**ТЕХНИЧЕСКИ УНИВЕРСИТЕТ – СОФИЯ**

Факултет по компютърни системи и технологии

Катедра: Компютърни системи

**ДИПЛОМНА РАБОТА**

*Уеб платформа за управление на паркоместа*

**Дипломант:**

Любка Каменова Кашкавелина  
Ф.Н: 121320001

**Научен ръководител:**

доц. д-р инж. Надежда Кучмова

София

2021 г.

Съдържание

[Въведение 4](#_Toc81759145)

[1. Анализ, цели и задачи 6](#_Toc81759146)

[2. Проектиране на платформата и базата данни 9](#_Toc81759147)

[2.1 Проектиране на бизнес логиката на приложението 9](#_Toc81759148)

[2.2 Архитектура на платформата 12](#_Toc81759149)

[2.2.1 MVC архитектура 13](#_Toc81759150)

[2.2.2 Трислойна архитектура 15](#_Toc81759151)

[2.3 Архитектура на комуникацията между клиентската и сървърната част 18](#_Toc81759152)

[2.3.1 Рутиращ механизъм(Routing) в администраторското приложение 19](#_Toc81759153)

[2.3.2 Крайни точки(endpoints) в потребителското приложение 21](#_Toc81759154)

[2.4 Архитектура на базата данни 22](#_Toc81759155)

[3. Въведение в използваните технологии 27](#_Toc81759156)

[3.1 Microsoft Visual Studio 2019 27](#_Toc81759157)

[3.2 ASP.NET Core 27](#_Toc81759158)

[3.3 .NET Framework 28](#_Toc81759159)

[3.4 Microsoft SQL Server 28](#_Toc81759160)

[3.5 HTTP протокол 28](#_Toc81759161)

[3.6 Bootstrap 30](#_Toc81759162)

[3.7 Angular 30](#_Toc81759163)

[4. Програмна реализация - ParkingManagementSystem Solution 31](#_Toc81759164)

[4.1 ParkingAdministration project 31](#_Toc81759165)

[4.1.1 Папка Models 32](#_Toc81759166)

[4.1.2 Папка Contollers 33](#_Toc81759167)

[4.1.2.1 UsersController 34](#_Toc81759168)

[4.1.2.2 BaseController 38](#_Toc81759169)

[4.1.2.3 LoginController 39](#_Toc81759170)

[4.1.2.4 HomeController 40](#_Toc81759171)

[4.1.3 Папка Views 40](#_Toc81759172)

[4.2 WebHost project 42](#_Toc81759173)

[4.2.1 Contracts 42](#_Toc81759174)

[4.2.2 DomainModels 43](#_Toc81759175)

[4.2.3 Ioc и Mapping 43](#_Toc81759176)

[4.2.4 Security 45](#_Toc81759177)

[4.2.5 Services 47](#_Toc81759178)

[4.3 WebClientCore project 50](#_Toc81759179)

[4.4 WebHost.Data и WebHost.Repository 53](#_Toc81759180)

[5. Ръководство на потребителя 55](#_Toc81759181)

[6. Заключение – бъдещи тенденции 55](#_Toc81759182)

[Използвана литература 56](#_Toc81759183)

[Приложение 58](#_Toc81759184)

# Въведение

Колко често Ви се случва да имате проблеми с намиране на място, където да паркирате, било то на работа или дори пред вашия дом? Масовият отговор би бил – често. В ежедневието ни многократно се сблъскваме с това предизвикателство и намирането на оптимално решение невинаги е лесно и е повлияно от множество променящи се фактори. Съществуват различни подходи – от комплексни „умни“ системи за управление на паркинги, използващи устройства за проследяване на потока и заетостта на даден паркинг до използване на споделени вътре в дадена организация документи, най-често екселски таблици, маркиращи кое паркомясто е свободно и кое не, както и кой е паркирал в момента на него. В съвременния изцяло управляван от технологии свят, първите естествено набират все по-голяма популярност, но съответно изискват и немалки инвестиции както за закупуване на самия софтуер и устройствата, принадлежащи към него за изграждане на една IoT система, така и за обучение на потребителите, които предстои да го използват. Сякаш липсва опростена, лесна за използване система, която да отговаря за базовите нужди на един функциониращ паркинг и която да може да бъде приложена и в по-малки бизнес или жилищни комплекси. Вторият споменат подход, използващ просто обновяване на споделени документи, елиминира фактора сложност, но естествено тук се появява рискът от допускането на грешки при твърде много ръчна работа и се губи гъвкавостта на функции като следене в реално време и т.н.

Тези неудобства са предпоставка за създаване на опростена, но ефективна система за управление на паркоместа, която да се опита да предостави най-необходимите функционалности както за собствениците на паркинга, така и за използващите го посетители. Темата дава широк спектър от възможности за реализацията на конкретно решение. Интересна е и има възможност да се използват съвременни технологии за създаване на приложението. Идеята на платформата е да бъде достъпна и лесна за използване от потребители от всякакви категории – от такива, които ще се възползват еднократно от услугите на системата и ще резервират паркомясто за кратък период от време, до постоянни собственици на паркоместа, които ще предпочетат да отдадат мястото си, когато не го ползват.

Надявам се темата да бъде различна и същевременно полезна. Да представи по един достъпен и интересен начин процеса по отдаване/резервиране на пракоместа, които ще бъдат налични в системата. Най-същественото, към което е насочен стремежа на уеб платформата, е да облекчи хората в процеса на намирането на удобно паркомясто на работа или пред вкъщи, тъй като той може да бъде изключително стресов и често безрезултатен.

# Анализ, цели и задачи

Преди да преминем към проектирането на настоящото приложение е необходимо да представим какъв е актуалният проблем, който стои в основата на избора на конкретната тема, както целите и задачите, които сме си поставили.

Най-лесно бихме вникнали в същността на проблема, ако разгледаме няколко примера от ежедневието ни. Голяма част от фирмите в различни сектори разполагат със собствени паркинги, като съответно отдават целите или части от тях на своите служители. Така всеки служител разполага със собствено паркомясто. Често обаче процедурата по отдаване на това място не е моментална и може да отнеме известно време – от няколко дни дори до седмици. Това би причинило неудобство на новоназанчените служители, които ще трябва да потърсят къде да спрат през първите си работни дни. Не би ли било хубаво, ако в първите си работни дни, човек не се налага да мисли и за този ангажимент?

Друга, дори още по-често срещана ситуация, е когато даден служител излиза в отпуск или отсъства по друга причина от работа и мястото, което той използва, остава празно. Би било практично, ако той има възможност да обяви своето място като свободно за периода, в който отсъства, и съответно то да бъде използвано от колегите му, които по една или друга причина са избрали да не наемат свое собствено паркомясто за постоянно. От една страна биха били спестени разходи на самия служител, който не се нуждае от постоянно паркомясто и използва по-рядко личен автомобил за превоз до работното място. От друга – фирмата би спестила от нуждата от закупуване на огромни площи за паркинги и съответно от разходите за поддръжката им. Макар и пряко свързана с управление на паркоместа, в известна степен настоящо разработваната система се старае да насърчи ограничаването на използването на големи площи за паркинги, като по-скоро акцентира върху разумното им и практично използване. В модерния свят, препълнен от автомобили, е изключително важно как те се организират и мястото, което заемат, и ако то може да бъде ограничено, с помощта на една система за управление на местата за паркиране, и използвано за разширяване на градските пешеходни зони, то това би бил голям успех за обществото.

Колко често се случва дадена фирма да получава доставки и невинаги куриерските автомобили да разполагат с обособено място или зона, където спокойно да могат да извършват дейността си, без да утежняват и без това натоварения трафик. Процесът би се случвал доста по-леко, ако доставчикът има свой профил в система, в която бързо може да види налични свободни места и да избере някое, което да използва за конкретния ден.

Да излезем за малко от корпоративния контекст. Подобна платформа, каквато беше спомената по-горе, би била изключително полезна и за жилищни комплекси с принадлежащи паркинги. Колко често се случва да каните гости и те да няма къде да паркират своите автомобили? С помощта на система, в която живущите маркират паркоместата си като свободни, когато не ги използват, и съответно на тях спират хора, които имат временна нужда от това, значително би се намалил огромният недостиг на паркоместа пред жилищни сгради и биха били избегнати множество конфликти между живущите по отношение на безразборното паркиране.

Настоящата дипломна работа, бидейки пряко свързана с управление на паркоместа, се старае облекчи натоварения трафик, който е един от наболелите проблеми на модерното общество. Естествено той е многоаспектен и фокусът тук е върху паркирането и как към него може да се подходи по-различен и по-гъвкав от досегашния начин. Ключовото тук е чрез разумно използване на паркоместа да бъдат разрешени поне частично проблемите с неправилното паркиране, породен от недостига на достатъчно свободни места, и губенето на абсурдно много време в търсене на такова. Споделеното пътуване вече е факт, защо споделеното паркиране да не е това нещо с неподозиран потенциал, което би променило пътната инфраструктура към по-добро. Уеб платформата, която ще бъде представена и изградена в тази дипломна работа, има за цел да адресира проблема с намирането на паркомясто, респективно обявяването на текущо неизползвано за свободно за резервация, по лесен, удобен, интерактивен и най-вече общодостъпен начин. Улесненият достъп до гореспоменатата платформа, предоставя възможност за потребители от всякакъв тип бързо и ефективно да прегледат наличните свободни места и по този начин да бъдат спестени многобройните усилия по намиране на място за паркиране, облекчавайки както хората, използващи даден паркинг, така и самите му собственици/наемодатели.

Платформата е полезна, защото се старае да представи една модерна концепция за това как се прави система с добро покритие на потребителските изисквания и съвременни архитектури.

От поставените цели, може да обобщим, че произтичат следните задачи:

* Да се избере подходящ подход за връзка с база данни;
* Да се проектира собствена база от данни;
* Да се изгради MVC архитектура за администраторското приложение, съответстваща на нуждите на уеб платформата;
* Да се изгради трислойна архитектура, използваща WCF сървиси за потребителската платформа;
* Да се осигури надеждна комуникация между отделните компоненти;
* Да се изгради визуална част(клиентска), която да предоставя различните услуги;
* Да има възможност за осъществяване на базовите операции – добавяне, преглеждане, редактиране и изтриване на връзка паркомясто-собстевник за администраторския панел;
* Да може дадено паркомясто да се отдава за резервация;
* Да може вече отдадено паркомясто да се резервира;
* Да може да се отдава/резервира паркомясто при определени изпълнени валидационни условия;
* Да може да се отмени вече направена резервация;
* Да може да се преглеждат наличните за резервация паркоместа;
* Да се изградят различни роли, чрез които да се осъществяват различни нива на достъп до системата;
* Да има възможност да се създават акаунти и съответно да се редактират паркоместата към тези акаунти;
* Да се дефинира администратор на уеб платформата, който може да управлява всички паркоместа;
* Администраторът да има възможност за търсене по номер на паркомястото, име на потребителя и номер на използващия паркомястото;
* Да има възможност за директно свързване на потребителя с администратора на системата чрез отваряне на имейл клиент;
* Да може приложението да работи на различни устройства – компютър, таблет, мобилен телефон.

# Проектиране на платформата и базата данни

## Проектиране на бизнес логиката на приложението

Основна задача при проектирането на една платформа е определянето на това с какви функционалности е желателно да разполага тя. Предвид вече дефинираните цели, за успешната реализация на приложението е необходимо да се осигури пълноценно общуване между него и потребителя. За да се осъществи това, трябва да бъдат налични операции за управление на потребителите и паркоместата, около които се изгражда цялата система. След тяхното дефиниране е редно да се определят нива на достъп или т.нар роли. Не би било редно случаен потребител да може да редактира информация за друг потребител например и като цяло да управлява съдържанието на платформата. Именно затова ще бъдат обособени три роли – потребител без паркомясто, потребител с паркомясто и администратор на платформата. Тук е важно да отбележим, че за работа със системата е необходима регистрация и последващо логване, като администраторът е предварително дефиниран.

Потребителят без обособено собствено паркомясто може да разглежда вече обявените за свободни за даден интервал от време, както и разбира се да наеме паркомясто, ако датите, за които е отдадено то, го устройват. За да направи това е необходимо потребителят да се регистрира в системата с потребителско име и парола. Затова ще бъде създадена и логин система. Тъй като уеб платформата е съсредоточена изцяло върху резервирането/отдаването на паркоместа, с цел улеснение, при регистрация от потребителя не се изисква въвеждането на някаква подробна лична информация - само и единствено потребителско име и парола. След успешна регистрация, ако е покрил критериите за име и парола, за които ще споменем по-късно, потребителят може да влезе в системата. За да резервира паркомясто, е необходимо да бъде попълнена информация за автомобила, който би следвало да паркира на резервираното място. Минималната информация, която трябва да бъде предоставена и без която не може да бъде извършена резервация, е номер на автомобила и телефонен номер на потребителя. Освен задължителната информация потребителят може да попълни и марка, модел и цвят на колата. След като е въвел необходимите данни, в раздел “Book”, ползвателят на системата може да види списък със свободни места, както и периода, за който са налични. При натискане на бутон за съответното място, се отваря календар, от който може да бъде избран както целия период, за който е налично мястото, така и по-малък период от целия обявен. При избор на по-малък период, оставащите дати преди и след избрания период, се появяват като отделни възможности за резервация с цел по-голяма гъвкавост и използване максимално на освободеното място. Естествено потребителят има правото и да отмени резервираното паркомясто, било то поради вече липса на необходимост от него или по погрешка, като слотът, за който е било резервирано мястото ще се появи като свободен при списъка с налични.

При ролята с необособено паркомясто, има възможност за подаване на заявка към администратора за получаване на такова или за сигнализиране при потенциално възникнал проблем. На мястото, където по принцип стои номерът на паркомястото, е поставен линк, при натискане на който се отваря имейл клиент, като потребителят е помолен да влезе с желания от него имейл, като имейлът на администратора на платформата е попълнен автоматично, както и самата тема на имейла. В дъното на страницата е поставен и още един допълнителен линк, който извършва същата операция, но с друг зададен имейл, като той служи за подаване на обратна връзка към цялата система.

Потребителят със собствено паркомясто може да извършва всички дотук изброени действия, като надгражда с функционалността за отдаване на паркомясто. За да може да се превърне от потребител без в такъв с отдадено място, администраторът трябва да отдаде налично паркомясто на този потребител. За да стане това, отново е необходимо да бъде попълнена минималната информация за номер на автомобила и телефон на собственика му. Едва след това разделът със собствено паркомясто – “My parking space” може да бъде видян и съответно чрез натискане на бутона за отдаване и избиране на период, за който това да се случи, паркомястото ще може да бъде резервирано от други потребители на системата.

Администраторът на уеб приложението е предварително дефиниран и неговата роля е изключително важна за функционирането на системата. Той разполага с отделно приложение, което е предназначено само за хората, които ще отговарят за управлението на системата. Обикновените потребители нямат достъп до това приложение. Избран е този подход с цел по-голяма сигурност на системата и намаляване на вероятността някой да се опита да проникне в администраторския панел и да извърши зловредни действия. Най-същественото негово действие е отдаване на дадено паркомясто на неговия собственик, като това може да стане чрез изпращане на заявка по имейл(основният предвиден подход), както споменахме по-горе, чрез телефонно обаждане или дори устно запитване, според начина на комуникация в съответната фирма или жилищен комплекс. При натискане на бутона за отдаване на паркомясто се отваря форма с падащи менюта за наличните паркоместа, потребителите в системата и съответно два календара за начална и крайна дата на асоцииране на зададеното паркомясто с конкретния потребител, като ако не е зададена крайна дата, се разбира, че мястото е отдадено за неограничен период. Това може да бъде променено в последствие. Администраторът разполага и със списък в раздела “Parking spaces” с вече отдадените паркоместа, който му помага да проследява заетостта на паркинга. Естествено при голямо количество данни, би било необходимо и търсенe из този списък. Затова е реализирана функционалност за търсене, който работи по ключовите и задължителни данни като потребителско име, номер на автомобила и номер на ползвателя на паркомястото. Гореспоменатият списък разполага с възможности за преглед и редакция на детайлите за даден потребител, както и за премахване на връзката паркомясто – потребител, което също е една от най-ключовите функционалности за администраторската роля. Тези привилегии са продиктувани от необходимостта за контрол върху цялостната система – например при смяна на принадлежността на дадено място към определен собственик и, добавяне на собственик към ново паркомясто и др. Тук е важно да отбележим, че въпреки важността на администраторската роля, системата е проектирана така, че действията на администратора да се извършват относително рядко или дори еднократно. Какво се има предвид – задачите на администратора се свеждат до разпределяне на местата по собственици при първоначалното внедряване на системата и в хода на по-нататъшното и развитие – до премахване на връзката на дадено място с вече бившия му собственик и евентуалното му отдаване на нов. Главната идея е, веднъж раздадени, собствениците на паркоместа да ги обявяват като свободни за резервация и евентуално други потребители, желаещи да използват паркинга, да могат да ги резервират, като това да се случва само с няколко клика от страна на потребителите, без намеса на трета страна, която да има нужда да одобрява и наблюдава целия процес.

На следващата фигура 1 е представена диаграма с функционалностите на системата и тяхната зависимост по роли:



Фиг 1. Функционалности в системата и тяхната зависимост по роли

## Архитектура на платформата

Настоящата дипломна работа представя система, съставена от две приложения с различни, но до голяма степен сходни архитектури. За потребителското приложение е избрана трислойна архитектура(three-tier architecture), а за администраторското – MVC архитектурен шаблон. Зад това решение стои желанието да се покаже как две отделни приложения, използващи различни технологии и архитектури, могат да работят съвместно и с обща база данни, което е често срещана практика в ежедневието, а и необходимост, когато например се налага надграждане на вече съществуващ проект, изграден с една технология, с друга, даваща повече възможности. И за двете споменати архитектури е характерно разделянето на бизнес логиката от графичния интерфейс и данните в дадено приложение. Това се прави с цел разграничаване на вътрешното представяне на информацията от начина, по който тя се представя и приема от крайния потребител.

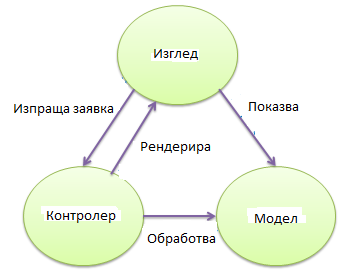
## MVC архитектура

Администраторското приложение е изградено по популярния архитектурен шаблон за дизайн – Модел-Изглед-Контролер(Model-View-Controller или MVC). Традиционно използван за десктоп потребителски графични интерфейси (GUIs), този шаблон с годините добива популярност и се доказва като успешен такъв и при проектирането на уеб приложения, каквото е и разработеното в тази дипломна работа. [1]

MVC шаблонът се състои от три основни компонента, всеки със своя собствена структура. За целите на нашия проект, този тип обособяване на компонентите се оказва изключително подходяща. Именно заради това ще се спрем върху отделните компоненти, погледнати през призмата на изграждането на уеб приложение.

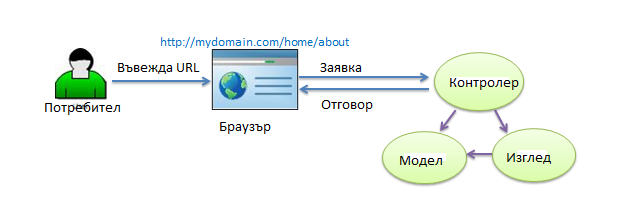
* Моделът: Този компонент представлява основните данни на приложението(domain data). Тук е важно да отбележим, че той не е базата данни. Той може да комуникира с нея, но само представлява данните. Може да бъде описан също и като динамична даннова структура, независима от потребителския интерфейс. Моделът директно управлява данните, логиката и правилата, свързани с тях [2]. В нашия случай, за по-голямо удобство и прегледност, данните представящи обектите от базата са обособени в отделна папка Data, а класът за модел(UserToParkingSpaceEntity) е съставен от свойства от тези Data типове.
* Изгледът: Този компонент отговаря за това какво всъщност бива представено на потребителя. Обикновено той съдържа HTML и CSS файлове. Изгледите в ASP.NET Core използват т.нар. Razor view engine, за да „вградят“ C# кода в HTML-ския таг-базиран език.[3] Тук се включва и цялостната информация за общия изглед(layout) на уеб приложението и как то се представя пред крайния потребител. От организационна гледна точка, изгледите в нашето приложение са структурирани така, че всеки съществен компонент разполага със своя собствена папка, в която са поместени неговите изгледи. Обособена е и т.нар споделена папка, в която се помещават частични изгледи(partial views), които се използват в повече от един от основните изгледи. На тази структура на изгледите ще се спрем по-подробно в раздела „Програмна реализация“.
* Контролерът: Стигнахме и до може би най-съществения компонент от MVC шаблона. Както и името подсказва, контролерът е отговорен за комуникацията между различните компоненти. Той получава заявка(request) чрез рутиращия модул, на който ще се спрем по-късно, комуникира с модела и изпраща подходящ изглед към потребителя. Контролерът е отговорен за обработката както за всяко входящо за системата действие от страна на потребителя, така и за изходящите действия от страна на системата в отговор на тези, които я използват. Контролерът е началната входна точка(entry point) и избира с кои типове модели да работи и кой точно изглед да бъде рендериран. Както и при изгледите, за всеки основен компонент на системата, има предвиден и контролер. [4][5]

На фиг. 2 е представена MVC архитектурата с описаните дотук компоненти.



Фиг 2. MVC архитектура

Нека към тази схема добавим и потребителя, който ще комуникира със системата, за до можем по-ясно да се представим как точно ще се осъществи самият процес – фиг. 3.



Фиг. 3 МVC архитектура с добавена комуникация с потребителя

Стъпките през, които преминава комуникацията, са следните:

1. Потребителят изпраща заявка към уеб страницата, като въвежда нейният URL адрес в лентата на браузъра на своя компютър
2. Базирайки се на подадения URL, се избира подходящ контролер и заявката се предава към него.
3. После контролерът си взаимодейства с модела, за да претърси/обнови данните
4. Контролерът предава данните на изгледа, който от своя страна отговаря за визуализиране на данните
5. Изгледът е връща, към потребителя, който е изпратил заявката.

Разпределените по този начин отговорности между компонентите, допринасят за по-голямата гъвкавост на нашето администраторско приложение. Отделните елементи могат да бъдат управлявани независимо и отделно един от друг, което пък от своя страна улеснява работата на разработчика и би дало възможност за паралелна работа върху различните компоненти от различни хора.

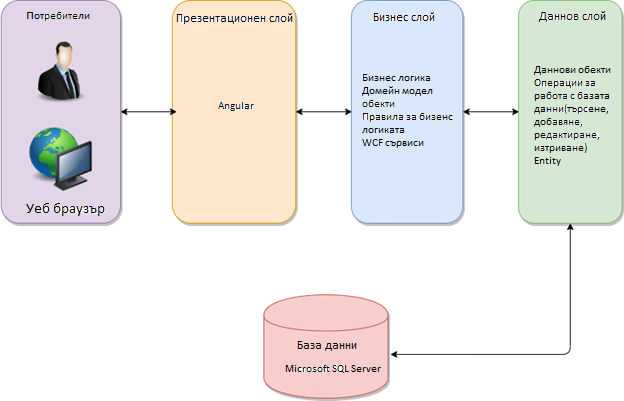
## Трислойна архитектура

Потребителското приложение е изградено, спазвайки модела за трислойна архитектура(three-tier architecture). Той е основен при изграждането на логическия дизайн както на десктоп, така и на уеб и мобилни приложения, и сегментира компонентите на приложението в три нива на услуги. Тези нива не съответстват непременно на физически местоположения на различни компютри в мрежа, а по -скоро на логически слоеве на приложението. Начинът, по който частите от приложение са разпределени във физическа топология, може да се промени в зависимост от системните изисквания. [666, <https://docs.microsoft.com/en-us/windows/win32/cossdk/using-a-three-tier-architecture-model>]

Разделянето по слоеве е изключително подходящо за пълноценната реализация на планираните функционалности за потребителското приложение от нашата платформа. Така те могат да бъдат проектирани и изградени независимо един от друг, което дава свобода при избиране на технологиите за сървърната и клиентската част. Ще разгледаме функциите на отделните слоеве, като ги съпоставим със спецификите на използваните от нас технологии и подходи при реализирането на уеб приложение.

* Презентационният слой или слоят на потребителските услуги дава на потребителя достъп до приложението. Този слой представя данни на потребителя и по избор позволява манипулиране и въвеждане на данни. Двата основни типа потребителски интерфейс за този слой са традиционното десктоп приложение и, в нашия случай, уеб базираното приложение. Уеб базираните приложения сега често съдържат повечето от функциите за манипулиране на данни, които са необходими за пълноценната комуникация с ползвателя на системата. Това се постига чрез използване на динамичен HTML, променящ се в реално време спрямо управляващите го данни. За този слой е избрано използването на Angular като гъвкава и модерна платформа и framework за изграждане на клиентски приложения с помощта на HTML и TypeScript.
* Средният слой или слоят на бизнес услугите се състои от правила за логиката на приложението и данните. Наричан още ниво на бизнес логика, средното ниво е мястото, където разработчиците могат да решават критични задачи за бизнес логиката на приложението и да постигат висока производителност. Компонентите, които съставляват този слой, могат да съществуват на сървърна машина, за да подпомогнат споделянето на ресурси. Тези компоненти могат да се използват за прилагане на бизнес правила, като бизнес алгоритми и законови или правителствени разпоредби, и правила за данни, които са предназначени да поддържат структурите на данните последователни и консистентни в конкретни или множество бази данни. Тъй като тези компоненти от средния логически слой не са обвързани с конкретен клиент, те могат да се използват от всички приложения и могат да бъдат преместени на различни места, според необходимото време за реакция(да речем мобилни приложения) и други правила. Например, прости редакции могат да бъдат поставени от страна на клиента, за да се сведат до минимум кръговите връзки в мрежата, или правилата за данни могат да бъдат поставени в съхранени процедури. За изграждане на този слой е избран Windows Communication Foundation (WCF) фреймуъркът за изграждане на сървис-базирани приложения. С помощта на WCF могат да бъдат изпращани данни като асинхронни съобщения от една крайна точка(endpoint) на сървиса до друга. Крайната точка на сървиса може да бъде част от непрекъснато достъпен сървър, хостван под IIS, както е в нашия случай с ParkingManagementSystemServer, или може да бъде сървис, хостван в приложение. Крайна точка може да се намира в клиент, използващ сървиса, като той изисква данни от крайната точка от другата страна на сървиса, намираща се на сървъра. Съобщенията могат да бъдат прости като един символ или дума, изпратени като XML, или сложни напр. поток от двоични данни. [666, <https://docs.microsoft.com/en-us/dotnet/framework/wcf/whats-wcf>]
* Данновият слой или слоят услуги за данни взаимодейства с постоянни данни, обикновено съхранявани в база данни или в постоянно хранилище. Това е действителният слой за достъп до СУБД(система за управление на база от данни). Той може да бъде достъпен чрез слоя бизнес услуги и понякога от слоя потребителски услуги. Този слой се състои от компоненти за достъп до данни – хранилища(repositories) и описващи ги интерфейси (вместо от необработени СУБД връзки), за да се подпомогне споделянето на ресурси и да се позволи конфигурирането на клиентите, без необходимост от инсталиране на библиотеките на СУБД и ODBC(Open Database Connectivity) драйверите за достъп до базата на всеки клиент. За това ниво е избран Entitiy Framework като фреймуърк предоставящ добри инструменти за изграждане, преглед и обработка на обектите, идващи от базата данни. Самият даннов слой е изграден от два проекта – един за самите данни(WebHost.Data) и един за хранилището с методи за тяхната пряка обработка(WebHost.Repository).

На фиг. 4 е представена трислойна архитектурата с посочени фреймуръковете по слоеве:



Фиг. 4 Трислойна архитектура

## Архитектура на комуникацията между клиентската и сървърната част

За изграждането на едно ефективно уеб приложение е необходимо да сме наясно с клиентската и сървърната му част и какви действия могат да се осъществяват на всяка една от тях. В контекста на уеб приложенията, клиентът всъщност е браузърът, сървърът може да бъде уеб сървър или сървър на приложението(application server). В нашия случай използваме уеб сървър поради необходимостта от обслужване на статични страници напр. HTML и CSS. Клиентската част представлява всичко, случващо се в браузъра. Това е мястото, където се изпълнява JavaScript, позиционират се HTML елементите и се прилагат каскадните стилове. Сървърната част, от своя страна, е всичко случващо се на сървъра на другия края на връзката, представен от нашия компютър. Заявката(request), която се стартира от браузъра изминава пътя си, най-често по интернет мрежа, за да изпълни някакъв код на сървърната част и да върне подходящ отговор(response). Браузърът не знае за технологията, която сървъра използва или за езика, на който е написан сървърния код. Сървърната страна всъщност е и мястото където се изпълнява и C# кода ни.

Текущият проект работи локално и не се нуждае от достъп до интернет, като устройството, където се намира браузърът, от който се изпращат заявките, и самият сървър, който ги обработва, са един и същ компютър. Уеб сървърите, както на администраторското приложение, така и на потребителското, се помещават на един компютър, деплойнти под Internet Information Services(IIS), което е Microsoft пакет от сървърни услуги. IIS e изключително подходящ както за локално деплойване на проекти при разработка, така и за по нататъшното им деплойване на продуктова среда. Той е изцяло Microsoft базиран и в случая перфектно се вписва в концепцията на разработваната уеб платформа. При пускане на приложението за масова употреба от множество потребители може да бъде избран подходящ план за хостинг, чрез който уеб платформата ще може да бъде достъпвана през интернет от различните потребители.

Нека се върнем обратно на темата за двете страни на уеб приложенията. Независимо какъв браузър използвате – било то Internet Explorer, Firefox или пък Chrome, фактът, че той може да работи само с HTML, CSS и JavaScript, остава непроменен. Той не може да разбере C# код или друг подобен програмен език. Това е причината потребителя да може да получи достъп до което и да било приложение, независимо от технологията, която е използвана за изграждането му, чрез един и същи браузър, тъй като той се интересува само от посочените по-горе езици. Целта на всеки един фреймуърк за разработка на уеб приложения, било то ASP.NET Core 3.1 или .NET Framework 4.6.1, които ние използваме, е да преобразува сървърния код до HTML, CSS и JavaScript. Това може да се осъществява по различни начини в зависимост от използвания фреймуърк.

### Рутиращ механизъм(Routing) в администраторското приложение

Рутирането е отговорно за съпоставянето на входящите HTTP заявки и изпращането на тези заявки до изпълнимите крайни точки на приложението(executable endpoints). Крайните точки са части от приложението, които отговарят за изпълнимия код за обработка на заявките. Те се дефинират в самото приложение се конфигурират при неговото стартиране. Процесът по съпоставяне на крайна точка може да извлече стойности от URL адреса на заявката и да предостави тези стойности за обработка на заявката. Използвайки информация за крайната точка от приложението, рутирането също е в състояние да генерира URL адреси, които се свързват с крайните точки.[10]

За конфигурацията на рутирането в администраторското приложение, ще използваме контролерите част от MVC шаблона. Контролерите в ASP.NET Core използват рутиращ мидълуеър(routing middleware), който представлява софтуер, вграден в пайплайна(pipeline) на приложението, за да обработва заявките и върнатите отговори. [11] Т.нар. рутиращи шаблони се дефинират в стартовия код(startup code) или в атрибути, описват как URL пътищата се съпоставят на действия(actions) и се използват за генериране на URL-и за линкове, които най-често се връщат от отговорите на заявките. Действията са или конвенционално или атрибутно рутирани. Ние ще използваме конвенцоналното рутиране. За целите на проекта ни е необходимо да имаме следния код в Configure метода в Startup.cs файла:

app.UseEndpoints(endpoints =>

{

endpoints.MapControllerRoute(

name: "default",

pattern: "{controller=Login}/{action=Login}/{id?}");

});

В извикването на UseEndpoints, MapControllerRoute[12] се използва за създаването на единичен маршрут, наречен дефолтен или маршрут по подразбиране. Той е подходящ и за нашата уеб платформа. Какво всъщност означава той обаче? Подобен шаблон би съвпаднал със следния URL път - Users/Details/c14f9665-a149-456a-a6e8-2a46bff9c793. Той извлича стойностите чрез т.нар tokenizing на пътя. В резултат на това се търси дали има съвпадение, при което е наличен UsersController с действие Details:

public IActionResult Details(Guid id)

{

if (id == null)

{

return NotFound();

}

var userToParkingSpaceEntity = this.GetUserToParkingSpaceEntity(id);

return View(userToParkingSpaceEntity);}

Users/Details/c14f9665-a149-456a-a6e8-2a46bff9c793 използва свързване по модел(model binding) свързва стойността id = c14f9665-a149-456a-a6e8-2a46bff9c793, за да предаде този уникален номер на id параметъра на действието. {controller=Login} определя Login като контролер по подразбиране. {action=Login} определя Login като действие по подразбиране. Въпросителният знак след {id?} го определя като незадължителен параметър, което означава, че за него не е необходимо да бъде зададена конкретна стойност, за да бъде заредена дадена страница.

Навигирането между страниците на администраторското приложение е изключително важно. Рутиращият процес спомага за пълноценното му осъществяване, а избраният за него .NET Core фреймуърк ни предоставя сравнително лесна възможност за конфигурация, като спестява писането на допълнителен сложен код, който да извършва този процес.

### Крайни точки(endpoints) в потребителското приложение

WCF клиент, какъвто се явява Angular приложението в платформата, се свързва с WCF сървис чрез крайна точка(endpoint). Всеки сървис предоставя договора(contract) си чрез една или повече крайни точки. Крайната точка има адрес (който е URL, указващ къде може да се осъществи достъп до крайната точка) и допълнителни свойства, които определят как ще се прехвърлят данните. Контрактите служат като своеобразни договори между сървърната и клиентската част. WCF контракт определя какво всъщност прави един сървис или действието, което един клиент може да извършва в него. Контрактът е един от елементите на WCF endpoint, който съдържа информация за самия WCF сървис. Контрактът също спомага за сериализирне на информацията, използвана в сървиса. По принцип има 2 тип контракти - Service Contracts и Data Contracts, като ние ще се спрем на Service Contracts, тъй като те се използват в интерефейсите на нашите сървиси. Service Contract-ът декларира интерфейс в WCF сървиса за клиента, така че той да може да осъществи достъп до него. Като податрибут е използван Operation Contract, който декларира метод вътре в интерфейса, който ще бъде извикван от клинета. Ако не бъде използван този атрибут, като предхождащ за даден метод, то той не би могъл да бъде извикан от клиента. [666] <https://www.c-sharpcorner.com/UploadFile/87b416/various-types-of-wcf-contracts/> Още един атрибут използван за методите на интерфейсите е WebInvoke. Той индикира, че дадена сървисна операция представлява операция, която ще бъде извиквана, и че се използва т.нар WCF REST programming model. Този атрибут разполага с различни свойства, като в настоящата имплементация са използвани Method, който отговаря за указване на метод на протокола, за обмен на данни, който ще бъде използван от функцията – напр. GET(HTTP протокол), UriTemplate, който представлява Uniform Resource Identifier template или шаблон, уникално име, с което ще бъде разпознат метода, и ResponseFormat, който определя в какъв формат ще бъде върнатия от сръвиса респонс(отговор), в случая JSON. [666] <https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.web.webinvokeattribute?view=netframework-4.8>

Използването на JSON формат и съответно JSON сериализация прави WCF платформата независима. Когато клиентът иска да получи достъп до сървиса чрез крайна точка, той не само трябва да знае пътят, чрез който да достъпи дадения метод , но също така трябва да се придържа и към метода за обмен на данни, посочен от крайната точка. По този начин и клиентът, и сървърът трябва ще имат съвместими крайни точки. Пример за това са описаните крайни точки на WCF сървисите в сървърната част:

[OperationContract]

[WebInvoke(Method = "GET",

UriTemplate = "CurrentUser",

ResponseFormat = WebMessageFormat.Json)]

DomainModels.CurrentUser GetCurrentUser();

и тяхната съпоставка от другата страна в Angular клиента:

getCurrentUserData(): Observable<ICurrentUser> {

return this.http.get<ICurrentUser>(this.baseUri + '/UserService.svc/CurrentUser');}

## 2.4 Архитектура на базата данни

Съвкупността от данни е структурирана по начин, който позволява лесното и бързо извличане, преглеждане, търсене и свежда до минимум дублирането на информация. За целта се използва база от данни. Базите от данни са независими от софтуера и това ги прави универсални за използване, както от различните приложения, така и от различните периоди от време.

Софтуерът, който осигурява възможността за работа с бази от данни, се нарича Система за управление на бази от данни (СУБД). Тя включва три основни компонента:

* Средства за разработване на приложения;
* Потребителски интерфейс;
* Ядро, което извършва операциите по търсене, сортиране и актуализиране на базата от данни.

В зависимост от това как базата данни описва връзките, съществуват три модела:

* Йерархичен – данните са представени под формата на дърво, състоящо се от корен и възли. Те са описани чрез връзки 1:N, насочени надолу. Не се позволяват връзки от тип M:N;
* Мрежов – данните са представени под формата на ориентиран граф. Отново не са връзки от тип М:N. Ако такива съществуват, те се разбиват на две връзки от вида 1:N;
* Релационен – в този модел данните се съхраняват в таблици, наречени още релации, между които има зададени връзки. Всяка таблица се състои от колони и редове.[19]

За конкретното решение е избрана релационна база от данни. Структурата на базата данни се обуславя на основата на обектите, които се съхраняват в нея.

Приложението дава възможност да бъде използвано свободно от всички потребители с регистрация в системата. Всеки потребител на платформата може да си създаде акаунт и да добави информация за автомобила си, да редактира или изтрие тази информация, да разгледа наличните за резервиране паркоместа и да осъществи контакт с администратора на платформата при подаване на заявка за паркомясто, възникнал проблем или просто за обратна връзка, чрез директно отваряне на имейл форма, в която има предварително зададен съответния имейл и тема на писмото. Дефинирана е и специална администраторска роля и предварително е регистриран потребител от нея. Чрез този потребител се осигурява управление на всички паркоместа в системата. Тази форма на контрол се използва при създаване на връзка между даден потребител и паркомясто, както и при премахването й. Това е краткото описание на приложението, което определя основните таблици и връзките между тях в базата от данни.

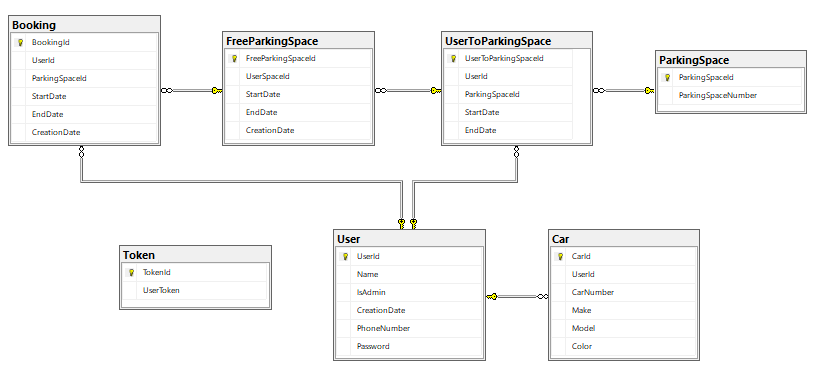
Основните обекти, които определят цялата система са потребителите, автомобилите им, паркоместата и резервациите на паркоместа. Таблицата с потребителите съдържа информация за уникалния номер на потребителя, който играе ролята на първичен ключ(primary key), име и телефон на потребителя, флаг дали той е администратор, дата на създаване на този потребител в системата, както и паролата на потребителя в хеширан вид. Не пазим по-подробна информация за отделните потребители, поради подхода, който сме избрали за изграждане на приложението, а именно ориентиран и към потребители без постоянно място в системата, който предполага резервирането на паркомясто и за по-кратко време. Таблиците за автомобили, паркоместа и резервации също съдържат уникален идентификационен номер, както и прилежащи данни за по-подробно характеризиране на обекта, като при тях има и специално обособени колони, които играят роли на външни ключове(foreign keys) към другите таблици. За тип на съхраняваните данни са избрани съответно uniqueidentifier за всички ключове в таблиците, nvarchar за текстовите полета, bit за флаговете и date за дати.

От тук следва, че са необходими таблица за потребителите, таблица за автомобилите им, таблица за паркоместата и запазването на паркоместа. Това са очевидните и задължителни таблици за нашата база данни. След кратък анализ се установява и нуждата от наличие на таблица за съхранение на свободните паркоместа, както и от свързваща таблица между потребителите и паркоместата. Добавена и таблица са съхранение на тоукъни, които имат участие в авторизиращата функционалност на системата при регистрация и се генерират и проверяват при логване в системата. Връзката на таблицата с автомобилите с тази с потребители в базата е „едно към едно“(1:1). Например един потребител може да има само една кола. Един потребител във времето може да използва много паркоместа, също и едно паркомясто може да бъде използвано от много потребители във времето. Тук споменахме думата много два пъти - това ни дава сигнал, че е редно да използваме връзка „много към много“(M:N). Тя се осъществява като между таблицата потребители и паркоместа се добавя свързваща таблица – UserToParkingSpace и в нея се добавят външни ключове към двете таблици, които представляват уникалните идентификатори на потребителя и паркомястото. Тази таблица съдържа и стартова и крайна дата, които обособяват периода, за който има връзка между съответния потребител и паркомясто. При връзка „много към много“ таблиците за потребители и паркоместа са свързани с връзка „едно към много“(1:M) със свързващата таблица. От своя страна свързващата таблица има връзка „едно към много“(1:M) с тази със свободните паркоместа, което ще рече, че една връзка собственик-паркомясто може да представлява много свободни места за различен интервал от време. Таблицата за свободни места се свързва с таблицата за резервации с връзка от тип „едно към едно“(1:1) т.е една свободно паркомясто може да има само едно резервация. От друга страна пък потребителската таблица има връзка с тази за резервации „едно към много“(1:M) т.е един потребител може да има много резервации. Нуждата от обуславяне на привидно сходни таблици като UserToParkingSpace и Booking е продиктувана от наличието на двете роли в системата – потребители със собствено паркомясто и такива без, които просто наемат, без да отдават. В таблицата UserToParkingSpace фигурират само собственици с техните паркоместа, докато в Booking се записва ид-то на резервиращия потребител, а в колоната ParkingSpaceId от същата таблица – уникалният номер на свободното място за паркиране и вече от таблицата за свободни паркоместа по UserSpaceId може да бъде проследен реалният номер на паркомястото, както и неговият собственик, който е различен от наемателят му.

Друга неразделна част от базата данни, проектирана за системата, са съхранените процедури(stored procedures). Те представляват предварително компилиран набор от един или повече SQL изрази, които се съхраняват в системата за управление на релационните бази данни, в случая SQL Server. Основното предимство на съхранените процедури е, че те се изпълняват от страна на сървъра и изпълняват набор от действия, преди да върнат резултатите на клиентската страна. Това позволява набор от действия да бъдат изпълнени за минимално време и също така намалява мрежовия трафик. Това са основните причини да бъде избран подходът с процедури пред този с LINQ завяки, директно написани в кода. Други предимства на съхранените процедури пред LINQ са, че са значително по-бързи, тъй като имат предвидим план за изпълнение и могат да се възползват напълно от SQL функциите. Следователно, когато съхранявана процедура се изпълнява повторен път, базата данни използва кеширания план за изпълнение, за да изпълни тази съхранена процедура. Процедурата е по-добрият начин за писане на сложни заявки в сравнение с LINQ, неправилно написана LINQ значително би влошила цялостното представяне на приложението и при потенциални промени, за да бъде доставени те не клиента, би било необходимо рекомпилиране и повторен деплой на библиотеки на сървъра. [666, <https://www.dotnettricks.com/learn/linq/comparing-linq-with-stored-procedure>]

За реализиране модела на базата данни са използвани Entitiy Framework(за потребителското приложение) и Entitiy Framework Core[23] (за администраторското), като и при двата е използван database-first подход, при който таблиците и връзките между тях предварително се описват чрез SQL заявки и после edmx моделите или класовете, репрезентиращи таблиците в базата, се обновяват спрямо промените в базата данни.

За да се покаже цялата структурата на базата от данни е генерирана диаграма с помощта на Microsoft SQL Server и е представена на фиг.4



Фиг. 4 Диаграма на база данни ParkingManagementSystem

# Въведение в използваните технологии

### Microsoft Visual Studio 2019

Microsoft Visual Studio е интегрирана среда за разработка (Integrated Development Environment - IDE) на софтуерни приложения за операционна система Windows и за платформа .NET Framework. Използва се за разработката на конзолни приложения, както и Windows Forms, Windows Presentation Foundation (WPF) приложения, уеб сайтове, уеб услуги, уеб приложения на всички поддържани платформи от Microsoft Windows. Microsoft Visual Studio поддържа различни езици за програмиране (C++, C#, VB.NET, F#, XML, HTML, JavaScript, CSS и т.н.) и различни технологии за разработка на софтуер (Win32, COM, ASP.NET, ASP.NET Core, ADO.NET Entity Framework, Entity Framework Core, Windows Forms, WPF, Silverlight и много други).

За реализацията на настоящата дипломна работа е избранa средата Microsoft Visual Studio 2019. Тя дава възможност за лесно използване на MVC шаблона и трислойната архитектура, като предлага някои предварително дефинирани темплейти. Освен това, е необходимо условие за използването на ASP.NET Core 3.1 Framework, на който се базира администраторското приложение. [21]

### .NET Framework

.NET Framework е софтуерен фреймуърк, разработена от Microsoft, който работи предимно на Microsoft Windows. Той включва голама библиотека с класове, наречена Framework Class Library (FCL) и осигурява езикова интероперативност (всеки език може да използва код, написан на други езици) в няколко езика за програмиране. Програмите, написани за .NET Framework, се изпълняват в софтуерна среда (за разлика от хардуерната среда), наречена Common Language Runtime (CLR). CLR е виртуална машина за приложения, която предоставя услуги като защита, управление на паметта и обработка на изключения. Като такъв, компютърният код, написан с помощта на .NET Framework, се нарича „управляван код“. FCL и CLR заедно съставляват .NET Framework. FCL осигурява потребителския интерфейс, достъпа до данни, свързаност с бази данни, криптография, разработка на уеб приложения, цифрови алгоритми и мрежови комуникации. Програмистите произвеждат софтуер чрез комбиниране на написания код с .NET Framework и други библиотеки. Фреймуъркът е предназначен за използване от повечето нови приложения, създадени за Windows платформата. [666, <https://en.wikipedia.org/wiki/.NET_Framework>]

Заедно с .NET Framework e използван Entity Framework (EF). Той предсатвялва обектно-релационен мапър (object-relational mapper, O / RM) с отворен код за .NET приложения, поддържани от Microsoft. Тя дава възможност на разработчиците да работят с данни, използвайки обекти от специфични за домейна класове, без да се фокусират върху основните таблици и колони в базата данни, където се съхраняват тези данни. С Entity Framework разработчиците могат да работят на по-високо ниво на абстракция, когато се занимават с данни, и могат да създават и поддържат ориентирани към данни приложения с по-малко код в сравнение с традиционните приложения. [666, <https://www.entityframeworktutorial.net/what-is-entityframework.aspx>]

### ASP.NET Core

ASP.NET Core e безплатен open-source фреймуърк за уеб разработка и наследник на ASP.NET , разработен от Microsoft. Представлява модуларен фреймуърк, който работи както с пълният .NET фреймуърк под Уиндоус, така и междуплатформено. Той може да работи с трите най-използвани операционни системи - macOS, Linux и Windows. Фреймуъркът е изцяло пренаписан, обединявайки преди това отделните ASP.NET MVC и ASP.NET Web API в единен модел за програмиране. Той разполага с множество удобства като т.нар side by side versioning, при който приложенията на ASP.NET Core, работещи на една и съща машина, могат да се насочват към различни версии на ASP.NET Core. Удобен и специално оптимизиран е за разработка в облачна среда, работи с олекотен и модуларен HTTP пайплайн, притежава вградена поддръжка на dependency injection и още много други.[22] За текущата реализация на администраторското приложение е използван ASP.NET Core 3.1.

Заедно с ASP.NET Core е използван Entity Framework (EF) Core, който също представлява олекотена, open source и междуплатформена версия на популярната технология за достъп до данни Entity Framework. EF Core може да служи като обектно-релационен мапър (object-relational mapper, O / RM), като позволява на .NET разработчиците да работят с база данни с .NET обекти и елиминира нуждата от писането на обемен код за достъп до данни, който обикновено трябва да бъде написан. EF Core поддържа много и различни database engines.[23]

### Microsoft SQL Server

Microsoft SQL Server е сървърна система за управление на бази от данни. Той представлява софтуерен продукт, чиято основна функция е да съхранява и да предоставя данните, когато други софтуерни продукти, направят запитвания, независимо дали са на същия компютър или са свързани в мрежа (включително и интернет). Transact-SQL (или T-SQL) е езикът, който използва SQL Server. Microsoft SQL Server се използва за релационни бази от данни. [16]

Релационната база от данни е колекция от данни, които са организирани в групи официално описани като таблици, от които може да се достъпват или събират данни, отново по много различни начини, без да се реорганизират таблиците. Стандартният език, чрез който се комуникира с базата от данни е SQL (Structured Query Language). Всяка таблица (наричана също релация) съдържа една или повече категории данни в колони. Всеки ред съдържа уникална инстанция на данните за категориите, определени от колоните.

За платформата за управление на паркоместа е избран продукта Microsoft SQL Server 2017 Developer Edition, който е безплатен.

### HTTP протокол

За комуникация между клиентските и сървърните части на нашата уеб платформа(администраторско и потребителско приложение) ще използваме добре познатия и наложил се като ефективен при изграждане на уеб проекти HTTP протокол. Той се базира на заявка-отговор концепцията(request-response) при клиент-сървър моделите. Клиентската част изпраща HTTP заявка със съобщение към сървърната част. Тя осигурява ресурси, като да речем HTML файлове и друго подобно съдържание, или извършва някакво действие, предизвикано от клиента и след това връща съобщение с отговор обратно към клиента. Отговорът съдържа статус за нивото на изпълнение на заявката, като може и да съдържа заявеното съдържание в тялото на съобщението. HTTP ресурсите се идентифицират и се намират в мрежата от т.нар Uniform Resource Locators (URLs), използвайки Uniform Resource Identifiers (URI's) схеми като http и https. [9] Въпреки че всички заявки, използващи HTTP протокола, се базират на request-response шаблона, начините, по които те се изпращат, може да варират.HTTP методът определя как заявката се изпраща към сървъра. Съществуващите HTTP методи са GET, HEAD, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT и PATCH. Най-често използвани са GET и POST и именно на тях ще се спрем по-подробно, тъй като те са и основните методи в нашето приложение.

* **GET** е метод на HTTP протокола, който се използва за получаване на ресурс от сървъра. Заявките, които използват GET метода трябва само да извличат данни и не би трябвало да пораждат никакви странични ефекти в следствие на този процес. Това означава, че при многократното изпълнение на такъв тип заявка, би трябвало да получаваме един и същ резултат и не би трябвало да има каквато и да е промяна в състоянието на сървъра в резултат на изпълнението на GET заявката. При GET метода параметрите се изпращат като част от URL-а на заявката и поради това са видими за крайния потребител. Предимство при този подход e, че потребителят може да си „отбележи“(bookmark) конкретното URL и да посети страницата отново когато пожелае. Подаването на параметри по този начин е доста удобно и именно поради това се използва в метода за търсене в администраторското приложение, на който се подава въведения от потребителя текст. При така подадените параметри, базата с данни се претърсва за записи, отговарящи на посочените критерии и връща отговор с намерените резултати, ако има такива, и в последствие те се представят по подходящ начин на администратора. GET метод се използва и при извличането на детайлна информация за връзките между собствениците и техните паркоместа, при взимането на информация за текущо логнатия потребител и при всякакви действия, вързани с извличане на данни от базата, вследствие на предизвикано действие от потребителя.
* **POST** методът също е метод на HTTP протокола, но той се използва за обновяване или създаване на ресурси на сървъра. Характерното при него е, че данните се предават в тялото на заявката. В резултат на това, чувствителна информация може да бъде изпращана към сървъра, тъй като данните са вградени в тялото на самата заявка и не са видими в URL-a за крайния потребител. Тъй като данните не се изпращат в URL-a адреса, те не заемат излишно място в него и следователно не се наблюдават проблеми, свързани с ограничението в дължината на адреса.

### Bootstrap

Bootstrap е фреймуърк, който позволява бързото и лесно създаване на дизайн за уеб сайтове. Включва HTML и CSS базирани шаблони за дизайн за типография, форми, бутони, таблици, навигация, модали, изображения и др. Също така предоставя поддръжка за JavaScript библиотеки. Основно негово предимство е адаптивния CSS, който може да бъде използван за дизайн, подходящ за различни устройства – телефони, таблети и настолни компютри. Компонентите му по подразбиране лесно могат да бъдат модифицирани според конкретните нужди на разработчика. Съвместим е с почти всички съвременни браузъри - Chrome, Firefox, Internet Explorer, Safari и Opera. [24]

### Angular

Angular е платформа и фреймуърк за изграждане на т.нар single-page клиентски приложения, използващ HTML и TypeScript. Single-page application (SPA) е уеб приложение или уебсайт, който взаимодейства с потребителя, като динамично пренаписва текущата уеб страница с нови данни от уеб сървъра, вместо да използва метода по подразбиране на уеб браузъра, който зарежда цели нови страници. Целта е по-бърза навигация, които карат уебсайта да се чувства по-скоро като приложение, характерно за операционната система на устройството, на което се използва. Angular е написан на TypeScript. Той реализира основна и незадължителна функционалност като набор от TypeScript библиотеки, които се импортират в приложенията. Като платформа, Angular включва в себе си компонентно базиран фреймуърк за изграждане на мащабируеми уеб приложения, колекция от добре интегрирани библиотеки, които обхващат голямо разнообразие от функции, включително рутиране, управление на форми, комуникация клиент-сървър и др., пакет от инструменти за програмисти, които помагат за разработката, изграждането, тестването и актуализирането на кода. [666, <https://angular.io/guide/what-is-angular>]

# 4. Програмна реализация - ParkingManagementSystem Solution

Solution-a представлява структура от различни по тип проекти, обособени спрямо тяхното предназначение. Организиран е така, за да покаже как две отделни приложения, използващи различни технологии, могат да бъдат обединени в една структура и да взаимодействат помежду си. ParkingAdministration проектът представлява администраторското приложение, а останалите – потребителския портал.

* ParkingAdministration
* WebClientCore
* WebHost
* WebHost.Data
* WebHost.Repository

Всяка един от проектите ще бъде разгледан подробно.

## 4.1 ParkingAdministration project

Проектът представлява .NET Core приложението, отговарящо за администраторските функционалности. То е изградено спрямо MVC шаблона и използва типичната за него структура, като е допълнено с някои помощни папки с цел по-добра организация на проекта. Това са съставните компоненти:

* Models
* Controllers
* Views
* Data

### 4.1.1 Папка Models

В тази секция е описан класът, който е създаден в частта Model на MVC шаблона - UserToParkingSpaceEntity.cs. Той е съставен от пропъртита с get и set методи, които са от съответните типове, описващи съдържащите са данни в базата. Направен е така, че максимално пълноценно да описва необходимата информация, която ще бъде показвана в администраторския панел.

namespace ParkingAdministration.Models

{

public class UserToParkingSpaceEntity

{

public UserToParkingSpace UserToParkingSpace { get; set; }

public Car Vehicle { get; set; }

public string PhoneNumber { get; set; }

}

}

За по-голяма прегледност класовете, описващи таблиците в базата, са обособени в отделна папка – Data.

* Booking.cs
* Car.cs
* FreeParkingSpace.cs
* ParkingSpace.cs
* Token.cs
* User.cs
* UserToParkingSpace.cs
* ParkingManagementSystemContext.cs

Те са изградени на базата на database-fisrt подход и използват C# типове, за да опишат максимално точно типовете на колоните в базата, както и връзките между тях. Това е често използван подход, предпочитан от голяма част от програмистите поради своята гъвкавост и интуитивност на работата, и доказал се във времето като изключително надежден. Enitity Framework Core не предлага визуален дизайнер за моделиране на обектите от базата или пък помощник(wizard) при генерирането им, както е при Entity Framework, който ще разгледаме по-късно. Затова използваме т.нар scaffolding процес. Чрез нея ще бъдат създадени entity(обектни) и context(контекстен) клас, базирайки се на схемата на вече съществуващата база данни. Съществуват различни параметри, които могат да бъдат подадени на командата за scaffolding:

Scaffold-DbContext [-Connection] [-Provider] [-OutputDir] [-Context] [-Schemas>] [-Tables>] [-DataAnnotations] [-Force] [-Project] [-StartupProject] [<CommonParameters>]

Тя се изпълнява в Package Manger конзолата(Tools -> NuGet Package Manger -> Package Manger Console) и в нашия случай се изпълнява със следните параметри:

PM>Scaffold-DbContext "Server=DESKTOP-LGMEH2L\SQLSERVER2017;Database=ParkingManagementSystem;Trusted\_Connection=True;" Microsoft.EntityFrameworkCore.SqlServer -OutputDir Data,

като указваме името на сървъра, базата, чиято схема ще използваме, информация за сигурността т.е да се използва доверена връзка през Уиндоус автентикаиция, кой провайдър за SQL Server използваме и име на директорията, където ще бъдат съхранени моделите.

### Папка Contollers

В тази секция са описани класовете, които са създадени в частта Controller на MVC шаблона. Тя е и може би най-съществената от целия проект, тъй като в нея е концентрирана основната бизнес логика на приложението. За всеки обект, носещ съществено смислово значение за администраторското приложение, е изграден собствен контролер, в който са поместени основните методи за работа с него. Всички контролери наследяват базовия Controller клас - System.Web.Mvc.Controller. По този начин няколко полезни метода на родителския клас могат да се използват от неговите наследници. Ако искаме да върнем изглед като резултат от ActionResult метода на контролера към браузъра, тогава ще извикаме View() метода на базовия клас. Ако искаме да пренасочим потребителя от едно действие на контролера към друго, ще извикаме RedirectToAction() метода.[13]

Първо ще изброим всички налични контролери в структурата, а в последствие ще се спрем по-подробно на всеки един от тях и ще опишем основните функционалности, за които той е отговорен.

* UsersController
* LoginController
* HomeController
* BaseController

#### UsersController

Eдин от основните смислови обекти, около които е организирано приложението, е потребителят. Именно поради това неговият контролер UsersController е най-комплексен и отговорен за най-голям брой функционалности. Тук се помещават т.нар CRUD операции (Create, Read, Update и Delete). Създаването на връзка собственик-паркомясто, обновяването и изтриването на тези връзки за всички потребители в системата са достъпни само за администраторската роля и именно затова приложението предоставя възможност за работа всичките базови операции с данни.

Методът за създаване на връзка собственик-паркомясто е маркиран с HttpPost атрибут, който указва, че от него се очаква да осъществява промени по базата данни, в случая, да добавя нов запис към таблицата UserToParkingSpace. Методът е от тип IActionResult, който е характерен при възможност за връщане на множество ActionResult връщани типове в дадено действие(action). Той приема обект от тип UserToParkingSpaceEntity. Най-напред се валидира самата заявка за създаване на нов обект чрез метода ValidateCreateRequest на BaseController-а, който ще опишем отделно. В метода се прави и проверка дали състоянието на модела е валидно чрез свойството на базовия MVC контролер(ModelState.IsValid) и при успешна проверка се създава нов обект от тип UserToParkingSpaces, чиито свойства се приравняват на свойствата, получени от view модела. Eстествено тук се извършват и необходими валидации като дали потребителя няма вече зададено паркомясто и дали има свързан към него автомобил т.е дали потребителят е попълнил необходимата информация за колата си. Едва след тях, към инстанция на ParkingManagementSystemContext и по-точно към сет от данни(DbSet< UserToParkingSpace>) се добавя вече попълнения обект, отговарящи на съответния тип. След успешното добавяне на обява, потребителят е пренасочен, чрез RedirectToAction() метода отново на базовия MVC контролер клас, към инициализиращото действие – Index, което се грижи за показването на списъка с вече отдадени на собственици паркоместа, до който само администратора има достъп. Ако има някакъв установен проблем със състоянието на модела, администраторът остава на същата страница за задаване на паркомясто към потребител.

[HttpPost]

public async Task<IActionResult> Create([Bind("UserToParkingSpace,Vehicle,PhoneNumber")] UserToParkingSpaceEntity userToParkingSpaceEntity)

{

try

{

string userName = userToParkingSpaceEntity.UserToParkingSpace.User.Name;

userToParkingSpaceEntity.UserToParkingSpace.User = null;

ViewData["ParkingSpaceId"] = new SelectList(this.\_context.ParkingSpaces.OrderBy(x => x.ParkingSpaceNumber), "ParkingSpaceId", "ParkingSpaceNumber", userToParkingSpaceEntity.UserToParkingSpace.ParkingSpaceId);

ViewData["UserId"] = getUsers(userName);

this.ValidateCreateRequest(userToParkingSpaceEntity);

if (ModelState.IsValid)

{

var user = this.\_context.Users

.FirstOrDefault(u => u.Name.Equals(userName));

if (user == null)

{

user = createUser(userName);

}

var userToParkingSpace = this.\_context.UserToParkingSpaces

.FirstOrDefault(x => x.UserId == user.UserId && x.EndDate != null);

if (userToParkingSpace != null)

{

throw new ArgumentException("User already have assigned parking space.");

}

var car = this.\_context.Cars

.FirstOrDefault(x => x.UserId == user.UserId);

if (car == null &&

(userToParkingSpaceEntity.Vehicle.CarNumber == null ||

userToParkingSpaceEntity.PhoneNumber == null))

{

throw new ArgumentException("User doesn't have releted car. User should have entered the needed information about their car!");

}

userToParkingSpaceEntity.UserToParkingSpace.UserId = user.UserId;

userToParkingSpaceEntity.UserToParkingSpace.UserToParkingSpaceId = Guid.NewGuid();

this.\_context.UserToParkingSpaces.Add(userToParkingSpaceEntity.UserToParkingSpace);

await this.\_context.SaveChangesAsync();

return RedirectToAction(nameof(Index));

}

return View(userToParkingSpaceEntity.UserToParkingSpace);

}

catch (Exception ex)

{

this.Notify(ex.Message, $"Failed to set parking space.", NotificationType.error);

return View(userToParkingSpaceEntity);

}

}

В UsersController класа разполагаме и с още един Create метод със същата сигнатура като гореспоменатия, но с тази разлика, че той не приема никакви параметри и не прави промени в базата. Той се извиква при първоначалното извикване на Create изгледа и се грижи за това да подаде данни от базата към падащите менюта, от които потребителя избира стойности за номера на паркомястото и името на потребителя, към когото иска то да бъде съпоставено то. Това става чрез т.нар ViewData, който представлява речник, съдържащ двойка ключ-стойност, където ключът трябва да бъде от тип низ. В конкретния случай ключът е уникалният идентификационен номер на паркомястото, а стойността – самият номер на мястото. Същото е направено и за падащия списък с потребители. Целта на ViewData е да служи като посредник между конторолите и изгледите в MVC архитектурата.

ViewData["ParkingSpaceId"] = new SelectList(\_context.ParkingSpaces.OrderBy(x => x.ParkingSpaceNumber), "ParkingSpaceId", "ParkingSpaceNumber");

ViewData["UserId"] = getUsers(null);

Разглеждането на вече отдадените паркоместа също е съществена част от интеракцията на администратора със системата. Условно тя може да се раздели на две – преглеждане на отдадените места в списък и преглеждане на детайлите на дадено паркомясто. За всяка една са реализирани отделни методи. Методът, който отговаря за показването на обявите в списък, всъщност е и Index метода на контролера и приема стрингов параметър, който представлява въведения текст за търсене. Първо се проверява дали в cookies има закачен параметър user\_id, който се добавя при успешен логин и ако има, се продължава със същинската работа на метода, в противен случай се прави редирект към логин страницата. От кореспондиращия изглед се взима въведения в полето за търсене текст, отново чрез ViewData["FindEmployee"] = searchString; и съответно се извършва търсене по потребителско име, телефонен номер и номер на паркомястото. Това са задължителни полета в системата и именно затова са избрани като най-уместни критерии за търсене. Ако не е въведено нищо, съответно се показват всички отдадени паркоместа без наложен филтър и подредени първо по номер на паркомястото и после по крайна дата, за която са отдадени.

[HttpGet]

public async Task<IActionResult> Index(string searchString)

{

var userId = HttpContext.Request.Cookies["user\_id"];

if (userId != null || !String.IsNullOrEmpty(userId))

{

ViewData["FindEmployee"] = searchString;

var query = from x in this.\_context.UserToParkingSpaces

.Include(u => u.ParkingSpace)

.Include(u => u.User)

select x;

if (!String.IsNullOrEmpty(searchString))

{

query = query.Where(x => x.User.Name.Contains(searchString) ||

x.User.PhoneNumber.Contains(searchString) ||

x.ParkingSpace.ParkingSpaceNumber.Contains(searchString));

}

var orderedQuery = query

.OrderBy(x => x.ParkingSpace.ParkingSpaceNumber)

.ThenBy(x => x.EndDate);

return View(await orderedQuery.ToListAsync());

}

else

{

return RedirectToAction("Login", "Login");

}

}

Разбира се има и възможност за подробно разглеждане на детайлите на всяка една от връзките паркомясто-собственик. При кликване върху менюто с трите точки и избиране на опцията “Details”, потребителят е прехвърлен на страница с детайлите за връзката. Самият метод за показване на детайлите приема уникален номер на двойка паркомясто-собственик – id и по него връща изглед с подадената връзка, отговаряща на този номер. Използва се метода GetUserToParkingSpaceEntity, който работи с контекста на системата и съдържа в себе си репрезентации на таблиците от базата под формата на сетове от данни - DbSets.

public IActionResult Details(Guid id)

{

if (id == null)

{

return NotFound();

}

var userToParkingSpaceEntity = this.GetUserToParkingSpaceEntity(id);

return View(userToParkingSpaceEntity);

}

Последните две базови функционалности, свързани с манипулирането на връзки паркоместа-собственици, са обновяването и изтриването на вече съществуващи такива. Те са неизменна част от профила на администратора. Програмната им реализация ще бъде разгледана паралелно поради голямото сходство. И при двете операции имаме два асинхронни метода, които се изпълняват преди същинската част на операциите. Тяхната цел е да извлекат от базата данни по подаден уникален номер – id връзката собственик-паркомясто, която ще бъде манипулирана. И двата връщат изглед, на който се подава обект от тип UserToParkingSpaceEntity. Това се прави с цел да бъде показана текущата информацията за връзката. Само при метода за редактиране отново използваме ViewData, за да можем да подадем стойности за падащите менюта. Същинските методи също са асинхронни с връщан тип Task<IActionResult> и притежават атрибута HttpPost, който указва, че те ще извършват промени по базата. За тази цел в и двата се работи с инстанция на ParkingManagementSystemContext. Методът за изтриване намира обявата по id, премахва я от класа за работа с базата и след това запазва промените асинхронно, като по този начин се уверява, че с нея не са били направени някакви конкурентни промени. Накрая потребителят е изпратен на страницата със списъка с паркоместа, както при успешното създаване на нова връзка. Редактирането има същия поток на работа, като се осъществява проверка за валидност на заявката за редактиране, както и за състоянието на самия модел, обекта от тип UserToParkingSpaceEntity се обновява, обновява се и обектът Car, тъй като той е самостоятелен и по него също могат да се правят корекции и накрая промените се запазват в базата. По-специфичното тук е, че освен id на обявата, се подават и полетата на модела - PhoneNumber, UserToParkingSpace, Vehicle, чиито стойности могат да се редактират чрез т.нар model binding[17].

#### BaseController

BaseController-a, за който стана дума по-рано, е наследяван от LoginController-a и UsersController-a, а пък той от своя страна наследява базовия MVC контролер. Това е направено с цел контролерите, които са негови наследници, да използват да речем ParkingManagementSystemContext, като инициализирането му става на едно място, а именно в конструктора на BaseController. Тук се помещават и съществени за системата функционалности като валидирането на заявка за създаване на връзка собственик-паркомясто, както и на заявка за редактиране на съществуваща. Методът ValidateCreateRequest валидира случаи като дали паркомястото има вече зададен друг собственик спрямо текущо избрания, дали даден потребител няма вече обособено паркомясто, дали паркомястото няма вече зададен друг собственик за избрания период, също и се задава ограничение крайната дата да не може да бъде преди началната, последното се реализира с метод, използван при валидиране както при създаване, така и при редакция на връзка. При неспазено условие в горния десен ъгъл се появява поп-ъп съобщение със съответната грешка. Методът ValidateEditRequest проверява дали има записи, различни от този със същото ид, които съдържат избраното паркомясто, които са още активни (End Date на записа е null), дали има записи, различни от този със същото ид, които съдържат избрания потребител, които са още активни (End Date на записа е null), дали има записи, различни от този със същото ид, в които участва избраното паркомясто и крайната дата не е null (т.е. записът е стар) и дали крайната дата на записа е по-малка от началната дата на новия запис и дали има записи, различни от този със същото ид, в които участва избрания потребител и крайната дата не е null (т.е. записа е стар) и дали крайната дата на записа е по-малка от началната дата на новия запис. Отново се активират съответните грешки, насочващи администратора.

#### LoginController

Този контролер е относително прост и съдържа в себе си 2 метода – за вход и изход от системата. Методът за логване е опростен, тъй като администраторското приложение не е достъпно за обикновения потребител ползвател на системата. Логиката за вход се свежда до проверка за валидността на стейта на логин модела, както и още една за валидност, която се изразява в това да се свери въведената от потребителя парола с предварително зададената администраторска, както и дали потребителя съществува в базата и е маркиран като администратор. При успешен логин, към кукито се прикача ид-то на потребителя, като то по-нататък служи като условие за извършване на различните операции. При неуспешен опит за влизане в системата, потребителят остава на логин страницата и се показва съответната грешка като "Incorrect username or password. Please try again.". Логаут методът пък премахва ид-то на потребителя от бисквитките и пренасочва към логин страницата.

#### 4.1.2.4 HomeController

Най-опростеният от всички контролери. В себе си той единствено проверява за налично закачено user\_id в кукито и извиква изгледа за визуализирана на началната страницата за администратора, на която е показана карта на паркинга, който ще бъде управляван от системата.

### Папка Views

В тази секция са описани cshtml страниците, които са създадени в частта View на MVC шаблона. За изграждане на потребителския интерфейс са използвани HTML, CSS, Javascript, както и Bootstrap фреймуърка, който обединява в себе си дотук изброените. Именно той се използва и за осигуряване на адаптивен потребителски интерфейс при различни резолюции. Такъв тип поведение трябва да бъде предвидено поради разнообразието от устройства, които се използват масово от потребителите. Минималните стойности, които се препоръчват от Microsoft са следните: 320 пиксела (телефон), 720 пиксела (таблет) и 1024 пиксела (лаптоп, компютър).

Изгледите, като съставна част на MVC шаблона, си комуникират с останалите компоненти. Комуникацията на изгледа с модела не са осъществява директно, а логично минава през контролера. В действието(action) на контролера се създава обект от тип на модела, обработва се и се подава на View() метода като параметър. Той от своя страна извиква изгледа с име като това на action метода на контролера. В самия изглед се използва @model, последвано от името на съответния модел, за да бъде включен самият той, заедно със свойствата си, в страницата на изгледа. Обратната комуникация също не се случва по директен начин, а отново преминава през контролера. От HTML-а частта най-често се предава (submit) форма към контролера, където се създава обект от модела и получените стойности от изгледа се предават на този обект. В самият HTML се използват таг хелъпри като asp-for, за да се укаже за кое точно свойство на модела се подава стойност, както и, когато имаме елемент, извършващ конкретно действие, asp-controller за определяне на контролера, който да бъде използван, asp-action за действието на контролера и asp-route с посочено име на параметъра, който да бъде подаден на действието на контролера. Последният не е задължителен, тъй като невинаги action методите разполагат с параметри.

<form **asp-controller**="Users" **asp-action**="Index" method="get">

<a class="action" **asp-action**="Edit" **asp-route-id**="@item.UserToParkingSpaceId">Edit</a>

Изгледите в приложението са обособени в подпапки за всеки от контролерите. За всяко действие, което е предвидено да се извършва на нова страница, е предвиден отделен изглед – например за добавяне на нова връзка собстевник-паркомясто, редактиране, изтриване и т.н. Има и споделена папка за частични изгледи(partial views), които се използват в повече от един от основните изгледи. Благодарение на Razor, който представлява ASP.NET генератор на изгледи (view engine), C# код може да бъде използван директно в cshtml страниците, като по този начин значително увеличава тяхната гъвкавост и възможности. За да бъде разпознат като C# код, блоковият сегмент или единичният оператор, трябва да бъде предхождан от кльомба - @. Пример за изграждане на логика вътре в самата HTML страница е показването на Logout бутон в навигационното меню спрямо това дали потребителят е логнат или не. Прави се проверка дали в кукитата има закачено user\_id и само ако тя е успешна, бутонът за изход от системата е видим.

@{ var userId = Context.Request.Cookies["user\_id"];

if (userId != null || !String.IsNullOrEmpty(userId))

{

<li class="nav-item" style="margin-left: auto">

<a class="nav-link text-dark" **asp-area**="" **asp-controller**="Login" **asp-action**="Logout">Log out</a>

</li>

}

}

С цел избягване повторението на код, необходим на няколко места в приложението, са използвани частични изгледи. Те се намират в споделената папка и за да бъдат чисто визуално разграничавани от останалите изгледи, се маркират с долна черта пред името си. Пример за такъв частичен изглед е \_Layout. Той представлява „скелета“ на страниците в приложението, като в себе си включва навигационното меню, клас контейнер, в който се рендерира тялото на съответната избрана страница и клас footer, който се показва винаги в долната част на страницата и съдържа информация за самото приложение.

## WebHost project

Този проект представлява средният и може би най-комплексен слой от 3-слойната архитектура на потребителското приложение. В него се съдържа същинската бизнес логика. Тук са дефинирани и имплементирани WCF сървиси, които ще бъдат експоузнати или иначе казано предоставени на клиентската част(Angular приложението, WebClientCore проект), която ще ги консумира/използва. Неговата структура е изградена от различни типове компоненти като контракти, домейн модели и може би най-важното – сървиси, като към тях като помощни се явяват мапъри между домейн моделите и класовете, представящи обектите от базата, dependency resolver-и, спомагащи за регистрирането на мапърите и класове за работа с тоукъни, които ще бъдат генерирани и използвани при регистрирането и логването на потребители. Всички компоненти са обособени по папки според предназначението си:

* Contracts
* DomainModels
* Ioc
* Mapping
* Security
* Services

## Contracts

Папката Contarct съдържа в себе си интерфейсите на използваните сървиси – този за резервиране на паркоместа и потребителския. Подробно тяхната роля беше описана в раздела „Архитектура на комуникацията между клиентската и сървърната част“.

## DomainModels

Папката DomainModels, подобно на Models от MVC шаблона, описва необходимите класове от данни, с които ще работи приложението, като те не са пряка репрезентация на таблиците от базата. Моделите групират свойства, чиито стойности могат да се попълват от базата, но могат да се надграждат като се добавят помощни свойства и се групират по най-удобния начин, така че да представят информацията възможно най-гъвкаво. Например домейн моделът CurrentUser, освен типичните за потребителя свойства като име и телефон, съдържа и такива за номер на паркомясто и резервирано за текущия ден паркомясто, като спрямо това дали последното има стойност или не, се визуализира различен лейбъл в потребителския интрефейс и т.н

public class CurrentUser

{

public string Name { get; set; }

public string ParkingSpaceNumber { get; set; }

public bool IsAdmin { get; set; }

public string PhoneNumber { get; set; }

public string BookedParkingSpaceForToday { get; set; }

}

В тази папка се намират и две подпапки – Requests и Responses, като в тях се помещават класове, описващи в свойства каква информация се очаква да бъде изпратена от клиентската част и в какъв формат - заявка, както и форматът, в който тя ще бъде върната от сървъра – отговор.

Съдържание на папката DomainModels:

* Requests
* Responses
* AuthModel
* CurrentUser
* User

## Ioc и Mapping

В тези две папки се намират два съвместно работещи класа за осъществяване на връзка или иначе казано „напасване“ на обектите, които описват точно данните от таблиците в базата данни към домейн моделите.

MappingProfile класът използва AutoMapper,[666 <https://www.infoworld.com/article/3192900/how-to-work-with-automapper-in-csharp.html>] който представлява популярна библиотека за напасване на обекти, които принадлежат към несходни типове. Тя спестява досадната работа по ръчно напасване на отделните свойства на несъвместими типове едно по едно. Именно заради това й приложение ще я използваме и ние. MappingProfile класът наследява абстрактния клас Profile, който съдържа основния използван метод в текущата имплементация – CreateMap< TSource, TDestination>(). Подават му се типове, като първият е типа източникът, а втория – типа „получател“. За текушата имплементация се извършва мапинг и в двете посоки – от ресурса, представляващ базата, към домейн моделите и обратно.

CreateMap<DomainModels.User, Data.User>();

MappingProfile-ът от своя страна се използва като обект в IocManager класа. Той разполага с два статични метода RegisterType и RegisterMapper, които се извикват в конструкторите на сървисите в проекта. IoC(inversion of control) представлява принцип в програмирането, при който се наблюдава обръщане, инверсия на потока на управление на изрази, инструкции или извикване на функции спрямо традиционния такъв. В традиционното програмиране, персонализираният код, който представя целта на програмата, се извиква в библиотеки за многократна употреба, за да се грижи за общите задачи, но с инверсия на контрола, фреймуърка извиква персонализирания или специфичен за задачата код. [666, <https://en.wikipedia.org/wiki/Inversion_of_control>]. В нашия код е използван Autofac, който представялва контейнер за прилагане именно на IoC принципа. Той управлява зависимостите между класовете(в конкретния случай тези на WebHost проекта – домейн модели и на WebHost.Data, където са поместени обектите, описващи таблиците в базата(ParkingManagementSystemEntities)), така че приложението да може лесно да бъде променяно при увеличаването му като размер и сложност. [666, <https://autofac.readthedocs.io/en/latest/>] И в двата гореспоменати метода на IocManager.cs се използва инстанция на ContainerBuilder, който позволява обикновени .NET класове да бъдат третирани като контейнери и по този начин да бъде постигната описаната модуларност на цялото приложение. ContainerBuilder-ът разполага с методи за регистриране на зависимости като типове и дори цели конфигурации, за да може да се осъществи пълноценното общуване между проектите представляващи слоя с бизнес логика и този с данни от 3-слойния модел.

## Security

В папка Security се съдържат 4 класа – JwtDecoder, JwtGenerator, SecurityKeyReader и TokenRecogniser. Както говори и името на папката, те се грижат за сигурността на потребителското приложение. За тази цел е използван JWT(JSON Web Token), който е отворен стандарт (RFC 7519), който дефинира компактен и самостоятелен начин за сигурно предаване на информация между различни страни като JSON обект. Тази информация може да бъде проверена и доверена, защото е подписана с цифров подпис. [666, <https://jwt.io/introduction>] В частност се използва jose-jwt(JavaScript Object Signing & Encryption), който представлява допълнително инсталиран към проекта NuGet пакет, за цифрово подписване и криптиране на JWT за .NET проекти.

Ще започнем с JwtGenerator-а. Той разполага с два метода GetJwt и Base64UrlDecode. В GetJwt се генерира секретен ключ, който представлява масив от байтове, чрез декодиране(тук се извиква Base64UrlDecode) на специален ключ за сигурност, който се задава в web.config-a на приложението, и представлява името на самото приложение ParkingManagementSystem. После, при хостването под IIS, е важно името на апликейшъна да бъде същото, за да сработва правилно оторизацията. След генерирането на секретния ключ, той се използва за създаване на тоукън чрез JWT.Encode, който приема обект, за който ще бъде генериран този тоукън, секретен ключ и алгоритъм, по който ще стане това, в случая - HS256.(представялва симетричен метод за подписване, което ще рече, че същия секретен ключ ще бъде използван и при създаване, и при верифициране на подипса [666, <https://www.loginradius.com/blog/async/jwt-signing-algorithms/>])

public string GetJwt(DomainModels.User targetUser)

{

byte[] secretKey = Base64UrlDecode(SecurityKeyReader.GetSecurityKey());

string token = JWT.Encode(targetUser, secretKey, JwsAlgorithm.HS256);

return token;

}

JwtDecoder, както отново подсказва името му, се грижи за декодиране на JSON Web Token-а. В него се извлича частта от рекуеста, която съдържа тоукън и след това, вече взетия тоукън, се използва за декодиране и получаване на обект от домейн типа User, чийто уникален номер ще бъде използван за връщана стойност. Начина на работа е подобен на този от GetJwt метода.

public class JwtDecoder

{

public Guid GetCurrentUserId()

{

var jwtGenerator = new JwtGenerator();

byte[] securityKey = jwtGenerator.Base64UrlDecode(SecurityKeyReader.GetSecurityKey());

var token = GetTokenFromRequest();

var user = JWT.Decode<DomainModels.User>(token, securityKey, JwsAlgorithm.HS256);

return user.UserId;

}

public string GetTokenFromRequest()

{

IncomingWebRequestContext request = WebOperationContext.Current.IncomingRequest;

WebHeaderCollection headers = request.Headers;

var token = headers.Get("token");

return token;

}

}

Последният клас е TokenRecogniser. Той работи с инстанция на хранилище от базата данни(repository) от WebHost.Repository проекта. Това е така, тъй като в единствения метод на този клас DecodeToken, е необходима проверка дали тоукънът, идващ от заявката, съществува в базата. Ако това е така, се връща уникалният номер на потребителя – userId, в противен случай се хвърля ексепшън, че тоукънът не съществува. Методът за декодиране на тоукън се използва в конструктора на сръвиса за резервации, както и в методите от потребителския сървис, които извличат или запазват информация за текущия потребител и неговия автомобил.

public Guid DecodeToken()

{

var tokenDecoder = new JwtDecoder();

var token = tokenDecoder.GetTokenFromRequest();

if (!this.\_repository.IsExistingToken(token))

{

throw new ArgumentException("Token does not exist.");

}

var userId = tokenDecoder.GetCurrentUserId();

return userId;

}

## Services

Тук се намира същинската част на бизнес логиката на приложението, а именно WCF сървисите. Чрез тях могат да се предават данни като асинхронни съобщения от една крайна точка на сървиса(endpoint) до друга. В нашата имплементация едните крайни точки се намират в Contracts папката на текущия проект, а другите – в WebClientCore проекта, папка services). За да бъдат предоставени за използване от Angular клиента е необходимо и да бъдат описани в Web.config-a на проекта, като се посочат име на сървиса, контракт, който ползва, байндинг и др. Напр.:

<service name="WebHost.Services.UserService" behaviorConfiguration="ServiceBehavior">

<endpoint contract="WebHost.Services.IUserService"

binding="webHttpBinding"

bindingConfiguration="TransportSecurity"

behaviorConfiguration="restEndpointBehaviorConfiguration" />

</service>

Използваните сървиси са два – един за операциите, които се извършват с потребителите в системата и техните автомобили, като регистриране, логване, извличане на текущ потребител, информация за автомобила и запазване на информация за автомобила – UserService.svc, и BookingService.svc, в който се намира логиката за резервациите на паркоместа. Няма да се спираме подробно на имплементацията на всеки един от методите, а само ще разгледаме начина на работа на по-комплексните от тях.

Двата най-съществени метода за UserService-а са тези за регистрация и логин. Те имат сходства, като това, че като параметър получават обект от домейн типа AuthModel, който съдържа свойствата име и парола, както и ползването на методи от UserRepository, което работи пряко с данните от базата за потребителя. Методът за регистрация проверява дали подаденият модел не е null т.е не са попълнени стойности за полетата име и парола и след това извиква метода за проверка дали потребителят вече не е регистриран в системата. Ако всичко е наред, с помощта на мапър, се напасват стойностите от домейн AuthModel-а към тези, идващи от данните, описващи базата - Data.User. Накрая потребителя се регистрира в базата отново чрез метод на хранилището за потребители – UserRepository и се извършва обратен мапинг от ресурс към домейн модел, за да може да се работи с DomainModels.User тип. За запазване на паролата в базата се използва SHA256 криптографски алгоритъм. Последно се сетва статус HttpStatusCode.OK на респонса, идващ от контекста.

public DomainModels.User Register(DomainModels.AuthModel authModel)

{

try

{

if (authModel == null)

{

throw new ArgumentNullException($"{authModel} can't be empty.");

}

this.\_userRepository.FindIfUserExists(authModel.Name);

RegisterValidator.IsValidUsername(authModel.Name);

RegisterValidator.IsValidPassword(authModel.Password);

var newUser = this.\_mapper.Map<DomainModels.AuthModel, Data.User>(authModel);

var newRegisteredUser = this.\_userRepository.RegisterUser(newUser);

var user = this.\_userRepository.GetUserById(newRegisteredUser.UserId);

var mappedUser = this.\_mapper.Map<Data.User, DomainModels.User>(user);

WebOperationContext ctx = WebOperationContext.Current;

ctx.OutgoingResponse.StatusCode = System.Net.HttpStatusCode.OK;

return mappedUser;

}

catch (Exception ex)

{

throw new WebFaultException<string>($"{ex.Message}", HttpStatusCode.BadRequest);

}

}

Логин методът работи по подобен начин, като при него вече се използва и вече създадения JwtGenerator. Гененрира се секретен тоукън за юзъра и се запазва в базата.

public string Login(DomainModels.AuthModel authModel)

{

try

{

var dataUser = this.\_mapper.Map<DomainModels.AuthModel, Data.User>(authModel);

var user = this.\_userRepository.FindIfUserExists(dataUser.Name, dataUser.Password);

var mappedUser = this.\_mapper.Map<Data.User, DomainModels.User>(user);

var jwtGenerator = new JwtGenerator();

string secureToken = jwtGenerator.GetJwt(mappedUser);

var jwt = new Data.Token

{

TokenId = Guid.NewGuid(),

UserToken = secureToken

};

this.\_userRepository.SaveJwtToken(jwt);

return jwt.UserToken;

}

catch (Exception ex)

{

throw new WebFaultException<string>($"{ex.Message}", HttpStatusCode.NotFound);

}

Останалите методи в сървиса за извличане и сетване на информация за потребителя и автомобила работят по сходен начин. Всичките използват инстанция на TokenRecogniser и декодират тоукъна, тъй като обработват информация за вече логнат потребител. След това се използват различни методи на UserRepository, спрямо търсения резултат, накрая се извършва мапинг от Data типа към DomainModels и се връща вече „напаснатия“ обект.

public DomainModels.CurrentUser GetCurrentUser()

{

var tokenRecogniser = new TokenRecogniser<Data.User>(this.\_userRepository);

var userId = tokenRecogniser.DecodeToken();

var user = this.\_userRepository.GetCurrentUser(userId, out string parkingSpaceNumber, out string bookedParkingSpaceForToday);

var mappedUser = this.\_mapper.Map<Data.User, DomainModels.CurrentUser>(user);

mappedUser.ParkingSpaceNumber = parkingSpaceNumber;

mappedUser.BookedParkingSpaceForToday = bookedParkingSpaceForToday;

return mappedUser;

}

BookingService-ът, както вече споменахме, съдържа методи за обработка на резервациите на паркоместа. Общото между тях е, че работят и инстанция на BookingRepository, който съдържа методи за пряка обработка на обектите, работещи с базата, и се извършва напасване от Data типа към DomainModel. Налични са методи за извличане на всички свободни паркоместа или на свободни места по зададен потребител, на всички резервации за даден потребител, за резервиране на свободно паркомясто, за отдаване на дадено място за резервация, за отмяна на вече дадено място за резервация и за отмяна на самата резервация.

public IEnumerable<DomainModels.Responses.GetAllFreeParkingSpacesByUserIdResponse> GetAllFreeParkingSpacesByUser()

{

var allFreeParkingSpaces = this.\_bookingRepository.GetAllFreeParkingSpacesByUser(this.userId);

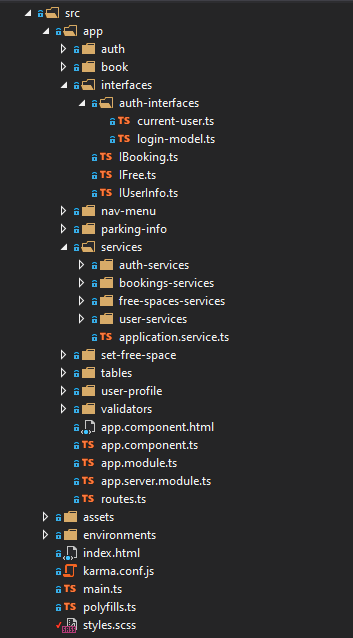
var mappedFreeParkingSpaces = this.\_mapper.Map<IEnumerable<Data.GetAllFreeParkingSpacesByUserId\_Result>, IEnumerable<DomainModels.Responses.GetAllFreeParkingSpacesByUserIdResponse>>(allFreeParkingSpaces);

return mappedFreeParkingSpaces;

}

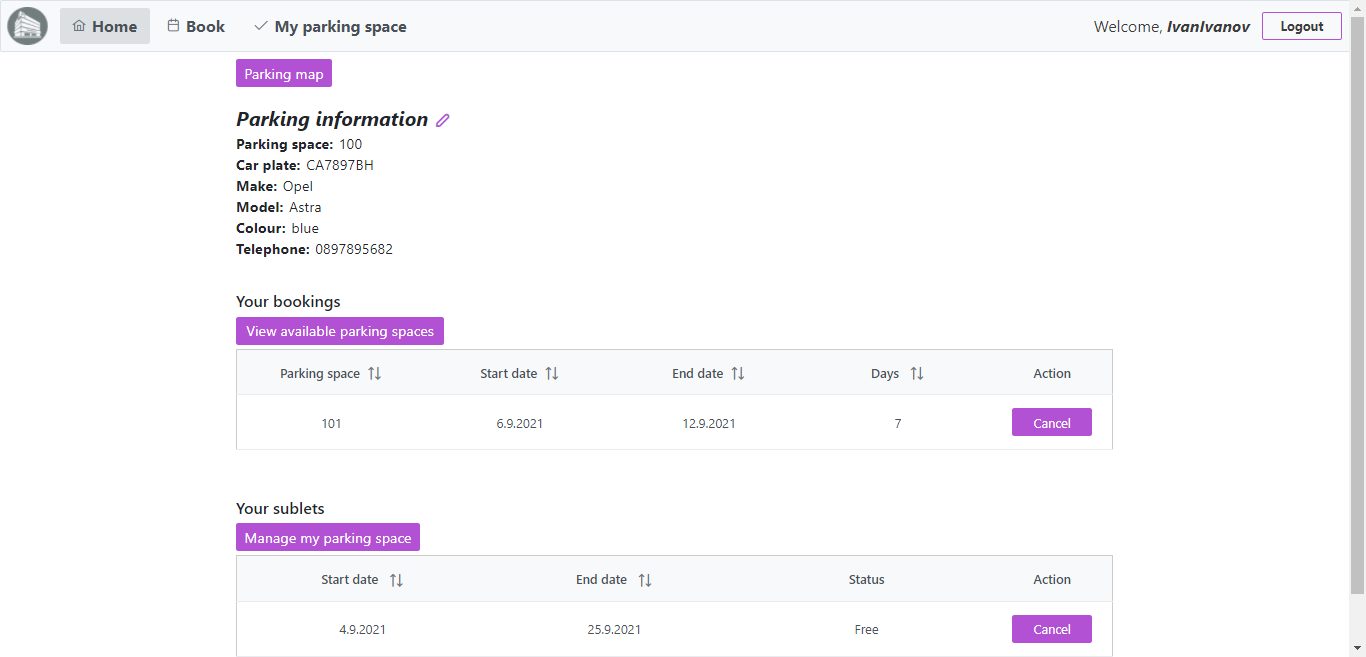
## WebClientCore project

Този проект представлява презентационния слой от 3-слойната архитектура или иначе казано слоят, който представя данните на потребителя и му дава възможност за взаимодействие с тях – добавяне, промяна, изтриване и др. Проектът има типичната Angular структура, като ние ще се спрем по-подробно на src папката, в която са поместени изходните файлове, съдържащи съществената логика за обработване и показване на данните по най-подходящия за потребителя начин.



Компонентът е основната изграждаща единица за Angular приложенията. Затова отделните функционалности по системата са разбити така, че да могат да бъдат обособени всяка в свой собствен. Като има и обобщаващи папки като auth, която съдържа в себе си два отделни компонента за оторизиращите функции в системата като регистриране на потребител и логването му в системата. Всеки компонент е съставен от HTML шаблон, който декларира какво ще бъде показано(рендерирано) на конкретната страница, за която се отнася компонента, TypeScript клас, който определя поведението на страницата, CSS селектор, който определя как компонентът ще се използва в шаблона, и CSS стилове, които ще бъдат приложени към шаблона. [666, <https://angular.io/guide/component-overview>] Ще разгледаме по-подробно един от компонентите в нашето Angular приложение, останалите функционират по много подобен начин с някои специфики, продиктувани от различното съдържание показвано в тях.

Компонентът отговарящ за потребителския профил или иначе казано началната страница, която вижда ползвателя на системата при успешен логин е user.profile.component. Визуално изглежда така:



Той е съставен от 3 файла с разширения .html, .scss и .ts. В тях се намира цялото съдържание, отговарящо за този компонент. Ще започнем с TypeScrpipt файла, който отговаря за логиката по показване на компонента, както и за извличането на данни от сървъра. В началото на файла се добавят необходими import-и на Angular компоненти, интерфейси и т.н, както и сървиси, чиито методи ще бъдат използвани за извличане на информация.

import { Component, OnInit } from '@angular/core';

import { LoginService } from '../services/auth-services/login.service';

import { FreeSpaceService } from '../services/free-spaces-services/free-space.service';

Може би тук е удобният момент да споменем, че за комуникацията между сървъра и Angular клиента, в клиентската част са създадени .ts файлове, намиращи с папката services, със същите имена като на WCF сървисите от сървърната част. Те използват инстанция на HttpClient в конструкторите си и съответно чрез get/post методи и по подадени общ базов URI, зададен в общата конфигурация за проектната среда(environment.prod.ts), и име и метод от конкретен сървис, създават ендпойнти за тези методи. Така създадените вече методи могат свободно да бъдат извиквани в клиентската част с подходящи параметри и съответно да бъде връщан отговор от сървъра, който да бъде обработван на клиента по подходящ начин.

getUserInfo(): Observable<IUserInfo> {

return this.http.get<IUserInfo>(this.baseUri + '/UserService.svc/UserInfo');

}

setUserInfo(userInfo: IUserInfo) {

return this.http.post(this.baseUri + '/UserService.svc/SetUserInfo', userInfo);

}

След импортите в .ts файла, следва указване на използвания за шаблон на страницата html файл, както и откъде да бъдат взети css стиловете за този шаблон.

@Component({

templateUrl: './user.profile.component.html',

styleUrls: ['./user.profile.component.scss']

})

После се посочва името на класа, както и дали имплементира интерфейси и/или наследява класове. В случая UserProfile класа имплементира вградения в Angular OnInit интерфейс, който представлява един от т.нар lifecycle hooks, които по принцип се грижат за състоянията в „живота“ на един компонент – от инициализирането му до унищожаването му. От този инттерфейс използваме ngOnInit метода, който позволява да задаваме наши задачи, които искаме да бъдат изпълнени при инициализиране на компонента [666, <https://angular.io/api/core/OnInit#ngOnInit>] – в случая извличане на информация за текущо логнатия потребител, всички резервации на паркоместа, направени от него, както и отдадени от него паркоместа, ако има такива. Извличането на информация, най-общо казано,става, като се извърши абониране за връщания резултат от метода и на декларирана конкретно за този клас променлива, се присвои стойността, идваща от отговора(response) от сървъра. Използват се инстанции на сървисите, зададени в конструктора на компонента.

constructor(private service: LoginService,

private freeSpaceService: FreeSpaceService) { }

ngOnInit() {

this.service.getUserData().subscribe(response => {

this.bookings = response;

});

}

HTML шаблонът също има основна роля в пълноценното изграждане на един компонент. Шаблонът на потребителския профил съдържа в себе си множество други помощни компоненти. Това е и едно от основните предимства на тези градивни елементи – основните функционалности могат да бъдат разделени на по-малки отделни самостоятелни единици, които могат да бъдат използвани на различни места, като незадължително могат да им бъдат зададени и параметри, които правят тяхната употреба още по-гъвкава. В текущия компонент например имаме вградено навигационно меню, информация за автомобила и паркомястото на потребителя, компонент за резервираните места, както и за отдадените. Интересното тук логиката на показване на последните два компонента. Спрямо върнатия резултат от сървъра, записан в променливите от .ts файла, те могат да бъдат показани, заедно със стилизацията си, или пък на тяхно място да бъде изписан текст, указващ, че няма наличие на резервирани или отдадени места. При наличие на стойност, стойността от променливата се свързва(bind), еднопосочно или двупосочно, с параметъра на компонента и става негов източник на информация. Например:

<div class="bookedSpaces">

<div \*ngIf="!bookings.length && !parkingSpaceNumber">

<p>Currently you have no booked parking spaces.</p>

</div>

<bookings-table \*ngIf="bookings.length" [bookings]="bookings"></bookings-table>

</div>

Разработен е и специален RegisterValidators клас, намиращ се в отделна папка validators, който се грижи за валидиране на полетата за потребителско име и парола в реално време, преди информацията да стигне до сървъра. Той се използва в компонента за регистриране на потребител и се грижи например потребителското име да започва с буква и да не е по-малко от 4 символа и повече от 21, паролата да съдържа 1 главна буква, 1 малка буква, 1 цифра и да бъде минимум 6 знака, както и въведената парола за потвърждение да съвпада с първоначално въведената.

## WebHost.Data и WebHost.Repository

Двата проекта заедно представляват слоя от 3-слойната архитектура, работещ пряко с данните от базата. Те включват в себе си самите данни, които представляват таблиците в системата и връзките между тях, както и методи за пряка обработка на тези данни като основните операции по добавяне, извличане, редактиране и изтриване. Данните и логиката по обработката им е разделена в два проекта, за постигане на по-голяма прегледност, лесна промяна по модела на базата и не на последно място – по-добра преизползваемост на кода. Много подобно на работата с EntityFramework Core, тук при Entitiy Framework също използваме database-first подход и ще генерираме обекти на базата на вече изградена база от данни, само че имаме възможността да се възползваме от предимствата на визуалния дизайнер и wizard-a за актуализиране на обекти, идващи от базата. Започваме като създаваме нов проект от тип Class Library и към него добавяме нов обект от тип ADO.NET Entity Data Model. При първоначално създаване на edmx модел ни се предлагат различни подходи като създаване от модел, от кода(code-first), ние ще изберем опцията EF Designer from database, тъй като се придържаме към database-first подхода. Създава се нова връзка към базата, която ще бъде използвана, където се посочва сървър, където се намира тя, детайли по логването към този сървър и т.н. На финалната стъпка са предложени обектите от базата, за които искаме да генерираме C# обекти, с които да работим в кода. Предимството тук е, че такива обекти могат да бъдат генерирани не само за таблици, но и за изгледи(views) и съхранени процедури(stored procedures). След като вече сме генерирали обектите чрез wizard-a, можем да отворим т.нар дизайнер и да ги разгледаме, заедно с връзките между тях. При промяна в схемата на базата, да речем добавяне на нова колона, процесът по актуализиране на edmx модела също става изключително лесно, като отново се използва същия wizard, в който се показват вече настъпилите промени в базата, което значително улеснява програмиста.

WebHost.Repository проекта включва в себе си методите за работа с данните от вече генерирания модел от базата. В него, подобно на WebHost проекта, също има папка с контракти – RepositoryContracts, която играе същата роля – съхранява интерфейси, които предоставят за използване методите от различните хранилища(repositories), на сървисите от слоя, отговарящ за бизнес логиката на приложението. За всеки сървис има по едно кореспондиращо репозитори, чиито методи той използва. Всяко от хранилищата от своя страна работи с инстанция на ParkingManagementSystemEntities, идваща от WebHost.Data проекта, за да има пряк достъп до модела, описващ данните от базата.

# Ръководство на потребителя

# Заключение – бъдещи тенденции

Текущата версия на платформата има всички необходими функционалности за нормална работа. Основните цели да предостави на потребителите възможност за създаване на акаунти, чрез които да се осъществява резервиране и отдаване на паркомясто, както и управление на информацията за автомобила, са имплементирани. Реализирано е и администраторско приложение за управление на всички паркоместа в системата , като се предоставени операции за търсене, добавяне, преглед, редактиране и изтриване на връзка собственик - паркомясто. Представена е една цялостна система, като са използвани различни технологии и архитектури.

Основните предимства на платформата за управление на паркоместа са, че е лесна за работа и може да си използва от различни типове потребители – било то само за наемане на паркомясто или пък за отдаване на собственото, има модерен дизайн, и може да се използва, както за настолни компютри или лаптопи, така и за таблети, и мобилни устройства. Може би най-полезното нещо за системата е нейната универсалност като сфера на приложение. Тя е максимално опростена откъм въвеждането на множество ненужна информация и е с изчистен потребителски интерфейс, който акцентира само на основните за нея функционалности. Именно поради това, платформата би могла да се използва за паркинги на бизнес сгради и различни предприятия, за паркоместа в градска среда, както и за паркинги на жилищни комплекси. Надявам се, че представената идея е била различна и иновативна, защото показва как може да облекчим процеса по управление на един паркинг, като с тази основна цел съчетаем модерен вид и адаптивен дизайн на приложението, съвременни технологии и работа на различни устройства.

Наред с показаните функционалности, съществуват идеи, които не са реализирани и остават като бъдещи планове за платформата. Една от тях е показване на интерактивна карта за наличните в паркинга места. Така при отдаване/резервиране на паркомясто от даден потребител то би се отразило в реално на картата и би направило намирането на свободно паркомясто още по-ефективно.

В бъдещи имплементации може да бъде предвиден и директен логин с потребителско име и парола от специфична за компанията или комплекса, предварително създадена мрежа с акаунти. Това би насърчило още по-масовото използване на системата от хора, притежаващи паркоместа в конкретния паркинг, да отдадат своето място за резервация, тъй като дори няма да им се налага да се регистрират, макар и процесът да е изключително олекотен.

Идеята за платформа за управление на паркоместа дава много бъдещи варианти за развитие на голяма система за управление на цялостни жилищни или бизнес комплекси. Темата е винаги актуална и може да се приеме, че има едно интересно и полезно начало. Потребителския интерфейс е съвременен и може да бъде надграден с по-нататъшни идеи за реализация.

# Използвана литература

1. Model-view-controller -<https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>
2. What is ASP.NET? - <https://subscription.packtpub.com/book/application_development/9781786463838/1/ch01lvl1sec10/what-is-asp-net>
3. Razor view engine – <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1#razor-view-engine>
4. Overview of ASP.NET Core MVC – <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-3.1>
5. Handle requests with controllers in ASP.NET Core MVC –<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions?view=aspnetcore-3.1>
6. Working with Data in ASP.NET Core Apps – <https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/work-with-data-in-asp-net-core-apps>
7. Object-relational mapping – <https://en.wikipedia.org/wiki/Object-relational_mapping>
8. Dependency injection in ASP.NET Core – <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-3.1>
9. Hypertext Transfer Protocol – <https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol>
10. Routing in ASP.NET Core – <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/routing?view=aspnetcore-3.1>
11. ASP.NET Core Middleware – <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1>
12. ControllerEndpointRouteBuilderExtensions.MapControllerRoute(IEndpointRouteBuilder, String, String, Object, Object, Object) Method – <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.builder.controllerendpointroutebuilderextensions.mapcontrollerroute?view=aspnetcore-3.1>
13. ASP.NET MVC Controller Overview (C#) – <https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions-1/controllers-and-routing/aspnet-mvc-controllers-overview-cs>
14. IFormFile Interface – <https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.http.iformfile?view=aspnetcore-3.1>
15. System.ComponentModel.DataAnnotations Namespace – <https://docs.microsoft.com/en-us/dotnet/api/system.componentmodel.dataannotations?view=netcore-3.1>
16. Introduction to Razor Pages in ASP.NET Core – <https://docs.microsoft.com/en-us/aspnet/core/razor-pages/?view=aspnetcore-3.1&tabs=visual-studio>
17. Model Binding in ASP.NET Core – <https://docs.microsoft.com/en-us/aspnet/core/mvc/models/model-binding?view=aspnetcore-3.1>
18. The Validation Tag Helpers - <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-3.1#the-validation-tag-helpers>
19. Different Types of Databases – <http://www.my-project-management-expert.com/different-types-of-databases.html>
20. Create a Job – <https://docs.microsoft.com/en-us/sql/ssms/agent/create-a-job?view=sql-server-ver15>
21. Visual Studio 2019 – <https://visualstudio.microsoft.com/vs/>
22. ASP.NET Core – <https://en.wikipedia.org/wiki/ASP.NET_Core>
23. Entity Framework Core - <https://docs.microsoft.com/en-us/ef/core/>
24. 10 Common Uses of Bootstrap - <https://www.htmlgoodies.com/html5/markup/10-common-uses-of-bootstrap.html>

# Приложение