ЧАСТНА ПРОФЕСИОНАЛНА ГИМНАЗИЯ ЗА ДИГИТАЛНИ НАУКИ „СОФТУНИ БУДИТЕЛ“, гр. София

**ДИПЛОМЕН ПРОЕКТ**

На Александър Ивов Боев

ученик от XII а клас

професия- код: **481030**, **“Приложен програмист”**

специалност- код: **4810301**, **“Приложно програмиране”**

**Тема: Разработване на софтуер за управление на болници "HealthEdge"**

**Ръководител-консултант:** Мариян Йорданов

Сесия: май-юни 2024г.

Дата: 29.04.2024

**СЪДЪРЖАНИЕ**

[1. Увод 3](#_Toc165301812)

[**1.1.** **Контекст и значимост на здравеопазването в България** 3](#_Toc165301813)

[**1.2.** **Проблеми и предизвикателства в сектора на здравеопазването** 4](#_Toc165301814)

[**1.4.** **Цел и обхват на проекта** 5](#_Toc165301815)

[**1.5.** **Предимства за потребителите на софтуера** 7](#_Toc165301816)

[2. Изложение 11](#_Toc165301817)

[**2.1.** **Използвани технологии** 11](#_Toc165301818)

[**2.1.1.** **Backend** 11](#_Toc165301819)

[**2.1.1.1.** **Какво е ORM и как функционира в Entity Framework?** 12](#_Toc165301820)

[**2.1.1.2.** **Защо C# е добър при разработката на уеб приложения?** 16](#_Toc165301821)

[**2.1.2.** **Frontend** 18](#_Toc165301822)

[**2.1.2.1.** **React и защо да използваме него?** 18](#_Toc165301823)

[**2.1.2.1.1.** **Компонентите в React** 19](#_Toc165301824)

[**2.1.2.1.2.** **Service-и в React** 21](#_Toc165301825)

[**2.2.** **Проектиране и имплементация** 23](#_Toc165301826)

[**2.2.1.** **Архитектура на системата** 24](#_Toc165301827)

[**2.2.2.** **Имплементация на системата** 25](#_Toc165301828)

[**2.2.2.1.** **Имплементация на Backend системата** 25](#_Toc165301829)

[**2.2.2.1.1.** **Инициализация на базата и конфигурация** 25](#_Toc165301830)

[**2.2.2.1.2.** **Модели** 28](#_Toc165301831)

[2.2.2.1.3. Repositories 44](#_Toc165301832)

[2.2.2.1.4. Контролери 65](#_Toc165301833)

[2.2.2.2. Имплементация на Frontend системата 68](#_Toc165301834)

[2.2.2.2.1. Компоненти 68](#_Toc165301835)

[**2.2.2.2.2.** **Services в “HealthEdge”** 75](#_Toc165301836)

[**2.2.2.2.3.** **Routing в „HealthEdge”** 77](#_Toc165301837)

[**2.2.2.2.4.** **Валидациите** 79](#_Toc165301838)

[3. Ръководство за потребителя. 80](#_Toc165301839)

[**3.1.** **Swagger** 80](#_Toc165301840)

[**3.2.** **Frontend** 83](#_Toc165301841)

[4. Заключение 85](#_Toc165301842)

[5. Информационни източници 86](#_Toc165301843)

[6. Приложения 87](#_Toc165301844)

# Увод

С "HealthEdge" се цели да бъде създадена среда, където взаимодействието между пациенти и медицински специалисти е улеснено, прозрачно и достъпно, независимо от времето и местоположението. Ангажиментът към иновациите и подобряването на здравеопазването, води към разработването на решение, което не само отговаря на съвременните предизвикателства в бранша, но и предоставя стабилна основа за бъдещо развитие и интеграция на нови функционалности.

От друга страна, "HealthEdge" предлага на лекарите и медицинския персонал удобен инструмент за управление на тяхната дейност, включително преглед на заетост, планиране на задълженията и проследяване на пациентската история. Това дава възможност за по-ефективно разпределение на ресурсите и подобряване на качеството на медицинското обслужване.

## **Контекст и значимост на здравеопазването в България**

В контекста на българската здравна система, здравеопазването заема централно място със своята роля за поддържане на общественото здраве и благополучие. Секторът на здравеопазването представлява комплексно поле, което непрекъснато изисква модернизация и иновации за подобрение на услугите и управлението на здравните заведения. Тези усилия са жизненоважни за подобряване на качеството на медицинското обслужване, което влияе директно на здравето и продължителността на живота на населението.

## **Проблеми и предизвикателства в сектора на здравеопазването**

Намирането или предлагането на качествени лекарски услуги може да бъде времеемко и не винаги лесна задача. За да бъде полезно посещението и евентуалното лечение за пациента, той трябва първо да намери достатъчно квалифициран лекар, второ да намери начин да запази час за преглед, трето да съобрази евентуално своя график с графика на лекаря, както и евентуално дали специалистът работи с НЗОК и/или с някой здравноосигурителен фонд. Всяка една от тези нужди, без наличието на “HealthEdge” биха били изключително трудни и ненужно дълги за постигане. В един такъв хипотетичен сценарий, пациентът трябва първо да получи отнякъде контакт на достатъчно квалифициран специалист. След това трябва да намери начин да установи контакт с него, което най-често при старото поколение лекари, които отказват да се модернизират, се случва като отиде на място, изчаква всички хора, които са на опашка преди него и накрая да се разбере с лекаря, обикновено с компромиси от двете страни. Точно от такъв род проблеми решава “HealthEdge”. От друга страна, високо квалифицираните лекари, които искат да помагат на пациентите, биха били обречени на дълги години популяризиране, тъй като методът “от уста на уста” не е най-ефективното решение за популяризиране на лекар. “HealthEdge” отново може да реши този проблем. Всеки лекар има възможността да се представи накратко, както и да разкаже по-подробно за образованието си, за квалификацията си, както и кратка биография. Също така като доказателство за квалитета на лекаря са отзивите от вече посетилите пациенти, които могат да споделят повече за опита си при този лекар, както и да споделят градивна критика към лекуващия.

* 1. **Важността на софтуера за управление на болници**

Уеб приложението "HealthEdge" предлага решения на изброените проблеми чрез интеграция на съвременни технологии, които улесняват диагностиката, лечението и управлението на здравни заведения. Платформата автоматизира множество процеси, което позволява на медицинските специалисти да се съсредоточат повече върху пациентското обслужване отколкото на административни задачи.

## **Цел и обхват на проекта**

Целите на дипломния проект „HealthEdge” са да бъде разработена уеб базирана платформа за управление на лечебни заведения, която да предоставя на потребителите бърз и ефективен начин за намиране на лекар, базирано на оплакванията на пациента.

Специфичните цели на проекта включват следното:

* Разработване на потребителски интерфейс, който да е удобен, лесен за използване и да предоставя ясна информация за всеки лекар, който е наличен в даденото лечебно заведение.

* Интегриране на търсачка, която чрез филтриране на зададени характеристики от потребителя, като например пол, възраст, хронични заболявания, придружаващи заболявания, оплаквания и т.н., да предоставя на потребителите най-добрите специалисти, на бази филтрите и наличните лекари.

* Интегриране на търсачка, чрез която пациентите ще могат да намерят медицинско заведение. Филтрите, чрез които пациентите ще могат да постигнат намирането на медицинско заведение са: **Тип клиника, Населено място или по име на медицинско заведение (опционално)**.

* Интегриране на функционалност, чрез която пациентите ще могат да запазят час при избрания от тях лекар бързо и лесно, без нужда от допълнителна комуникация със специалиста.
* Създаване на AI модел, който ще **анализира** и изпраща информация към МЗ (Министерство на здравеопазването) за **количество заболели, техните диагнози и региона на пациентите** на месечна база, както на тримесечие и на годишна основа. Целта на модела е да помогне на националното здравеопазване за опазването на общественото здраве.

***(Планирано)***

* Интегриране на система за онлайн плащания, която да позволява на пациентите да заплащат своите прегледи, както и назначените медикаменти директно през уебсайта, без посредник.

***(Планирано)***

* Разработване на система за управление на приложението, която да позволява на администраторите на сайта да упражняват мониторинг и администрация на запазените часове, с цел избягването на спам и недостоверни прегледи, както и  да проследяват статуса на прегледите и плащанията

***(Планирано)***

* Интегриране на чат система, чрез която ще се подобри комуникацията между пациент и лекар. Пациентът ще може да обменя с лекаря информация без нуждата от физическо посещение при специалист.

***(Планирано)***

* Модифициране на архитектурата на “HealthEdge”. Проектът да бъде изграден като **microservice architecture,** но при условие, че проекта се разрасне, съответно започнат да работят голям брой от хора, за да има смисъл от архитектурата. Това е така, тъй като микросървисната архитектура е скъпа за изграждане и разработка и има смисъл от нея за голям екип например. Това е така, тъй като осигурява независимост на екипа. Всеки domain или function може да бъде самостоятелно поддържан от назначен екип.

***(Бъдещо развитие)***

## **Предимства за потребителите на софтуера**

Потребителите на "HealthEdge" ще извлекат значителни ползи, включително по-бърз достъп до пациентски данни, подобрена координация между отделите, намаление на времето за обработка на медицинска информация и повишаване на ефективността при лечението. Софтуерът ще допринесе за намаляване на административните разходи и оптимизация на ресурсите в болницата.

* 1. **Предимства за потребителите на софтуера "HealthEdge" в контекста на намиране и предлагане на лекарски услуги**

Софтуерът "HealthEdge" предоставя множество предимства, които адресират гореспоменатите предизвикателства:

1. **Достъп до информация за лекари:** "HealthEdge" позволява на лекарите да представят своята квалификация, образование и специализации чрез детайлни профили. Това улеснява пациентите в избора на подходящ специалист посредством преглед на образователен и професионален фон.
2. **Електронно запазване на часове:** Платформата предлага функции за електронно запазване на часове, които минимизират необходимостта от телефонни обаждания и лични посещения.
3. **Оценки и отзиви:** Пациентите могат да оставят отзиви за своите преживявания с различни лекари, което помага на други потребители да направят информиран избор, базиран на качеството на медицинското обслужване.
4. **Синхронизация на графици:** Платформата позволява съобразяване на личния график с този на лекарите, облекчавайки процеса на планиране и намалявайки времето за чакане.

Тези функционалности на "HealthEdge" не само улесняват процесите на намиране и запазване на час при лекар, но и подобряват общото качество на медицинското обслужване, като правят здравната система по-достъпна, ефективна и отзивчива към нуждите на пациентите.

* 1. **Възможности за иновации и развитие на здравеопазването в България**

Здравеопазването в България, подобно на много други страни, стои на прага на технологична трансформация, която предлага значителни възможности за иновации и развитие. Софтуерът за управление на болници "HealthEdge" може да играе ключова роля в тази промяна, като внедри нови технологии и процеси, които значително подобряват ефективността, качеството и достъпността на медицинските услуги. Някои от основните направления за иновации включват:

1. **Интегриране на изкуствен интелект (AI) и машинно обучение (Machine learning):**
   * **Диагностика и лечение:** AI може да анализира медицински данни за предоставяне на предложения за диагностика и лечение, ускорявайки тези процеси и повишавайки тяхната точност.
   * **Персонализирана медицина:** Машинното обучение може да помогне за разработването на персонализирани лечебни планове базирани на индивидуалните характеристики на всеки пациент, като генетични данни и здравна история.
2. **Внедряване на блокчейн технологии:**
   * **Сигурност на данните:** Блокчейн може да предостави сигурна и непроменима база данни за медицинските записи, което минимизира риска от фалшификации и злоупотреби.
   * **Транспарентност в сделките:** Технологията може да се използва за проследяване и верификация на медицински трансакции и услуги, гарантирайки тяхната прозрачност и надеждност.
3. **Разработка на мобилни здравни приложения:**
   * **Дистанционно мониторинг:** Приложенията могат да позволят на пациентите да следят своето здраве от дома, предоставяйки данни в реално време на лекарите за по-добър надзор и управление на хронични заболявания.
   * **Подобрена комуникация:** Мобилните приложения могат също да улеснят комуникацията между пациенти и медицински специалисти, ускорявайки процеса на консултации и повишавайки удовлетвореността на пациентите.
4. **Умни медицински устройства:**
   * **Диагностични устройства:** Разработката на умни медицински устройства, които автоматично измерват и записват здравни показатели, може да помогне за ранното откриване и предотвратяване на заболявания.
   * **Роботизирана хирургия:** Въвеждането на роботи в хирургическата зала може да доведе до по-малко инвазивни, по-точни и с по-висок успех операции.
5. **Телемедицина:**
   * **Дистанционни консултации:** Развитието на телемедицински услуги позволява на пациенти в отдалечени или недостъпни райони да получават качествени медицински консултации без нужда от физическо присъствие в медицинско заведение.

Тези иновации не само ще подобрят качеството на здравеопазването в България, но също така могат значително да намалят разходите и да увеличат достъпа до медицински услуги за по-широк кръг от населението. "HealthEdge" предоставя платформата, необходима за интеграция на тези технологии, като по този начин спомага за трансформацията на здравната система в страната.

# Изложение

## **Използвани технологии**

В раздела с „използвани технологии“ ще бъдат разгледани технологиите, заемащи централно място в изграждането на „HealthEdge”. Технологиите са подбрани, за да осигуряват на стабилност, сигурност и ефективност на уеб-базираното приложение, като същевременно гарантират на клиента добро потребителско изживяване. Включени са инструменти за съхранение на данни, стилизиране на интерфейса, потребителска и лекарска автентикация, както и още много други. Съчетавайки всички тези инструменти, те осигуряват функционалност и удобство за крайния потребител.

### **Backend**

Първата стъпка в разработката на уеб приложения е създаването на backend частта. Това е именно частта от софтуерната архитектура на дадена уеб или софтуерна система, която обработва заявки и осигурява връзката между frontend-а (потребителския интерфейс) и базата данни. Той обикновено се състои от различни компоненти и технологии, които работят „зад кулисите“, за да осигурят функционалността и данните, нужни на потребителите. Backend-а се грижи за бизнес логиката на приложението, съхраняването и манипулирането на данни, обработката на заявки и осигуряването на сигурност.

A diagram of a software application

Description automatically generated

Фигура 1: Структурата на уеб приложението (технологии)

## **Какво е ORM и как функционира в Entity Framework?**

ORM е начин за конвертиране на данни между релационна база данни и обектно-ориентиран език за програмиране, чрез създаване на база данни с виртуални обекти, която може да се използва от обектно-ориентирания език за програмиране.

В Entity Framework това се случва чрез помощта на модели и миграции - процес на автоматично създаване и актуализиране на релационната база данни на основата на промените в моделите. Това означава, че Entity Framework Core може да проследява промените в модела на данните и да създава, актуализира или премахва таблиците и колоните в базата данни, когато е необходимо.

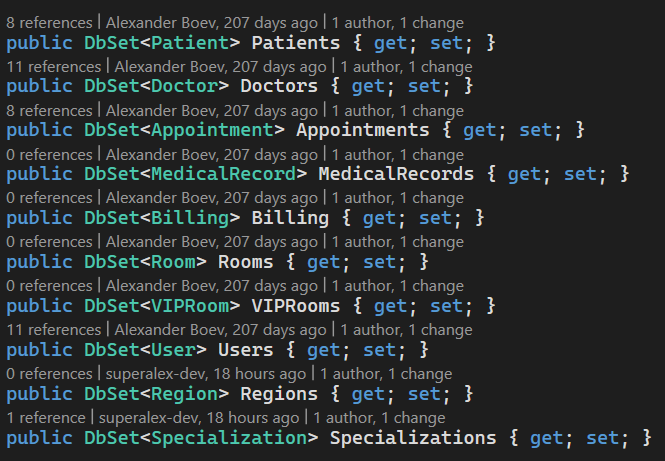
Един модел в Entity Framework представлява клас, който има различни property-та. Те могат да бъдат различни типове, като Entity Framework сам ще ги превърне в типове, които MSSQL server или PostgreSQL да разбере – например тип ***string*** ще бъде превърнат в тип ***nvarchar***, тип ***DateTime*** ще бъде превърнат в ***datetime2,*** тип ***byte*** ще бъде превърнат в тип ***tinyint*** и така нататък.

A screen shot of a computer code

Description automatically generated

Фигура 2: Модел в Entity Framework

Тези модели можем да превърнем в таблици, използвайки т.нар. ***DbSet***, където задаваме и желаното име на таблицата:



Фигура 3: Използване на DbSet в Entity Framework

След това трябва да имаме и конфигурация, която да посочи на Entity Framework как да намери / къде да създаде базата данни, която искаме да създаваме, използваме или модифицираме. Това можем да постигнем чрез метода **OnConfiguring**. Конкретно в случая на “HealthEdge”, connection string-а към базата се намира в application settings (appsettings.json), който бива игнориран от gitignore, чрез което се постига максимална сигурност за връзките в него. След като приложението спре да бъде в режим на development, следващата стъпка, която може да се предприеме, когато приложението е в продукционен режим да се използва криптиране на конфигурационния файл, използвайки **GitCrypt**.

Това е инструмент за криптиране на файлове в **Git** хранилища, позволяващ безопасното съхранение на конфиденциални файлове дори в общодостъпни репозитории.

При инициализация с **git-crypt init**, се създава криптиращ ключ, който се използва за автоматично криптиране на файлове, определени в **.gitattributes** файл. Криптираните файлове се добавят и комитват в хранилището в защитен вид, като могат да бъдат декриптирани само от потребители с достъп до съответния ключ.

Основните предимства на **GitCrypt** включват висока степен на сигурност, прозрачност в криптирането и декриптирането на файлове, улеснена споделянето на ключове чрез GPG, и лесната интеграция в съществуващи проекти. **GitCrypt** осигурява ефективно решение за управление на конфиденциални данни, като същевременно поддържа контрол и гъвкавост в достъпа до тях.

A screen shot of a computer program

Description automatically generated

Фигура 4: Метод OnConfiguring в Entity Framework

Методът **OnModelCreating** е важен за конфигурирането на модела на данните в приложението, като се дефинират взаимоотношенията, външните ключове и поведението при изтриване на записи. Използването на този метод е изключително нужно за поддържането на целостта на данните и правилното взаимодействие между различните същности в базата данни.

A computer screen shot of a program

Description automatically generated

Фигура 5: Метод OnModelCreating в Entity Framework

В този метод се настройват отношенията между **Doctor**, **Patient**, **Specialization**, и **Region** същностите, което позволява на приложението да управлява комплексни данни и взаимоотношения ефективно и безопасно. Той осигурява на **"HealthEdge"** да може да поддържа реалистични и точни медицински записи, което е от съществено значение за неговата функционалност и надеждност.

След като имаме модел и сме решили, че той ще бъде таблица, задавайки го като DbSet, трябва да направим миграция. Миграцията в Entity Framework се извършва с помощта на инструмента за миграции на Entity Framework, който позволява на разработчиците да генерират т.нар. миграционни скриптове и да ги приложат върху базата данни. Този процес включва изграждане на миграционна история, която записва всички промени в модела на данните, като например добавяне или премахване на таблица или колона. Когато се генерира нова миграция, инструмента за миграции автоматично генерира необходимите SQL команди за актуализиране на базата данни.

A screen shot of a computer program

Description automatically generated

Фигура 6: Миграционен скрипт в Entity Framework

## **Защо C# е добър при разработката на уеб приложения?**

Ето някои от основните причини защо **C#** е сигурен избор при разработването на уеб приложения:

* Обектно-ориентиран подход: **C#** е език със строга типизация, който поддържа обектно-ориентирано програмиране. Това позволява на програмистите да създават модулни и структурирани приложения, които са лесни за поддръжка и мащабиране.
* **ASP.NET: ASP.NET** е платформа за създаване на уеб приложения, която използва **C#** като основен език за програмиране. **ASP.NET** предоставя мощни инструменти за създаване на сигурни и ефективни уеб приложения.
* Голяма общност: **C#** е един от най-популярните езици за програмиране, който има голяма и активна общност от разработчици. Това означава, че има много ресурси, като форуми, уроци и документация на разположение.
* Лесно мащабиране: **C#** езикът и **ASP.NET** платформата предлагат много възможности за мащабиране на уеб приложенията. Това означава, че програмистите могат да добавят нови функции и да променят кода на приложението, без да се налага да го изграждат отново.
* Сигурност: **C#** и **ASP.NET** са добре известни със своята сигурност. Например, **ASP.NET** предоставя вградена защита срещу най-различни видове атаки, като SQL инжекции, кръстосан скриптинг и други.

A purple hexagon with a white circle and a white letter

Description automatically generated

## **Frontend**

След като backend-ът е готов, идва ред на реализацията на приложението в уеб среда, за което ни е нужен Frontend. Frontend-ът в едно уеб приложение е частта, която потребителите виждат и с която взаимодействат.

## **React и защо да използваме него?**

React е JavaScript библиотека за изграждане на потребителски интерфейси, особено подходяща за създаването на бързи и интерактивни едностранични приложения (SPA - Single Page Applications). Разработена от Facebook, React предоставя гъвкава и ефективна модел за разработка на уеб и мобилни приложения и включва следните основни характеристики:

* **JSX (JavaScript XML):** JSX е разширение на синтаксиса, което позволява комбинирането на HTML структура с JavaScript логика. Това улеснява разработчиците да описват интерфейси по декларативен начин и с висока четимост.
* **Components (Компоненти):** React е изграден върху идеята за компоненти – изолирани блокове код, които управляват своето собствено състояние. Те позволяват разработчиците да изграждат сложни UI структури, които са лесни за управление и повторна употреба.
* **One-way data binding (Еднопосочно свързване на данни):** В React, данните текат в една посока от родителски към дъщерни компоненти, което улеснява управлението на сложни приложения и подобрява производителността.
* **Virtual DOM (Виртуален DOM):** React използва концепцията за Виртуален DOM, където промените в интерфейса се прилагат първо във виртуална копия на DOM, което минимизира броя на операциите в реалния DOM и подобрява производителността на приложенията.
* **Lifecycle methods (Методи на жизнения цикъл):** Компонентите в React имат множество „жизнени“ методи, които се извикват в определени моменти от жизнения цикъл на компонента (например при създаването, обновяването или унищожаването му). Тези методи предоставят голяма гъвкавост и контрол върху поведението на компонентите.
* **Hooks (Куки):** Въведени във версия 16.8, куките позволяват на функционалните компоненти да използват състояние и други React характеристики без да се налага да са класове. Това прави кода по-кратък и по-лесен за разбиране.

• **Testing (Тестване):** React може да се интегрира с инструменти като Jest за тестване на компоненти в изолация, което улеснява тестването и поддържането на голям мащаб от приложения.

Използването на React в проектите предлага значителни предимства като ефективност при обработка на данни, голяма екосистема от библиотеки и инструменти, и широката общност от разработчици, които непрекъснато допринасят за нейното развитие. Това го прави предпочитан избор за разработка на съвременни уеб приложения.

## **Компонентите в React**

В React, компонентите са основните строителни блокове на приложението. Компонентите са самостоятелни единици, които обхващат HTML, CSS и JavaScript кода и позволяват на разработчиците да създават многократно използваеми UI Компоненти за приложенията си.

A screenshot of a computer

Description automatically generated

Фигура 7: Компоненти в React.js

В конкретния пример от фигура 7, в папката с компонентите, използвани за view-та, свързани с докторите, имаме 3 компонента. Първият – DoctorCard.js е компонент, който се използва за показване на информация за лекар. Той извлича и показва от сървъра най-ранния свободен слот за среща с лекаря. Ето разбивка на неговата функционалност:

Той приема обект на лекар като реквизит. Този обект трябва да съдържа информация за лекаря, като например неговия идентификатор, име, снимкаUrl, специализация, рейтинг и град.

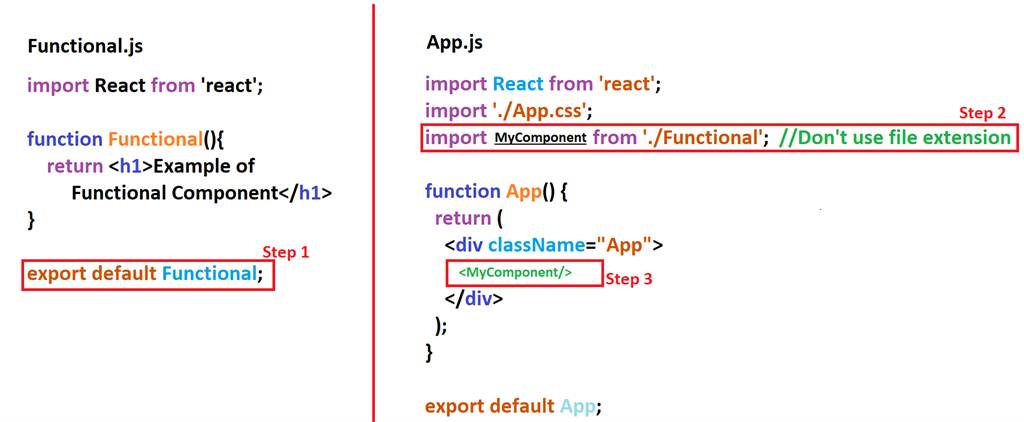
Използва куката useState, за да създаде променлива на състоянието earliestSlot, в която се съхранява най-ранният наличен слот за среща с лекаря.

Куката useEffect се използва за извличане на най-ранния наличен слот от сървъра, когато компонентът се монтира или когато doctor.id се промени. След това изтеглените данни се задават в променливата на състоянието earliestSlot.

Компонентът връща JSX, който показва снимката на лекаря, името, специализацията, рейтингите, града и най-ранния наличен слот за среща. Включва и бутон за резервиране на среща.

StarIcon от @heroicons/react/solid се използва за показване на 5-звездна оценка за лекаря.

Накрая компонентът е експортиран за използване в други части на приложението.



Фигура 8: Пример за принципа на работа на компонентите в React.js

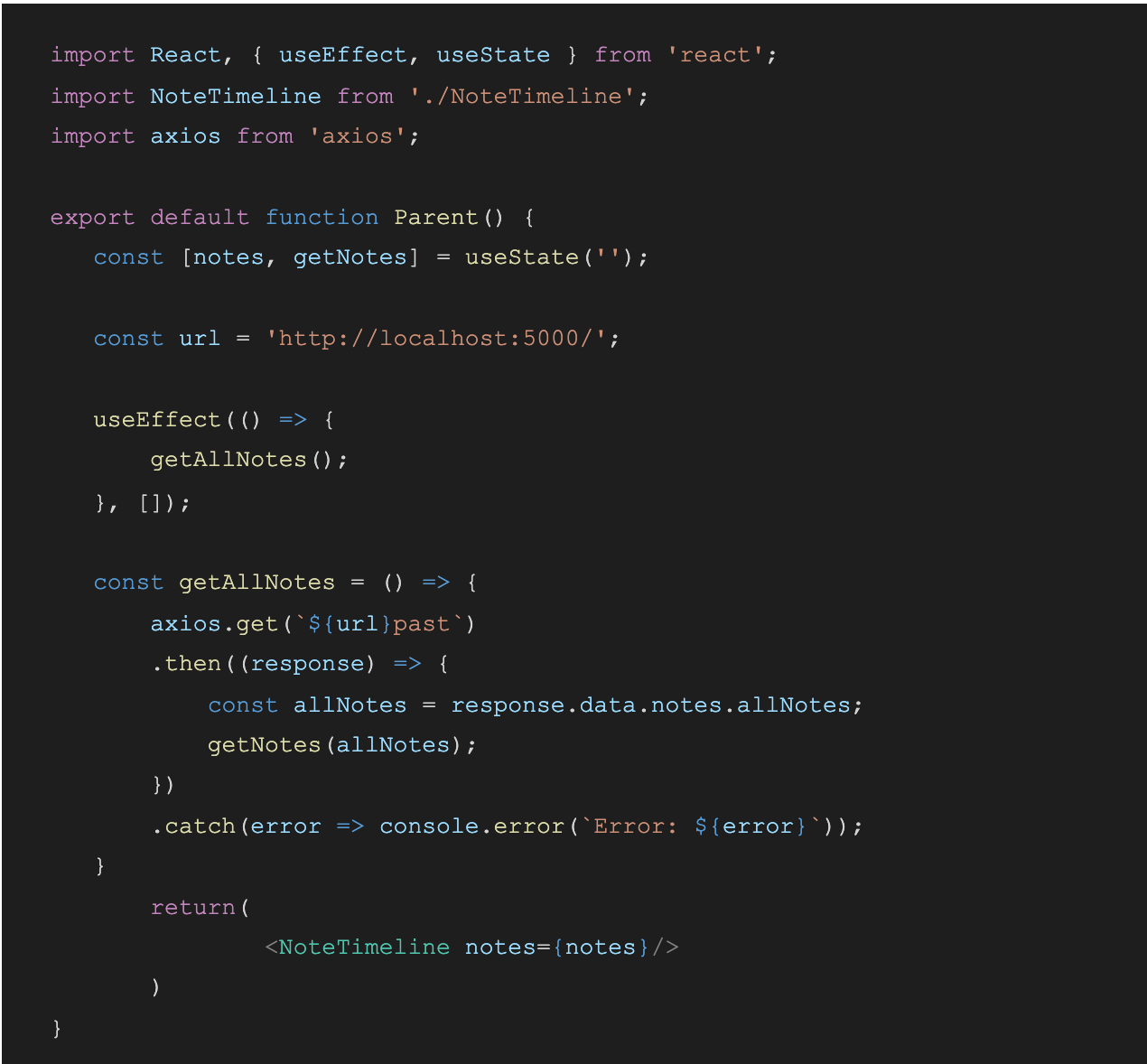
## **Service-и в React**

В React.js, сървисите играят ключова роля за облекчаване на обмена на данни между фронтенда и бекенда на уеб приложението. Ефективното използване на услугите е съществено, тъй като насърчава повторното използване на съществуващия код, което е ключов аспект в оптимизацията в разработката на софтуер.

В типично приложение на React.js, особено когато е интегрирано с бекенд на ASP.NET Core, използващ Entity Framework, услугите се използват за взаимодействие с бекенда чрез HTTP заявки. Ето най-често използваните видове HTTP заявки:

* GET: Извлича информация или данни от сървъра за определен ресурс. Този метод често се използва за получаване на страници, изображения и други файлове от сървъра. В контекста на React.js и ASP.NET Core, заявка GET може да се използва за извличане на данни от база данни чрез операции на Entity Framework, които са предоставени чрез API крайни точки.
* POST: Изпраща данни към сървъра за обработка, обикновено като част от изпращане на формуляр или за създаване на нов ресурс. Например, когато потребител изпраща формуляр в приложение React за добавяне на нов запис в базата данни, заявка POST може да изпрати тези данни към ASP.NET Core API, което след това обработва вмъкването на данните с помощта на Entity Framework.
* PUT: Актуализира съществуващ ресурс на сървъра с нови данни. В интеграцията на React.js с ASP.NET Core, заявка PUT може да се използва за актуализация на запис в базата данни. Бекендът получава новите данни и Entity Framework обработва операцията по актуализация.
* DELETE: Премахва ресурс от сървъра. В сценария, където приложение React трябва да изтрие запис от базата данни, се издава заявка DELETE към ASP.NET Core API, което след това използва Entity Framework за премахване на съответните данни от базата данни.

Разработчиците установяват връзката между фронтенда и бекенда чрез тези заявки, което позволява гладкото протичане на данните и взаимодействието в приложението. Тази структура гарантира, че приложенията на React.js могат ефективно да комуникират с бекенд на ASP.NET Core, използвайки Entity Framework за правилното управление и операции с данни.



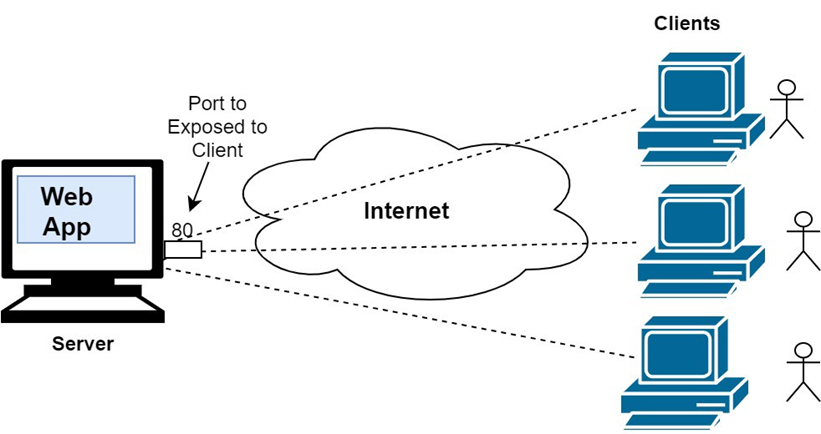
Фигура 9: Пример за service в React.js

## **Проектиране и имплементация**

В тази глава, идва ред на обяснението как е проектирано и имплементирано уеб приложението **“HealthEdge”.** Ще разгледаме логиката в backend-а, data-access layer-a, миграциите, моделите, frontend-а, как се осъществява връзката между клиентската и сървърната част и др.

## **Архитектура на системата**

Архитектурата на проекта е client-server, която изобразена, чрез схема изглежда по следния начин:



Фигура 10: Client-server архитектура

Client-server архитектурата е модел на софтуерен дизайн, който е разделен на два основни компонента: Client (клиент) и server (сървър). Клиентът е софтуер, който изпълнява заявки към сървъра, за да получава информация или, за да извършва операции. Сървърът е софтуер, който отговаря за заявките на клиента и обработката на информация, която е необходима за изпълнението им.

Системата работи като клиентите комуникират със сървъра чрез мрежата, като изпращат заявки за информация или изпълнение на операции. Сървърът получава тези заявки, обработва ги и изпраща отговори (response/s) обратно към клиентите. Това позволява на различни клиенти да работят едновременно с един и същ сървър, като споделят обща информация и ресурси.

Предимствата на този модел включват улеснена съвместна работа и управление на данни, както и по-лесно скалиране на приложенията. В допълнение, този модел предоставя по-голяма сигурност на данните, тъй като те са съхранени централизирано на сървъра и могат да бъдат защитени с подходящи механизми за достъп и сигурност.

## **Имплементация на системата**

В следващите точки и подточки ще бъде показано как е имплементирана система, както в backend-а, така и във frontend-a.

## **Имплементация на Backend системата**

Първа стъпка в имплементацията на backend системата е инициализацията и конфигурацията на базата данни, след това се правят моделите, първите миграции, след това се правят repository-та и controller-и. В репозиториите е логиката, а в контролерите са самите endpoint-и отговорите (response-ите) на сървъра.

## **Инициализация на базата и конфигурация**

След като е направена архитектурата на базата и таблиците на теория, следва инициализацията, която е сравнително проста задача, благодарение на **Entity Framework,** тъй като той генерира голяма част от кода, който е нужен за целта. В следващата фигура е представена първата част от DatabaseContext файлът:

A screen shot of a computer program

Description automatically generated Фигура 11: Пример за първата част от DatabaseContext файлът.

Втората част от DatabaseContext файлът е представена в следващата фигура. Там се намира **OnModelCreating метода:**

A computer screen shot of a program code

Description automatically generated

Фигура 12: Пример за втората част от DatabaseContext файлът.

## **Модели**

Моделите са необходими, за да зададем на Entity Framework скелета, по който да създаде базата данни и таблиците, които ще са необходими. В тези модели съществуват различни анотации, които служат за различни неща, като ограничение на дължината на записа, означаване на Id и т.н.

В „HealthEdge” понастоящем има 13 модела. В следващото изображение са представени всички модели, като част от тях са създадени с цел улеснение на бъдещо развитие.

A screenshot of a computer

Description automatically generated

Фигура 13: Всички модели на HealthEdge към настоящия момент

Конкретен пример за модел от „HealthEdge” за таблица Patients е моделът Patient:

A screen shot of a computer program

Description automatically generated

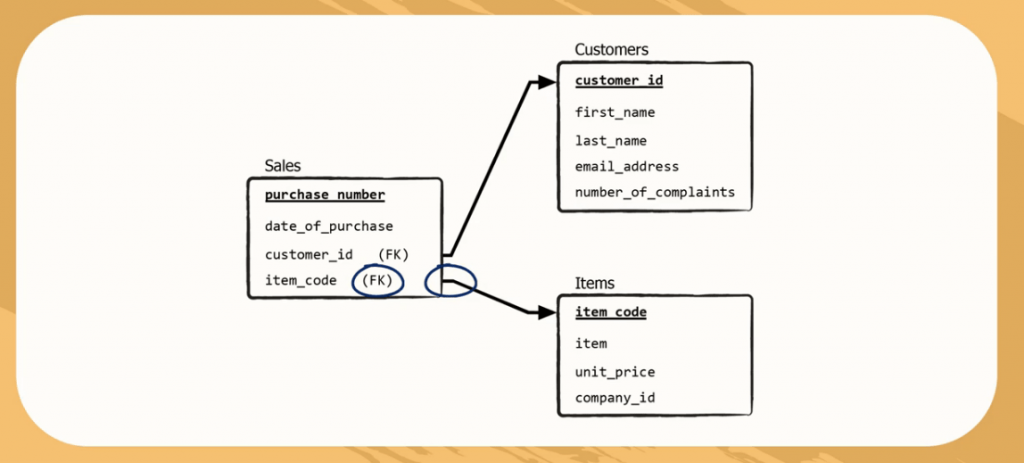
Фигура 14: Модел Patient в „HealthEdge”

Моделът Patient служи за създаване на таблицата Patients. Разликата между двете е името, моделът (entity-то) е в единствено число, докато името на таблицата е в множествено число. Това е така, тъй като в обектно-ориентираното програмиране, моделът обикновено представлява една същност или обект. Например моделът Patient представлява един пациент. Ето защо моделите обикновено се именуват в единствено число. Така става по-ясно, че класът на модела и неговите екземпляри се отнасят до отделни същности. В случая в класа Patient се работи с данните на един пациент. От друга страна, една таблица съдържа множество екземпляри на подобни същности. Следователно именуването на таблица Patients показва, че таблицата съдържа данни за много пациенти, а не само за един. Това наименование в множествено число помага на разработчиците бързо да разберат, че таблицата е колекция от много записи, всеки от които съответства на различен пациент.

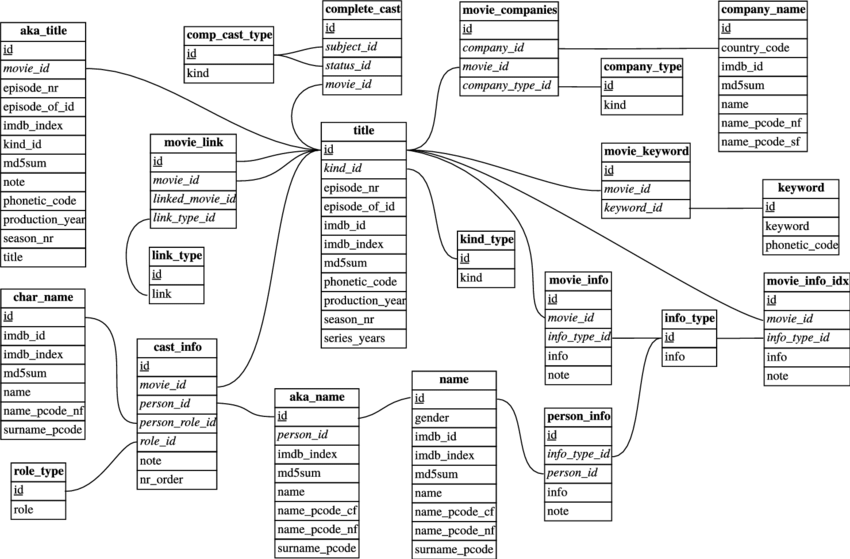
В примера с модела на таблицата Patients, можем да разбием модела на няколко основни части.

* **Основни атрибути и свойства**
  + **[Key]: Означава, че свойството Id е основен ключ на таблицата.**
  + **[Required]: Указва, че полето е задължително. Използва се за Id, FirstName, LastName, Email, Gender, BloodType, ContactNumber и Address.**
  + **[DatabaseGenerated(DatabaseGeneratedOption.Identity)]: Показва, че стойността на Id се генерира автоматично от базата данни при добавяне на нов запис.**
* **Свойства за въвеждане на данни**
  + **FirstName, LastName, Email, ContactNumber, Address: Тези свойства съхраняват лична информация и контактни данни за пациента.**
  + **[StringLength(50)], [StringLength(100)], [StringLength(1)], [StringLength(4)], [StringLength(255)]: Задават максималните дължини за съответните низове, управляващи ограниченията на базата данни за различните полета.**
* **Специфични типове и формати**
  + **[Column(TypeName = "Date")]: Указва, че DateOfBirth трябва да се съхранява като тип дата в базата данни, без време.**
  + **Gender: Съхранява пола на пациента, ограничен до един символ.**
  + **BloodType: Съхранява кръвната група на пациента, ограничена до четири символа.**
* **Foreign Keys and Navigation Properties**
  + **UserId и съответното User: Опционални свойства, които свързват пациента с потребителски профил в системата, ако такъв е наличен.**
* **Колекция от данни**
  + **Appointments: Колекция, която съдържа всички медицински назначения/приеми асоциирани с пациента, показва връзката му с различни медицински услуги.**

**Моделът Patient е създаден така, че да предоставя структуриран начин за съхранение и управление на лични и медицински данни за пациентите в рамките на софтуерна система, улеснявайки достъпа до важна информация и поддържайки висока степен на даннова цялостност.**



Фигура 15: Пример за начина на работата на Foreign key



Допълнителна фигура 1: Пример за начина на работата на множество Foreign keys

Друг пример, в който може да открием малко по-сложно устройство на таблица можем да открием в Doctor model:

A screen shot of a computer program

Description automatically generated

Фигура 15: Модел Doctor в „HealthEdge”

В примера с модела на таблицата Doctors, Можем да разбием модела на няколко основни части:

* **Атрибути и свойства**
  + **[Key] Този атрибут отбелязва свойството Id като primary key (първичен ключ) на таблицата.**
  + **[Required] Указва, че полето е задължително. Използва се за свойства като Id, FirstName, LastName, ContactNumber и Email.**
  + **[DatabaseGenerated(DatabaseGeneratedOption.Identity)]: Показва, че стойността на Id се генерира от базата данни, когато се вмъква нов запис. Това е типично за полета на основни ключове, които се увеличават автоматично (auto increment).**
  + **[StringLength(50)] или [StringLength(100)]: Задава максималната дължина на символите, позволена в свойства от тип низ, използвани за дефиниране на ограничения в схемата на базата данни.**
  + **Свойства като FirstName, LastName, Username, ContactNumber и Email са пряко разбираеми; те съхраняват лична и контактна информация за доктора.**
* **Foreign Keys and Navigation Properties**
  + **[ForeignKey("FieldName")]: Този атрибут определя кое свойство е чуждестранен ключ във връзка между таблиците. Например, RegionId е чуждестранният ключ, който свързва Doctor с ентити-то Region.**
  + **Navigation properties (Region, Specialization, Insurance)**: Тези свойства представляват връзките за взаимоотношения с други таблиците/ентитети. Те позволяват лесно зареждане на свързаните данни (например, регионът на доктора, специализацията и застраховката).
* **Колекции**
  + **Appointments: Това е колекция, която съдържа списък на всички назначения, асоциирани с доктора. Илюстрира връзка от тип един към много, където един доктор може да има много назначения.**
* **Булеви стойности**
  + **IsPediatrician**: Булево свойство (свойство, което има два типа стойности (true или false), което се използва при търсето на лекар и чрез него се отбелязва дали желаният лекар трябва да бъде педиатър.
  + **Nzok**: Това булево свойство се използва при търсенето на лекар и чрез него се отбелязва дали желаният лекар трябва да работи с НЗОК. (Национална здравноосигурителна каса)
* **Допълнителна логика и характеристики**
  + Този модел конкретно не съдържа методи или допълнителна логика пряко, но има модели, в които се създава допълнителна логика и характеристики, но в конкретния модел - Patient, той предимно дефинира структурата и взаимоотношенията на данните Doctor в базата данни, улесняващи управлението на информацията за докторите ефективно, докато осигурява съответствие на данните и изпълнява бизнес правила чрез ограничения и взаимоотношения.

Следващият модел, който ще разгледаме е **User** моделът:

A screen shot of a computer program

Description automatically generated

Фигура 17: Модел User в „HealthEdge”

В примера с модела на таблицата Users, можем да разбием модела на няколко основни части:

* Основни атрибути и свойства
  + **[Key]**: Показва, че свойството Id е основен ключ на таблицата.
  + **[Required]**: Задължителност на свойствата FirstName, LastName, UserName, Email, и Password, което гарантира, че тези полета трябва да бъдат попълнени при регистрация или създаване на нов потребител.
  + **[DatabaseGenerated(DatabaseGeneratedOption.Identity)]**: Стойността на Id се генерира автоматично от базата данни при добавяне на нов запис, което е типично за полета, които служат като уникални идентификатори.
* Свойства за въвеждане на данни
  + **FirstName, LastName, UserName, Email, Password**: Съхраняват основната информация за идентификация на потребителя и осигуряват достъп до системата чрез име и парола.
  + **DateOfCreation**: Записва датата и часът на създаване на потребителския профил, което помага за административни и сигурностни цели.
* Връзки и зависимостти
  + **Patients**: Колекция от Patient ентитети, които са свързани с конкретния потребител. Това отношение подчертава, че докато всеки потребител може да има асоциирани с него пациенти, не всеки потребител задължително се явява пациент. От друга страна, всеки пациент трябва да има свързан User профил, отразен чрез UserId, което е ключова връзка в базата данни.

Тази структура позволява на системата да разграничава потребителските акаунти (такива, които могат да бъдат на служители, администратори или други лица с достъп до системата) от профилите на пациентите, които се нуждаят от медицински услуги и се управляват чрез акаунта на свързания потребител.

Следващият пример за модел, част от архитектурата на „HealthEdge” е моделът **MedicalRecord,** който служи за архив на пациентите от техните посещения. В него можем да открием:

A screen shot of a computer program

Description automatically generated

Фигура 18: Модел MedicalRecord в „HealthEdge”

В примера с модела на таблицата Users, можем да разбием модела на няколко основни части:

* **Основни атрибути и свойства**
  + **[Key]**: Показва, че свойството Id е основен ключ на таблицата.
  + **[Required]**: Указва задължителността на всички основни полета в модела, включително PatientId, DoctorId, RecordDate, Diagnosis, и Treatment.
  + **[DatabaseGenerated(DatabaseGeneratedOption.Identity)]**: Стойността на Id се генерира автоматично от базата данни при добавяне на нов запис.
* **Свойства за въвеждане на данни**
  + **PatientId, DoctorId**: Числови идентификатори, които свързват записа съответно към конкретен пациент и лекар.
  + **RecordDate**: Датата и часът на създаване на медицинския запис.
  + **Diagnosis**: Диагнозата поставена на пациента, с максимална дължина на текста 255 символа.
  + **Treatment**: Описанието на лечението, предписано на пациента, с максимална дължина до 500 символа.
* Navigation Properties
  + **Patient**: Навигационно свойство към модела Patient, което позволява директен достъп до информацията за пациента, свързан с медицинския запис.
  + **Doctor**: Навигационно свойство към модела Doctor, което улеснява достъпа до информация за лекаря, отговорен за съответния медицински запис.

Този модел е критичен за функционирането на здравната информационна система, тъй като предоставя структуриран начин за съхранение на важни медицински данни, което позволява ефективно проследяване и управление на медицинското обслужване на пациентите. Освен това, моделът поддържа високо ниво на даннова цялостност и улеснява регулярното обновление и анализ на медицинската информация.

Следващите модели, които са интересни, заради тяхната функция са моделите **Insurance, Region и Specialization:**

A screen shot of a computer program

Description automatically generated

Фигура 18: Модел Specialization в „HealthEdge”

A screen shot of a computer program

Description automatically generated

Фигура 19: Модел Region в „HealthEdge”

A screen shot of a computer code

Description automatically generated

Фигура 20: Модел Insurance в „HealthEdge”

Общото между тези модели е, че те съдържат еднакви property-та. Това е така, тъй като тези модели са основни за структурирането на информацията в базата данни и позволяват създаването на връзки между различните entity-та чрез foreign ключове. Например, Doctors таблицата може да включва InsuranceId, RegionId, и SpecializationId като чуждестранни ключове, което позволява лесното свързване на лекари с техните застраховки, региони и специализации. Това улеснява търсенето и филтрирането на данни в системата, както и предоставя детайлна информация за квалификациите и работната област на лекарите.

Property-тата, които са еднакви и за трите:

* **Id:**
  + [Key], [Required], [DatabaseGenerated(DatabaseGeneratedOption.Identity)]: Уникален Id идентификатор, който бива така генериран автоматично( auto increment).
* **Name:**
  + **[Required]**: Името на специализацията, което е задължително. Това осигурява, че всяка специализация има ясно идентифицирано наименование.

Последните по-важни модели, които можем да ги разбием на отделни елементи са моделите **UserLogin и AuthenticatedResponseModel**:

A screen shot of a computer program

Description automatically generated

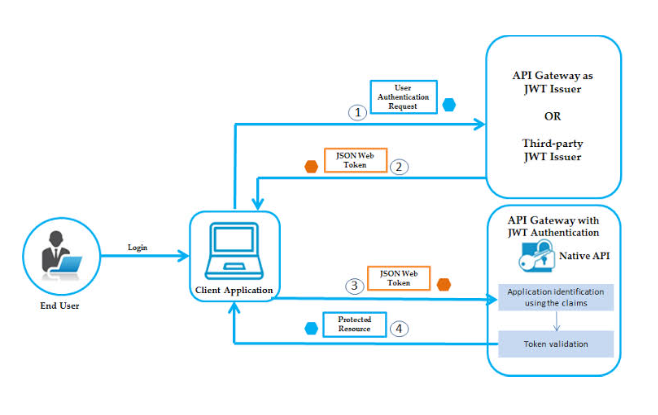
Фигура 21: Модел UserLogin в „HealthEdge”

В UserLogin модела съдържа следното:

* **Email**:
  + **[Required]**: Атрибутът указва, че полето за електронна поща е задължително. Това означава, че потребителят трябва да предостави валиден email адрес за успешно завършване на процеса на вход.
  + **[StringLength(100)]**: Определя максималната дължина на email адреса до 100 символа. Това помага за поддържане на консистентността на данните и предотвратява проблеми свързани със съхранението и обработката на данни.
* **Password**:
  + **[Required]**: Атрибутът показва, че паролата също е задължително поле. Потребителят трябва да въведе своята парола при всеки опит за вход.
  + **[StringLength(100)]**: Задава максималната дължина на паролата до 100 символа, което осигурява достатъчно място за съхранение на силни и сигурни пароли.

Моделът UserLogin е основен за механизмите за сигурност в рамките на приложението, като улеснява проверката на идентичността на потребителите чрез проверка на тяхната електронна поща и парола. При успешна валидация на креденциалите, потребителят получава достъп до системата или до определени ресурси в нея.

Този модел е важен за защита на приложението от неоторизиран достъп и осигурява основата за всякакви допълнителни мерки за сигурност, като двуфакторна аутентикация или логиране на потребителска активност.



Фигура 22: Пример за имплементация на User Authentication

В AuthenticatedResponseModel откриваме:

* **Token:**
  + **Това е свойството, което съхранява аутентикационния токен. Токенът е стринг, който обикновено съдържа зашифрована информация, идентифицираща потребителя, валидността на токена и други важни атрибути за сигурността.**
  + **Токенът се издава след успешна аутентикация и се използва от клиентските приложения за удостоверяване на потребителя при всяка последваща заявка към сервера. Това позволява на системата да потвърди, че заявките са извършени от легитимен и аутентициран потребител.**

**Предназначение на модела**

Моделът AuthenticatedResponseModel изпълнява ключова роля в системите за управление на идентичността (identity management systems), като улеснява сигурната комуникация между клиентски и сървърни компоненти. Използването на токени за аутентикация помага за намаляване на нуждата от многократно предоставяне на потребителски имена и пароли, което подобрява както сигурността, така и удобството за крайните потребители.

След успешна аутентикация, системата генерира токен, който се връща чрез този модел като част от отговора на API. Токенът обикновено се използва в HTTP заглавката на последващи заявки за удостоверяване на потребителската сесия. Това осигурява един сигурен механизъм за управление на сесии и достъп до ресурси без постоянно логване от страна на потребителя.

A screen shot of a computer

Description automatically generated

Фигура 23: Модел AuthenticatedResponseModel в „HealthEdge”

## Repositories

Всяко репозитори (repository) в контекста на .net core е главна централна концепция, особено във фреймуърковете (framework) за програмиране, които следват модел на дизайн **Repository pattern.** Репозиторията действа като посредник между домейна на приложението или бизнес логиката и слоя за управление на данни. Основната цел на репозиторията е да капсулира логиката за достъп до данни или друг източник на данни. **Някои от главните аспекти на репозиториите са:**

* Data access abstraction (Абстракция на достъпа до данни)

Репозиторията абстрахира и централизира управлението на достъпа до данните. Това означава, че потребителите на репозиториите не трябва да знаят за сложността на базата данни или механизмите на заявките. Вместо това, те използват опростени методи като **Add, Remove, GetById, FindAll и тн.**

* Facilitation of testing (Спомагане за тестване)

Чрез използването на репозитории, тестването на компонентите, които използват данни става по-лесно. Може да се създадат макети на репозитория или да се използват фалшиви имплементации, което улеснява тестването на бизнес логиката на приложението, без да е необходим достъп до реалната база данни.

* Separation of concerns (Разделяне на отговорностите)

Репозиториите помагат за постигането на принципа за разделение на отговорностите в софтуерната архитектура. Те ясно разграничават логиката за управление на данните от останалата част от приложението. Това води до по-чист код и по-лесно поддръжка.

* Improving scalability and maintainability (Подобряване на скалирането и поддръжката

Когато измененията в структурата на базата данни или в бизнес правилата налагат промени в управлението на данни, обикновено е достатъчно да се обнови само репозиторият. Това изолира останалата част от кода от пряко влияние на такива промени, което улеснява поддръжката и мащабируемостта на приложението.

* Code reusability (Повторно използване на код)

Репозиториите позволяват повторното използване на код, тъй като функционалността за достъп до данни може да се използва от различни части на приложението без необходимостта да се дублира код.

* Implementation example (Пример за реализация)

В типично .NET приложение, репозиторията може да бъде имплементиран като клас, който имплементира интерфейс IRepository. Този интерфейс обикновено дефинира стандартен набор от операции за достъп до данни. Репозиторията след това използва Entity Framework (или друг ORM инструмент) за да изпълни тези операции със съответната база данни.

Използването на репозитории в софтуерните архитектури, особено с .net, е широко приет подход за подобряване на организацията, тестването и поддръжката на софтуерни проекти, особено в приложения с богати на данни домейни.

**Защо имплементацията на repository наследява интерфейс от репозиторията.**

Тази техника е особено важна в разработването на големи и модуларни приложения, където поддръжката на кода и разширяемостта са от съществено значение. Някои от плюсовете на използването на четвъртия SOLID принцип, а именно Interface segregation principle **(Принцип на разделяне на интерфейсите)**, който гласи, че не трябва да се принуждават клиентите да зависят от интерфейси, които те не използват. Този принцип насърчава създаването на специфични интерфейси, вместо един голям, което също поддържа по-добра организация и чистота на кода.

Ето и някои от плюсовете от наследяването.

* Separation of concerns (отделяне на отговорностите)
  + Интерфейсите позволяват отделяне на логиката за достъп до данни от останалата част на приложението. Това дава възможност бизнес логиката да бъде разделена от подробностите на това как данните се съхраняват и извличат, което води до по-чист и по-организиран код.
* Flexibility and scalability (гъвкавост и скалиране)
  + Интерфейсите предоставят определение за това какви операции може да изпълнява репозиторията, без да специфицират как точно тези операции се изпълняват. Това позволява на разработчиците да променят имплементацията на репозиторията без да влияят на компонентите, които го използват. Например, може да преминете от една система за управление на бази данни към друга без нужда да променяте бизнес логиката.
* Facilitation of testing (Улеснение на тестването)
  + Използването на интерфейси улеснява тестването на приложението чрез въвеждане на зависимости. Това позволява на разработчиците да вмъкнат макети или фалшиви репозитории по време на тестване, което може да помогне за симулиране на различни сценарии или за тестване на бизнес логиката без реален достъп до базата данни.
* Improved code maintenance and management (Подобряване на поддръжката и управлението на кода)
  + Интерфейсите дефинират ясен контракт, който трябва да се спазва от всички имплементации, което гарантира консистентност в начина, по който се обработват операциите с данни в различните части на приложението. Тази консистентност улеснява разбирането, поддръжката и разширението на приложението, тъй като всички имплементации на репозитории спазват един и същ набор операции, дефинирани от интерфейса.
* Enforcement of standards (Обезпечаване на стандарти)
  + Когато репозиториите наследяват от интерфейс, те трябва да имплементират всички методи, дефинирани от интерфейса, което гарантира, че определени стандарти и поведения се спазват в цялото приложение. Това стандартизиране помага за поддържане на еднородност в функционалността и качеството на приложението.
* Decoupling and dependency injection (Отделяне и въвеждане на зависимости)
  + Репозиториите, имплементирани чрез интерфейси, позволяват по-добро отделяне на кода. Това отделяне е критично за въвеждането на зависимости, което от своя страна улеснява по-доброто тестване и спазването на принципите SOLID, особено Принципа за обръщане на зависимостите. Фреймуърците (framework) за въвеждане на зависимости могат динамично да предоставят конкретните имплементации на репозиториите, необходими на компонентите на приложението, без да свързват тези компоненти тясно с конкретни класове на репозиториите.
* Interchangeability and versioning (Заменяемост и версиониране)
  + Интерфейсите позволяват различните имплементации на репозиторий да бъдат заменени без промяна на зависимия код. Това е полезно за актуализиране или подобряване на компоненти на системата с течение на времето без да се нарушава цялостната функционалност на приложението. Също така поддържа версионирането на интерфейси, където нови функции могат да бъдат добавени по начин, съвместим с предишните версии.

В следващото изображение е нагледно представен пример за интерфейс на репозитория в „HealthEdge”:

A computer screen shot of a program code

Description automatically generated

Фигура 24: Интерфейс на repository PatientRepository

След като интерфейсът IPatientRepository за репозиторията е готов, следва имплементацията на PatientRepository:

A computer screen shot of a program

Description automatically generated

Фигура 25: Първата част от PatientRepository

A computer screen with text on it

Description automatically generated

Фигура 26: Втората част от PatientRepository

Репозиторията PatientRepository наследява IPatientRepository интерфейса и предоставя конкретна имплементация за управление на данните за пациенти в рамките на болнична система. Класът използва Entity Framework за достъп до база данни, определена в контекста **HospitalDbContext.** Подробно обяснение на всяка функция в репозиторито:

* Конструктор

A screen shot of a computer

Description automatically generated

Фигура 27: Конструктор в PatientsRepository

* + Конструкторът получава инстанция на **HospitalDbContext,** който се използва за всяка операция с данни. Този контекст е свързан с базата данни и управлява състоянието на entity-тата.
* Методи за управление на данни
  + GetAllPatients():
    - * Връща списък на всички пациенти в базата данни чрез асинхронно извличане на всички записи от таблицата Patients.
  + GetPatientByIdAsync(int Id):
    - * Извлича специфичен пациент по дадено Id. Използва FindAsync метода, който е оптимизиран за търсене по ключ.
  + CreatePatientAsync(Patient patient):
    - * Добавя нов пациент в базата данни. Първо добавя пациента към контекста и след това извършва запис с SaveChangesAsync, който асинхронно актуализира базата данни.
  + GetTotalPatientsCountAsync():
    - * Връща общия брой на пациентите в базата данни, използвайки CountAsync за асинхронно броене на записите в таблицата Patients.
  + UpdatePatientAsync(Patient patient):
    - * Актуализира информацията за съществуващ пациент. Методът Entry(patient).State = EntityState.Modified маркира целото entity като модифицирано, което ще предизвика обновяване на всички полета на пациента при изпълнението на SaveChangesAsync.
  + DeletePatientAsync(int id):
    - * Изтрива пациент по зададено Id. Намира пациента с FindAsync и ако съществува, го премахва от контекста и базата данни с SaveChangesAsync. Тази функционалност се използва почти единствено и само от администраторите на „HealthEdge”
  + SearchPatientAsync(string patientName):
    - * Търси пациенти по име. Методът извлича всички пациенти, чиито имена съдържат подадения низ, като проверява както първото (личното), така и фамилното име.
  + SearchPatientByDateOfBirthAsync(DateOnly patientDateOfBirth):
    - * Търси пациенти по дата на раждане. Използва Where клаузата за филтриране на пациенти, чиято дата на раждане съответства на зададената (търсената).

Следващата репозитория (repository), което ще разгледаме е DoctorRepository. DoctorRepository наследява IDoctorRepository интерфейса и предоставя конкретна имплементация за управление на данните за докторите в рамките на болнична система. В нея можем да открием:

* Конструктор
  + A screen shot of a computer code

    Description automatically generatedКонструкторът приема обект **HospitalDbContext,** който осигурява на репозиторията достъп до контекства на базата данни за изпълнение на операции.

Фигура 28: Конструктор в DoctorsRepository

* Методи за управление на лекарите
  + GetAllDoctorsAsync() :
    - * Асинхронно извлича всички лекари от базата данни и ги връща като списък. Използва се методът ToListAsync() за изпълнение на заявката.
  + GetDoctorAsync(int id) :
    - * Извлича един лекар по техния уникален идентификатор(id). За това се използва FindAsync(), който е оптимизиран за намиране на entity-та по техния primary key.
  + CreateDoctorAsync(Doctor doctor) :
    - * Добавя нов лекар в базата данни. Преди добавянето, генерира уникално потребителско име, като проверява потребителското име на последния лекар в базата данни и го увеличава. Това включва парсване на числовата част от потребителското име, увеличаването ѝ и присвояването на новото потребителско име на новия лекар.
      * След задаването на потребителското име, лекарят се добавя към контекста и се запазва с SaveChangesAsync().
  + UpdateDoctorAsync(Doctor doctor) :
    - * Актуализира информацията за съществуващ лекар в базата данни. Състоянието на ентитета се задава като Modified, което гарантира, че всички свойства ще бъдат обновени в базата данни, когато се извика SaveChangesAsync().
  + DeleteDoctorAsync(int id) :
    - * Премахва лекар от базата данни на база техния id. След като намери лекаря с FindAsync(), ако лекарят съществува, той се премахва от контекста и промените се запазват.Тази функционалност се използва почти единствено и само от администраторите на „HealthEdge”
  + DeleteMultipleDoctorsAsync(IEnumerable<int> ids) :
    - * Изтрива няколко лекари, идентифицирани по списък от ids. Това става като първо се намерят всички лекари, чиито ID-та са в предоставения списък, след което всички те се премахват наведнъж с RemoveRange(), и накрая се извиква SaveChangesAsync(). Тази функционалност се използва почти единствено и само от администраторите на „HealthEdge”.
  + SearchForDoctorAsync(int? specializationId, bool needsToBePediatrician, int? regionId, int? insuranceId, string? firstName, string? lastName) :
    - * Търси лекари на база на опционални критерии: Id на специализацията, дали трябва да бъде педиатър, Id на региона, Id на застраховката, и филтри за име (първо и фамилно име). Методът постепенно изгражда заявка, като добавя условия въз основа на предоставените параметри и включва свързани назначения в резултата с помощта на Include().
* Ползи и предназначение
  + DoctorRepository предлага обширен набор от методи, които капсулират общите операции, необходими за управление на записите на лекарите в болнична управленска система. Тази капсулация подобрява поддръжката на кода, улеснява тестването (чрез изолиране на кода за достъп до данни) и следва принципите на модела на репозитория за отделяне на бизнес логиката от въпросите, свързани с достъпа до данни. Специфичният метод за генериране на уникално потребителско име за новите лекари осигурява целост на данните и предоставя последователен подход за управление на уникалните ограничения в рамките на логиката на приложението.

A screen shot of a computer program

Description automatically generated

Фигура 29: Първа част от DoctorsRepository

A screen shot of a computer code

Description automatically generatedФигура 30: Втора част от DoctorsRepository

След PatientsRepository и DoctorsRepository, следващата репозитория, която ще разгледаме е AuthenticationRepository:

Тя наследява IAuthenticationRepository интерфейсa.

* Конструктор
  + Инициализира нова инстанция на **HospitalDbContext** с предоставените **DbContextOptions.**
  + Съхранява конфигурационни настройки (вероятно от appsettings.json или подобен файл) в private field \_configuration за по-късно използване, особено за четене на настройките на JWT.

A screen shot of a computer program

Description automatically generated

Фигура 31: Конструктор в AuthenticationRepository

* Login метод
  + **Проверка за Съществуващ Потребител**: Първоначално проверява дали съществува потребител с предоставения имейл в базата данни. Използва се FirstOrDefaultAsync(), което е ефективно за тази операция, тъй като връща резултат веднага щом намери съвпадение.
  + **Проверка на Паролата**: Ако бъде намерен потребител, паролата се проверява с BCrypt, метод известен със своята сигурност за съхранение и проверка на хеширани пароли. Това гарантира, че дори ако базата данни бъде компрометирана, реалните пароли остават защитени.
  + **Генериране на Токен**: Ако паролата бъде потвърдена, генерира се JWT за потребителя. Токенът включва:
    - * **Издател и Аудитория**: Прочетени от конфигурацията, тези полета посочват кой издава токена и за кого е предназначен токенът.
      * **Изтичане на Срока**: Зададен да изтече след 5 минути. Този кратък срок е типичен за чувствителни приложения, изискващи от потребителите често да се аутентикират отново за поддържане на сигурността.
      * **Подписващи Удостоверения**: Използва симетричен сигурен ключ (извлечен от стойност на конфигурация) и хеширащия алгоритъм HMAC SHA256 за подписване на токена, гарантирайки, че токенът не може да бъде променен без да се анулира.
  + **Връщане на Токен**: Методът връща AuthenticatedResponseModel, съдържащ JWT. Този модел вероятно е много прост и съдържа само низа на токена.

**Съображения за Сигурност**

* + **Сигурно Съхранение на Пароли**: Използва BCrypt за хеширане на пароли, което включва вградена сол за защита срещу атаки с дъгови таблици.
  + **Сигурно Управление на Токени**: JWT е подписан със секретен ключ, който трябва да се съхранява сигурно и никога да не се разкрива публично. Използването на HMAC SHA256 осигурява силна сигурност за валидиране на токена.
  + **Кратък Срок на Токена**: Краткият живот на токена ограничава риска от злоупотреба, ако токенът бъде прехванат или откраднат.

**Използване в Приложението**

* A computer screen shot of a program

  Description automatically generatedТози репозиторен метод може да се използва по време на процеса на вход в уеб, мобилни или настолни приложения, където сигурността и аутентикацията на потребителите са от критично значение. Той обработва както логиката за аутентикация, така и генерирането на сигурни токени, които клиентите могат да използват за удостоверяване на последващи заявки към сървъра.

Фигура 32: AuthenticationRepository в „HealthEdge”

Следващата репозитория (repository), която ще разгледаме и ще анализираме е AppointmentRepository. Този клас AppointmentRepository, който имплементира интерфейса IAppointmentRepository, предоставя функционалности за управление на записи за медицински назначения (appointments) в болничната база данни. Ето подробен преглед на неговите методи и функции:

* Конструктор
  + Инициализира репозитория с контекст към базата данни (HospitalDbContext) и помощник за управление на назначенията (AppointmentHelper), който помага във валидацията и логистиката на назначенията.

A screen shot of a computer code

Description automatically generated

Фигура 33: Конструктор в AppointmentRepository

* Методи
  + GetAllAppointmentsAsync():
    - * Връща всички appointment-и, като включва детайли за съответните лекари и пациенти. Използва Include за добавяне на свързаните данни към резултата.
  + **GetAppointmentByIdAsync(int appointmentId)**:
    - * Извлича специфичен appointment по даден идентификатор, също включвайки данни за съответния лекар и пациент.
  + **GetAppointmentsByDoctorIdAsync(int doctorId)**:
    - * Връща всички appointment-и за даден лекар, включвайки данни за пациентите.
  + **GetAppointmentsByPatientIdAsync(int patientId)**:
    - * Връща всички appointment-и за даден пациент, включвайки данни за лекарите.
  + **CreateAppointmentAsync(Appointment appointment)**:
    - * Създава нов appointment, като първо валидира времето и детайлите на appointment-a с помощта на AppointmentHelper. Проверява за пресичащи се назначения и добавя новото назначение към базата данни след валидация.
  + **EditAppointmentAsync(Appointment appointment)**:
    - * Актуализира детайлите на съществуващ appointment. Маркира съответния обект като модифициран и запазва промените в базата данни.
  + **DeleteAppointmentAsync(int appointmentId)**:
    - * Изтрива назначение по даден идентификатор след като първо извлече appointment-а от базата данни. Ако такова съществува, премахва го и запазва промените.
  + **GetAvailableSlots(int doctorId, DateTime desiredDate)**:
    - * Търси свободни времеви слотове за appointment-и за даден лекар в избрана дата. Определя началото и края на работния ден, продължителността на едно назначение и итерира през възможните времеви слотове, за да намери свободно време.

**Значение и употреба:**

AppointmentRepository има важна роля в управлението на appointment-ите в болничната информационна система. Той улеснява извличането, създаването, актуализирането и изтриването на назначения, като осигурява релевантна логика за валидация и обработка на назначенията. Този репозиторий е ключов за ефективното планиране и управление на ресурсите в здравното заведение, като осигурява точност и надеждност в управлението на назначенията и тяхната координация с пациентите и медицинския персонал

A screen shot of a computer program

Description automatically generated

Фигура 34: Първа част от AppointmentRepository в „HealthEdge”

A computer screen shot of a program code

Description automatically generated

Фигура 35: Втора част от AppointmentRepository

A computer screen shot of a program code

Description automatically generated

Фигура 36: Трета част от AppointmentRepository

## Контролери

Контролерите са отговорни за комуникацията на приложението с backend логиката. Пример за това може да бъде контролерът за потребителя, като останалите работят по подобрен принцип, но по малко по-различен начин имплементирано, с цел удовлетворяването на конкретните нужди.

A computer screen shot of a program code

Description automatically generated

Фигура 37: Контролер за потребител в “HealthEdge”

В този контролер комуникацията се извършва в различни методи, които имат същата цел като методите в repository-тата. Той представлява API контролер в ASP.NET Core, управляващ операциите, свързани с потребителите. Той използва репозитория IUserRepository за достъп до данни, което позволява изолация на логиката за достъп до данни от API логиката. Ето анализ на различните методи в контролера и тяхната функционалност:

* Конструктор:
  + Конструкторът инициализира контролера с инстанция на IUserRepository, която се използва за извършване на операции с данни за потребителите.

A screen shot of a computer code

Description automatically generated

Фигура 38: Конструктор в UserController, част от „HealthEdge”

* Методи
  + **GetTotalUsersCountAsync()**:
    - * HTTP GET заявка, която връща общия брой на потребителите в системата. Използва метода GetTotalUsersCountAsync() от репозитория и връща резултата в HTTP OK отговор.
  + **GetUsersAsync()**:
    - * HTTP GET заявка, която извлича всички потребители. Ако няма потребители, връща HTTP No Content отговор; в противен случай връща списъка на потребителите.
  + **GetUserByIdAsync(int Id)**:
    - * HTTP GET заявка за извличане на специфичен потребител по идентификатор. Ако потребителят не е намерен, връща HTTP Not Found отговор.

* + **CreateUserAsync([FromBody] UserCreationDto userDto)**:
    - * HTTP POST заявка, която създава нов потребител. Първо проверява дали вече съществува потребител със същото имейл или потребителско име. Ако такъв съществува, връща HTTP Conflict отговор. Ако не, хешира паролата, създава нов потребител, записва го и връща данните на новосъздадения потребител.
  + **EditUserAsync(int Id, [FromBody] EditUserDto userDto)**:
    - * HTTP PUT заявка, която актуализира данните на съществуващ потребител. Проверява дали потребителят съществува и дали има друг потребител с новата посочена имейл или потребителско име. Ако промените са валидни, обновява потребителя и връща обновените данни.
  + **DeleteUserAsync(int Id)**:
    - * HTTP DELETE заявка, която изтрива потребител по даден идентификатор. Ако изтриването е успешно, връща HTTP No Content отговор.
  + **DeleteMultipleUsersAsync([FromBody] IEnumerable<int> userIds)**:
    - * HTTP DELETE заявка за изтриване на множество потребители едновременно. Изпълнява операцията и връща HTTP OK отговор при успех.
* **Обобщение:**

UserController предоставя RESTful API интерфейс за управление на потребителите в системата. Той поддържа стандартните операции CRUD (създаване, четене, обновяване, изтриване), което го прави централен компонент за управление на потребителските данни в приложението. Използването на DTO (Data Transfer Objects) помага за сигурността и правилното управление на данните, които се предават между клиента и сървъра.

## Имплементация на Frontend системата

Frontend-ът се състои от компоненти, които се свързват помежду си чрез Routing-а, който е вграден в React, service-ите служат за комуникацията с backend-а, използвайки т.нар Response модели.

## Компоненти

В React, компонентите са основните строителни блокове на потребителския интерфейс. Те са независими и повторно използваеми части код, които отговарят за рендерирането на части от потребителския интерфейс. Компонентите могат да бъдат функционални или класови, като функционалните компоненти често използват hooks (на български – куки) за управление на състоянието и странични ефекти.

A computer screen shot of a program code

Description automatically generated

Фигура 39: Първа част от компонент DoctorCard.js

A screen shot of a computer program

Description automatically generated

Фигура 40: “Втора част от компонент DoctorCard.js”

A computer screen shot of text

Description automatically generated

Фигура 41: Трета част от компонент DoctorCard.js

Компонентът DoctorCard в конкретния случай на „HealthEdge” е визуален елемент, който представя информация за даден лекар. Той използва различни hooks(куки) на React, включително useState и useEffect, за да управлява локалните състояния и да извършва странични ефекти, като заявки към API за получаване на данни. Ето по-подробно обяснение на основните части на този компонент:

* **useState Hooks:** 
  + **earliestSlot за най-ранната свободна дата и час за прием;**
  + **specialization, region и insurance за съхранение на специализацията, региона и застраховката на лекаря.**
* **useEffect Hooks:**
  + **Има четири useEffect кука, всеки от които се активира при промяна на определено ID (за доктор, специализация, регион, застраховка). Всеки хук изпълнява HTTP заявка към определен URL за получаване на съответната информация и обновява състоянието при успешен отговор.**
* **Функция за форматиране на дати customFormatDate:**
  + Тази функция приема низ, представляващ дата, и връща форматиран низ с дата и час. Ако годината на датата е преди 2000, функцията връща низ, посочващ да се обадите за повече информация.
* Визуализация на компонента
  + Изображение на лекаря, име, специализация и регион се показват в компонента.
  + Използва се иконата StarIcon от библиотеката @heroicons/react за визуализация на оценки.
  + Показва най-ранния свободен час за прием.
  + Бутон, който позволява на потребителя да запази час при лекаря.
* Динамично съдържание:
  + Текстовите полета и датите се актуализират динамично в зависимост от получените от API-то данни.

Стилизацията може да се получи по няколко различни начина, някои от най-известните от тях са:

* + External (Външни) CSS файлове:
    - * Използването на стандартни CSS файлове е най-простият и традиционен метод. Компонентите в React могат да включват CSS файлове, които се прилагат глобално или могат да се изолират чрез конвенции за именуване.
  + Inline стилове:
    - * Inline стиловете се дефинират директно в JSX като обект, който предоставя динамичност и лесен контрол над стиловете, които се прилагат веднага. Това е удобно за динамични стилове, базирани на т.нар. състояние или пропсове.
  + CSS модули
    - * CSS модулите са популярен метод, който позволява инкапсулация на CSS стиловете за отделни компоненти. Всеки модул генерира уникални класове, които помагат да се избегне проблемът с колизии на имената на класовете.
  + Styled Components:
    - * Това е библиотека за стилизиране, която използва тагнати шаблонни низове за създаване на стилизирани компоненти. Styled Components автоматично гарантира уникалност на стиловете, което позволява лесно и удобно управление на темите и стиловете.
  + CSS-in-JS библиотеки
    - * CSS-in-JS е подход, където CSS правилата се дефинират в JavaScript, което позволява използването на JS променливи и функции директно в стиловете. Примери за такива библиотеки са Emotion и JSS.
  + CSS frameworks
    - * CSS фреймуърци (CSS frameworks) са предварително написани библиотеки, които съдържат готови за употреба стилови решения, които могат да улеснят и ускорят процеса на уеб разработка. Те предоставят консистентен стилистичен вид на уеб страници и приложения и подпомагат разработчиците да изградят адаптивни (responsive) и атрактивни интерфейси с минимално. Ето някои от най-популярните CSS фреймуърци:
        + Bootstrap
        + Foundation
        + Tailwind CSS
        + Semantic UI
        + Material UI

A purple logo with text

Description automatically generated



## **Services в “HealthEdge”**

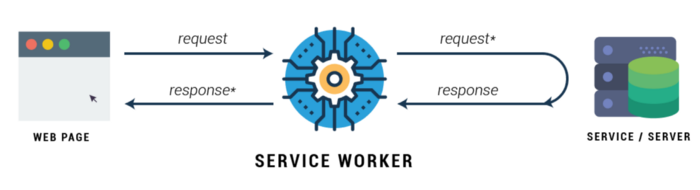
A screen shot of a computer program

Description automatically generated

Фигура 42:Пример за service в patientService.js

Сървисът съдържа следните функции:

* + Функция за извличане на данни за конкретен пациент.
  + Функция за обновяване на данните на пациент.
  + Функция за добавяне на пациент.
  + Функция за изтриване на пациент.



Допълнителна фигура 2: Пример за начина на работата на service worker в React.js

## **Routing в „HealthEdge”**

A computer screen shot of a program code

Description automatically generated

Фигура 43:Route-path-ове в “HealthEdge”

В „HealthEdge” има следните маршрути:

* **“/”**. Това е стандартен маршрут, който се отваря веднага след достъпването на уеб приложението. Съответства на LoginPage. Може да бъде достъпен от всеки потребител.
* **register:** Този маршрут е за страницата за регистрацията. Съответсва на RegisterPage и може да бъде достъпен от всеки потребител.
* **home:** Този маршрут е за достъпване на началната страница за търсене на лекари. Съответства на HomePage. Чрез добавяне на **ProtectedRoute** може да се ограничи достъпването само за пациенти.
* **doctors:** Този маршрут е за показване на филтрираните лекари, базирано на желанията на пациента. Съответства на компонента DoctorsList и чрез добавяне на **ProtectedRoute** може да се ограничи достъпването само за пациенти.
* **dashboard:** Този маршрут служи за потребителите с администраторски права. Отговаря на компонента Dashboard, който отговаря за статистика, администрация и т.н. Чрез добавяне на **ProtectedRoute** може да се ограничи достъпването само за администратори.
* **book-appointment:** Този маршрут служи за запазване на часове при избрания от пациента лекар. Отговаря на BookAppointment компонента и чрез добавяне на **ProtectedRoute** може да се ограничи достъпването само за пациенти.
* **patients:** Този маршрут служи за показване на всички пациенти, както и тяхното администриране. Отговаря на Patients компонента. Чрез добавяне на **ProtectedRoute** може да се ограничи достъпването само за администратори.
* **edit-patients/:id:** Маршрут за редактиране на информация за конкретен пациент, идентифициран по id, чрез компонента EditPatient. Чрез добавяне на **ProtectedRoute** може да се ограничи достъпването от пациенти и администратори.
* **Patient-details/:id:** Този маршрут води до PatientDetails, където могат да се видят детайли за конкретен пациент. Чрез добавяне на **ProtectedRoute** може да се ограничи достъпването от пациенти и администратори.
* **doctors-list:** Този маршрут води до DoctorManage компонента, където могат да се управляват профилите на докторите от тях самите или от администраторите при нужда. Чрез добавяне на **ProtectedRoute** може да се ограничи достъпването от конкретния лекар за конкретен профил или от администраторите.
* Този маршрут води до UsersManage компонента, където могат да се управляват профилите на потребителите от тях самите или от администраторите при нужда. Чрез добавяне на ProtectedRoute може да се ограничи достъпването от конкретния потребител за конкретен профил или от администраторите.

## **Валидациите**

Нека хвърлим и един бърз поглед на валидациите, които са използвани за формите в приложението:

A computer screen with text and symbols

Description automatically generated

Фигура 44: Пример за валидация за email в „HealthEdge”

От примера откриваме валидация за email, които са под формата на Regex – в конкретния пример е показано валидацията за email.

# Ръководство за потребителя.

Source Code-ът на проекта може да бъде открит на следното Github хранилище:

<https://github.com/superalex-dev/HealthEdge>

За да достъпите приложението, моля отидете на <https://healthedge.onrender.com>

За да стартирате приложението локално, трябва да изпълните следните стъпки:

**Backend**: Отворете файла с разшиерение .*sln* и стартирайте проекта от **Visual Studio.**

**Frontend**: Изпълнете командата *npm* *install* в папката /healthedgefrontend*,* за да инсталирате необходимите *npm* пакети, след това се уверете, че сте в попката /healthedgefrontend и след това изпълнете командата *npm start*, за да стартирате приложението на локален сървър.

## **Swagger**

Swagger UI, част от Swagger, е инструмент с отворен код, който генерира уеб страница, която документира API-то. Това представяне на API-то в потребителския интерфейс е удобно за потребителя и лесно за разбиране, като цялата сложност на логиката остава зад екрана.

Цялата идея на Swagger е да улесни визуализацията на едно уеб приложение за разработчиците, когато то е все още в етап на раработка на backend-a – така се улеснява изпращането на заявки, получаването на отговор от сървъра и тн.

Документацията на Swagger е обширна и инструментът е лесен и приятен за използване, поради което винаги го включвам, когато разработвам уеб приложение.

A screenshot of a computer

Description automatically generated

Фигура 45:Първа част от Swagger в „HealthEdge”

A screenshot of a computer

Description automatically generated

Фигура 46:Втора част от Swagger в „HealthEdge”

## **Frontend**

След стартиране на frontend-a, ще попаднете на началната страница на уеб приложението, именно login страницата. В нея ако няма потребителски акаунт, при натискане на бутон Join us се отваря страницата за регистриране:

A screenshot of a login form

Description automatically generated

Фигура 47:Началната страница на „HealthEdge”

При натискане на червения бутон за регистриране, се отваря страницата за регистриране на пациент:

A screenshot of a screen

Description automatically generated

Фигура 48:Страницата за регистриране в „HealthEdge”

След успешната регистрация, потребителят е пренасочен към началната страница за пациенти, в която пациентите могат да търсят лекар, спрямо техните оплаквания.

При администраторите е малко по-различно, те могат да управляват практически всичко по уеб приложението. Могат да управляват пациенти, потребители, доктори и т.н. Те имат отделен профил и т.н.

# Заключение

Уеб приложението HealthEdge е разработено добре, като са следвани всички приети принципи в програмирането. Кодът е лесен за разбиране, добре описващ себе си. Страниците са лесни за усвояване, UI-я и UX-a са на ниво.

Като цяло смятам, че HealthEdge е един достоен проект и ще бъде продължен да бъде разработван, като някои от бъдещите feature-и ще са:

* **Бърз и усъвършенстван Search за търсене на лечебни заведения**
* **AI модел за анализиране на националното здраве на локално, местно и държавно ниво.**
* **Система за онлайн плащане**
* **Час система за връзка директно с лекар**
* **По-голям и обхватен администраторски профил.**

Както и още други…

# Информационни източници

* <https://react.dev/learn>
* <https://learn.microsoft.com/en-us/ef/core>
* <https://stackoverflow.com>
* <https://github.com/>
* <https://swagger.io/docs>

# Приложения

В рамките на разработката на уеб приложението „HealthEdge”, е създадено Github repository, което служи като централно място за съхранение на всички source кодове, свързани с проекта. То дава възможност на общността да допринесе към проекта чрез предложения за подобрения.

В README файла на репозиторията може да се намери допълнителна важна информация относно проект „HealthEdge”. Този файл служи като първоначално ръководство за всеки, който желае да разбере повече за структурата и функционалностите на „HealthEdge”, както и за онези, които искат да допринесат към неговото непръкъснато развитие и подобрение.

За достъп до github repository, моля посетете следния URL:

Там ще намерите всички необходими ресурси и документация, за да разберете по-добре структурата и функционалностите на уебсайта, както и да допринесете към неговото развитие.