ТЕХНИЧЕСКИ УНИВЕРСИТЕТ - ВАРНА

ФАКУЛТЕТ ПО ИЗЧИСЛИТЕЛНА ТЕХНИКА И АВТОМАТИЗАЦИЯ

Катедра „Софтуерни и интернет технологии“

# ДИПЛОМНА РАБОТА

## Тема:

## Проектиране и реализация на Web API за система на времеви контрол на служители

**Изготвил:** Росен Яворов Лечев

**Специалност:** Софтуерни и интернет технологии

**Факултетен номер:** 61662133

ТУ Варна, 2020 г.

Ръководител:

/проф. Тодор Ганчев/

# Съдържание

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

## 1.1. Актуалност на проблема и мотивация

За всеки един служител и работодател е важна организацията със започването и приключването на работния ден. На всяко едно работно място, за осъществяване на времеви контрол на служителите се използват различни средства, като записване на хартиен носител, записване в таблици на Excel и други, които усложняват работния процес. За целта съществуват редица решения свързани с времеви контрол на достъп, осигуряващи на служители и работодатели прозрачност и лесен отчет. Това е изключително важно, като по този начин се изгражда доверие и от двете страни. Контролираният отчет дава възможност на малки и големи фирми да проследят започването и приключването на работния ден на служителите. Този казус придобива актуална значимост в днешното така забързано ежедневие на работещия човек. В текущата разработка е представена реализация на , уеб – базиран софтуерен продукт, който предоставя тази функционалност. Мотивацията за създаването на този продукт е да се улесни и да намали времето за извършване на по – горе описаните дейности. Приложението включва само необходимата функционалност, за да не се натоварва излишно с функционалности които могат да натоварят и объркат средностатистическия работещ човек. Съществуват продуктови решения на големи компании, които включват многобройни и разнообразни функционалност, но това носи обратен ефект, защото голяма част от допълните функционалности просто не се използват в реалното ежедневие, затова предложеното решение предоставя необходимото за целите, но без да се натоварва излишно с модули които няма да бъдат използвани.

## 1.2. Обзор на съществуващи решения

### 1.2.1. CakeHR

CakeHR е идеалното решение за компании с малък до среден бизнес които не искат да използват хартия, за да управляват служителите си. Създаден е да помага на бизнесите да останат гъвкави в отношение на планирането на работния ден на служителите. CakeHR премахва нуждата от използване на таблици в Excel. Налични са и мобилни приложения за Android и iOS.

**Системата предлага следните функционалности:**

- Мобилен достъп
- Създаване на репорти
- Следене на болничните
- Следене на отпуската
- Изпращане на съобщения
- Управление на календар
- Калкулация на работа извънработно време
- Управление на задачи

Според клиенти на CakeHR предимствата са:

- Лесна работа с продукта;
- Лесна настройка на продукта;
- Перфектна опция за разрастващи се екипи;
- Отлична поддръжка;
- Безплатен пробен период в рамките на 30 дни;

Според клиенти на CakeHR недостатъците са:

- Няма модул за набиране на персонал;
- Няма интеграция с други системи;
- Не добре написана документация, хаотична;

### 1.2.2. Jibble

Jibble е лесно за използване приложение със 100% безплатен стартов план. Следене на стартовите и крайните часове на служителите използвайки телефони, таблети, уеб браузъри, Slack или MS Teams, и може да използва биометрични данни и GPS локация. Jibble е предназначен за използване от всякакви компании за отчитане на присъствие, плащане и продуктивност. Освен безплатната стартова версия Jibble предлага и 30 дена безплатен тестов период като след това цената е $1.50 на месец на клиент.

**Системата предлага следните функционалности:**

- Биометрично разпознаване
- Мобилен достъп
- Управление на заплащане
- Създаване на репорти
- Следене на изплатени суми
- Калкулация на работа извънработно време
- База от данни за служителите
- Следене на проекти

**Предимства**:

- Лесно използване;
- Отлични репорти;
- Полезни интеграции с други платформи;

**Недостатъци**:

- Понякога има проблем с времето за зареждане;

- Понякога при лоша интернет връзка на мобилно устройство не работи логването, но и не дава съобщение за проблем
- Проблем, че следващата седмица се показва като настояща
- Премахването на изпълнени таскове ги премахва от репортинг историята

### 1.2.3. Kronos Workforce Ready

Kronos Workforce Ready е предназначен за малкия до среден бизнес. Това е cloud базирано решение което предоставя нужните инструменти за следене, мениджмънт на производителността и други.

Kronos Workforce Ready помага в управлението на целия служителски lifecycle от преди наемането до пенсионирането. Продукта предлага решения за следене на времето, работни часове, и други които са лесно достъпни през интерфейса. Могат да се правят и репорти.

**Системата предлага следните функционалности:**

- База от данни за служители
- Управление на производителността
- Управление на отпуски
- Управление на компенсации
- Профили на служителите
- Управление на заплати
- Управление на подбора на персонал

**Предимства:**

- Безплатно демо;
- Добри репорти;

- Добра поддръжка;

**Недостатъци:**

- Много функционалности не работят;
- Няма обявена цена;
- Продуктът е труден за използване според клиенти;

| | **CakeHR** | **Jibble** | **Kronos Workforce Ready** |
|---|---|---|---|
| **Начална цена** | $1.50/месечно/служител | $1.50/месечно/служител | Няма упомената цена |
| **Брой потребители** | 1 - 999 | 2 - 1000+ | 1 - 499 |
| **Рейтинг** | 4.7 / 5 (123) | 4.5 / 5 (283) | 4.1 / 5 (731) |
| **Платформи** | Cloud, Windows, Mac, iOS, Android | Cloud, iOS, Android | Cloud, Windows, iOS, Android |

*Таблица 1 Сравнение между разгледаните решения*

## 1.3. Цели на дипломния прокет

- Създаване на API, работещо с база данни която съдържа информация за служители.
  - 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

## 1.4.Задачи на дипломния проект

- Избиране на система и разработване на база данни
  - проектиране на архитектурата на базата
  - използване на подходящи constraint[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]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
  - използване на автентикация и авторизация на базата на роли
    - .при заявка за логин - генериране token
    - при заявка за информация - получаване на token и неговата проверка за валидност и наличност на права за достъп
  - създаване на контролери, обработващи идващите заявки

[1] Constraint – представляват правилата за данните в таблица
[2] CRUD – Create, Read, Update, Delete са базовите операции за работа с база данни

# II. Обзор на използваните програмни средства и технологии.

## 2.1. Използвани технологии за разработка на сървърно приложение

### 2.1.1. .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.

### 2.1.2. 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.

- Microsoft.AspNetCore.Authentication.JwtBearer – библиотека позволяваща получаването на OpenID токън. Използвана за автентикация и ауторизация.

- 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

### 2.1.3. 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.

## 2.2. Използвани технологии за разработка на база от данни

### 2.2.1. 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.

### 2.2.2. SQL Server Management Studio

SQL Server Management Studio е софтуерно приложение което се използва за конфигуриране ,менажиране и администритане 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.

## 2.3. Език за програмиране на системата

### 2.3.1. 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]и ламбда[4] изрази
- лесно многонишково програмиране
- интеграция с Windows

## 2.4. 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.

[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.

[4] Ламбда функция – това е удобен начин за дефиниране на анонимна функция която може да бъде подавана като променлива или параметър на функция

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.

### 2.4.1. 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.

## 2.5. Система за управление на проекти

### 2.5.1. 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.

# III. Проектиране на системата. Описание на алгоритмите.

## 3.1. 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. Осигурена е сигурност като се използва JWT[5] стандарт.

[5] Json Web Token – това е интернет стандарт за създаване на съхранен ключ, който носи кодирана информация за изпращача.

### 3.1.1. Проектиране на база данни

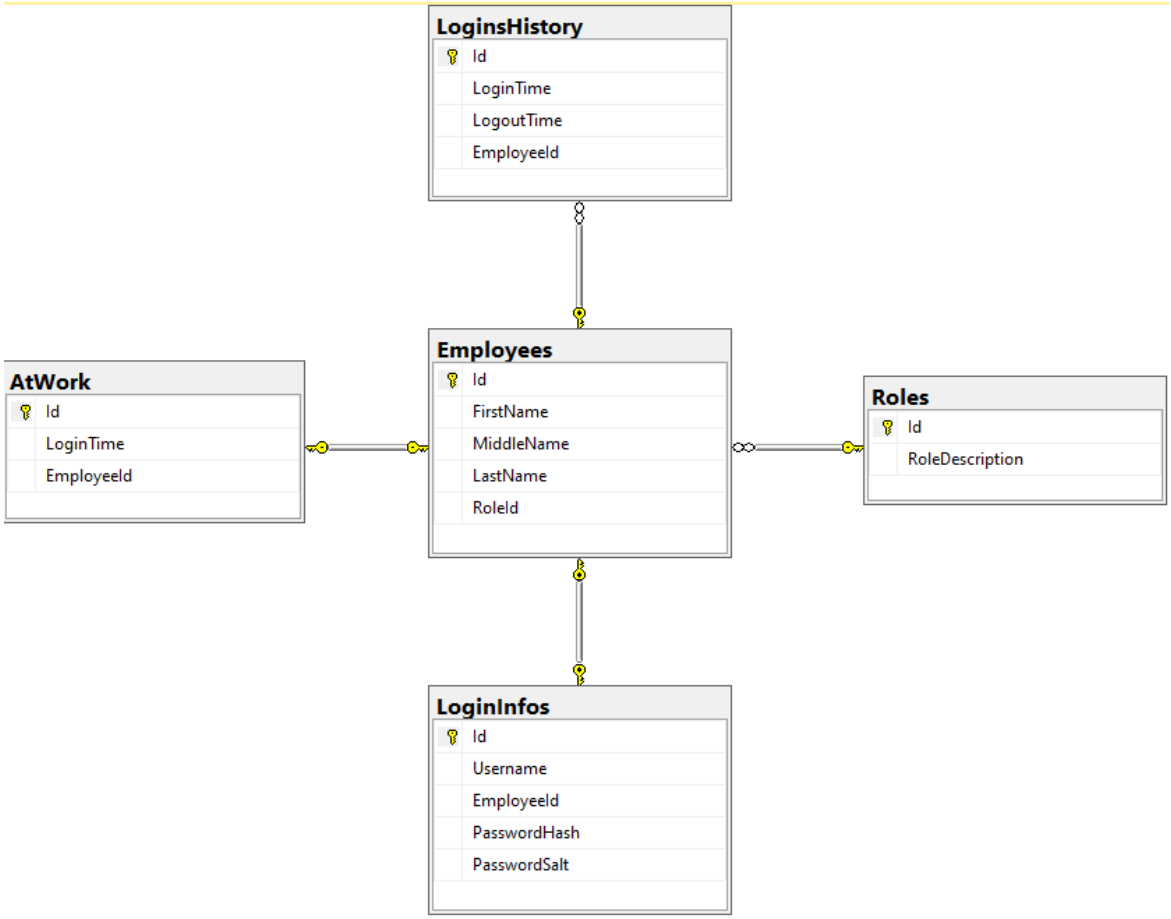
**Определяне на таблиците в базата данни**

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 личната информация на служителите
- Таблица за съхранение на име и парола за логване в системата
- Таблица за съхранение на работните позиции (роли)
- Таблица за съхранение на цялостната история на работното време за всички служители
- Таблица за съхранение на информация за служителите, които са на работа в съответния момент

Чрез диаграма ясно се представя структурата на базата данни, таблиците, техните колони и връзките между таблиците. За създаването на диаграмата е използван вграден инструмент в SSMS (SQL Server Management Studio).




*Фиг. 1 Диаграма на базата данни*

Връзките между таблиците са както следва:

- Връзката между таблици Roles и Employees е едно към много (one to many), защото много служители могат да бъдат назначени на една позиция.
- Връзката между таблици Employees и LoginInfos е едно към едно (one to one), защото един служител може да има само една информация за логване.
- Връзката между таблици Employees и AtWork е едно към едно (one to one), защото един служител може да е или на работа или не.
- Връзката между таблици Employees и LoginsHistory е едно към много (one to many), защото един служител има запис в историята за всеки работен ден.

## Структурно описание на таблиците, съставляващи базата

### Таблица „Служители“ (Employees)

Таблицата съдържа основна информация за служители и има следния формат:

| | Id | FirstName | MiddleName | LastName | RoleId |
|---|---|---|---|---|---|
| 1 | 114 | Ivan | Ivanov | Ivanov | 1 |

*Фиг. 2 Съдържание на ред от таблица Employees*

Всеки ред съдържа:

- Уникален идентификатор (id)
- Име
- Презиме
- Фамилия
- Външен ключ към позицията на служителя (роля)

### Таблица „Данни за логване“ (LoginInfos)

Таблицата съдържа информация за потребителското име и паролата на потребителите и има следния формат:

| | Id | Username | EmployeeId | PasswordHash | PasswordSalt |
|---|---|---|---|---|---|
| 1 | 100 | username84 | 98 | 0xB639EC4FE91BC81C884B9C5... | 0xA8BAC0319A8A952F46... |

*Фиг. 3 Съдържание на ред от таблица LoginInfos*

Всеки ред съдържа:

- Уникален идентификатор (id)
- Потребителско име
- Външен ключ към служител от таблица служители
- Хеш код на паролата
- Salt[6] на паролата

**Таблица „Роли“ (Roles)**

Таблицата съдържа наименованието на ролята и има формат:

| | Id | RoleDescription |
|---|---|---|
| 1 | 1 | Admin |

*Фиг. 4 Съдържание на ред от таблица Roles*

Всеки ред съдържа:

- Уникален идентификатор (id)
- Наименование на ролята

**Таблица „На работа“ (AtWork)**

Съдържа информация за време на започване и информация за служител и има следния формат:

[6] Password salt – това е произволно генерирана стойност която се комбинира с паролата преди хеширане

| | Id | LoginTime | EmployeeId |
|---|---|---|---|
| 1 | 91 | 2020-06-20 12:23:57.4780413 | 47 |

*Фиг. 5 Съдържание на ред от таблица AtWork*

Всеки ред съдържа:

- Уникален идентификатор (id)
- Дата и час на започване на работния ден
- Външен ключ към служител от таблица служители

**Таблица „Работна история“ (LoginHistory)**

Таблицата съхранява информация за начало и край на работния ден, както и информация за служител и има следния фомрат:

| Id | LoginTime | LogoutTime | EmployeeId |
|---|---|---|---|
| 52 | 2020-04-30 12:52:06.5817425 | 2020-04-30 13:06:45.4266159 | 65 |

*Фиг. 6 Съдържание на ред от таблица LoginHistory*

Всеки ред съдържа:

- Уникален идентификатор (id)
- Дата и час на започване на работния ден
- Дата и час на завършване на работния ден
- Външен ключ към служител от таблица служители

### 3.1.2. 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 пазят 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.

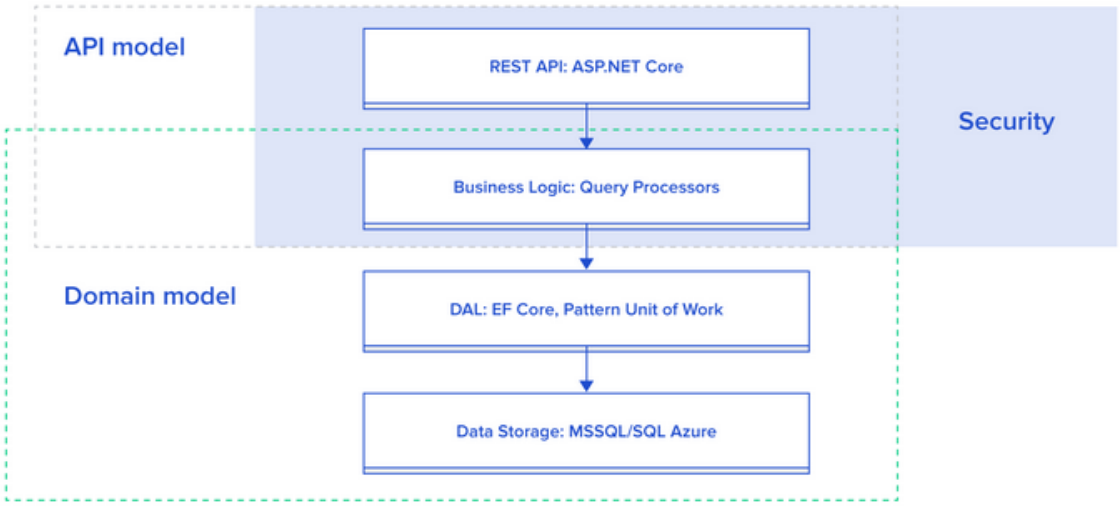
**API** (съкратено от *Application Programming Interface*) е набор от правила, които позволяват на една част от софтуерна програма да комуникира с друга част.

**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.

### 3.1.3. Добавяне на сигурност към REST API

**Структура на JWT**

За добавяне на сигурност към REST API-то е избрана защита чрез JWT (JSON Web Token). JWT е стандарт, дефиниращ компактен и самостоятелен начин за сигурно предаване на информация между страни. Информацията е предавана като JSON обект.

JWT се състои от три части, разделени със символа “.”. JWT токен изглежда по следния начин: **xxxxx.yyyyy.zzzzz**. Частите са хедър, полезен товар и сигнатура.

- Хедърът се състои от две части, които са: типът на токъна, който е JWT, и сигнатурен алгоритъм, който е използван. Пример за хедър:

  {

  "alg": "HS256",

  "typ": "JWT"

  }

- Полезният товар е втората част от токъна, която съдържа предявени искания. Пример за полезен товар:

  {

  "sub": "1234567890",

  "name": "John Doe",

  "admin": true

  }

Полезният товар е кодиран в Base64Url формат, за да формира втората част от JSON уеб токъна.

- Сигнатурната част е получена при вземане на кодирания хедър, кодирания полезен товар, тайна, алгоритъм, специфициран в хедъра, и подписване на всичко. Сигнатурата се използва, за да се верифицира, че съобщението не е било променено по пътя.

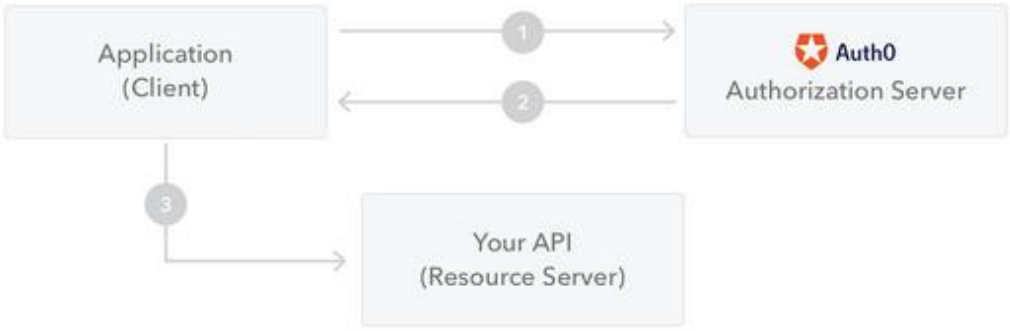
Пример за цялостен JWT токън:


*Фиг. 8 Примерен JWT токън*

**Начин на работа на JWT**

При автентикация, когато потребител успешно се впише, използвайки своите креденциали, JWT бива върнат. Токенът трябва да бъде съхранен при клиента. Този подход се различава от традиционния, при който се създава сесия на сървъра и се връща бисквитка. Всеки път, когато потребителят желае да достъпи защитен път, той трябва да изпрати JWT. Това е механизъм за автентикация без запазване на състоянието, тъй като състоянието за потребителя никога не бива съхранено в паметта на сървъра. Защитените пътища на сървъра проверяват за валиден JWT в оторизиращия хедър, и ако той е наличен, потребителят получава достъп. Тъй като JWT са самостоятелни, всичката нужна информация е в тях, чрез което се намаряват запитванията към базата на сървъра.

Следващата диаграма показва как се получава и използва JWT за достъп до ресурсите на API:




*Фиг.9 Диаграма показваща как се използва JWT*

1. Приложение или клиент изпращат поискване за авторизация към авторизиращ сървър
2. Когато е разрешен достъп, авторизиращият сървър връща токън за достъп към приложението
3. Приложението използва този токън за да достъпи защитени ресурси.

**Предимства при използване на JWT:**

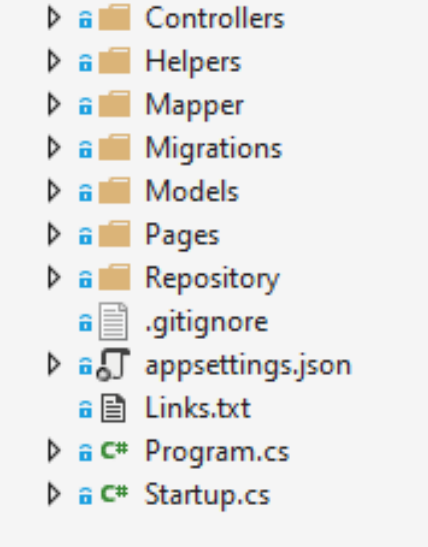
- Когато е кодиран, размерът на JWT е по малък от този на XML, което прави JWT добър избор за използване в среди с HTML и HTTP.
- JWT може да използва публичен или частен ключ под формата на сертификат.
- JSON парсерите са често срещани, и мапват директно към обекти което прави работата с JWT лесно.

# IV. Реализация на приложението. Основни програмни модули.

## 4.1. 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 приложението*

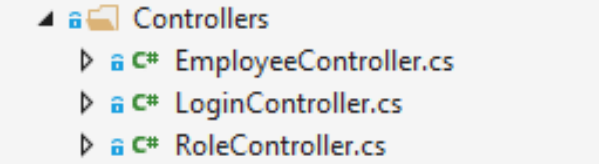
### 4.1.1. Папка Контролери (Controllers)

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*

#### 4.1.1.1. Контролер за служители (EmployeeController)

Класът EmployeeController съдържа действията които се използват за обработване на информация за служителите.

Действията са:

- Authenticate([FromBody]LoginModel loginModel) – това действие се използва за авторизация на потребител при опит за логване в системата. Методът на заявката е POST и се извиква от адрес /api/Employee/Authenticate. Данните нужни за логване се намират в body частта на заявката.

  ```
  {
      "Username" : "username",
      "Password" : "password"
  }
  ```

- При неуспешно намиране на потребител се връща отговор с код 400 Bad Request и съобщение за грешно име или парола. При успешно намиране на потребител се генерира JWT токън и към клиента се връща намерения потребител заедно с токъна за автентикация и код 200 OK.

- Използва се атрибут [AllowAnonymous] за да може да се изпраща заявка без нуждата от JWT токън за автентикация.

- GetAllEmployees() – това действие се използва за извличане и връщане на всички служители от базата. Методът на заявката е GET и се извиква от адрес /api/Employee/GetAllEmployees. Връща се колекция с всички служители и код 200 OK.

- Използва се атрибут за оторизация [Authorize(Roles = "Admin")], като по този начин единствено заявка притежаващи токън, който указва че ролята е администраторска, ще бъде допусната за изпълнение. Ако ролята не е администраторска се връща отговор с код 405 Not Allowed.

- GetById(int id) – това действие се използва за извличане и връщане на служител по зададен уникален ключ (id) на служител. Методът на заявката е GET и се извиква от адрес /api/Employee/GetById/?id=X,

като X е съответния уникален идентификатор. При не намиране на служител се връща отговор с код 404 Not Found. Създадено е ограничение, което не позволява на не администраторски потребител да извлича информация за друг потребител по уникален идентификатор, освен за себе си. Ако е направен такъв опит се връща отговор с код 403 Forbidden. При успешно изпълнение на заявката се връща служителят и код 200 OK.

- CreateEmployee([FromBody] CreateEmployeeModel createEmployeeModel) – това действие се използва за създаването и записването на нов служител в системата. Методът на заявката е POST и се извиква от адрес /api/Employee/CreateEmployee. Данните нужни за създаване на служител се намират в body частта на заявката и са:

```
{
        "FirstName" : "Ivan",
        "MiddleName" : "Ivanov",
        "LastName" : "Ivanov",
        "Username" : "ivan2341",
        "Password" : "Ivan_231",
        "RoleDescription" : "User"
}
```

- Съществува ограничение за уникалност на потребителското име. При успешно създаване се връща създадения потребител и код 200 OK. При неуспешно изпълнение се връща съобщение за грешка.

- Използва се атрибут за оторизация [Authorize(Roles = "Admin")], като по този начин единствено заявка притежаващи токън, който указва че ролята е администраторска, ще бъде допусната за изпълнение. Ако ролята не е администраторска се връща отговор с код 405 Not Allowed.

- DeleteEmployee(int id) – действието се използва за изтриване на потребител от системата. Методът на заявката е DELETE и се извиква от адрес /api/Employee/DeleteEmployee?id=X, където X е уникалният идентификатор на служителя, който трябва да бъде изтрит. При успешно изпълнение на заявката се връща отговор с код 200 OK. . При неуспешно изпълнение се връща съобщение за грешка.

- Използва се атрибут за оторизация [Authorize(Roles = "Admin")], като по този начин единствено заявка притежаващи токън, който указва че ролята е администраторска, ще бъде допусната за изпълнение. Ако ролята не е администраторска се връща отговор с код 405 Not Allowed.

- UpdateEmployee([FromBody] UpdateEmployeeModel employeeModel) – използва се обновяване на информацията за потребител. Методът на заявката е PUT и се извиква от адрес /api/Employee/UpdateEmployee като нужните данни за изпълнение на заявката се намират в body частта на заявката и са:

```
{
        "EmployeeId" : 4,
        "FirstName" : "Ivan",
        "MiddleName" : "Ivanov",
        "LastName" : "Ivanov",
        "Password" : "Ivan_1231",
        "RoleDescription"  : "Admin"
}
```

- Действието позволява да се ъпдейтват само данните които се искат, като единственото задължително поле е уникалният идентификатор EmployeeId. При успешно изпълнение на заявката се връща отговор с код 200 OK. При неуспешно изпълнение се връща съобщение за грешка.

- UpdateEmployeePassword([FromBody] UpdateEmployeeModel employeeModel) – методът се използва за обновяване на паролата на потребител. Методът на заявката е PUT и се извиква на адрес /api/Employee/UpdateEmployeePassword, като нужните данни за изпълнението се намират в body частта на заявката и са:

  ```
  {
      "EmployeeId" : 4,
      "Password" : "Ivan_12314",
  }
  ```

- Създадено е ограничение, което позволява служител да променя само собствената си парола, както и администратора да може да променя паролите на всички служители. При успешно изпълнение се връща отговор с код 200 OK. При неуспешно изпълнение се връща съобщение за грешка.

#### 4.1.1.2. **Контролер за логванията (LoginController)**

Класът LoginController съдържа действия, които се използват за обработване на информацията за логване.

Действията са:

- GetAllRowsAtWork() – действието се използва за извличане и връщане на всички служители които са отчели, че са на работа в съответния момент. Методът на заявката е GET и се извиква от адрес /api/Login/ GetAllRowsAtWork. Връща се колекция от служителите, които са на работа, техните часове на започване на работа и код 200 OK.

- Използва се атрибут за оторизация [Authorize(Roles = "Admin")], като по този начин единствено заявка притежаващи токън, който указва че ролята е администраторска, ще бъде допусната за изпълнение. Ако ролята не е администраторска се връща отговор с код 405 Not Allowed.

- GetAtWorkByEmployeeId(int id) – използва се за извличане на служител, който е на работа в момента по уникалния му идентификатор. Методът на заявката е GET и се извиква от адрес /api/Login/ GetAtWorkByEmployeeId?id=X, където X е уникалният идентификатор на търсения служител. Съществува ограничение служител да може да търси само себе си, както и администратор да може да търси всички служители. При успешно намиране на служител, който е на работа се връща информацията за него, както и начален час на работния ден и код 200 OK. Ако съответния човек не е намерен се връща код 404 Not Found.

- LoginAtWork() – използва се за отчитане на началото на работния ден. Методът на заявката е PUT и се извиква от адрес /api/Login/ LoginAtWork. При успешно отчитане се връща информацията за началния час на работа и код 200 OK. При грешка се връща съобщение за грешка, а ако служителят е вече на работа и се опита да се отчете пак се връща код 400 Bad Request.

- LogoutFromWork() – действието се използва за отчитане на край на работния ден. Методът на заявката е PUT и се извиква от адрес /api/Login/ LogoutFromWork. След отчитане на край на работния ден, служителят вече не се води на работа. При успешно изпълнение се връща код 200 OK. Ако служител се опита да отчете край на работния ден, а не е отчел начало ще се върне код 400 Bad Request.

- GetLoginHistory() – действието се използва за извличане и връщане на цялостната история на работа на всички служители. Методът на заявката е GET и се извиква от адрес /api/Login/ GetLoginHistory. При успешно изпълнение се връща колекция с данни за служителите както и начало и край на съответния работен ден.

- Използва се атрибут за оторизация [Authorize(Roles = "Admin")], като по този начин единствено заявка притежаващи токън, който указва че ролята е администраторска, ще бъде допусната за изпълнение. Ако ролята не е администраторска се връща отговор с код 405 Not Allowed.

- GetLoginHistoryByEmployeeId(int id) – действието се използва за извличане и връщане на цялостната история на работа на служител като се използва неговия уникален идентификатор. Методът на заявката е GET като се извиква от адрес /api/Login/ GetLoginHistoryByEmployeeId?id=X, където X е уникалният идентификатор на търсения служител. При успешно изпълнение се връща колекция с данните за служителя, както и начало и край на съответните работни дни и код 200 OK. Съществува ограничение, че служител може да търси само собствената си история за работното време, както и администратор може да търси за всички служители.

#### 4.1.1.3. **Контролер за позициите (RoleController)**

Класът RoleController съдържа действия които се използват за обработване на данните за позициите.

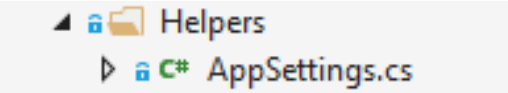
Действията са:

- GetAll() – действието се използва за извличане на всички позиции. Методът на заявката е GET и се извиква от адрес /api/Role/GetAll. При успешно изпълнение се връща колекция от позициите както и код 200 OK. При възникнала грешка се връща съобщение за грешка.

- Използва се атрибут за оторизация [Authorize(Roles = "Admin")], като по този начин единствено заявка притежаващи токън, който указва че ролята е администраторска, ще бъде допусната за изпълнение. Ако ролята не е администраторска се връща отговор с код 405 Not Allowed.

- AddRole(string RoleDescription) – действието се използва да добавяне на нова позиция. Методът на заявката е POST и се извиква от адрес /api/Role/AddRole?RoleDescription=AAA, където AAA е наименованието на позицията за добавяне. При успешно изпълнение се връща код 200 OK. При възникнала грешка се връща съобщение за грешка.

### 4.1.2. Папка с помощни файлове (Helpers)



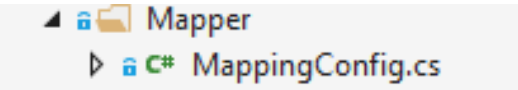

*Фиг. 12 Съдържание на папка Helpers*

Класът AppSettings съдържа променливи дефинирани във файла appsettings.json. Използват се за достъп до настройките на приложението чрез обекти, които се инжектират в класовете като използват вградената функционалност на .Net Core, dependency injection (DI). Мапването на конфигурационните секции към класовете се извърша в конфигурационния метод (ConfigureServices) на Startup.cs.

Класът AppSettings съдържа една променлива – Secret, която се използва при подписване и верифициране на JWT токъните и може да бъде всякакъв стринг.

### 4.1.3. 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:

- EmployeeMap() – метод който указва как да се мапнат данните от обект Employee към обект EmployeeModel
- RoleMap()– метод който оказва как да се мапнат данните от обект Role към обект RoleModel
- LoginInfoMap()– метод който оказва как да се мапнат данните от обект LoginInfo към обект LoginModel
- CreateEmployeeMap()– метод който оказва как да се мапнат данните от обект CreateEmployeeModel към обект Employee
- CreateEmployeeLoginMap()– метод който оказва как да се мапнат данните от обект CreateEmployeeModel към обект LoginInfo
- UpdateEmployeeMap()– метод който оказва как да се мапнат данните от обект UpdateEmployeeModel към обект Employee
- UpdateEmployeeLogin()– метод който оказва как да се мапнат данните от обект UpdateEmployeeModel към обект LoginInfo
- AtWorkMap()– метод който оказва как да се мапнат данните от обект AtWork към обект AtWorkModel
- LoginHistoryMap()– метод който оказва как да се мапнат данните от обект LoginHistory към обект LoginHistoryModel

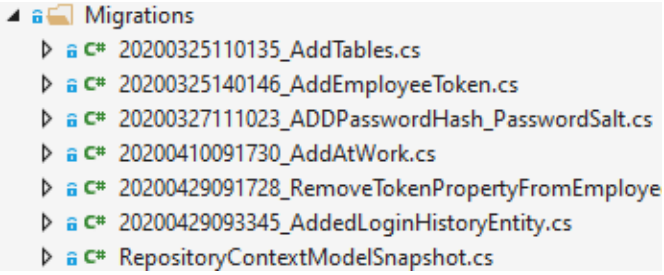
### 4.1.4. Папка с миграции (Migrations)

Entity Framework (EF) е Object-relation mapper. Използва се за създаване, упдейтване, 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.



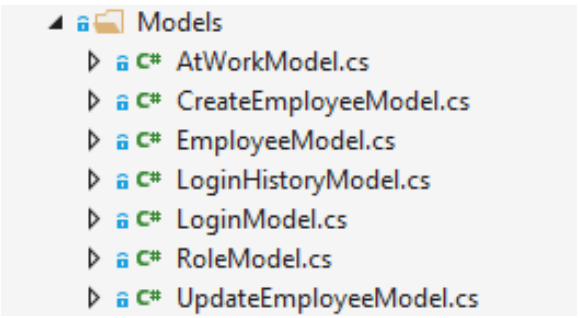

*Фиг. 14 Съдържание на папка Migrations*

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.

### 4.1.5. 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*

#### 4.1.5.1. Файл AtWorkModel

Този файл съдържа променливи за:

- Id от тип int за уникален идентификатор
- LoginTime от тип DateTime за време на логване в системата
- Employee от тип EmployeeModel, съдържащ данните за съответния служител

За класа се използва атрибут [Serializable], за да окаже, че атрибутът може да бъде сериализиран в JSON.

#### 4.1.5.2. Файл CreateEmployeeModel

Този файл съдържа променливи за:

- FirstName от тип string за име на слжител
- MiddleName от тип string за бащино име на служител
- LastName от тип string за фамилия на служител
- Username от тип string за потребителско име на служител
- Password от тип string за парола на служител
- RoleDescription от тип string за позицията която ще изпълнява служителят

  Всички променливи използват атрибут [Required], което оказва, че задължително всички полета трябва да имат стойности, за да бъде създаден служител

#### 4.1.5.3. Файл EmployeeModel

Този файл съдържа променливи за:

- Id от тип int за уникален идентификатор

- FirstName от тип string за име на слжител
- MiddleName от тип string за бащино име на служител
- LastName от тип string за фамилия на служител
- Role от тип RoleModel за позицията заемана от служител
- Token от тип string, генерираният JWT токън за автентикация

За класа се използва атрибут [Serializable], за да окаже, че атрибутът може да бъде сериализиран в JSON.

#### 4.1.5.4. Файл LoginHistotyModel

Този файл съдържа променливи за:

- Id от тип int за уникален идентификатор
- LoginTime от тип DateTime за време на логване в системата
- LogoutTime от тип DateTime за време на излизане от системата
- Employee от тип EmployeeModel, съдържащ данните за съответния служител

За класа се използва атрибут [Serializable], за да окаже, че атрибутът може да бъде сериализиран в JSON.

#### 4.1.5.5. Файл LoginModel

Този файл съдържа променливи за:

- Id от тип int за уникален идентификатор
- Username от тип string за потребителско име на служител
- Password от тип string за парола на служител
- PasswordHash от тип byte[] за хешираната парола
- PasswordSalt от тип byte[] за salt на парола

- Employee от тип EmployeeModel, съдържащ данните за съответния служител

За класа се използва атрибут [Serializable], за да окаже, че атрибутът може да бъде сериализиран в JSON.

#### 4.1.5.6. Файл RoleModel

Този файл съдържа променливи за:

- Id от тип int за уникален идентификатор
- RoleDescription от тип string за позицията която ще изпълнява служителят

За класа се използва атрибут [Serializable], за да окаже, че атрибутът може да бъде сериализиран в JSON.

#### 4.1.5.7. Файл UpdateEmployeeModel

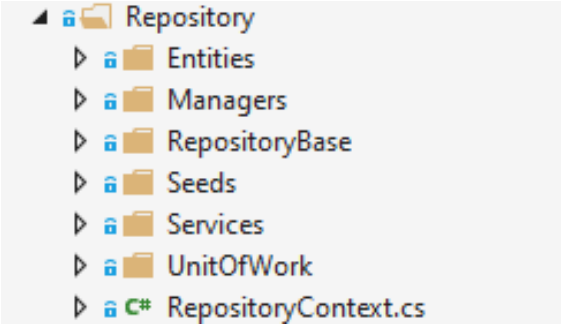
Този файл съдържа променливи за:

- Id от тип int за уникален идентификатор
    - Използва се атрибут [Required]
- FirstName от тип string за име на слжител
- MiddleName от тип string за бащино име на служител
- LastName от тип string за фамилия на служител
- Password от тип string за парола на служител
- RoleDescription от тип string за позицията която ще изпълнява служителят

За класа се използва атрибут [Serializable], за да окаже, че атрибутът може да бъде сериализиран в JSON.

### 4.1.6. Папка Repository

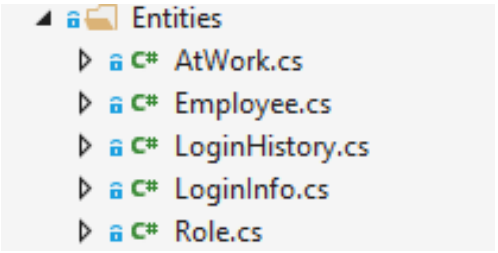
Тази папка съдържа всички нужни компоненти за работа с базата данни. Структурата е:




*Фиг. 16 Съдържание на папка Repository*

#### 4.1.6.1. 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*

##### 4.1.6.1.1. Файл AtWork

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
- LoginTime от тип DateTime за време за започване на работа
    - Използва се атрибут:
        - [Required]
- Employee от тип Employee използван за външен ключ (foreign key) към таблица Employees
    - Използва се атрибут:
        - [ForeignKey(nameof(EmployeeId))] оказващ, че променливата е външен ключ с Id EmployeeId
- EmployeeId от тип int за съхранение на Id от връзката с таблица Employees

Класът има атрибути:

- [Table("AtWork")] оказващ, че името на таблицата в базата да е AtWork.

#### 4.1.6.1.2. Файл Employee

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
- FirstName от тип string за име на служителя
    - Използва атрибути:
        - [MaxLength(64)] оказващ максимална дължина от 64 символа
        - [Required]
- MiddleName от тип string за бащино име на служител
    - Използва атрибути:
        - [MaxLength(64)] оказващ максимална дължина от 64 символа
        - [Required]
- LastName от тип string за фамилия на служител
    - Използва атрибути:
        - [MaxLength(64)] оказващ максимална дължина от 64 символа
        - [Required]
- LoginInfo от тип LoginInfo оказваща връзка с таблица LoginInfos
- AtWork от тип AtWork оказваща връзка с таблица AtWork
- LoginsHistory от тип IEnumerable<LoginHistory> (колекция) оказваща връзка с таблица LoginsHistory
- Role от тип Role за използван за външен ключ (foreign key) към таблица Roles
    - Използва се атрибут:
        - [ForeignKey(nameof(RoleId))] оказващ, че променливата е външен ключ с Id RoleId
- RoleId от тип int за съхранение на Id от връзката с таблица Roles
    - Използва се атрибут:
        - [Required]

Класът има атрибут:

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

#### 4.1.6.1.3. Файл LoginHistory

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
- LoginTime от тип DateTime за време за започване на работа
    - Използва се атрибут:
        - [Required]
- LogoutTIme от тип DateTime за време за приключване на работа
    - Използва се атрибут:
        - [Required]
- Employee от тип Employee използван за външен ключ (foreign key) към таблица Employees
    - Използва се атрибут:
        - [ForeignKey(nameof(EmployeeId))] оказващ, че променливата е външен ключ с Id EmployeeId
- EmployeeId от тип int за съхранение на Id от връзката с таблица Employees
    - Използва атрибут:
        - [Required]

Класът има атрибут:

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

#### 4.1.6.1.4. Файл LoginInfo

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
- Username от тип string за потребителското име на служител
    - Използва атрибути:
        - [MaxLength(64)] оказващ максимална дължина от 64 символа
        - [Required]
- PasswordHash от тип byte[] за хешираната парола
    - Използва атрибут:
        - [Required]
- PasswordSalt от тип byte[] за salt на парола
    - Използва атрибут:
        - [Required]
- Employee от тип Employee използван за външен ключ (foreign key) към таблица Employees
    - Използва се атрибут:

        - [ForeignKey(nameof(EmployeeId))] оказващ, че променливата е външен ключ с Id EmployeeId
- EmployeeId от тип int за съхранение на Id от връзката с таблица Employees
    - Използва атрибут:
        - [Required]

Класът има атрибут:

- [Table("LoginInfos")] оказващ, че името на таблицата в базата да е LoginInfos.

#### 4.1.6.1.5. Файл Role

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

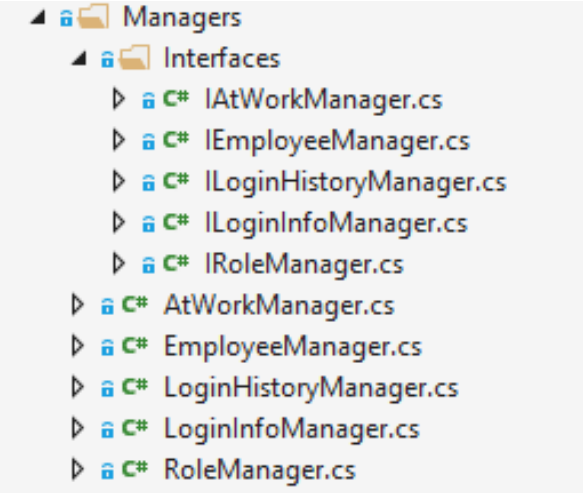
- Id от тип int за уникален идентификатор
    - Използват се атрибути:
        - [Key] оказващ, че променливата е главен ключ (primary key)
        - [DatabaseGenerated(DatabaseGeneratedOption.Identity)] оказва, че на променливата ще бъде генериран поредно уникално число
- RoleDescriptionот тип string за име на позиция
    - Използва атрибути:
        - [MaxLength(64)] оказващ максимална дължина от 64 символа
        - [Required]
- Employees от тип IEnumerable<Employee> (колекция) оказваща връзка с таблица Employee

Класът има атрибут:

- [Table("Roles")] оказващ, че името на таблицата в базата да е Roles.

#### 4.1.6.2. 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 е 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.

#### 4.1.6.2.1. IAtWorkManager и AtWorkManager

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

- GetAllWithEmployees() – този метод се използва, за да се извлекат от базата всички редове от таблица AtWork, като се използва LINQ и left join, за да се вземе и информацията за служител. Данните се връщат в колекция IEnumerable<AtWork>.
- GetByEmployeeId(int employeeId) – този метод се използва за намиране на конкретен ред от таблицата за служители, които са на работа в съответния момент, като се търси по уникалния идентификатор на служителя. За извличане на служителите се използва методът GetAllWithEmployees() след което се използва ламбда функция за намирането на конкретния служител. Данните се връщат с тип AtWork.
-

#### 4.1.6.2.2. IEmployeeManager и EmployeeManager

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

- GetAllWithRoles() - този метод се използва, за да се извлекат от базата всички редове от таблица Employees, като се използва LINQ и left join, за да се вземе и информацията за ролята. Данните се връщат в колекция IEnumerable<Employee>.

- GetByIdWithRole(int id) - този метод се използва за намиране на конкретен ред от таблицата за служители, като се търси по уникалния идентификатор на служителя. За извличане на служителите се използва методът GetAllWithRoles () след което се използва ламбда функция за намирането на конкретния служител. Данните се връщат с тип Employee.

#### 4.1.6.2.3. ILoginHistoryManager и LoginHistoryManager

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

- GetAllByEmployeeId(int employeeId) - този метод се използва за извличане на всички редове от таблица за LoginHistory, като се търси по уникалния идентификатор на служителя. Използва се left join, за да се вземе информацията за служител и неговата роля. Данните се връщат в колекция IEnumerable<LoginHistory>.
- GetAllWithEmployees() - този метод се използва, за да се извлекат от базата всички редове от таблица LoginHistory, като се използва LINQ и left join, за да се вземе и информацията за служителя и ролята. Данните се връщат в колекция IEnumerable<LoginHistory>.

#### 4.1.6.2.4. ILoginInfoManager и LoginInfoManager

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

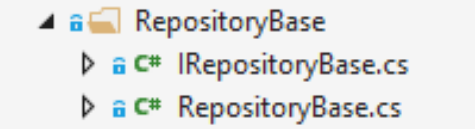
- GetAllWIthEmployee()() – този метод се използва, за да се извлекат от базата всички редове от таблица LoginInfos, като се използва LINQ и left join, за да се вземе и информацията за служител. Данните се връщат в колекция IEnumerable<LoginInfo>.

#### 4.1.6.2.5. IRoleManager и RoleManager

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

### 4.1.6.3. 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

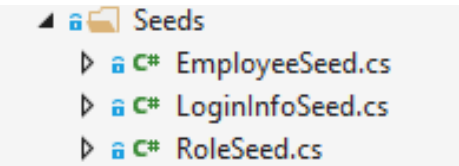
#### 4.1.6.3.1. IRepositoryBase и RepositoryBase

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 висчки редове от съответна таблица

### 4.1.6.4. Папка Seeds

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



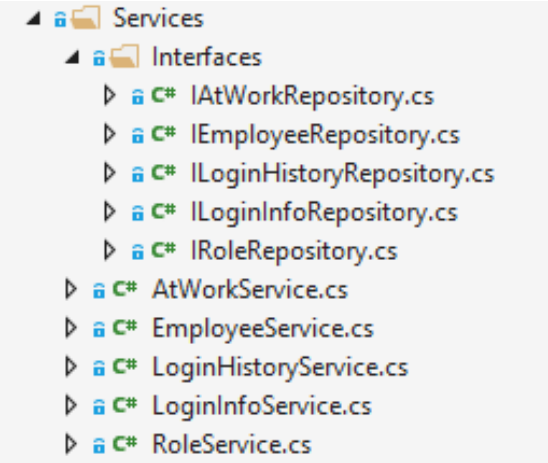

*Фиг. 20 Съдържание на папка Seeds*

- Файл EmployeeSeed – съдържа метод, който връща масив от тип Employee[], данните от който да ще се запишат в таблица Employees при създаването и.
- Файл LoginInfoSeed – съдържа метод, който връща масив от тип LoginInfo[], данните от който да ще се запишат в таблица LoginInfo при създаването и.
- Файл RoleSeed - съдържа метод, който връща масив от тип Role [], данните от който да ще се запишат в таблица Roles при създаването и.

#### 4.1.6.5. 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*

##### 4.1.6.5.1. 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.

#### 4.1.6.5.2. 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 слуцител 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.

#### 4.1.6.5.3. 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.

#### 4.1.6.5.4. 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

#### 4.1.6.5.5. IRoleRepository и IRoleService

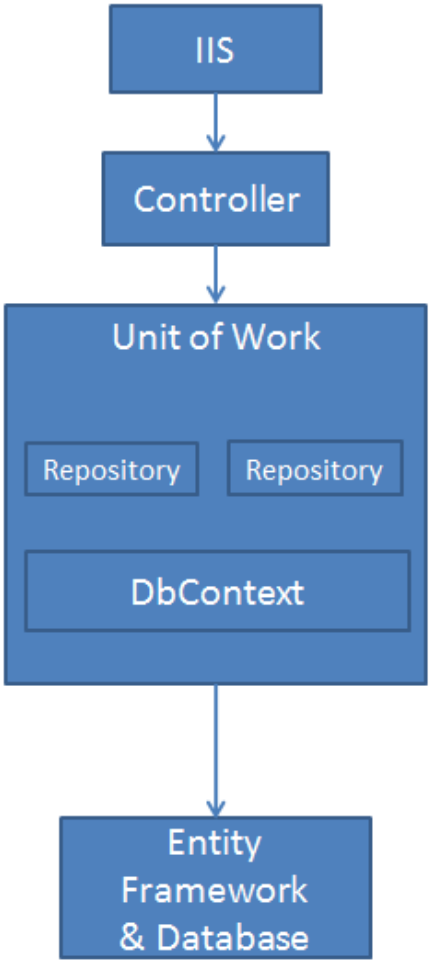
В тези файлове е описана бизнес логиката за работа с таблица Roles. Методите са

- 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>.

### 4.1.6.6. Папка UnitOfWork

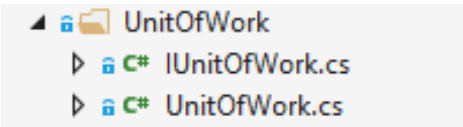
Unit of Work се разглежда като една транзакция, която включва множество операции insert, update, delete. Ако не се използва този патърн, всяко ентити ще използва собствен контекст за базата данни, което може да представлява проблем. Ако се изпълни операция, която включва две или повече ентитита ,и при 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.

#### 4.1.6.7. 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.

```
#region Employee
modelBuilder.Entity<Employee>()
    .HasOne(a => a.LoginInfo)
    .WithOne(b => b.Employee)
    .HasForeignKey<LoginInfo>(b => b.EmployeeId);

modelBuilder.Entity<Employee>()
    .HasOne(a => a.AtWork)
    .WithOne(b => b.Employee)
    .HasForeignKey<AtWork>(b => b.EmployeeId);

modelBuilder.Entity<Employee>()
    .HasData(EmployeeSeed.Seed());

modelBuilder.Entity<Employee>()
    .HasMany(a => a.LoginsHistory)
    .WithOne(b => b.Employee);
#endregion
```

*Фиг. 24 Задаване на допълнителни условия за създаването на таблица Employees във файл RepositoryContext*

### 4.1.7. 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, JWT авентикацията, AutoMapper, и connection string за връзка с базата данни.
- Метод Configure се използва за конигуриране 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.

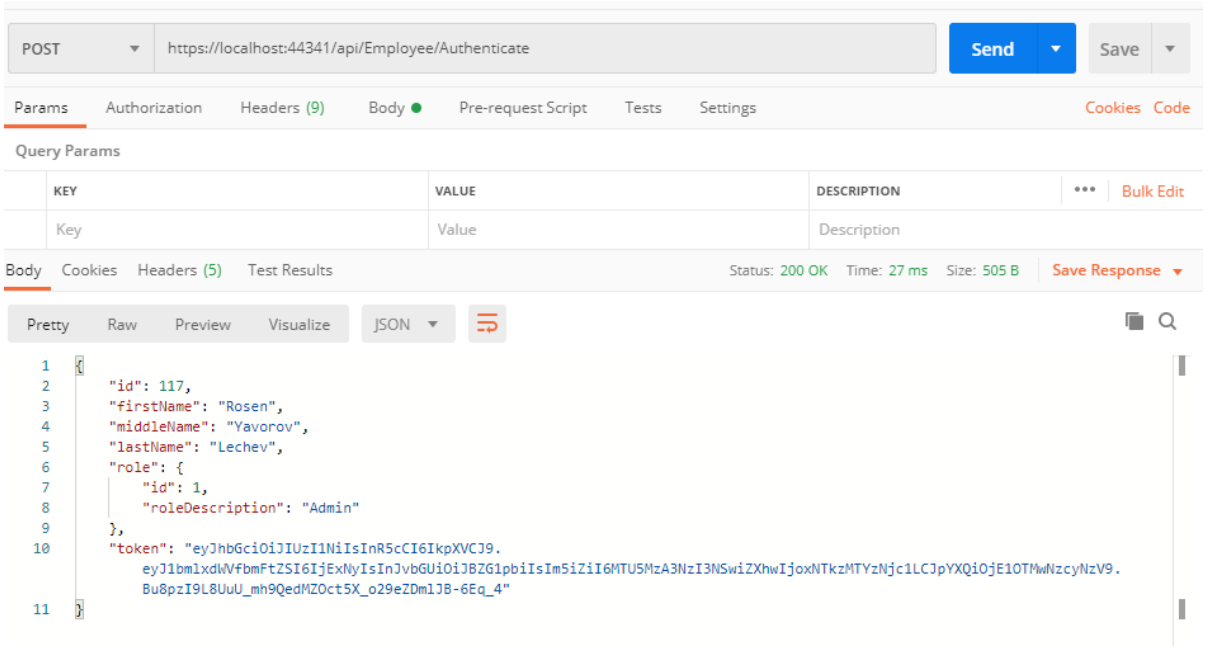
# V. Тестване на системата, изводи и заключения

## 5.1. Тестване на работоспособността на системата

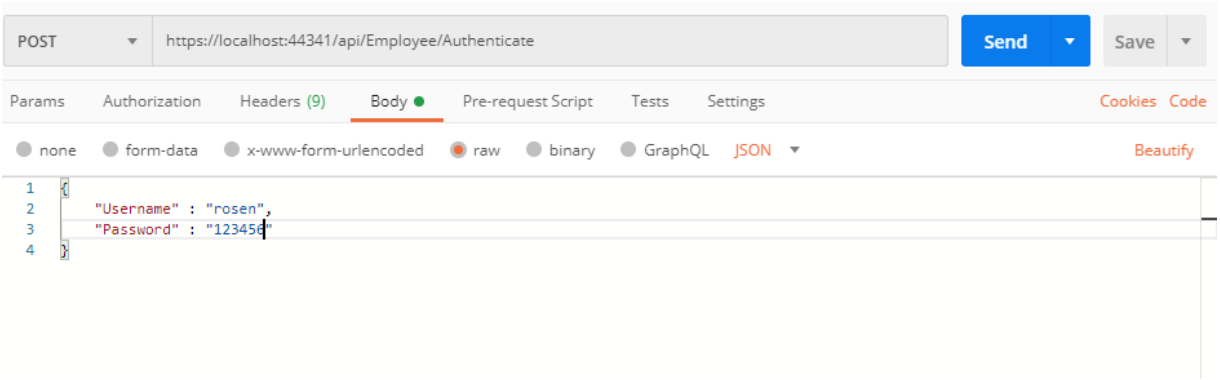
За изпробване на работоспособността на системата се използва програмата Postman. Postman е инструмент за правене на HTTP заявки и добавяне на автоматизирани тестове към техните резултати Поради тези причини, Postman може да се използва успешно както от отделни членове на екипа, така и при колаборация 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:

- Тестване на автентикацията на служител.

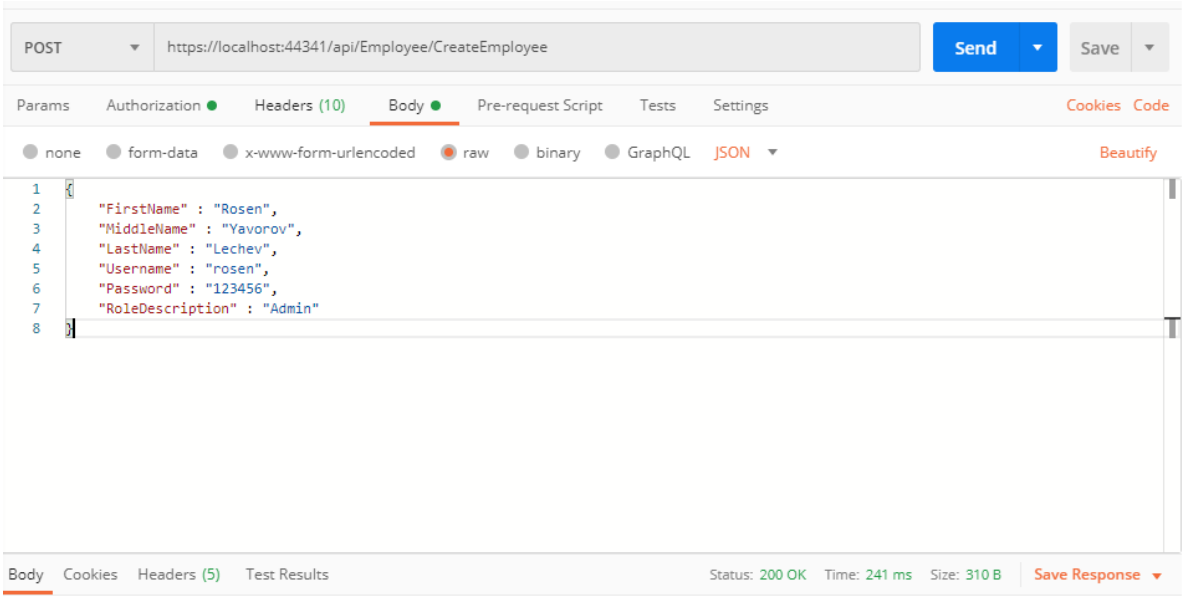



*Фиг. 25 Отговор на изпратената заявка за автентикация*

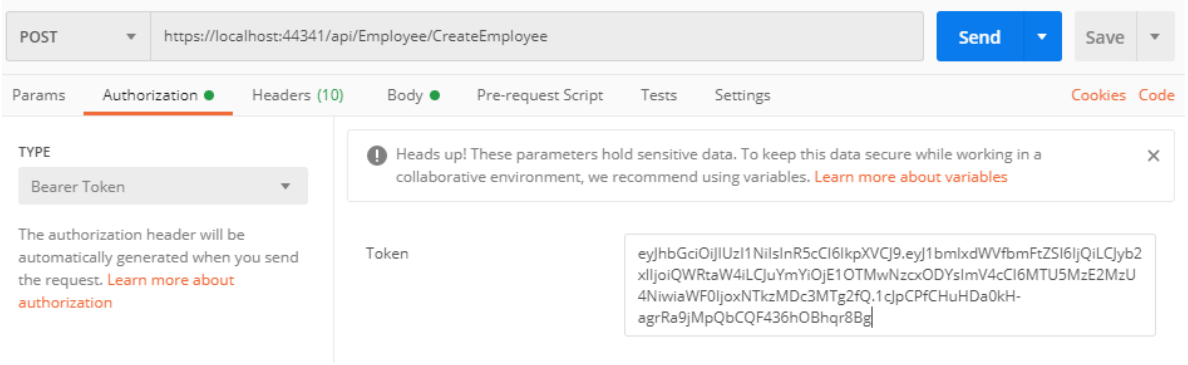



*Фиг. 26 Информацията изпратена в body на заявката*

- Създаване на служител. Използва се токън за оторизация

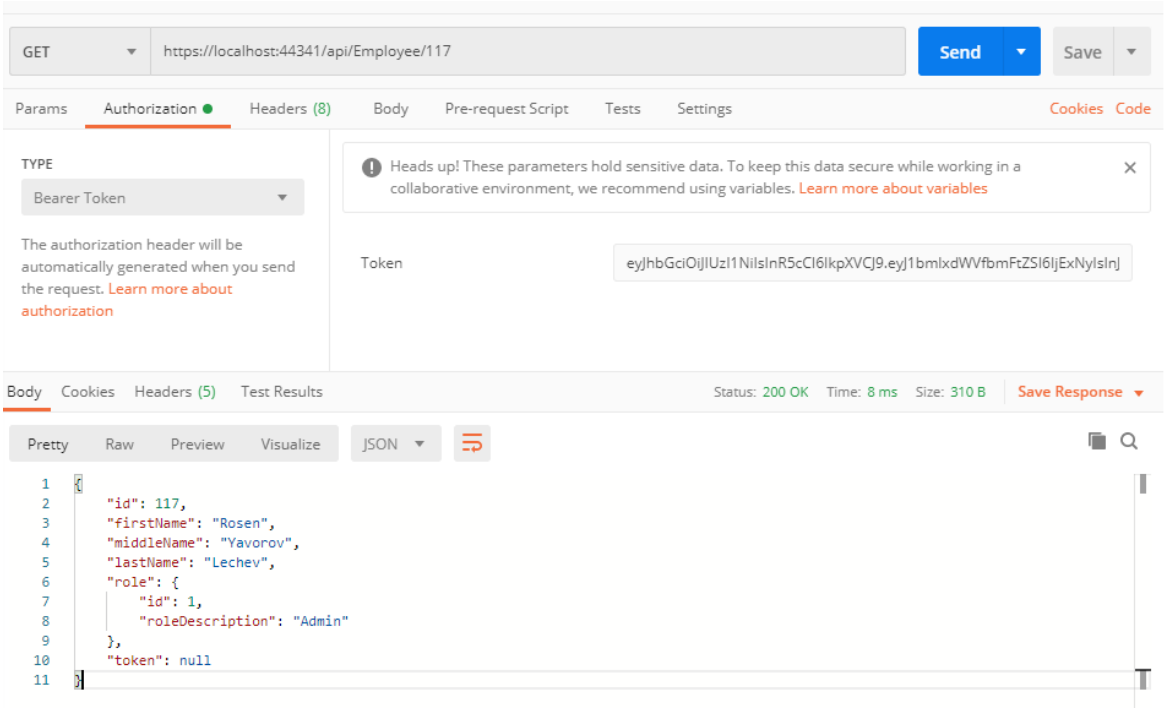



*Фиг. 27 Информацията изпратена в body на заявката за създаване на служител*



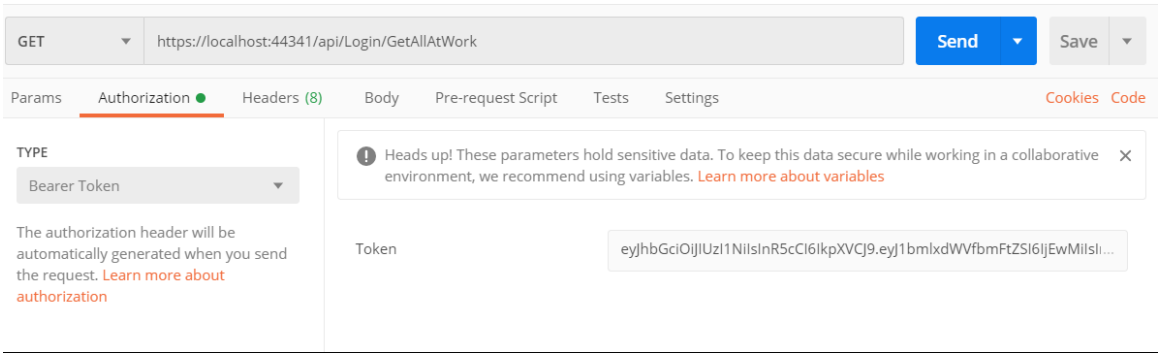

*Фиг. 28 Използване на токън за оторизация*

- Извличане на служител по уникалният му идентификатор. Използва се токън за оторизация.




*Фиг. 29 Отговор от заявка за извличане на служител по ID*

- Извличане на данните са всички служители на работа в момента. Използва се токън за оторизация




*Фиг. 30 Използване на токън за оторизация при извличане на всички служители*


*Фиг. 31 Отговор от заявка за извличане на всички служители*

## 5.2. Изводи и заключения

Разработената дипломна работа на тема „Проектиране и реализация на Web Api за система за времеви контрол на служители“ 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 създават екипи и да се задават техните ръководители.
- Създаване на функционалност, която позволява да се задават дните за отпуска и те да се изпращат към началника на служителя за одобрение.

- Създаване на втори начин за логване в системата, като се използва лицево разпознаване.
- Разширяване на системата с лицево разпознаване за използване в офис сградата, за оторизация при влизане в отделни помещения на база на заемата позиция, както се използва в самото приложение.

# 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
2. 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


### MS SQL Server


4. https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15


### Entity Framework Core


5. https://docs.microsoft.com/en-us/ef/core/get-started/?tabs=netcore-cli


### JWT


6. https://devblogs.microsoft.com/aspnet/jwt-validation-and-authorization-in-asp-net-core/
7. https://jwt.io/introduction/


### Auto Mapper


8. https://automapper.org/


### Unit of Work и Repository патърн


9. 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

### EmployeeController.cs

```
[Authorize]
[ApiController]
[Route("api/[controller]")]
public class EmployeeController : Controller
{
    private readonly IEmployeeRepository employeeRepository;

    public EmployeeController(IEmployeeRepository employeeRepository)
    {
        this.employeeRepository = employeeRepository;
    }

    [AllowAnonymous]
    [HttpPost]
    [Route("Authenticate")]
    public IActionResult Authenticate([FromBody]LoginModel loginModel)
    {
        if (loginModel.Username == null && loginModel.Password == null)
            return NotFound();

        EmployeeModel employee = employeeRepository.Authenticate(loginModel.Username,
loginModel.Password);

        if (employee == null)
            return BadRequest(new { message = "Username or password is incorrect" });

        return Ok(employee);
    }

    [Authorize(Roles = "Admin")]
    [HttpGet]
    [Route("GetAllEmployees")]
    public IActionResult GetAllEmployees()
    {
        IEnumerable<EmployeeModel> employees = employeeRepository.GetAllEmployees();

        return Ok(employees);
    }

    [HttpGet("{id}")]
    public IActionResult GetById(int id)
    {
        EmployeeModel employee = employeeRepository.GetById(id);

        if (employee == null)
            return NotFound();
```

```
            int currentUserId = int.Parse(User.Identity.Name);
            if (id != currentUserId && !User.IsInRole("Admin"))
            {
                return Forbid();
            }

            return Ok(employee);
        }

        [Authorize(Roles = "Admin")]
        [HttpPost("CreateEmployee")]
        public IActionResult CreateEmployee([FromBody] CreateEmployeeModel
createEmployeeModel)
        {
            string message = "";
            EmployeeModel employee =
employeeRepository.CreateEmployee(createEmployeeModel, ref message);

            if (employee == null)
            {
                return Problem(message);
            }

            return Ok(employee);
        }

        [Authorize(Roles = "Admin")]
        [HttpDelete]
        [Route("DeleteEmployee")]
        public IActionResult DeleteEmployee(int id)
        {
            try
            {
                employeeRepository.DeleteEmployee(id);
                return Ok();
            }
            catch (Exception)
            {
                return Problem();
            }
        }


        [Authorize(Roles = "Admin")]
        [HttpPut("UpdateEmployee")]
        public IActionResult UpdateEmployee([FromBody] UpdateEmployeeModel
employeeModel)
        {
            try
            {
                return Ok(employeeRepository.UpdateEmployee(employeeModel));
            }
            catch (Exception e)
            {
```

```
            return Problem(e.Message);
        }
    }

    [HttpPut("UpdateEmployeePassword")]
    public IActionResult UpdateEmployeePassword([FromBody] UpdateEmployeeModel
employeeModel)
    {
        int currentUserId = int.Parse(User.Identity.Name);

        if (currentUserId != employeeModel.EmployeeId)
        {
            return Forbid();
        }

        try
        {
            return Ok(employeeRepository.UpdateEmployee(employeeModel));
        }
        catch (Exception e)
        {
            return Problem(e.Message);
        }
    }
}
```

## MappingConfig.cs

```
public class MappingConfig : Profile
{
    public MappingConfig()
    {
        EmployeeMap();
        RoleMap();
        LoginInfoMap();
        CreateEmployeeMap();
        CreateEmployeeLoginMap();
        UpdateEmployeeMap();
        UpdateEmployeeLogin();
        AtWorkMap();
        LoginHistoryMap();
    }

    private void EmployeeMap()
    {
        CreateMap<Employee, EmployeeModel>()
            ;
    }

    private void RoleMap()
    {
        CreateMap<Role, RoleModel>()
            ;
```

```
    }

    private void LoginInfoMap()
    {
        CreateMap<LoginInfo, LoginModel>()
            ;
    }

    private void CreateEmployeeMap()
    {
        CreateMap<CreateEmployeeModel, Employee>()
            ;
    }

    private void CreateEmployeeLoginMap()
    {
        CreateMap<CreateEmployeeModel, LoginInfo>()
            ;
    }

    private void UpdateEmployeeMap()
    {
        CreateMap<UpdateEmployeeModel, Employee>()
            ;
    }

    private void UpdateEmployeeLogin()
    {
        CreateMap<UpdateEmployeeModel, LoginInfo>()
            ;
    }

    private void AtWorkMap()
    {
        CreateMap<AtWork, AtWorkModel>()
            ;
    }

    private void LoginHistoryMap()
    {
        CreateMap<LoginHistory, LoginHistoryModel>()
            ;
    }
}
```

### 20200410091730_AddAtWork.cs

```
public partial class RemoveTokenPropertyFromEmployeeEntity : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropColumn(
            name: "Token",
            table: "Employees");
    }
```

```
    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.AddColumn<string>(
            name: "Token",
            table: "Employees",
            type: "nvarchar(max)",
            nullable: true);
    }
}
```

**EmployeeModel.cs**

```
[Serializable]
public class EmployeeModel
{
    public int Id { get; set; }
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
    public string LastName { get; set; }
    public RoleModel Role { get; set; }

    [JsonIgnore]
    public string Token { get; set; }
}
```

**Employee.cs**

```
[Serializable]
[Table("Employees")]
public class Employee
{
    [Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
    public int Id { get; set; }

    [MaxLength(64)]
    [Required]
    public string FirstName { get; set; }

    [MaxLength(64)]
    [Required]
    public string MiddleName { get; set; }

    [MaxLength(64)]
    [Required]
    public string LastName { get; set; }

    public LoginInfo LoginInfo { get; set; }
    public AtWork AtWork { get; set; }

    public IEnumerable<LoginHistory> LoginsHistory { get; set; }

    [ForeignKey(nameof(RoleId))]
    public Role Role { get; set; }
```

```
        [Required]
        public int RoleId { get; set; }

    }
```

### IEmployeeManager.cs

```
public interface IEmployeeManager : IRepositoryBase<Employee>
    {
        public IEnumerable<Employee> GetAllWithRoles();

        public Employee GetByIdWithRole(int id);
    }
```

### EmployeeManager.cs

```
public class EmployeeManager : RepositoryBase<Employee>, IEmployeeManager
    {
        public EmployeeManager(RepositoryContext repositoryContext) : base(repositoryContext)
        {
        }

        public IEnumerable<Employee> GetAllWithRoles()
        {
            return RepositoryContext.Employees
                .Include(x => x.Role)
                .ToList();
        }

        public Employee GetByIdWithRole(int id)
        {
            return RepositoryContext.Employees.Include(x => x.Role).Where(x => x.Id ==
id).FirstOrDefault();
        }
    }
```

### IRepositoryBase.cs

```
public interface IRepositoryBase<T> where T : class
    {
        IEnumerable<T> GetAll();

        T FindById(int id);
        void Create(T entity);
        void Update(T entity);
        void Delete(T entity);
        int Count();
    }
```

### RepositoryBase.cs

```
Public class RepositoryBase<T> : IRepositoryBase<T> where T : class
    {
        protected RepositoryContext RepositoryContext { get; set; }
        public RepositoryBase(RepositoryContext repositoryContext)
        {
```

```
        RepositoryContext = repositoryContext;
    }

    public virtual IEnumerable<T> GetAll()
    {
        return RepositoryContext.Set<T>();
    }

    public virtual T FindById(int id)
    {
        return RepositoryContext.Set<T>().Find(id);
    }

    public virtual void Create(T entity)
    {
        RepositoryContext.Set<T>().Add(entity);
    }

    public virtual void Update(T entity)
    {
        RepositoryContext.Set<T>().Update(entity);
    }

    public virtual void Delete(T entity)
    {
        RepositoryContext.Set<T>().Remove(entity);
    }

    public virtual int Count()
    {
        return RepositoryContext.Set<T>().Count();
    }

}
```

**EmployeeSeed.cs**

```
public class EmployeeSeed
{
    public static Employee[] Seed()
    {
        return new Employee[]
        {
            new Employee{Id=1, FirstName="Rosen",MiddleName="Yavorov",
LastName="Lechev", RoleId=1},
            new Employee { Id = 2, FirstName = "Neli", MiddleName = "Lychezarova",
LastName = "Zarkova", RoleId=2},
        };
    }
}
```

**IEmployeeRepository.cs**

```
public interface IEmployeeRepository
```

```
    {
        public EmployeeModel Authenticate(string username, string password);

        public IEnumerable<EmployeeModel> GetAllEmployees();

        public EmployeeModel GetById(int id);

        public EmployeeModel CreateEmployee(CreateEmployeeModel createEmployeeModel);

        public void DeleteEmployee(int id);

        public void UpdateEmployee(UpdateEmployeeModel employeeModel);
    }
}
```

**EmployeeService.cs**

```
    public class EmployeeService : IEmployeeRepository
    {
        private readonly IUnitOfWork unitOfWork;
        private readonly IMapper mapper;
        private readonly ILoginInfoRepository loginInfoRepository;
        private readonly AppSettings appSettings;


        public EmployeeService(IUnitOfWork unitOfWork, IOptions<AppSettings> appSettings,
IMapper mapper, ILoginInfoRepository loginInfoRepository)
        {
            this.unitOfWork = unitOfWork;
            this.mapper = mapper;
            this.loginInfoRepository = loginInfoRepository;
            this.appSettings = appSettings.Value;
        }


        public EmployeeModel Authenticate(string username, string password)
        {
            int? userId = GetLoginInfoUserId(username, password);
            if (userId == null)
                return null;


            EmployeeModel employee = GetById(userId.GetValueOrDefault());
```

```
            JwtSecurityTokenHandler tokenHandler = new JwtSecurityTokenHandler();
            byte[] key = Encoding.ASCII.GetBytes(appSettings.Secret);
            SecurityTokenDescriptor tokenDescriptor = new SecurityTokenDescriptor
            {
                Subject = new ClaimsIdentity(new Claim[]
                {
                    new Claim(ClaimTypes.Name, employee.Id.ToString()),
                    new Claim(ClaimTypes.Role, employee.Role.RoleDescription)
                }),
                Expires = DateTime.UtcNow.AddDays(1),
                SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(key),
SecurityAlgorithms.HmacSha256Signature)
            };
            SecurityToken token = tokenHandler.CreateToken(tokenDescriptor);
            employee.Token = tokenHandler.WriteToken(token);

            return employee;
        }

        public IEnumerable<EmployeeModel> GetAllEmployees()
        {
            IEnumerable<Employee> employees =
unitOfWork.EmployeeManager.GetAllWithRoles();

            IEnumerable<EmployeeModel> employeeModels =
mapper.Map<IEnumerable<EmployeeModel>>(employees);

            return employeeModels;
        }

        public EmployeeModel GetById(int id)
        {
            EmployeeModel employee =
mapper.Map<EmployeeModel>(unitOfWork.EmployeeManager.GetByIdWithRole(id));
```

```
            return employee;
        }

        private Employee GetEntityById(int id)
        {
            return unitOfWork.EmployeeManager.FindById(id);
        }

        public int? GetLoginInfoUserId(string username, string password)
        {
            LoginInfo login =
unitOfWork.LoginInfoManager.GetAllWIthEmployee().SingleOrDefault(x => x.Username ==
username);
            if (login == null)
                return null;

            if (!VerifyPasswordHash(password, login.PasswordHash, login.PasswordSalt))
                return null;

            return login.EmployeeId;
        }

        public EmployeeModel CreateEmployee(CreateEmployeeModel createEmployeeModel,
ref string message)
        {
            if (unitOfWork.LoginInfoManager.GetAllWIthEmployee().SingleOrDefault(x =>
x.Username == createEmployeeModel.Username) != null)
            {
                message = "Username already exists!";
                return null;
            }

            byte[] passwordHash, passwordSalt;
            CreatePasswordHash(createEmployeeModel.Password, out passwordHash, out
passwordSalt);
```

```
        Employee employee = mapper.Map<Employee>(createEmployeeModel);
        int roleId = unitOfWork.RoleManager.GetAll().Where(x => x.RoleDescription ==
createEmployeeModel.RoleDescription).FirstOrDefault().Id;
        employee.RoleId = roleId;
        LoginInfo loginInfo = mapper.Map<LoginInfo>(createEmployeeModel);
        loginInfo.PasswordSalt = passwordSalt;
        loginInfo.PasswordHash = passwordHash;

        try
        {
            unitOfWork.BeginTransaction();

            unitOfWork.EmployeeManager.Create(employee);
            unitOfWork.SaveChanges();

            loginInfo.EmployeeId = employee.Id;
            unitOfWork.LoginInfoManager.Create(loginInfo);
            unitOfWork.SaveChanges();

            unitOfWork.CommitTransaction();
        }
        catch
        {
            unitOfWork.RollbackTransaction();
            message = "Could not create employee";
            return null;
        }

        EmployeeModel employeeModel = mapper.Map<EmployeeModel>(employee);

        return employeeModel;

    }
```

```
    private void CreatePasswordHash(string password, out byte[] passwordHash, out byte[]
passwordSalt)
    {
        if (password == null) throw new ArgumentNullException("password");
        if (string.IsNullOrWhiteSpace(password)) throw new ArgumentException("Value
cannot be empty or whitespace only string.", "password");

        using (var hmac = new System.Security.Cryptography.HMACSHA512())
        {
            passwordSalt = hmac.Key;
            passwordHash =
hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
        }
    }

    private bool VerifyPasswordHash(string password, byte[] storedHash, byte[] storedSalt)
    {
        if (password == null) throw new ArgumentNullException("password");
        if (string.IsNullOrWhiteSpace(password)) throw new ArgumentException("Value
cannot be empty or whitespace only string.", "password");
        if (storedHash.Length != 64) throw new ArgumentException("Invalid length of
password hash (64 bytes expected).", "passwordHash");
        if (storedSalt.Length != 128) throw new ArgumentException("Invalid length of
password salt (128 bytes expected).", "passwordHash");

        using (var hmac = new System.Security.Cryptography.HMACSHA512(storedSalt))
        {
            var computedHash =
hmac.ComputeHash(System.Text.Encoding.UTF8.GetBytes(password));
            for (int i = 0; i < computedHash.Length; i++)
            {
                if (computedHash[i] != storedHash[i]) return false;
            }
        }
```

```
            return true;
        }


        public void DeleteEmployee(int id)
        {
            Employee employee = GetEntityById(id);
            if (employee != null)
            {
                unitOfWork.EmployeeManager.Delete(employee);
                unitOfWork.SaveChanges();
            }

        }

        public string UpdateEmployee(UpdateEmployeeModel employeeModel)
        {
            bool updateUser = false;
            bool updateLoginInfo = false;

            Employee employeeDB = GetEntityById(employeeModel.EmployeeId);
            LoginInfo loginDB =
loginInfoRepository.GetEntityByUserId(employeeModel.EmployeeId);

            if (employeeDB == null || loginDB == null)
            {
                return "Update failed! Could not find employee.";
            }

            int roleId = -1;
            if (employeeModel.RoleDescription != null)
            {
                roleId = unitOfWork.RoleManager.GetAll().Where(x => x.RoleDescription ==
employeeModel.RoleDescription).FirstOrDefault().Id;
```

```
        }

        if ((!string.IsNullOrWhiteSpace(employeeModel.FirstName) &&
employeeModel.FirstName != employeeDB.FirstName)
            || (!string.IsNullOrWhiteSpace(employeeModel.MiddleName) &&
employeeModel.MiddleName != employeeDB.MiddleName)
            || (!string.IsNullOrWhiteSpace(employeeModel.LastName) &&
employeeModel.LastName != employeeDB.LastName)
            || (roleId != -1 && employeeDB.RoleId != roleId))
        {
            employeeDB.FirstName = employeeModel.FirstName;
            employeeDB.MiddleName = employeeModel.MiddleName;
            employeeDB.LastName = employeeModel.LastName;

            employeeDB.RoleId = roleId;
            updateUser = true;
        }
        if (!string.IsNullOrWhiteSpace(employeeModel.Password) &&
!VerifyPasswordHash(employeeModel.Password, loginDB.PasswordHash,
loginDB.PasswordSalt))
        {
            byte[] passwordHash, passwordSalt;
            CreatePasswordHash(employeeModel.Password, out passwordHash, out
passwordSalt);

            loginDB.PasswordSalt = passwordSalt;
            loginDB.PasswordHash = passwordHash;

            updateLoginInfo = true;
        }

        if (updateUser && updateLoginInfo)
        {
            try
            {
                unitOfWork.BeginTransaction();
```

```
            unitOfWork.EmployeeManager.Update(employeeDB);
            unitOfWork.SaveChanges();

            unitOfWork.LoginInfoManager.Update(loginDB);
            unitOfWork.SaveChanges();

            unitOfWork.CommitTransaction();
        }
        catch(Exception e)
        {
            unitOfWork.RollbackTransaction();
            return e.Message;
        }
    }
    else if (updateUser)
    {
        try
        {
            unitOfWork.EmployeeManager.Update(employeeDB);
            unitOfWork.SaveChanges();
        }
        catch(Exception e)
        {
            return e.Message;
        }
    }
    else if (updateLoginInfo)
    {
        try
        {
            unitOfWork.LoginInfoManager.Update(loginDB);
            unitOfWork.SaveChanges();
        }
```

```
            catch(Exception e)
            {
                return e.Message;
            }
        }
        return "Update sucsessfull.";
    }
}
```

**IUnitOfWork.cs**

```
public interface IUnitOfWork
{
    IEmployeeManager EmployeeManager { get; }
    ILoginInfoManager LoginInfoManager { get; }
    IRoleManager RoleManager { get; }
    IAtWorkManager AtWorkManager { get; }

    ILoginHistoryManager LoginHistoryManager { get; }

    void SaveChanges();

    void BeginTransaction();

    void CommitTransaction();

    void RollbackTransaction();
}
```

**UnitOfWork.cs**

```
public class UnitOfWork : IUnitOfWork
{
    private readonly RepositoryContext repositoryContext;

    public UnitOfWork(RepositoryContext repositoryContext)
    {
        this.repositoryContext = repositoryContext;
        EmployeeManager = new EmployeeManager(repositoryContext);
        LoginInfoManager = new LoginInfoManager(repositoryContext);
        RoleManager = new RoleManager(repositoryContext);
        AtWorkManager = new AtWorkManager(repositoryContext);
        LoginHistoryManager = new LoginHistoryManager(repositoryContext);
    }

    public IEmployeeManager EmployeeManager { get; }

    public ILoginInfoManager LoginInfoManager { get; }

    public IRoleManager RoleManager { get; }
```

```
        public IAtWorkManager AtWorkManager { get; }

        public ILoginHistoryManager LoginHistoryManager { get; }

        public void SaveChanges()
        {
            repositoryContext.SaveChanges();
        }

        public void BeginTransaction()
        {
            repositoryContext.Database.BeginTransaction();
        }

        public void CommitTransaction()
        {
            repositoryContext.Database.CommitTransaction();
        }

        public void RollbackTransaction()
        {
            repositoryContext.Database.RollbackTransaction();
        }
    }
```

**RepositoryContext.cs**

```
public class RepositoryContext : DbContext
    {
        public RepositoryContext(DbContextOptions options) : base(options)
        {
        }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            #region Employee
            modelBuilder.Entity<Employee>()
                .HasOne(a => a.LoginInfo)
                .WithOne(b => b.Employee)
                .HasForeignKey<LoginInfo>(b => b.EmployeeId);

            modelBuilder.Entity<Employee>()
                .HasOne(a => a.AtWork)
                .WithOne(b => b.Employee)
                .HasForeignKey<AtWork>(b => b.EmployeeId);

            modelBuilder.Entity<Employee>()
                .HasData(EmployeeSeed.Seed());

            modelBuilder.Entity<Employee>()
                .HasMany(a => a.LoginsHistory)
                .WithOne(b => b.Employee);
            #endregion

            #region LoginInfo
```

```
            modelBuilder.Entity<LoginInfo>()
                .HasIndex(prop => prop.Username)
                .IsUnique();

            modelBuilder.Entity<LoginInfo>()
                .HasData(LoginInfoSeed.Seed());
            #endregion

            #region Role
            modelBuilder.Entity<Role>()
                .HasMany(a => a.Employees)
                .WithOne(b => b.Role);

            modelBuilder.Entity<Role>()
                .HasData(RoleSeed.Seed());
            #endregion

            #region AtWork
            //No seed needed
            #endregion

            #region LoginHistory
            // no seed needed
            #endregion
        }

        public DbSet<Employee> Employees { get; set; }
        public DbSet<LoginInfo> LoginInfos { get; set; }
        public DbSet<Role> Roles { get; set; }
        public DbSet<AtWork> AtWork { get; set; }
        public DbSet<LoginHistory> LoginHistory { get; set; }
    }
```