаване на валидацията, балансът на потребителя се увеличава, създава се новата сметка, и се изпраща уведомление за успешно създаване.

**GetAccounts.tsx**

**const GetAccounts = () => {**

**const [filter, setFilter] = useState<CompositeFilterDescriptor>({**

**logic: "and",**

**filters: [],**

**});**

**const { data, isLoading, isError, error } = useGetAccounts();**

**const handleFilterChange = (*newFilter*: CompositeFilterDescriptor) => {**

**setFilter(*newFilter*);**

**};**

**if (isLoading) return <p>Loading...</p>;**

**if (isError) return <p>Error: {error!.message}</p>;**

**return (**

**<>**

**<AccountFilter *onFilterChange*={handleFilterChange} /> <br />**

**<div *className*="flex flex-col lg:flex-row gap-4">**

**<div *className*="flex-grow">**

**<div *className*="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-5 gap-4">**

**<AccountCards *accounts*={filterBy(data!, filter)} />**

**</div>**

**</div>**

**</div>**

**</>**

**);**

**};**

**export default GetAccounts;**

Компонентът **GetAccounts** отговаря за визуализирането на всички създадени сметки на потребителя. Използва се custom hook (useGetAccounts) за извличане на данните от бекенда, като междувременно се показва съобщение за зареждане или грешка, ако е необходимо.

Компонентът също така включва функционалност за филтриране чрез **AccountFilter**, като избраните филтри се съхраняват в локалното състояние. След това сметките се визуализират в мрежа чрез AccountCards, като се прилагат избраните филтри с помощта на **filterBy**.

На фигура 5 е показана последователността на действия, които се извършват когато потребителят си създаде акаунт.

A screenshot of a computer screen

AI-generated content may be incorrect.

Фигура 5. Последователност от действия при създаването на акаунт

**Създаване на категории**

След като създаде акаунт, потребителят може да навигира към страницата “Категории” и да си създаде съответната категория, за която си задават име, текущ бюджет, цвят и дали е доходоносна.

Създаването на Категории е аналогично също като създаването на Акаунти, кодът за имплементация е подобен на Акаунтите и може да бъде разгледан тук:

<https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeBackend/Controllers/CategoryController.cs#L19> – backend

<https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeFrontend/src/Components/CategoriesPage/CRUD/IncomeCategories.tsx#L8> – frontend

**Създаване на транзакции**

След като създаде акаунт и категории, потребителят може да навигира към страницата “Транзакции” и да си създаде нова транзакция, за която си задават име, текущ бюджет, цвят и дали е доходоносна.

**TransactionController.cs**

**[HttpPost]**

**[Route("create")]**

**public async Task<IActionResult> CreateTransaction([FromBody] TransactionDTO transactionDto)**

**{**

**User currentUser = await \_userService.GetCurrentUserAsync();**

**if (!\_transactionService.Validate(transactionDto))**

**{**

**await \_notificationService.CreateNotificationAsync(new Notification()**

**{**

**Title = "Please fill in the Transaction fields!",**

**Description = $"The Transaction with name '{transactionDto.Name}' must have valid properties",**

**NotificationType = NotificationType.Error,**

**IsRead = false,**

**User = currentUser,**

**UserID = currentUser.ID**

**});**

**return BadRequest();**

**}**

**Category category = await \_categoryService.GetCategoryForCurrentUserByNameAsync(transactionDto.CategoryName!, currentUser);**

**Account account = await \_accountService.GetAccountForCurrentUserByNameAsync(transactionDto.AccountName!, currentUser);**

**await \_transactionService.UpdateUserBalanceAsync(true, transactionDto, null, currentUser, category, account);**

**await \_transactionService.CreateTransactionAsync(new Transaction()**

**{**

**Name = transactionDto.Name,**

**Amount = transactionDto.Amount,**

**UserID = currentUser.ID,**

**User = currentUser,**

**AccountID = account.ID,**

**AccountName = account.Name,**

**Account = account,**

**CategoryID = category.ID,**

**Category = category,**

**CategoryName = category.Name,**

**Color = category.Color,**

**IsRepeating = transactionDto.IsRepeating,**

**NextRepeatDate = transactionDto.IsRepeating ? DateTime.UtcNow.AddMonths(1) : null**

**});**

**await \_notificationService.CreateNotificationAsync(new Notification()**

**{**

**Title = $"Transaction {transactionDto.Name} created successffuly",**

**Description = $"Your transaction '{transactionDto.Name}' has been successfully created!",**

**NotificationType = NotificationType.Success,**

**IsRead = false,**

**User = currentUser,**

**UserID = currentUser.ID**

**});**

**return Created();**

Методът **CreateTransaction** се използва за създаване на нова транзакция от потребителея. Първо се извършва валидация на подадените данни чрез ***\_transactionService***, като при невалидна транзакция се връща грешка и се създава съответното известие. След това се извличат категорията и акаунтът по име, към които ще бъде обвързана транзакцията.

След успешно извличане се актуализира общият баланс на потребителя чрез сървис метода ***UpdateUserBalanceAsync***, и се създава нов обект от тип **Transaction**, който се записва в базата данни.

Ако транзакцията е отбелязана като повтаряща се, се задава и дата за следващо повторение. Накрая потребителят получава известие за успешно създаване и се връща Created статус към клиента.

**GetTransactions.tsx**

**const GetTransactions = () => {**

**const [filter, setFilter] = useState<CompositeFilterDescriptor>({**

**logic: "and",**

**filters: [],**

**});**

**const { data, isLoading, isError, error } = useGetTransactions();**

**const handleFilterChange = (*newFilter*: CompositeFilterDescriptor) => {**

**setFilter(*newFilter*);**

**};**

**if (isLoading) return <p>Loading...</p>;**

**if (isError) return <p>Error: {error?.message}</p>;**

**return (**

**<>**

**<TransactionFilter *onFilterChange*={handleFilterChange} />**

**<div *className*="flex flex-col lg:flex-row gap-4">**

**<div *className*="flex-grow">**

**<div *className*="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-5 gap-4">**

**<TransactionCards *transactions*={filterBy(data?.incomeTransactions!, filter)} />**

**<TransactionCards *transactions*={filterBy(data?.expenditureTransactions!, filter)} />**

**</div>**

**</div>**

**</div>**

**</>**

**);**

**};**

**export default GetTransactions;**

Компонентът **GetTransactions** се използва за визуализиране на всички транзакции, регистрирани от потребителя. Данните се зареждат чрез hook-а ***useGetTransactions***.

Потребителят може да прилага филтри чрез **TransactionFilter**, а резултатите се визуализират с компонента **TransactionCards**, като се разделят по тип – приходи (incomeTransactions) и разходи (expenditureTransactions).

Може да разгледате пълния код как са имплементирани транзакциите тук:

* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeFrontend/src/Components/TransactionsPage/CRUD/GetTransactions.tsx#L7> - Frontend
* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeBackend/Controllers/TransactionController.cs#L21> - Backend

На фигура 6 е показана последователността на действия, които се извършват когато потребителят си създаде акаунт.

A screenshot of a diagram

AI-generated content may be incorrect.

Фигура 6. Последователност от действия при създаването на транцакция

**Интеграция на AI асистент**

В проекта е реализирана AI функционалност, която предоставя финансови препоръки на потребителите въз основа на техните данни. За целта е използван GPT модел чрез външно API, като интеграцията е реализирана в два основни етапа:

* **Backend:** Сървърът подготвя потребителските данни и правила, генерира prompt и го изпраща към GPT4-о, след което връща отговора обратно.
* **Frontend:** Потребителят въвежда въпрос чрез интерфейс, получава отговор от AI асистента и го вижда във визуализиран формат.

**FinancialRecommendationController.cs**

**ICollection<Transaction> transactions = \_transactionService.GetTransactionsFromSpecifiedDate(currentUser.Transactions, parsedDate);**

**JsonSerializerOptions options = new JsonSerializerOptions { WriteIndented = true };**

**string accountsJson = JsonSerializer.Serialize(currentUser.Accounts, options);**

**string categoriesJson = JsonSerializer.Serialize(currentUser.Categories, options);**

**string transactionsJson = JsonSerializer.Serialize(transactions, options);**

**string allData = string.Join(",", accountsJson, categoriesJson, transactionsJson);**

**string rules = @$"I want your response to always be in the scope of the project, that is Financial Recommendations, any other out-of-scope prompts need to turneed down with a sentence like 'This request is out of scrope for me',**

**start with something like 'Recommendations based on data from {parsedDate.ToShortDateString()} to today (or similar sentences).";**

**string textTweaks = @$"Please respond in short plain text (maximum of 3-4 sentences only), without any formatting, such as bold text, dashes, slashes, numbering, ordered/unordered list, or special characters.";**

**string prompt = string.Join(" ", promptRequestData.Prompt!, allData, rules, textTweaks).Trim();**

**GPTResponseDTO gPTResponse = await \_gPTService.Ask(prompt, currentUser);**

От транзакциите на текущия потребител се взимат само тези, направени след определена дата. След това акаунтите, категориите и филтрираните транзакции се сериализират в JSON. Към тези данни се добавят инструкции (като например отговорите да бъдат кратки и свързани само с финансови препоръки). Всичко това се комбинира в един prompt, който по-късно ще бъде подаден към GPT4-о.

**GPTService.cs**

**private readonly IConfiguration \_configuration = configuration;**

**public async Task<GPTResponseDTO> Ask(string prompt, User currentUser)**

**{**

**string apiKey = \_configuration.GetSection("Appsettings:OpenAIAPIKEY").Value!;**

**string model = \_configuration.GetSection("Appsettings:Model").Value!;**

**ChatClient chatClient = new ChatClient(model, apiKey);**

**ClientResult<ChatCompletion> completion = await chatClient.CompleteChatAsync(prompt);**

**string response = completion.Value.Content[0].Text.Trim();**

**return new GPTResponseDTO**

**{**

**Response = response**

**};**

**}**

От конфигурационен файл се взимат API ключът и моделът. С тях се създава ChatClient, който подава prompt-а към GPT. Полученият отговор се връща като обект от тип GPTResponseDTO, съдържащ текста на генерирания отговор.

**CreateFinancialRecommendations.tsx**

**const CreateFinancialRecommendations = () => {**

**const queryClient = useQueryClient();**

**const [promptRequestData, setPromptRequestData] = useState<PromptRequestData>({**

**prompt: "",**

**dateString: `${new Date().getDate()}-${new Date().getMonth() + 1}-${new Date().getFullYear()}`**

**});**

**const handleDateChange = (*e*: DatePickerChangeEvent) => {**

**const dateString: string = `${*e*.value!.getDate()}-${*e*.value!.getMonth() + 1}-${*e*.value!.getFullYear()}`;**

**setPromptRequestData({**

**...promptRequestData,**

**dateString: dateString**

**});**

**}**

**const createRecommendations = async () => {**

**const response = await axios**

**.post(createFinancialRecommendationEndPoint, promptRequestData, { withCredentials: true })**

**.catch((*error*: AxiosError) => {});**

**return response?.data as FinancialRecommendation;**

**}**

**const { mutateAsync } = useMutation({**

**mutationFn: createRecommendations,**

**onSuccess: () => {**

**queryClient.invalidateQueries({ queryKey: ["recommendations"] });**

**},**

**});**

**const generateFinancialRecommendation = async (*prompt*: string): Promise<FinancialRecommendation> => {**

**setPromptRequestData({**

**...promptRequestData,**

**prompt: *prompt*,**

**});**

**const newRecommendation = await mutateAsync();**

**return newRecommendation;**

**}**

**return (**

**<>**

**<div *className*="mb-3">**

**<DatePicker *width*={200} *max*={new Date()} *onChange*={handleDateChange} *className*='m-6' />**

**</div>**

**<div>**

**<AIPromptComponent *generateFinancialRecommendation*={generateFinancialRecommendation} />**

**</div>**

**</>**

**)**

**}**

**export default CreateFinancialRecommendations;**

**CreateFinancialRecommendations.tsx** служи като родителски компонент, който управлява логиката по създаване на AI препоръки. Той пази локално състояние с дата и prompt, позволява избор на дата чрез DatePicker и при натискане на бутон или избор на предложение, изпраща заявка към backend-а чрез **axios**. След успешна заявка, автоматично се обновяват свързаните данни с **useQueryClient**.

**AIPromptComponent.tsx**

**const AIPromptComponent = ({ *generateFinancialRecommendation* }: { generateFinancialRecommendation: (*prompt*: string) => Promise<FinancialRecommendation> }) => {**

**const { data, isLoading, isError, error } = useGetPromptSuggestions();**

**const [activeView, setActiveView] = useState<string>(promptViewDefaults.name);**

**const [outputs, setOutputs] = useState<AIPromptOutputInterface[]>([]);**

**const handleActiveViewChange = (*viewName*: string) => {**

**setActiveView(*viewName*);**

**}**

**const handlePromptRequest = (*prompt*?: string) => {**

**if (!*prompt*) {**

**return;**

**}**

**const recommendation = generateFinancialRecommendation(*prompt*);**

**recommendation.then((*data*) => {**

**setOutputs([{**

**id: *data*.id,**

**title: *prompt*,**

**responseContent: *data*?.responseContent!,**

**prompt,**

**},**

**...outputs,**

**]);**

**setActiveView(outputViewDefaults.name);**

**}).catch((*error*) => {**

**console.log(*error*);**

**});**

**}**

**if (isLoading) return <p>Loading...</p>;**

**if (isError) return <p>Error: {error!.message}</p>;**

**return (**

**<>**

**<AIPrompt *className*="px-5 w-70 sm:w-90 md:w-139 shadow-sm" *activeView*={activeView} *onActiveViewChange*={handleActiveViewChange} *onPromptRequest*={handlePromptRequest}>**

**<AIPromptView *promptSuggestions*={data} />**

**<AIPromptOutputView *outputs*={outputs} *showOutputRating*={true} />**

**</AIPrompt>**

**</>**

**);**

**}**

**export default AIPromptComponent;**

**AIPromptComponent.tsx** е отговорен за визуалната част на AI асистента. Той показва списък с предложени prompt-и, позволява потребителят да избере или въведе свой, изпраща го към родителския компонент чрез **generateFinancialRecommendation**, и след това визуализира получения отговор под формата на изход, който може да се оценява или преглежда повторно.

На фигура6 е показано визуално как изглежда имплементацията на AI асистента (AIPrompt)

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Фигура 6. Визуализация на AI Асистента (AIPrompt)

На фигура 7 е представена последователност от действия при интеракция на AIPrompt компонент с OpenAI API ва страницата Financial Recommendation

A diagram of a software program

AI-generated content may be incorrect.

**Фигура 7. Последователност от действия при интеракция на AIPrompt компонент с OpenAI API ва страницата Financial Recommendation**

Може да разгледате пълния код как са имплементирани транзакциите тук:

* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeFrontend/src/Components/RecommendationsPage/CreateFinancialRecommendations.tsx#L9> - Frontend
* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeFrontend/src/Components/RecommendationsPage/AIPromptComponent.tsx#L6> - Frontend
* <https://github.com/P1l3T0/FinEdge/blob/master/FinEdgeBackend/Controllers/FinancialRecommendationController.cs#L13> - Backend

**Глава 5: Ръководство за използване**

**Стъпки за локална инсталация**

Предварително трябва да имате инсталирани [SQL Server Management Studio](https://learn.microsoft.com/en-us/ssms/download-sql-server-management-studio-ssms), [.NET 9.0](https://dotnet.microsoft.com/en-us/download) и [Vite](https://vite.dev/guide/), и да имате [API key за OpenAI](https://www.youtube.com/watch?v=SzPE_AE0eEo)

1. Клонирайте репото от GitHub (<https://github.com/P1l3T0/FinEdge>)
2. Отворете FinEdge.sln файла и билднете проекта (F6), за да инсталирате нужните Nuget пакети
3. Направете локална база данни в SQL Server Management Studio и ѝ вземете connection string-a и го сложете в appsettings.json в DefaultConnection (изгледайте [това видео](https://youtu.be/EmV_IBYIlyo?list=PL82C6-O4XrHdiS10BLh23x71ve9mQCln0&t=62) от 1:00 до 5:20 ако се затруднявате със сетъпването на базата)
4. Отворете Package Manager конзолата и напишете Add-Migration initialCreate, изчакайте да се създадат миграциите и след това напишете Update-Database
5. За да ползвате AI интеграцията, навигирайте към **.env** файла и сложете вашия OpenAI API ключ на съответното място
6. Отворете основната директория (C:\Users\[USERNAME]\Desktop\FinEdge) във Visual Studio Code (може да го направите от терминала/powershell/git bash)
7. Влезте във папката FinEdgeFrontend (cd FinEdgeFrontend) и напишете **npm i**, за да инсталирате нужните пакети (ако даде грешка за version mismatch на някои пакети, напишете **npm i –force**)
8. След като пакетите са инсталирани, стартирайте сървъра и напишете **npm run dev** за да стартирате фронтенда
9. A screen shot of a computer program

   AI-generated content may be incorrect.A screenshot of a computer

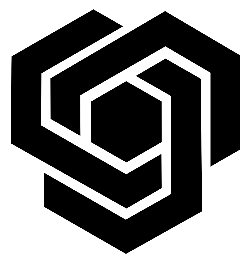
   AI-generated content may be incorrect.Накрая в конзолата в която сте писали комманди трябва да се появи локалния адрес на който да влезете

**Заключение**

В заключение, разработката на системата Finedge демонстрира завършен и практичен подход към изграждането на съвременна уеб платформа за управление на лични финанси. Интеграцията на AI асистент, използващ OpenAI, добавя интелигентна стойност към приложението чрез персонализирани финансови препоръки, базирани на реални данни и потребителско поведение.

Проектът е създаден с внимание към добрите практики, скалируемост и модулност, което го прави отлична основа за бъдещо развитие. Сред планираните надграждания е създаването на мобилно приложение чрез **React Native**, което ще разшири достъпността и ще подобри потребителското изживяване на различни устройства. Освен това се предвижда **пускане на продукта на пазара** като реална услуга за крайни потребители, което би предоставило допълнителна стойност на хората, търсещи ефективен начин за управление на личните си финанси. В бъдеще също могат да бъдат добавени и нови функционалности, които да помагат още повече на крайния потребител.

С реализирането на проекта придобих ценен практически опит, задълбочих знанията си по уеб разработка и архитектура на софтуерни системи и изградих реално приложение, което има потенциал да бъде полезно и приложимо в ежедневието на много хора.

**ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

Факултет Приложна математика и информатика

**ДЕКЛАРАЦИЯ**

**ЗА АВТОРСТВО НА ДИПЛОМНА РАБОТА**

Долуподписаният: Васил Паскалев Ганев

специалност : Информатика и софтуерни науки фак. № 471221010

**ДЕКЛАРИРАМ:**

Представената от мен дипломна работа на тема: FinEdge - приложение за разпределяне на личните финанси, интегрирано с AI асистент е лична моя авторска разработка, резултат от собствени изследвания.

Потвърждавам, че тя в нейната цялост и отделни части не е била използвана за придобиване на образователна и/или научна степен в ТУ - София или в други университети.

Формулировки, идеи и текстове, взети от други източници, са цитирани точно и с коректно посочване на техните автори. Дипломната работа не е публикувана на друго място.

Декларирам, че предоставям правото на Факултет Приложна

математика и информатика при ТУ - София, съгласно процедурите и правилниците на университета, да архивира и съхранява тази дипломна работа с цел доказване във времето на моето авторство.

Дата: 20.06.2025г. Дипломант:

София

**ИЗПОЛЗВАНА ЛИТЕРАТУРА**

* [C#](https://en.wikipedia.org/wiki/C_Sharp_(programming_language))
* [.NET Core Web API](https://dotnet.microsoft.com/en-us/apps/aspnet/apis)
* [Entity Framework Core](https://learn.microsoft.com/en-us/ef/core/)
* [JWT (JSON Web Token)](https://en.wikipedia.org/wiki/JSON_Web_Token)
* [Swashbuckle](https://swagger.io/tools/swagger-ui/)
* [Hangfire](https://docs.hangfire.io/en/latest/)
* [OpenAI Developer API](https://platform.openai.com/docs/overview)
* [SQL Server Management Studio (SSMS)](https://learn.microsoft.com/en-us/ssms/sql-server-management-studio-ssms)
* [Vite](https://vite.dev/guide/)
* [TypeScript](https://www.typescriptlang.org/)
* [React](https://react.dev/)
* [React Router DOM](https://reactrouter.com/home)
* [TanStack Query (React Query)](https://tanstack.com/query/latest/docs/framework/react/overview)
* [Axios](https://axios-http.com/docs/intro)
* [KendoReact](https://www.telerik.com/kendo-react-ui/components/introduction)
* [Tailwind](https://v2.tailwindcss.com/docs)