**СОФИЙСКА ПРОФЕСИОНАЛНА ГИМНАЗИЯ ПО ЕЛЕКТРОНИКА „ДЖОН АТАНАСОВ“**

**ДИПЛОМЕН ПРОЕКТ**

**Тема: „**Система за организация на продукти и функционалност в пакети и планове“

**Практическа част: „**Разработка на Система за организация на продукти и функционалност в пакети и планове“

Дипломант: Даниел Емилов Тодоров 12 „В“ клас

Професия: 481030 „Приложен програмист“

Специалност: 4810301 „Приложно програмиране“

Ръководител на дипломен проект: инж. Любица Димитрова

Дипломант:

*/подпис/*

Ръководител на Дипломен проект:

*/подпис/*

София

2023

Съдържание

[Теоретична част 1](#_Toc132626080)

[Увод 1](#_Toc132626081)

[Основна част 3](#_Toc132626082)

[Описание на продукта 3](#_Toc132626083)

[Back-End (Server) 4](#_Toc132626084)

[.NET Core и C# 4](#_Toc132626085)

[ASP.NET Core Web API 4](#_Toc132626086)

[Entity Framework 5](#_Toc132626087)

[База данни 6](#_Toc132626088)

[Модели 9](#_Toc132626089)

[Контролери 10](#_Toc132626090)

[Сървиси 12](#_Toc132626091)

[Структура 16](#_Toc132626092)

[Front-End (Client) 17](#_Toc132626093)

[Angular 17](#_Toc132626094)

[HTML 17](#_Toc132626095)

[CSS 18](#_Toc132626096)

[Typescript 18](#_Toc132626097)

[Компоненти 18](#_Toc132626098)

[App Component 19](#_Toc132626099)

[Right-Navbar Component 21](#_Toc132626100)

[Contract Type Component 22](#_Toc132626101)

[License Product Component 24](#_Toc132626102)

[Installation Type Component 25](#_Toc132626103)

[Selection Component 26](#_Toc132626104)

[Структура 29](#_Toc132626105)

[Заключение 30](#_Toc132626106)

[Практическа част 31](#_Toc132626107)

[Използвана литература 32](#_Toc132626108)

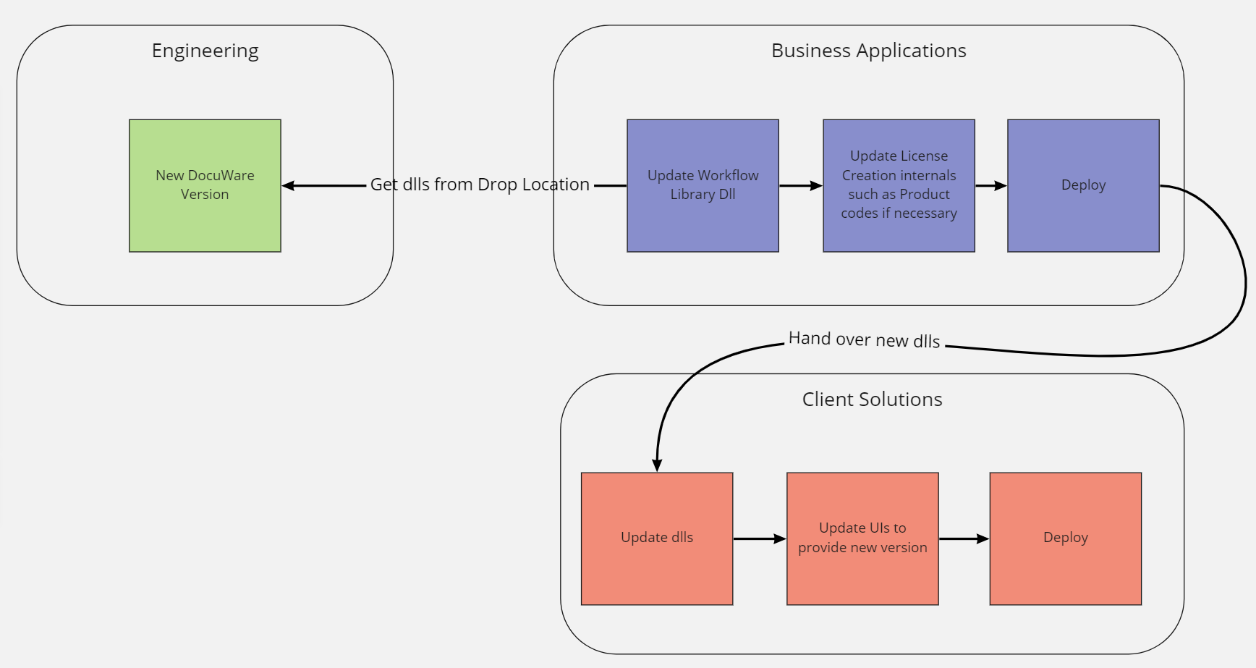
# Теоретична част

# Увод

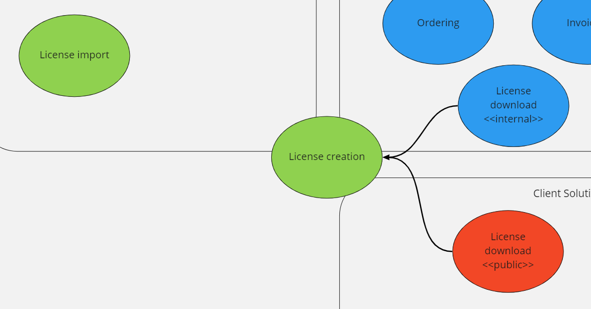
В днешно време, документацията е важна част от дейността на една организация. С увеличаване на броя на служителите в една компания се увеличава и броят на документите, което от своя страна увеличава обема на работа за обработка. Решение за сериозно увеличаване на ефективността за обработка на документи е използването автоматизирани процеси, които да обработват документи с бързина и ефективност и без да се налагат тромави ръчни работни процеси или трудна за намиране информация.

За тази цел е подходящо приложението Docuware, което представлява система за управление на документи (Document Management System - DMS) и е дигитално решение, помагащо на организациите да обработват, улавят, съхраняват, управляват и проследяват документи и цялостна автоматизация на процеса по обработка на документите. Чрез стриктно управление на бизнес информацията, компаниите могат да разработват процеси, които стартират, изпълняват и завършват по стабилен, предвидим и измерим начин. Продуктът Docuware е сложна система от множество модули и функционалности. Бизнес модела на Docuware се състои от продажба на основния продукт, допълнителни модули към него и брой потребители които могат да ползват системата. Отделните модули на Docuware, които могат да бъдат инсталирани зависят от лиценза закупен от съответния клиент. Такъв лиценз се издава конкретно за организацията на крайния потребител, закупила софтуера.

Внедряването на нови модули и функционалности, а също така пускането на нова версия на пазара води до сложен процес в разработката на продукта. Именно в момента проблемът е, че 3 екипа в организацията са зависими един от друг. Когато се пусне нова версия на Docuware, заедно с която се пуска и обновен сървис за лицензите, от съответен екип, следващият екип трябва да обнови сървисите си, спрямо промененият сървис, и когато е готов, изпраща промените на третия екип, който също трябва да променя собствените си сървиси спрямо това, което води до нежелана зависимост между тях.

Целта на този проект е да се оправи проблема със зависимостите, като се направи сървис за създаване на лицензи, който може да се разработва и ползва независимо от разработката на останалите модули.

Фигура 1 - Зависимост между екипите



Фигура 2 - Решение за зависимостта

Сървиса за издаване на лицензи се състои от следните части: административна част (конфигуриране на видове инсталации, видове продукти и видове договори) и самото създаване на лицензния файл. Този проект се занимава само с направата на административната част. Задачите за постигане на целта са следните:

* Да се създаде база данни
* Да се разработи приложение, служещо като сървърна част, което включва контролери, сървиси, модели и Entities класове необходими за базата данни.
* Да се разработи приложение, служещо за интерфейс, изобразяващ данните и да може да се борави с тях
* Накрая, всички тези компоненти да се внедрят в един сървис за създаване и конфигуриране на лицензи

# Основна част

## Описание на продукта

Продукта, който разработихме се нарича Docuware License Creator и е подобрена версия на предишното му издание. Приложението е Single-page application, което означава, че има един начален екран от който се достигат всички функционалности. Функционалностите са разделени на 2 главни части – конфигурационна част и модул за създаване на лицензи.

В конфигурационната част се намират функционалностите които отговарят за създаването и поддръжката на различните компоненти необходими за да се създаде лицензен файл. Тя се състои от:

* Дефиниране и преглеждане на видове договори
* Дефиниране и преглеждане на продуктите в системата които изискват лиценз
* Дефиниране и преглеждане на стандартни пакети от продукти

В модула за създаване на лицензи се намира цялата функционалност за самото създаване на лицензния файл. Той се състои от:

* Прилагане на инсталационен тип
* Прилагане на вид договор
* Даване на личната информация на потребителя - имейл, телефонен номер, компания
* Прилагане на версия на Docuware която използва потребителя
* Прилагане на специален Organization Key

Приложението се базира на Client-Service архитектурен шаблон и се състои от 3 главни и неразделни части – Database, Back-end (Server) и Front-end (Client). За комуникация между клиента и сървъра се използва HTTP REST комуникация, а връзката между сървъра и базата данни се осъществява с помощта на Entity Framework. Базата данни е основана на MySQL. Сървърната част е реализирана чрез Microsoft .NET, C# и ASP Web API. Клиентската част е реализирана чрез Angular Framework, който включва и езиците HTML, CSS и Typescript.

## Back-End (Server)

### .NET Core и C#

Приложението, което се разработи, е основано на програмния език C#, който работи с помощта на .NET Core. .NET Core е нова версия на .NET Framework, която е безплатна платформа за разработка с open source код с общо предназначение и поддържана от Microsoft. Това е cross-platform framework, който работи на операционни системи Windows, macOS и Linux. .NET Core може да се използва за изграждане на различни типове приложения като мобилни, десктоп, уеб, Cloud, IoT, machine learning, micro-services, игри и други. Старата версия - .NET Framework - за разлика от .NET Core, има някои ограничения. Например, тя работи само на платформата Windows, докато .NET Core е създаден специално да работи на няколко операционни системи и това съответно го прави по-достъпен. В днешно време е обичайно човек да има компютър с различна от Windows операционна система, което означава, че няма да има пряк достъп до функционалностите на .NET framework. Точно поради тази причина има нужда от единен framework, който да работи навсякъде. И като се има предвид това, Microsoft създава .NET Core, чиято основна цел е да бъде съвместим с различни платформи. Относно C#, това е език за програмиране, който е забележително мащабируем и лесен за поддръжка. Поради строгият характер на начина, по който трябва да се пише кода, програмите на C# са последователни, което ги прави много по-лесни за настройване и поддръжка от програми, които са написани с помощта на други езици. .NET Core и C# вървят ръка за ръка, понеже framework-а предоставя необходимите рестрикции и инструменти на езика за да работи качествено и да може да се разчита на него когато става дума за мащабни проекти.

Използвана литература – 1.

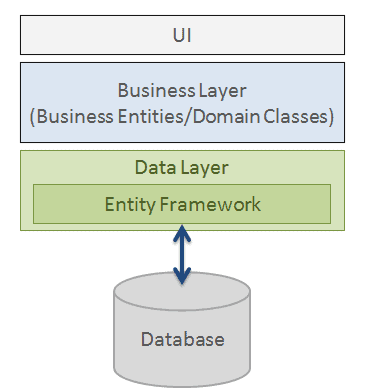
### ASP.NET Core Web API

ASP.NET Core Web API е framework за изграждане на уеб приложения и сървиси, който е основан на платформата .NET Core. Той позволява на софтуерните разработчици да създават RESTful приложения със .NET езици. Това означава, че с този framework могат да се правят API endpoints които да получават и да отговарят на HTTP заявки, като например GET, PUT, POST, DELETE. Също така, framework-a осигурява поддръжка за различни формати на данни, като JSON и XML. Една от основните характеристики на ASP.NET Core Web API е поддръжката на Dependency Injection (това е когато на даден клас през конструктора се подават променливите му без той да се занимава да създава или да има информацията как те са създадени), което дава възможността лесно да се интегрират сървиси в контролерите на приложението. В случая на този дипломен проект, цялата сървърна част е писана на C# и е основана на този framework. Със него се осъществи връзката между клиентската и сървърната част.

Използвана литература – 3.

### Entity Framework

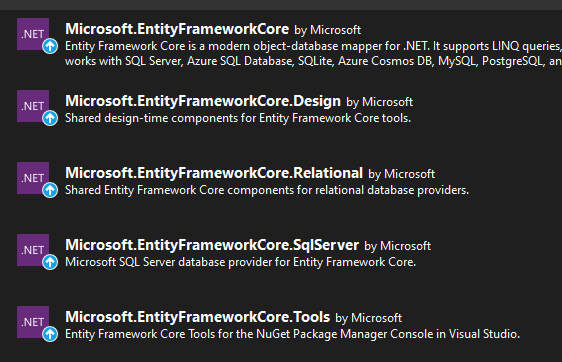
Entity Framework e Object-Relational Mapping (ORM) framework, който работи за .NET апликации. Той позволява да се работи с данните на база данни, директно използвайки обекти от .NET, без разработчика на софтуер да се фокусира върху съхраняването на данните в таблиците.



Съгласно фигурата, Entity Framework се вписва между бизнес частта и базата данни на приложението. Той може да запазва данни в property-тата на обекти и след това да ги трансформира в данни съвместими за SQL които да се съхранят в базата данни, а също така може и обратното – автоматично да взима информация от базата и да я трансформира в обекти на .NET.

Фигура 3 - App layers

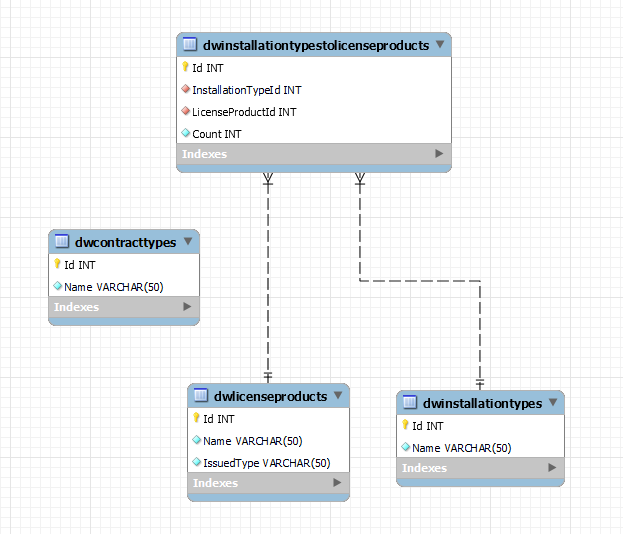
За да може да се използва Entity Framework в текущия проект, трябва да бъдат изтеглени няколко NuGet пакета:

Всеки един от пакетите допринася различни функционалности към Entity Framework, като например допълва System.Linq библиотеката, която е предоставена от самият .NET, с функционалности служещи за по-лесна обработка на данните, и също дава възможността да се пишат конзолни команди за миграции и други операции свързани с базата данни.

Фигура 4 - NuGet Пакети

Използвана литература – 2.

### База данни

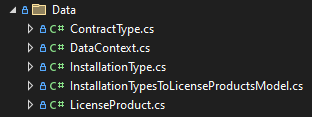
Базата данни на приложението е направена на езика MySQL с помощта на MySQL Server и MySQL Workbench. Ето и нейната ER Диаграма:

Фигура 5 - ER диаграма на базата данни

Тя се състои от 4 таблици. Всяка таблица има DW пред името си, което идва от конвенцията за именуване на DocuWare (DW = DocuWare). Първата таблица се казва DWContractTypes и в нея се съхраняват различните видове договори, състои се от ID и Name колони. Следващата таблица се нарича DWLicenseProducts и в нея се съхраняват различните видове продукти. Съставена е от ID, Name и IssuedType колони. Третата таблица се казва DWInstallationTypes и тя служи са съхраняване на видовете инсталационни типове. Има полета ID и Name. Последната таблица е така наречената Multi-Relational Table и в себе си държи информация за това кой продукт за кой инсталационен тип се отнася. Името и е DWInstallationTypesToLicenseProducts и в нея има следните колони: ID, InstallationTypeID – това е ForeignKey репрезентиращ ID-то на съответния инсталационен тип, LicenseProductID – пак ForeignKey сочещ към ID-то на продукта, и накрая Count – бройката от конкретния продукт в инсталационния тип.

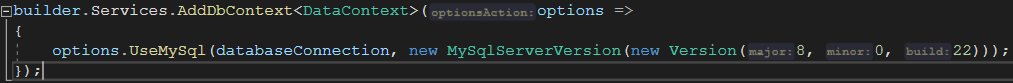
Подхода към проекта е Code First, което означава, че първо са писани моделите в Back-End и след това чрез миграции се създава базата заедно с нейните таблици отговарящи на моделите. В случая на нашето приложение, този подход е най-добрият ни вариант, понеже когато се предаде на друг екип, служителите в него просто ще трябва да направят миграция или да стартират стара миграция, без да се занимават да пишат базата данни на ръка.

Така изглежда папката в която се държат моделите за базата данни:



Фигура 6 - Подредба на класовете в папка “Data”

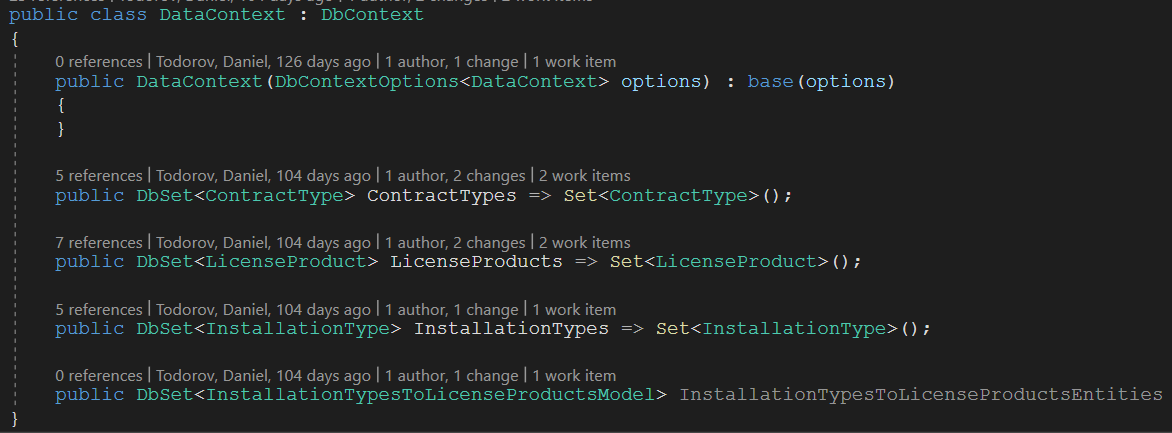
В нея освен самите модели се съдържа и специален клас DataContext който служи за достъп до таблиците на базата. Относно свързването на MySQL със сървърната част, то е направено като във Program класа, който е създаден при създаването на проекта, се вика команда със информацията за MySQL Server:



Фигура 7 - Инициализирането на DataContext в Program класа

“DatabaseConnection” променливата пази в себе си паролата и username-a на локалната връзка към MySQL Server.

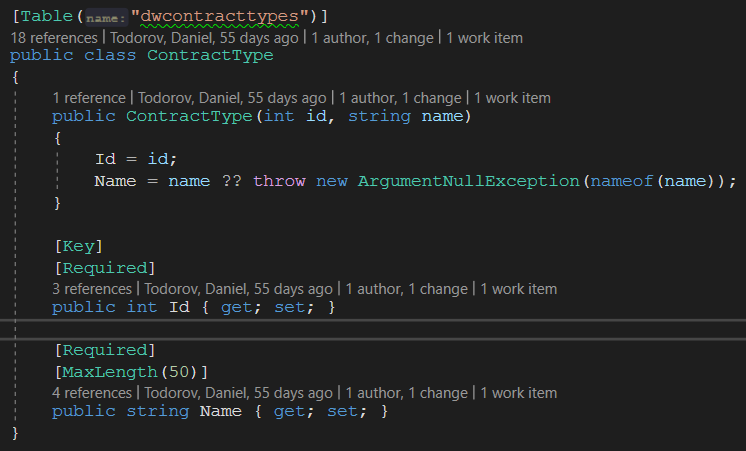
Нека сега преминем към класовете и първо ще разгледаме DataContext класа:

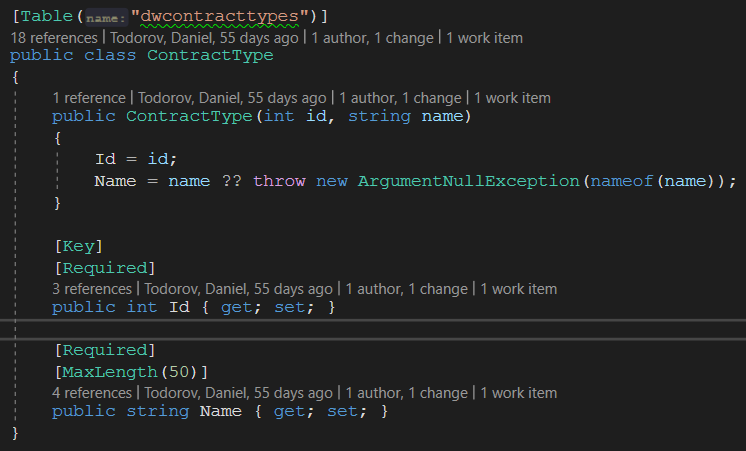


Фигура 8 - DataContext клас

Както казахме, той директно има достъп до базата данни, за това и наследява DBContext. Това е клас, предоставен от Entity Framework, който идва с готови методи, репрезентиращи различните възможни функционалности във MySQL – Add, Update, Delete, Get, SaveChanges и т.н.. Чрез тези неща може да се обхожда всичката информация в базата, може да се взима, трие, променя и запазва. DataContext класа има конструктор, който със Dependency Injection вкарва нужните данни като параметър и ги предава на конструктора от базовия клас (DBContext), и 4 декларирани полета, представляващи колекции от различните типове вкарани в базата.

Класовете модели са направени по един шаблон – имат конструктор и полета, с имена отговарящи на таблиците. Ето ContractType.cs за пример:





Фигура 9 - ContractType клас

Както казахме, в него има конструктор, приемащ нужните параметри - в случая ID и Name. Когато му се дадат, той валидира даденият Name аргумент да бъде със стойност, иначе програмата ще хвърли грешка, и след това ги присвоява на съответстващите клас променливи (ID няма нужда от проверка, понеже знаем че винаги ще има стойност). Клас полетата са ID и Name и си имат техните специфики за това, като какви да ги разпознава програмата и какви рестрикции да имат. Това е постигнато с анотациите:

* Означава променливата, че ще бъде запазена като Primary Key колона в таблицата
* Валидира променливата, че има стойност и не е null
* Валидира дължината на променливата, че няма да е повече от 50 символа

Освен тези анотации е използвана и още една при името на класа - . Тя обозначава, че класа ще бъде запазен като таблица с име ‘dwcontracttypes’ в базата данни.

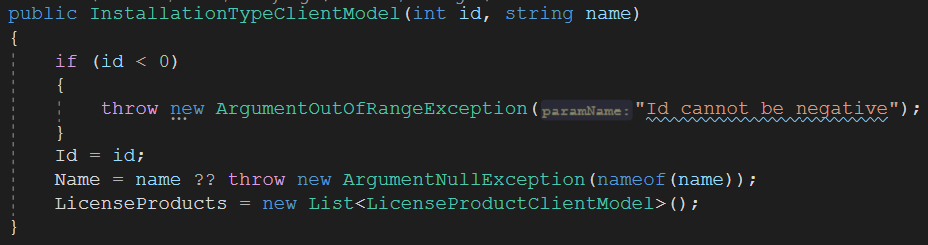
Разбира се, другите модели си имат специфичните за тях променливи и анотации.

### Модели

В сървърната част на приложението се изнасят и внасят данни чрез помощта на модели. Тези модели не са същите като моделите от базата данни, които служат за Entity Framework, но са по техен шаблон и капсулират техните данни. В приложението се работи с общо 3 модела, 2 от които засягат този дипломен проект. Те са именно InstallationTypeClientModel и LicenseProductClientModel. Съдържанието им е същото като на моделите от базата – InstallationType и LicenseProduct, с разликата, че вече могат да бъдат изпращани до клиентската част на приложението, без да се разкрива сензитивна информация, до която потребителя не трябва да има достъп. По този начин се спазва принципа „Енкапсулация“. Основната цел на тези модели е да бъдат връщани от клиента, след което да бъдат преработени от даден сървис клас, после да се запази новата информация в базата данни и накрая да се върне към клиента такъв тип модел. В тях също достъпа на информация е улеснен, което улеснява и работата на разработчика. Пример за това е как в InstallationTypeClientModel класа има колекция с пряк достъп до продуктите на инсталационния тип, което съответно позволява директно да се вземе нужната информация за продукт, без да се правят Linq заявки:

Фигура 10 - LicenseProducts колекция

В конструкторите се подават нужните параметри и се прави проверка дали са валидни:



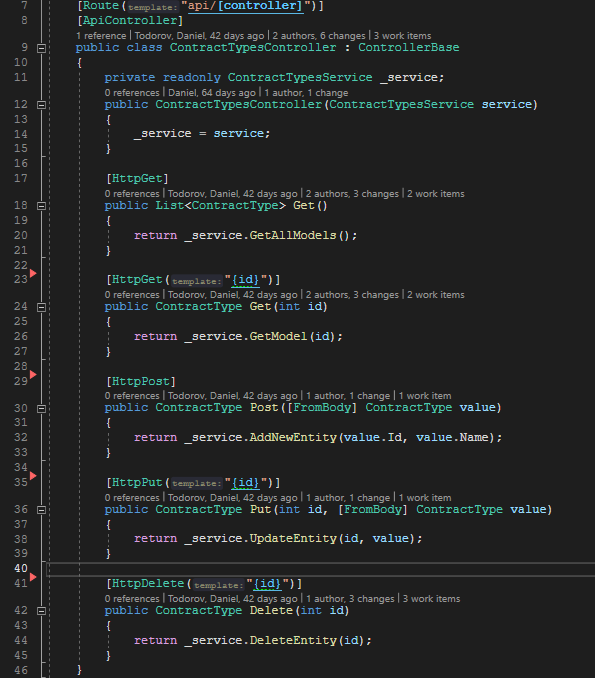
Фигура 11 - Валидация за модел класовете

В случая на приложението тези валидации не са от съществена нужда, понеже самите модели се взимат наготово от клиента с вече валидни данни, но пак е добре да ги има, в случай ако се даде на друг екип и те искат да променят как функционира клиента, и да го направят така че да не се дават със сигурност валидни данни, например.

### Контролери

Компонентите на Back-End-a, които осъществяват самата връзка между главните части на програмата са контролерите, като всички те образуват “API” (Application Programming Interface). Те представляват класове, които се занимават с приемането, обработването и изпращането обратно на информация от клиента. Това се прави с помощта на HTTP заявки. Всеки контролер клас има в себе си:

* Конструктор – в него чрез Dependency Injection се присвоява стойност на полето в класа
* Service поле – това е единственото поле в класовете, съдържа информация за възможните функционалности
* Методи – всеки от които се отнася за конкретна HTTP заявка направена от клиентската част
  + Всеки метод е направен според това каква HTTP заявка е направена, с помощта на анотации

Нека вземем за пример ContractTypesController:

Фигура 12 - ContractTypesController клас

Така изглежда контролерът, който се занимава с типовете договори.

Използвани са анотации за самият клас –

Те оказват линка по който ще се намери контролера по даден шаблон (в случая ще бъде …api/ContractTypes) и описват като какво да се третира и с какви допълнителни свойства разполага.

В него се съдържат общо 1 поле, 1 конструктор и 5 метода. Всеки метод отговаря за съответната HTTP заявка, която е изпратена от потребителя.

* GET Request
* GET Request (с ID)
* POST Request
* PUT Request (с ID)
* DELETE Request (с ID)

Тези методи още се наричат и “API endpoints”. Някои от методите приемат и параметри от тялото на заявката с помощта на атрибут след което информацията се изпраща към помощен клас за съответния контролер, който се нарича Service.

Всички контролери са написани почти еднакво, с малки изключения поради различните им функционалности.

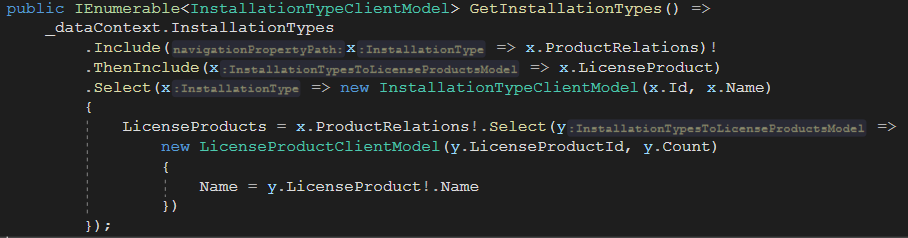
### Сървиси

Сървисите представляват класове с бизнес логика за обработването, взимането и запазването на информация в базата данни. Точно тези класове се подават с Dependency Injection на контролерите. Сървисите са пряко свързани с контролерите, защото без тях, те няма да работят изобщо.

Всеки сървис съдържа:

* Конструктор – в него чрез Dependency Injection се присвоява стойност на полето в класа
* Клас променлива (поле) – това е така наречения Database Context Field и е пряката връзка на Back-End-a с базата данни
* Методи – всеки, от които се извиква в конкретния метод на контролера

Всички сървиси са с почти еднаква структура. Нека вземем за пример InstallationTypesService и да разгледаме различните му функционалности:

* Get Installation Types

Фигура 13 - GetInstallationTypes метод

Цялата работа на този метод е да върне всички инсталационни типове от базата данни и да ги предаде на контролера. Начина по който се постига това е чрез библиотеката System.Linq. Със нея имаме достъп до много функции които биха били нужни за извличането на информацията от базата. Именно в този контекст става дума за методите Include(), ThenInclude() и Select(). Последователността на действията е следната:

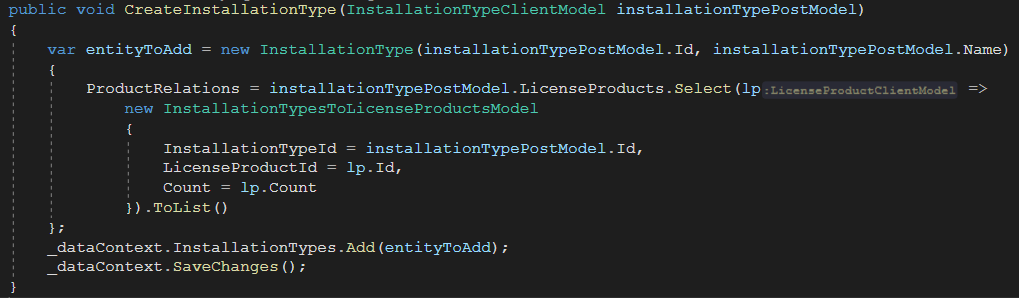
1. Взимат се всички InstallationTypes
2. Със Include се взимат и всички техни релации към продукти
3. Със ThenInclude се взимат самите LicenseProducts обвързани към релациите
4. След това със Select метода всички инсталационни типове се трансформират в друг вид модели, с които Front-End-a ще може да работи
5. Накрая GetInstallationTypes метода връща тези нови модели и ги предава на контролера

* Get Installation Type

Подобно на предишната функционалност, тази също връща InstallationType от базата, но само 1 конкретен, определящ се от даденият параметър ID:

Фигура 14 - GetInstallationType метод

Тук отново чрез помощта на System.Linq имаме достъп до функционалността която ни е необходима, а по-точно метода First(). Преди той да се използва, се извиква предишният метод GetInstallationTypes, които връща всички инсталационни типове и с First() се отсява само елемента със даденото ID който се връща на контролера.

* Create Installation Type

Фигура 15 - CreateInstallationType метод

Това е метода отговорен за създаването на нови инсталационни типове. Той приема като параметър InstallationTypeModel, който представлява класа репрезентиращ InstallationType във Front-End-a. Ето какво точно прави метода:

1. Първо се прави нов обект от типа InstallationType с подадените данни. Тук пак се използва Select() функция която служи за преобразуването на продуктите на дадения модел във продукти, които ще се пазят в базата данни.
2. След това се извиква метод Add() на InstallationTypes полето в \_dataContext, който добавя новосъздадения обект към базата.
3. Накрая се извиква SaveChanges() метод на полето \_dataContext, който запазва промените в базата. Метода не връща нищо.

* Update Installation Type

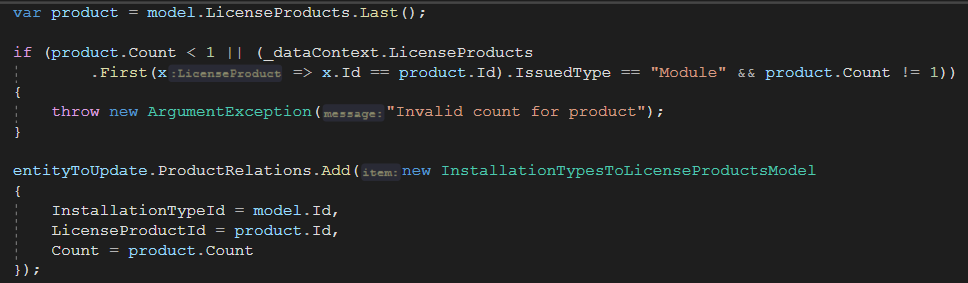
Тази функция се занимава с обновяването на информация за даден инсталационен тип. Начинът й на работа е следния:

1. Намира обекта от базата данни който ще се обновява, заедно с релациите му, и ако не успее се хвърля грешка “Entity not found”

Фигура 16 - Търсене на обект с дадено ID

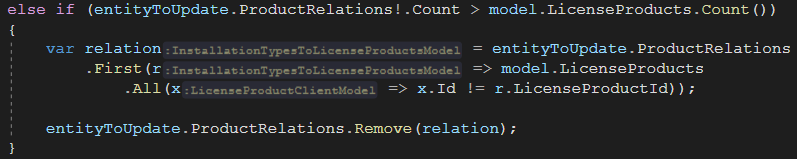
1. След това, според броя на продукти в обекта от базата дали са повече или по-малко от продуктите на модела който се подава се влиза в if конструкция.

Фигура 17 - IF проверка за броя на продуктите от 2та обекта

Ако продуктите на обекта са по-малко, означава че клиента е добавил нов. В този случай се влиза в първото условие на if-a и в него се изпълнява следния код:

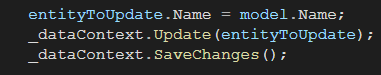
Фигура 18 - Тялото на първото условие

Тук се взима последният на брой елемент от колекцията с продукти на модела подаден от клиента (т.е. новодобавеният продукт), минава през валидация за броя на продукта, която ако не бъде изпълнена хвърля грешка “Invalid count for product”. Накрая добавя към обекта от базата новият продукт с неговата бройка.

Ако обаче продуктите на модела са по-малко от продуктите на обекта се влиза в else частта на if-а.

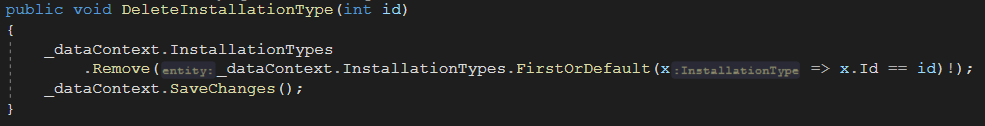
Фигура 19 - Тялото на второто условие

Тук чрез 2 Linq функции се взима обекта от базата, който е бил махнат в модела и се подава на remove метод извикан от текущия инсталационен тип.

1. Накрая всички направени промени се обновяват и запазват в базата данни.

Фигура 20 - запазване на промените

* Delete Installation Type

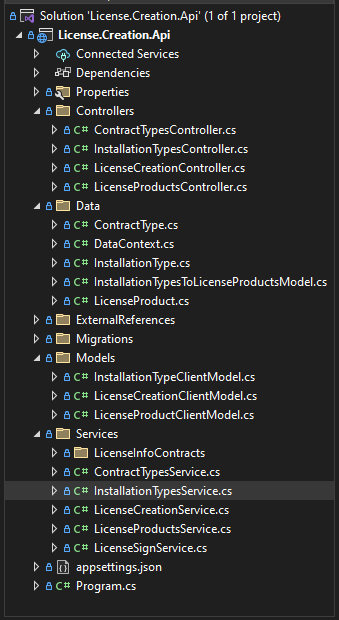
Последната функционалност на сървиса се занимава с премахването на InstallationType. 

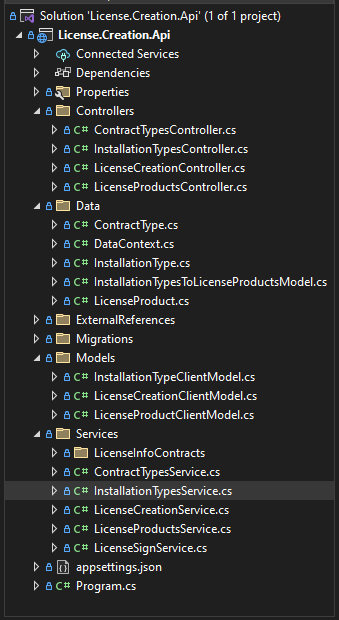
Фигура 21 - DeleteInstallationType метод

Тук, чрез ID подадено от клиента и Linq функция, се изтрива инсталационният тип с това ID.

### Структура

Back-End проекта се казва License.Creation.Api и вътре в него има няколко отделни части. Както вече казахме, проекта има контролери, сeрвиси, модели и база данни с която контактува. Всички файлове съответно са разделени по категории и събрани по папки със следната структура:





Фигура 22 - Структура на Back-End приложението

## Front-End (Client)

### Angular

Angular е известен open-source framework за програмния език Javascript и се използва за разработката на уеб приложения. Той е разработен и поддържан от Google и с него най-често се създават динамични и комплексни single-page приложения (тоест има само една начална страница с достъп до всички компоненти на апликацията). Angular предоставя стабилен набор от функционалности и средства за създаване на апликации, които включват архитектура базирана на компоненти, two-way data binding и мощен набор от вградени директиви. Angular framework също се интегрира много добре със други библиотеки и frameworks, което го прави популярен избор между уеб разработчици. Състои се от следните езици:

#### HTML

HTML е един от основните езици с които Angular работи. Той e Standard Markup Language и се използва за създаването на уеб страници, по-точно подредбата на елементите в една страница (скелета). Той се състои от много на брой различни видове елементи – headings, paragraphs, links, images, lists – с помощта на които се описва структурата на страницата. Кода на HTML може да се напише в обикновен текстов файл, който браузъра ще може да интерпретира за да изобрази самата страница.

#### CSS

Cascading Style Sheets, още познато като CSS, е stylesheet език, който служи за описване на външния вид и форматирането на една уеб страница написана на HTML. Той може да определя оформлението на уеб страниците като цвят на елементите, фон на текста, размер на елементите, разстояние и разположение на един елемент. Понеже CSS е независим от HTML, дава възможността на разработчиците да променят изгледа на уебсайтовете си, без да изменят самата структура на елементите в HTML. CSS е основна част от модерния уеб дизайн и e широко използвана технология за създаване на визуално привлекателни и отзивчиви уеб страници.

#### Typescript

Това е обектно ориентиран език, разработен и поддържан от Microsoft, който е надграждане на Javascript, и това се състои от възможното типизиране на променливите, възможността да се правят класове, namespaces, интерфейси и други неща, които не са възможни в Javascript. Typescript също дава възможността на програмистите да хващат грешки при compile time (по време на компилиране) вместо при runtime (по време на изпълнение), и това се дължи на факта, че първо кода трябва да се компилира до Javascript за да бъде изпълнен. Typescript е съвместим със библиотеки и среди на разработка, първоначално предназначени за Javascript, като например node.js сървъри или в уеб браузърите. В случая, езика служи за описване на логиката на отделните компоненти на Angular.

Използвана литература – 3.

### Компоненти

В Angular, компонентите представляват съставни части от визията на приложението. Състоят се от следните неща:

* Шаблон – това е HTML шаблона, който дефинира структурата и външния вид на компонента. Той се описва в отделен HTML файл (като той отделно може да се обработва със CSS)
* Клас на компонента – това е Typescript класа който имплементира логиката на компонента и предава данни на HTML файла (шаблона). Той дефинира свойствата, функциите и event-ите (събитията) които компонента ще използва
* Метаданните на компонента – това е допълнителна информация за компонента като: елемента в HTML с който ще бъде разпознат компонента (selector), Inputs и Outputs (служат за предаване и взимане на данни от един компонент на друг)

Използвана литература – 3.

Начина по който Angular функционира и свързва компонентите си в едно, е чрез основният клас на име app.component.ts. Той е по подразбиране за всяко приложение направено с този framework и в него по конвенция се използват всички други компоненти които са направени в последствие. Нека първо разгледаме него:

#### App Component

Фигура 23 - App-component клас

Този компонент е основният, към който се закачат всички останали. Той служи за изобразяване на страница на приложението. Във typescript файла, над имплементацията на класа му е споменато, че Angular ще третира този клас като компонент. Това е постигнато с decorator @Component(), който в случая гласи, че този клас ще отговаря на HTML селектор “app-root” и шаблона чрез който ще му се изобразява информацията е с името “app.component.html”. Освен това към този шаблон ще се прилага и стил, който се взима от “app.component.css”.

Тази информация се дава със същия decorator при всеки друг компонент със съответстващите класове и информация за него.

В основата на app компонента се съдържат 3 на брой клас променливи и 2 на брой метода.

Променливата ”title” служи единствено за да се даде име на страницата. Следващите 2 променливи са флагове от тип Boolean, с които се дава информация на класа, дали прозореца на администрацията или частта за създаване на лицензи е отворена. Тези променливи след това се използват при методите които им променят стойностите при отваряне или затваряне на определената част. Методите се извикват от HTML файла съответстващ за компонента:



Фигура 24 - HTML файл на App-component

Тук е направена подредбата на елементите на главната страница. В нея има 2 бутона – от ляво (за създаване на лицензен файл) и от дясно (за отваряне на администрация), отговарящи за съответните части на програмата. При натискане на някой от тях се извиква функцията, която ще промени стойността на променливата отговаряща за определената част. Това е направено с Event Listener специален за Angular –. Най-отгоре в HTML файла са сложени селекторите на 2 компонента, като всеки от тях приема стойността на съответстващата променлива от този компонент в определена за него променлива – Input. Начина по който е описано това действие е с 2 вида скоби, уточняващи, че променливата в която се присвоява стойността може също и да върне стойност –

Това като понятие се нарича “Two-Way Binding”, което всъщност е съкратен запис при присвояване на стойности между Parent и Child компонентите. Другият вариант за запис е по-дълъг и по-неясен, понеже трябва да се уточняват по отделно, какво ще се присвои на Input и какво ще се случи, когато се върне Output. В съкратения вариант е направено по подразбиране, че Parent компонента винаги ще присвоява стойност на променливата в Child и Child компонента винаги ще присвоява стойност на променливата в Parent. За да може да се осъществи съкратеният запис, трябва Output променливата да бъде наименувана по същия начин както и Input, само че със добавено “Change” накрая на името.

Ето как се различават и двата синтаксиса:

* Нормален синтаксис:
* Two-Way binding:

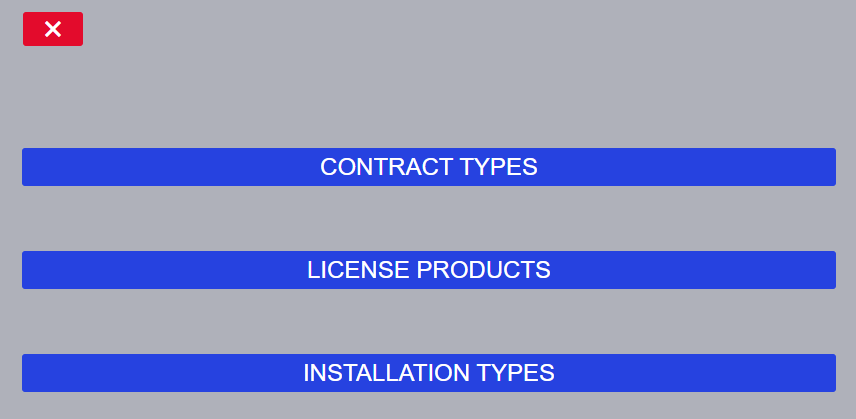
Използвана литература – 4.

#### Right-Navbar Component



Фигура 25 - Right-Navbar клас

Във самият typescript файл се съдържа минимална информация, само за компонента, и това е именно кога е отворен или затворен прозореца му. Както беше споменато по-горе, тук се съдържат Input и Output променливите, с които се взима и изпраща информация към parent класа – app-component. В метода Close() се сменя булевата променлива на “false” и след това се вика emit() метода на Output променливата и изпраща “false” към app-component, който от своя страна ще промени неговата булева променлива, и прозореца ще се затвори.

Ето го и съответстващият UI на компонента:

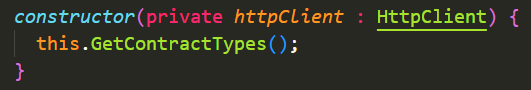
Фигура 26 - Right-Navbar UI

Работата на HTML файла е да изобрази секция със всичките функционалности за Administration частта. Именно за това в себе си има селекторите на 3те компонента, отговарящи за тези функционалности – Contract Type, License Product и Installation Type. Заедно с тях е и бутона за затваряне на прозореца, който от своя страна извиква Close() функцията (фиг. 25).

Left Navbar Component функционира по същия начин, като там съответно се вграждат специалните за него компоненти.

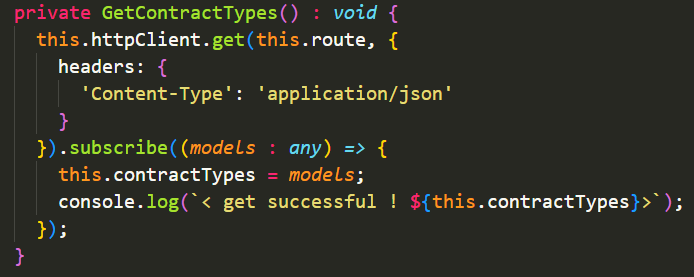
#### Contract Type Component

Това е компонента, занимаващ се с организацията и създаването на нови договорни типове. Той разполага с всички инстанции на договори, които са добавени от клиента или са взети от Back-End-a.



Фигура 27 - Конструктор на Contract Type Component

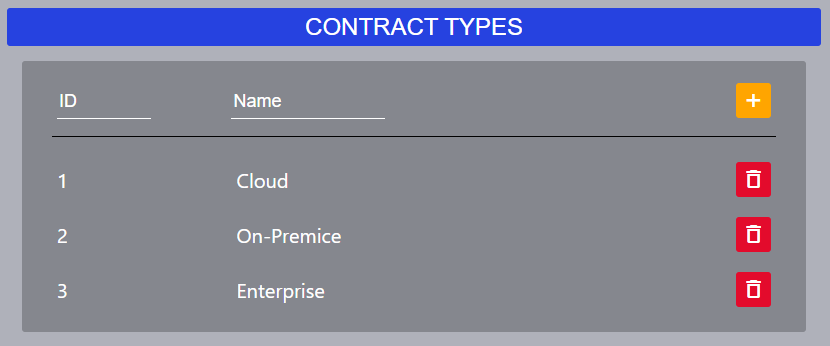
Това е конструктора в Typescript класа, който разбира се, се достъпва при инициализация. Чрез специален правопис, се създава ново поле на класа във облите скоби на конструктора от тип “HttpClient”, което предоставя нужните функционалности за правене на HTTP заявки. В тялото на конструктора се извиква функция GetContractTypes():



Фигура 28 - GetContractTypes функция

Това е пример за една обикновена HTTP заявка в Angular. Тя служи за взимане на всички обекти от базата данни чрез достъпване на Back-End-a. Използва променливата инициализирана от конструктора, с която може да направи Get Request и да вземе нужната информация, като се уточнява какъв request искаме да направим чрез функцията която се извиква след HttpClient променливата (т.е. ако искаме да направим POST, правописа ще бъде – this.httpClient.post(). Това съответно важи и за всички други методи). В Angular, това е алтернативният вариант на функцията fetch() от JavaScript. В този случай, функцията връща Promise, което означава, че тя е асинхронна (тоест програмата може да се изпълнява докато тази функция работи) и не е ясно дали изобщо ще се върнат нужните данни. Поради тази причина трябва да се извика Subscribe() метода, за да се уверим, че наистина са върнати данни, съответно ако вземането на данни е неуспешно, се хвърля грешка.

След като тези обекти се вземат и се запазят в клас променлива, те биват изобразявани на екрана чрез HTML файла. Ето как изглежда този UI



Фигура 29 - Contract Types UI

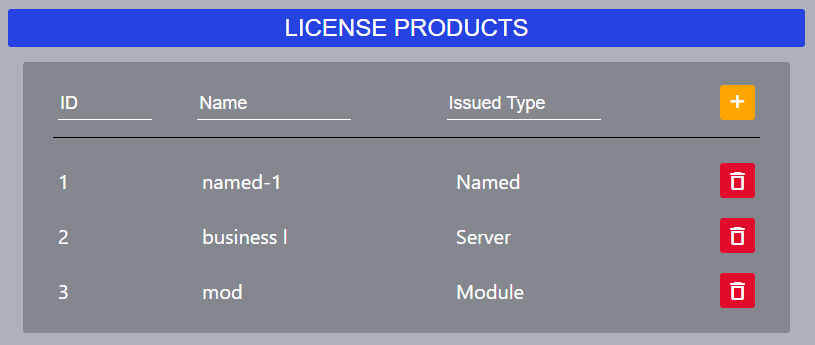
В горната част се намира форма, чрез която се правят нови contract типове. Тя се състои от 2 полета за ID и Name, както и Submit бутон ( ), с който се извиква SubmitContractType() функцията и й се подават данните от полетата. Тази функция от своя страна прави POST Request и изпраща данните към сървъра, където те се обработват и запазват в базата данни.

Отдолу на формата се изобразяват имената и ID-тата на тъкмо взетите елементи. Всеки един от тях има опцията да бъде премахнат със Delete бутона ( ), който извиква DeleteContractType() функцията и й подава ID-то на съответния елемент. След което функцията прави DELETE Request към сървъра, където се изтрива съответния Contract Type от базата данни.

#### License Product Component

Това е компонент който се занимава с лицензните продукти. Както предишният компонент и този също разполага с всички инстанции на лицензни продукти, след като премине през конструктора, както и методите който извършват GET, POST и DELETE Request-и са същите, за това няма нищо ново от към структурата му.

В такъв случай нека погледнем неговият интерфейс:

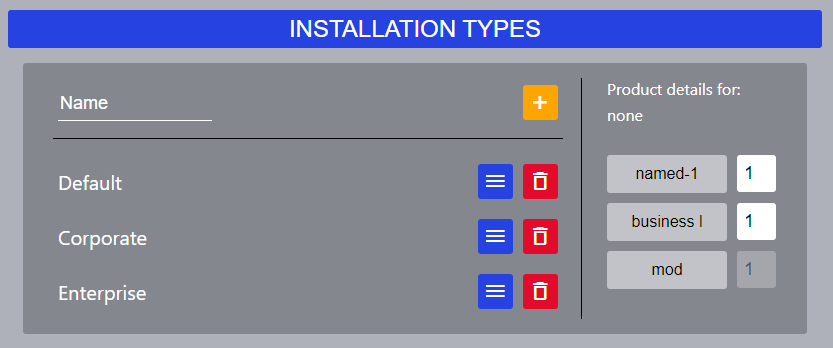
Подобно на интерфейса на Contract Type Component, тук структурата е почти непроменена. Най-отгоре се намира формата за правене на нов продукт и под нея лист със всички налични продукти. Разликата е, че тук има поле за Issued Type променлива, която е специфична за лицензните продукти. Разбира се, Submit и Delete бутоните пак изпращат информация към отнасящите се за тях функции извършващи HTTP Request.

Фигура 30 - License Products UI

#### Installation Type Component

Компонента за инсталационните типове е малко по-различен от останалите, понеже в себе си използва друг. Както и другите два, и този има конструктор инициализиращ променлива за HTTP заявки и лист от обекти, който се взима от метод, който прави GET заявка. Тези обекти се изобразяват на екрана по следния начин:

Фигура 31 - Installation Types UI



Тук отново можем да видим, че от горната страна се намира формата за правене на нови обекти, а под нея – съществуващите обекти на компонента. Пак има Delete и Submit бутони, които са вързани към методите правещи DELETE Request и POST Request, респективно.

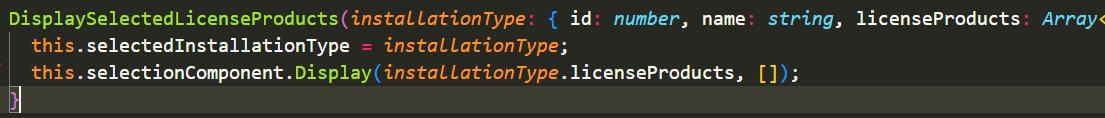
Освен тези неща, срещу всеки обект, вече има и Details бутон ( ), който изобразява продуктите му в компонента който се намира най-вдясно – Selection Component.

#### Selection Component

Работата на компонента е да може да селектира и показва продукти за даден Installation Type. Това включва и бройката им. По този начин, чрез multi-relational таблицата в базата данни могат да се пазят няколко вида лицензни продукти в един инсталационен тип. Това съответно е нужно на потребителя, когато иска да направи нов лицензен файл, да може да избере вече направен тип за инсталация с вградени в него продукти, като всеки продукт дава различни възможности за използването на приложението.

Инициализацията се осъществява по същия начин както и при другите компоненти – чрез конструктора се взима “HttpClient” полето и след това със GET Request функция се взимат всички лицензни продукти. Важно е да споменем, че в класа се съдържа променлива, от която компонента е пряко зависим, с име “SelectedLicenseProducts” –

Когато се натисне Details бутона във Installation Type Component, той изпраща информация към Selection Component чрез променлива, със специфичен за Angular декоратор - @ViewChild. Ето го и метода който се извиква при натискане на бутона:



Фигура 32 - DisplaySelectedLicenseProducts метод

Вътре в него се присвоява стойност на “selectedInstallationType” променливата, както и се извиква Display метод във Selection Component-a. Именно това е полето с декоратор @ViewChild:

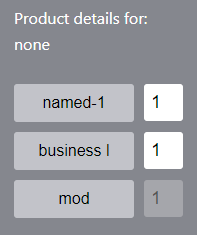
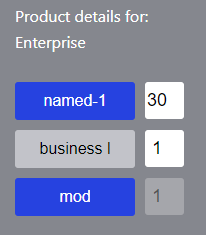


Display функцията се извиква, като й се подават продуктите на току-що запазеният инсталационен тип и се запазват в променливата от която зависи Selection Component – “SelectedLicenseProducts”. Когато това се случи, Angular отчита промяната на стойностите в променливите, и обновява HTML-a. Ето как изглежда той:



Фигура 33 - Selection Component HTML

Всичката информация тук е оградена в <div> елемент. Този елемент използва директива \*ngFor, която е функционалност на Angular, позволяваща на даден елемент да се повтаря няколко пъти на екрана – подобно на for конструкция в C#, където на всяка итерация се принтира нещо на конзолата. В този случай се принтират всички възможни лицензни продукти и те формират самият компонент.

Всеки изобразен обект се състои от 2 части – бутон за селектиране на продукта и поле за даване на бройка на дадения продукт. Бутона се оцветява в синьо, ако продукта е селектиран. Над листа с продукти пише за кой инсталационен пакет се отнасят.

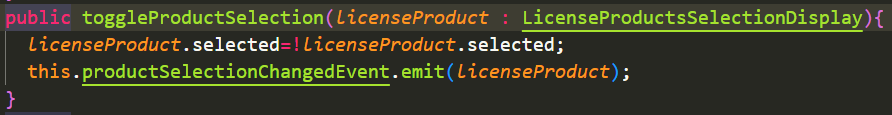
Фигура 34 - Selection Component UI – без селектиран пакет

Фигура 35 - Selection Component UI – със селектиран пакет

Относно бройката на продукта, тя може да бъде само положително цяло число, като стойността й по подразбиране е 1. Единственият начин по който може да му се даде стойност е със стрелки за нагоре и надолу, появяващи се до полето, когато потребителят постави мишката върху него:

Ако обаче полето “IssuedType” на даден продукт има стойност “Module”, неговата бройка винаги е 1 и не принадлежи на промяна. Също така полето за бройка се оцветява в тъмносиво и придобива статус “disabled”:

След като потребителят е дал бройка и е селектирал дадения продукт, програмата се обръща към “toggleProductSellection()” функцията, намираща се в Selection Component. Ето как изглежда тя:



Фигура 36 - ToggleProductSelection метод

В нея се предава току-що избраният продукт към Parent компонента (Installation Type Component) чрез Output променлива. Ето как е имплементиран Selection Component във Installation Type Component, за да може да му бъде връщана информация:

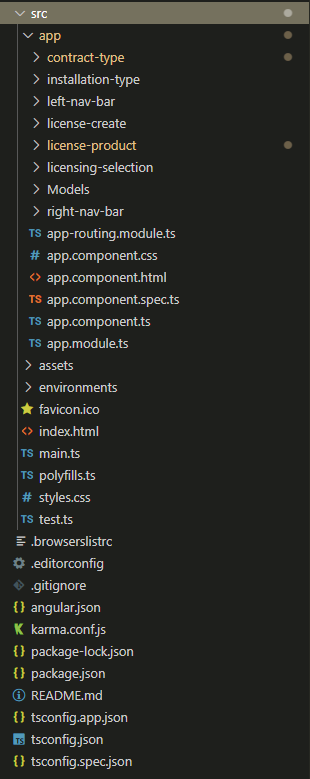


Фигура 37 - Selection Component в контекста на Installation Type Component

В случая не се използва Two-Way Binding и се взимат данни само от Child компонента. Тези данни се дават като аргумент на функция с име “UpdateProductSelection()” и вътре в нея се изпълняват различни команди, в зависимост от това, дали потребителят е махнал или е добавил продукт в пакета. В края на функцията се прави PUT Request, който обновява базата данни със променената информация.

### Структура

Така изглежда файловата структура на Front-End-a. Както вече споменахме, главната задача на Angular Framework е да направи възможно разделянето на отделните части на програмата в независими компоненти, което прави разработката много по-лесна и изчистена.



Фигура 38 - Структура на Front-End проекта

# Заключение

Успешно беше създадена административната част на авторския проект DocuWare License Creator.

Първо беше създадена база данни, в която чрез миграции се създават самите таблици.

Към нея е пряко свързана Back-End частта на проекта, която използва Entity Framework за миграциите и Code First начина на кодиране. Back-End-a разполага с пряк достъп до обектите от базата с помощта на модели. Бяха създадени също и контролери, които правят самата връзка с базата, и заедно с тях и сървисите които извършват различните функционалности.

След това беше създаден Front-End-a, който служи като интерфейс на всички тези данни. Той беше направен с Angular Framework, който включва езиците HTML, CSS и TypeScript. От тук се извършват HTTP заявки с които се достъпва сървърната част. Благодарение на Angular, проекта беше възможно да бъде разделен на отделни компоненти и по този начин беше постигнат чист код.

Основното нещо което би могло да се подобри за в бъдеще, е валидирането на данни, понеже в момента са сложени само най-базовите и най-нужните валидации. Другото нещо което би могло да се усъвършенства, е стила на уеб частта, въпреки че приложението е за вътрешна употреба на DocuWare.

# Практическа част

Тема: Система за организация на продукти и функционалност в пакети и планове.

Направен е DocuWare License Creator, чиято главна цел е разделянето на зависимостите между екипите в DocuWare. С този дипломен проект се реализира административната част на приложението, която включва менажирането на договори, лицензни продукти и пакети с тези продукти.

За изграждането на административната част са изпълнени следните стъпки:

* Създадена е база данни на MySQL Server.
* Реализирано е приложение на програмния език C#, служещо като Back-End, което прави връзка с базата данни. Това е постигнато с помощта на конкретни средства, интегрирани в средата на разработка, а именно – ASP.NET Core Web API и Entity Framework.
* Направена е миграция от сървърната част (Back-End) към базата данни. Това става с помощта на Entity Framework.
* Изграден е Front-End / клиентско приложение, което да служи за интерфейс на проекта. То е направено с помощта на Angular, който включва езиците HTML, CSS и Typescript. Чрез тази функционалност, клиентското приложение е разделено на компоненти, които са независими един от друг.
* Накрая са събрани всички тези модули и са внедрени в един сървис, съответно служещ за създаване и конфигуриране на лицензи и техните компоненти.

В разгледаната темата „Система за организация на продукти и функционалност в пакети и планове“ се описва проблем, поставя се цел и се правят петте важни стъпки за постигането й, в резултат на които е изготвена система за организация на различни продукти, договори и инсталации за един лиценз.

Бъдеща препоръка за подобрение на приложението е да се добавят повече валидации при вкарването на данни от потребителя.

# Използвана литература

(1) .NET Core и C# - <https://learn.microsoft.com/en-us/dotnet/core/introduction>, <https://www.tutorialsteacher.com/core/dotnet-core>

(2) Entity Framework - <https://www.entityframeworktutorial.net/what-is-entityframework.aspx>, <https://learn.microsoft.com/en-us/ef/core/>, <https://www.cleverti.com/software-development/net/entityframework-yes-or-no/>

(3) ASP.NET Core Web API, Angular Framework, HTML, CSS, TypeScript, Components in Angular - <https://chat.openai.com/>

(4) 2-Way Binding - <https://angular.io/guide/two-way-binding>