ЧАСТНА ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО ДИГИТАЛНИ НАУКИ

„СОФТУНИ СВЕТЛИНА“

ДИПЛОМНА РАБОТА

ЗА ПРИДOБИВАНЕ НА ПРОФЕСИОНАЛНА КВАЛИФИКАЦИЯ

НА ТЕМА:

**“Разработване на уеб приложение за онлайн пазаруване”**

ИЗГОТВИЛ: НАУЧЕН РЪКОВОДИТЕЛ:

**Лъчезар Димов Колев Виктор Красимиров Костов**

03.05.2023 г.

ГР. СОФИЯ

# Съдържание

* [Увод 4](#_heading=h.3znysh7)
* [Проблем](#_heading=h.2et92p0) 5
* [Цели на дипломния проект](#_heading=h.tyjcwt) 6
* [Глава 1. Проучване](#_heading=h.1t3h5sf) 7
  + 1.1 Съществуващи унифицирани магазини и технологиите които използват 7
  + 1.2 Плюсове и минуси на унифицираните магазини 8
  + 1.3 Трудности при създаване на унифициран магазин 9
* [Глава 2. Използвани технологии](#_heading=h.4d34og8) 10
  + 2.1 Back End 10
    - 2.1.1 Начин на работа на ORM 11
    - 2.1.2 Начин на работа на така наречените „Миграции“ 11
    - 2.1.3 Поддръжка на различни видове бази данни 11
  + 2.2 Причини за използване на програмния език: C# 12
  + 2.3 Front End 12
    - 2.3.1 Какво е Angular и защо се използва в този проект? 12
    - 2.3.2 Компонентите в Angular 13
    - 2.3.3 “Service” в Angular 14
* [Глава 3. Проектиране и имплементация](#_heading=h.2s8eyo1) 16
  + 3.1 Архитектура на системата 16
  + 3.2 Имплементация на Back End системата 16
    - 3.2.1 Модели 17
      * 3.2.1.1 Модел на продуктовия тип 17
      * 3.2.1.2 Модел на продуктa 18
      * 3.2.1.3 Модел на хранилището 19
      * 3.2.1.4 Модел на потребителя 20
    - 3.2.2 Свързване на Entity Framework с база данни 22
    - 3.2.3 Контролери 23
      * 3.2.3.1 Repository pattern 24
      * 3.2.3.2 Контролерите и връзката им с репозиториите 24
      * 3.2.3.3 Контролерите на продуктите, хранилищата и продуктовия тип 26
      * 3.2.3.4 Завършеното API 29
  + 3.3 Имплементация на Front End системата 29
    - 3.3.1 Файлова структура на проекта 29
      * 3.3.1.1 „src“ директория в Angular 29
      * 3.3.1.2 „src/app“ директория в Angular 30
    - 3.3.2 Routing в Angular 31
      * 3.3.2.1 Routing чрез navbar в Angular 31
    - 3.3.3 Свързване на Angular с Back End 32
      * 3.3.3.1 Използване на ендпоинти дефинирани в Backend 32
      * 3.3.3.2 Изпращането на HTTP заявки от Angular към Backend 33
      * 3.3.3.3 Observable в Angular 34
      * 3.3.3.4 Създаване на Response модели 34
    - 3.3.4 Визуализиране на информацията получена от Back End 35
    - 3.3.5 Автентикация и регистриране на потребителите 37
      * 3.3.5.1 Създаването на необходимите форми 38
      * 3.3.5.2 Регистриране на нови потребители 40
      * 3.3.5.3 Автентикация на потребители 41
      * 3.3.5.4 JWT Token 43
      * 3.3.5.5 Автентикация чрез JWT Token 43
      * 3.3.5.6 Права на различните потребители 44
      * 3.3.5.7 Interceptors в Angular и как работи автентикацията при тях 46
* [Глава 4. Ръководство за потребителя](#_heading=h.z337ya) 50
  + 4.1 Системни изисквания 50
  + 4.2 Описание на различните страници от уеб приложението 50
* [Заключение](#_heading=h.1y810tw) 54
* [Информационни източници](#_heading=h.4i7ojhp) 54

# Увод

Възходът на електронната търговия промени драстично начина, по който потребителите пазаруват, като направи покупката на продукти и услуги онлайн по-лесна от всякога. Въпреки това, при наличието на толкова много възможности, клиентите могат бързо да бъдат претоварени, което води до разочарование и потенциални загуби на продажби. Именно тук се появява концепцията за "единен онлайн магазин", който предлага на клиентите една платформа за достъп до продукти и услуги от различни марки.  
  
Единният онлайн магазин е платформа, която обединява различни марки и техните продукти в рамките на един уебсайт или приложение. Тя предлага на клиентите удобството да пазаруват множество продукти от различни марки на едно място, като същевременно осигурява по-ефективно пазаруване.  
  
Тази концепция набра популярност през последните години, като има няколко забележителни примера като Amazon, Alibaba и eBay, където клиентите имат достъп до разнообразни продукти от различни марки. Успехът на тези унифицирани онлайн магазини се дължи отчасти на способността им да предоставят широка гама от продукти и услуги, съчетана с лекотата и удобството на единна платформа за пазаруване.  
  
Въпреки успеха на тези големи играчи обаче все още има достатъчно възможности за по-малките предприятия да се възползват от предимствата на единния онлайн магазин. Например нишов магазин, който предлага редица продукти от по-малки, бутикови марки, би могъл да се възползва от единния онлайн магазин, за да създаде по-значимо онлайн присъствие и да разшири клиентската си база.  
  
Ползите от единния онлайн магазин не се ограничават само до клиентите. Компаниите също могат да се възползват от тази концепция, като получат достъп до по-голяма клиентска база, без разходите и времето, необходими за изграждане на самостоятелен уебсайт за електронна търговия. Като обединяват усилията си с други марки, фирмите могат да намалят разходите си за маркетинг и да увеличат достъпа си до нови клиенти.  
  
Освен това един обединен онлайн магазин предлага рационализирана верига за доставки, като намалява сложността и разходите, свързани с управлението на множество доставчици и логистика. Освен това единният онлайн магазин може да осигури анализ на данни, който да помогне на фирмите да разберат по-добре своите клиенти и да адаптират продуктите и услугите си към техните нужди.  
  
Съществуват обаче и потенциални недостатъци на единния онлайн магазин, като например намалена видимост на продуктите и потенциал за засилена конкуренция от страна на други марки на същата платформа. Тези недостатъци могат да бъдат смекчени чрез внимателно планиране и изпълнение, включително разработване на стабилна стратегия за брандиране и създаване на уникални предложения за стойност за всяка марка, представена в платформата.  
  
Накратко, единният онлайн магазин е мощна концепция, която има потенциал да бъде от полза както за клиентите, така и за бизнеса. Като създава единна платформа за множество марки, единният онлайн магазин предлага оптимизирано пазаруване за клиентите, като същевременно осигурява на бизнеса достъп до по-голяма клиентска база и по-значителна експозиция пред потенциалните клиенти. Потенциалните ползи са многобройни, но е необходимо внимателно планиране и изпълнение, за да се гарантира успехът на единния онлайн магазин.

## Проблем

В днешния свят на електронна търговия има безброй възможности за онлайн пазаруване, което затруднява потребителите да се ориентират и да намерят желаните продукти. Единният онлайн магазин осигурява решение на този проблем, като предлага единна платформа, където потребителите могат лесно да получат достъп до широка гама продукти и услуги от различни марки. С единния онлайн магазин потребителите могат да спестят време и усилия, като пазаруват множество продукти от различни марки на една платформа, премахвайки необходимостта да навигират в множество уебсайтове или приложения. Това удобство може да направи пазаруването по-ефикасно и по-малко натоварващо за потребителите.

## 

## 

## 

## 

## 

## 

## Цели на дипломния проект

Основната цел на изграждането на единен онлайн магазин е да се осигури централизирана платформа, която да обединява множество марки и продукти, до които клиентите да имат достъп и да купуват. По този начин се опростява онлайн пазаруването за клиентите, като им се предлага единна платформа, където те могат лесно да получат достъп до широка гама продукти и услуги от различни марки. Така може да се постигне по-голяма удовлетвореност на клиентите и увеличаване на достъпа до нови клиенти за различните фирми, които са част от единния онлайн магазин.

Задачи, произтичащи от целите

* Извършване на проучване и подготовка на обзор на проблемната област
* Избиране на технологиите чрез които ще се изгради проектът
* Изграждане на сървърна част
* Реализация на унифицирания магазин в уеб среда: създаване на уеб приложение свързано със сървърната част
* Създаване на приятен UI и UX, чрез които потребителите ще могат да използват създадения проект с лекота и удоволствие
* Извеждане на изводи и заключения на базата на разработения проект

# Глава 1. Проучване

Унифицираните онлайн магазини са онлайн платформи, които съчетават повече от един тип продукти или услуги в един магазин. Това означава, че потребителите могат да намерят всичко, което търсят, на едно място. Тези магазини обикновено предлагат различни продукти, като например дрехи, обувки, аксесоари, козметика, електроника, книги, мебели и много други. Освен това, те предлагат различни услуги, като например пътувания, хотели, ресторанти и други.

## 1.1 Съществуващи унифицирани магазини и технологиите които използват

Съществуват множество известни унифицирани онлайн магазини, които се използват от милиони хора по света. По-долу са изброени някои от най-известните от тях, заедно с технологиите, които използват:  
  
Amazon: Amazon е един от най-големите онлайн търговци на дребно в света, който предлага голямо разнообразие от продукти - от електроника до дрехи. Технологичният стек на Amazon включва Java, Python и Ruby on Rails за уеб приложенията и използва Amazon Web Services (AWS) за изчисления в облак и съхранение.  
  
eBay: eBay е онлайн пазар, на който потребителите могат да купуват и продават стоки. eBay използва комбинация от Java, PHP и Node.js за своите уеб приложения, а също така използва AWS за нуждите си от изчисления в облак.  
  
Alibaba: Alibaba е китайска компания за електронна търговия, която предоставя платформа на бизнеса за продажба на продукти на потребители по целия свят. Alibaba използва Java за своите уеб приложения, а също така разполага със собствена платформа за изчисления в облак, наречена Alibaba Cloud.  
  
Shopify: Shopify е платформа за електронна търговия, която позволява на бизнеса да създава онлайн магазини. Shopify използва Ruby on Rails за своите уеб приложения, а също така има собствена рамка за уеб разработка, наречена Liquid.  
  
Magento: Magento е платформа за електронна търговия с отворен код, която позволява на бизнеса да създава онлайн магазини. Magento е написана на PHP и използва MySQL като система за управление на бази данни.  
  
Това са само няколко примера за някои от най-известните унифицирани онлайн магазини и технологиите, които те използват. Трябва да се отбележи, че много от тези компании използват различни технологии и инструменти за поддържане на своите платформи и често разполагат с големи екипи от разработчици, които работят за постоянното подобряване и актуализиране на системите им.

## 1.2 Плюсове и минуси на унифицираните магазини

Ето някои от плюсовете и минусите на унифицираните магазини:

Плюсове:

* Голямо разнообразие от стоки: В унифицираните магазини може да се намерят много различни продукти и услуги на едно място, което прави пазаруването удобно за клиентите.
* Намалени разходи: Като резултат от общото управление на магазините, унифицираните магазини могат да намалят разходите за закупуване на стоки, логистика, маркетинг и други разходи, което може да доведе до по-ниски цени за клиентите.
* Подобрени услуги: По-големите магазини имат повече ресурси да инвестират в обучение на персонала и на нови технологии, което може да доведе до по-добро обслужване на клиентите.
* Лоялност на марката: За клиентите, които ценят консистентността и лоялността на марката, унифицираните магазини могат да бъдат предпочитани пред индивидуалните магазини.

Минуси:

* Ограничен избор: Въпреки че унифицираните магазини предлагат голямо разнообразие от стоки, може да има липса на специализиран продукт или услуга, която клиентът може да намери само в по-специализиран магазин.
* Не персонализирано обслужване: Когато персоналът на магазина има да се справи с голям брой клиенти, може да е трудно да се предостави персонализирано обслужване на всеки един клиент.

## 1.3 Трудности при създаване на унифициран магазин

Едно от най-важните решения при изграждането на унифициран онлайн магазин или какъвто и да е проект е избирането на правилните технологии и платформи, които ще бъдат основа на проекта.

Изборът на подходящата платформа може да бъде труден, тъй като всяка от тях има свои предимства и недостатъци. Необходимо е да се избере такава платформа, която отговаря на нуждите и целите на проекта.

Следващата трудност при създаване на такъв магазин е самият дизайн на магазина(това как ще изглежда). Изграждането на привлекателен дизайн на магазина може да бъде трудно. Дизайнът трябва да бъде лесен за навигация, атрактивен и интуитивен.

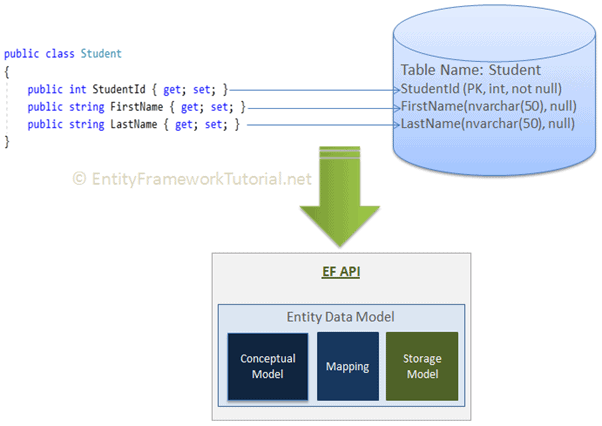
Инвентара също трябва да се поддържа по някакъв начин. Управлението на инвентара може да бъде трудно, особено ако има голямо разнообразие от продукти. Необходимо е да се осигури система за управление на инвентара, която да позволява лесна промяна на наличностите и обработка на поръчки.

# Глава 2. Използвани технологии

За да се постигне така наречения унифициран магазин са ни необходими доста компоненти, които трябва да работят в синхрон за да се постигне безпроблемно извършване на работа. Един от тези компоненти е гореспоменатия бекенд, който ще служи за поддръжка на инвентара на магазина и менажиране на потребителите, регистрирани в системата.

## 2.1 Back End

Първата стъпка при създаването на такъв магазин е изграждането на бекенда, в случая той е базиран на ORM(Обектно-релационно картографиране) на Entity Framework, накратко казано този framework прави C# класовете на таблици в базата данни, това се случва по следния начин:



Фигура 1: Entity Framework и как работи

### 2.1.1 Начин на работа на ORM

Обектно-релационното картографиране се нуждае от следното за да работи:

Дефиниране схемата на базата данни: Първата стъпка е да се определи схемата на базата данни, която включва таблиците, колоните и връзките между таблиците. Обикновено тази схема се дефинира с помощта на език за дефиниране на данни като SQL.  
  
Определяне на модела на обекта: След това се дефинира моделът на обекта, който включва класовете и техните свойства, които ще се използват за взаимодействие с базата данни.  
  
Съпоставяне на модела на обекта със схемата на базата данни: След това ORM framework съпоставя модела на обекта със схемата на базата данни. Това съпоставяне обикновено се извършва с помощта на конфигурационни файлове или анотации в кода.  
  
Генериране на SQL оператори: Когато приложението трябва да взаимодейства с базата данни, ORM framework генерира SQL команди въз основа на модела на обекта и съпоставянето със схемата на базата данни.  
  
Изпълнение на SQL команди: Накрая ORM framework изпълнява SQL операциите и връща резултатите на приложението като обекти. След това приложението може да манипулира обектите, като използва методите и свойствата на ORM framework, а промените автоматично се запазват в базата данни.

Накратко, ORM (Object Relational Mapping) е инструмент, който позволява на разработчиците да работят с бази данни като обекти. ORM-ите превръщат таблични данни в обекти и обратно, като автоматизират задачите за заявки, мапинг и транзакции.

### 2.1.2 Начин на работа на така наречените „Миграции“

EF(Entity Framework) позволява на разработчиците да работят с механизъм за миграции, който автоматизира процеса на промяна на структурата на базата данни. Това означава, че когато е нужно да се направи промяна в базата данни, EF ще генерира автоматично миграционен скрипт, който ще направи промяната без да нарушава съществуващите данни.

### 2.1.3 Поддръжка на различни видове бази данни

EF поддържа множество бази данни, включително SQL Server, Oracle, MySQL, PostgreSQL и др. Това означава, че разработчиците могат да работят с бази данни от различни доставчици, като използват един и същи код.

## 2.2 Причини за използване на програмния език: C#

C# е обектно-ориентиран език, който позволява създаването на обекти, които могат да се използват за решаване на сложни проблеми. Това прави програмния код по-лесен за разбиране и поддръжка. Също така строгата типизация на C# гарантира, че кодът е по-безопасен и по-малко податлив на грешки, свързани с типовете данни.

Един от големите плюсове на C# езика е богатата му библиотека, която включва много стандартни функции, графични интерфейси, мрежови и сигурностни функции и много други.

Езикът е част от платформата .NET, която е една от най-големите и използвани платформи за разработка на софтуер. Освен това, C# е езикът, който е препоръчван от Microsoft за разработка на уеб приложения в платформата ASP.NET, именно тази платформа е използвана в този проект.

## 2.3 Front End

Отиваме сега до реализирането на нашето приложение в уеб среда. За това ни е нужен така наречения Front End. Това е частта от приложението, която ще бъде видима за потребителите и с която ще могат да взаимодействат(клиентската част). За Front Еnd нашето приложение ще използва платформата: Angular

### 2.3.1 Какво е Angular и защо се използва в този проект?

Angular е популярен клиентски framework, който се използва за създаване на уеб приложения. Той е разработен от екип на Google и е базиран на TypeScript. Angular позволява създаването на едностранични приложения (SPA), които могат да бъдат използвани в различни браузъри и на мобилни устройства.

Един от големите плюсове на Angular е, че той използва модулна архитектура. Това означава, че приложенията могат да бъдат разделени на по-малки, независими части, които могат да се разработват и тестват отделно. Това прави процеса на разработка по-лесен, по-бърз и по-ефективен.

Angular също така предлага много възможности за декларативно програмиране, което улеснява създаването на приложения, като се използват структури от HTML, CSS и TypeScript. Angular предоставя също много вградени компоненти и услуги(service), които могат да бъдат използвани за създаване на сложни приложения.

Друга важна функционалност на Angular е двустранното свързване на данни. Това означава, че промените в сорс кода на приложението автоматично ще се отразят във визуалната част на приложението и обратно, като се осигурява по-голяма гъвкавост и интерактивност на приложението.

### 2.3.2 Компонентите в Angular

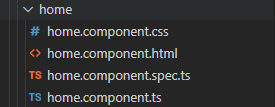
Компонентите са основните строителни блокове в Angular. Те представляват отделни части на уеб приложението, като съдържат HTML, CSS и TypeScript код, които се използват за дефиниране на визуални елементи и логика в приложението. Компонентите се използват за създаване на сложни интерфейси, които са динамични и отзивчиви на действията на потребителя.

В Angular, всеки компонент има свой жизнен цикъл, който включва няколко различни етапа, включително инициализация, обновяване и унищожаване. Това позволява на разработчиците да управляват поведението на компонентите в зависимост от потребностите на приложението.

Компонентите могат да се взаимодействат помежду си, като използват входни и изходни параметри. Входните параметри позволяват на компонентите да получават данни от родителски компонент, докато изходните параметри позволяват на компонентите да изпращат данни към родителския компонент. Това улеснява преноса на данни и управлението на комуникацията между различни компоненти в приложението.

Angular предоставя много вградени компоненти и библиотеки, които могат да се използват за създаване на различни видове компоненти, като например формуляри, таблици, бутони и други. Разработчиците могат също да създават свои собствени компоненти, които да отговарят на специфичните нужди на приложението. Компонентите играят ключова роля в разработката на съвременни, интерактивни и динамични уеб приложения с Angular.

Пример за това какво съдържа един Angular компонент:



Фигура 2: Angular компонент

### 2.3.3 “Service” в Angular

"Service" в Angular е един от основните компоненти, използвани за споделяне на код и данни между различни компоненти и други части на приложението. Те са създадени с цел да подпомагат модуларността и повторното използване на кода.

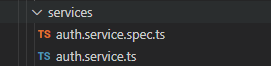
Една от най – важните функционалности на "Service" с способността на него да свърже Front End частта на нашето приложение с Back End-a. Това се случва чрез HTTP заявки, които ни позволяват да имаме връзка между уеб частта на нашето приложение и сървърната част. Т.нар "Service" също ни позволява да достъпваме локалното съхранение на нашия браузър. "Service" също така може да предоставя обща функционалност за множество компоненти, като например методи за автентикация, управление на сесии и други.

За да се използва "service" в компонент, той трябва да бъде инжектиран в конструктора на компонента като зависимост(dependency). Това става с помощта на Angular Dependency Injection механизма, който автоматично създава инстанции на "service" и ги предоставя на компонентите.

Една от големите предимства на "service" е, че те могат да бъдат споделяни между множество компоненти, което позволява на разработчиците да избегнат дублирането на кода и да постигнат по-голяма модуларност и поддръжаемост на приложението.

Създаването на "service" е сравнително лесно и може да бъде извършено чрез генериране на нов TypeScript клас с Angular CLI команда "ng generate service". Това ще създаде нов "service" клас и ще го добави към списъка със зависимости във "providers" масива на модула, в който се използва.

Примерен Angular “Service”:



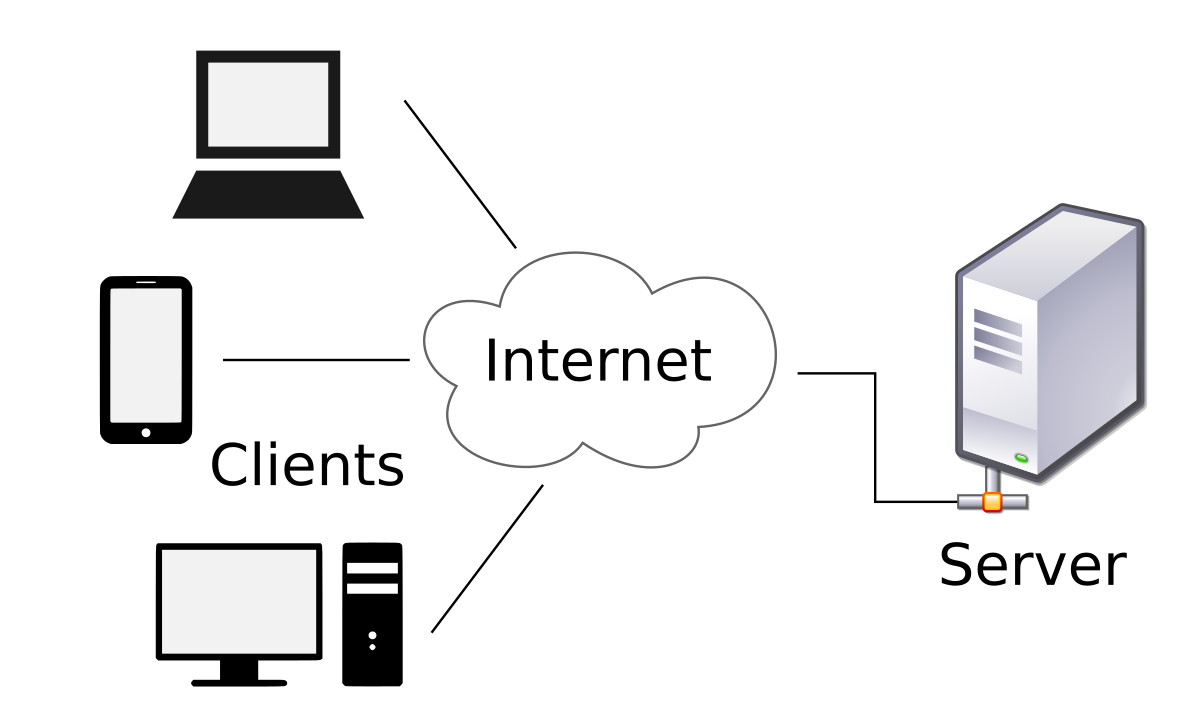
Фигура 3: Angular Service

# Глава 3. Проектиране и имплементация

В тазиглава от документацията ще разгледаме стъпка по стъпка как е изграден т.нар унифициран магазин. Ще разгледаме как работи всеки един компонент в системата и каква е неговата роля.

## 3.1 Архитектура на системата

Проектът се базира на типичната клиент-сървър архитектура, тя изглежда по следния начин:



Фигура 4: клиент-сървър архитектура

Като клиентите имат достъп само да Фронтенд частта на приложението, която част представлява самото уеб приложение видимо в браузъра на потребителя. Фронтендът и бекендът комуникират помежду си - чрез HTTP заявки. Фронтендът например ще изпрати данни към бекенда. След това бекендът може отново да валидира тези данни и накрая ги съхранява в някоя база данни.

Проектът е изградена от две основни части а именно: Front End и Back End, като както вече споменах за бекенд използваме Entity Framework, а за Front End – Angular.

## 3.2 Имплементация на Back End системата

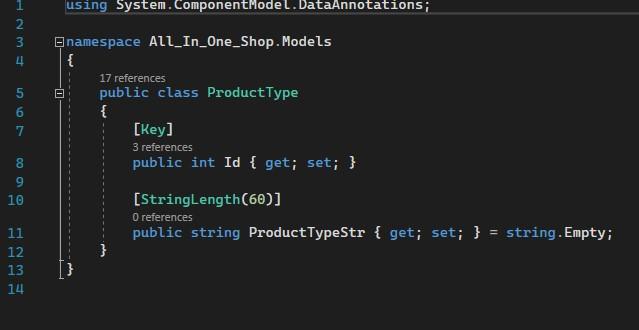
Имплементацията започва от гореспоменатия бекенд. Той ще бъде създаден чрез т.нар Code-First approach, което означава че таблиците в базата данни ще представляват C# класове, които по-късно ще бъдат „преведени“ към SQL таблици. Това „превеждане“ ще бъде направено от Entity Framework чрез Обектно-релационното картографиране, на което се основава целия framework. Тези C# класове служещи като template за SQL таблици са т.нар „Модели“.

### 3.2.1 Модели

Моделите в Entity Framework се използват за описание на структурата на данните в базата данни, както и за управление на тези данни чрез код. Моделите предоставят абстракция върху базата данни, като позволяват да се работи с обекти, които са по-близо до бизнес логиката на приложението, вместо да се занимаваме директно със записите в базата данни.

#### 3.2.1.1 Модел на продуктовия тип

Този модел е сравнително минималистичен, това е неговия вид:

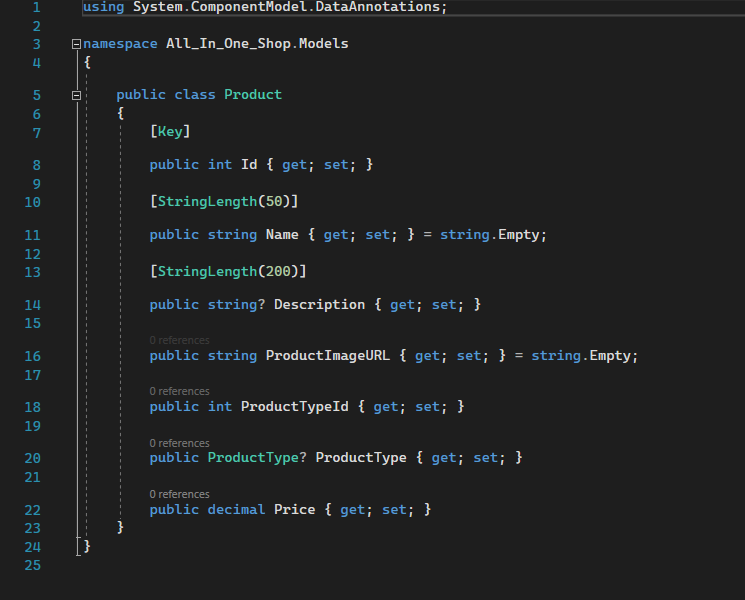


Фигура 5: модел на продуктовия тип

Всички модели в базата данни трябва да имат т.нар ключ за да могат да бъдат достъпвани, именно това е първото пропърти на този модел. Последван от ключа, имаме string на типа който ще позволи да достъпиването именно на самия тип. Ползата от този модел ще бъде обяснена по-долу в текста, когато разглеждаме модела на продуктa.

#### 3.2.1.2 Модел на продуктa

Моделът на продуктите изглежда по сления начин:

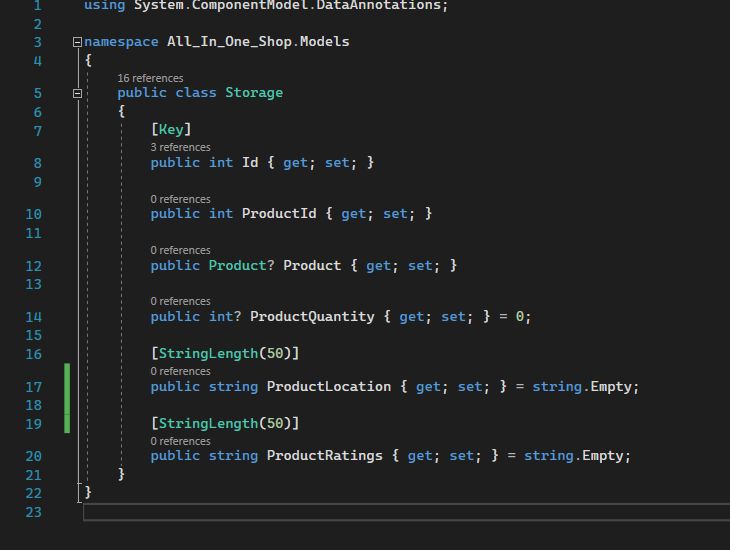


Фигура 6: модел на продукта

Продуктите ще имат име, описание и URL, което ще служи за визуализиране на нашия продукт. Всеки един продукт ще си има и тип за да може да се филтрират по тип, това пропърти обаче е изцяло нов модел, понеже по този начин ще може да имаме само определени разновидности на продуктите. Този нов модел позволява да имаме dropdown меню от което администраторите на магазина ще могат да изберат вида на продукта. Моделът за тип обаче трябва да бъде свързан с всеки един от продуктите. В Entity Framework връзката между два модела се нарича "връзка на обектно-релационна мапинг". Връзката може да бъде едно към едно, много към едно или много към много, в случая е „едно към едно“. За да се създаде връзка между два модела в Entity Framework, трябва да имаме така наречените "ForeignKey" и "InverseProperty", в случая ForeignKey е пропъртито от тип int “ProductTypeId”, а InverseProperty е самия модел с който искаме да свържем нашия продукт. И последното пропърти на продукта е цената, тя от тип decimal, понеже това е препоръчителния тип за цени. Следващия модел, който ще разгледаме е на хранилищата за отделните продукти.

#### 3.2.1.3 Модел на хранилището

Моделът на хранилищата изглежда по следния начин:

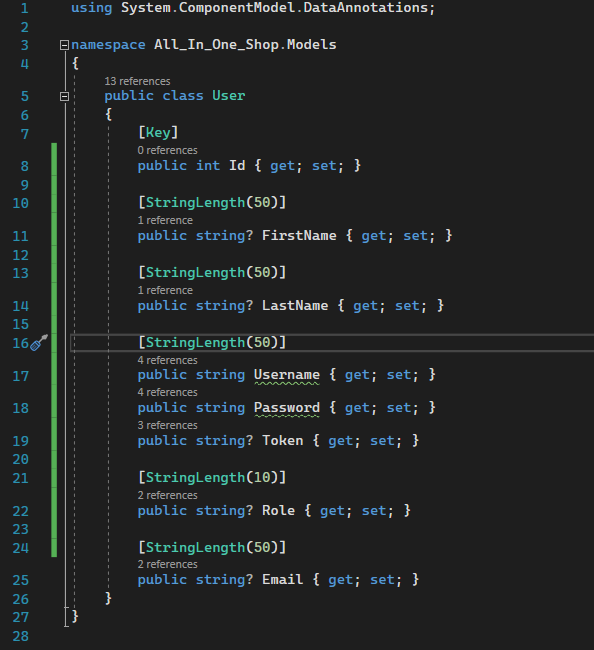


Фигура 7: модел на хранилището

Хранилищата са свързани с продуктите по същият начин по който са свързани типовете с продуктите, тоест връзката е „едно към едно“. Всяко хранилище има определен брои продукти, това е отразено именно чрез пропъртито „ProductQuantity“. Също така имаме пропъртита за локация на продуктите и за техните отзиви.

#### 3.2.1.4 Модел на потребителя

Този модел ще служи за следенето и менажирането на регистрираните потребители в системата, моделът изглежда така:



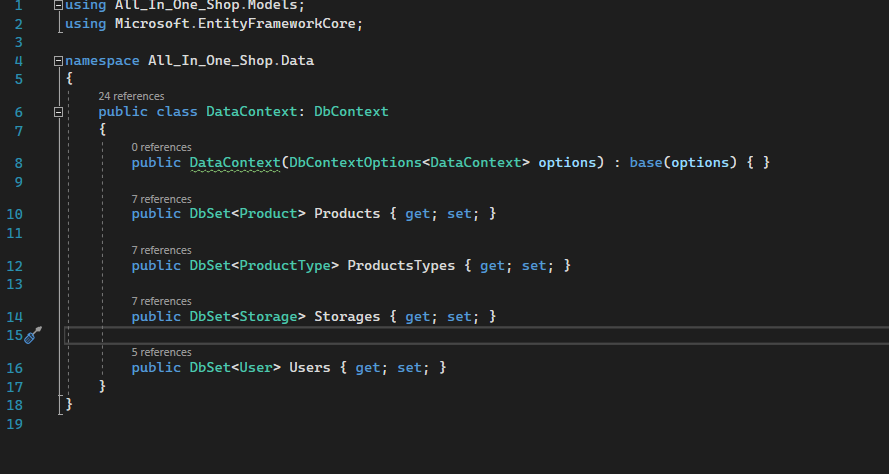
Фигура 8: модел на потребителя

Потребителите ще имат потребителско име и парола, чрез които ще могат да си влизат в профилите, като паролата ще се криптира преди да бъде запазена в базата данни. Всички потребители ще имат и т.нар „Token“, който ще осигурява валидността на профилите на влезналите в системата потребители. Клиентите ще имат и роля, която ще определя какви права има дадения потребител.

### 3.2.2 Свързване на Entity Framework с база данни

След създаването на моделите е необходимо да се запазват в база данни. За да могат да бъдат тези запазвани моделите в база данни трябва Entity Framework да се свърже с базата данни(в случая SQL Server), за да се случи това трябва да:

1. Създадем клас, който наследява DbContext. Този клас трябва да съдържа DbSet пропъртита, които представят таблиците в базата данни. Именно тези DbSet-ове превръщат моделите в таблици от базата данни като им се предостави класа на модела, това се случва по следния начин:

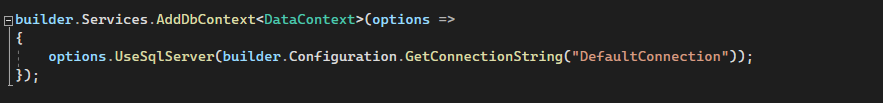


Фигура 8: DbContext клас

1. Конфигурираме връзката към базата данни, в случая това е направено чрез ConnectionString изглеждащ така:

Фигура 9: Connection String

след създаването на този ConnectionString трябва да го подадем на DbContexт класа, това се случва чрез конструктора на дадения клас, като му подаваме информацията по следния начин:

Фигура 10: конфигурационни способностти на builder обекта

Този builder е обект, който представлява средата за конфигуриране на приложението. Той предоставя API за добавяне на различни компоненти, настройка на средата, в която се изпълнява приложението, и др.

3. Създадем първата си миграция и да я прехвърлим към базата данни. Добавянето на миграция в Entity Framework се случва чрез конзолата за мениджър на пакети(Package manager console), като трябва да напишем следното в нея: „dotnet ef migrations add MyMigration“, последвано от „dotnet ef database update“ за да прехвърлим тази миграция в базата данни.

### 3.2.3 Контролери

След създаването на моделите и връзката на тях с базата данни е необходимо да се направи начин за достъп до тях, именно тук идва ролята на т.нар контролери. Всеки модел си има отговарящ за него контролер. Контролерите са важна част от ASP.NET Core и .NET 6 Web API приложенията, тъй като представляват точките на вход на приложението и се използват за приемане на HTTP заявки от клиента и генериране на отговори.

Контролерите обикновено се използват за обработка на HTTP заявки, които имат определен маршрут и HTTP метод (GET, POST, PUT, DELETE и др.).

#### 3.2.3.1 Repository pattern

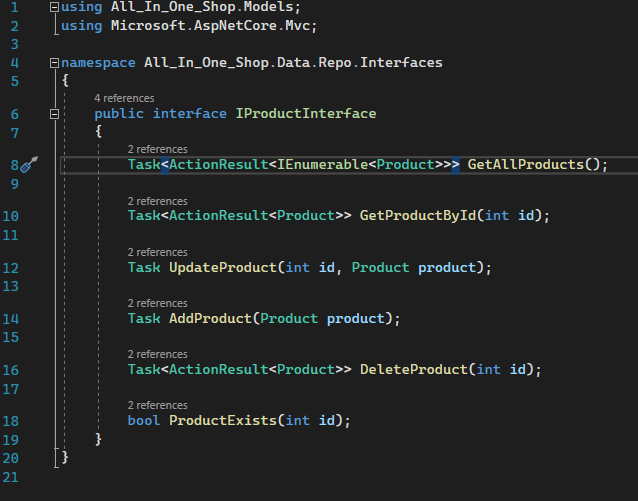
Проектът използва т.нар Repository pattern за да се осигури отделно от контролерите извличане и съхранение на данни. Repository pattern е популярен дизайнерски шаблон(design pattern), който се използва за абстрахиране на данните от приложението и предоставя унифициран интерфейс за достъп до тези данни. Това позволява да се осигури по-лесно преизползване на кода, улеснява поддръжката и намалява зависимостта на контролерите от конкретните технологии за достъп до данни.

При използването на Repo pattern, контролерите не трябва да знаят какви са конкретните данни, които те обработват. Вместо това контролерите работят с интерфейси на репозиториите, които предоставят общи методи за достъп до данните. Конкретните репозитории, които се използват, се конфигурират и инжектират чрез механизма за инжектиране на зависимости на .NET 6 Web API.

#### 3.2.3.2 Контролерите и връзката им с репозиториите

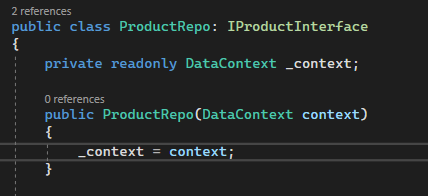
Връзката между репозиториите и контролерите изисква няколко стъпки:

1. Създаване на интерфейс за репозиторията: в този интерфейс ще имаме всички действия които искаме да може да извършва нашия контролер. Примерен такъв интерфейс:



Фигура 11: репозиторен интерфейс

1. Създаване на самата репозитория за дадения контролер: репозиторията представлява C# клас който наследява гореспоменатия интерфейс. Работата на тази репозитория е да връща информация от базата данни. За да стане това трябва се инжектира DbContext класа в даденото репо, това се случва по следният начин:



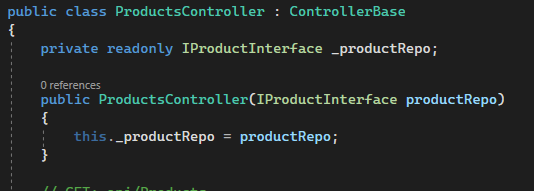
Фигура 12: инжектиране на DbContext класа

1. Свърване на интерфейса с репозиторията: тази връзка се осъществява чрез гореспоменатия builder обект и се случва по следния начин:



Фигура 13: свързване на интерфейса с репозиторията

1. Свърване на репозиторията с контролера: последната стъпка е именно свърването на репозиторията с контролера. Това се случва отново чрез инжектиране на зависимости(dependency injection), връзката между репозиторията и контролера се случва по следния начин:

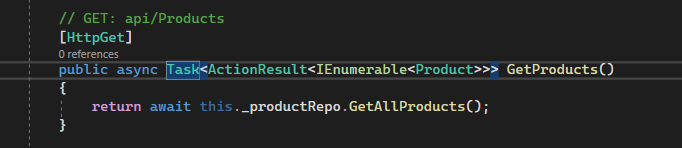


Фигура 14: свързване на контролера с репозиторията

#### 3.2.3.3 Контролерите на продуктите, хранилищата и продуктовия тип

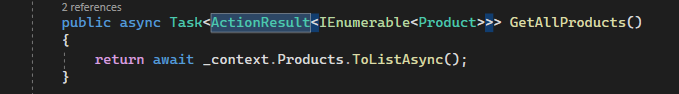
Контролерите за продуктите, хранилищата и продуктовия тип поддържат всички CRUD опрации и съответно всички видове HTTP заявки а именно:

* Get: За да получим информация от базата данни ни е необходима „Get” заявка. Понеже използваме Repository pattern обаче ще е необходимо да получим информацията чрез репозиторията на дадения контролер. Това се случва по следния начин:



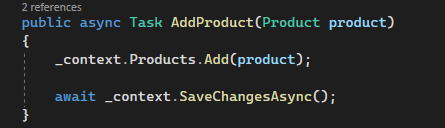
Фигура 15: използване на репозитория в контролер

След това репозиторията връща необходимата информация по следния начин:



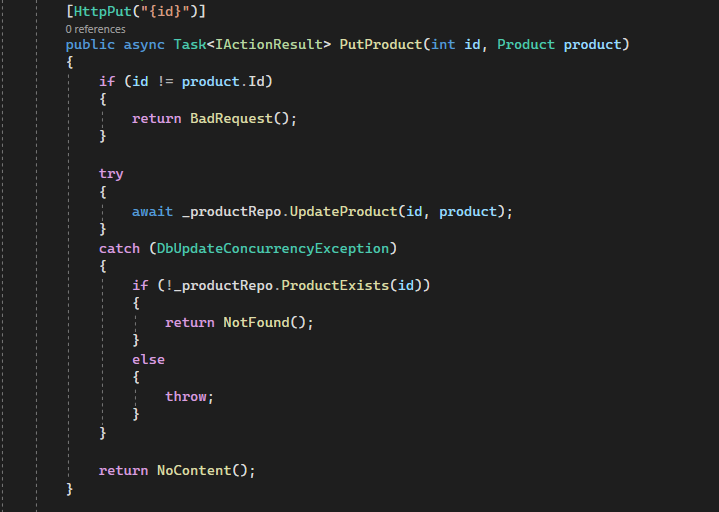
Фигура 16: достъпване на база данни чрез репозитория

* Post: Ако искаме да добавим информация към базата данни използваме метода „Post“. Пример за това би било добавянето на продукт към базата данни. Важно е да се отбележи че при промяна на информация в базата данни задължително трябва да се извика метода „SaveChanges()“. Когато се извика метода „SaveChanges()“, Entity Framework създава транзакция, която съдържа всички промени в контекста на данните. След това, транзакцията се изпълнява и всички промени се записват в базата данни. Пример за това как се използва методът:



Фигура 17: добавяна на елемент към база данни

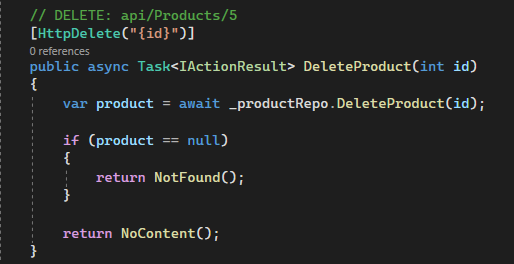
* Put: „Put“ метода се използва при модифицирането на вече съществуващи елементи в базата данни. За да модифицираме даден елемент от базата данни е необходимо да знаем кой е този елемент, това се случва чрез неговия ключ или „Id“, ето как проверяваме дали съществува дадения елемент:



Фигура 18: модифицираме на елемент от база данни

друго важно нещо за отбелязване е че контролера очаква Id като параметър в самата заявка.

* Delete: Последният вид HTTP заявка която ще разгледаме е „Delete“, името на метода подсказва за какво служи дадената заявка, а именно за триене на елементи от базата. Отново е необходимо да знаем кой елемент искаме да изтрием и съответно пак използваме ключа, ето как изглежда метода приложен в контролер:



Фигура 19: изтриване на продукт от база данни

#### 3.2.3.4 Завършеното API

След приключването със създаването на контролерите и техните репозитории имаме API готово за използване във Front End системата.

## 3.3 Имплементация на Front End системата

Както вече е споменато Front End частта на проекта е базирана на платформата: “Angular”. За да се създаде един Angular проект е необходимо да се разбере каква е неговата файлова структура, какво достъпва какво и кога го достъпва.

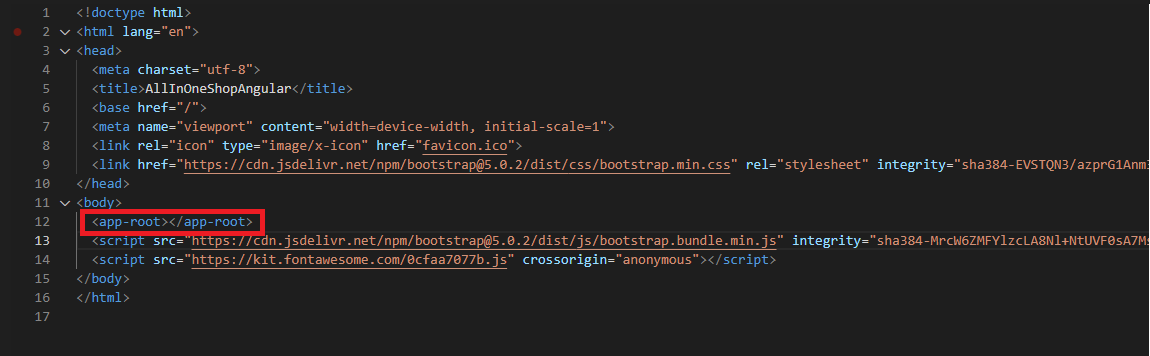
### 3.3.1 Файлова структура на проекта

Файловата структура на едно Angular приложение включва много различни файлове, които са организирани в определени директории. Ето някои от основните директории и файлове, които се срещат във всяко Angular приложение:

#### 3.3.1.1 „src“ директория в Angular

Всяко Angular приложение има най-малко един коренов директорен файл с име src. В този директорен файл се съдържат всички изходни файлове на проекта, включително TypeScript файлове, HTML шаблони, CSS файлове, изображения и други ресурси.

В кореновата директория се съдържа main.ts, който е точката на вход на приложението, и index.html, който е HTML документа, който се зарежда в браузъра и който съдържа статичните елементи на приложението. В index.html се добавят всички визуални библиотеки които ще се използват в проекта, също така от тук започва визуализацията на самия проект, това се случва чрез извикването на началния компонент на проекта чрез т.нар selector, ето как изглежда това:



Фигура 20: Използване на selector в Angular

Съответно този „<app-root></app-root>“ е началото на проекта, оттам се викат останалите компоненти.

#### 3.3.1.2 „src/app“ директория в Angular

Това е директорията, където се намират всички компоненти, сървиси и други файлове, които са специфични за приложението. Тук е средата за работа в един Angular проект.

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 

### 3.3.2 Routing в Angular

Routing в Angular се използва за манипулиране на състоянието на приложението в зависимост от URL адреса в браузъра. С помощта на routing можем да зареждаме различни компоненти и да показваме различна информация на потребителя в зависимост от това къде се намира в приложението.

#### 3.3.2.1 Routing чрез navbar в Angular

Имплементирането на routing чрез navbar в Angular е полезно, тъй като позволява на потребителя лесно да навигира в различните секции на приложението. Това означава, че вместо да се налага да зарежда отново цялата страница, когато потребителят иска да премине към друга част от приложението, можем да заредим само необходимите компоненти за тази секция, което ще подобри бързодействието на приложението. Освен това, имплементирането на routing чрез navbar прави навигацията в приложението по-лесна и интуитивна за потребителя, тъй като той винаги има достъп до навигационното меню, където може да избере желаната секция. Изграждането на Routing чрез navbar в Angular изисква следните стъпки:

1. Първата стъпка при имплементирането на routing чрез navbar в Angular е създаването на самия navbar, в случая той изглежда така:

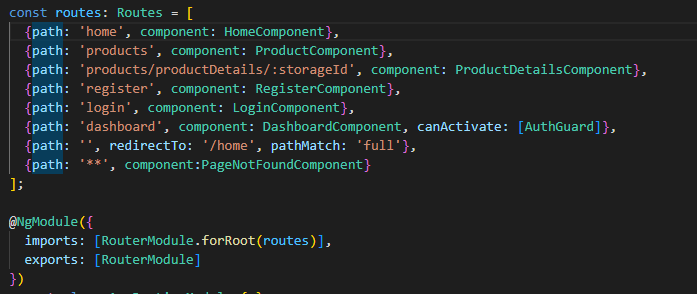


Фигура 21: използван navigation bar

1. Следващата стъпка е създаването на компонентите към които ще води този navbar, в случая компонентите са:

* “home” - началната страница
* “show-product” - тук ще се виждат всички продукти от базата данни
* “register” - оттук потребителите ще могат да се регистрират в системата
* “login” - този компонент ще позволява на потребителите да влязат във вече създадените си профили

1. Последната стъпка за имплементиране на routing чрез navbar в Angular е дефинирането на различните маршрути и това към кои компоненти ще водят те. Това е възможно да се постигне чрез т.нар RouterModule, дефинирането на маршрутите изглежда така:



Фигура 22: RouterModule

### 3.3.3 Свързване на Angular с Back End

Свързването на Angular приложението с база данни ни позволява да запазваме данните и да ги предоставяме на потребителите на приложението, което е от съществено значение за персистентността на данните и динамичността на приложението. Освен това, свързването на Angular приложението с база данни ни позволява да управляваме данните и да осигурим по-добра сигурност и ефективност в работата с тях. Тази връзка се случва чрез гореспоменатите service-и в Angular.

#### 3.3.3.1 Използване на ендпоинти дефинирани в Backend

Първата стъпка след създаването на “service”, който ще свързва Angular приложението с Back End частта от проекта е дефинирането на базовия URL адрес на създадения Backend API, това се случва по следният начин:



Фигура 23: базов URL адрес за свързване с Back End

След дефинирането на базовият адрес вече можем да достъпваме самите ендпоинти създадени в контролерите на Back End-а, достъпването на ендпоинтите се случва по следния начин:



Фигура 24: използване на ендпоинтове в “service”

#### 3.3.3.2 Изпращането на HTTP заявки от Angular към Backend

За изпращане на HTTP заявки от Angular към Backend се използва модула HttpClient. Този модул предоставя възможност за изпращане на заявки към сървърни API-та чрез HTTP протокола. За да използваме HttpClient модула първо трябва да го импортираме, импортирането в Angular се случва по сленият начин:



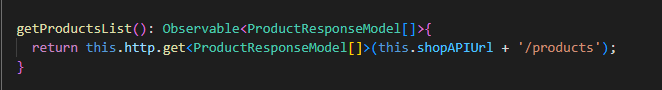
Фигура 24: импортиране в Angular

След импортирането на модула трябва да го инжектираме в компонента в който искаме да го използваме, това се случва чрез конструктора на дадения компонент и изглежда така:



Фигура 25: инжектиране в Angular

Пример за изпращане на HTTP GET заявка към Backend, използвайки HttpClient, може да изглежда по следния начин:



Фигура 26: използване на HttpClient модула

Методът getProductList() връща Observable съдържащ масив от тип ProductResponseModel, този Observable може да бъде абониран в компоненти, за да се получат данните от Backend-а. По този начин свързахме Back End-a с Front End-a. Следва обяснение на това какво е Observable и какви са тези Response модели.

#### 3.3.3.3 Observable в Angular

Observable е тип от RxJS (Reactive Extensions for JavaScript), който се използва за асинхронно получаване на данни. В Angular, този тип се използва за обработка на HTTP заявки към Backend, но също така може да се използва за асинхронна обработка на данни в приложението.

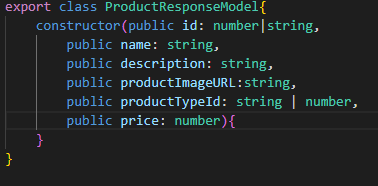
Observable в Angular е особенно полезен при работа с HTTP заявки, тъй като те обикновено са асинхронни и може да се използва Observable, за да се изчака отговор от сървъра, преди да се продължи с обработката на данните в приложението.

#### 3.3.3.4 Създаване на Response модели

Response моделите в Angular са класове, които се използват за представяне на отговорите, получени от API-то във формат JSON. Те се използват за организиране и манипулиране на данните, върнати от API-то в Angular приложението.

Response моделите са дефинирани със пропъртита, които трябва да съвпадат с пропъртитата на моделите от Back End-a.

Създаването на Response модели в Angular е важна практика, защото това улеснява работата с отговорите от API-то, към което приложението комуникира. Response моделите могат да помогнат за улесняване на четимостта на кода, подобряване на валидността на данните, улесняване на рефакторирането, подобряване на производителността и улесняване на тестването на приложението. Въвеждането на Response модели може да намали броя на грешките, които се появяват в процеса на разработване на приложението. Пример за това как изглежда един response модел:

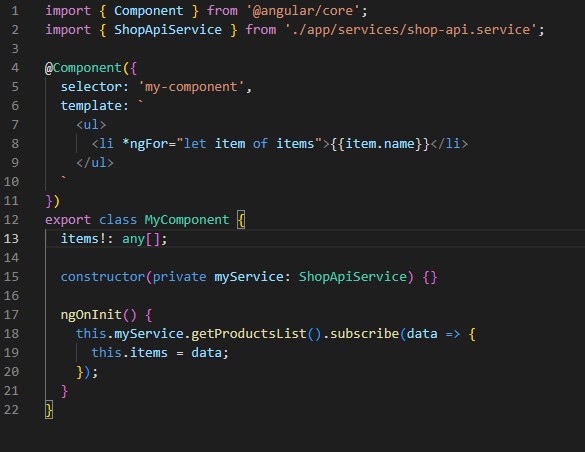


Фигура 27: примерен Response модел

### 3.3.4 Визуализиране на информацията получена от Back End

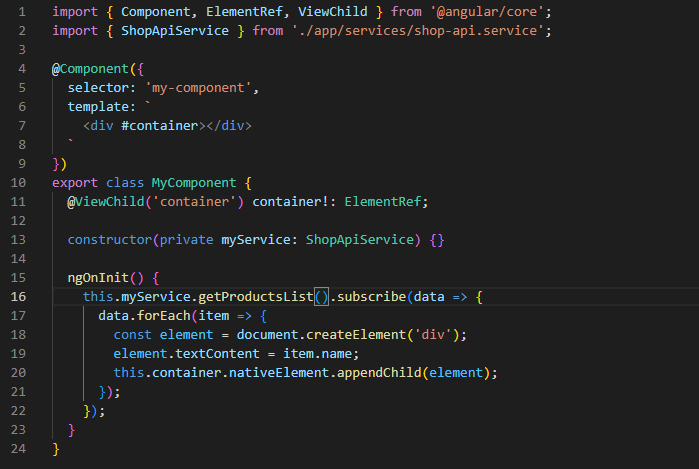
За да се визуализира информация, получена от Backend чрез сервиз в Angular, може да се използва шаблониране или генериране на HTML елементи динамично чрез кода на компонента. Примери за това:

1. Използване на шаблониране: В компонента се инжектира сервиза, след което се изпраща заявка към Backend за информация. Отговорът се получава във формат JSON и се съхранява в променлива. След това се използва директивата \*ngFor за да се итерира през списъка с информацията и да се генерират HTML елементи за всяко едно от тях. Например:



Фигура 28: шаблониране в Angular

1. Генериране на HTML елементи динамично чрез кода на компонента: В компонента отново се инжектира сервиза и се изпраща заявка към Backend за информация. Отговорът се съхранява в променлива. След това се генерират HTML елементи динамично чрез кода на компонента, като се използва DOM манипулацията. Нагледно:



Фигура 29: генериране на HTML елементи динамично чрез кода на компонента в Angular

### 3.3.5 Автентикация и регистриране на потребителите

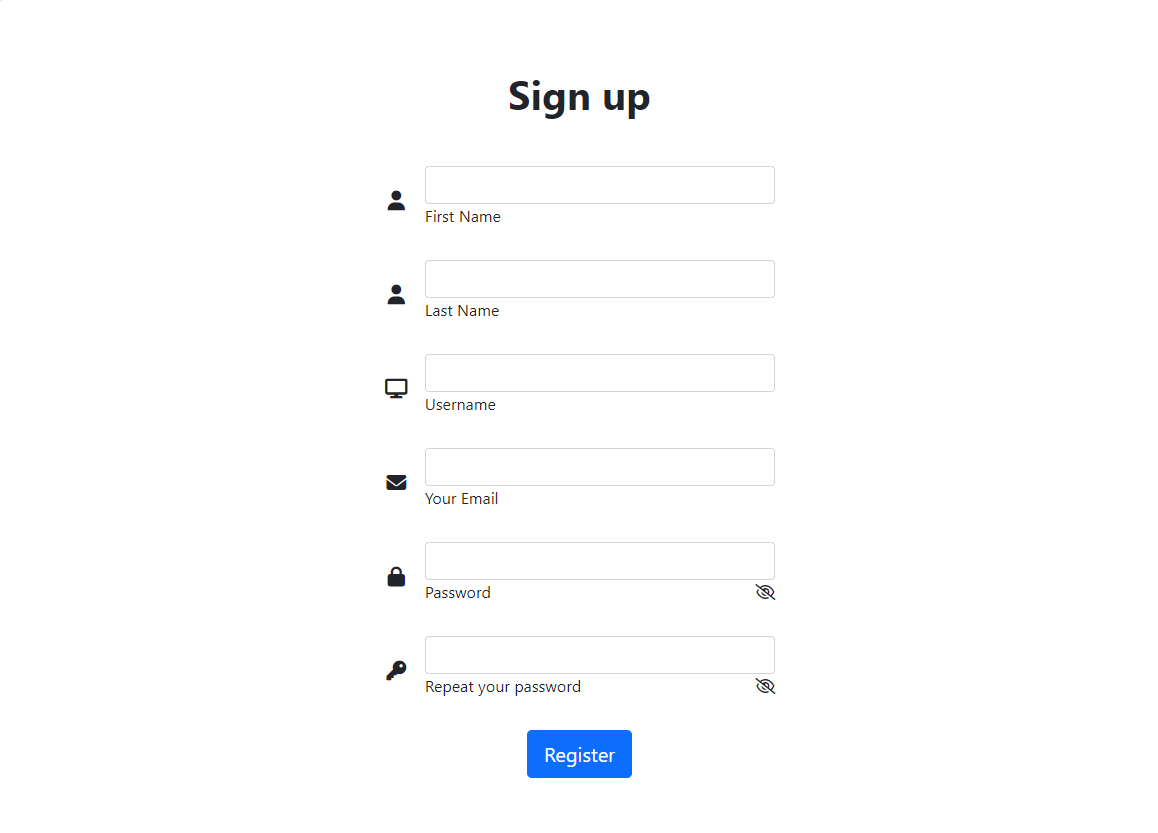
Автентикацията е важен аспект от сигурността на уеб приложенията и представлява процеса на идентифициране на потребителите, които искат да използват приложението, и проверка на техните права за достъп до защитени ресурси.

Използването на автентикация в Angular е от съществено значение за сигурността на уеб приложенията. Това не само предотвратява неоторизиран достъп до защитени данни, но също така помага да се предотвратят злоупотреби и забавяне на приложението поради прекомерно използване от страна на неупълномощени потребители.

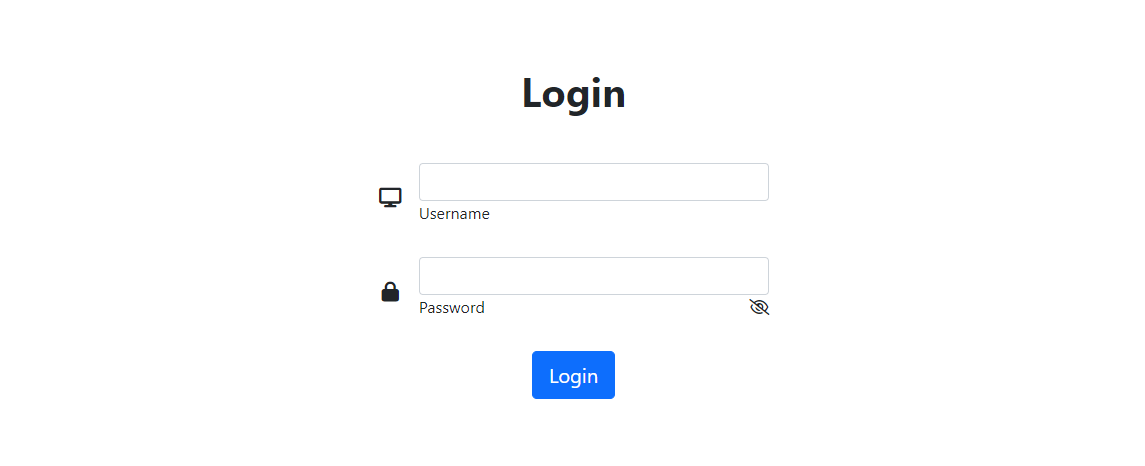
Освен това, автентикацията позволява на приложението да използва персонализирани настройки за потребителите, като например настройки за езика, предпочитанията на потребителя и др.

#### 3.3.5.1 Създаването на необходимите форми

Първата стъпка при автентикация и регистриране на потребителите, е създаването на съответните форми(forms), в случая формите изглеждат по следния начин:

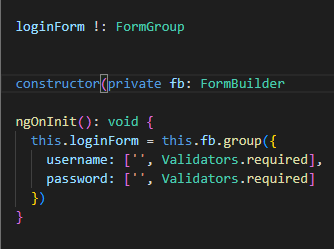


Фигура 30: форма за регистриране на нови потребители



Фигура 31: форма за автентикация на потребители

Разбира се информацията изпратена чрез тези форми бива валидирана чрез FormGroup структурата, която в себе си съдържа FormControl елементи отговарящи за всяко поле на формата, като всеки FormControl си има отделни валидатори(Validators). Цялото нещо бива сглобено от т.нар FormBuilder, ето как изглежда структурата:

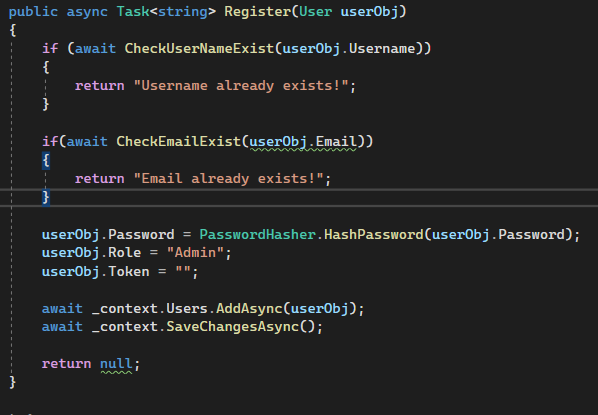


Фигура 32: валидиране на форми в Angular

#### 

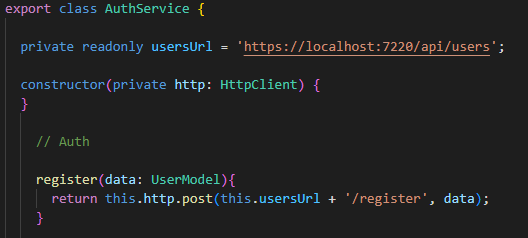
#### 3.3.5.2 Регистриране на нови потребители

Създаването на формата за регистрация на нови потребители е необходимо да я свържем с BackEnd частта на проекта. Но преди това трябва да създадем метод в контролера на потребителите за добавяне на нов потребител в базата данни. В него ще се проверява дали вече няма регистриран потребител със същия имейл и потребителско име. Също така преди да се регистрира клиента в базата данни, неговата парола ще бъде криптирана. Регистрирането в базата данни изглежда така:



Фигура 33: регистриране на потребител в базата данни

След създаването на ендпойнт за регистриране на потребител, вече може да се използва и във Front End-а, като се създаде “service” за автентикация. Ето как може да се регистрира клиент чрез т.нар AuthService:



Фигура 34: свързване на метода за регистриране с Front End

Последната стъпка е викането на този сървис при натискането на “submit” бутона на формата за регистрация. Това отново се случва, чрез т.нар инжектиране на зависимости, което ни позволява да извикаме готовия метод за регистрация, като му се поддаде стойността на формата. Ето как се извиква този метод:

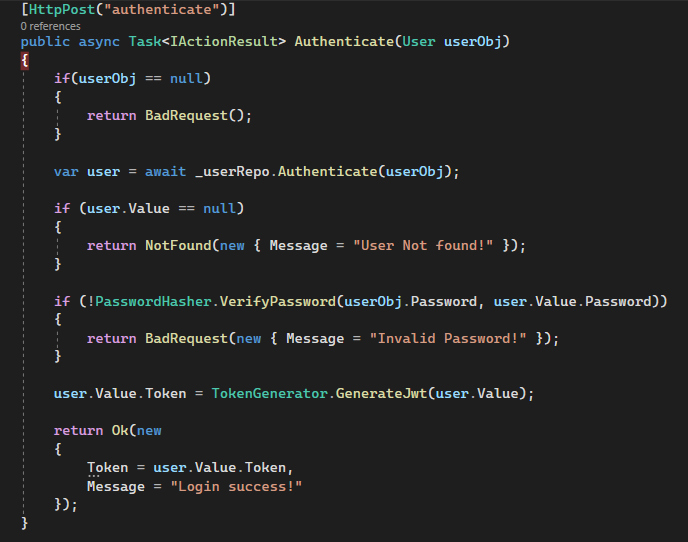


Фигура 35: регистриране на потребители чрез service

#### 3.3.5.3 Автентикация на потребители

За автентикация на потребителите отново е необходим метод в Backend контролера на потребителите.

Метода изглежда по следния начин:



Фигура 36: автентикация в Backend контролер

Първото нещо което прави този метод е да провери дали има входни данни(за допълнителна защита), следващото нещо за проверка е дали този потребител е вече регистриран в базата. За това отговаря репозиторията за потребители тъй като само оттам може да се достъпи базата данни. Ако потребителя е в системата се проверява дали паролата му съвпада с тази в записана в базата като преди това трябва да се декриптира. Ако всичко съвпада се генерира т.нар JWT Token.

#### 

#### 3.3.5.4 JWT Token

JWT (JSON Web Token) е формат за представяне на компактни и самодостатъчни заявки за сигурност на данни, основаващи се на JSON формат. JWT токените се използват често в мрежови приложения като метод за удостоверяване на потребителите и разпределяне на достъпа до ресурси.

JWT токените се състоят от три части: заглавие (header), payload и сигнатура (signature). Заглавието съдържа информация за алгоритъма за криптиране, използван за създаване на токена, а payload-a съдържа информация за потребителя и разрешенията му. Сигнатурата се използва за проверка на токена и удостоверяване на неговата автентичност.

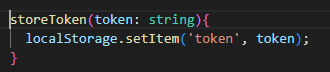
#### 3.3.5.5 Автентикация чрез JWT Token

След като Backend частта генерира JWT Token-а с който ще се автентикира дадения потребител, трябва да го използваме в Frontend частта. Този token го получаваме като резултат от бекенда чрез гореспоменатия AuthService. След като го получим трябва да го запазим някъде, най-удобното място за това в случая е локалното хранилище на браузъра(localStorage). Запазването се случва отново чрез сървиса за автентикация по следния начин:



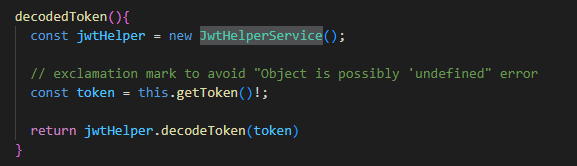
Фигура 37: запазване на JWT Token в локалното хранилище на браузъра

Като самият метод за запазването изглежда така:



Фигура 38: запазване на JWT Token в сървис

След запазването на token-a е необходим начин да разберем какво има на него, какво информация дава той. Тук идва ролята на “JwtHelperService”, чрез този сървис можем да дешифрираме получения от бекенда token и да разберем каква информация има на него, дали е валиден и т.н. Това се случва по следния начин:



Фигура 39: дешифриране на JWT Token

След като имаме дешифриран token вече можем да визуализираме името на автентикирания потребител в навигационния бар, ето как изглежда това:

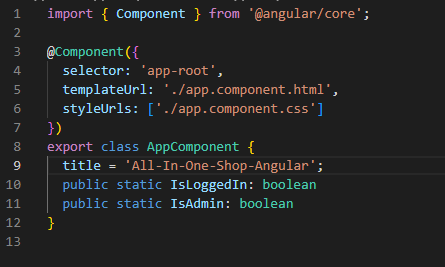


Фигура 40: актуализиране на навигационния бар при автентикиране на потребител

Това означава че успешно сме се автентикирали.

#### 3.3.5.6 Права на различните потребители

Всички потребители ще имат различни права, базирани на тяхната роля. Ролите биват: “User” и “Admin”. Начина по който се определя до какво имат достъп отделните потребители е чрез две променливи намиращи се в началната точка на Angular проекта а именно: “app” компонента, ето как изглеждат тези променливи:



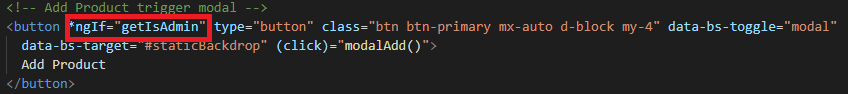
Фигура 41: статични променливи определящи правата на отделните потребители

При автентикация на определен потребител се актуализират тези две променливи, като това позволява да се визуализират само определени части от приложението базирано на това с каква роля е дадения потребител и дали е автентикиран. Ето как се актуализират глобалните променливи:



Фигура 42: актуализация на глобалните променливи

След като се определи с какви права е дадения потребител можем да крием/разкриваме отделните части на HTML-а чрез “\*ngIf”, например:



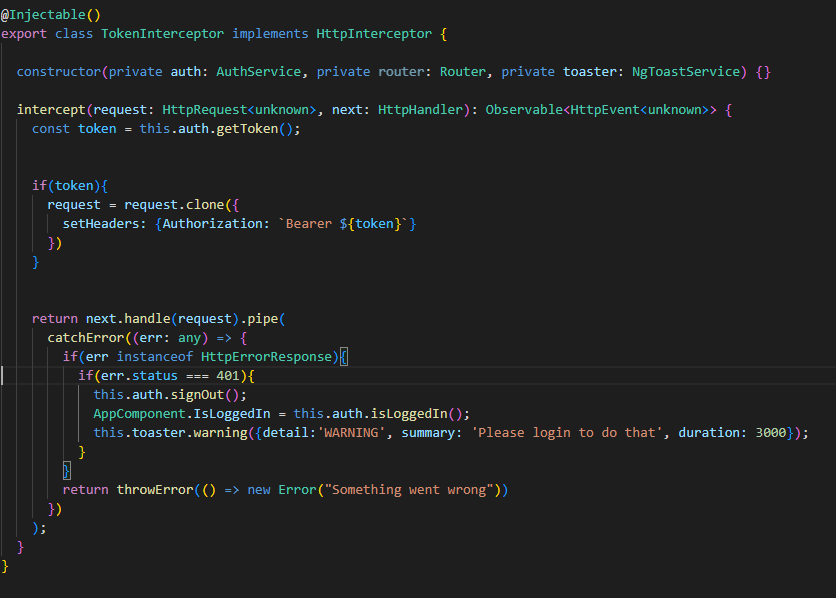
Фигура 43: използване на \*ngIf в Angular

#### 3.3.5.7 Interceptors в Angular и как работи автентикацията при тях

Interceptors са една от функционалностите на Angular HTTP клиента, която позволява да се манипулират заявките и отговорите на HTTP заявките преди да достигнат до приложението.

Interceptors са междинен слой между Angular HTTP клиента и сървъра, който може да променя заявките и отговорите преди да бъдат изпратени или получени от сървъра. Това прави интерсепторите много полезни, когато се работи с HTTP заявки, тъй като могат да се използват за автоматично добавяне на headers, промяна на URL адреси, обработка на грешки, добавяне на логика за автентикация и оторизация и много други.

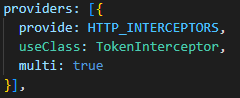
За да се създаде един interceptor трябва да се създаде клас, който имплементира интерфейса HttpInterceptor. Този клас трябва да има метод intercept(), който ще бъде извикан от Angular HTTP клиента, когато се прави HTTP заявка. В този метод могат да се направят всички необходими манипулации на заявката или отговора. Ето как е добавен authorization header в създадения проект:



Фигура 44: добавяне на authorization header

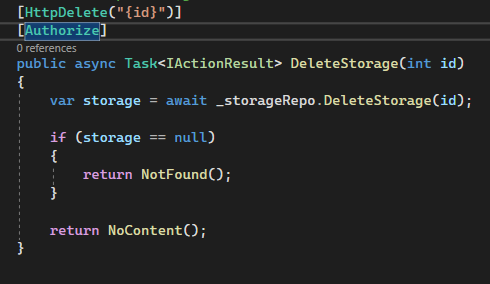
Този интерсептор ще добави Authorization header с токена на аутентикация, който е записан в localStorage на клиента, към всяка HTTP заявка, която се изпраща към сървъра.

За да работи дадения interceptor трябва да се добави в Angular app модула, като се използва provide ключовата дума и HTTP\_INTERCEPTORS константата. Това казва на Angular да използва създадения интерсептор, когато се правят HTTP заявки.



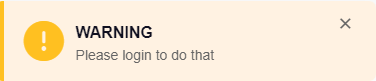
Фигура 45: добавяне на interceptor в app модула

Последното нещо което трябва да се направи за да работи такъв вид автентикация е да се добави атрибута [Authorize] на методите които не искаме да бъдат достъпвани без автентикация. Този атрибут се добавя върху различните методи на контролерите в бекенда. Например:



Фигура 46: автентикация чрез interceptor

Тоест сега ако потребител се опита примерно да изтрие някой продукт, заявката ще бъде спряна и ще се появи следния попъп:



Фигура 47: попъп появяващ се при невалидна автентикация

# Глава 4. Ръководство за потребителя

Проектът е достъпен като сорс код от следното Github хранилище: [All-In-One-Shop](https://github.com/Lacho123/All-In-One-Shop)

## 4.1 Системни изисквания

## 

За да може да стартирате и използвате Angular приложение, което използва Entity Framework за backend, имате нужда от компютър с определени системни изисквания. Тук са някои от основните изисквания:

1. Операционна система: Windows 10, Linux или MacOS.
2. Инсталиране на Node.js - това е необходима зависимост за Angular, която се използва за компилацията на приложението. В момента на написване на този отговор, последната стабилна версия на Node.js е 16.x.
3. Инсталиране на Angular CLI - това е инструментът, който ще ви позволи да управлявате вашето Angular приложение.
4. Инсталиране на .NET Core SDK - това е необходима зависимост за Entity Framework, която се използва за създаване на вашата backend част.

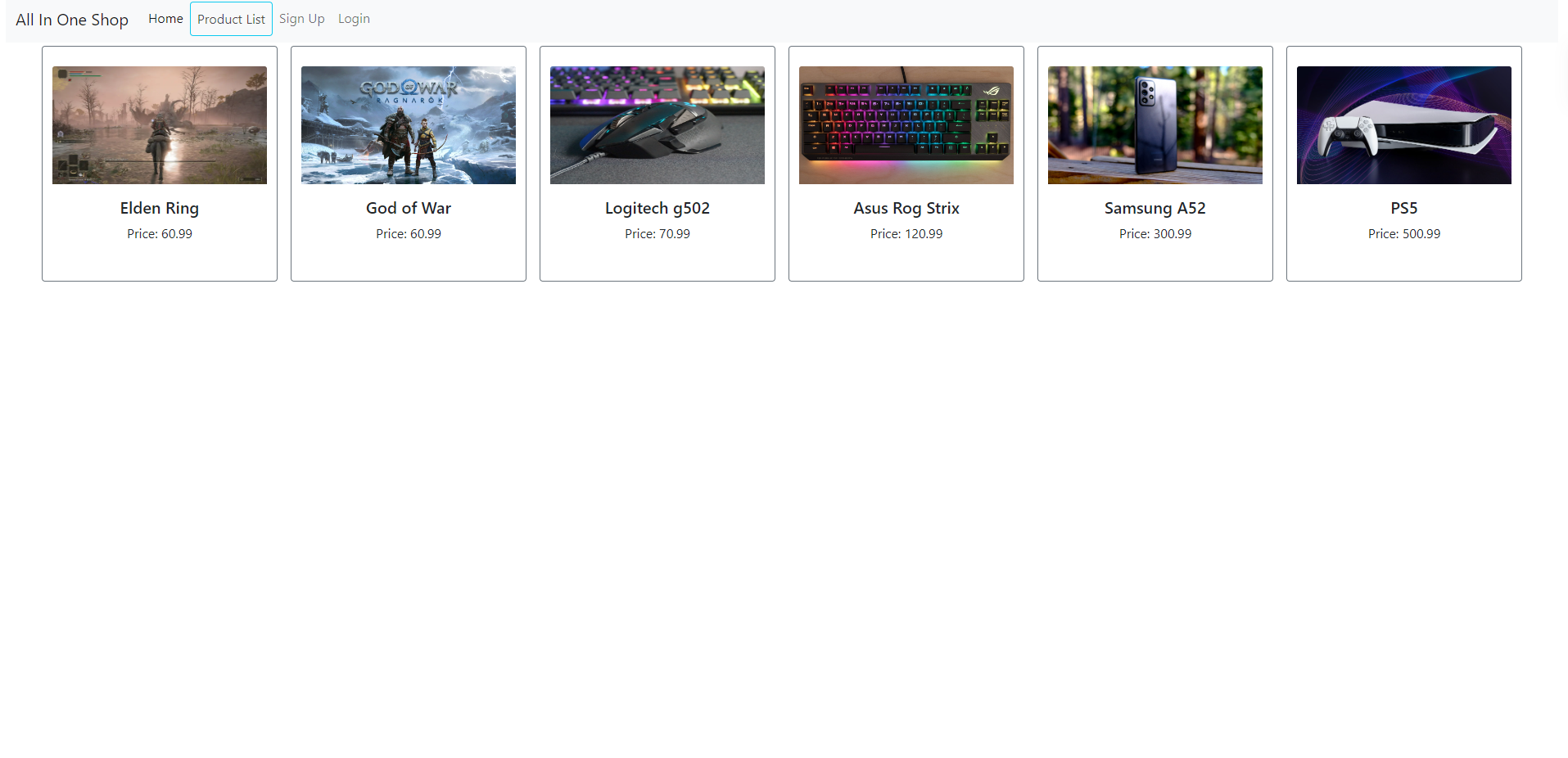
## 4.2 Описание на различните страници от уеб приложението

При стартиране на сайта от потребителя, се зарежда началната страница или home страницата, на ния е изобразен т.нар slide box, показващ последните добавени продукти. Ето как изглежда на практика:



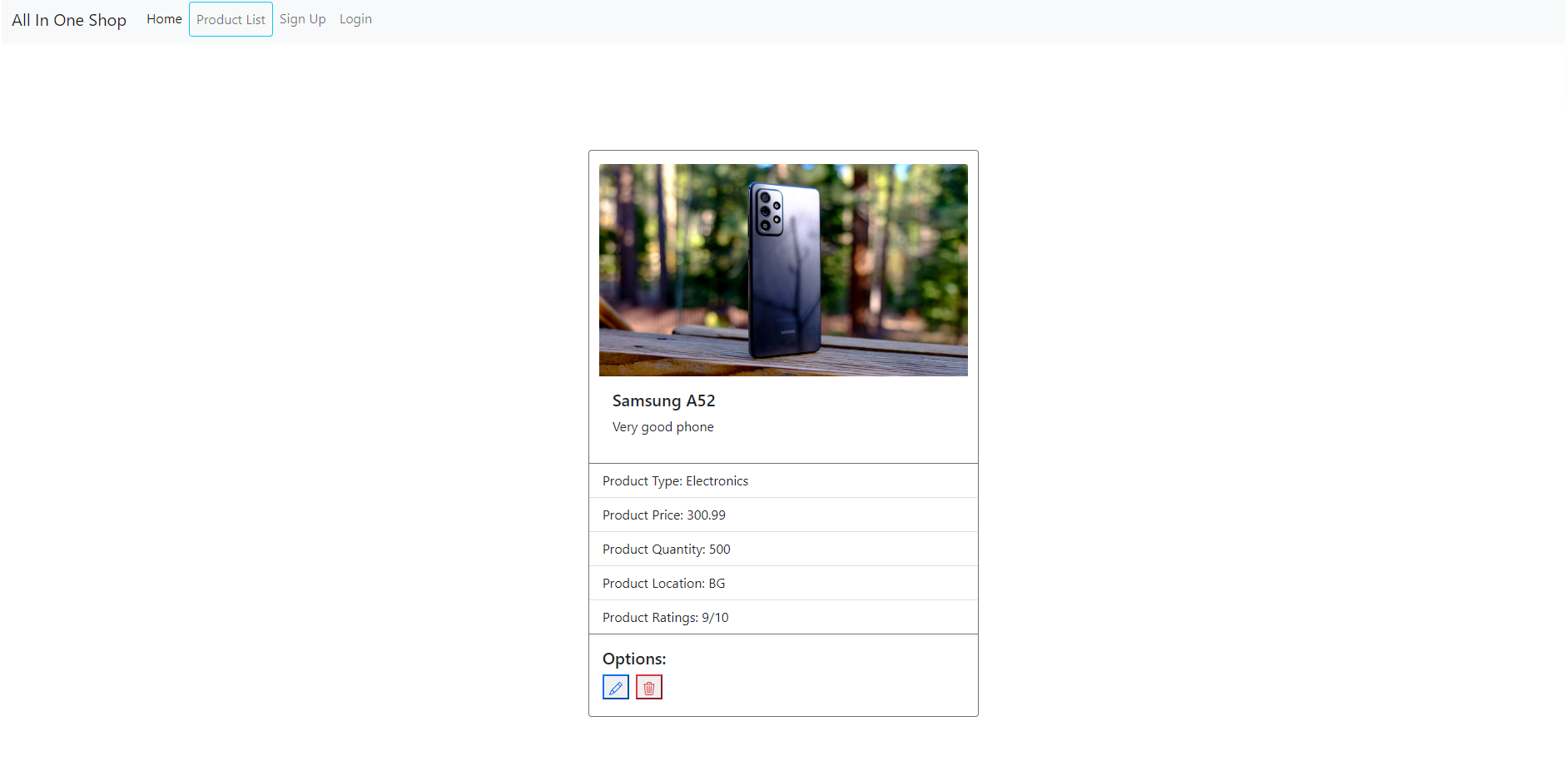
Фигура 48: начална страница

От тук потребителя може да избере коя от основните страници на сайта да посети чрез навигационната лента а именно: “Product List”, “Sign Up” и “Login”. Product List частта на приложението визуализира всички продукти в базата данни, страницата изглежда така:



Фигура 49: страница с всички продукти от базата

Всички продукти съдържат в себе си допълнителна страница, показваща допълнителна информация за дадения продукт, тя се активира при кликването върху някои от продуктите и изглежда по следния начин:



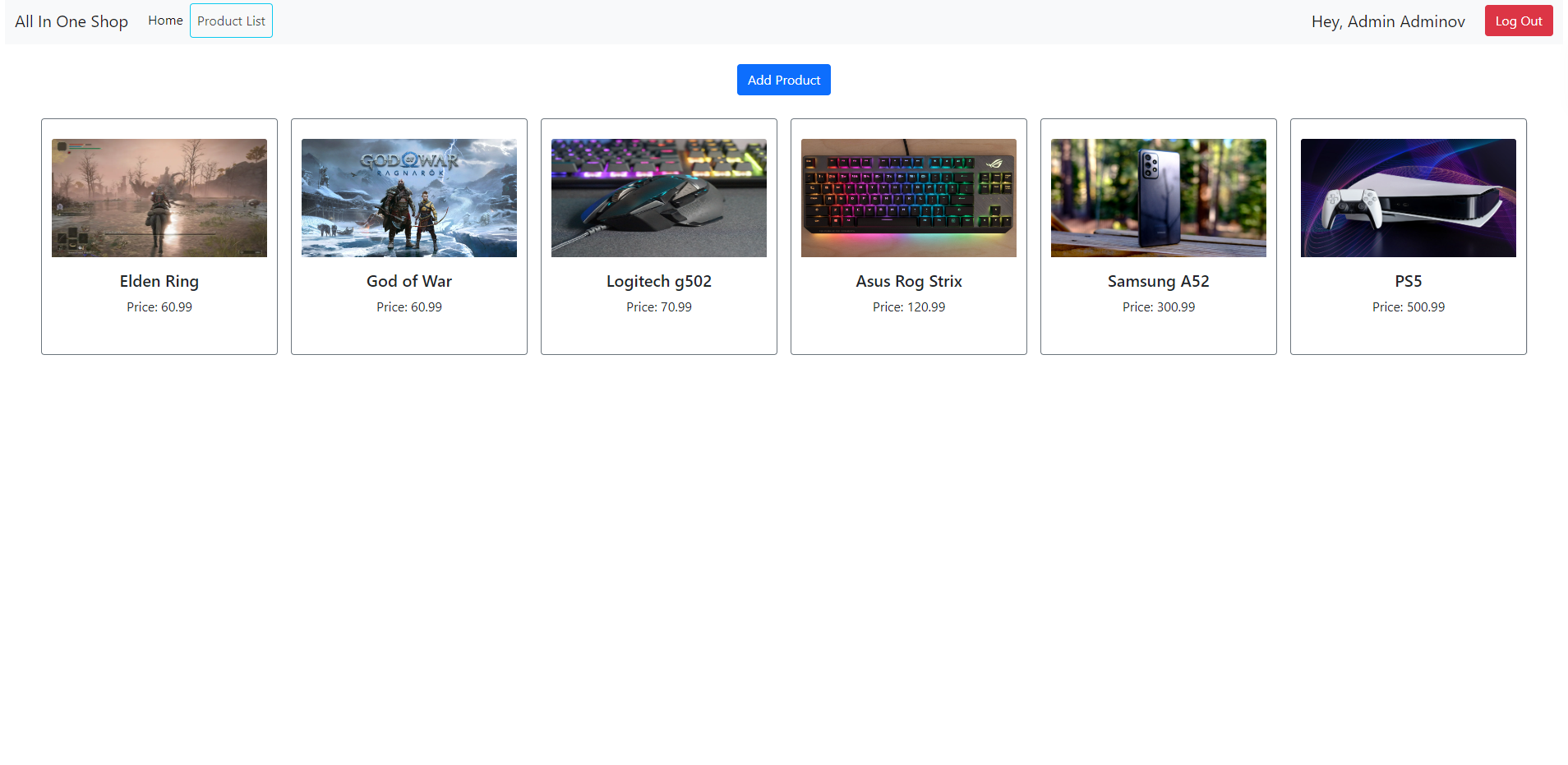
Фигура 50: страница съдържаща подробности за дадения продукт

От тук потребителя може да модифицира или да трие продуктите, само че за да има това право трябва да се регистрира в системата.

Регистрирането се достъпва от Sign Up бутона на навигационния бар, изгледа на страницата може да се види на Фигура 30. След регистрацията потребителя автоматично влиза в системата и вече може да прави промени по продуктите.

Ако потребителя е с администраторски права в страница с всички продукти от базата ще му се появи допълнителен бутон “Add Product”, чрез който може да добавя изцяло нов продукт в базата данни.

Страницата ще изглежда така:



Фигура 51: администраторски права показани нагледно

При натискането на бутона “Log Out” потребителя се връща на началната страница и вече е излязъл от профила си, позволявайки му да започне приключението си наново.

# Заключение

Дипломният проект за унифициран онлайн магазин е успешно завършен. Използвайки Angular и Entity Framework, беше разработена уеб платформа, която позволява на потребителите да пазаруват в удобството на своя дом или офис.

Проектът съдържа множество функционалности като: добавяне, модифициране и триене на продукти, детайлни страници на продуктите, както и напълно имплементирана автентикационна система. Освен това, е изградена визуална част на сайта, която предоставя гъвкав и лесен за използване интерфейс.

Има много възможности за усъвършенстване на проекта, като подобряване на дизайна на страниците и имплементиране на кошница за пазаруване. Може да се добавят и допълнителни функционалности, като възможност за отзиви на потребителите, интеграция с плащането и други.

И най-важното, дипломният проект доказа успешно способността на автора да използва високо ниво на знания и умения за изграждане на сложни уеб приложения. В резултат на това, проектът може да послужи като основа за бъдещи разработки на онлайн магазини.

# Информационни източници

1. [Entity Framework documentation](https://learn.microsoft.com/en-us/ef/)
2. [Angular documentation](https://angular.io/docs)
3. [TypeScript documentation](https://www.typescriptlang.org/docs/)
4. [JWT documentation](https://jwt.io/introduction)
5. [Rfc2898DeriveBytes](https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rfc2898derivebytes?view=net-7.0)
6. [RNGCryptoServiceProvider](https://learn.microsoft.com/en-us/dotnet/api/system.security.cryptography.rngcryptoserviceprovider?view=net-7.0)
7. [Claim Class](https://learn.microsoft.com/en-us/dotnet/api/system.security.claims.claim?view=net-7.0)