МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ЯРОШ ОЛЕГ ЛЕОНІДОВИЧ

Допускається до захисту:
завідувач кафедри
інформаційних технологій,
доктор технічних наук, доцент
_____Т. В. Нескородєва
«____» _____ 20__ р.

РОЗРОБКА API ДЛЯ СИСТЕМИ УПРАВЛІННЯ ПРОЄКТАМИ

Спеціальність 122 «Комп’ютерні науки»
Кваліфікаційна (бакалаврська) робота

Керівник:
Бабаков Р. М., доцент кафедри
інформаційних технологій,
д.т.н, доцент

Оцінка: ___/___/_____
(бали за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____
(підпис)

Вінниця – 2022

АНОТАЦІЯ

Ярош О.Л. Розробка API для системи управління проєктами. Спеціальність 122 «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2022.

Основною метою виконання кваліфікаційної роботи є побудова архітектури і реалізація API для системи управління проєктами, враховуючи зростання попиту на такого роду програмні продукти.

У вступі наведено актуальність розробки систем управління проєктами. У першому розділі проведений аналіз предметної області та розглянуті існуючі аналоги, виділені їх недоліки та переваги.

У другому розділі розглянуті основні технології які були використані при створенні додатку, а також інструментів і серверу бази даних. Виділені їх переваги.

У третьому розділі було розглянуто основні архітектурні рішення, процес побудови бази даних, реалізацію архітектурних рішень, аутентифікації та авторизації. Описаний принцип взаємодії з додатком та його можливості.

Ключові слова: API, система управління проєктами, веб-додаток, .NET, ASP.NET, багатошарова архітектура, REST API

ANNOTATION

Yarosh O.L. API development for project management system. 122 "Computer Science" Major. Vasyl' Stus Donetsk National University, Vinnytsia, 2022.

The main purpose of the qualification work is to build an architecture and implement an API for the project management system, due to the growing demand for this type of software products.

The introduction presents the relevance of the development of project management systems. First section contains analysis of the subject area and considering of the existing analogues, their pros and cons are highlighted.

The second section discusses the main technologies that were used in creating the application, as well as tools and database server. Their advantages are highlighted.

The third section discusses the main architectural solutions, the process of building a database, implementation of architectural solutions, authentication and authorization. The principle of interaction with the application and its capabilities are described.

Keywords: API, project management system, web application, .NET, ASP.NET, multilayered architecture, REST API

ЗМІСТ

ВСТУП

У світі в якому людство намагається диджіталізувати все можливе, всі звичайні речі вже давно перейшли в світ цифрових технологій. Нотатники були замінені додатками-записниками в смартфонах, ведення бухгалтерії перейшло у відповідні програми. Хвиля діджиталізації звичайно не могла не зачепити сфери з якими ми стикаємось кожен день. Так наприклад, звичні нам шкільні щоденники в школах перейшли в онлайн таблиці, а тестування на листочках вже відбувається на відповідних сайтах. Тому не дивно, що така важлива частина нашого суспільства як освіта та бізнес вже давно користується системами управління проєктами в тому чи іншому вигляді. Від них залежить ефективність відповідних процесів.

Коли йде мова про розробку сервісу для управління проєкта, основною метою є створення основи для будь-якого подібного сервісу задля подальшого покращення під власні потреби.

Тема бакалаврської роботи – «Розробка API для системи управління проєктами». Основною метою роботи є:

- Огляд існуючих додатків
- Вдосконалення навичок розробки серверної частини додатків з поглибленим вивченням та застосування найрозповсюдженіших архітектурних підходів, технологій та інструментів
- Реалізація кінцевого продукту, а саме API для системи управління проєктами
- Тестування продукту

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

В сучасному світі досить велика частина бізнесу є проєкто-орієнтована. Частка такого бізнесу в Україні становить близько половини, в тому числі і аграрної сфери. Така велика кількість пов’язана з тим, що необхідність управління проектами, тобто ефективність координації використання людських та матеріальних ресурсів протягом усього життєвого циклу проєкту використовуючи сучасні методи управління для отримання бажаного результату, тобто високої якості продукції та прибутків відповідного рівня, випливає в масовий ріст масштабів та складності проєктів і відповідно зростання вимог до термінів виконання.

Також проєкт-орієнтована модель широко використовується в навчанні задля надання необхідних навичок студентам, учням та зручності освітнього процесу.

З початку 2020 року епідемія коронавірусу охопила весь світ. Пандемія вплинула на всі сфери життя, в тому числі на бізнес та навчання. Більшість компаній та учбових закладів в тому чи іншому вигляді були вимушені перейти на дистанційну форму роботи та навчання відповідно[1].

В до пандемічний період процес навчання відбувався в приміщеннях університету де тестування і виконання завдань відбувалося за фізичної присутності студента чи учня в аудиторії і фіксувалось на папері. Те саме стосується й роботи різних компаній, де працівники збирались в офісі, командою проєкту обговорювали необхідні задачі та час, який необхідний для їх виконання, та розподіляли їх по всім членам команди фіксуючи закріплення задачі за певним працівником шляхом запису на листку або на спеціальній дошці.

Період карантину вніс свої зміни в звичайний для нас перебіг цих подій та змусив адаптуватись. Всі ці процеси вимушені були бути оцифровані, відповідно попит на необхідні сервіси все більше зростав і буде зростати, в тому числі і на системи управління проектами[2].

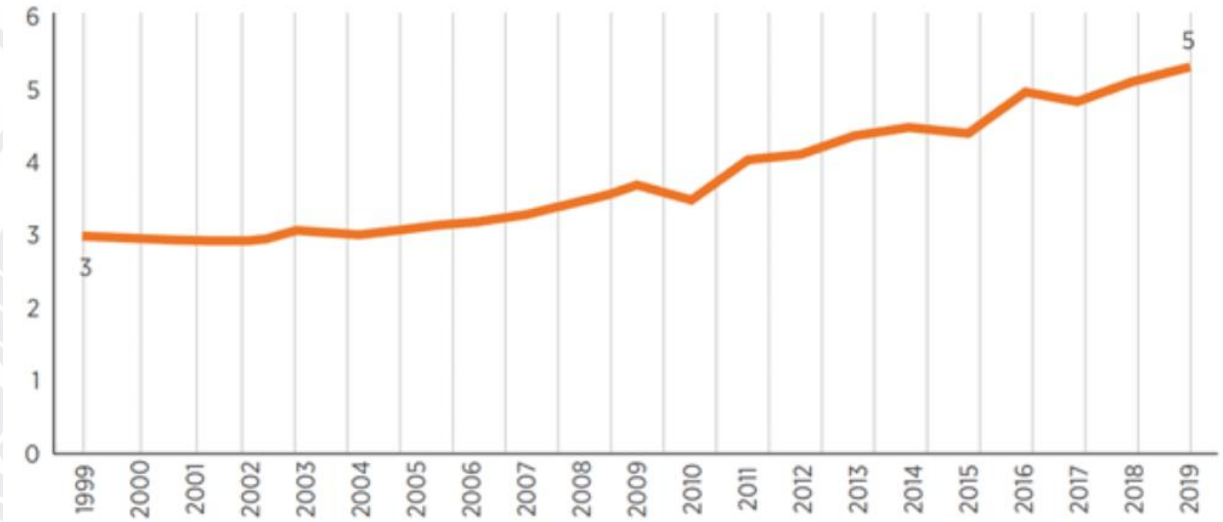

Рис. 1 – Зростання дистанційного місця роботи як основного у відсотках [1]

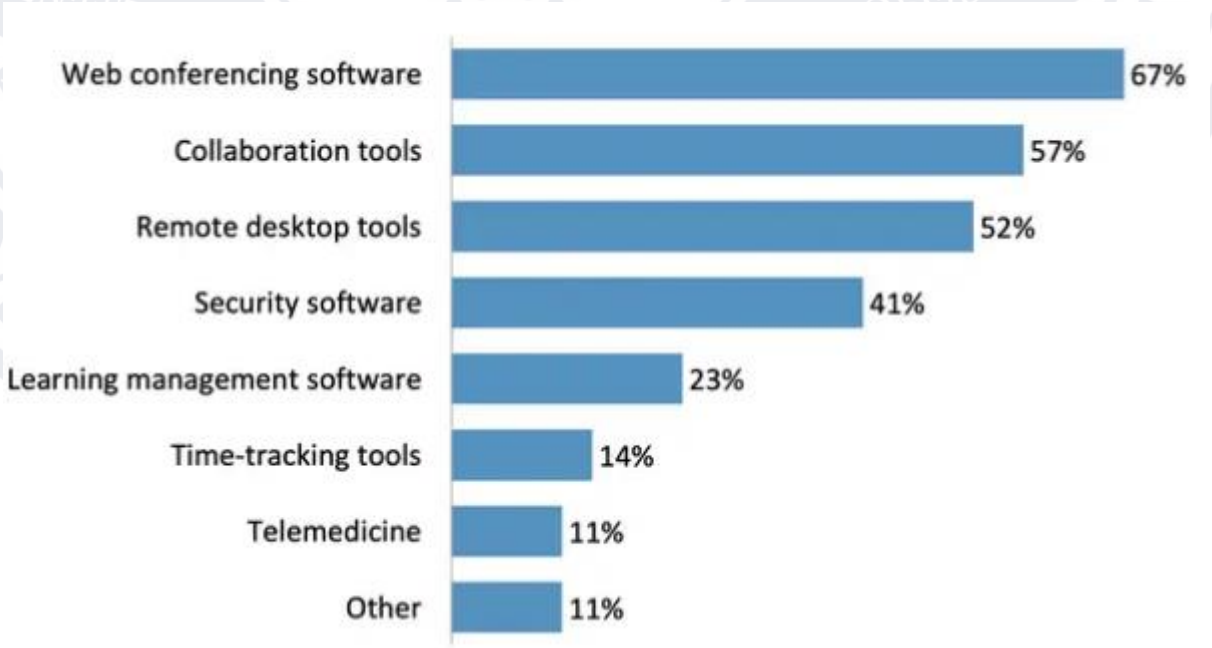

Рис. 2 – Зростання попиту на ПЗ відповідних категорій за 2020 рік [2]

1.2 Постановка задачі

Мета бакалаврської роботи полягає в створені REST API для системи управління проектами. Основною ціллю є створення саме проєкту з використанням багаторівневою архітектури та дотриманням загальноприйнятих правил REST, а також SOLID.

- Визначити основні можливості API, а саме:
    - Аутентифікація та реєстрація, з авторизацією по ролям
    - Створення та редагування користувачем з роллю «Менеджер» проєктів
    - Створення, редагування та присвоєння завдань підв’язаних до відповідного проєкту іншим користувачам
    - Визначення ролі конкретного користувача на конкретному проєкті
    - Редагування профілю з персональною інформацією користувача
    - Можливість зміни ролей користувачів
    - Зміна статусу завдання відповідно до робочого процесу
    - Відсоток виконаних завдань на проєкті
    - Відсоток виконаних завдань закріплених за певним користувачем
- Визначити набір технологій для створення додатку
- Визначити основні принципи багаторівневої архітектури
- Визначити основні правила REST
- Побудувати схему бази даних
- Визначити набір HTTP запитів
- Реалізувати задуманий Web API

1.3 Огляд існуючих аналогій

Системи управління проєктами[6] вже давно не є чимось інноваційним, тому вибір серед таких систем досить великий.

Ми розглянемо лише декілька з них:

1. Asana [3]

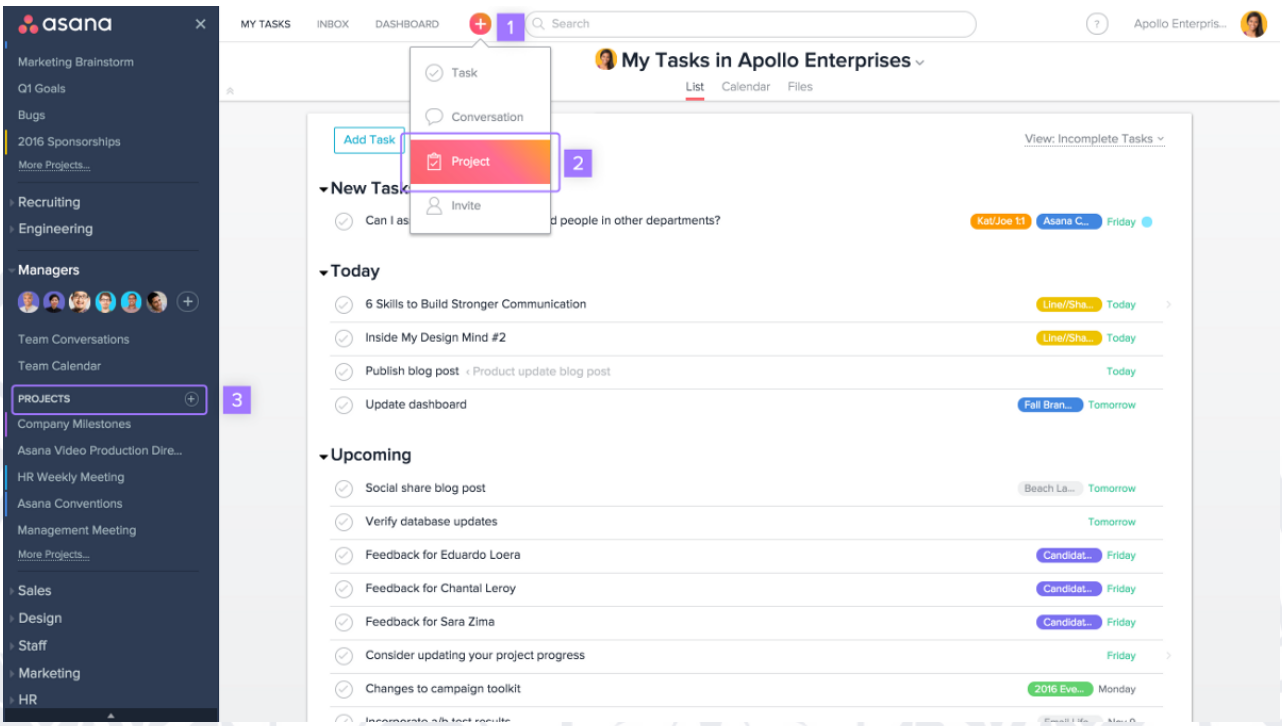

Рис. 3 – Система управління проєктами “Asana”

Asana підходить для маленьких команд, а також для користування декількома особами. Система допоможе тримати фокус на цілях проєкту, самих проєктах, а також завданнях та дедлайнах. В цілому система може використовуватися для ведення так званої дорожньої карти кінцевого продукту.

Сервіс добре візуалізований та відмінно підходить для ведення маркетингових та креативних кампаній.

Також цей сервіс також гарно підходить для ведення та управління проектами в дизайні, IT відділах, HR, фінансах та продажах, розробці, організації заходів та ін.

У 2018 році було реалізовано принцип часової шкали, так званий таймлайн, що становить основу діаграми Ганта.

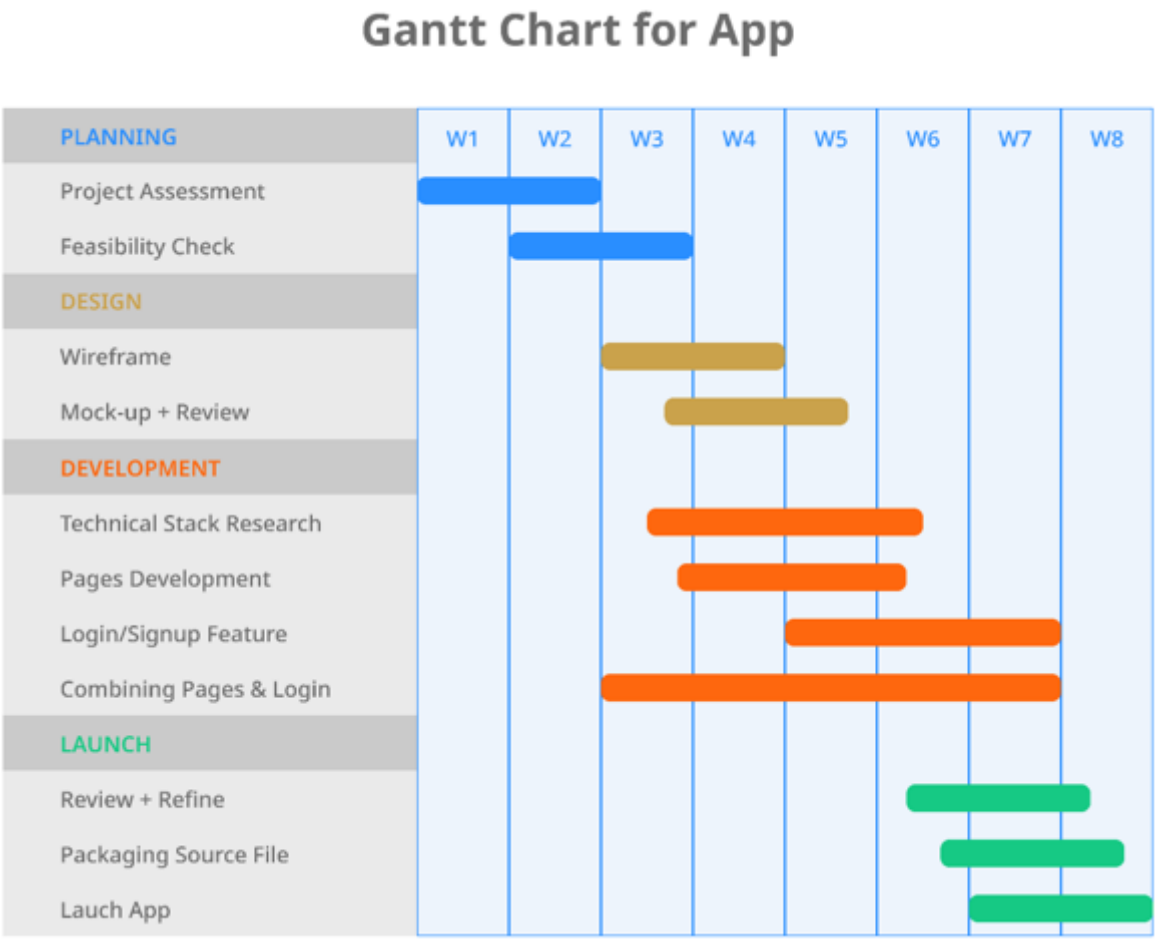

Рис. 4 – Приклад діаграма Ганта у процесі розробки додатку [4]

Переваги Asana:

- Дизайн.
- Багато синхронізацій.
- Застосуємо у багатьох сферах.

Недоліки Asana:

- Не для великих команд.
- Відсутність української мови.

2. Active Collab [5]

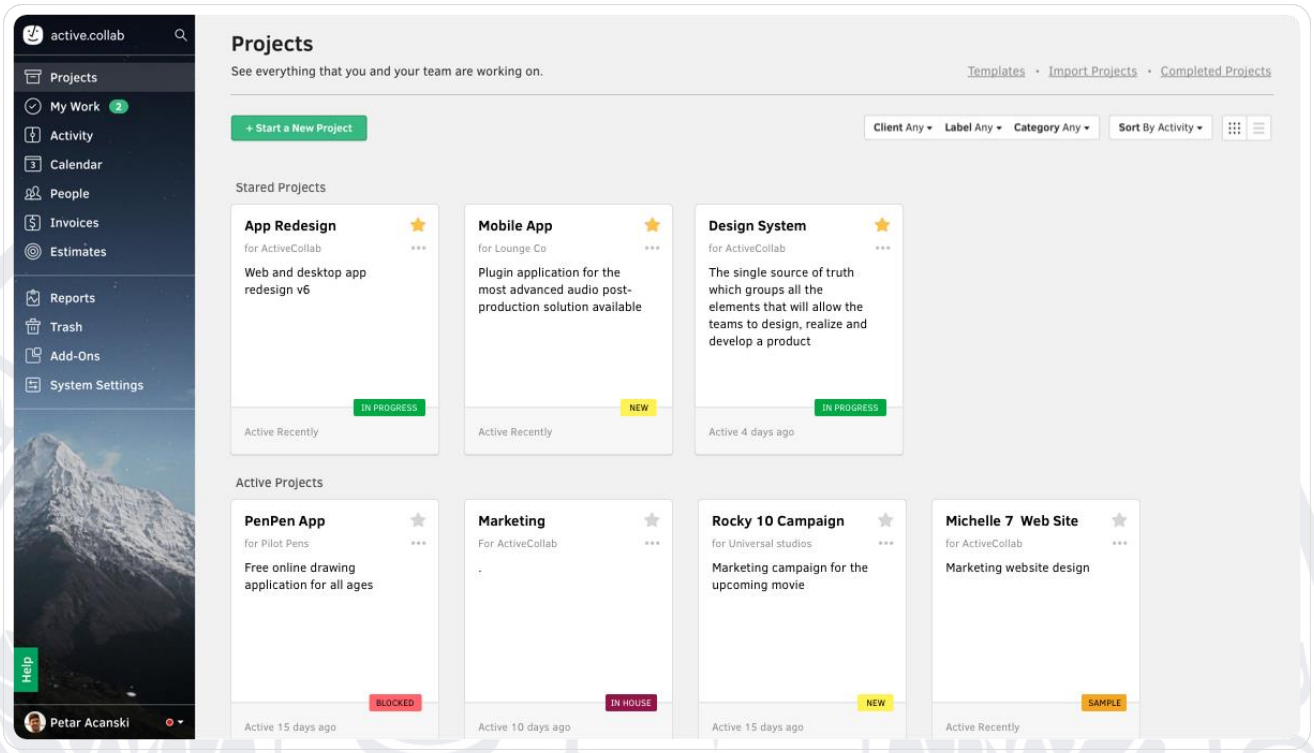

Рис. 5 – Система управління проєктами “Active Collab”

Active Collab, має в наявності типовий набір можливостей. Цей онлайн сервіс управління проектами дає змогу розбити проект на завдання та підзавдання, а також об’єднати їх залежностями. Усі дедлайни, спілкування та активність централізовані, що допоможу слідкувати з активністю на проєкті.

Також достатньо інтеграцій з іншими сервісами. Наприклад для управління планами, завданнями, проведення платежів та іншими.

Переваги Active Collab:

- Декілька способів відображення завдань.
- Сортування, яке можна налаштувати.
- Мобільні додатки для iOS та Android.

- Можливість міграції з інших систем управління проєктами. Наприклад: Trello, Basecamp, Asana, Wrike.
- Можливість завантажувати документи на Google Drive та Dropbox.
- Тісна інтеграція зі Slack, Zapier.
- Інтеграції з інструментами для відстеження часу Hubstaff та TimeCamp, а також в наявності є власний відповідний інструмент.

Недоліки Active Collab:

- Потребує багато часу для освоєння.
- Не підтримується українська мова.

3. Wrike [7]

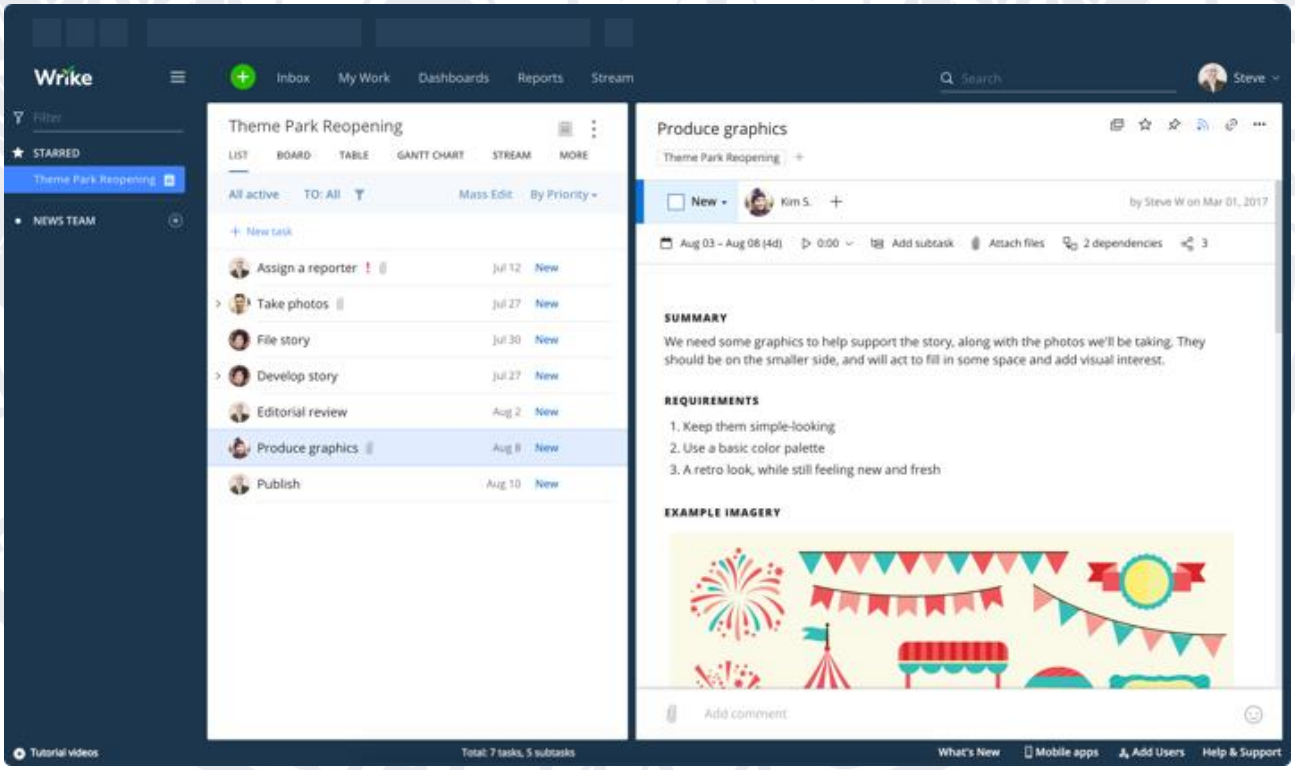

Рис. 7 – Система управління проєктами “Wrike”

Wrike більшою мірою сервіс орієнтований на великі команди, хоча на сайті зазначено, що для спільної роботи він підходить для команд всіх розмірів.

Інструмент допоможе розподілити обов'язки та визначити пріоритети. В інструментарії є діаграма Ганта, можливість керувати ресурсами та завантаженням членів команди, багато опцій налаштування та різних статусів.

Переваги Wrike:

- Можливість повноцінної роботи над проектом.
- Є додатки для iOS та Android.
- Багато синхронізацій.
- Зображення та відео.
- Звіти з інтерактивними можливостями.

Недоліки Wrike:

- Високий поріг входження в користування програмою.
- Відсутня українська мова.
- Досить висока вартість сервісу.

Висновок до розділу 1

У цьому розділі було розглянуто актуальність систем управління проєктами, передумови збільшення попиту на такі сервісі та їхню необхідність в сьогодення. Також були описані функції програми які повинні бути реалізовані та інформація з якою необхідно ознайомитись. Також були розглянуті декілька найпопулярніших аналогів, в взвішені їх переваги та недоліки.

РОЗДІЛ 2
АНАЛІЗ ТА ВИБІР АКТУАЛЬНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Великим фактор успіху реалізації проєкту є правильність вибору стеку технологій. В даному розділі будуть оглянуті ключові технології та інструменти, які були використанні для розробки API. Опис їх можливостей, переваг та недоліків.

2.1 Технології

Платформа .NET

[12] Прагнення Microsoft надати розробникам одну платформу для вирішення будь-яких проблем було реалізовано в .NET.

Без сумніву, .NET відіграє центральну роль у галузі розробки програмного забезпечення. Популярність .NET у спільноті розробників є фактом. Його можна виміряти за кількістю проєктів із відкритим кодом у всьому світі та наявністю C# серед п’яти найпопулярніших мов програмування. Популярність платформи буде ще більше зростати через регулярну підтримку зі сторони розробників.

Щоб зрозуміти потужність .NET, давайте розглянемо деякі її технічні основи.

.NET — це платформа для створення багатьох типів додатків. Розроблена Microsoft з відкритим кодом, платформа підтримує крос-платформну розробку, а також розробку за допомогою кілька мов програмування та бібліотек для створення веб-додатків, мобільних, десктоп, IoT-додатків тощо.

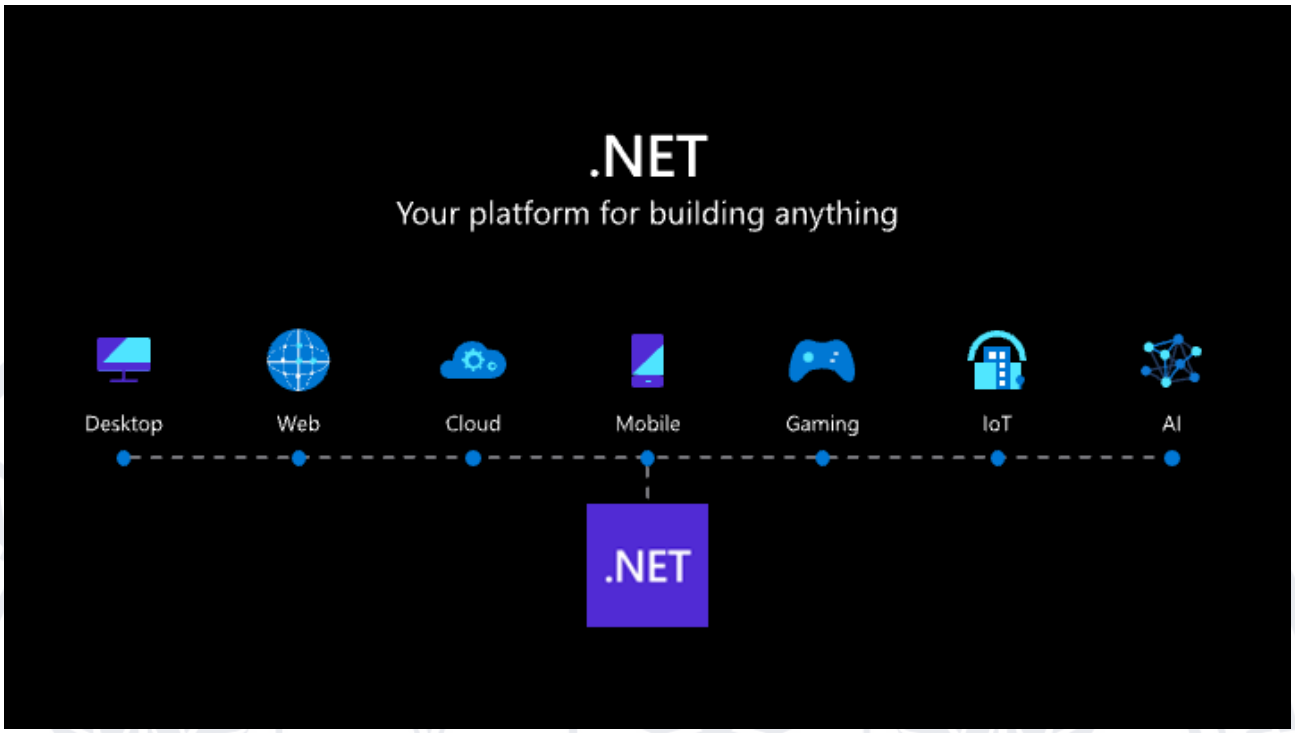

Рис. 10 – сфери розробки за допомогою .NET [9]

Microsoft безпосередньо підтримує такі мови для розробники на платформі .NET:

- C# (C sharp) : об'єктно-орієнтована мова програмування сучасного зразку, яка належить до сімейства мов C. Його синтаксис може здатися знайомим розробникам C, C++, Java та JavaScript.
- F# (F-sharp) : функціонально-орієнтована мова програмування, член сімейства мов ML . Він також підтримує парадигму об'єктно-орієнтованого програмування.
- Visual Basic : історична мова програмування Microsoft. Він став повноцінною об'єктно-орієнтованою мовою програмування в контексті .NET.

.NET підтримує *Common Language Infrastructure* (CLI), тому написаний код, написаний на одній з цих мов, компілюється в *Common Intermediate Language* (CIL). Це гарантує відмінну взаємодію між мовами на платформі.

Також, є досить велика кількість інших мов програмування можуть компілюватися в .NET CIL. Наприклад, ClojureCLR, Eiffel, IronPython, PowerBuilder та інші.

Архітектура та компоненти .NET

.NET Components

Основа архітектури .NET полягає в двох принципах:

- *CoreCLR* - це середовище виконання .NET. Він відповідає за виконання програм CLI і включає в себе компілятор «just in time».
- *CoreFX* - API платформи, яка реалізує стандартні бібліотеки CLI, а саме бібліотеки, які забезпечують найпоширеніші функціональні можливості, такі як керування файловою системою, обробка винятків, мережеве комунікація, потоки, відображення тощо. Компонент CoreFX іноді називають *Unified Base Class Library* .

.NET Application Models

Поверх основних компонентів є різні application model frameworks, тобто бібліотеки, які пропонують підтримку для розробки різних типів додатків. Так, наприклад, .NET містить:

- ASP.NET : фреймворк, що дозволяє створювати веб-додатки та веб-апі .
- Windows Presentation Foundation (WPF) : графічний інтерфейс користувача для настільних програм Windows .
- Xamarin : фреймворк з підтримкою крос-платформної розробки мобільних та десктоп додатків,.

- Blazor : фреймворк для створення клієнтських веб-додатків за допомогою C#.
- ML.NET : фреймворк для програмування в області машинного навчання.

На додаток до цих фреймворків, .NET містить інструментарій для більшості поширених завдань програмування, таких як: керування файлами, мережеві комунікації, безпеки доступу, бази даних та інші

Також є величезна кількість бібліотек у загальнодоступному репозиторії NuGet. По своїй суті, NuGet - це менеджер пакетів для .NET. Він дає змогу створювати, ділитися та використовувати багато бібліотек .NET.

Підтримка проектування та розробки .NET

Підтримка .NET для розробки програмного забезпечення не обмежується кількома мовами програмування, які ви можете використовувати. .NET також сприяє використанню деяких найкращих практик, дозволяючи вам використовувати підхід, якому ви віддаєте перевагу для створення програми.

Наприклад, він заохочує вас використовувати Dependency Injection для роз'єднання компонентів вашої програми. DI допомагає вам розробляти краще програмне забезпечення, обмежуючи взаємозалежність компонентів і полегшуючи повторне використання. Крім того, це полегшує тестування ваших компонентів.

До речі, що стосується тестування, .NET також має підтримку модульних та інтеграційних тестів через xUnit .

Що стосується досвіду розробки, розробники можуть вибирати з різних підходів. Вони можуть використовувати середовище .NET CLI , використовуючи командний рядок для створення нових проектів, додавання залежностей, створення, запуску тощо.

Вони можуть використовувати Visual Studio Code для проміжного підходу: розширений між платформний редактор поверх .NET CLI. Або вони можуть використовувати потужну IDE, як-от Visual Studio, доступну як для Windows, так і для Mac, яка надає досвід інтерактивного програмування.

Незалежно від інструменту, який буде вибраний, ви можете використовувати багато шаблонів проектів, щоб швидко розпочати створення нової програми.

Мова програмування C#

[13] C# - сучасна об'єктно-орієнтована і строго типізована мова програмування. Вона дає змогу розробникам створювати різні безпечні та надійні програми, які виконуються в середовищі .NET.

C# - це мова програмування, яка орієнтована на компоненти. Це дає змогу використовувати в розробці відповідні мовні конструкції. Дивлячись на це можна зробити висновок, що дана мова добре підходить для створення та застосування програмних компонентів. За весь час свого існування, мова програмування збагатилася великої кількістю функцій для підтримки сучасний робочих навантажень та відповідності сучасним рекомендаціями щодо розробки програмного забезпечення. В основні лежить принцип об'єктно-орієнтованості, де розробник визначає типи та їх поведінку.

- Декілька основних функції, якими забезпечена мова C# для створення програм:
- Збирання сміття, яке за допомогою певних механізмів автоматично звільняє пам'ять, зайняту об’єктами на які вже немає посилання.
- Типи, які можуть набувати значення null, що забезпечує захист від змінних, до яких ще не прив’язані жодні об’єкти.
- Обробка помилок або винятків(exception), що надає зручний механізм для виявлення помилок та подальшого позбавлення від них.

- Лямбда-вирази, які дають змогу використовувати принципу функціонального програмування.
- Синтаксис LINQ, який дає зручний інструментарій для роботи з даними будь-якого джерела.
- Асинхронних операції, які дають підтримку та можливість використання розподілених систем в розропці.
- Єдина система типів, що означає чітку ієрархію спадкування всіх типів від одного кореневого типу object.

З останнього пункту можна зробити висновок, що у всіх об’єктів є спільний набір методів та з ним можна працювати схожим шляхом. C# типи посилання, що визначаються користувачами, а також типи значень. Дає можливість для динамічного виділення пам’яті під об'єкти, а також зберігати у стеку певні спрощені структури. C# підтримує універсальні методи, типи, що дає змогу забезпечити більшу продуктивність та убезпечити роботу з типами. Також є так звані ітератори, які дають змогу для колекцій створених користувачем, реалізовувати спеціальний принцип їхнього обходу.

C# підкреслює ***Керування версіями*** , щоб забезпечити сумісність програм та бібліотек з часом. Роздільні модифікатори virtual і override, правила дозволу та можливість оголошення явних членів інтерфейсу це все наслідок впливу питання управління версіями.

ASP.NET Core[15]

ASP.NET Core - це технологія для створення веб-додатків на платформі .NET, розвитком якої займається компанія Microsoft. Як мови програмування для розробки програм на ASP.NET Core використовуються C# і F#.

Історія ASP.NET фактично почалася з виходом першої версії. . В той же час спочатку ASP.NET була націлена на роботу виключно в Windows на веб-

сервері IIS (хоча завдяки проекту Mono програми на ASP.NET можна було запускати і на Linux).

Однак 2014 ознаменував великі зміни, фактично революцію в розвитку платформи: компанія Microsoft взяла курс на розвиток ASP.NET як кросплатформеної технології, яка розвивається як opensource-проект. Даний розвиток платформи надалі отримав назву ASP.NET Core , що як її офіційно називають Microsoft до цих пір. Перший реліз оновленої платформи побачив світ у червні 2016 року. Тепер вона почала працювати не тільки на Windows, а й на MacOS та Linux. Вона стала легковажнішою, модульнішою, її стало простіше конфігурувати, загалом, вона стала більше відповідати вимогам поточного часу.

Поточна версія ASP.NET Core, яка власне і буде охоплена у поточному посібнику, вийшла разом із релізом .NET 6 у листопаді 2021 року.

ASP.NET Core повністю підтримується за принципом opensource-фреймворком. Усі файли фреймворку доступні на github у репозиторії [10].

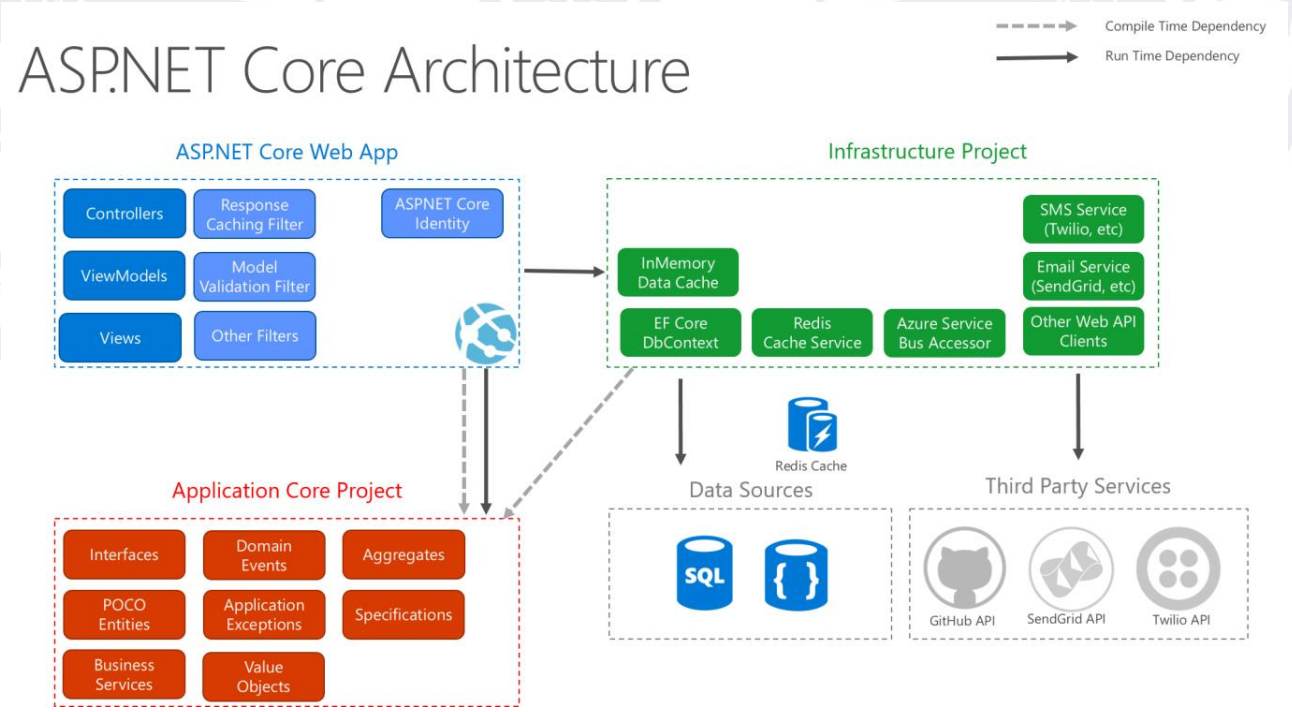

Рис. 11 – Приклад чистої архітектури ASP.NET Core [11]

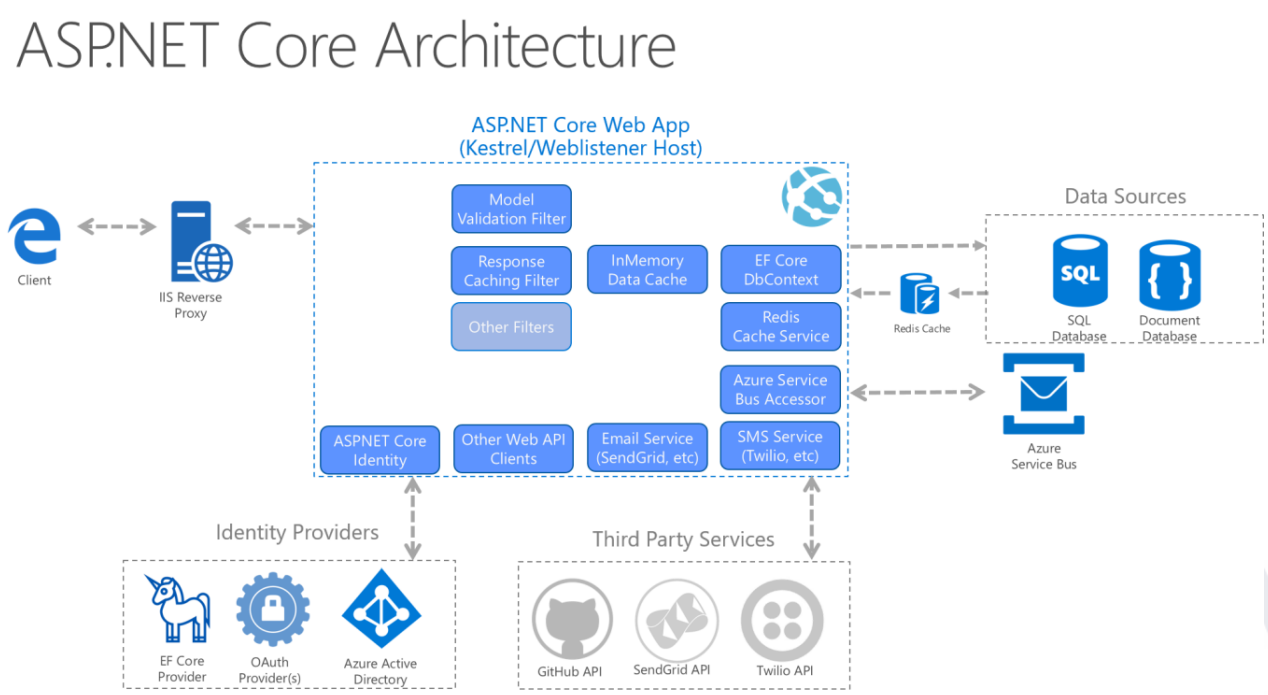

Рис. 12 – Приклад архітектури додатку ASP.NET Core під час виконання [11]

Моделі розробки

ASP.NET Core дозволяє створювати веб-застосунки за допомогою різних моделей розробки.

- ASP.NET Core , який дає повний інструментарій, необхідний для розробки сучасного веб-додатку, а саме: маршрутизація, конфігурація, логування, робота з різними системами баз даних та інші. Було додано Minimal API, що свого є мінімізованою та спрощеною моделлю, яка допомогла спростити процес розробки та написання коду програми. Решта моделей розробки працюю поверх базового функціоналу ASP.NET Core.
- ASP.NET Core MVC[8] представляє у загальному вигляді побудови програми навколо трьох основних компонентів - Model (моделі), View (уявлення) та Controller (контролери), де контролеи містять логіку обробки запитів, моделі відповідають за роботу з даними, а уявлення визначають візуальну складову.

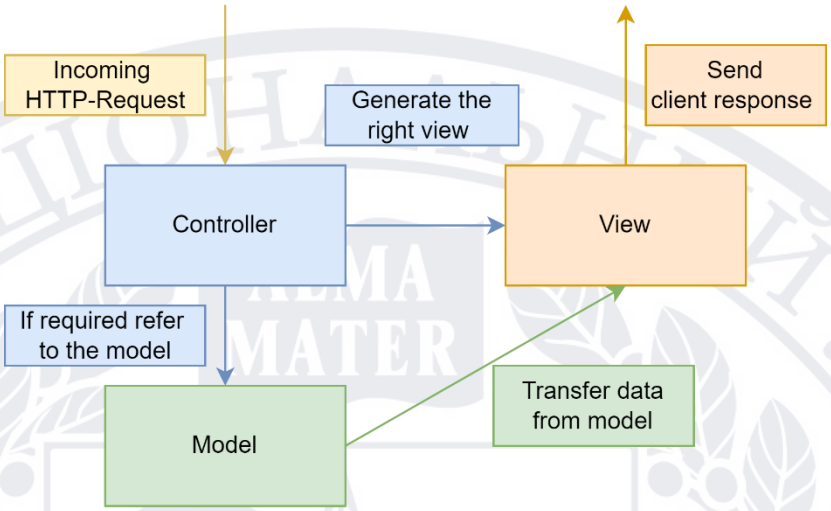

Рис. 11 – Принцип роботи MVC

- ASP.NET Core Web API це пряма реалізація додатку за принципами REST, де для кожного http-запиту (GET, POST, PUT, DELETE) призначенні окремі ресурси, які визначаються, як методи контролера Web API. Ця модель особливо підходить для односторінкових програм, але не тільки.

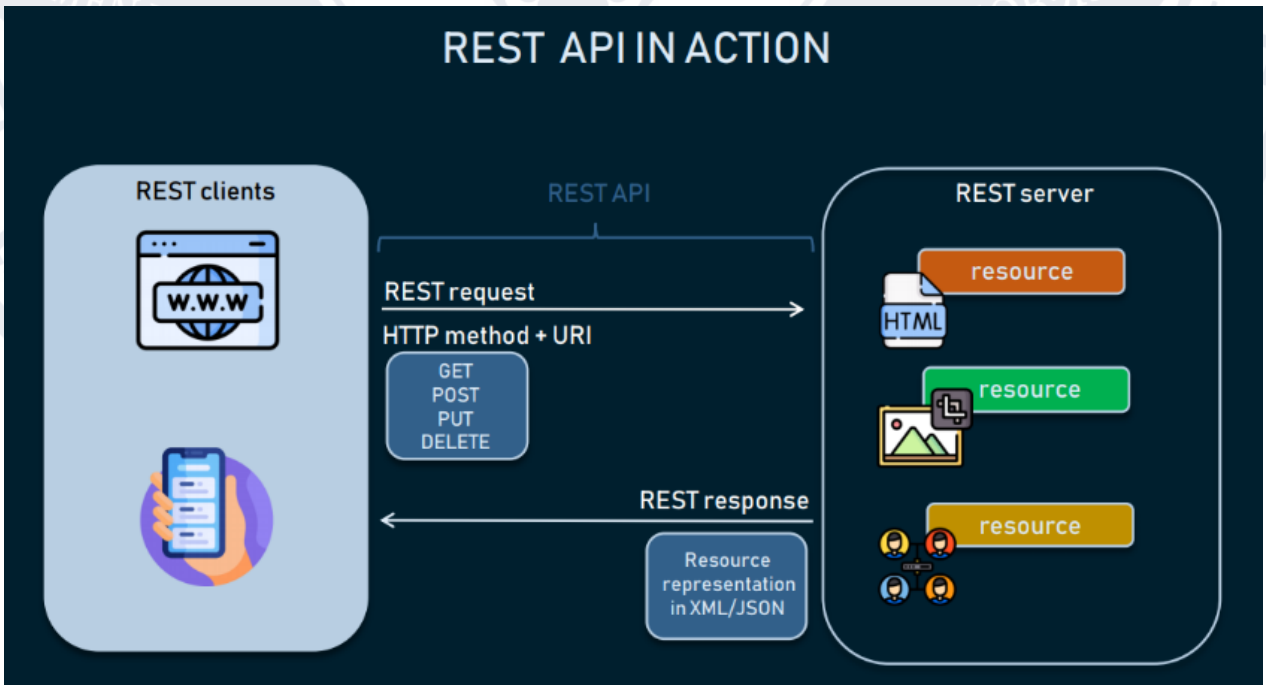

Рис. 11 – Принцип роботи REST Web API

Особливості платформи

- ASP.NET Core працює поверх платформи .NET і, таким чином, дозволяє використовувати весь її функціонал.
- Як мови розробки застосовуються мови програмування, що підтримуються платформою .NET. Офіційно вбудована підтримка для проектів ASP.NET Core має мови C# і F#
- ASP.NET Core представляє крос-платформний фреймворк, додатки якого можуть бути розгорнені на більшості популярних операційних системах: Windows, Mac OS, Linux. Виходячи з цього ця платформа розробки дає можливість як створювати крос-платформні програми на Windows, на Linux та Mac OS, так і запускати на цих ОС.
- Завдяки модульності фреймворку, всі необхідні компоненти веб-програми можуть завантажуватися як окремі модулі через пакетний менеджер NuGet.
- Підтримка роботи з більшістю поширених систем баз даних: MS SQL Server, MySQL, Postgres, MongoDB
- Фреймворк побудований із набору відносно незалежних компонентів. Є можливість використати вбудовану реалізацію цих компонентів, або розширити їх за допомогою механізму спадкування, або створити і застосовувати свої компоненти зі своїм функціоналом.
- Багатий інструментарій для розробки програм. Як інструментарій розробки ми можемо використовувати таке середовище розробки з багатим функціоналом як Visual Studio від компанії Microsoft.
- Також можна використовувати для розробки середовище Rider від компанії JetBrains.
- Крім того, наявне оснащення .NET CLI дозволяє створити і запускати проекти ASP.NET в консолі. І таким чином, для написання коду можна використовувати звичайний текстовий редактор, наприклад, Visual Studio Code.

2.2 Обрана IDE

Microsoft Visual Studio[17]

[14] Microsoft Visual Studio є інтегрованим середовищем розробки. По суті, це програма, яка дозволяє розробляти, писати та редагувати код.

Visual Studio робить більше, ніж більшість, дозволяючи використовувати практично будь-яку мову кодування. Серед інших, до них належать:

- C, C++, C++/CLI
- .NET
- JavaScript
- TypeScript
- XML
- XSLT
- HTML
- CSS

Інші, такі як Python, Ruby, Node.js і N, доступні через плагіни. Цей майже повний список означає, що у вас є широкий спектр варіантів. Microsoft Visual Studio має на меті бути повною та вичерпною для всіх ваших потреб у кодуванні.

Для цього він працює з вами на чотирьох рівнях: розробка, налагодження, тестування та спільна робота.

Розробка

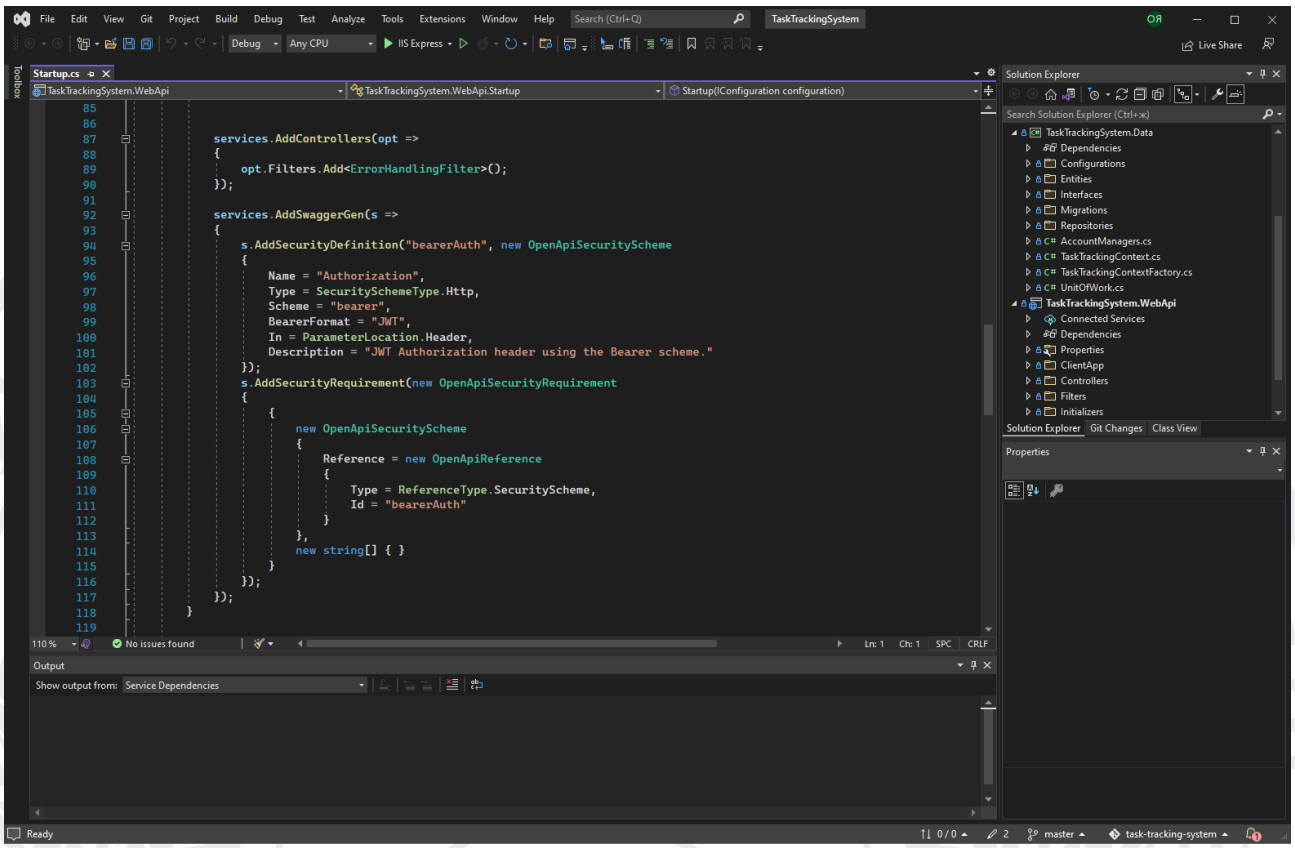

Рис. 14 – Розробка за допомогою Visual Studio

- Visual Studio пропонує вказівки та допомогу під час кодування, незалежно від мови.
- IntelliSense описує APIS під час введення тексту та використовує автозавершення для прискорення кодування
- Програма відстежує, де ви перебуваєте у процесі кодингу, навіть коли ви шукаєте інші частини коду
- Функція «Знайти всі посилання» дозволяє групувати, фільтрувати та здійснювати пошук у результатах

- Code Lens дозволяє вам зрозуміти структуру виклику вашого коду та перейти до пов’язаних функцій, а також дасть вам знати, хто останній редагував код
- Піктограми лампочок повідомляють вам, коли вам потрібно виправити поширену помилку кодування, навіть коли ви вводите код вперше
- Список помилок об’єднує всі ваші проблеми з кодуванням в один простір, тому ви можете легко вирішити всі проблеми
- Code Link може шукати потенційні рішення складних проблем
- Visual Studios виконує всю основну роботу з рефакторингу за вас у міру розвитку вашого проекту

Налагодження (Debugging)[18]

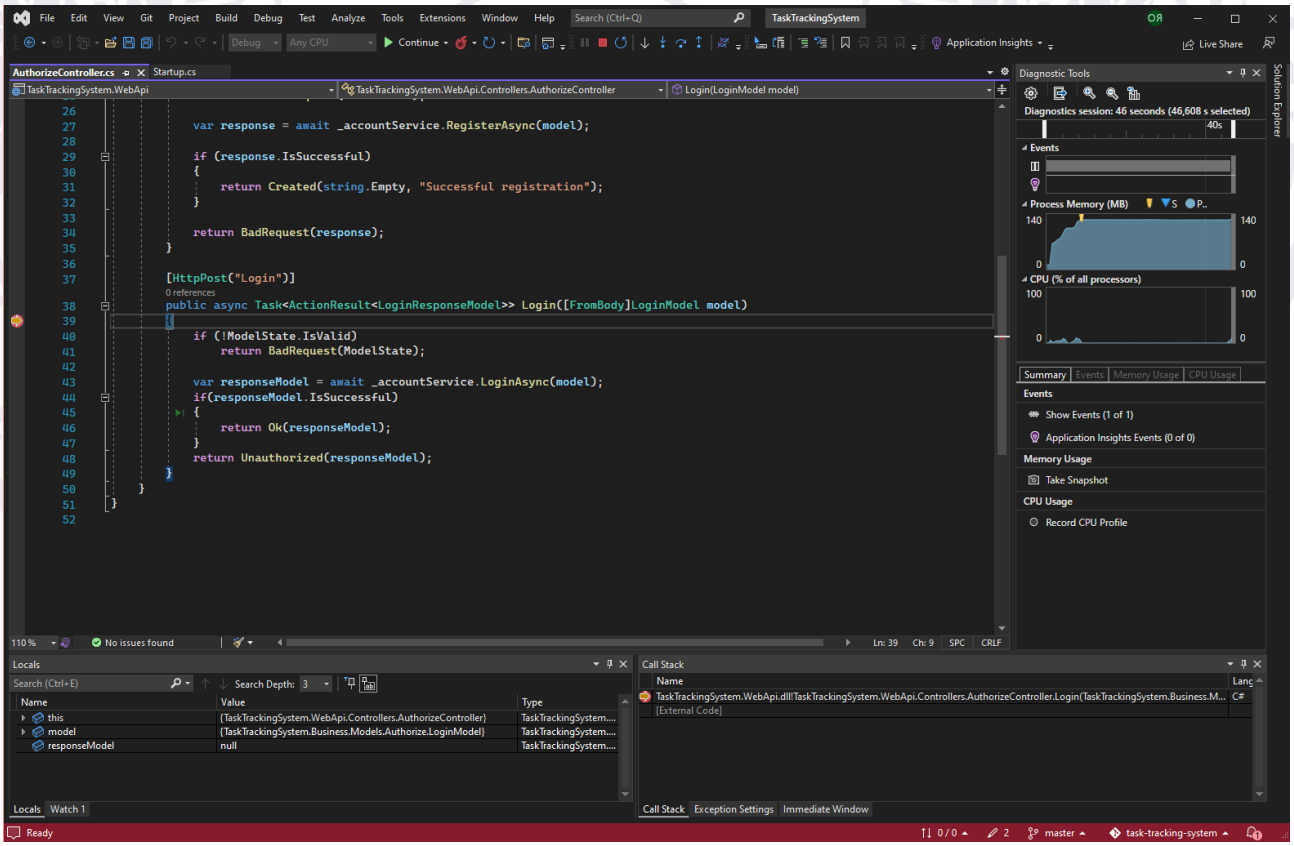

Рис. 14 – Процес налагодження використовуючи Visual Studio

- Налагодження працює, навіть якщо ви кодуєте кількома мовами
- Ви можете налагоджувати незалежно від того, де працює ваш код (додатки Windows, Android, Azure, iOS тощо).
- Visual Studio пропонує вам повний контроль над налагодженням і дозволяє вибирати, де ви хочете призупинити потоки та перевірити код
- Програма також пропонує вам гнучкість у перевірці коду, надаючи вам можливість переглядати змінні та складні вирази в будь-якому місці коду.
- Ви можете отримувати сповіщення, коли трапляються помилки
- Програма дозволяє легко перевіряти код у кількох потоках
- PerfTips та інструменти діагностики дозволяють дізнатися більше про продуктивність вашого коду та характеристики пам’яті

Тестування

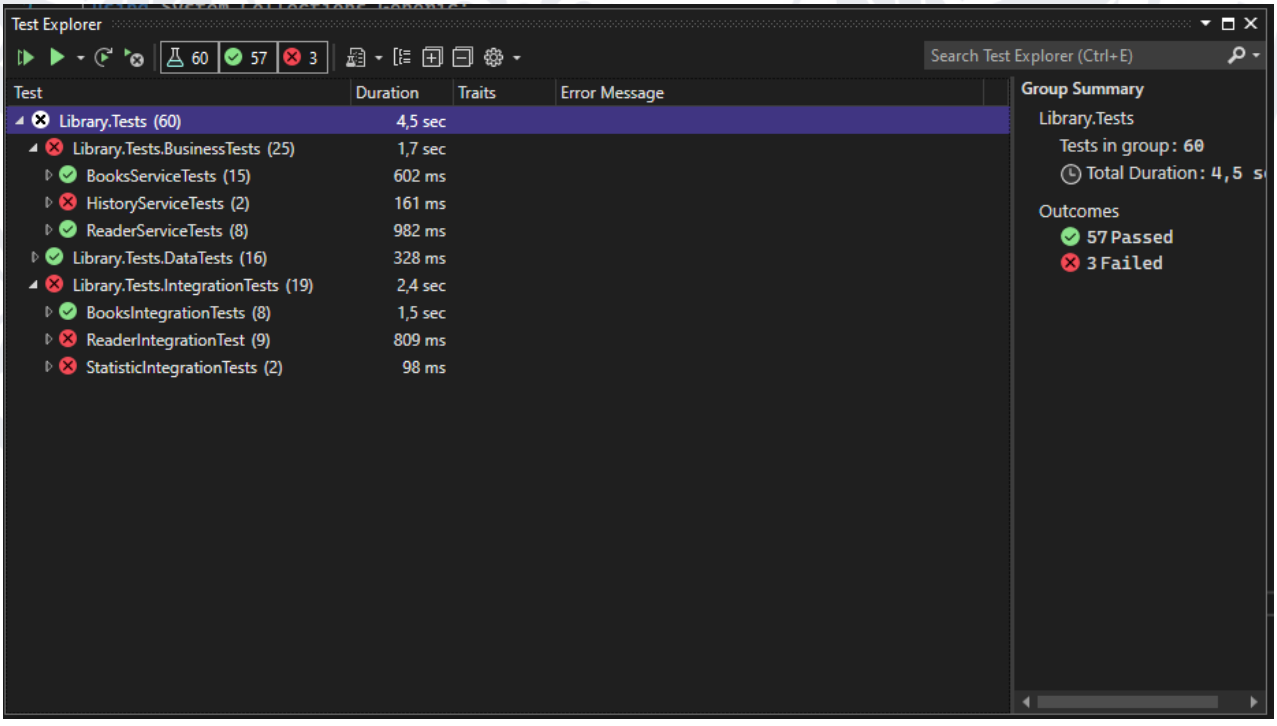

Рис. 14 – Інструмент для тестування у Visual Studio

- Visual Studio надає велику кількість шаблонів і фреймворків, щоб була змога могли писати, виконувати та налагоджувати модульні тести
- IntelliTest зменшує зусилля, необхідні для створення та підтримки модульних тестів
- Модульне тестування в реальному часі дозволяє вам переконатися, що ваші зміни не порушують тести
- Тестування користувальницького інтерфейсу дозволяє вам управляти вашим додатком через інтерфейс користувача
- У великому масштабі ви можете протестувати свій код для сотень тисяч одночасних користувачів з усього світу
- Є можливість бачити результати тестування навіть під час організації, виконання та налагодження тесту. Ви можете автоматично запускати тести після кожної збірки

2.3 База даних

Як сервер для бази даних був обраний Microsoft SQL Server[19]. Сервер Microsoft SQL[20] або більш відомий як сервер MS SQL — це сервер реляційної бази даних , розроблений корпорацією Microsoft. Сервер бази даних - це в основному програма бази даних, яка використовується для зберігання даних, а інші програмні програми отримують і зберігають дані за допомогою певної мови, яка у випадку сервера MS SQL називається SQL (Structured Query Language).

Основним інструментом інтерфейсу для MS SQL Server є SQL Server Management Studio (SSMS), який підтримує 64-розрядне та 32-розрядне операційне середовище. MS Server підтримує ANSI SQL. ANSI SQL — це стандартна мова структурованих запитів або SQL. Хоча MS SQL Server має

власне застосування мови SQL, тобто Transact-SQL або T-SQL. Власна мова Microsoft, T-SQL, додатково надає можливості представлення збереженої процедури, обробки винятків, змінних тощо.

До основних преваг можна віднести:

- Просте встановлення - Усі продукти Microsoft легко встановити за допомогою процедури встановлення в один клік та читабельного графічного інтерфейсу з великою кількістю інструкцій для неспеціаліста. MS SQL Server містить усі ці характеристики і був надзвичайно зручним інтерфейсом встановлення.
- Покращена продуктивність - Сервер MS SQL має чудові можливості стиснення та шифрування, що призводить до покращення функцій зберігання та пошуку даних.
- Безпека - сервер MS SQL вважається одним з найбільш безпечних серверів баз даних зі складними алгоритмами шифрування, що робить практично неможливим зламати рівні безпеки, які накладаються користувачем. Сервер MS SQL не є сервером баз даних з відкритим вихідним кодом , який знижує ризик атак на сервер бази даних.
- Кілька випусків і варіації цін - перевагою сервера MS SQL є те, що він доступний у кількох версіях, щоб задовольнити потреби величезних організацій корпоративного сектору для домашніх користувачів. Діапазон цін також варіюється, що дозволяє будь-кому придбати продукт, який відповідає його ціновому діапазону.
- Відмінний механізм відновлення та відновлення даних - сервер MS SQL повністю усвідомлює важливість ваших даних. Отже, MS SQL Server містить функції, які необхідні для відновлення даних, які були втрачені або пошкоджені. Можна відновити повну базу даних за допомогою деяких розширених інструментів відновлення, які містяться в базі даних MS SQL Server. MS SQL керує зберіганням даних за допомогою свого основного

компонента, яким є SQL Server Database Engine. Безпека, обробка та зберігання даних регулюються сервером**.** Database Engine може виконувати вимоги та запити. Він може керувати транзакціями, індексами, файлами тощо.

Висновок до розділу 2

У даному розділі розглянуті інструменти та технології для сучасної та ефективної розробки Web Api. Розглянуто основний інструментарій та переваги вибраного серверу баз даних.

РОЗДІЛ 3
РОЗРОБКА ТА АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

3.1 Архітектура

REST (Representational State Transfer)[22]

REST – це найпопулярніший у сьогодення архітектурний стиль, набір правил мережевих протоколів, які надають доступ до інформаційних ресурсів. Щоб веб-апі можна було назвати RESTful воно повинно відповідати декільком правилам:

- Клієнт-сервер, тобто спілкування шляхом запитів клієнтів до серверу та відповідей, якщо це необхідно, від нього ж. Простими словами в нас є сервер, тобто комп'ютер який має необхідні ресурси, наприклад веб апі з реалізованими HTTP запитами через які ми можемо доступатись до потрібної нам інформації та є клієнт, наприклад мобільний додаток через який ми хочемо отримати певну інформацію. Цей мобільний додаток відправляєш певний HTTP запит нашому серверу, а він в свою чергу, на основі отриманих даних генерує і відправляє відповідь клієнту.
- Уніфікований інтерфейс, задля зменшення зв'язності між компонентами і сервісами, які через це в перспективі буде досить легко модифікувати і змінювати. Уніфікованість полягає в чіткій визначеності в якому форматі буде здійснений обмін даними між клієнтом та сервером. Частіше всього обмін даними відбувається шляхом надсилання JSON, XML або HTML файлів. Клієнт має мати певний набір запитів для взаємодії з ресурсами серверу, та отримувати чіткі та зрозумілі відповіді на свої запити. Клієнтом отримує доступ до REST API за допомогою гіперпосилань.
- Відсутність стану, тобто при взаємодії між сервером та клієнтом, клієнт в своєму запиті повинен надсилати необхідну інформацію для його обробки

серверу, не покладаючись на те, що сервер знає цю інформацію з минулих запитів.

- Багатошарова архітектура, тобто розділення складних компонентів на декілька більш простих проміжних слоїв в певній ієрархії

Багатошарова архітектура (Multilayered Architecture)[21]

Багатошарова архітектура є однією з найбільш розповсюджених в сучасних практиках розробки додатків. Вона дозволяє зменшити залежність між певними компонентами, оскільки рішення ділиться на декілька шарів(рівнів), кожен з яких відповідає спектр функцій в ієрархії. Відповідно процес розробки, внесення змін, модифікування, тестування та підтримки написаного коду стає набагато легше, оскільки ми будемо вносити зміни в необхідний нам шар, не ламаючи нічого в інших.

Цю архітектура також часто називають n-layer, де n кількість шарів. Однією з найбільш використовуваних є тришарова архітектура, в якій програма поділяється на три шари:

Шар доступу до даних (Data Access Layer), який містить в собі основні сутності, якій повністю описують об’єкті, що зберігаються в базі даних. Частими практиками для цього шару є реалізація патерну Unit of Work та Repository відповідно. А також класи контексту бази даних, наприклад DbContext, якщо ми працюємо з Entity Framework, та інші класи які напряму взаємодіють з базою даних.

Шар бізнес-логіки (Business Logic Layer), який є проміжним шар між шаром доступу до даних та шаром представлення. Його суть полягає в отримані даних з шару представлення, реалізації всієї необхідної логіки, тобто певні обчислення, перевірка коректності даних тощо, та у взаємодії з шар доступу даних задля отримання результату та подальшої передачі його на шар представлення. Він

відповідає за перетворення(mapping) сутності яка надходить з бази даних у модель, яка передається далі вище по ієрархії.

Шар представлення (Presentation Layer), який відповідає за взаємодії з клієнтом. Тобто надає певний інтерфейс, у випадку API контролери, за допомогою якого у додатку є змога отримати дані від клієнту та повернення результату запиту. Він напряму взаємодіє з шаром бізнес-логіки.

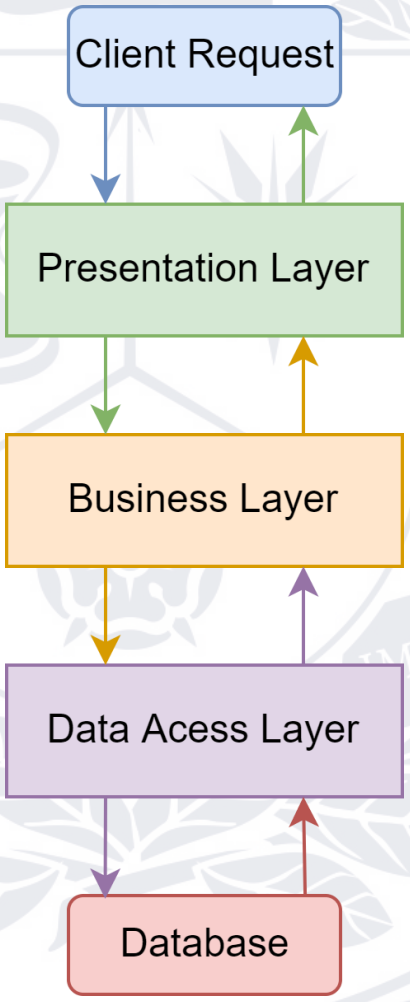

Рис. 15 – Тришарова архітектура

Важливо, щоб верхні рівні не мали доступу до нижніх, тобто шар представлення не повинен напряму звертатися до рівня доступу до даних.

3.2 Створення бази даних та реалізація Data Acess Layer

Варто зазначити, що була вибрана реляційна модель бази даних. Принцип якої полягає в визначених зв’язках між собою наборами даних. Набір даних в даній модель представляє собою таблицю, в якій зберігаються дані та яка складається з стовпців та рядків. В стовпці зберігається найменований тип даних, а в рядках відповідно значення. В кожній таблиці, один з стовпців повинен відповідати за унікальний ідентифікатор, первинний ключ, по значенню якого знаходиться унікальний об’єкт даних цієї таблиці, та за допомогою яких будуються необхідні нам зв’язки між таблиця, в даному випадку первинний ключ виступає так званим зовнішнім ключом.

Проектування та створення бази даних було реалізовано використовуючи підхід Code First[24] (Спершу код). Його суть полягає в тому, що ми створюємо моделі об’єктів, тобто пишеться код створюючи класи, а ORM (Object Relational Mapping), в нашому випадку Entity Framework Core[23], побудує потрібні нам таблиці та зв’язки в базі даних. Задля застосування цього інструменту в додатку, встановимо відповідні пакети використовуючи NuGet Package Manager:

- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Design
- Microsoft.EntityFrameworkCore.SqlServer
- Microsoft.EntityFrameworkCore.Tools

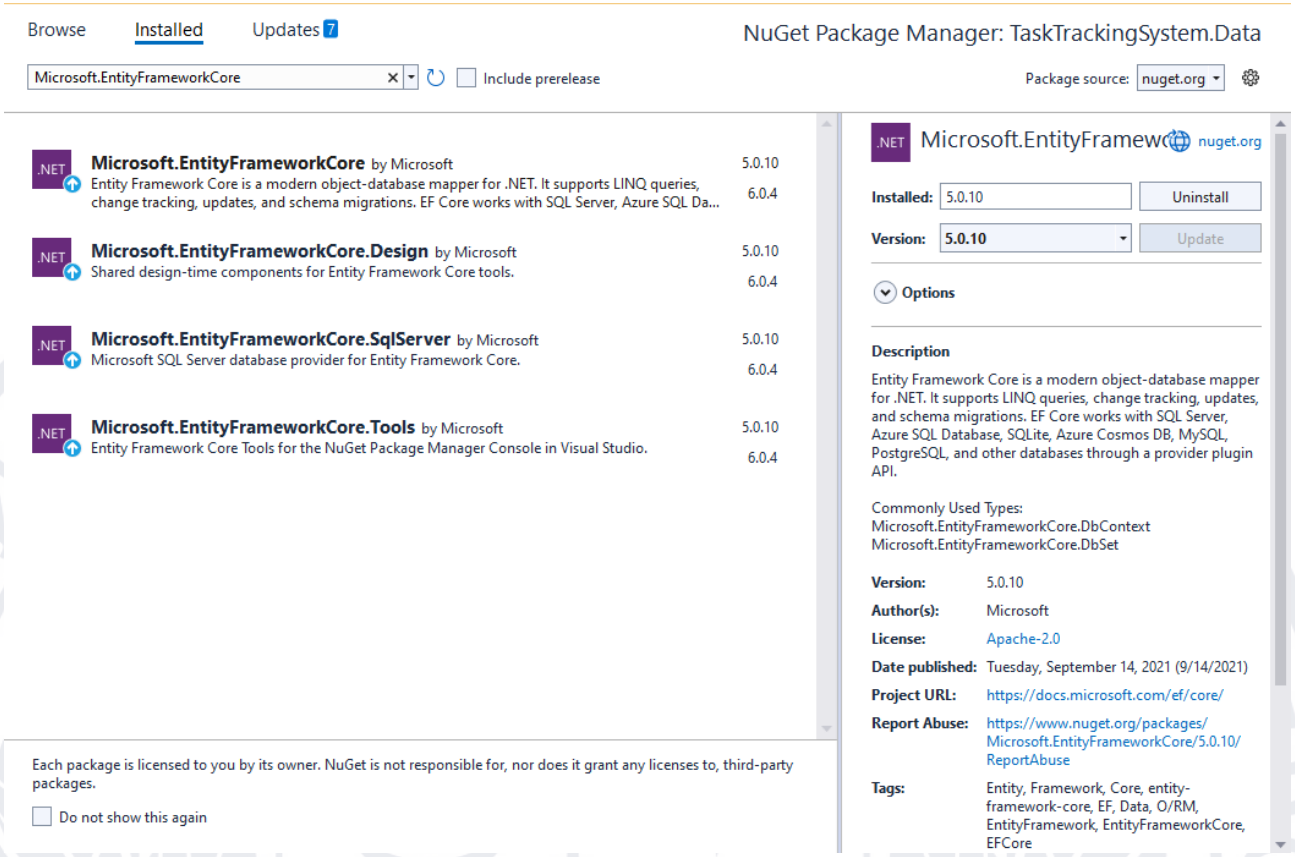

Рис.16 – NuGet пакети EntityFrameworkCore

Основна необхідна функціональність знаходиться в просторі імен Microsoft.EntityFrameworkCore:

- DbContext – відповідає за контекст даних для взаємодії з базою даних
- DbSet<TEntity> - відповідає за набір сутностей, які зберігаються в базі даних
- ModelBuilder – відповідає за детальне налаштування сутностей в базі даних

Варто зазначити, що в додатку використаний інший клас, що відповідає за контекст даних IdentityDbContext. В більшості він схожий на звичайний DbContext за виключенням деяких відміностей, про які буде йтись в розділі про авторизацію та аутентифікацію.

Детальніше про кожну створену сутність:

```
1   using Microsoft.AspNetCore.Identity;
2
3   namespace TaskTrackingSystem.Data.Entities
4   {
        13 references
5       public class Account : IdentityUser
6       {
7
8       }
9   }
```

Рис.17 – Сутність Account

Сутність **Account** наслідується від класі IdentityUser взятого з простору імен Microsoft.AspNetCore.Identity. Вона відповідно містить усі необхідні для акаунту дані, такі як e-mail, пароль та унікальний ідентифікатор користувача. Відповідно EF Core побудував таку таблицю:

Name	Data Type	Allow Nulls	Default
Id	nvarchar(450)	☐	
UserName	nvarchar(256)	☑	
NormalizedUserName	nvarchar(256)	☑	
Email	nvarchar(256)	☑	
NormalizedEmail	nvarchar(256)	☑	
EmailConfirmed	bit	☐	
PasswordHash	nvarchar(MAX)	☑	
SecurityStamp	nvarchar(MAX)	☑	
ConcurrencyStamp	nvarchar(MAX)	☑	
PhoneNumber	nvarchar(MAX)	☑	
PhoneNumberConfirmed	bit	☐	
TwoFactorEnabled	bit	☐	
LockoutEnd	datetimeoffset(7)	☑	
LockoutEnabled	bit	☐	
AccessFailedCount	int	☐	
		☐	

- **Keys** (1)
  - PK_AspNetUsers (Primary Key, Clustered: Id)
- **Check Constraints** (0)
- **Indexes** (2)
  - EmailIndex (NormalizedEmail)
  - UserNameIndex (Unique: NormalizedUserName)
- **Foreign Keys** (0)
- **Triggers** (0)

```sql
CREATE TABLE [dbo].[AspNetUsers] (
    [Id]                   NVARCHAR (450)     NOT NULL,
    [UserName]             NVARCHAR (256)     NULL,
    [NormalizedUserName]   NVARCHAR (256)     NULL,
    [Email]                NVARCHAR (256)     NULL,
    [NormalizedEmail]      NVARCHAR (256)     NULL,
    [EmailConfirmed]       BIT                NOT NULL,
    [PasswordHash]         NVARCHAR (MAX)     NULL,
    [SecurityStamp]        NVARCHAR (MAX)     NULL,
    [ConcurrencyStamp]     NVARCHAR (MAX)     NULL,
    [PhoneNumber]          NVARCHAR (MAX)     NULL,
    [PhoneNumberConfirmed] BIT                NOT NULL,
    [TwoFactorEnabled]     BIT                NOT NULL,
    [LockoutEnd]           DATETIMEOFFSET (7) NULL,
    [LockoutEnabled]       BIT                NOT NULL,
    [AccessFailedCount]    INT                NOT NULL,
    CONSTRAINT [PK_AspNetUsers] PRIMARY KEY CLUSTERED ([Id] ASC)
);


GO
CREATE NONCLUSTERED INDEX [EmailIndex]
    ON [dbo].[AspNetUsers]([NormalizedEmail] ASC);


GO
CREATE UNIQUE NONCLUSTERED INDEX [UserNameIndex]
    ON [dbo].[AspNetUsers]([NormalizedUserName] ASC) WHERE ([NormalizedUserName] IS NOT NULL);

```

Рис.18 – SQL таблиця сутності Account

Сутність **BaseEntity**, абстрактна сутність, яка містить в собі унікальний ідентифікатор та від якої будуть наслідувати усі сутності які зберігаються в базі даних.

```csharp
namespace TaskTrackingSystem.Data.Entities
{
    7 references
    public abstract class BaseEntity
    {
        22 references
        public int Id { get; set; }
    }
}
```

Рис.19 – Сутність BaseEntity

Сутність **Employee**, яка відповідно насліднується від сутності **BaseEntity**. Унікальним ідентифікатором, цієї сутності виступає зовнішній ключ сутності **Account**, тому було застосовано механізм «приховання» для того, щоб тип цього ідентифікатора був відповідним до типу ідентифікатора сутності **Account**, тобто типу string, в той час, як у сутності **BaseEntity** тип int. А для подальшого застосування принципу поліморфізму необхідно, щоб усі сутності, за допомогою який відбувається процес спілкування з базою даних наслідувались від сутності **BaseEntity.** Також всередині сутності **Employee,** варто уточнити, що саме на рівні сутності, зберігається сутність **EmployeeProfile** та колекція сутностей **EmployeeWork** про які буде йтись далі. Зв'язки між цими сутностями якраз буде створювати **Entity Framework Core** за принципами побудови реляційних баз даних.

```
1   using System.Collections.Generic;
2   using System.ComponentModel.DataAnnotations;
3   using System.ComponentModel.DataAnnotations.Schema;
4
5   namespace TaskTrackingSystem.Data.Entities
6   {
        20 references
7       public class Employee : BaseEntity
8       {
9           [Key]
10          [ForeignKey("Account")]
            2 references
11          public new string Id { get; set; }
12
            7 references
13          public Account Account { get; set; }
14
            6 references
15          public EmployeeProfile EmployeeProfile { get; set; }
16
            2 references
17          public ICollection<EmployeeWork> EmployeeWorks { get; set; }
18      }
19  }
20
```

Рис.20 – Сутність Employee

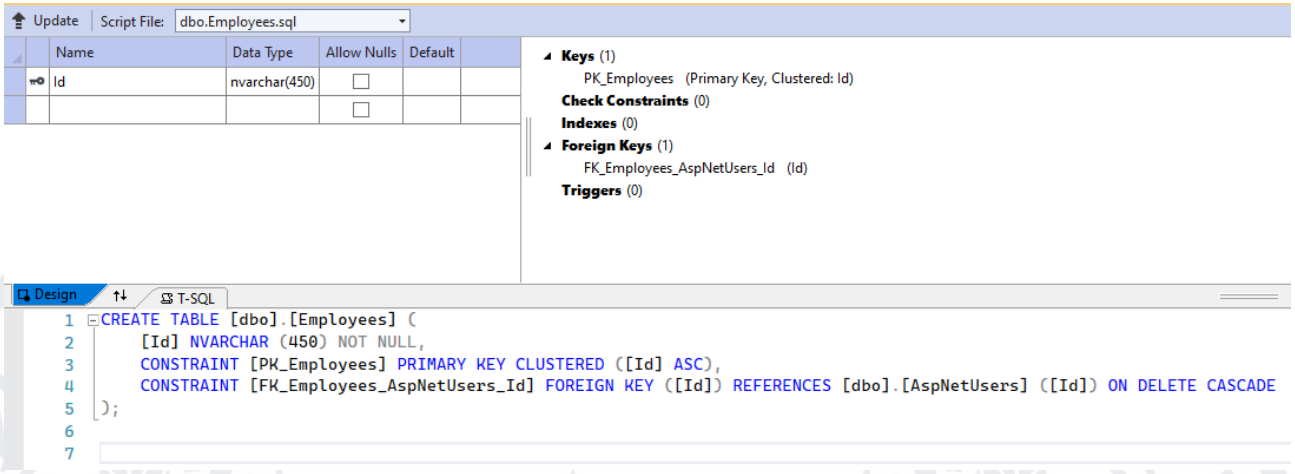

Рис.21 – SQL таблиця сутності Employee

Подивившись на SQL реалізацію цієї таблиці, можна побачити, що вона містити лише унікальний ідентифікатор, який якраз виступає зовнішнім ключем з посиланням на Id в таблиці **AspNetUsers(Account)**. Тобто ці 2 таблиці мають єдиний унікальний ідентифікатор. Варто також звернути увагу, що при побудові такого зв'язку потрібно зазначити каскадне видалення(якщо воно необхідно), тобто при видаленні таблиці сутності з таблиці **Employee**, також буде видалена відповідна сутність в таблиці **AspNetUsers(Account)**. Це реалізовано за подогою ключових слів **ON DELETE CASCADE** з посилання на потрібний нам Id.

Сутність **EmployeeProfile** описує профіль користувача, тобто містить ім'я, прізвище, номер телефону. Унікальний ідентифікатор відсутній, адже це допоміжна таблиця, лише знаходиться сутність **Employee** та Id цієї сутності, задля того, щоб EntityFramework Core зміг коректно побудувати зв'язки між таблицями. Ця сутність буде також каскадно видалена при видаленні сутності **Employee.**

```
using System.ComponentModel.DataAnnotations;

namespace TaskTrackingSystem.Data.Entities
{
    4 references
    public class EmployeeProfile
    {
        [Key]
        0 references
        public string EmployeeId { get; set; }

        1 reference
        public string Name { get; set; }

        1 reference
        public string Surname { get; set; }

        1 reference
        public string Phone { get; set; }

        0 references
        public Employee Employee { get; set; }
    }
}
```

Рис.22 – Сутність EmployeeProfile

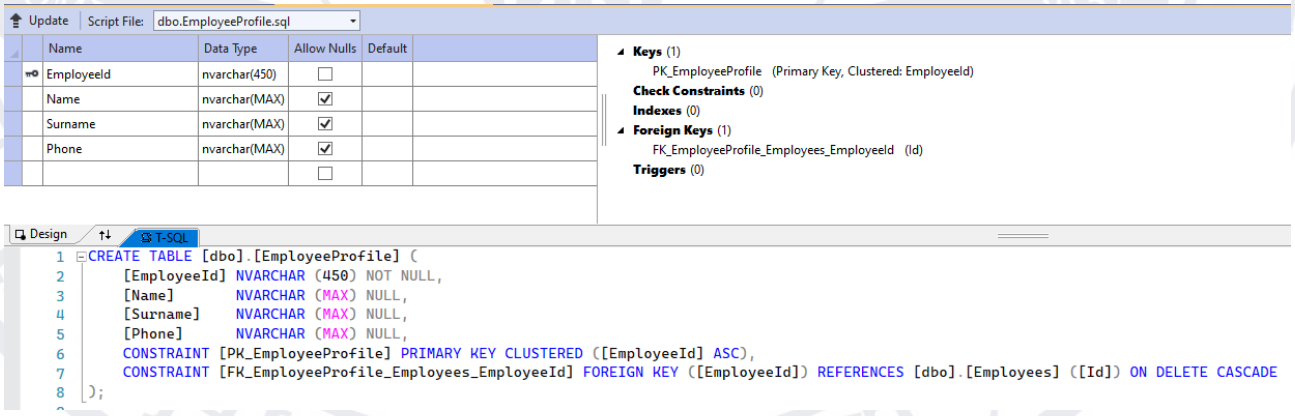

Рис.23 – SQL таблиця сутності EmployeeProfile

Сутність **EmployeeWork**, яка виступає сполучною ланкою багатьох сутностей, які буде описано згодом. Саме ця сутність має унікальний

ідентифікатор, тобто вона наслідує сутність **BaseEntity**, адже вона по своїй суті вона описує конкретного користувача на конкретному проєкті. Основна інформація це список завдань, це колекція сутностей **ProjectTask**, які закріплені саме за цим користувачем та його позицію на проєкті, тобто сутність **EmployeePosition**. Допоміжними параметрами виступає посилання на сутності: **Employee**, **Project**, по яким Entity Framework Core збудував зв'язки в базі даних.

```
1   using System.Collections.Generic;
2   using System.ComponentModel.DataAnnotations;
3
4   namespace TaskTrackingSystem.Data.Entities
5   {
        18 references
6       public class EmployeeWork : BaseEntity
7       {
8           [Required]
            4 references
9           public string EmployeeId { get; set; }
10
11          [Required]
            2 references
12          public int ProjectId { get; set; }
13
14          [Required]
            1 reference
15          public int EmployeePositionId { get; set; }
16
            3 references
17          public EmployeePosition EmployeePosition { get; set; }
18
            5 references
19          public Project Project { get; set; }
20
            4 references
21          public Employee Employee { get; set; }
22
            1 reference
23          public ICollection<ProjectTask> ProjectTasks { get; set; }
24      }
25  }
```

Рис.24 – Сутність EmployeeWork

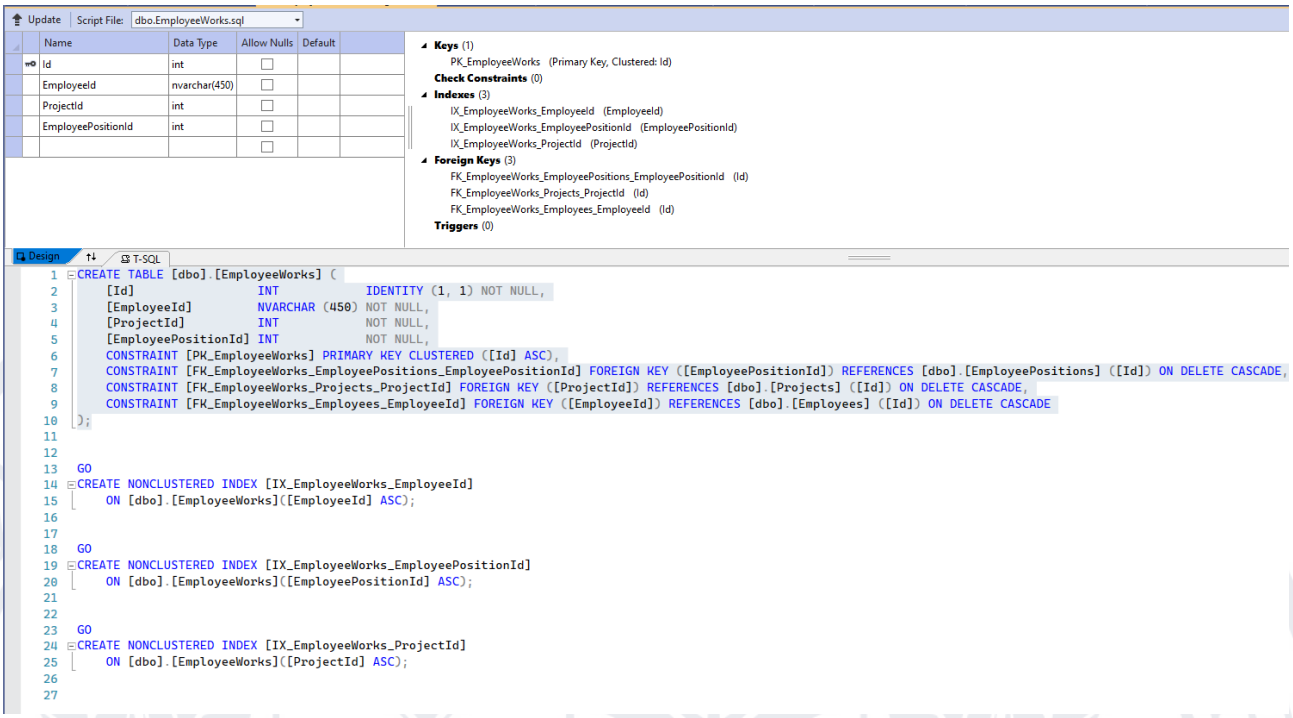

Рис.25 – SQL таблиця сутності EmployeeWork

Сутність **EmployeePosition** містить унікальний ідентифікатор, тобто наслідується від сутності **BaseEntity** та ім'я, яке описує позицію користувача на проєкті. Також найменування позиції задане, як альтернативний первинний ключ, відповідно воно має бути унікальне.

```
1   using System.ComponentModel.DataAnnotations;
2
3   namespace TaskTrackingSystem.Data.Entities
4   {
        13 references
5       public class EmployeePosition : BaseEntity
6       {
7           [Required]
            4 references
8           public string Name { get; set; }
9       }
10  }
```

Рис.26 – Сутність EmployeePosition

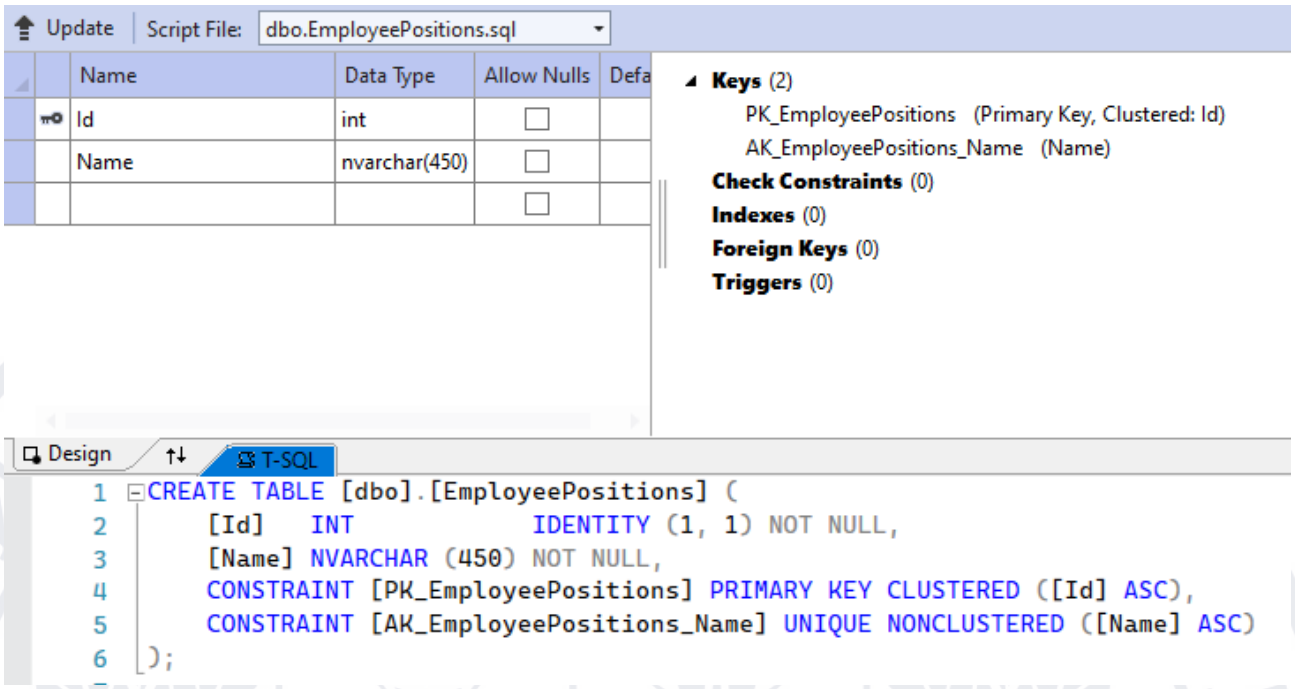

Рис.27 – SQL таблиця сутності EmployeePosition

Сутність **Project** містить унікальний ідентифікатор, тобто наслідується від сутності **BaseEntity.** Описує ім’я проєкту, яке також виступає альтернативним первинним ключом таблиці та повинно бути унікальним. Інформацію про сам проєкт, дату його створення, дату закриття, булеве значення чи закритий проєкт та списки завдань, тобто сутність **ProjectTask**, та користувачів на цьому проєктів, тобто сутність **EmployeeWork.**

```
1   using System;
2   using System.Collections.Generic;
3   using System.ComponentModel.DataAnnotations;
4
5   namespace TaskTrackingSystem.Data.Entities
6   {
        18 references
7       public class Project : BaseEntity
8       {
9           [Required]
            4 references
10          public string Name { get; set; }
11
12          [Required]
            1 reference
13          public string Description { get; set; }
14
15          [Required]
            1 reference
16          public DateTime CreationDate { get; set; }
17
            2 references
18          public DateTime ClosingDate { get; set; }
19
            4 references
20          public bool IsClosed { get; set; }
21
            2 references
22          public ICollection<ProjectTask> ProjectTasks { get; set; }
23
            3 references
24          public ICollection<EmployeeWork> EmployeeWorks { get; set; }
25      }
26  }
```

Рис.28 – Сутність **Project**

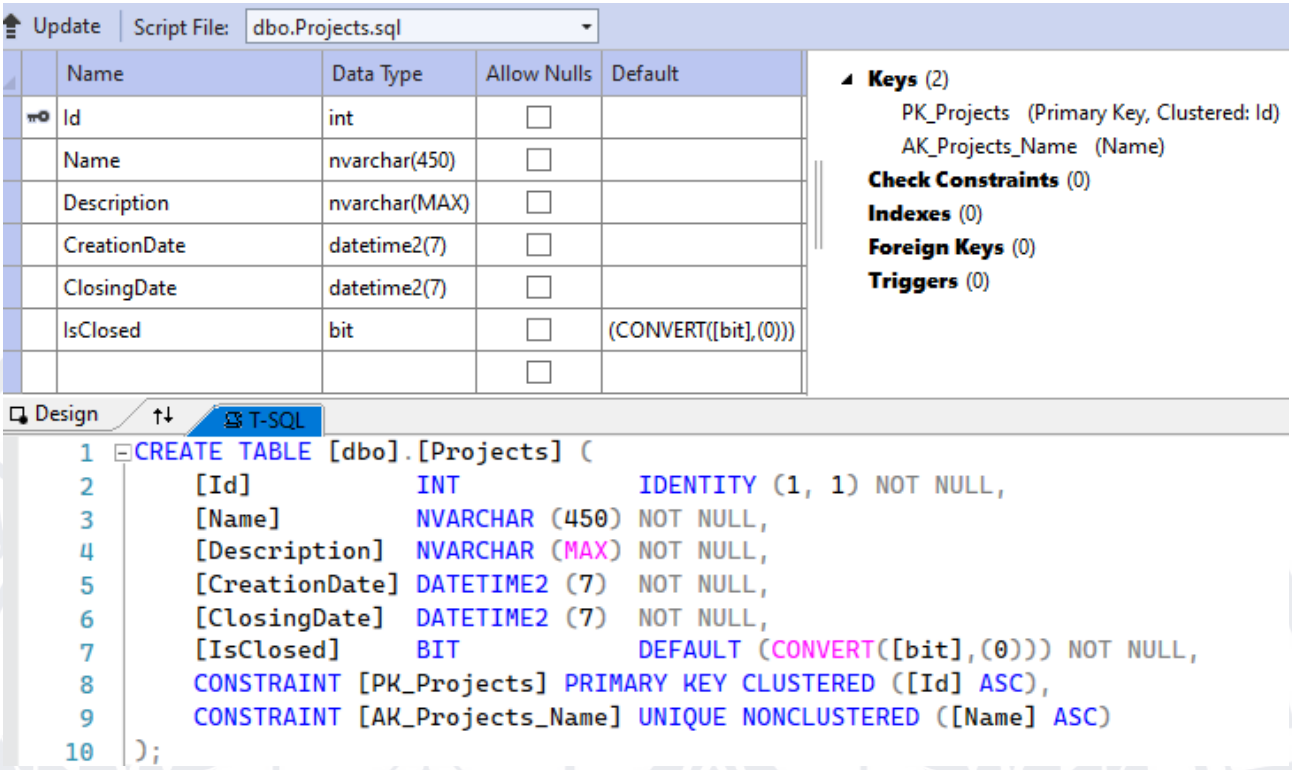

Рис.29 – SQL таблиця сутності Project

Сутність **ProjectTask** наслідує сутність **BaseEntity**, тобто має унікальний ідентифікатор. Містить назву самого завдання, його опис, статус завдання, тобто сутність **TaskStatus**, дата зміни статусу завдання. Посилання на сутність **EmployeeWork**, тобто на конкретного користувача на конкретному проєкті, до якого прив’язується це завдання. Також варто зазначити, що цей параметр заданий як *nullable*, тобто який можеш дорівнювати значенню null, що означає пусте значення. Це зроблення задля того, щоб при видаленні сутності **EmployeeWork**, каскадне видалення сутності **ProjectTask**. А також посилання на сутність **Project**, до якого прив’язується завдання. Тут вже, при видаленні проєкту буде каскадно видалятись і всі завдання прив’язанні до нього.

```
1   using System;
2   using System.ComponentModel.DataAnnotations;
3
4   namespace TaskTrackingSystem.Data.Entities
5   {
        17 references
6       public class ProjectTask : BaseEntity
7       {
8           [Required]
            4 references
9           public string Title { get; set; }
10
11          [Required]
            2 references
12          public string Info { get; set; }
13
14          [Required]
            2 references
15          public int TaskStatusId { get; set; }
16
17          [Required]
            2 references
18          public DateTime SetStatusDate { get; set; }
19
            5 references
20          public int? EmployeeWorkId { get; set; }
21
22          [Required]
            3 references
23          public int ProjectId { get; set; }
24
            5 references
25          public ProjectTaskStatus TaskStatus { get; set; }
26
            2 references
27          public Project Project { get; set; }
28
            2 references
29          public EmployeeWork EmployeeWork { get; set; }
30      }
31  }
```

Рис.30 – Сутність ProjectTask

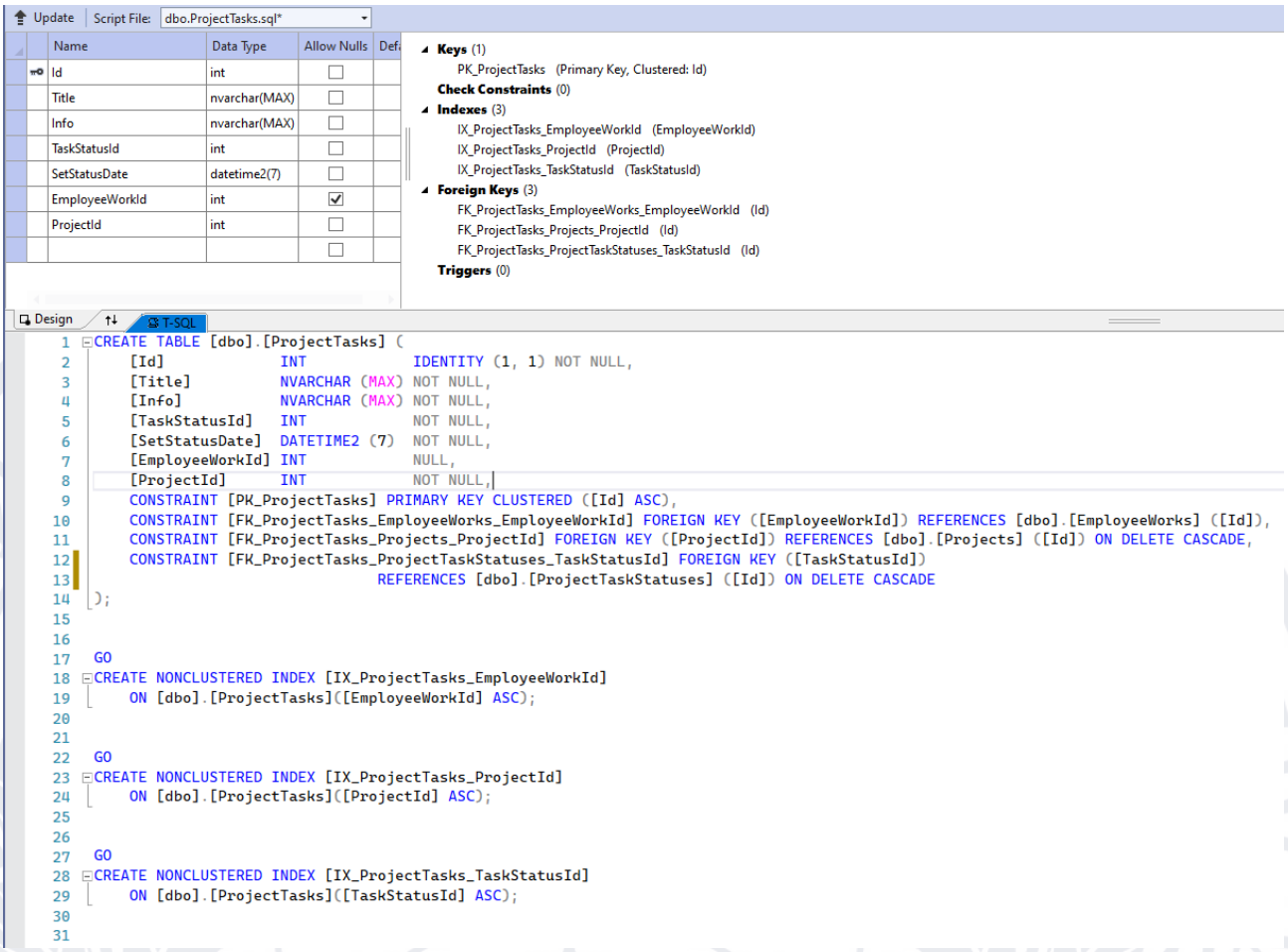

Рис.31 – SQL таблиця сутності ProjectTask

Сутність **ProjectTaskStatus,** яка містить унікальний ідентифікатор, тобто наслідується від сутності **BaseEntity**. Та ім’я, яке описує статус завдання відповідно до процесу роботи. Також найменування статусу задане, як альтернативний первинний ключ, це було зроблено, щоб уникнути дублювання імен.

```
1   using System.ComponentModel.DataAnnotations;
2
3   namespace TaskTrackingSystem.Data.Entities
4   {
        19 references
5       public class ProjectTaskStatus : BaseEntity
6       {
7           [Required]
            10 references
8           public string Name { get; set; }
9       }
10  }
```

Рис.32 – Сутність ProjectTaskStatus

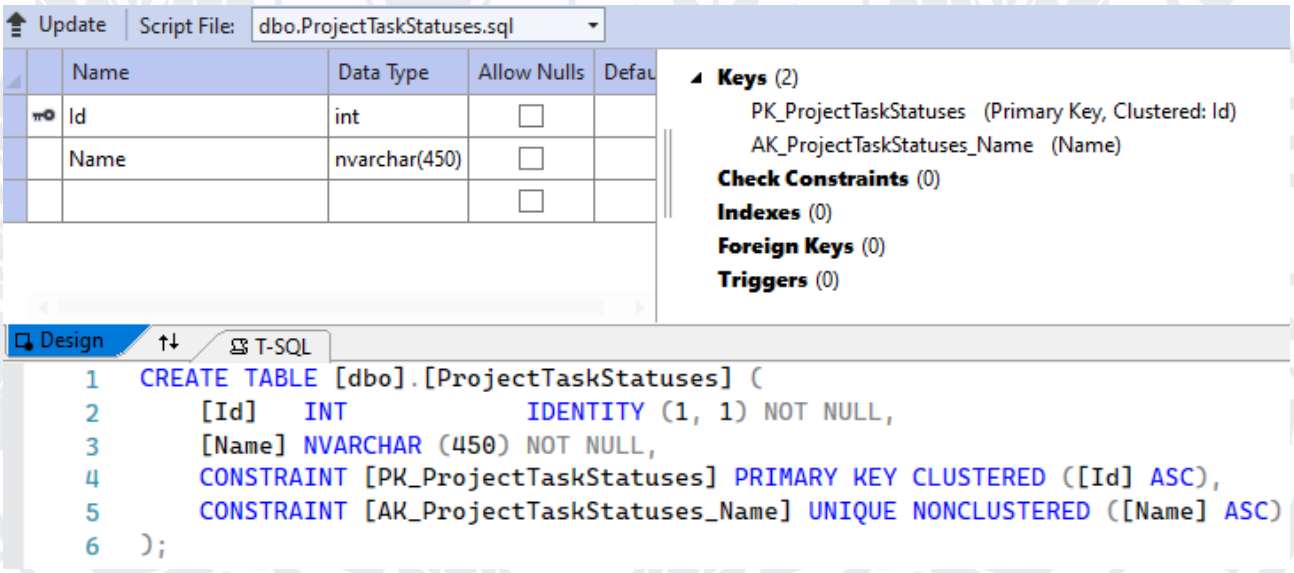

Рис.33 – SQL таблиця сутності ProjectTaskStatus

В налаштуванні контексту даних використовується клас IdentityDbContext з простору імен Microsoft.AspNetCore.Identity.EntityFrameworkCore, це спеціальний NuGet пакет для реалізації авторизації та аутентифікації. Клас IdentityDbContext відрізнається від звичайного DbContext тим, що він повинен мати сутність який буде відповідати за аккаунт користувача, в нашому випадку

Account та одразу в собі містить таблиці зі всіма необхідними даним для реалізації цього механізму.

Для всіх **BaseEntity** виділений DbSet, це свого роду доступ до таблиці.

```
1   using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
2   using Microsoft.EntityFrameworkCore;
3   using TaskTrackingSystem.Data.Configurations;
4   using TaskTrackingSystem.Data.Entities;
5
6   namespace TaskTrackingSystem.Data
7   {
        29 references
8       public class TaskTrackingContext : IdentityDbContext<Account>
9       {
            1 reference
10          public TaskTrackingContext(DbContextOptions<TaskTrackingContext> options)
11              : base(options)
12          {
13          }
14
            0 references
15          protected override void OnModelCreating(ModelBuilder builder)
16          {
17              builder.Entity<Project>().HasAlternateKey(p => p.Name);
18              builder.Entity<EmployeePosition>().HasAlternateKey(ep => ep.Name);
19              builder.Entity<ProjectTaskStatus>().HasAlternateKey(ts => ts.Name);
20
21              builder.ApplyConfiguration(new RoleConfiguration());
22              builder.ApplyConfiguration(new TaskStatusConfiguration());
23
24              base.OnModelCreating(builder);
25          }
26
            6 references
27          public DbSet<Employee> Employees { get; set; }
28
            7 references
29          public DbSet<Project> Projects { get; set; }
30
            6 references
31          public DbSet<ProjectTask> ProjectTasks { get; set; }
32
            6 references
33          public DbSet<EmployeeWork> EmployeeWorks { get; set; }
34
            6 references
35          public DbSet<ProjectTaskStatus> ProjectTaskStatuses { get; set; }
36
            6 references
37          public DbSet<EmployeePosition> EmployeePositions { get; set; }
38      }
39  }
```

Рис.34 – Налаштування контексту даних

Реалізація шару доступу до бази даних відбувалася за допомогою патернів Unit of Work[24] та Repository[25] відповідно. В додатках такого типу це доволі часта практика. Патерн Repository містить логіку роботи з джерелами даних. Патерн Unit of Work спрощує роботи з різними репозиторіями та забов'язують всі ропозиторії використовувати спільний контекст даних. Для усіх основних сутностей створюється репозиторій з певним набіром методів для взаємодії з контекстом даних. Основою цих методів завжди є CRUD функціонал, тобто створення, зчитування, оновлення, видалення.

```
8   namespace TaskTrackingSystem.Data.Repositories
9   {
        3 references
10      public class TaskStatusRepository : ITaskStatusRepository
11      {
12          private readonly TaskTrackingContext _context;
13
            1 reference
14          public TaskStatusRepository(TaskTrackingContext context)
15          {
16              _context = context;
17          }
18
            6 references
19          public async Task AddAsync(ProjectTaskStatus entity)
20          {
21              await _context.ProjectTaskStatuses.AddAsync(entity);
22          }
23
            2 references
24          public void Delete(ProjectTaskStatus entity)
25          {
26              _context.ProjectTaskStatuses.Remove(entity);
27          }
28
            6 references
29          public async Task DeleteByIdAsync(int id)
30          {
31              await Task.Run(() => Delete(GetByIdAsync(id).Result));
32          }
33
            3 references
34          public IQueryable<ProjectTaskStatus> FindAll()
35          {
36              return _context.ProjectTaskStatuses.AsQueryable();
37          }
38
            20 references
39          public async Task<ProjectTaskStatus> GetByIdAsync(int id)
40          {
41
42              return await _context.ProjectTaskStatuses.FindAsync(id);
43          }
44
            2 references
45          public async Task<ProjectTaskStatus> GetByNameAsync(string name)
46          {
47              return await _context.ProjectTaskStatuses.FirstOrDefaultAsync(p => p.Name.Equals(name));
48          }
```

Рис.34 – Приклад патерну Repository

В свою чергу клас Unit of Work надає доступ до всіх репозиторіїв через окремі властивості, а також містить метод який зберігає всі зміни.

```
5   namespace TaskTrackingSystem.Data
6   {
        2 references
7       public class UnitOfWork : IUnitOfWork
8       {
9           private readonly TaskTrackingContext _context;
10
11          private EmployeeRepository _employeeRepository;
12          private ProjectRepository _projectRepository;
13          private TaskRepository _taskRepository;
14          private WorkRepository _workRepository;
15          private TaskStatusRepository _taskStatusRepository;
16          private EmployeePositionRepository _employeePositionRepository;
17
            0 references
18          public UnitOfWork(TaskTrackingContext context)
19          {
20              _context = context;
21          }
22
            9 references
23          public IEmployeeRepository EmployeeRepository
24          {
25              get
26              {
27                  if (_employeeRepository == null)
28                  {
29                      _employeeRepository = new EmployeeRepository(_context);
30                  }
31                  return _employeeRepository;
32              }
33          }
34
            13 references
35          public IProjectRepository ProjectRepository
36          {
37              get
38              {
39                  if (_projectRepository == null)
40                  {
41                      _projectRepository = new ProjectRepository(_context);
42                  }
43                  return _projectRepository;
44              }
45          }
46
            16 references
47          public ITaskRepository TaskRepository
48          {
49              get
50              {
51                  if (_taskRepository == null)
52                  {
53                      _taskRepository = new TaskRepository(_context);
54                  }
```

Рис.35 – Приклад патерну Unit of Work

3.3 Реалізація Business Logic Layer

В шарі BLL реалізована вся бізнес-логіка та всі необхідні обчислення. Він отримує дані з Data Access Layer(DAL) та передає на шар представлення тобто Presentation Logic Layer(PLL). Для цього був створений окрема бібліотека класів, в яку добавлене посилання на шар доступу до даних (DAL). Так як PLL не має мати посилання на шар DAL, були створені так звані Data transfer object(DTO). Це об’єкти через які передається необхідна інформація. За допомогою інструмента AutoMapper, ми можемо легко перетворити об’єкт одного класу в об’єкт іншого згідно заданих налаштувань.

```
namespace TaskTrackingSystem.Business.Models
{
    public class EmployeeModel
    {
        public string Id { get; set; }

        public string Name { get; set; }

        public string Surname { get; set; }

        public  string Phone { get; set; }

        public string Email { get; set; }

        public ICollection<int> EmployeeWorksIds { get; set; }
    }
}
```

Рис.35 – Приклад DTO для класу Employee

```
namespace TaskTrackingSystem.Business
{
    2 references
    public class AutomapperProfile : Profile
    {
        1 reference
        public AutomapperProfile()
        {
            CreateMap<Employee, EmployeeModel>()
                .ForMember(em => em.Id, i => i
                    .MapFrom(m => m.Id))
                .ForMember(em => em.Email, i => i
                    .MapFrom(m => m.Account.Email))
                .ForMember(em => em.Name, n => n
                    .MapFrom(e => e.EmployeeProfile.Name))
                .ForMember(em => em.Surname, n => n
                    .MapFrom(e => e.EmployeeProfile.Surname))
                .ForMember(em => em.Phone, p => p
                    .MapFrom(m => m.EmployeeProfile.Phone))
                .ForMember(em => em.EmployeeWorksIds, l =>
                    l.MapFrom(ew => ew.EmployeeWorks.Select(w => w.Id))).ReverseMap();

            CreateMap<EmployeeProfile, EmployeeProfileModel>().ReverseMap();

            CreateMap<EmployeeWork, WorkModel>().ReverseMap();
```

Рис.36 – Приклад налаштування AutoMapper

Посередництво між двома шарами відбувається за рахунок так званих класів-сервісів, в яких через конструктор впроваджена залежність від інтерфейсів раніше згаданих класів UnitOfWork та AutoMapper[26] та які імплементують логіку отримання DTO перетворення її в модель сутності та здійснення необхідних операцій з базою даних, тобто, якщо у нас йде процес створення нового користувача, то клас-сервіс у певний метод отримує відповідний DTO, який за допомогою налаштованого AutoMapper перетворюється в обє’кт сутності, потім в цьому методі йде звернення до Repository який відповідає за цю сутність та який знаходиться в об’єктів класу UnitOfWork та через відповідний метод відбувається запис в базу даних.

У випадку коли нам потрібно отримати певну інформацію з бази даних, то клас-сервіс звертається до відповідного Repository, та знову ж таки через потрібний метод отримує об'єкт сутності, який потім за допомогою AutoMapper

перетворюється в DTO та повертається шару з якого був викликаний метод, тобто на шар представлення (PLL).

```
namespace TaskTrackingSystem.Business.Services
{
    2 references
    public class EmployeeService : IEmployeeService
    {
        private readonly IUnitOfWork _uow;

        private readonly IMapper _mapper;

        0 references
        public EmployeeService(IUnitOfWork uow, IMapper mapper)
        {
            _uow = uow;
            _mapper = mapper;
        }

        1 reference
        private EmployeeProjectModel CreateEmployeeProjectModel(EmployeeWork eWork)
        {
            var tasks = _uow.TaskRepository.FindAllWithDetails().Where(t => t.EmployeeWorkId == eWork.Id);

            var tasksCount = (decimal)tasks.Count();
            var completedTasksCount = (decimal)tasks.Count(d => d.TaskStatus.Name == "Completed");
            var completedTaskPercentage = Convert.ToDecimal(completedTasksCount > 0 ? completedTasksCount / tasksCount * 1

            return new EmployeeProjectModel
            {
                ProjectId = eWork.Project.Id,
                ProjectName = eWork.Project.Name,
                Position = eWork.EmployeePosition.Name,
                ProjectCreationDate = eWork.Project.CreationDate,
                ProjectClosingDate = eWork.Project.ClosingDate,
                CompletedTasksPercentage = completedTaskPercentage
            };
        }

        1 reference
        public async Task AddAsync(EmployeeModel model)
        {
            await _uow.EmployeeRepository.AddAsync(_mapper.Map<Employee>(model));
            await _uow.SaveAsync();
        }

        2 references
        public IEnumerable<EmployeeModel> GetAll()
        {
            return _mapper.Map<IEnumerable<EmployeeModel>>(_uow.EmployeeRepository.FindAllWithDetails());
        }

        2 references
        public IEnumerable<EmployeeProjectModel> GetAllEmployeeProjects(string id)
        {
            var works = _uow.WorkRepository.FindAllWithDetails().Where(ew => ew.EmployeeId == id).ToList();
            return !works.Any() ? null : works.Select(CreateEmployeeProjectModel);
```

Рис.36 – Приклад класу-сервісу

У методах сервісу також знаходиться додаткова логіка, така як валідація, обрахування певних необхідних значення, виклик помилок у разі передачі помилкових даних.

3.4 Реалізація Presentation Logic Layer

Основна логіка програми вже реалізована, залишилось реалізувати обробку клієнтського запиту до WebApi. За це якраз і відповідає шар представлення(PLL). В новостворений проєкт Web Api було добавлене посилання на бібліотеку класів шару BLL. Також було впроваджено усі необхідні залежності через спеціальний метод ConfigureServices в класі Startup.cs.

```
31      public void ConfigureServices(IServiceCollection services)
32      {
33          /*
34          services.AddSpaStaticFiles(configuration =>
35          {
36              configuration.RootPath = "ClientApp/dist";
37          });
38          */
39          var jwtSettings = Configuration.GetSection("JwtSettings");
40
41          services.AddAutoMapper(cfg => cfg.AddProfile(new AutomapperProfile()));
42          services.AddDbContext<TaskTrackingContext>(c =>
43              c.UseSqlServer(Configuration.GetConnectionString("TaskTrackingDb")));
44
45          services.AddIdentity<Account, IdentityRole>(opt =>
46          {
47              opt.Password.RequiredLength = 8;
48              opt.Password.RequireNonAlphanumeric = false;
49
50              opt.User.RequireUniqueEmail = true;
51          }).AddEntityFrameworkStores<TaskTrackingContext>();
52          services.AddAuthentication(opt =>
53          {
54              opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
55              opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
56          }).AddJwtBearer(options =>
57          {
58              options.TokenValidationParameters = new TokenValidationParameters()
59              {
60                  ValidateIssuer = true,
61                  ValidateAudience = true,
62                  ValidateLifetime = true,
63                  ValidateIssuerSigningKey = true,
64
65                  ValidIssuer = jwtSettings.GetSection("ValidIssuer").Value,
66                  ValidAudience = jwtSettings.GetSection("ValidAudience").Value,
67                  IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(j
68              };
69          });
70
71          services.AddTransient<JwtHandler>();
72
73          services.AddTransient<IAccountManagers, AccountManagers>();
74
75          services.AddTransient<IUnitOfWork, UnitOfWork>();
76
77          services.AddTransient<IAuthorizeService, AuthorizeService>();
78          services.AddTransient<IRoleService, RoleService>();
79          services.AddTransient<IEmployeeService, EmployeeService>();
80          services.AddTransient<IProjectService, ProjectService>();
81          services.AddTransient<IPositionService, PositionService>();
```

Рис.37 – Метод ConfigureServices

Обробка запитів в ASP.NET Core побудована пронципу конвейера. Данні запиту попадають в перший компонент в конвейреї, де після обробки він передає дані HTTP-запити в наступний і так далі. Ці компаненти звуться middleware. Підключення middleware відбувається через використання методу Configure з класу Startup.cs.

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
0 references
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseSwagger();

    app.UseSwaggerUI(s =>
    {
        s.SwaggerEndpoint("/swagger/v1/swagger.json", "WebApi");
    });

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthentication();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Рис.38 – Метод Configure

В методі добавлені компоненти middleware для обробки запиту, а саме:

- компонент обробки помилок - app.UseDeveloperExceptionPage()
- компонент застосування інструменту Swagger – app.UseSwagger();
- компонент HTTPS– app.UseHttpsRedirection();
- компонент маршрутизації – app.useRouting();

- компонент аутентифікації – app.UseAuthentication();
- компонент авторизації – app.UseAuthorization();
- компонент відправляє відповіді при звертані по заданому маршруту app.UseEndpoints

Метод Configure виконується один раз при створені об’єкту класу Startup.cs і існує протягом всього життєвого циклу додатку.

Запит клієнта відбувається по принципу звернення до контроллера в якого є так звані ендпоінти. Це методи через які безпосереднього відбувається звернення до додатку. Є декілька видів http запитів, які за правилами REST мають:

- GET – запит на отримання даних
- POST – запит на внесення даних, а також для авторизації
- PUT – запит на зміну існуючих даних
- PATCH – запит на часткову зміну існуючих даних
- DELETE – запит на видалення даних

На основі цих запитів, програма генерує відповідь та повертає клієнту. В контролерах також відбувається валідація вхідних об’єктів та у разі неправильних даних повертає відповідний результат.

Також був створений ExceptionFilter, який відповідає за обробку виникнувших помилок(Exceptions) та повертає визначений результат в залежності від типу помилки.

```
namespace TaskTrackingSystem.WebApi.Controllers
{
    [Authorize]
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class EmployeesController : Controller
    {
        private readonly IEmployeeService _employeeService;

        0 references
        public EmployeesController(IEmployeeService employeeService)
        {
            _employeeService = employeeService;
        }

        [HttpGet]
        0 references
        public ActionResult<IEnumerable<EmployeeModel>> GetAllEmployee()
        {
            var result = _employeeService.GetAll();
            if (result == null)
                return NotFound("There are no employees");
            return Ok(result);
        }

        [HttpGet("{id}", Name = "GetEmployee")]
        0 references
        public async Task<ActionResult<EmployeeModel>> GetById(string id)
        {

            var result = await _employeeService.GetByIdAsync(id);
            if (result == null)
                return NotFound("There is no such employee");
            return Ok(result);
        }

        [HttpGet("{id}/projects")]
        0 references
        public ActionResult<IEnumerable<EmployeeProjectModel>> GetAllEmployeeProjects(string id)
        {
            var result = _employeeService.GetAllEmployeeProjects(id);
            if (result == null)
                return NotFound("Employee doesn't have any projects");
            return Ok(result);
        }

        [HttpGet("{id}/tasks")]
        0 references
        public ActionResult<IEnumerable<EmployeeTaskModel>> GetAllEmployeeTasks(string id)
        {
            var result = _employeeService.GetAllEmployeeTasks(id);
            if (result == null)
                return NotFound("Employee doesn't have any tasks");
            return Ok(result);
```

Рис.39 – Приклад контролеру

```
namespace TaskTrackingSystem.WebApi.Filters
{
    1 reference
    public class ErrorHandlingFilter : ExceptionFilterAttribute
    {
        0 references
        public override void OnException(ExceptionContext context)
        {
            HandleExceptionAsync(context);
            context.ExceptionHandled = true;
        }

        1 reference
        private static void HandleExceptionAsync(ExceptionContext context)
        {
            var exception = context.Exception;

            switch (exception)
            {
                case BadRequestException _:
                    SetExceptionResult(context, exception, HttpStatusCode.BadRequest);
                    break;
                case NotFoundException _:
                    SetExceptionResult(context, exception, HttpStatusCode.NotFound);
                    break;
                case ConflictException _:
                    SetExceptionResult(context, exception, HttpStatusCode.Conflict);
                    break;
                default:
                    SetExceptionResult(context, exception, HttpStatusCode.BadRequest);
                    break;
            }
        }

        4 references
        private static void SetExceptionResult(
            ExceptionContext context,
            Exception exception,
            HttpStatusCode code)
        {
            context.Result = new JsonResult(new
            {
                StatusCode = (int) code,
                Message = exception.Message
            });
        }
    }
}
```

Рис.40 – ExceptionFilter

3.5 Реалізація аутентифікації і авторизації

Для процесу аутентифікації[27] створений спеціальний контролер з двома ендпоінтами, один для реєстрації, інший для логіну. З використанням ASP.NET Core Identity та JWT Bearer Token[29].

В метод ConfigureServices в класі Startup добавляється налаштування для аутентифікація та JWT токена[28], де для JWT токена беруться параметри з файлу appsettings.json

```
services.AddAuthentication(opt =>
{
    opt.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    opt.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
    options.TokenValidationParameters = new TokenValidationParameters()
    {
        ValidateIssuer = true,
        ValidateAudience = true,
        ValidateLifetime = true,
        ValidateIssuerSigningKey = true,

        ValidIssuer = jwtSettings.GetSection("ValidIssuer").Value,
        ValidAudience = jwtSettings.GetSection("ValidAudience").Value,
        IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtSettings.GetSection("SecurityKey").Value))
    };
});
```

Рис.36 – Налаштування аутентифікації та JWT токена

```
"JwtSettings": {
  "ValidIssuer": "https://localhost:44339",
  "ValidAudience": "https://localhost:44339",
  "SecurityKey": "UltimateSecretKey2021",
  "ExpiryInDays":  7
},
```

Рис.37 – Параметри для JWT токена

Основна логіка авторизації знаходиться в класі-сервісі AuthorizeService а також в допоміжному класі JwtHandle.

```
namespace TaskTrackingSystem.Business.Services
{
    2 references
    public class AuthorizeService : IAuthorizeService
    {
        private readonly IUnitOfWork _uow;

        private readonly IMapper _mapper;

        private readonly IAccountManagers _identityManagers;

        private readonly JwtHandler _jwtHandler;

        0 references
        public AuthorizeService(JwtHandler jwtHandler, IUnitOfWork uow, IMapper mapper, IAccountManagers identityManagers)
        {
            _uow = uow;
            _mapper = mapper;
            _identityManagers = identityManagers;
            _jwtHandler = jwtHandler;
        }

        2 references
        public async Task<RegistrationResponseModel> RegisterAsync(RegisterModel model)
        {
            Account account = new Account
            {
                Email = model.Email,
                UserName = model.Email
            };
            var result = await _identityManagers.UserManager.CreateAsync(
                account, model.Password);
            if (result.Succeeded)
            {
                await _uow.EmployeeRepository.AddAsync(new Employee
                {
                    Account = account,
                    EmployeeProfile = new EmployeeProfile()
                });
                await _identityManagers.UserManager.AddToRoleAsync(account, "Employee");

                await _uow.SaveAsync();
                await _identityManagers.SignInManager.SignInAsync(account, false);
            }

            var response = new RegistrationResponseModel
                {IsSuccessful = result.Succeeded, Errors = result.Errors.Select(e => e.Description)};
            return response;
        }

        2 references
        public async Task<LoginResponseModel> LoginAsync(LoginModel model)
        {
            var account = await _identityManagers.UserManager.FindByNameAsync(model.Email);
            if (account == null || !await _identityManagers.UserManager.CheckPasswordAsync(account, model.Password))
                return new LoginResponseModel { ErrorMessage = "Invalid login or password" };

            var signingCredentials = _jwtHandler.GetSigningCredentials();
            var claims = await _jwtHandler.GetClaims(account);
            var tokenOptions = _jwtHandler.GenerateTokenOptions(signingCredentials, claims);
            var token = new JwtSecurityTokenHandler().WriteToken(tokenOptions);

            return new LoginResponseModel {IsSuccessful = true, Token = token};
        }
    }
```

Рис.38 – Клас AuthorizeService

При реєстрації викликається метод RegisterAsync класу AuthorizeService в який передається об’єкт класу RegisterModel, та потім, за допомогою класу інструменту UserManager реєструється користувач в системі, тобто записуються всі необхідні данні та шифрується пароль. В разі успіху, створюється для цього акаунту сутність Employee та добавляється відповідна роль.

При авторизації ми користуємось допоміжним класом JwtHandler.

```
namespace TaskTrackingSystem.Business
{
    4 references
    public class JwtHandler
    {
        private readonly IConfiguration _configuration;
        private readonly IConfigurationSection _jwtSettings;
        private readonly IAccountManagers _accountManagers;

        0 references
        public JwtHandler(IConfiguration configuration, IAccountManagers accountManagers)
        {
            _accountManagers = accountManagers;
            _configuration = configuration;
            _jwtSettings = _configuration.GetSection("JwtSettings");
        }

        1 reference
        public SigningCredentials GetSigningCredentials()
        {
            var key = Encoding.UTF8.GetBytes(_jwtSettings.GetSection("SecurityKey").Value);
            var secret = new SymmetricSecurityKey(key);

            return new SigningCredentials(secret, SecurityAlgorithms.HmacSha256);
        }

        1 reference
        public async Task<List<Claim>> GetClaims(Account account)
        {
            var claims = new List<Claim>
            {
                new Claim(ClaimTypes.Name, account.Email)
            };

            var roles = await _accountManagers.UserManager.GetRolesAsync(account);
            claims.AddRange(roles.Select(role => new Claim(ClaimTypes.Role, role)));

            return claims;
        }

        1 reference
        public JwtSecurityToken GenerateTokenOptions(SigningCredentials signingCredentials, List<Claim> claims)
        {
            var tokenOptions = new JwtSecurityToken(
                issuer: _jwtSettings.GetSection("ValidIssuer").Value,
                audience: _jwtSettings.GetSection("ValidAudience").Value,
                claims: claims,
                expires: DateTime.Now.AddDays(Convert.ToDouble(_jwtSettings.GetSection("ExpiryInDays").Value)),
                signingCredentials: signingCredentials);

            return tokenOptions;
        }
    }
}
```

Рис.39 – Клас JwtHandler

В метод LoginAsync приходить LoginModel, за даними якої перевіряється наявність такого користувача та порівняння паролів. У разі успіху, викликаються методи для створення токену. Згенерований токен повертаємо на клієнтську частину. Авторизація по ролям реалізована за допомогою middleware app.UseAuthorization, де використовуючи ключове слово [Authorize] та задуючи ролі ми вказуємо, для яких ролей який ендпоінт буде доступний.

```
[Authorize]
[Route("api/[controller]")]
[ApiController]
1 reference
public class TasksController : ControllerBase
{
    private readonly ITaskService _taskService;

    0 references
    public TasksController(ITaskService taskService)
    {
        _taskService = taskService;
    }

    [HttpGet("{id}", Name = "GetTask")]
    0 references
    public async Task<ActionResult<TaskModel>> GetTaskById(int id)
    {
        var result = await _taskService.GetByIdAsync(id);
        if (result == null)
            return NotFound("There is no Task with that Id");
        return Ok(result);
    }

    [Authorize(Roles = "Admin, Manager")]
    [HttpPost]
    0 references
    public async Task<ActionResult> AddTask(TaskCreateModel model)
    {
        if (!ModelState.IsValid)
            return BadRequest(ModelState);

        await _taskService.AddAsync(model);
        return CreatedAtRoute("GetTask", new {model.Id}, model);
    }
```

Рис.40 – Приклад використання ключового слова Authorize

3.6 Використання Swagger

Swagger[30] це інструмент створений для специфікації REST API. Його суть полягає в тому, що він дає можливість інтерактивно переглядати специфікацію, відправляти запити та бачити отриманий результат.

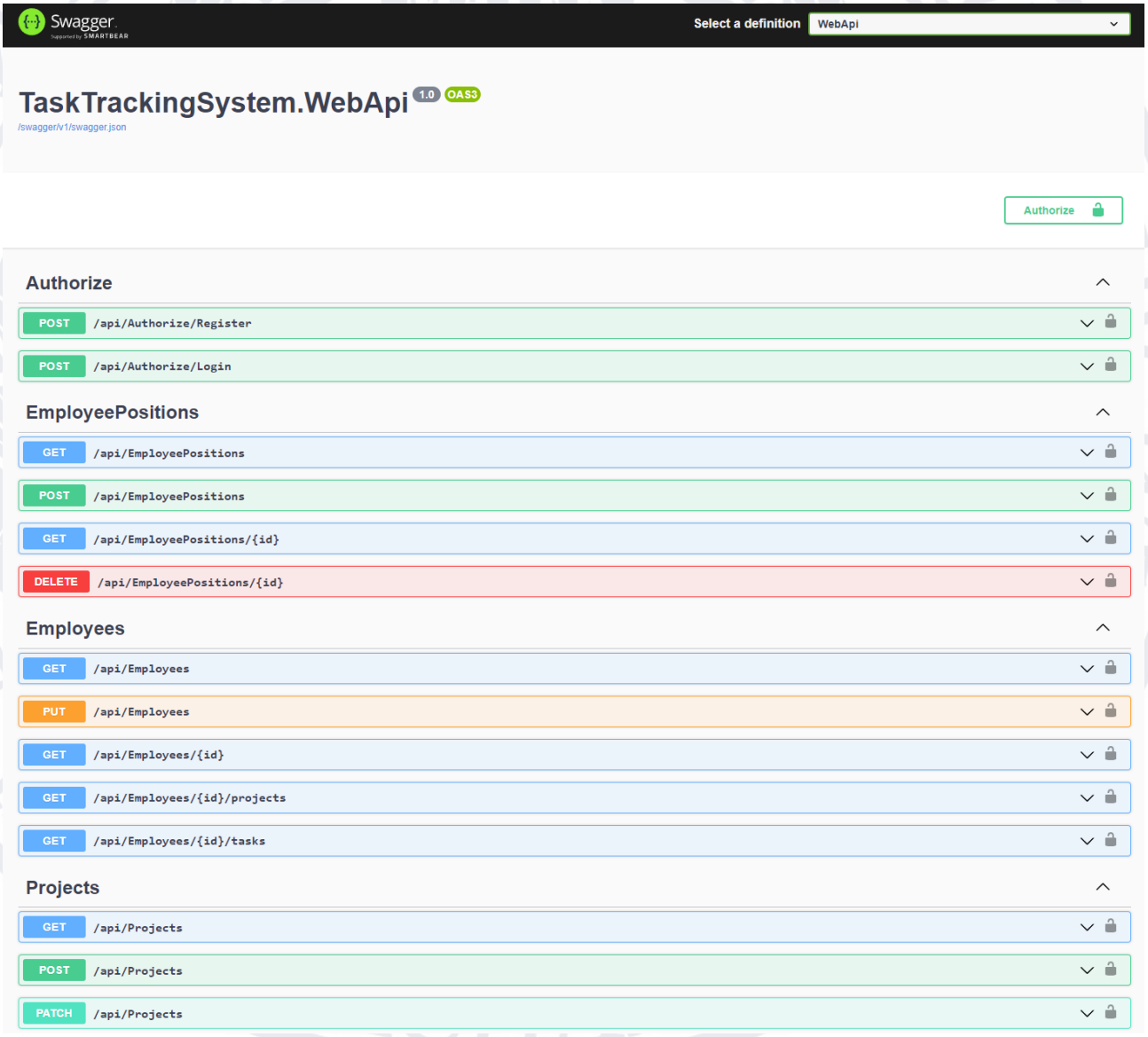

Рис.41 – Swagger з переліком контролерів та http запитів

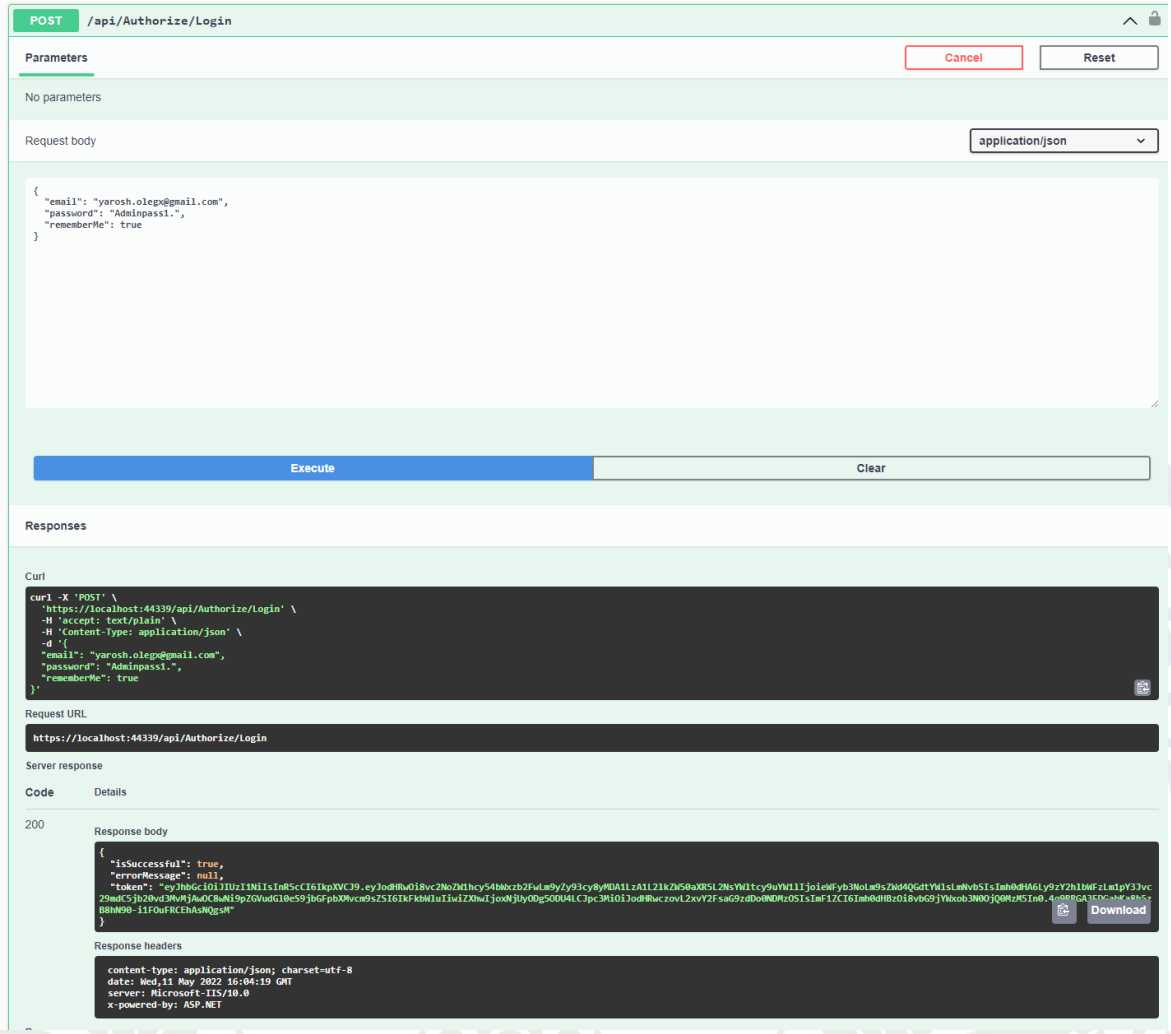

Рис.42 – Виклик запиту та отримання результату

3.7 Висновок до розділу 3

Було оглянуто та проаналізовано архітектурні принципи REST та Багатошарової архітектури. Детально описано реалізацію кожного шару трьох-шарової архітектури. Опис реалізації аутентифікації та авторизації та принципу взаємодії з додатком.

ВИСНОВКИ

У рамках даної роботи було розглянуто актуальність систем управління проєктами та їх поширення в сьогодення.

Були проаналізовані вибрані технології та інструменти розробки Web Api виділенні їх переваги та можливості. Розглянуто можливості вибраного серверу баз даних.

Був створений та протестований додаток відподно до принципів REST. Реалізовано багатошарову архітектуру в додатку. Набуті навички роботи з сучасними інструментами розробки Web Api та впровадження новітніх архітектурних підходів.

СПИСОК ЛІТЕРАТУРИ

[1] – The Future of Remote Working the good, the bad and the ugly [Електронний ресурс]. Режим доступу до ресурсу: https://luminalearning.com/the-future-of-remote-working-the-good-the-bad-and-the-ugly/

[2] – Organizations are embracing shifts to remote work, presenting opportunities for tech companies [Електронний ресурс]. Режим доступу до ресурсу: https://www.businessinsider.com/work-from-home-presents-opportunity-for-tech-providers-2020-5

[3] – Asana [Електронний ресурс]. Режим доступу до ресурсу: https://asana.com/

[4] – Gannt Chart [Електронний ресурс]. Режим доступу до ресурсу: https://www.figma.com/community/file/992321381646665135

[5] – Active Collab [Електронний ресурс]. Режим доступу до ресурсу: https://activecollab.com/

[6] – Project Management System: Definition & Features [Електронний ресурс]. Режим доступу до ресурсу: https://www.proofhub.com/articles/project-management-system

[7] – Wrike [Електронний ресурс]. Режим доступу до ресурсу: https://www.wrike.com/vm/

[8] – MVC Pattern [Електронний ресурс]. Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Glossary/MVC

[9] – Announcing .NET 6 — The Fastest .NET Yet [Електронний ресурс]. Режим доступу до ресурсу: https://devblogs.microsoft.com/dotnet/announcing-net-6/

[10] – ASP.NET Core GitHub Repository [Електронний ресурс]. Режим доступу до ресурсу: https://github.com/dotnet/aspnetcore/

[11] – Common web application architectures [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/ru-ru/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

[12] – What is .NET platform overview [Електронний ресурс]. Режим доступу до ресурсу: https://auth0.com/blog/what-is-dotnet-platform-overview/

[13] – A tour of the C# language [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/

[14] – Microsoft Visual Studio Capabilities [Електронний ресурс]. Режим доступу до ресурсу: https://softwarekeep.com/help-center/what-is-microsoft-visual-studio-where-can-i-download-it

[15] – Overview of ASP.NET Core [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/en-us/aspnet/core/introduction-to-aspnet-core?view=aspnetcore-6.0

[16] – Introducing to ASP.NET Identity [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/en-us/aspnet/identity/overview/getting-started/introduction-to-aspnet-identity

[17] – Overview of Visual Studio [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022

[18] – First look at the Visual Studio Debugger [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/en-us/visualstudio/debugger/debugger-feature-tour

[19] – Microsoft SQL Server [Електронний ресурс]. Режим доступу до ресурсу: https://ru.wikipedia.org/wiki/Microsoft_SQL_Server

[20] – What is SQL Server? [Електронний ресурс]. Режим доступу до ресурсу: https://www.techtarget.com/searchdatamanagement/definition/SQL-Server

[21] – Understanding Multilayered Architecture in .NET [Електронний ресурс]. Режим доступу до ресурсу: https://www.c-sharpcorner.com/UploadFile/1492b1/understanding-multilayered-architecture-in-net/

[22] – What is REST [Електронний ресурс]. Режим доступу до ресурсу: https://restfulapi.net/

[23] – Overview of Entity Framework Core [Електронний ресурс]. Режим доступу до ресурсу: https://docs.microsoft.com/en-us/ef/core/

[24] – What is Code-First? [Електронний ресурс]. Режим доступу до ресурсу: shorturl.at/kqBL4

[24] – Unit of Work in Repository Pattern [Електронний ресурс]. Режим доступу до ресурсу: https://www.c-sharpcorner.com/UploadFile/b1df45/unit-of-work-in-repository-pattern/

[25] – Repository and Unit of Work Pattern [Електронний ресурс]. Режим доступу до ресурсу: https://www.programmingwithwolfgang.com/repository-and-unit-of-work-pattern/

[26] – What is AutoMapper and using it in ASP.NET Core [Електронний ресурс]. Режим доступу до ресурсу: https://www.pragimtech.com/blog/blazor/using-automapper-in-asp.net-core/

[27] – Difference between Authentication and Authorization [Електронний ресурс]. Режим доступу до ресурсу: https://www.sailpoint.com/identity-library/difference-between-authentication-and-authorization/

[28] – What is JSON Web Token? [Електронний ресурс]. Режим доступу до ресурсу: https://jwt.io/introduction

[29] – JWT Authentication And Authorization In .NET 6.0 With Identity Framework [Електронний ресурс]. Режим доступу до ресурсу: https://www.c-

sharpcorner.com/article/jwt-authentication-and-authorization-in-net-6-0-with-identity-framework/

[30] – Swagger: API Documentation & Design Tools [Електронний ресурс]. Режим доступу до ресурсу: https://swagger.io/

Декларація щодо унікальності текстів роботи
та невикористання матеріалів інших авторів без посилань

Ярош Олег Леонідович
Прізвище, ім’я, по батькові
Факультет інформаційних і прикладних технологій
Факультет
122 Комп’ютерні науки
Шифр і назва спеціальності
Сучасні інформаційні технології та програмування
Освітня програма

ДЕКЛАРАЦІЯ

Усвідомлюючи свою відповідальність за надання неправдивої інформації, стверджую, що подана кваліфікаційна (бакалаврська) робота на тему: «Розробка API для системи управління проєктами» є написаною мною особисто.

Одночасно заявляю, що ця робота:

- не передавалась іншим особам і подається до захисту вперше;
- не порушує авторських та суміжних прав, закріплених статтями 21-25 Закону України «Про авторське право та суміжні права»;
- не отримувались іншими особами, а також дані та інформація не отримувались у недозволений спосіб.

Я усвідомлюю,. що у разі порушення цього порядку моя кваліфікаційна (бакалаврська) робота буде відхилена без права її захисту, або під час захисту за неї буде поставлена оцінка «незадовільно».

_____ _____

дата підпис