ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА

ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ

****

Катедра „Софтуерни и интернет технологии“

**ДИПЛОМНА РАБОТА**

**Тема:**

Проектиране и реализация на Web API и десктоп приложение за паркинг система

**Изготвил:** Росен Яворов Лечев

**Специалност:** Софтуерно инженерство

**Факултетен номер:** 20651224

ТУ Варна, 2022 г. Ръководител:

/доц. Д-р инж. Виолета Божикова/

Съдържание

[**I.** **Въведение. Постановка на дипломно задание. Цели и задачи на разработката.** 4](#_Toc89607392)

[**1.1.** **Актуалност на проблема и мотивация** 4](#_Toc89607393)

[**1.2.** **Обзор на съществуващи решения** 5](#_Toc89607394)

[**1.3.** **Цели на дипломния проект** 8](#_Toc89607395)

[**1.4.** **Задачи на дипломния проект** 9](#_Toc89607396)

[**II.** **Обзор на използваните програмни средства и технологии.** 10](#_Toc89607397)

[**2.1.** **Използвани технологии за разработка на сървърно приложение** 10](#_Toc89607398)

[**2.1.1.** **.Net Core** 10](#_Toc89607399)

[**2.1.2.** **Използван мениджър на библиотеките** 11](#_Toc89607400)

[**2.1.3.** **Интегрирана среда за програмиране** 12](#_Toc89607401)

[**2.2.** **Използвани технологии за разработка на база от данни** 13](#_Toc89607402)

[**2.2.1.** **Microsoft SQL Server** 13](#_Toc89607403)

[**2.2.2.** **SQL Server Management Studio** 13](#_Toc89607404)

[**2.3.** **Език за програмиране на системата** 14](#_Toc89607405)

[**2.3.1.** **C#** 14](#_Toc89607406)

[**2.4.** **Система за контрол на версиите** 14](#_Toc89607407)

[**2.4.1.** **Git** 19](#_Toc89607408)

[**2.5.** **Система за управление на проекти** 19](#_Toc89607409)

[**2.5.1.** **Trello** 19](#_Toc89607410)

[**III.** **Проектиране на системата. Описание на алгоритмите.** 21](#_Toc89607411)

[**3.1.** **Проектиране на сървърно приложение** 21](#_Toc89607412)

[**3.1.1.** **Проектиране на база данни** 22](#_Toc89607413)

[**3.1.2.** **Проектиране на REST API** 26](#_Toc89607414)

[**IV.** **Реализация на приложението. Основни програмни модули.** 29](#_Toc89607415)

[**4.1.** **Разработка на сървърно приложение** 29](#_Toc89607416)

[**4.1.1.** **Папка Контролери (Controllers)** 29](#_Toc89607417)

[**4.1.3.** **Папка за мапингите (Mapper)** 35](#_Toc89607418)

[**4.1.4.** **Папка с миграции (Migrations)** 37](#_Toc89607419)

[**4.1.5.** **Папка с модели (Models)** 38](#_Toc89607420)

[**4.1.6.** **Папка Repository** 42](#_Toc89607421)

[**4.1.7.** **Файл Startup** 60](#_Toc89607422)

[**V.** **Тестване на системата, изводи и заключения** 62](#_Toc89607423)

[**5.1.** **Тестване на работоспособността на системата** 62](#_Toc89607424)

[**5.2.** **Изводи и заключения** 65](#_Toc89607425)

[**Използвана литература** 67](#_Toc89607426)

[**Приложения** 68](#_Toc89607427)

# **Въведение. Постановка на дипломно задание. Цели и задачи на разработката.**

## **Актуалност на проблема и мотивация**

В днешно време, най – разпостраненият начин за транспорт са автомобилите. Използват за придвижване както из градовете така и между тях. След като стигнем крайната си дестинация идва проблемът къде да паркираме автомобила си. В големите и натоварени градове места за паркиране почти няма и се налага да се обикаля и да се търси място, 20 – 30 минути, а дори и повече. Платени паркинги съществуват, но не се срещат често и цените им са високи поради факта че винаги на бариерата стои човек, който се занимава с пускането на автомобили и извършването на разплащанията. Мотивацията за проектиране на тази система е да се създаде автоматизирана алтернатива, на ръчното извършване на тази дейност, като разходите за системата, освен първоначалната инвестиция в софтуер и хардуер, ще е поддръжка на хардуера, която и в момента се извършва. Системата ще извършва всички дейности които извършва работник на бариери в момент, като могат да се довавят и допълнителни функционалности след обсъждане с клиентите. Системата е автономна, като огромно предимство е, че клиентите сами ще могат да си избират предпочитания хардуер, като бариери и камери, който лесно ще може да се интегрира в паркинг системата.

## **Обзор на съществуващи решения**

След разлгеждане на съществуващиите решения, установих че те не са много и са доста скъпи. Също така те представляват цялостен продукт, който включва в себе си както софтуера така и хардуера, като на клиентите не се предлага никакъв избор за персонализавия. Продуктът се закупува и се инсталира.

## **Цели на дипломния проект**

* Създаване на API, работещо с база данни която съдържа
  + създаване на база данни
  + създаване на таблици и връзки в базата данни
  + създаване на платформено независимо, уеб приложение
  + изграждане на архитектурата
  + обработване и връщане на информация във вид на json

## **Задачи на дипломния проект**

* Избиране на система и разработване на база данни
  + проектиране на архитектурата на базата
  + използване на подходящи constraint[[1]](#footnote-1)-и, ключове, индекси и връзки
* Избиране на технологии и проектиране на приложението
  + създаване на архитектурата на приложението
  + създаване на модели за работа с таблиците от базата
  + създаване на методи за извършване на CRUD [[2]](#footnote-2)операции с базата
  + създаване на алгоритми за добавяне, извличане, изтриване и подновяване на информация, включваща транзакции където е необходимо
  + създаване на сървиси за работа с моделите
* създаване на контролери, обработващи идващите заявки

# **Обзор на използваните програмни средства и технологии.**

## **Използвани технологии за разработка на сървърно приложение**

### **.Net Core**

.Net Core е безплатен, open-source софтуерен framework за Windows, Linux и macOS операционни системи за процесори x64, x86, ARM32 и ARM64 като могат да се използват няколко програмни езика. То е платформено независим наследник на .Net Framework. .Net Core осигурява висока производителност и еднакво поведение на различните операционни системи и архитектури. Езиците които могат да се използват за програмиране са C#, Visual Basic и F#. Те могат да се използват с предпочитаната от всеки среда както и Visual Studio и Visual Studio Code. .Net Core може да използва допълнителни framework-и за разработването на всякакъв вид приложения:

* облачни приложения с ASP.NET Core
* мобилни приложения с Xamarin
* IoT приложения със System.Device.GPIO
* Windows приложения с WPF и Windows Forms
* машинно обучени с ML.NET

Също са включени api-та които да доставят допълнителна функционалност като примитиви, колекции, типове като HttpClient, DataSetи типове с висока производителност като вектори.

.Net Core се поддържа от Microsoft, обновява се редовно и използва лицензите MIT и Apache 2.

### **Използван мениджър на библиотеките**

За .Net механизмът за споделяне на код е NuGet, който определя начина по който пакетите се създават, хостват и използват и предлага инструментите за всяка от тези стъпки. NuGet пакетите за архив с разширение .nupkg които съдържат компилиран код (DLL-и), други файлове свързани с кода и описателна част която включва информация като версията на пакета. Разработчиците които искат да споделят код създават пакети и ги качват на публичен или частен хост, а разработчиците които искат да ги използват ги получават от хостовете, добавят ги в проектите си и ги използват като NuGet се справя с подробностите в процеса.

Използвани пакети:

* AutoMapper – това е малка библиотека създадена да реши заблуждаващо сложен проблем, а именно да се отървем от код който се използва за мапване на един обект към друг. Веднъж зададени правилата за мапване на дадени обекти, те могат да се използват бързо и лесно.
* RestSharp – това е библиотека за изпращане на HTTP заявки за .NET. Включва автоматична сериализация и десериализация, детекция на типа на заявката и отговора и различни допълнителни функционалности като автентикация.
* AForge.NET – това е библиотека с отворен код, предназначена за използване от среда работеща върху .NET. Използша се от програмисти в сферите на изкуствения интелект, компютърно зрение и различна обработка на снимки.
* Entity Framework Core – това е лека, open source, cross-platform версия на популярния Entity Framework. EF Core е object-relational mapper (O/RM) и като такъв се използва за намаляване на разликата между релационните и обектно ориентираните модели, като позволява на разработчиците да взаимодействат с информацията съхранена в базите данни като използват прости обекти.

EF Core имплементира много от популярните O/RM свойства:

* Мапване на POCO (Plain old CLR object) класове
* Автоматично следене на промените
* Използване на Unit of Work патърн
* Eager, lazy и excplicit loading
* Използване на заявки чрез използване на LINQ (Language INtegrated Query)
* Богати способности за мапване които включват:
* връзки едно към едно, едно към много и много към много
* наследяване на таблици
* сложни типове
* съхранени процедури
* Code First подход за създаване на базата данни от готовите и свързани класове
* Database First подход за създаване на класовете от готова база данни

### **Интегрирана среда за програмиране**

Visual Studio е среда за разработване от Microsoft. Използва се за създаване на програми както и уеб сайтове, уеб приложения, уеб сървиси и мобилни приложения. Съществува Community версия която е безплатна. Средата съдържа много вградени инструменти, но възможността за използване на плъгини увеличава възможностите на средата дори още.

* Visual Studio съдържа компонент за допълване на кода наречен IntelliSense както и рефакториране на код.
* Включеният дебъгър работи като дебъгър на кода така и като дебъгър на машината.
* Други вградени инструменти включват дизайнер за създаване на GUI приложения, уеб дизайнер, клас дизайнер, дизайнер за диаграма на база данни. Позволява добавянето на плъгини като увеличават функционалността на всяко ниво като добавяне на връзка към система за контрол на версиите като Git.

## **Използвани технологии за разработка на база от данни**

### **Microsoft SQL Server**

Microsoft SQL Server е система за управление на релационни бази данни, създадена от Microsoft. Основната му функционалност е да съхранява и връща информация, поискана от други софтуерни приложения, които могат да работят на същата машина или на други машини в интернет. Microsoft поддържа различни версии с различни характеристики, които да удовлетворяват различните потребителски групи. Базата данни е релационна като се състои от таблици с колони и редове. Поддържат се различни типове на данните като включват примитивни типове като Integer, Float, Decimal, Char, Varchar, Binary, Text и други.

### **SQL Server Management Studio**

SQL Server Management Studio е софтуерно приложение което се използва за конфигуриране ,менажиране и администриране на всички компоненти в един MS SQL Server. Инструментът съдържа както скриптови редактори така и графични инструменти за работа с обектите и свойствата на сървъра. Съществува Express версия на продукта който може да бъде изтеглен и използвам безплатно. Новите версии са обратно съвместими и могат да се използват за връзка и управление на стари версии на MS SQL.

## **Език за програмиране на системата**

### **C#**

C# е обектно ориентиран език за програмиране създаден от Microsoft. Той цели да комбинира мощта на езика C++ с леснотата на езика Visual Basic. C# е базиран на C++ и съдържа функционалности подобни на тези на Java. Езикът е предназначен за работа с платформата .Net.

Няколко причини които правят C# широко използван в професионалната среда език:

* обектно ориентиран
* компонентно ориентиран
* лесен за научаване
* структуриран език
* пишат се ефективни програми
* може да бъде компилиран на различни машини
* част е от .Net Framework
* LINQ [[3]](#footnote-3)и ламбда[[4]](#footnote-4) изрази
* лесно многонишково програмиране
* интеграция с Windows

## **Система за контрол на версиите**

В компютърното софтуерно инженерство, контрол на версиите е всяка практика, която следи и предлага контрол върху промените в изходния код. Понякога, софтуерните разработчици използват Система за Контрол на Версиите (VCS, на английски: Version Control System) както за кода, така и за да поддържат документацията и конфигурационните файлове по даден проект. Най-простата употреба и обяснение на VCS е, че потребителят би могъл лесно да се върне към предишна работеща версия на документ, по който работи в случай, че нещо се обърка.

VCS са станали важна част от този процес поради следните предимства, които предлагат:

* Сътрудничество – група от разработчици, дори и да се намират на различни географски местоположения, могат да работят по един и същ набор от документи или файлове без да пречат един на друг.
* Архивиране и Възстановяване – Файловете се поставят в хранилището на проекта след редактирането им и след това можете да се върнете към всеки един момент от развитието на тези файлове.
* Синхронизация – Дава възможност на членовете на екипа да споделят документи и да могат да разполагат с най-актуалните им версии.
* Отмяна – В случай, че нещо сериозно се обърка, системата дава възможност да се върнем към последната „ работеща“ версия.
* Следене на Промените – Всяка промяна по файловете в хранилището се асоциира с пореден номер и се запазва в история на промените. Когато файла бива променян, може да се добави и кратко обяснение за промените, тяхната необходимост и проблемите, които те решават, което обяснение се запазва заедно с поредния номер в историята на промените на VCS, а не в самия файл. Това прави лесно проследяването на развитието на документа през времето.
* Управление на промените – промените могат да бъдат инспектирани, обсъждани, одобрявани или отхвърляни по необходимост, като при желание състоянието на документа или файла може да се върне такова каквото е било на определен етап от развитието си.
* Следене на Принадлежността – VCS запазва името на редактиращия към всяка промяна направена от него.
* Разклоняване и Обединяване – Възможност за разклоняване на копие от кода в отделна локация и редактирането му в изолация, като при това промените му се следят отделно. По-късно е възможно сливането на кода обратно с друго разклонение.
* Непрекъсната Интеграция – възможността, която предлагат системите за контрол на версиите, развитието на даден софтуерен продукт да се раздели на малки части (промени), дава възможността да се провеждат операции, тестове и проверки върху всяка последователна промяна, в непрекъснат стил

Повечето системи за контрол на версиите включват следните концепции, въпреки че наименованията могат да варират:

* Основна структура
* Repository (Хранилище) – базата данни съхраняваща файловете
* Server (Сървър) – компютърът, на който е разположена базата данни
* Client (Клиент) – компютърът свързващ се с хранилището
* Working Copy (Работно копие) – локалното копие на файловете, където се правят промените
* Trunk (Стъбло) – Основната локация в хранилището, от която започва развитието на проекта. Главната линия, посока за разработка. Можем да мислим за кода почти като горски път, разклоняващ се, и сливащ се обратно, или продължаващ разклонен.
* Branch (Разклонение) – Копие на кода от даден момент на развитие на проекта, което може да се развива независимо, или да се обедини обратно.
* Основни функции
* Add (Добавяне) – поставяне на файл в хранилището за първи път. Т.е. започване на проследяването му чрез VCS.
* Revision (Ревизия) – поредната версията на файла(напр. v1, v2, v2.3 и т.н.)
* Head (Хед) – последната версия на проекта в дадено разклонение в хранилището.
* Check out (Изтегляне) – Изтеглянето на файл от хранилището. Тази функция има 2 основни задачи. На сървъра – VCS запомня, че файла е бил изтеглен за редактиране и при нужда уведомява останалите членове на екипа. На клиента – подготвя работна версия на файла за редактиране давайки му съответните разрешения. Тази функция е начин да заявите вашите намерения пред останалите, така че те могат да се съобразят и да избегнат паралелното редактиране на същия файл, което би довело до допълнителни усложнения (конфликт) при опит за качване. Някои VCS предлагат автоматичното заключване на файла, ако той е бил изтеглен, така че да бъде невъзможно паралелното му изтегляне от друг сътрудник.
* Check in, Commit (Качване) – Качването на файл в хранилището(ако е бил променен). Файла приема нов номер на ревизия и вече не е маркиран като изтеглен, така че членовете на екипа могат да изтеглят последната му версия. Някои VCS заключват за редактиране локалното копие на файла след качване.
* Коментар за Качването – кратък коментар описващ промените по файла. Добра практика е промените да се описват оптимално, така че в историята на промените да бъдат ясно упоменати и разбираеми причините и корективите, които са били предприети. Това може да бъде изключително полезно след време, когато е нужно да се разбере подобна информация, а спецификата е забравена.
* Changelog/History (История на промените) – Списък с всички промени направени по даден файл от началото на неговото добавяне в VCS
* Update/Sync (Синхронизиране) – Синхронизиране на локалните файлове с последните техни версии от хранилището
* Revert (Връщане) – Отхвърляне на локалните промени по файла и презареждането на последната му версия от хранилището.
* Разширени Функции
* Branch(Разклонение) – създаване на отделно копие на файл или папка за независимо ползване (тестване, дебъгване и т.н.).
* Diff/Change/Delta (Разлики) – откриване на различията между два файла. Полезно за визуализирането на настъпили промени между определени ревизии.
* Merge(Съединяване, сливане) – Прилагане на промените от един документ към друг, който да бъде актуализиран. Например могат да се обединят елементи от едно разклонение с друго.
* Conflict (Конфликт) – когато чакащи промени към един файл в хранилището си противоречат (не могат да се приложат и двете промени)
* Resolve (Решение) – разрешаване на проблема с противоречащи си чакащи промени и качването на коректен вариант
* Locking (Заключване) – установяване на контрол върху файл, така че никой не може да го редактира докато не бъде отключен. Някои VCS използват това за избягване на конфликти
* Breaking the lock (Силово Отключване) – силово отключване на файла, така че той да може да бъде редактиран. Налага се когато някой заключи файла и се отправи на почивка например.
* Tag/Label (Тагване) – разклоняване, обикновено съдържащо стабилна версия на проекта готова за пускане, при което се задава име или пореден номер на версията. Прави по-удобно проследяването на развитието на проекта и възможността за бързо връщане към стабилно негово състояние.

### **Git**

Git (произнася се „гит“) е децентрализирана система за контрол на версиите на файлове. Създадена е от Линус Торвалдс за управление на разработката на Linux. Поради нуждата да се контролира огромната база от код на Linux ядрото, основна цел при разработката на Git е била бързината. Всяка локална Git директория е хранилище с пълна история и възможности за следене на версиите. Това прави Git независим от мрежови връзки към централен сървър. Git е свободен софтуер и се разпространява под GPL лиценз версия 2. Текстовият интерфейс на програмата е преведен на български. Разработката на Git започва след като много от разработчиците на Линукс ядрото преустановяват използването на BitKeeper – тогавашният избор за система за контрол на версиите. В търсенето си на алтернатива на BitKeeper, Торвалдс не намира свободна система, която да посреща изискванията му. Той решава непосредствено след завършване на 2.6.12-rc2 ядрото да се отдаде изцяло на разработването на Git. Изборът на името Git не е много ясен и има различни версии защо точно така Торвалдс е избрал да нарече проекта си. В жаргон на Английски git означава „неприятен човек“. Линус се шегува, че понеже е егоцентричен обича да кръщава проектите на себе си. Торвалдс преотстъпва разработването на 26 юли 2005 на Джунио Хамано.

## **Система за управление на проекти**

### **Trello**

Trello е безплатно уеб-базирано приложение за управление на проекти първоначално създадено от Fog Creek Software през 2011 г. Използва фримиум бизнес модел, както и финансиране от други продукти на Fog Creek Software. Основната услуга се предлага безплатно, а платената услуга за бизнес класа стартира през 2013 г.

Trello използва Канбан парадигмата за управление на проекти, първоначално популяризирана от Тойота през 80-те години на 20 век за управление на доставките. Проектите са представени от табла, които съдържат списъци (отговарящи на списъците със задачи). Списъците се състоят от карти (отговарящи на задачите).

Картите би трябвало да напредват от един списък към друг (чрез влачене и поставяне), например отразявайки потока на проект от идея към реализиране. Потребителите могат да бъдат назначавани към карти. Потребителите и таблата могат да бъдат групирани в организации.

Trello поддържа iPhone, Android и Windows 8 мобилни платформи, но неговият уебсайт е създаден достъпен за повечето от мобилните уеб браузъри. Картите приемат коментари, прикачвания, гласуване, крайни срокове и списъци за отбелязване. Trello има API. Потребителите могат да организират проектите чрез използване на табла, списъци и карти, които формират направена специално за проекта йерархия, която благоприятства ефективното управление на проекти и задачи.

# **Проектиране на системата. Описание на алгоритмите.**

## **Проектиране на сървърно приложение**

**Проектиране на софтуерен дизайн на сървърното приложение**

Сървърното приложение е проектирано по начин, който цялостната му поддръжка. Архитектурата организирана в слоеве което позволява лесни промени и добавяне на нови функционалности в бъдеще. Ключът към тази архитектура е използването на интерфейси които позволяват алтернативни имплементации. Използването на интерфейси гарантира, че методите които са заложени там ще присъстват в съответния клас който имплементира интерфейс.

Приложението се състои от следните модули:

* **База данни –** използва се за съхранение на всички нужни данни, както и при обръщане към базата те да могат да бъдат извличани. Важен аспект в проектирането на базата е съхранението на списък с позиции на служителите, които ще бъдат използвани за определяне на права за достъп.
* **REST Web API –** съдържа цялостната функционалност за работа с базата. Съдържа бизнес логика за обработване на данните.

### **Проектиране на база данни**

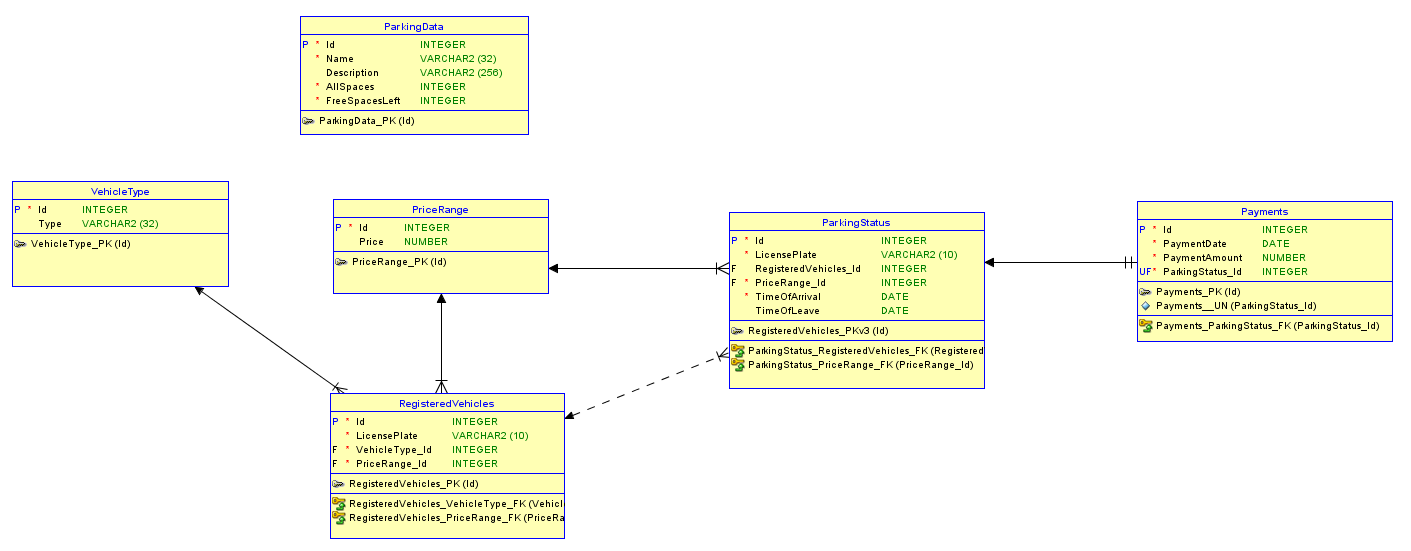
**Определяне на таблиците в базата данни**

В базата се съхранява цялата нужна информация, която е разделена на таблици. Използва се релационна база данни, като е важно да се зададат правилни типове на колоните и да се направят правилни връзки между таблиците.

Таблиците са:

* Таблица за съхранение на тарифите
* Таблица за съхранение на регистрираните автомобили
* Таблица за съхранение на типовете автомобили
* Таблица за съхранение на цялостната история на паркинга
* Таблица за съхранение на информация плащанията
* Таблица съдържаща основна информация за паркинга

Чрез диаграма ясно се представя структурата на базата данни, таблиците, техните колони и връзките между таблиците. За създаването на диаграмата е използван Oracle SQL Developer Data Modeler.



Фиг. 1 Диаграма на базата данни

Връзките между таблиците са както следва:

* Връзката между таблици VehicleType и RegisteredVehicles е едно към много (one to many), защото много автомобили могат да бъдат от един тип.
* Връзката между таблици PriceRange и RegisteredVehicles е едно към много (one to many), защото много автомобили могат да използват една тарифа.
* Връзката между таблици PriceRange и ParkingStatus е едно към много (one to many), защото много статуси могат да използват една тариифа.
* Връзката между таблици RegisteredVehicles и ParkingStatus е едно към много (one to many), но не е задължителен ключ, защото автомобил може да влиза в паркинга повече от веднъж.
* Връзката между таблици Payments и ParkingStatus е едно към едно (one to one), защото към едно излизане на автомобил може да има едно плащане.

**Структурно описание на таблиците, съставляващи базата**

**Таблица „Типове Автомобили“ (VehicleTypes)**

Таблицата съдържа основна информация за типовете автомобили и има следния формат:



Фиг. 2 Съдържание на ред от таблица VehicleTypes

Всеки ред съдържа:

* Уникален идентификатор (id)
* Тип на автомобил

**Таблица „Тарифи“ (PriceRanges)**

Таблицата съдържа информация за тарифата и има следния формат:



Фиг. 3 Съдържание на ред от таблица PriceRanges

Всеки ред съдържа:

* Уникален идентификатор (id)
* Тарифа

**Таблица „Регистритани Автомобили“ (RegisteredVehicles)**

Таблицата съдържа информация за регистрираните автомобили и има формат:



Фиг. 4 Съдържание на ред от таблица RegisteredVehicles

Всеки ред съдържа:

* Уникален идентификатор (id)
* Номер на автомобила
* Външен ключ към тарифа
* Външен ключ към тип на автомобила

**Таблица „Паркинг статус“ (ParkingStatus)**

Съдържа информация за за влизане и излизане от паркинга и има следния формат:



Фиг. 5 Съдържание на ред от таблица ParkingStatus

Всеки ред съдържа:

* Уникален идентификатор (id)
* Дата и час на влизане
* Дата и час на излизане
* Външен ключ към регистриран автомобил(ако е такъв)
* Външен ключ към тарифа
* Номер на автомобила

**Таблица „Плщания“ (Payments)**

Таблицата съхранява информация за разплащанията и има следния формат:



Фиг. 6 Съдържание на ред от таблица Payments

Всеки ред съдържа:

* Уникален идентификатор (id)
* Дата и час на плащане
* Заплатена сума
* Външен ключ към данните за паркине на автомобила

### **Проектиране на REST API**

**REST** съкратено от Representational State Transfer е стил софтуерна архитектура за реализация на уеб услуги. Това е концепция за заделяне на ресурс, който се променя въз основа на взаимодействието между клиент и сървър. Клиентът прави заявка, сървърът я обработва и връща отговор, съответстващ на заявката. Архитектурният стил на REST прилага 6 условия и когато дадено приложение покрива тези условия, то може да се нарече RESTful.

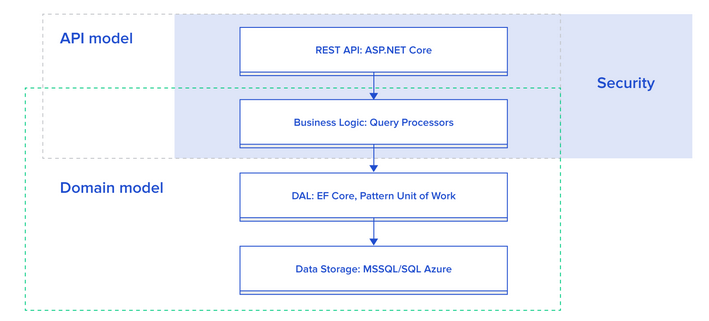
* Клиент – сървър архитектура – клиентът е front – end, а сървърът е back – end. Двете са независими един от друг.
* Stateless (без запазване на състоянието) – да не се запазват статуси на сесиите. Заявките от клиента съдържат цялата нужна информация за обработване на заявката. Статуси на сесиите могат да се пазаят единствено при клиента.
* Кеширане – клиентът има право да запазва информация, получена като отговор от сървъра с цел да се намалят ненужните взаимодействия между клиента и сървъра, като по този начин се подобрява бързината.
* Многослойна система – съществуват сървъри посредници които подобряват производителността като увеличават капацитета за обработване на заявки. Също така допринасят за повишаването на сигурността.
* Код при поискване (незадължително) – сървърът може временно да разреши изпълнението на скриптове директно при клиента.
* Единен интерфейс – единния интерфейс разделя и опростява архитектурата.

**API** (съкратено от Application Programming Interface) е набор от правила, които позволяват на една част от софтуерна програма да комуникира с друга част.

**RESTful API** е уеб приложение, което базирано на принципите на REST и HTTP. Често се използва с медийния тип JSON, но работи и с типове като XML и текст. Базовите операции, които поддържа, са:

* GET – предоставя достъп за четене до ресурс
* POST – използван за създаване на нов ресурс
* DELETE – използван за премахване на ресурс
* PUT – използван за модифициране на съществуващ ресурс или за създаване на нов ресурс

**Проектиране на софтуерна архитектура на REST API**

****

Фиг. 7 Концептуален модел на проектираното REST API

На диаграмата е показано, че системата се състои от четири слоя:

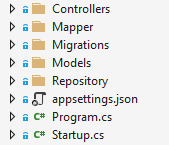
* **База данни** – тук се съхраняват данните
* **DAL (Database Access Layer)** –слой за достъпване на данните от базата. Използва се Unit Of Work pattern, Repository pattern както и Entity Framework Core с code first и патърн с миграции.
* **Бизнес логика** – за да се капсулира бизнес логиката се използват сървиси които управляват заявките. Цялата бизнес логика за всяко ентити се съдържа в сървис. Може да съдържа CRUD операции, както и всеки друг нужен метод за работа.
* **REST API** – интерфейсът чрез който клиентите могат да работят с API-то.

# **Реализация на приложението. Основни програмни модули.**

## **Разработка на сървърно приложение**

Файловете в сървърното приложение са разделени и организирани в отделни папки, според тяхната принадлежност. Чрез групирането на класовете се постига добра организация на кода.

Организацията на приложението е следната:



Фиг. 10 Цялостно съдържание на сървърното приложението

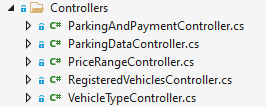
### **Папка Контролери (Controllers)**

Контролерите са класове, които се създават в приложението. Намират се в папка Controllers. Всеки един клас, който е от този тип, трябва да има име завършващо с наставка „Controller“. Контролерите обработват постъпващите заявки, въведени от потребителя и изпълняват подходящата логика за изпълнение на приложението. Класът контролер е отговорен за следните етапи на обработка:

* Намиране и извикване на най-подходящия метод за действие (action method) и валидиране, че може да бъде извикан.
* Взимането на стойности, които да се използват като аргументи в метода за действие.
* Отстраняване на всички грешки, които могат да възникнат по време на изпълнението метода за действие.

Взаимодействията с потребителя са организирани чрез контролери и методи за действие. Контролерът определя метода за действие и може да включва толкова методи за действие, колкото са необходими. Методи за действие обикновено имат функции, които са пряко свързани с взаимодействието с потребителя. Примери за взаимодействие с потребителите са въвеждане на URL адрес в браузъра, кликване върху линк, и подаването на формуляр. Всяко едно от тези потребителски взаимодействия изпраща заявка към сървъра. Във всеки един от тези случаи, URL адреса от заявката съдържа информация, която се използва за да включи метод за действие.

Структура на папка Controllers:



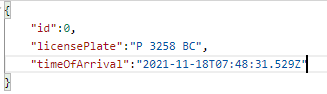
Фиг. 11 Съдържание на папка Controllers

#### **Контролер за паркиране и плащане (ParkingAndPayment)**

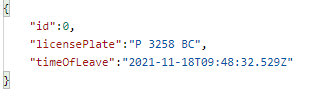
Класът ParkingAndPaymentController съдържа действията които се използват за обработване на информация за паркиране и плащане.

Действията са:

* EnterParking([FromBody] ParkingStatusModel parkingStatusModel) – това действие се използва за вкарване на автомобил в системата. Методът на заявката е POST и се извиква от адрес /ParkingAndPayment/EnterParking. Данните нужни за логване се намират в body частта на заявката.



* GetCalculatedPaymentForLicensePlate([FromBody] ParkingStatusModel parkingStatusModel) – това действие се използва за пресмятане на дължимата сума за плащане. Методът на заявката е POST и се извиква от адрес /ParkingAndPayment/GetCalculatedPaymentForLicensePlate. Нужните данни се намират в body частта на заявката.



* VehicleLeave([FromBody] ParkingStatusModel parkingStatusModel) – това действие се използва за записване че автомобилът е напуснал паркинга. Методът на заявката е POST и се извиква от адрес /ParkingAndPayment/VehicleLeave.



* GetAllParkingStatuses() – това действие се използва за извличане на паркинг историята. Методът на заявката е GET и се извиква от адрес /ParkingAndPayment/ParkingStatuses.
* GetAllPyments() – действието се използва за извличане на всички плащания. Методът на заявката е GET и се извиква от адрес /ParkingAndPayment/GetAllPayments.
* GetPaymentsByLicensePlate([FromBody] string licensePlate) – използва се за извличане на всички плащанич по номер на автомобил. Методът на заявката е POST и се извиква от адрес /ParkingAndPayment/GetPaymentsByLicensePlate като нужните данни за изпълнение на заявката се намират в body частта на заявката и са:



#### **Контролер за основната информация за паркинг (ParkingDataController)**

Класът ParkingDataController съдържа действия, които се използват за извличане и обновяване на информация.

Действията са:

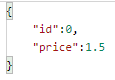
* GetParkingData() – действието се използва за извличане и връщане на основната информация на паркинга. Методът на заявката е GET и се извиква от адрес /ParkingData/GetParkingData.
* UpdateParkingData([FromBody] ParkingDataModel parkingDataModel) – използва се за обновяване на основната информация на паркинг. Методът на заявката е PUT и се извиква от адрес /ParkingData/UpdateParkingData.

#### **Контролер за тарифите (PriceRangeController)**

Класът PriceRangeController съдържа действия които се използват за работа с тарифите.

Действията са:

* GetAllPrices() – действието се използва за извличане на всички тарифи. Методът на заявката е GET и се извиква от адрес /PriceRange/GetAllPrices.
* AddPriceRange([FromBody] PriceRangeModel priceRangeModel) – действието се използва да добавяне на нова тарифа. Методът на заявката е POST и се извиква от адрес /PriceRange/AddPriceRange.



#### **Контролер за регистрираните автомобили (RegisteredVehiclesController)**

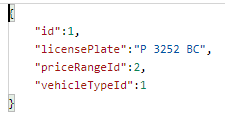
Класът RegisteredVehiclesController съдържа действия които се използват за работа с данните за регистрирани автомобили.

Действията са:

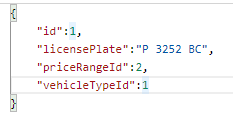
* GetAll() – действието се използва за извличане на всички регистрирани автомобили. Методът на заявката е GET и се извиква от адрес /RegisteredVehicles/GetAll.
* RegisterVehicle([FromBody] RegisteredVehicleModel registeredVehicleModel) – дейтвияето се използва за регистриране на автомобил. Методът на заявката е POST и се извиква от адрес /RegisteredVehicles/RegisterVehicle като нужните данни за изпълнение на заявката се намират в body частта на заявката и са:



* UpdateRegisteredVehicleInformation([FromBody] RegisteredVehicleModel registeredVehicleModel) – действието се използва за обновяване на информацията на регистриран автомобил. Методът на заявката е PUT и се извиква от адрес /RegisteredVehicles/UpdateRegisteredVehicleInformation като нужните данни за изпълнение на заявката се намират в body частта на заявката и са:



* RemoveRegisteredVehicle([FromBody] RegisteredVehicleModel registeredVehicleModel) – действието се изплзва за премахване на регистриран автомобил. Методът на заявката е DELETE и се извиква от адрес /RegisteredVehicles/RemoveRegisteredVehicle като нужните данни за изпълнение на заявката се намират в body частта на заявката и са:



#### **Контролер за типове регистрирани автомобили(VehicleTypeController)**

Класът VehicleTypeController съдържа действия които се използват за работа типовете регистрирани автомобили.

Действията са:

* GetAllVehicleTypes() – действието се използва за извличане на всички типове регистрирани автомобили. Методът на заявката е GET и се извиква от адрес /VehicleType/GetAllVehicleTypes.
* AddVehicleType([FromBody] VehicleTypeModel vehicleTypeModel) – действието се използва за добавяне на нов тип автомобил. Методът на заявката е POST и се извиква от адрес /VehicleType/AddVehicleType. като нужните данни за изпълнение на заявката се намират в body частта на заявката и са:

### **Папка за мапингите (Mapper)**

В тази папка се съдържа файл, в който са описани правилата по които AutoMapper ще преобразува един обект в друг.



Фиг. 13 Съдържание на папка Mapper

Класът MappingConfig съдържа методи които указват как точно да се осъществи преобразуването между обектите. AutoMapper следва конвенция, при която ще се мапнат променливите между двата класа които са с едни и същи имена, без допълнителни указания. Това обаче не пречи да се задават променливи с различни имена, защото AutoMapper позволява и да се зададат точно кои променливи към кои да се мапнат.

Класът съдържа следните методи:

* PriceRangeMap() – метод който указва как да се мапнат данните от обект PriceRange към обект PriceRangeModel
* PriceRangeModelMap() – метод който оказва как да се мапнат данните от обект PriceRangeModel към обект PriceRange
* VehicleTypeMap() – метод който оказва как да се мапнат данните от обект VehicleType към обект VehicleTypeModel
* VehicleTypeModelMap() – метод който оказва как да се мапнат данните от обект VehicleTypeModel към обект VehicleType
* ParkingDataMap() – метод който оказва как да се мапнат данните от обект ParkingData към обект ParkingDataModel
* ParkingDataModelMap() – метод който оказва как да се мапнат данните от обект ParkingDataModel към обект ParkingData
* RegisteredVehicleMap() – метод който оказва как да се мапнат данните от обект RegisteredVehicle към обект RegisteredVehicleModel
* RegisteredVehicleModelMap() – метод който оказва как да се мапнат данните от обект RegisteredVehicleModel към обект RegisteredVehicle
* ParkingStatusMap() – метод който оказва как да се мапнат данните от обект ParkingStatus към обект ParkingStatusModel
* ParkingStatusModelMap() - метод който оказва как да се мапнат данните от обект ParkingStatusModel към обект ParkingStatus
* PaymentMap() - метод който оказва как да се мапнат данните от обект Payment към обект PaymentModel
* PaymentModelMap() - метод който оказва как да се мапнат данните от обект PaymentModel към обект Payment

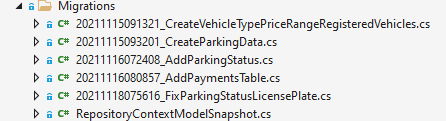
### **Папка с миграции (Migrations)**

Entity Framework (EF) е Object-relation mapper. Използва се за създаване, ъпдейтване, изтриване, добавяне на таблици, редове в таблици, връзки между таблици. Като цяло EF предлага на разработчиците функционалности, които позволяват да се разработва базата данни по време на разработване на приложението.

Съществуват три подхода за създаване на таблиците в базата и обектите в приложението:

* Code first подход – това е използваният подход при разработка. Първо се създават класовете, които ще отговарят за таблиците в базата данни, заедно с техните променливи които ще са колоните в таблицата. След като се създадат класовете, EF позволява да се зададат и връзките между таблиците, поддържат се и трите вида връзки – one to one, one to many, many to many. Ако класовете в приложението са готови, този подход позволява лесно да се създаде базата данни от тях. За самата работа с базата се използват миграции. Предимството на този подход е, че при промяна на базата, тя не се дропва и създава отново, при което се губят данните, а се ъпдейтва. Основните правила за мигриране на промените в базата са:
  + Enable Migrations – с тази команда се разрешават миграциите
  + Add Migration – с тази команда се създава миграция по последните промени в кода. Може да бъде зададено специфично има на миграцията което да показва какви са промените.
  + Update Database – при тази команда се оказва името на миграцията, по която трябва да се направят промени в базата.

При този подход могат да се използват така наречените сийдове (Seed), за да се заредят примерни данни в новосъздадените таблици.



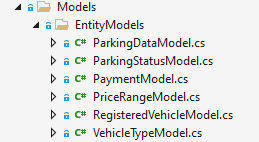
Фиг. 14 Съдържание на папка Migrations

Всеки файл съдържа точни указания, как да бъде създадена или ъпдейтната таблица, връзка или колона, или тяхното изтриване

* Database First подход – този подход може да се използва ако базата данни е вече проектирана и готова. При този подход EF създава класовете, съответстващи на таблиците, директно от базата. Този подход позволява при промяна в базата данни да се обновят и класовете в приложението.
* Model First подход – при този подход се използва визуален дизайнер за създаването на таблиците и връзките, след което на базата на получената схема може да се генерира код за създаването на базата и EF да създаде базата данни.

### **Папка с модели (Models)**

Тази папка съдържа модели на класовете които съответстват на таблиците от базата, и са готови за използване в приложението.



Фиг. 15 Съдържание на папка Models

#### **Файл ParkingDataModel**

Този файл съдържа променливи за:

* Id от тип int за уникален идентификатор
* Name от тип string
* Description от тип string
* AllSpaces от тип int
* FreeSpacesLeft от тип int

#### **Файл ParkingStatusModel**

Този файл съдържа променливи за:

* Id от тип int за уникален идентификатор
* LicensePlate от тип string
* TimeOfArrival от тип DateTime
* TimeOfLeave от тип DateTime?
* RegisteredVehicleId, RegisteredVehicleModel от типове int? и RegisteredVehicleModel за релация към другата таблица
* PriceRangeId, PrieRangeModelId от типове int и PrieRangeModel за релация към другата таблица
* Payment от тип PaymeentModel за релация към другата таблица

#### **Файл PaymentModel**

Този файл съдържа променливи за:

* Id от тип int за уникален идентификатор
* PaymentDate от тип DateTime
* PaymentAmount от тип double
* ParkingStatusId от тип int
* ParkingStatusModel от тип ParkingStatusModel

#### **Файл PriceRangeModel**

Този файл съдържа променливи за:

* Id от тип int за уникален идентификатор
* Price от тип double

#### **Файл RegisteredVehicleModel**

Този файл съдържа променливи за:

* Id от тип int за уникален идентификатор
* LicensePlate от тип string
* PriceRangeId от тип int
* PriceRange от тип PriceRangeModel
* VehicleTypeId от тип int
* VehicleType от тип VehicleTypeModel

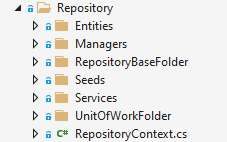
#### **Файл VehicleTypeModel**

Този файл съдържа променливи за:

* Id от тип int за уникален идентификатор
* Type от тип string

### **Папка Repository**

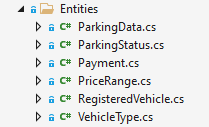
Тази папка съдържа всички нужни компоненти за работа с базата данни. Структурата е:



Фиг. 16 Съдържание на папка Repository

#### **Папка Entities**

Тази папка съдържа класовете които са използвани за генериране на таблиците в базата данни. Това са класовете използвани с подхода Code First. Класовете директно съответстват на таблиците в базата данни.



Фиг. 17 Съдържание на папка Entities

##### **Файл ParkingData**

Този файл съдържа променливи и указания как те да бъдат трансформирани в таблица за базата данни. Съдържанието е:

* Id от тип int за уникален идентификатор
  + Използват се атрибути:
    - [Key] оказващ, че променливата е главен ключ (primary key)
    - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
* Name от тип string
  + Използва атрибути:
    - [MaxLength(32)] оказващ максимална дължина от 64 символа
    - [Required]
* Description от тип string
  + Използва атрибути:
    - [MaxLength(256)] оказващ максимална дължина от 64 символа
    - [Required]
* AllSpaces от тип int
  + Използва атрибути:
    - [Required]
* FreeSpacesLeft от тип int
  + Използва атрибути:
    - [Required]

Класът има атрибути:

* [Table(“ParkingData”)] оказващ, че името на таблицата в базата да е ParkingData.

##### **Файл ParkingStatus**

Този файл съдържа променливи и указания как те да бъдат трансформирани в таблица за базата данни. Съдържанието е:

* Id от тип int за уникален идентификатор
  + Използват се атрибути:
    - [Key] оказващ, че променливата е главен ключ (primary key)
    - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
* LicensePlate от тип string
  + Използва атрибути:
    - [MaxLength(10)] оказващ максимална дължина от 64 символа
    - [Required]
* TimeOfArrival от тип DateTime
  + Използва се атрибут:
    - [Required]
* TimeOfLeave от тип DateTime
  + Използва се атрибут:
    - [Required]
* RegisteredVehicle от тип RegisteredVehicle оказваща връзка с таблица RegisteredVehicle
* RegisteredVehicleId от тип int?
* PriceRange от тип PriceRange оказваща връзка с таблица PriceRange
* PriceRangeId от тип int?
* Payment от тип Payment оказваща връзка с таблица Payment

Класът има атрибут:

* [Table(“ParkingStatus”)] оказващ, че името на таблицата в базата да е ParkingStatus.

##### **Файл Payments**

Този файл съдържа променливи и указания как те да бъдат трансформирани в таблица за базата данни. Съдържанието е:

* Id от тип int за уникален идентификатор
  + Използват се атрибути:
    - [Key] оказващ, че променливата е главен ключ (primary key)
    - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
* PaymentDate от тип DateTime
  + Използва се атрибут:
    - [Required]
* PaymentAmoung от тип double
  + Използва се атрибут:
    - [Required]
* ParkingStatus от тип ParkingStatus използван за външен ключ (foreign key) към таблица ParkingStatus
* ParkingStatusId от тип int за съхранение на Id от връзката с таблица ParkingStatus

Класът има атрибут:

* [Table(“Payments”)] оказващ, че името на таблицата в базата да е Payments.

##### **Файл PriceRange**

Този файл съдържа променливи и указания как те да бъдат трансформирани в таблица за базата данни. Съдържанието е:

* Id от тип int за уникален идентификатор
  + Използват се атрибути:
    - [Key] оказващ, че променливата е главен ключ (primary key)
    - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
* Price от тип double
  + Използва атрибути:
    - [Required]
* RegisteredVehicles от тип IEnumerable<RegisteredVehicle>
* ParkingStatuses от тип IEnumerable<ParkingStatus>

Класът има атрибут:

* [Table(“PriceRanges”)] оказващ, че името на таблицата в базата да е PriceRanges.

#### **Файл RegisteredVehicles**

Този файл съдържа променливи и указания как те да бъдат трансформирани в таблица за базата данни. Съдържанието е:

* Id от тип int за уникален идентификатор
  + Използват се атрибути:
    - [Key] оказващ, че променливата е главен ключ (primary key)
    - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
* LicensePlate от тип string
  + Използва атрибути:
    - [MaxLength(10)] оказващ максимална дължина от 64 символа
    - [Required]
* PriceRange от тип PriceRange използван за външен ключ (foreign key) към таблица PriceRange
* PriceRangeId от тип int за съхранение на Id от връзката с таблица PriceRanges
* VehicleType от тип VehicleType използван за външен ключ (foreign key) към таблица VehicleTypes
* VehicleTypeId от тип int за съхранение на Id от връзката с таблица VehicleTypes
* ParkingStatuses от тип IEnumerable<ParkingStatus>

Класът има атрибут:

* [Table(“RegisteredVehicles”)] оказващ, че името на таблицата в базата да е RegisteredVehicles.

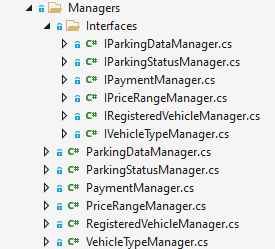
#### **Файл RegisteredVehicles**

Този файл съдържа променливи и указания как те да бъдат трансформирани в таблица за базата данни. Съдържанието е:

* Id от тип int за уникален идентификатор
  + Използват се атрибути:
    - [Key] оказващ, че променливата е главен ключ (primary key)
    - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
* Type от тип string
  + Използва атрибути:
    - [MaxLength(32)] оказващ максимална дължина от 64 символа
    - [Required]
* RegisteredVehicles от тип IEnumerable< RegisteredVehicle>

#### **Папка с мениджъри (Managers)**

Тази папка съдържа класове, които се използват за управление на данните от базата. За всички таблици от базата са създадени основните операции CRUD, както и за всяка таблица са дефинирани специфични допълнителни операции, които са дефинирани в интерфейс, а след това имплементирани в клас. Интерфейсите са отделени в отделна папка наречена Interfaces. Съдържанието на папка Managers е:



Фиг. 18 Съдържание на папка Managers

За изпълнение на заявките към базата се използва LINQ. LINQ (Language-Integrated Query) представлява редица разширения на .NET Framework, които включват интегрирани в езика заявки и операции върху елементи от източник на данни. LINQ e инструмент, който прилича на SQL и по синтаксис и по логика на изпълнение. LINQ реално обработва колекциите по подобие на SQL езиците, които обработват редовете в таблици в база данни. Той е част от C**#** и Visual Basic синтаксиса.

Също така за допълнителна обработка са използвани ламбда изрази. Те представляват анонимни функции, които съдържат изрази или последователност от оператори. Всички ламбда изрази използват оператора **=>**. Лявата страна на ламбда оператора определя входните параметри на анонимната функция, а дясната страна представлява израз или последователност от оператори, която работи с входните параметри и евентуално връща някакъв резултат.

##### **IParkingDataManager и ParkingDataManager**

В тези файлове не са дефинирани допълнителни функционалности, но файловете са създадени, за да може да се използва Unit Of Work патърн.

##### **IParkingStatusManager и ParkingStatusManager**

В тези файлове са дефинирани и имплементирани, различни операции от основните и специфични за таблица ParkingStatuses.

* GetEntryByDate(DateTime datetime) - този метод се използва, за да се извлекат от базата всички редове от таблица ParkingStatuses, като се използва LINQ, за да се вземе и информацията за всички статуси по на конкретна дата. Данните се връщат в колекция IEnumerable<ParkingStatus>.
* GetByLicensePlate(string licensePlate) - този метод се използва за намиране на редове от таблицата, като се търси по уникалния идентификатор номер на автомобила. Данните се връщат с тип IEnumerable<ParkingStatus>.
* GetByLicensePlateLastEntry(string licensePlate) – този метод се използва за намиране дали автомобил с даден номер към настоящия момент се намира в паркинга. Дали има регистриран час за вход и няма час за изход. Данните се връщат с тип ParkingStatus.

##### **IPaymentManager и PaymentManager**

В тези файлове са дефинирани и имплементирани, различни операции от основните и специфични за таблица Payments.

* GetAllByDate(DateTime date) - този метод се използва за извличане на всички редове от таблица, като се търси по дата на плащането. Данните се връщат в колекция IEnumerable<Payment>.
* GetAllByLicensePlate(string licensePlate) - този метод се използва, за да се извлекат от базата всички редове от таблица, като се използва LINQ, за да се вземе и информацията за всички разплащания на конкретен автомобил. Данните се връщат в колекция IEnumerable<Payment>.

##### **IPriceRangeManager, PriceRangeManager и IVehicleTypeManager, VehicleTypeManager**

В тези файлове не са дефинирани допълнителни функционалности, но файловете са създадени, за да може да се използва Unit Of Work патърн.

##### **IRegisteredVehicleManager и RegisteredVehicleManager**

В тези файлове са дефинирани и имплементирани, различни операции от основните и специфични за таблица RegisteredVehicles.

* GetAllWithRelations() - този метод се използва за извличане на всички редове от таблицата, като се прави връзка към другите свързани таблици за да се извлече информацията по foreign keys. Данните се връщат в колекция IEnumerable<RegiteredVehicle>.
* GetByLicensePlate(string licensePlate) - този метод се използва, за да се извлече конкретен автомобил, като се използва LINQ се прави връзка към другите свързани таблици за да се извлече информацията по foreign keys. Данните се връщат в колекция RegisteredVehicle.

#### **Папка RepositoryBase**

В тази папка са файловете, които отговарят за реализацията на Repository патърн.



Фиг. 19 Съдържание на папка RepositoryBase

В същността си, Repository патърн предоставя абстракция на информацията, като по този начин, приложението може да работи с най – простата абстракция, която има интерфейс, отговарящ за колекция. Добавяне, изтриване, ъпдейтване и извличане от съответната колекция се осъществява чрез няколко метода, без нуждата от работа с връзка към базата, команди, курсори и други. Има различни начини за реализация на патърна, като създаване на патърн за всеки клас отговарящ за таблица (entity). В приложението патърна е с една стъпка напред, като е създаден generic клас, който да може да работи с всички ентитита.

Generic клас позволява да се преизползват алгоритми, без нуждата да се репликира код за определен тип данни. Дефинират се методи или класове, които отлагат задаването на тип докато кодът не се изпълни при клиента.

Функциите на generic типа:

* Помага за преизползването на код
* Могат да се създават собствени generic класове, методи, интерфейси
* Могат да се създават собствени колекции
* Може да се получи информация за използваните типове в run-time

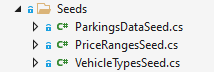
##### **IRepositoryBase и RepositoryBase**

В тези файлове са дефинирани и имплементирани основните операции за работа на ентититата с базата данни и те са:

* GetAll() – извличане на всички редове от определена таблица. Резултатът се връща в колекция с тип на ентити отговарящо за таблицата - IEnumerable<T>
* FindById(int id) – извлича ред от таблица, като се търси по уникален идентификатор. Резултатът е с тип на ентитито T.
* Create(T entity) – създава се ред в таблица съответстваща на подаденото ентити. Методът е void.
* Update(T entity) – ъпдейтва се ред от таблицата по Id, което се взема от подаденото ентити
* Delete(T entity) – изтриван се ред от таблицата по Id, което се взема от подаденото ентити
* Count() – връща се броят на всички редове от съответна таблица

#### **Папка Seeds**

Тази папка съдържа класове, които се използват, за да се зареди начална информация в таблиците от базата данни. Структурата е:



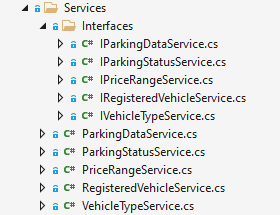
Фиг. 20 Съдържание на папка Seeds

* Файл ParkingDataSeed – съдържа метод, който връща масив от тип ParkingData[], данните от който да ще се запишат в таблица ParkingData при създаването и.
* Файл PriceRangesSeed – съдържа метод, който връща масив от тип PriceRange[], данните от който да ще се запишат в таблица PriceRanges при създаването и.
* Файл VehicleTypeSeed - съдържа метод, който връща масив от тип VehicleType[], данните от който да ще се запишат в таблица VehicleTypes при създаването и.

#### **Папка Services**

Тази папка съдържа интерфейси, които дефинират методи за бизнес логиката за всяка таблица, и класове които имплементират интерфейсите и реализират самите методи.

Структурата на папката е:



Фиг. 21 Съдържание на папка Services

##### **IAtWorkRepository и AtWorkService**

В тези файлове е описана бизнес логиката за работа с таблица AtWork. Методите са

* GetAllRows() – връща всички редове от таблица AtWork като преди това са мапнати чрез AutoMapper към AtWorkModel. Връща се колекция IEnumerable<AtWorkModel>.
* GetByEmployeeId(int employeeId) – ако съществува се връща ред от таблица AtWork, търсен по уникалния идентификатор на служител. Преди връщане данните се мапват към AtWorkModel.
* GetEntityByEmployeeId(int employeeId) - ако съществува се връща ред от таблица AtWork, търсен по уникалния идентификатор на служител. Данните се връщат в тип AtWork.
* LoginAtWork(int employeeId) – използва се уникалният идентификатор на служител, за да се създаде ред в таблица AtWork. Служителят е отчел начало на работния ден.

##### **IEmployeeRepository и EmployeeService**

В тези файлове е описана бизнес логиката за работа с таблица Employees. Методите са

* Authenticate(string username, string password) – по подадени потребителско име и парола се намира служител в базата. Ако не се намери се връща празен обект, ако се намери се създава JWT токън след което данните за служителя както и токъна се връщат на клиента в тип EmployeeModel
* GetAllEmployees() – връщат се всички редове от таблица Employees, като преди връщане се мапват към EmployeeModel. Връща се колекция IEnumerable<EmployeeModel>
* GetById(int id) – връща се ред от таблица Employees, като се търси по id на служител. Преди връщане се мапва към EmployeeModel
* GetEntityById(int id) - връща се ред от таблица Employees, като се търси по id на служител. Връща се от тип Employee.
* GetLoginInfoUserId(string username, string password) – използва се за вземане на уникалния идентификатор на служител, по потребителско име и парола. Ако не е намерен служител се връща празен обект null.
* CreateEmployee(CreateEmployeeModel createEmployeeModel) – използва се за регистриране на служител в системата. Ако потребителското име съществува се прекратява операцията. Ако е свободно се създава хеш и salt на паролата и се взема уникалният идентификатор на избраната позиция. След това се записва в базата. Ако записът е успеше се връщат данните в тип EmployeeModel, а ако е възникнала грешка – null.
* CreatePasswordHash(string password, out byte[] passwordHash, out byte[] passwordSalt) – използва се за създаване на хеш и salt на паролата на потребителя, след което те се връщат. Ключовата дума out се използва за връщане на повече от един резултата.
* VerifyPasswordHash(string password, byte[] storedHash, byte[] storedSalt) – използва се за сравнение на въведената парола от служителя и паролата съхранена в базата данни. При съвпадение се връща true, а при разлика се връща false.
* DeleteEmployee(int id) – използва се за изтриване на служител от таблица Employees по уникален идентификатор. Също така каскадно се изтриват и данните за служителя от таблица LoginInfos.
* UpdateEmployee(UpdateEmployeeModel employeeModel) – използва се за ъпдейтване на данните за служител. Първо се изтеглят данните за служителя от базата, като се използва уникалният идентификатор. След това се намират разликите в новите данни и тези от базата. Ако има промяна в личните данни на служителя се прави ъпдейт само на записа от таблица служители, ако има промяна на паролата се прави промяна само на записа от таблица LoginInfos, ако има промяна и на двете места се ъпдейтват редовете от двете таблици.

##### **ILoginHistoryRepository и LoginHistoryService**

В тези файлове е описана бизнес логиката за работа с таблица LoginHistory. Методите са

* GetAllRows() – използва се за извличане на всички редове от таблица LoginHistory. Преди връщане данните се мапват към LoginHistoryModel. Връща се колекция IEnumerable<LoginHistoryModel>.
* GetAllRowsByEmployeeId(int employeeId) – използва се за връщане на всички редове от таблица LoginHistory като се използва като критерий уникалният идентификатор на служител. Преди връщане данните се мапват към LoginHistoryModel. Връща се колекция IEnumerable<LoginHistoryModel>.
* LogoutFromWork(AtWork atWork) – използва се за отбелязване на край на работния ден.

##### **ILoginInfoRepository и LoginInfoService**

В тези файлове е описана бизнес логиката за работа с таблица LoginInfos. Методите са

* GetByUserId(int id) – използва се за връщане на ред от таблица LoginInfos, като се търси по уникалния идентификатор на служител. Преди връщане данните се мапват към LoginModel.
* GetEntityByUserId(int id) - използва се за връщане на ред от таблица LoginInfos, като се търси по уникалния идентификатор на служител. Данните се връщат с тип LoginInfo

##### **IRoleRepository и IRoleService**

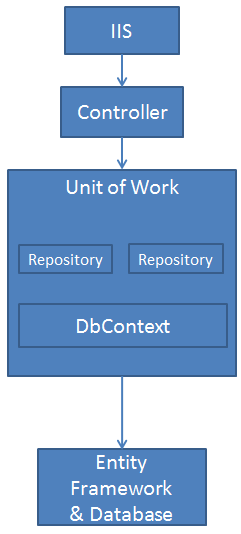
В тези файлове е описана бизнес логиката за работа с таблица Roles. Методите са

* AddRole(string RoleDescription) – използва се за добавяне на ред в таблица Roles. Методът не връща резултат.
* GetAllRoles() – използва се за извличане на всички редове от таблица Roles. Преди връщане данните се мапват към тип RoleModel. Връща се колекция с тип IEnumerable<RoleModel>.

#### **Папка UnitOfWork**

Unit of Work се разглежда като една транзакция, която включва множество операции insert, update, delete. Ако не се използва този патърн, всяко ентити ще използва собствен контекст за базата данни, което може да представлява проблем. Ако се изпълни операция, която включва две или повече ентитита ,и при някое от тях възникне проблем и транзакцията е неуспешна, това няма да спре останалите транзакции на другите ентитита и информацията записана в базата данни няма да е правилна.

За да се избегне тази ситуация се добавя допълнителен слой, който действа като централизиран източник, който зарежда всички репозиторита с инстанция на контекста на базата данни. По този начин се осигурява, че когато се изпълнява транзакция с много ентитита, или всичко ще проработи или всичко ще се прекрати поради факта, че споделят един и същ контекст на базата данни.



Фиг. 22 Диаграма показваща разделението на слоеве при използване на Unit of Work и Repository патърн

Съдържанието на папка UnitOfWork:

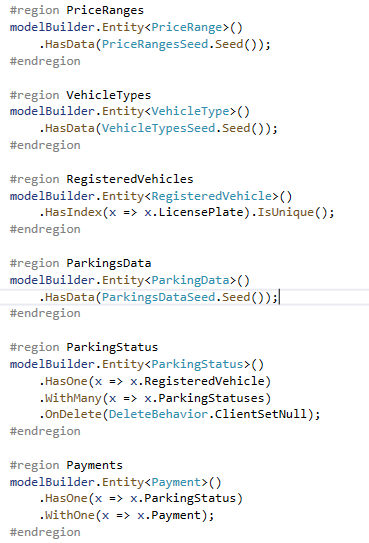


Фиг. 23 Съдържание на папка UnitofWork

* IUnitOfWork – в този интерфейс се съдържат интерфейсите на мениджърите на таблиците, на които трябва да бъде зареден контекст на базата данни както и методите:
  + SaveChanges() – използва се за записване в базата на направените промени
  + BeginTransaction() – използва се за започване на нова транзакция в рамките на контекста.
  + CommitTransaction() – използва се за запазване на промените направени в цялата транзакция до момента
  + RollbackTransaction() – използва се за премахване на направените промени в транзакцията до момента. Няма записани данни от транзакцията в базата.
* UnitOfWork – имплементира интерфейса, като в конструктора си подава като параметър за създаване на мениджърите на таблиците контекста на базата данни. Така всички мениджъри използват една и съща инстанция на контекста. Също така са реализирани методите на интерфейса.

#### **Файл RepositoryContext**

Този клас съдържа член променливи, които се използват за директна манипулация на данните от базата данни. Съществува промелива от тип DbSet за всяко ентити. Също така в метод OnModelCreating е зададена допълнителна информация за самите таблици. Уточнено е как да се направят връзките между таблиците, както и се използват примерните данни взети от Seed файловете, за да се запишат в базата при създаването им.



Фиг. 24 Задаване на допълнителни условия за създаването на таблиците

### **Файл Startup**

В клас Startup се конфигурират сървисите и канала на приложението за заявките. Този клас съдържа два метода:

* Метод ConfigureServices се използва за конфигуриране на сървисите на приложението. Те се регистрират в метода след което могат да се използват в цялото приложение, посредством dependency injection (DI). Този метод не е задължителен. В този метод са конфигурирани всички сървиси, AutoMapper, и connection string за връзка с базата данни.
* Метод Configure се използва за конфигуриране на канала за обработване на заявки. Приложението има конфигурации, които позволяват изпращането на заявки от други домейни – Cross-Origin Resource Sharing (CORS). Също така има конфигурация, позволяваща използването на авентикация и оторизация.

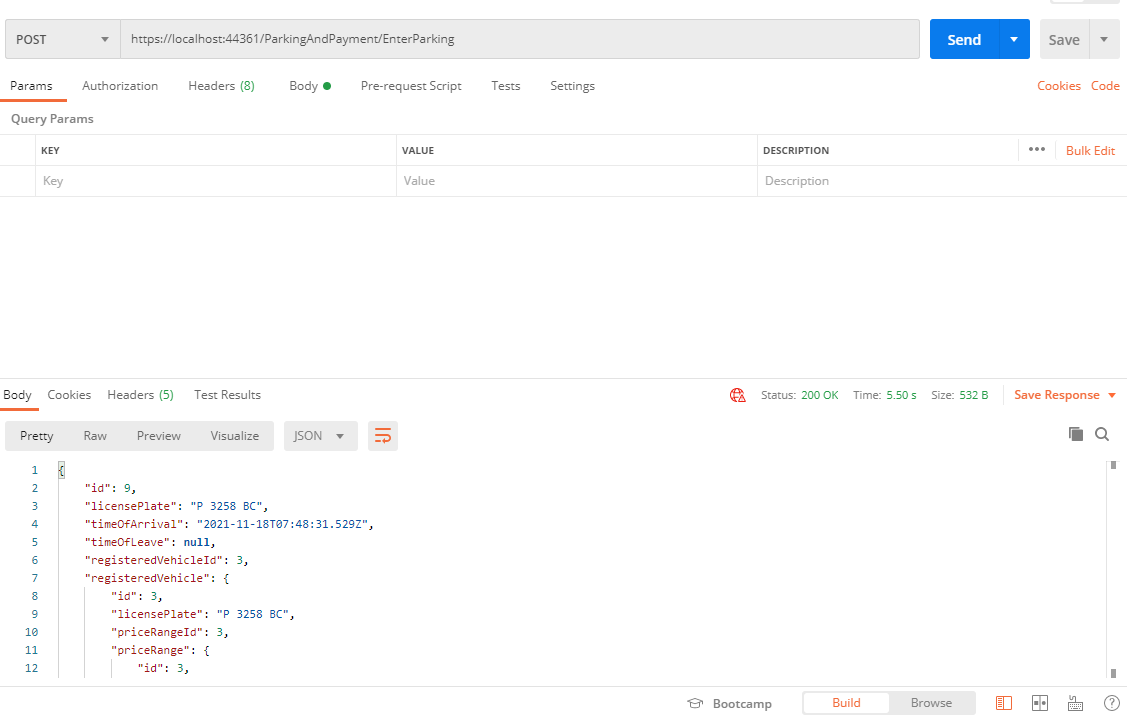
# **Тестване на системата, изводи и заключения**

## **Тестване на работоспособността на системата**

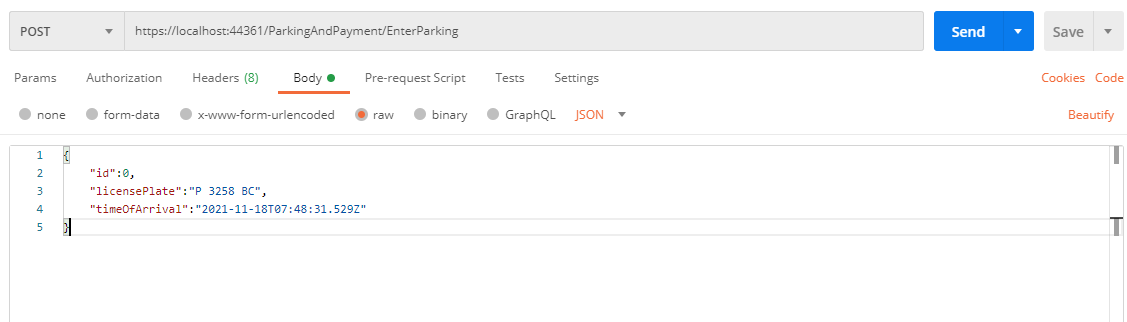
За изпробване на работоспособността на системата се използва програмата Postman. Postman е инструмент за правене на HTTP заявки и добавяне на [автоматизирани тестове](https://dev.bg/%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D0%BD%D0%BE-%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82%D0%B8%D0%B7%D0%B8%D1%80%D0%B0%D0%BD%D0%BE-%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B0%D0%BD/) към техните резултати Поради тези причини, Postman може да се използва успешно както от отделни членове на екипа, така и при колаборация между цели екипи. Чрез изпратените заявки от Postman са прегледани всички входни точки към системата. Отговорът и данните са проверени дали са тези, които трябва да бъдат. При несъответствие или грешка е използван дебъгерът, вграден във Visual Studio, за намиране и отстраняване на грешка или бъг.

Някои тестови заявки и техните отговори:

* Влизане в паркинга

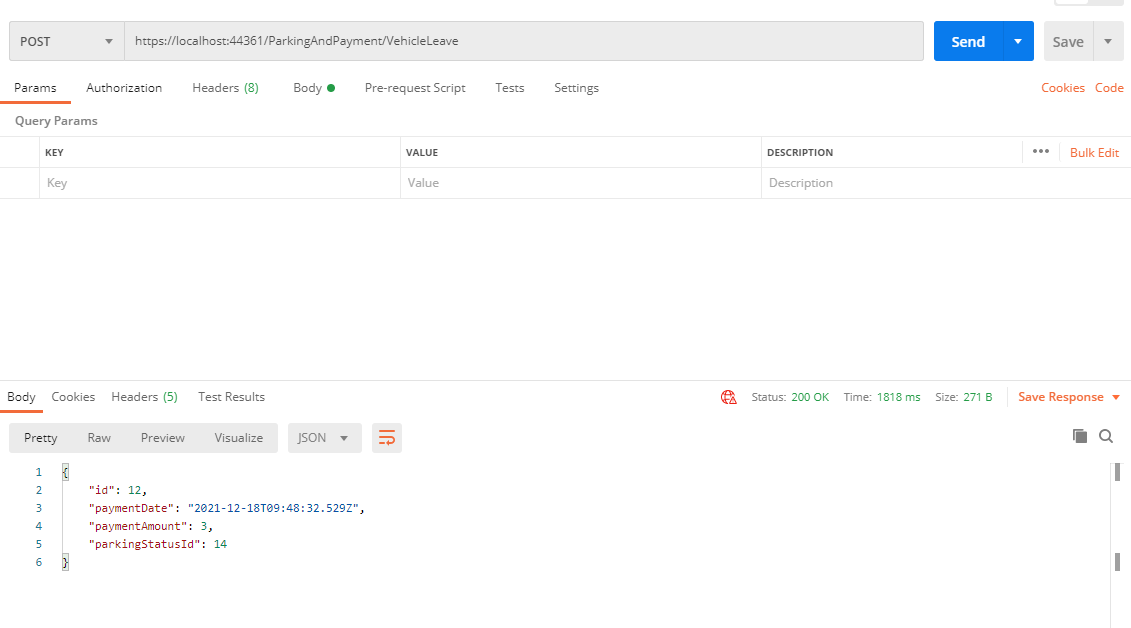


Фиг. 25 Отговор на изпратената заявка влизане в паркинга

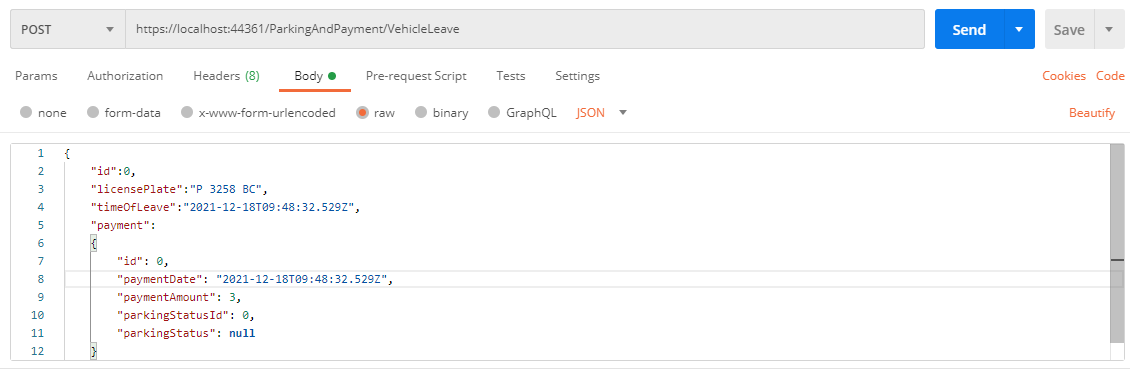


Фиг. 26 Информацията изпратена в body на заявката

* Излизане от паркинг

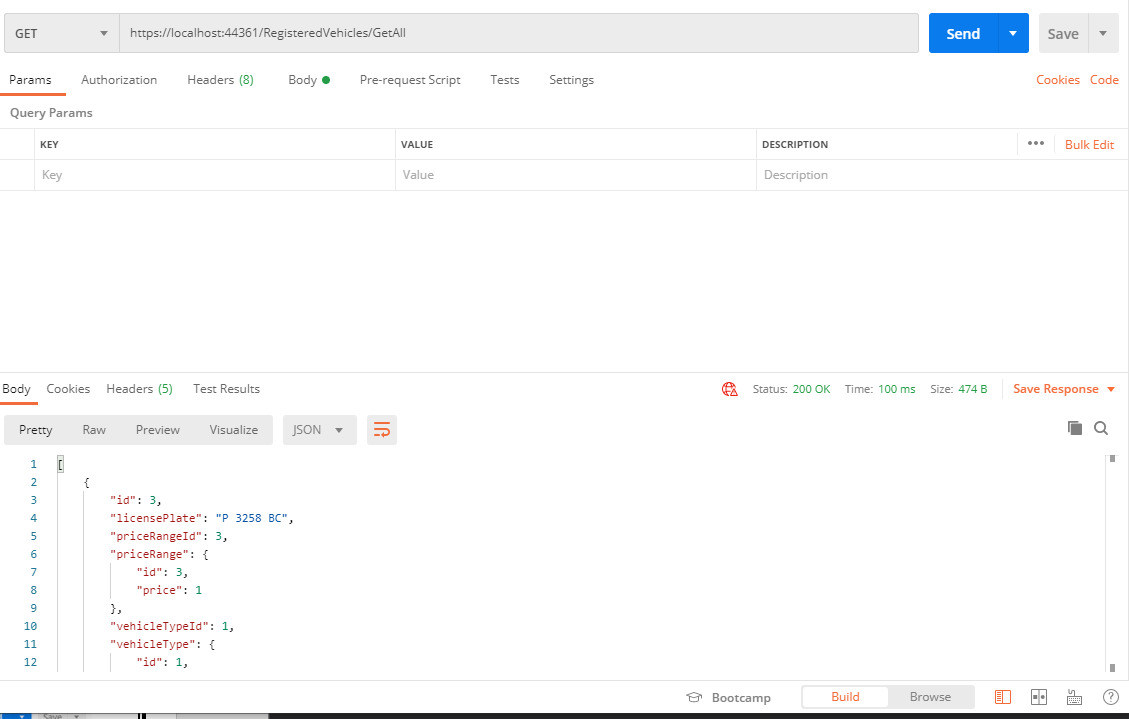


Фиг. 27 Отговор на изпратената заявка за излизане от паркинга



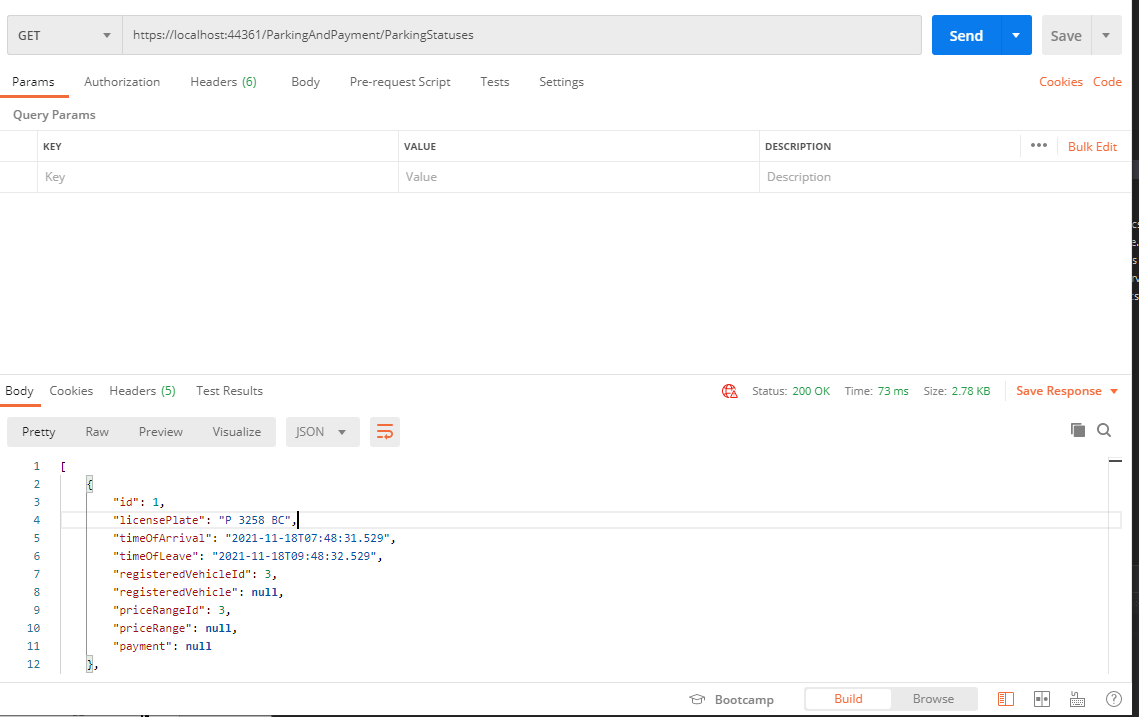
Фиг. 28 Информацията изпратена в body на заявката

* Извличне на всички регистрирани автомобили

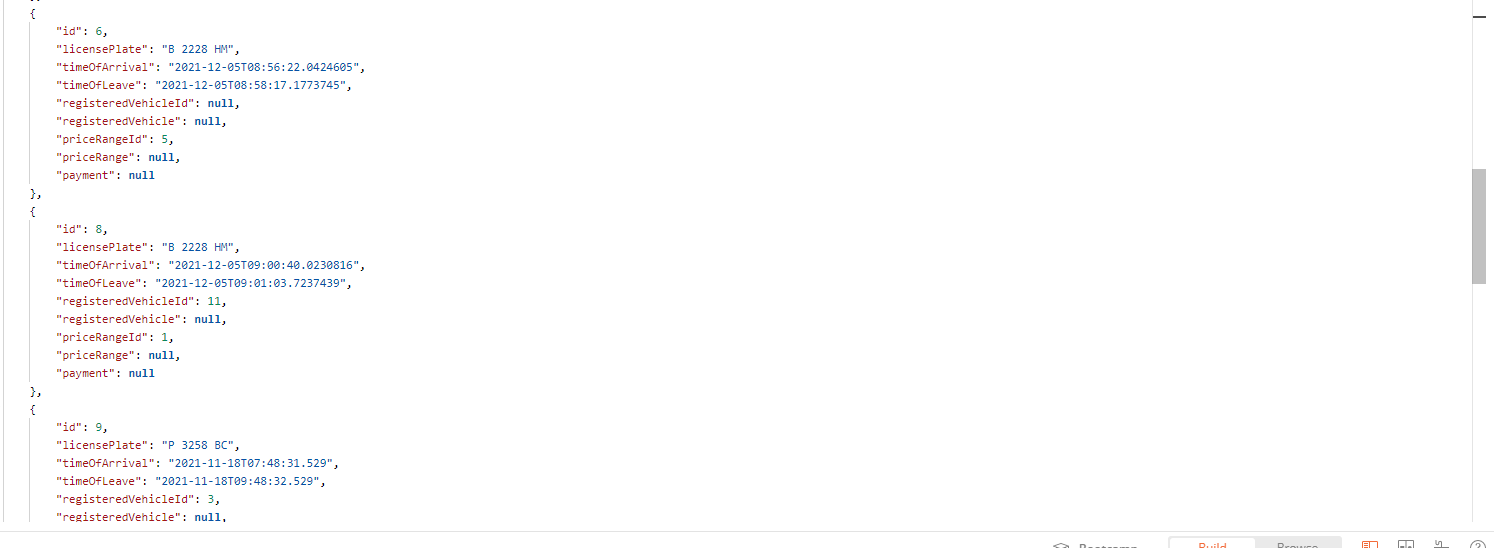


Фиг. 29 Отговор от заявка

* Извличане на цялата история на паркинга



Фиг. 30 Отговор от заявка



Фиг. 31 Отговор от заявка

## **Изводи и заключения**

Разработената дипломна работа на тема „Проектиране и реализация на Web Api и десктоп приложение за паркинг система“ представлява цялостен и завършен бекенд, отразяващ провесите проектиране и разработване на системата, използвайки съвременни технологии и методи. Проектираната архитектура и нейната имплементация позволяват лесно и модифициране на разработените модули, като този начин могат да се удовлетворят изискванията на клиента.

**Като бъдещо развитие могат да бъдат добавени следните функционалности:**

* Изграждане на допълнителна архитектура в базата данни, за да могат да се поддържат множество паркинги.
* Интефриране с хардуер като различни видове бариери.
* Интегриране на различни начини за разплащане
* Интегриране с различни библиотеки за разпознаване на регистрационните номера.
* Разпознаване на номера от различни държави

# **Използвана литература**

**Книги**

1. Принципи на програмирането със C# - д-р Светлин Наков, Веселин Колев и колектив

**Сайтове**

**.Net Core**

1. [**https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-aspnet-core-ef-step-01?view=vs-2019**](https://docs.microsoft.com/en-us/visualstudio/get-started/csharp/tutorial-aspnet-core-ef-step-01?view=vs-2019)
2. [**https://docs.microsoft.com/en-us/dotnet/core/tutorials/**](https://docs.microsoft.com/en-us/dotnet/core/tutorials/)
3. [**https://www.udemy.com/course/build-an-app-with-aspnet-core-and-angular-from-scratch/learn/lecture/11237580#overview**](https://www.udemy.com/course/build-an-app-with-aspnet-core-and-angular-from-scratch/learn/lecture/11237580#overview)

**Auto Mapper**

1. [**https://automapper.org/**](https://automapper.org/)

**Entity Framework Core**

1. [**https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=netcore-cli**](https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=netcore-cli)

**RestSharp**

1. **https://restsharp.dev**

**MS SQL Server**

1. [**https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15**](https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15)

**Unit of Work и Repository патърн**

1. [**https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application**](https://docs.microsoft.com/en-us/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application)

# **Приложения**

**Приложена част от кода на сървърното приложение**

**ParkingAndPaymentController.cs**

[ApiController]

[Route("[controller]")]

public class ParkingAndPaymentController : Controller

{

private readonly IParkingStatusService parkingStatusService;

public ParkingAndPaymentController(IParkingStatusService parkingStatusService)

{

this.parkingStatusService = parkingStatusService;

}

[HttpPost]

[Route("EnterParking")]

public IActionResult EnterParking([FromBody] ParkingStatusModel parkingStatusModel)

{

try

{

var parkingStatus = parkingStatusService.VehicleEntry(parkingStatusModel);

return Ok(parkingStatus);

}

catch (Exception ex)

{

return Problem(ex.Message);

}

}

[HttpPost]

[Route("GetCalculatedPaymentForLicensePlate")]

public IActionResult GetCalculatedPaymentForLicensePlate([FromBody] ParkingStatusModel parkingStatusModel)

{

try

{

var payment = parkingStatusService.GetCalculatedPayment(parkingStatusModel);

return Ok(payment);

}

catch (Exception ex)

{

return Problem(ex.Message);

}

}

[HttpPost]

[Route("VehicleLeave")]

public IActionResult VehicleLeave([FromBody] ParkingStatusModel parkingStatusModel)

{

try

{

var payment = parkingStatusService.Leave(parkingStatusModel);

return Ok(payment);

}

catch (Exception ex)

{

return Problem(ex.Message);

}

}

[HttpGet]

[Route("ParkingStatuses")]

public IActionResult GetAllParkingStatuses()

{

var data = parkingStatusService.GetAllParkingStatus();

return Ok(data);

}

[HttpGet]

[Route("GetAllPayments")]

public IActionResult GetAllPayments()

{

var data = parkingStatusService.GetAllPayments();

return Ok(data);

}

[HttpPost]

[Route("GetPaymentsByLicensePlate")]

public IActionResult GetPaymentsByLicensePlate([FromBody] string licensePlate)

{

try

{

var payments = parkingStatusService.GetParkingStatusByLicensePlate(licensePlate);

return Ok(payments);

}

catch (Exception ex)

{

return Problem(ex.Message);

}

}

}

}

**MappingConfig.cs**

public class MappingConfig : Profile

{

public MappingConfig()

{

PriceRangeMap();

PriceRangeModelMap();

VehicleTypeMap();

VehicleTypeModelMap();

ParkingDataMap();

ParkingDataModelMap();

RegisteredVehicleMap();

RegisteredVehicleModelMap();

ParkingStatusMap();

ParkingStatusModelMap();

PaymentMap();

PaymentModelMap();

}

private void PriceRangeMap()

{

CreateMap<PriceRange, PriceRangeModel>()

;

}

private void PriceRangeModelMap()

{

CreateMap<PriceRangeModel, PriceRange>()

;

}

private void VehicleTypeMap()

{

CreateMap<VehicleType, VehicleTypeModel>()

;

}

private void VehicleTypeModelMap()

{

CreateMap<VehicleTypeModel, VehicleType>()

;

}

private void ParkingDataMap()

{

CreateMap<ParkingData, ParkingDataModel>()

;

}

private void ParkingDataModelMap()

{

CreateMap<ParkingDataModel, ParkingData>()

;

}

private void RegisteredVehicleMap()

{

CreateMap<RegisteredVehicle, RegisteredVehicleModel>()

;

}

private void RegisteredVehicleModelMap()

{

CreateMap<RegisteredVehicleModel, RegisteredVehicle>()

;

}

private void ParkingStatusMap()

{

CreateMap<ParkingStatus, ParkingStatusModel>()

;

}

private void ParkingStatusModelMap()

{

CreateMap<ParkingStatusModel, ParkingStatus>()

;

}

private void PaymentMap()

{

CreateMap<Payment, PaymentModel>()

;

}

private void PaymentModelMap()

{

CreateMap<PaymentModel, Payment>()

;

}

}

}

**20211115091321\_CreateVehicleTypePriceRangeRegisteredVehicles.cs**

public partial class CreateVehicleTypePriceRangeRegisteredVehicles : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.CreateTable(

name: "PriceRange",

columns: table => new

{

Id = table.Column<int>(nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Price = table.Column<double>(nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_PriceRange", x => x.Id);

});

migrationBuilder.CreateTable(

name: "VehicleTypes",

columns: table => new

{

Id = table.Column<int>(nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

Type = table.Column<string>(maxLength: 32, nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_VehicleTypes", x => x.Id);

});

migrationBuilder.CreateTable(

name: "RegisteredVehicles",

columns: table => new

{

Id = table.Column<int>(nullable: false)

.Annotation("SqlServer:Identity", "1, 1"),

LicensePlate = table.Column<string>(maxLength: 10, nullable: false),

PriceRangeId = table.Column<int>(nullable: false),

VehicleTypeId = table.Column<int>(nullable: false)

},

constraints: table =>

{

table.PrimaryKey("PK\_RegisteredVehicles", x => x.Id);

table.ForeignKey(

name: "FK\_RegisteredVehicles\_PriceRange\_PriceRangeId",

column: x => x.PriceRangeId,

principalTable: "PriceRange",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

table.ForeignKey(

name: "FK\_RegisteredVehicles\_VehicleTypes\_VehicleTypeId",

column: x => x.VehicleTypeId,

principalTable: "VehicleTypes",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

});

migrationBuilder.InsertData(

table: "PriceRange",

columns: new[] { "Id", "Price" },

values: new object[,]

{

{ 1, 0.0 },

{ 2, 0.5 },

{ 3, 1.0 }

});

migrationBuilder.InsertData(

table: "VehicleTypes",

columns: new[] { "Id", "Type" },

values: new object[,]

{

{ 1, "Ambulance" },

{ 2, "Police Car" },

{ 3, "Fire Truck" },

{ 4, "Ambulance" },

{ 5, "Company Car" },

{ 6, "Paid Subscription" }

});

migrationBuilder.CreateIndex(

name: "IX\_RegisteredVehicles\_LicensePlate",

table: "RegisteredVehicles",

column: "LicensePlate",

unique: true);

migrationBuilder.CreateIndex(

name: "IX\_RegisteredVehicles\_PriceRangeId",

table: "RegisteredVehicles",

column: "PriceRangeId");

migrationBuilder.CreateIndex(

name: "IX\_RegisteredVehicles\_VehicleTypeId",

table: "RegisteredVehicles",

column: "VehicleTypeId");

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(

name: "RegisteredVehicles");

migrationBuilder.DropTable(

name: "PriceRange");

migrationBuilder.DropTable(

name: "VehicleTypes");

}

}

**ParkingStatusModel.cs**

public class ParkingStatusModel

{

public int Id { get; set; }

public string LicensePlate { get; set; }

public DateTime TimeOfArrival { get; set; }

public DateTime? TimeOfLeave { get; set; }

//Relations

public int? RegisteredVehicleId { get; set; }

public RegisteredVehicleModel RegisteredVehicle { get; set; }

public int PriceRangeId { get; set; }

public PriceRangeModel PriceRange { get; set; }

public PaymentModel Payment { get; set; }

}

**ParkingStatus.cs**

[Table("ParkingStatus")]

[Serializable]

public class ParkingStatus

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int Id { get; set; }

[Required]

[MaxLength(10)]

public string LicensePlate { get; set; }

[Required]

public DateTime TimeOfArrival { get; set; }

public DateTime? TimeOfLeave { get; set; }

//Relations

public int? RegisteredVehicleId { get; set; }

public RegisteredVehicle RegisteredVehicle { get; set; }

public int PriceRangeId { get; set; }

public PriceRange PriceRange { get; set; }

public Payment Payment { get; set; }

}

**IParkingStatusManager.cs**

public interface IParkingStatusManager : IRepositoryBase<ParkingStatus>

{

public IEnumerable<ParkingStatus> GetByEntryDate(DateTime dateTime);

public IEnumerable<ParkingStatus> GetByLicensePlate(string licensePlate);

public ParkingStatus GetByLicensePlateLastEntry(string licensePlate);

}

**ParkingStatusManager.cs**

public class ParkingStatusManager : RepositoryBase<ParkingStatus>, IParkingStatusManager

{

public ParkingStatusManager(RepositoryContext repositoryContext) : base(repositoryContext)

{

}

public IEnumerable<ParkingStatus> GetByEntryDate(DateTime dateTime)

{

return RepositoryContext.ParkingStatuses.Where(x => x.TimeOfArrival.Equals(dateTime));

}

public IEnumerable<ParkingStatus> GetByLicensePlate(string licensePlate)

{

return RepositoryContext.ParkingStatuses.Where(x => x.LicensePlate.Equals(licensePlate));

}

public ParkingStatus GetByLicensePlateLastEntry(string licensePlate)

{

return RepositoryContext.ParkingStatuses.Where(x => x.LicensePlate.Equals(licensePlate) && x.TimeOfLeave == null)

.Include(x => x.PriceRange)

.SingleOrDefault();

}

}

**VehicleTypeSeed.cs**

public static class VehicleTypesSeed

{

public static VehicleType[] Seed()

{

return new VehicleType[]

{

new VehicleType{Id=1, Type="Ambulance"},

new VehicleType{Id=2, Type="Police Car"},

new VehicleType{Id=3, Type="Fire Truck"},

new VehicleType{Id=4, Type="Ambulance"},

new VehicleType{Id=5, Type="Company Car"},

new VehicleType{Id=6, Type="Paid Subscription"},

};

}

}

**IParkingStatusService.cs**

public interface IParkingStatusService

{

public IEnumerable<ParkingStatusModel> GetAllParkingStatus();

public IEnumerable<PaymentModel> GetAllPayments();

public IEnumerable<ParkingStatusModel> GetParkingStatusByLicensePlate(string licensePlate);

public IEnumerable<ParkingStatusModel> GetParkingStatusByEntryDate(DateTime entryDate);

public IEnumerable<PaymentModel> GetPaymentsByDate(DateTime paymentDate);

public IEnumerable<PaymentModel> GetPaymentsByLicensePlate(string licensePlate);

public ParkingStatusModel VehicleEntry(ParkingStatusModel parkingStatusModel);

public PaymentModel GetCalculatedPayment(ParkingStatusModel parkingStatusModel);

public PaymentModel Leave(ParkingStatusModel parkingStatusModel);

}

**ParkingStatusService.cs**

public class ParkingStatusService : IParkingStatusService

{

private readonly IUnitOfWork unitOfWork;

private readonly IMapper mapper;

public ParkingStatusService(IUnitOfWork unitOfWork, IMapper mapper)

{

this.unitOfWork = unitOfWork;

this.mapper = mapper;

}

public IEnumerable<ParkingStatusModel> GetAllParkingStatus()

{

var parkingStatuses = unitOfWork.ParkingStatusManager.GetAll();

return mapper.Map<IEnumerable<ParkingStatusModel>>(parkingStatuses);

}

public IEnumerable<PaymentModel> GetAllPayments()

{

var payments = unitOfWork.PaymentManager.GetAll();

return mapper.Map<IEnumerable<PaymentModel>>(payments);

}

public PaymentModel GetCalculatedPayment(ParkingStatusModel parkingStatusModel)

{

var lastEntry = unitOfWork.ParkingStatusManager.GetByLicensePlateLastEntry(parkingStatusModel.LicensePlate);

PaymentModel paymentModel = CalculatePayment(lastEntry.TimeOfArrival, parkingStatusModel.TimeOfLeave.GetValueOrDefault(), lastEntry.PriceRange.Price);

return paymentModel;

}

private static PaymentModel CalculatePayment(DateTime entry, DateTime leave, double amountHour)

{

var totalHours = Math.Ceiling(((leave - entry).TotalHours));

var price = totalHours \* amountHour;

var paymentModel = new PaymentModel { PaymentDate = leave, PaymentAmount = price };

return paymentModel;

}

public IEnumerable<ParkingStatusModel> GetParkingStatusByEntryDate(DateTime entryDate)

{

var parkings = unitOfWork.ParkingStatusManager.GetByEntryDate(entryDate);

return mapper.Map<IEnumerable<ParkingStatusModel>>(parkings);

}

public IEnumerable<ParkingStatusModel> GetParkingStatusByLicensePlate(string licensePlate)

{

var parkings = unitOfWork.ParkingStatusManager.GetByLicensePlate(licensePlate);

return mapper.Map<IEnumerable<ParkingStatusModel>>(parkings);

}

public IEnumerable<PaymentModel> GetPaymentsByDate(DateTime paymentDate)

{

var payments = unitOfWork.PaymentManager.GetAllByDate(paymentDate);

return mapper.Map<IEnumerable<PaymentModel>>(payments);

}

public IEnumerable<PaymentModel> GetPaymentsByLicensePlate(string licensePlate)

{

var payments = unitOfWork.PaymentManager.GetAllByLicensePlate(licensePlate);

return mapper.Map<IEnumerable<PaymentModel>>(payments);

}

public PaymentModel Leave(ParkingStatusModel parkingStatusModel)

{

var lastEntry = unitOfWork.ParkingStatusManager.GetByLicensePlateLastEntry(parkingStatusModel.LicensePlate);

lastEntry.TimeOfLeave = parkingStatusModel.TimeOfLeave.GetValueOrDefault();

//TODO: payment may be present in parking status model

//PaymentModel payment = CalculatePayment(lastEntry.TimeOfArrival, parkingStatusModel.TimeOfLeave.GetValueOrDefault(), lastEntry.PriceRange.Price);

Payment paymentEntity = mapper.Map<Payment>(parkingStatusModel.Payment);

lastEntry.Payment = paymentEntity;

try

{

unitOfWork.ParkingStatusManager.Update(lastEntry);

unitOfWork.SaveChanges();

return mapper.Map<PaymentModel>(lastEntry.Payment);

}

catch

{

throw;

}

}

public ParkingStatusModel VehicleEntry(ParkingStatusModel parkingStatusModel)

{

var parkingStatusEntity = mapper.Map<ParkingStatus>(parkingStatusModel);

var registeredVehicle = unitOfWork.RegisteredVehicleManager.GetByLicensePlate(parkingStatusEntity.LicensePlate);

if (registeredVehicle != null)

{

parkingStatusEntity.PriceRangeId = registeredVehicle.PriceRangeId;

parkingStatusEntity.RegisteredVehicleId = registeredVehicle.Id;

}

else

{

var highestPrice = unitOfWork.PriceRangeManager.GetAll().Aggregate((i1, i2) => i1.Price > i2.Price ? i1 : i2);

parkingStatusEntity.PriceRangeId = highestPrice.Id;

}

try

{

unitOfWork.ParkingStatusManager.Create(parkingStatusEntity);

unitOfWork.SaveChanges();

return mapper.Map<ParkingStatusModel>(parkingStatusEntity);

}

catch

{

throw;

}

}

}

**RepositoryContext.cs**

public class RepositoryContext : DbContext

{

public DbSet<PriceRange> PriceRanges { get; set; }

public DbSet<VehicleType> VehicleTypes { get; set; }

public DbSet<RegisteredVehicle> RegisteredVehicles { get; set; }

public DbSet<ParkingData> ParkingsData { get; set; }

public DbSet<ParkingStatus> ParkingStatuses { get; set; }

public DbSet<Payment> Payments { get; set; }

public RepositoryContext(DbContextOptions options) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

#region PriceRanges

modelBuilder.Entity<PriceRange>()

.HasData(PriceRangesSeed.Seed());

#endregion

#region VehicleTypes

modelBuilder.Entity<VehicleType>()

.HasData(VehicleTypesSeed.Seed());

#endregion

#region RegisteredVehicles

modelBuilder.Entity<RegisteredVehicle>()

.HasIndex(x => x.LicensePlate).IsUnique();

#endregion

#region ParkingsData

modelBuilder.Entity<ParkingData>()

.HasData(ParkingsDataSeed.Seed());

#endregion

#region ParkingStatus

modelBuilder.Entity<ParkingStatus>()

.HasOne(x => x.RegisteredVehicle)

.WithMany(x => x.ParkingStatuses)

.OnDelete(DeleteBehavior.ClientSetNull);

#endregion

#region Payments

modelBuilder.Entity<Payment>()

.HasOne(x => x.ParkingStatus)

.WithOne(x => x.Payment);

#endregion

}

}

1. Constraint – представляват правилата за данните в таблица [↑](#footnote-ref-1)
2. CRUD – Create, Read, Update, Delete са базовите операции за работа с база данни [↑](#footnote-ref-2)
3. LINQ - Language INtegrated Query е унифициран синтаксис за заявки интегриран в C# и Visual Basic. По този начин се премахва несъответствието между програмен код и база данни. Също така предлага единен интерфейс за работа с различни типове бази данни. [↑](#footnote-ref-3)
4. Ламбда функция – това е удобен начин за дефиниране на анонимна функция която може да бъде подавана като променлива или параметър на функция [↑](#footnote-ref-4)