МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

Ярош Олег Леонідович

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій,
канд. техн. наук, доцент

_____Оксана ЗЕЛІНЬСКА
«__»_____20__ р.

КРОСПЛАТФОРМОВИЙ API ДЛЯ СИСТЕМИ РОЗПОДІЛУ ЗАВДАНЬ ТА АНАЛІЗУ ПРОДУКТИВНОСТІ ВИКОНАВЦІВ

Спеціальність 122 «Комп’ютерні науки»

Кваліфікаційна (магістерська) робота
(відповідно до стандарту спеціальності та ОП)

Науковий керівник:
Р.М. Бабаков, професор кафедри
інформаційних технологій,
докт. техн. наук, доцент

_____
(підпис)

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

Вінниця – 2024

АНОТАЦІЯ

Ярош О.Л. Кросплатформовий API для системи розподілу завдань та аналізу продуктивності виконавців

Спеціальність 122 «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2024.

Головною метою виконання даної кваліфікаційної роботи є побудова архітектури та реалізація API для системи розподілу завдань та аналізу продуктивності виконавців. Це зумовлено зростанням попиту на такі програмні продукти.

У вступі акцентовано на актуальності розробки систем управління розподілу завдань та аналізу продуктивності виконавців.

Перший розділ присвячено аналізу предметної області та розгляду існуючих аналогів з визначенням їхніх переваг і недоліків.

Другий розділ детально розглядає ключові технології, використані при створенні додатку, а також інструменти та сервер баз даних, і висвітлює їхні переваги.

Третій розділ аналізує основні архітектурні рішення, процес побудови бази даних, втілення архітектурних рішень. Також розкривається принцип взаємодії з API та його функціональні можливості.

Ключові терміни: API, система розподілу задач, аналіз продуктивності виконавців, веб-додаток, .NET, ASP.NET, чиста архітектура, REST API, CQRS.

ANNOTATION

Yarosh O.L. Cross-platform API for the system of task distribution and performance analysis.

122 "Computer Science" Major. Vasyl' Stus Donetsk National University, Vinnytsia, 2024.

The main goal of this qualification work is to build an architecture and implement an API for a system for distributing tasks and analyzing the performance of executors. This is due to the growing demand for such software products.

The introduction emphasizes the relevance of developing management systems for task distribution and performance analysis.

The first section is devoted to analyzing the subject area and reviewing existing analogs with the identification of their advantages and disadvantages.

The second section discusses in detail the key technologies used to create the application, as well as the tools and database server, and highlights their advantages.

The third section analyzes the main architectural solutions, the process of building a database, and the implementation of architectural solutions. The principle of interaction with the API and its functionality is also revealed.

Keywords: API, task distribution system, executor performance analysis, web application, .NET, ASP.NET, clean architecture, REST API, CQRS.

ЗМІСТ

ВСТУП

У сучасному світі, де технології швидко розвиваються, ефективне управління завданнями та аналіз продуктивності виконавців є важливими складовими успіху в багатьох сферах. Розробка системи, яка дозволяє ефективно розподіляти завдання та аналізувати продуктивність виконавців, може принести значний внесок у покращення процесів управління та досягнення кращих результатів.

У магістерській роботі досліджується тема "Кросплатформовий API для системи розподілу завдань та аналізу продуктивності виконавців". Метою даної роботи є розробка універсального програмного інтерфейсу (API), який дозволить інтегрувати систему розподілу завдань та аналізу продуктивності виконавців з різними платформами та середовищами.

Одним з основних викликів управління завданнями та аналізу продуктивності виконавців є різноманітність платформ, на яких працюють виконавці. Компанії використовують різні інструменти та середовища розробки, такі як веб-додатки, мобільні додатки, настільні програми тощо. Ця різноманітність ускладнює процес розподілу завдань та збіру даних про продуктивність, оскільки необхідно підтримувати сумісність з різними платформами.

Розробка кросплатформового API дозволить уникнути проблем, пов'язаних з різноманітністю платформ. Використовуючи таке API, система розподілу завдань та аналізу продуктивності виконавців буде здатна спілкуватися з будь-якими платформами незалежно від їх типу та характеристик. Це забезпечить зручність та ефективність управління завданнями, а також забезпечить збір та аналіз даних про продуктивність виконавців незалежно від їхнього робочого середовища.

Мета магістерської роботи полягає в розробці кросплатформового API, яке буде забезпечувати стандартизований спосіб комунікації між системою розподілу завдань та аналізу продуктивності виконавців та різними платформами. Результатом дослідження буде програмний інтерфейс, який

забезпечить зручність і сумісність управління завданнями та аналізу продуктивності виконавців незалежно від платформ, на яких вони працюють.

Дана магістерська робота важлива, оскільки вона сприятиме покращенню процесів управління завданнями та аналізу продуктивності виконавців, забезпечивши їхню більшу ефективність та сумісність з різними платформами. Розроблений кросплатформовий API може бути використаний в різних галузях, де важлива організація та контроль робочих процесів.

РОЗДІЛ 1
ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Аналіз предметної області

Кросплатформовий API для системи розподілу завдань та аналізу продуктивності може бути корисним для розробників або команд, які прагнуть створити власні додатки або інтегрувати функціональність розподілу завдань та аналізу продуктивності виконавців у свої проекти. Він надає зручний спосіб взаємодії з системою без необхідності використання графічного інтерфейсу користувача.

У зв'язку з масовими переміщеннями громадян через війну в Україні такий застосунок стає ще більш актуальним. Війна та переміщення можуть суттєво вплинути на організацію та ефективність роботи команд, особливо якщо члени команди знаходяться в різних географічних регіонах або працюють на віддалених робочих місцях[1]. Також не варто забувати про наслідки пандемії коронавірусу, які теж досить суттєво вплинули на теперішне положення[2, 3].

API дозволяє забезпечити комунікацію та співпрацю між виконавцями, незалежно від їх місцезнаходження. Це дозволить ефективно розподіляти завдання між членами команди, забезпечити контроль та відстеження їх виконання, незалежно від того, де знаходяться виконавці.

Особливо у випадку переміщених громадян, які можуть працювати у нових умовах та середовищах, такий API допоможе забезпечити стабільність та організацію роботи. Він дасть можливість керувати завданнями, встановлювати пріоритети, розподіляти ресурси та контролювати виконання завдань у режимі реального часу.

Крім того, аналіз продуктивності виконавців стає важливим в умовах переміщень і змін у робочих умовах. Він дозволить оцінити ефективність та результативність виконавців незалежно від їх географічного положення. Це

допоможе виявити проблемні моменти, покращити процеси та забезпечити оптимальне використання ресурсів.

Основні компоненти, які можуть бути включені в кросплатформовий API для системи розподілу завдань та аналізу продуктивності виконавців, можуть включати:

1. Модуль розподілу завдань: Цей модуль відповідає за прийом та розподіл завдань між виконавцями. Він може включати функції, такі як створення завдань, призначення виконавців, контроль стану виконання завдань та сповіщення про зміни.

2. Модуль аналізу продуктивності: Цей модуль збирає та аналізує дані про продуктивність виконавців. Він може включати функціональність, таку як збір статистики, вимірювання часу виконання завдань, оцінка ефективності та генерація звітів.

3. Модуль автентифікації та авторизації: Цей модуль забезпечує механізми перевірки прав доступу та ідентифікації користувачів API. Він дозволяє обмежувати доступ до функцій API лише авторизованим користувачам та контролювати їхні дії.

4. Модуль комунікації: Цей модуль відповідає за комунікацію між додатками та системою розподілу завдань. Він може підтримувати різні протоколи передачі даних, такі як HTTP, WebSocket або інші, залежно від вимог системи.

5. Документація та приклади використання: Важливим аспектом кросплатформового API є наявність документації, яка описує доступні функції, параметри та способи використання API. Також можуть бути надані приклади коду для допомоги розробникам у розумінні та використанні API.

Переваги використання кросплатформового API для системи розподілу завдань та аналізу продуктивності виконавців включають:

- Універсальність: Кросплатформовий API може бути використаний на різних платформах та мовах програмування, що робить його більш доступним для розробників з різних середовищ.

- Легкість інтеграції: За допомогою API розробники можуть швидко інтегрувати функціональність розподілу завдань та аналізу продуктивності виконавців у свої додатки без необхідності розробки всієї системи з нуля.

- Розширюваність: Кросплатформовий API може бути розширений та модифікований для врахування специфічних потреб користувачів та додатків.

- Централізоване керування: Використання API дозволяє централізовано керувати розподілом завдань та аналізом продуктивності виконавців, що полегшує адміністрування та моніторинг цих процесів.

Однак, при розробці кросплатформового API необхідно враховувати такі фактори:

- Безпека: API повинен мати механізми захисту від несанкціонованого доступу та зловживання.

- Масштабованість: API повинен бути спроектований таким чином, щоб витримувати великі навантаження та забезпечувати ефективну роботу системи.

- Сумісність: API повинен бути сумісним з різними версіями програмного забезпечення та здатним легко адаптуватися до змін в системі розподілу завдань та аналізу продуктивності виконавців.

1.2 Постановка задачі

Наша задача полягає в створенні кросплатформового API для системи розподілу завдань та аналізу продуктивності виконавців. Цей API буде служити як міст між фронтенд-клієнтами (наприклад, веб-додатками, мобільними додатками) і серверною частиною системи.

Метою створення цього застосунку є спрощення, а також підвищення ефективності та організації робочих процесів у команді чи організації.

Основні функціональні вимоги до API:

- Автентифікація та авторизація: API повинен забезпечувати механізми для ідентифікації та аутентифікації користувачів, а також контроль доступу до різних ресурсів на основі ролей та прав доступу.
- Управління завданнями: API повинен надавати можливість створювати, оновлювати, видаляти та отримувати інформацію про завдання. Кожне завдання повинно мати опис, статус, термін виконання та призначеного виконавця.
- Управління виконавцями: API повинен дозволяти створювати, оновлювати, видаляти та отримувати інформацію про виконавців. Кожен виконавець повинен мати профіль з даними про його навички, досвід, рейтинг та доступність.
- Статистика та аналітика: API повинен надавати можливість отримувати статистичні дані та аналізувати продуктивність виконавців, такі як кількість завдань, успішно виконаних завдань, середній час виконання тощо.
- Комунікація та сповіщення: API повинен підтримувати можливість спілкування між користувачами системи, надсилання сповіщень про оновлення стану завдань або повідомлення про нові завдання.

Технічні вимоги до API:

- Кросплатформовість: API повинен бути доступний на різних платформах, включаючи веб, мобільні пристрої та настільні додатки.

- RESTful архітектура: API повинен бути побудований на основі принципів REST (Representational State Transfer), що дозволить зручне взаємодіяти з ресурсами системи через стандартні HTTP методи (GET, POST, PUT, DELETE).
- JSON формат обміну даними: API повинен використовувати JSON (JavaScript Object Notation) як основний формат обміну даними між клієнтом та сервером.
- Захист даних: API повинен забезпечувати захист передачі даних за допомогою шифрування (наприклад, HTTPS).
- Документація: API повинен мати детальну технічну документацію, яка пояснює всі доступні ендпоінти, параметри, відповіді та приклади використання.

1.3 Огляд існуючих аналогій

Існує багато систем розподілу завдань та аналізу продуктивності виконавців[4], що можуть бути використані в різних сферах діяльності. Ось кілька популярних систем, які можна розглянути:

1. Trello

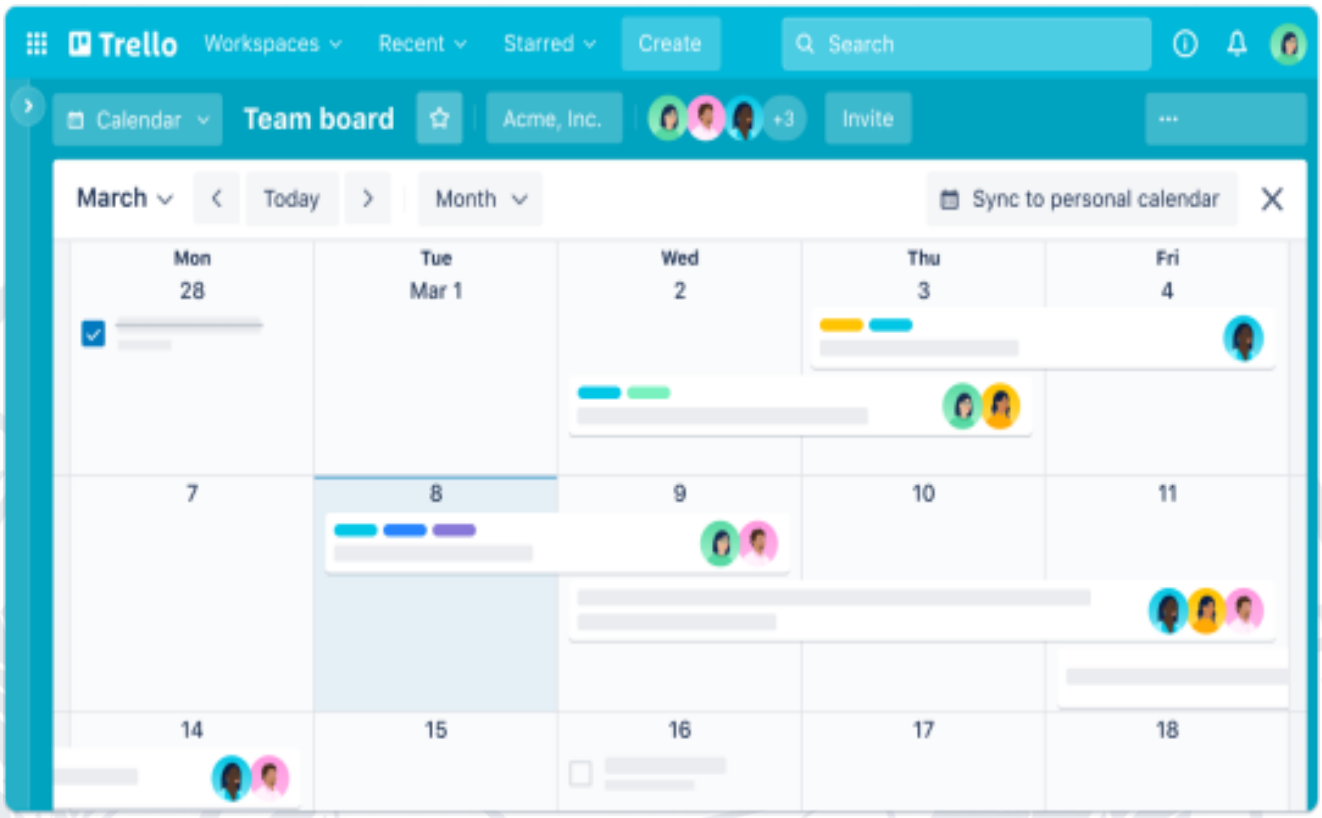


Рис. 1 – Система “Trello”

Trello[5] - це онлайн-дошка, яка дозволяє створювати завдання, організовувати їх у списки та визначати відповідальних осіб. Ви можете додавати описи, коментарі, встановлювати терміни виконання завдань, а також використовувати мітки для категоризації завдань. Trello також надає зручні засоби аналізу продуктивності, включаючи графіки, діаграми та звіти.

Переваги:

- Простий та інтуїтивно зрозумілий інтерфейс.
- Гнучкість у використанні завдань, списків та дошок.
- Зручна система міток та фільтрів.
- Можливість додавання коментарів та вкладення файлів до завдань.
- Доступність додатків для різних платформ.

Недоліки:

- Обмежені можливості аналізу продуктивності порівняно з деякими іншими системами.
- Відсутність деяких розширених функцій у безкоштовній версії.
- Можливість перевантаження дошок при великій кількості завдань.

2. Asana

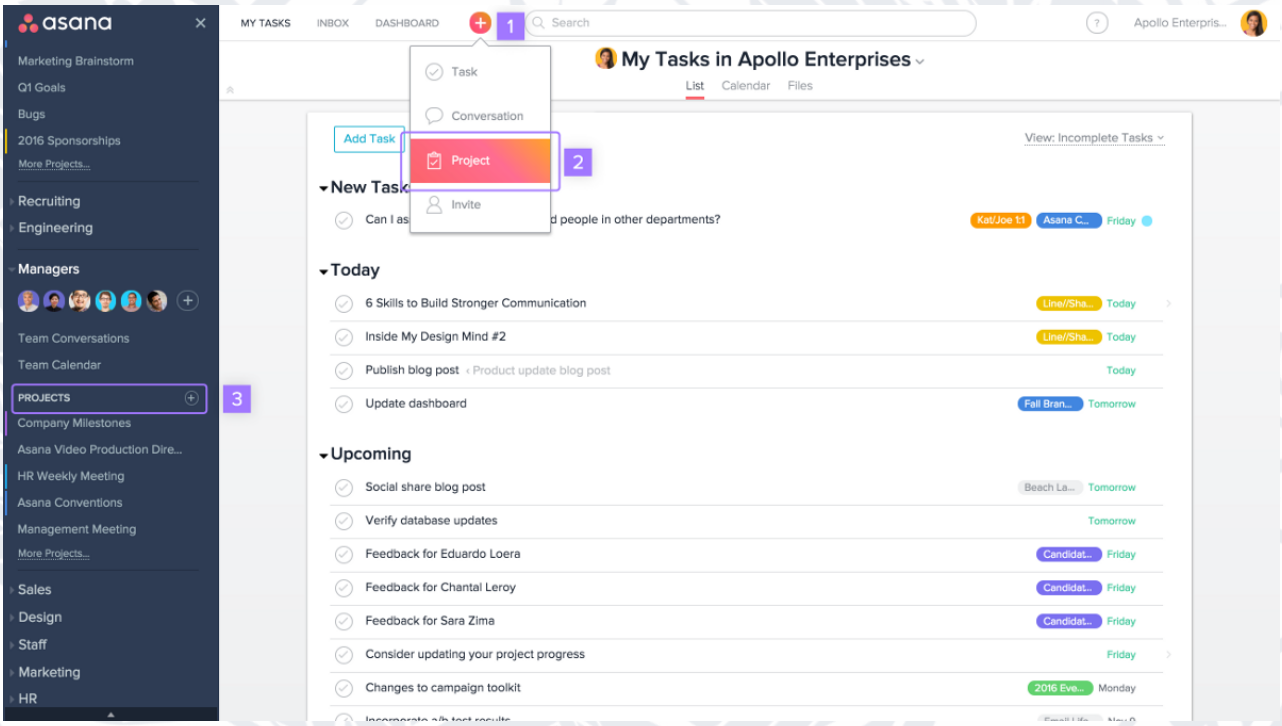


Рис. 2 – Система “Asana” [6]

Asana - це інструмент для керування проектами, який дозволяє створювати завдання, делегувати їх, встановлювати терміни та відстежувати прогрес виконання. Ви можете організовувати завдання за проектами, створювати команди та надавати доступ до відповідних завдань виконавцям. Asana також надає різноманітні засоби аналізу продуктивності, такі як звіти про прогрес, графіки завершеності та інші.

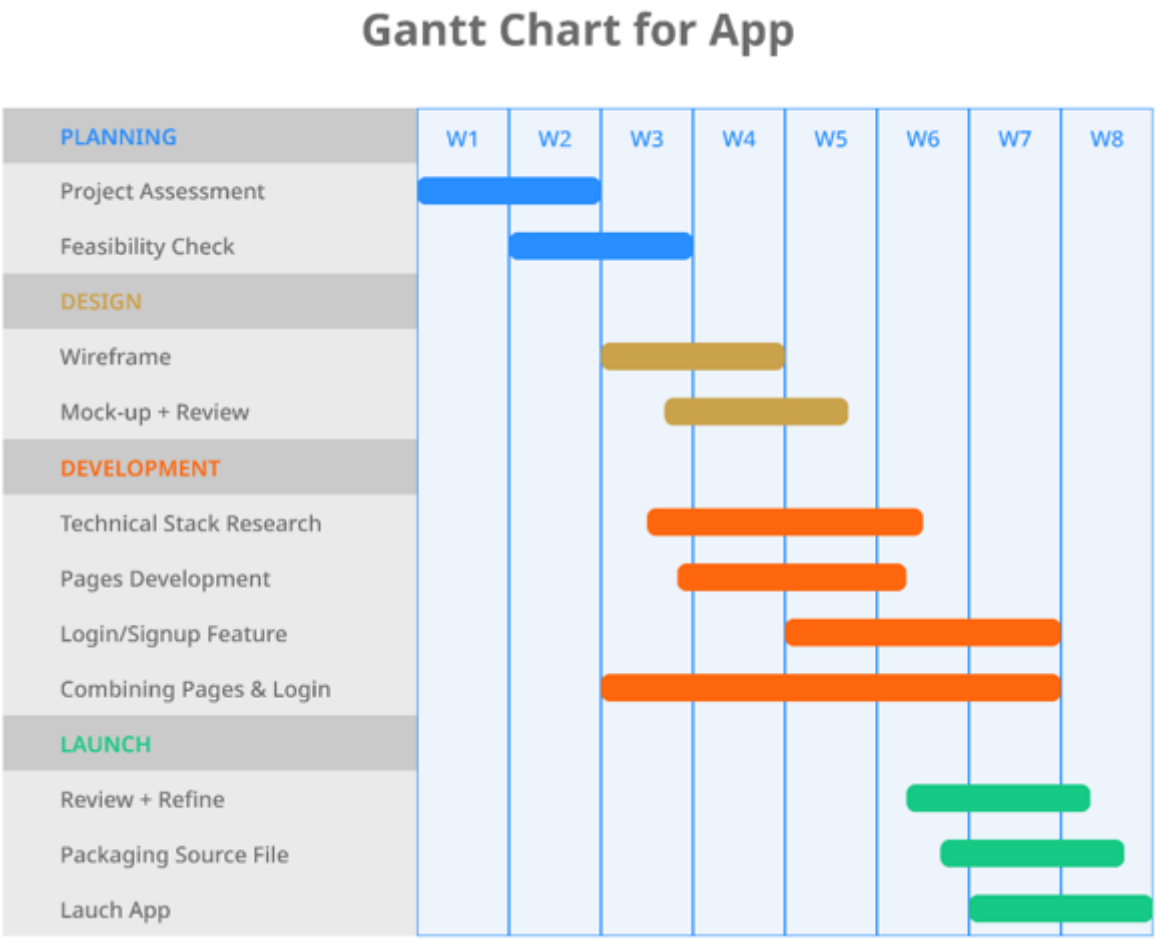


Рис. 3 – Приклад діаграма Ганта у процесі розробки додатку [7]

Переваги:

- Розширені можливості для керування проектами та завданнями.
- Зручний інтерфейс зі зрозумілим способом організації завдань.
- Гнучкість у призначенні виконавців та установці термінів.
- Багато засобів аналізу продуктивності та генерації звітів.
- Інтеграція з іншими інструментами та платформами.

Недоліки:

- Не так простий у використанні, як деякі інші системи.
- Обмеження у безкоштовній версії, особливо для більших команд або проектів.

- Можливе затримування у роботі інтерфейсу при великій кількості завдань.

3. Jira

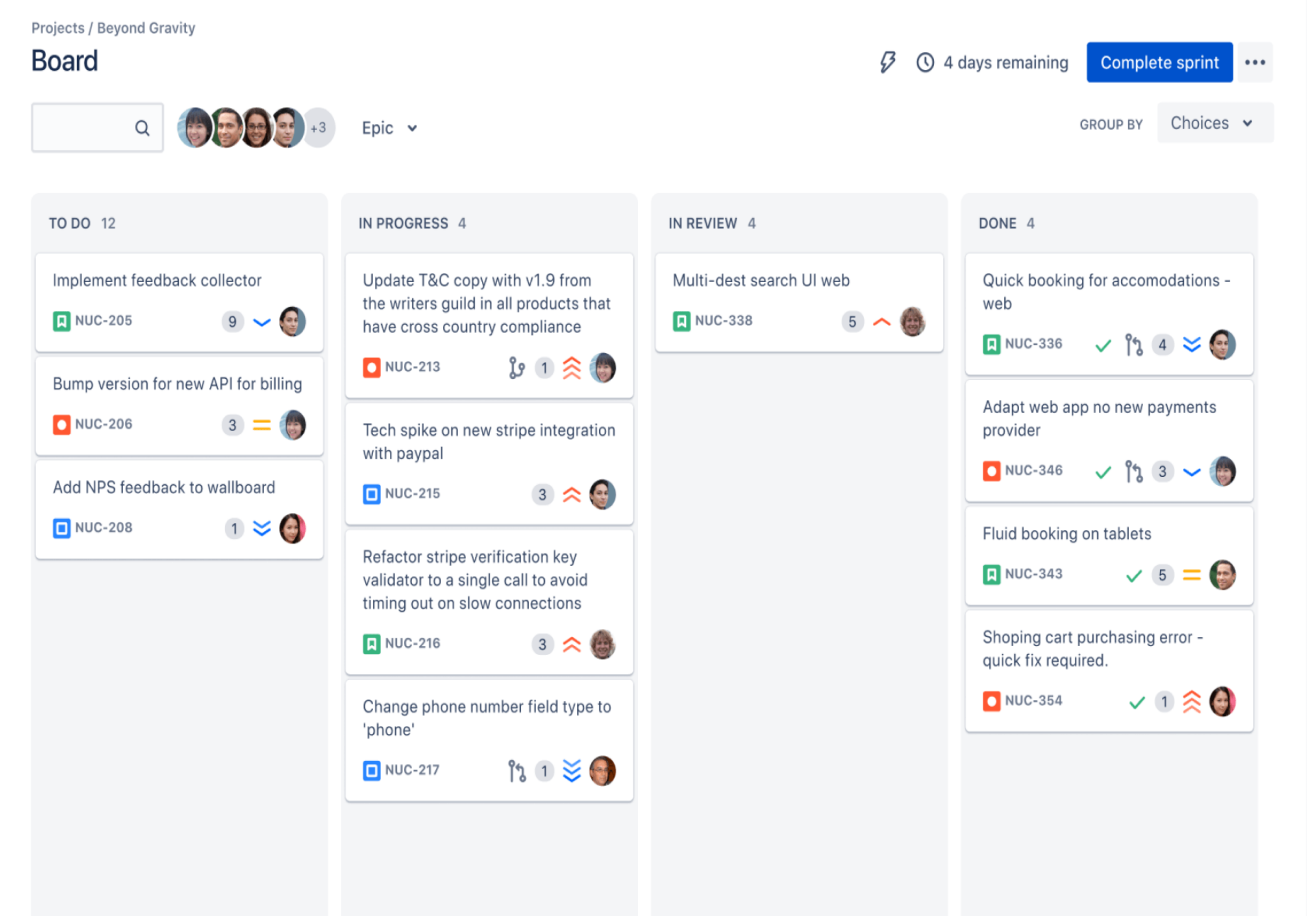


Рис. 4 – Система “Jira” [8]

Jira - це популярна система управління проектами, особливо в IT-сфері. Вона дозволяє створювати та відстежувати завдання, призначати їх виконавцям, встановлювати пріоритети та терміни виконання. Jira також надає можливості для аналізу продуктивності, такі як звіти про прогрес, швидкість виконання завдань та інші показники продуктивності.

Переваги:

- Потужний інструмент для керування проектами, особливо в IT-сфері.
- Велика кількість налаштувань та розширень для різних потреб.
- Детальне відстеження прогресу завдань та розподілу ресурсів.
- Зручні засоби аналізу продуктивності та статистики проектів.
- Інтеграція з багатьма іншими інструментами розробки.

Недоліки:

- Складніший для вивчення та використання порівняно з іншими системами.
- Висока вартість ліцензій та підтримки.
- Можливість перенавантаження інтерфейсу та ускладнення роботи при неадекватному налаштуванні.

4. Microsoft Planner

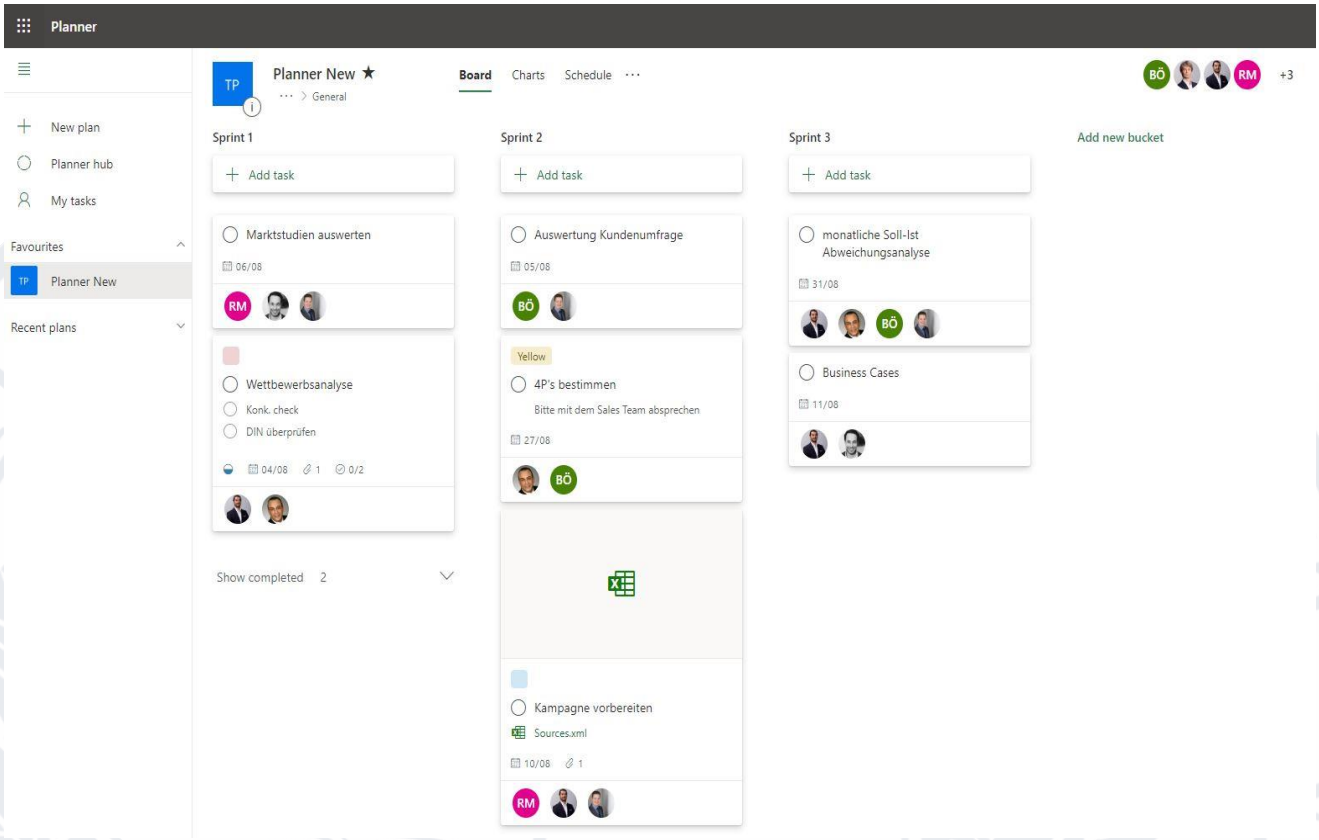


Рис. 5 – Система “Microsoft Planner”

Microsoft Planner - це інструмент, вбудований у платформу Office 365, який дозволяє створювати завдання, встановлювати терміни, делегувати їх виконавцям та відстежувати прогрес. Ви можете організовувати завдання у вигляді дошок та списків, а також спостерігати за змінами стану завдань та отримувати сповіщення.

Переваги:

1. Інтегрований з іншими продуктами Office 365.

2. Зручний інтерфейс з легкою організацією завдань та списків.

3. Можливість призначення виконавців та установки термінів.

4. Прості інструменти аналізу продуктивності та сповіщення про зміни.

5. Доступність на різних платформах та пристроях.

Недоліки:

1. Обмежені можливості порівняно з іншими системами управління проектами.

2. Відсутність деяких розширених функцій та налаштувань.

3. Залежність від екосистеми Office 365 та підписки на платформу.

Висновок до розділу 1

В даному розділі була обговорена важливість системи розподілу завдань та аналізу продуктивності виконавців, фактори, що призводять до зростання попиту на такі сервіси і їх актуальність у сучасному світі. Крім того, були наведені опис функцій, які повинні бути реалізовані в таких програмах, і інформація, з якою необхідно ознайомитись. Також було розглянуто кілька найпопулярніших аналогів, зроблено порівняння їх переваг та недоліків.

РОЗДІЛ 2
АНАЛІЗ ТА ВИБІР АКТУАЛЬНИХ ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Вибір правильного стеку технологій є важливим фактором успіху при реалізації проєкту. В цьому розділі будуть розглянуті основні технології та інструменти, які були використані для розробки API. Будуть надані опис їх можливостей, переваг та недоліків.

2.1 Технології

Платформа .NET [9-11]

Мета Microsoft полягала у створенні єдиної платформи для вирішення різних проблем розробників, і ця мета була досягнута з випуском .NET.

Безсумнівно, .NET відіграє ключову роль у галузі розробки програмного забезпечення. Його популярність серед розробників є очевидною, що можна побачити за кількістю проектів з відкритим кодом у всьому світі і включенням мови програмування C# в п'ятірку найбільш популярних мов програмування. Завдяки постійній підтримці з боку розробників, популярність платформи .NET буде продовжувати зростати.

Для кращого розуміння потужності .NET, давайте розглянемо його технічні основи.

.NET - це платформа, яка дозволяє створювати різноманітні типи додатків. Розроблена компанією Microsoft з відкритим вихідним кодом, ця платформа підтримує кросплатформену розробку, що означає, що додатки можуть працювати на різних операційних системах.

Крім того, .NET підтримує використання різних мов програмування та бібліотек для створення веб-додатків, мобільних додатків, десктопних додатків, додатків для "речей Інтернету" (IoT) та багатьох інших.

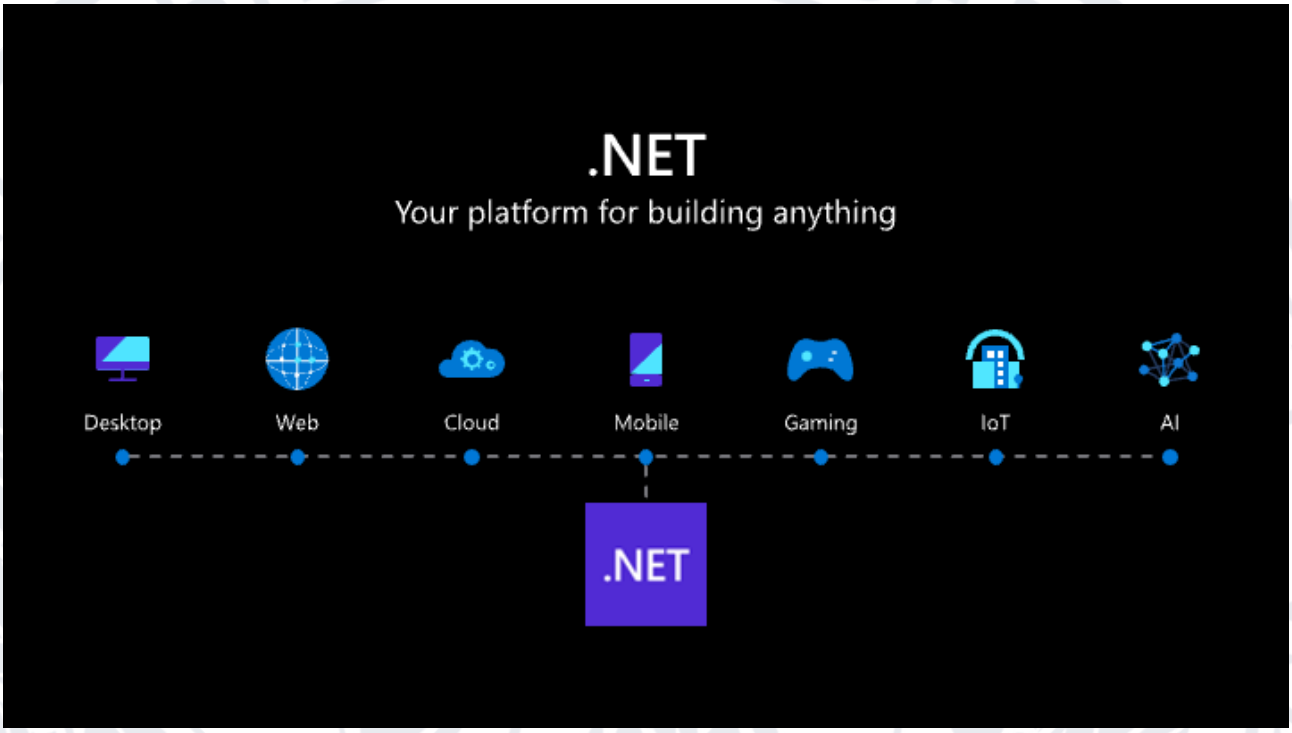


Рис. 6 – сфери розробки за допомогою .NET

Microsoft прямо підтримує декілька мов програмування на платформі .NET. Ось деякі з них:

- C# (C sharp): Це сучасна об'єктно-орієнтована мова програмування, яка належить до сімейства мов C. Синтаксис C# може бути знайомим розробникам, які працювали з C, C++, Java або JavaScript.
- F# (F-sharp): Це функціонально-орієнтована мова програмування, яка також підтримує об'єктно-орієнтовану парадигму. F# належить до сімейства мов ML.
- Visual Basic: Це історична мова програмування від Microsoft, яка стала повноцінною об'єктно-орієнтованою мовою програмування в контексті .NET.

Платформа .NET підтримує Common Language Infrastructure (CLI)[12-14], що означає, що код, написаний на будь-якій з цих мов, компілюється в Common Intermediate Language (CIL). Це забезпечує взаємодію між мовами на платформі .NET.

Крім того, є багато інших мов програмування, які можуть компілюватися в .NET CIL. Наприклад, ClojureCLR, Eiffel, IronPython, PowerBuilder та багато інших.

Архітектура та компоненти .NET

.NET Components

Архітектура .NET базується на двох основних принципах:

- CoreCLR: Це середовище виконання .NET, яке відповідає за виконання програм на мові CLI (Common Intermediate Language). Воно включає в себе "just-in-time" компілятор, який перетворює код на мові CLI в машинний код під час виконання.
- CoreFX: Це API платформи .NET, яке реалізує стандартні бібліотеки CLI. Ці бібліотеки надають широкий спектр функціональних можливостей, таких як робота з файловою системою, обробка винятків, мережева комунікація, потоки, відображення та інше. Компонент CoreFX іноді називають Unified Base Class Library.

NET Application Models

.NET також має різні framework'и для створення різних типів додатків, які побудовані поверх основних компонентів:

- ASP.NET: Це framework для створення веб-додатків та веб-API.

- Windows Presentation Foundation (WPF): Це framework для створення графічного інтерфейсу користувача для настільних програм Windows.

- Xamarin: Це framework з підтримкою крос-платформної розробки мобільних та настільних додатків.

- Blazor: Це framework для створення клієнтських веб-додатків за допомогою мови програмування C#.

- ML.NET: Це framework для програмування в області машинного навчання.

Крім цих framework'ів, .NET також надає інструменти для різних завдань програмування, таких як робота з файлами, мережевою комунікацією, безпекою доступу, базами даних та іншими.

У загальнодоступному репозиторії NuGet також доступна велика кількість бібліотек. NuGet - це менеджер пакетів для .NET, який дозволяє створювати, ділитися та використовувати різноманітні бібліотеки .NET для різних потреб.

Підтримка проектування та розробки .NET

Підтримка .NET для розробки програмного забезпечення не обмежується лише кількома мовами програмування, які можна використовувати. Вона також сприяє використанню найкращих практик, дозволяючи розробникам застосовувати обрані підходи при створенні програм.

Наприклад, .NET підтримує використання Dependency Injection (DI) для роз'єднання компонентів програми. DI допомагає створювати якісне програмне забезпечення, зменшуючи взаємозалежність компонентів та полегшуючи їх повторне використання. Крім того, DI сприяє тестуванню компонентів.

У сфері тестування, .NET також надає підтримку модульних та інтеграційних тестів за допомогою xUnit.

Оскільки розробники мають різний досвід та вподобання, .NET надає їм вибір з різних підходів. Вони можуть використовувати .NET CLI, що дозволяє

створювати нові проекти, додавати залежності, збирати та запускати їх, через командний рядок. Вони також можуть скористатись Visual Studio Code, який є розширеним міжплатформним редактором, побудованим на основі .NET CLI. Іншою можливістю є використання потужного інтегрованого середовища розробки (IDE), такого як Visual Studio, яке доступне як для Windows, так і для Mac, і надає інтерактивний досвід програмування.

Без залежності від обраного інструменту, розробники можуть скористатись широким спектром шаблонів проектів, щоб швидко почати розробку нової програми.

Мова програмування C# [15-19]

C# є сучасною мовою програмування, яка поєднує об'єктно-орієнтований підхід зі строгим типізуванням. Вона дозволяє розробникам створювати безпечні та надійні програми, які працюють у середовищі .NET. C# належить до широко відомої родини мов C і буде знайомою для тих, хто вже працював з C, C++ або Java.

C# має спрямованість на компоненти як мова програмування і надає мовні конструкції, які безпосередньо підтримують цю концепцію. Це робить C# ідеальним вибором для створення та використання програмних компонентів. З часом мова C# розширилася функціями, що підтримують нові завдання розробки програмного забезпечення та сучасні рекомендації. Головною характеристикою C# є його об'єктно-орієнтований підхід, де ви визначаєте типи та їх поведінку.

Ось лише кілька функцій мови C#, які допомагають створювати надійні та стабільні програми[20-22]:

- Збирання сміття автоматично звільняє пам'ять, яку займають об'єкти, недоступні для використання.

- Типи, які допускають значення null, забезпечують захист від змінних, що не посилаються на виділені об'єкти.

- Обробка винятків надає структурований та розширюваний підхід до виявлення та обробки помилок.

- Лямбда-вирази підтримують прийоми функціонального програмування.

- Синтаксис LINQ створює єдиний шаблон для роботи з даними з будь-якого джерела.

- Підтримка асинхронних операцій надає можливість створювати розподілені системи.

- C# має єдину систему типів, де всі типи успадковуються від кореневого типу object, включаючи типи-примітиви, такі як int або double.

- Всі типи використовують загальний набір операцій, а значення будь-якого типу можна зберігати, передавати та обробляти однаковим способом.

- C# підтримує як типи посилання, що визначаються користувачами, так і типи значень. Він також дозволяє динамічне виділення об'єктів та зберігання простих структур у стеку.

- Універсальні методи та типи забезпечують підвищену безпеку типів та продуктивність.

- C# надає ітератори, які дозволяють розробникам класів колекцій визначати поведінку для коду клієнта.

- C# надає підтримку керування версіями, що забезпечує сумісність програм та бібліотек з часом. Управління версіями суттєво вплинуло на такі аспекти розробки C# як роздільні модифікатори virtual і override, правила перевантаження методів та підтримка явного оголошення членів інтерфейсу.

ASP.NET Core [22-25]

ASP.NET Core є технологією розробки веб-додатків на платформі .NET, розробленою компанією Microsoft. Для програмування веб-додатків на ASP.NET Core використовуються мови C# і F#.

Історія ASP.NET почалася з випуску першої версії, що спочатку була призначена для роботи виключно на веб-сервері IIS під управлінням Windows (хоча завдяки проекту Mono ASP.NET додатки також могли запускатися на Linux).

Однак у 2014 році відбулися значні зміни, які фактично революціонізували розвиток платформи: Microsoft почала активно розвивати ASP.NET як кросплатформену технологію і випустила її як проект з відкритим вихідним кодом. Цей розвиток отримав назву ASP.NET Core[26], і так його офіційно називає Microsoft й до цього часу. Перша версія оновленої платформи була випущена в червні 2016 року. Тепер ASP.NET Core працює не лише на Windows, але і на MacOS та Linux. Вона стала більш легковажною, модульною, зручною для конфігурації і відповідає вимогам сучасності.

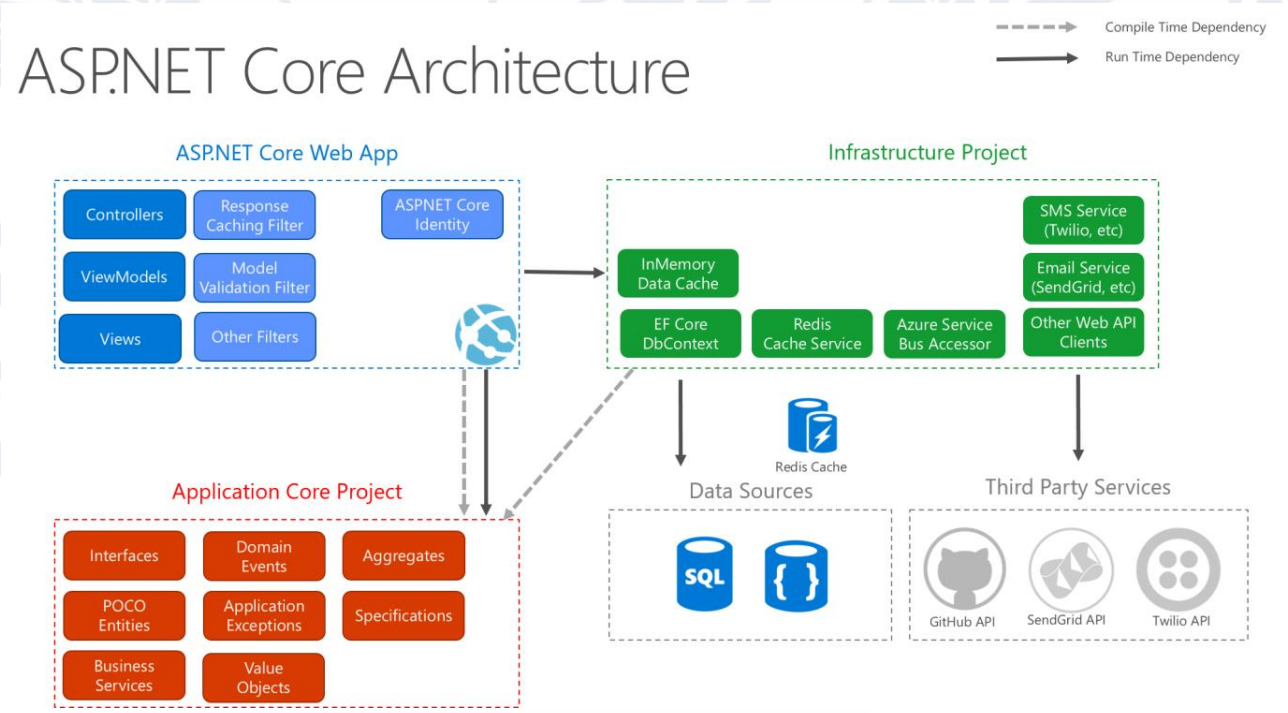


Рис. 7 – Приклад чистої архітектури ASP.NET Core [27]

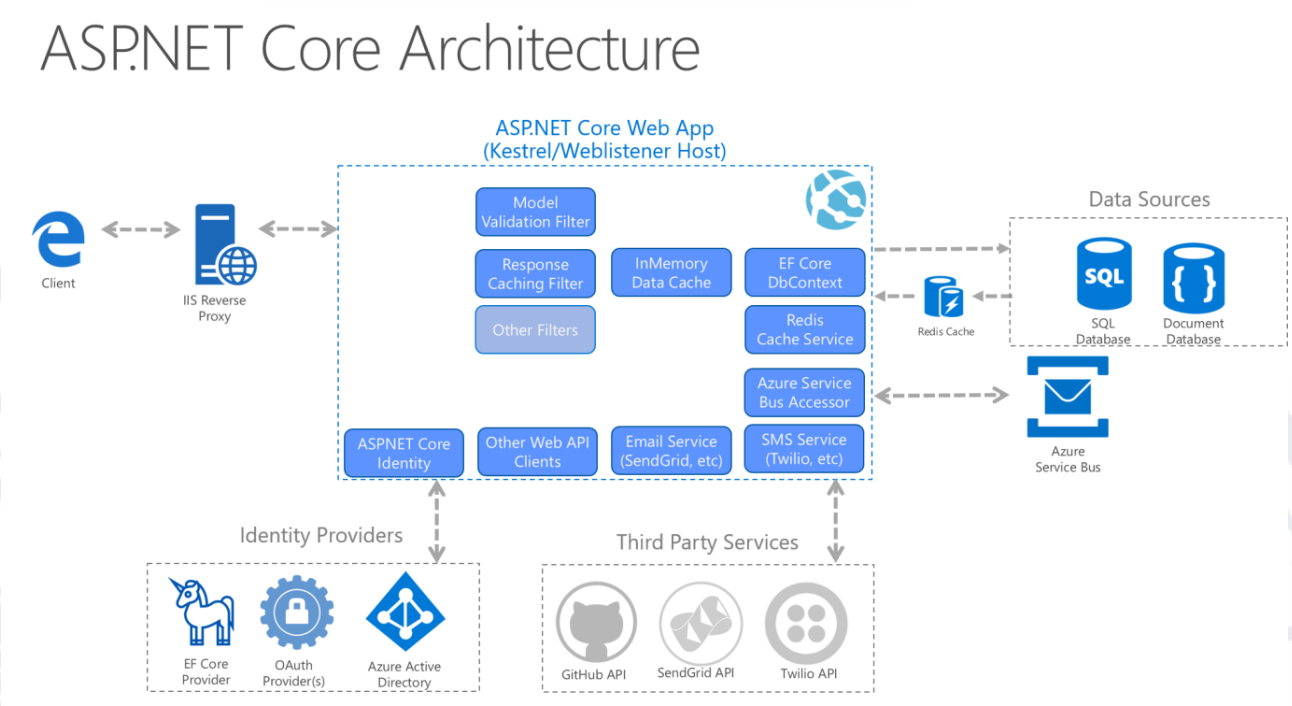


Рис. 8 – Приклад архітектури додатку ASP.NET Core під час виконання [27]

Моделі розробки

ASP.NET Core [27] надає можливість створювати веб-застосунки з використанням різних моделей розробки.

- Основною моделлю є базовий ASP.NET Core, який підтримує всі основні складові для створення сучасного веб-додатку, включаючи маршрутизацію, конфігурацію, логування та роботу з різними системами баз даних. Додатково була впроваджена спрощена модель, відома як Minimal API, яка спрощує процес розробки та написання коду програми. Інші моделі розробки працюють на основі базового функціоналу ASP.NET Core.

- ASP.NET Core MVC представляє загальний підхід до побудови програми навколо трьох основних компонентів: моделей (Model), уявлень (View) і контролерів (Controller). Моделі відповідають за роботу з даними, контролери

містять логіку обробки запитів, а уявлення визначають візуальну частину додатку.

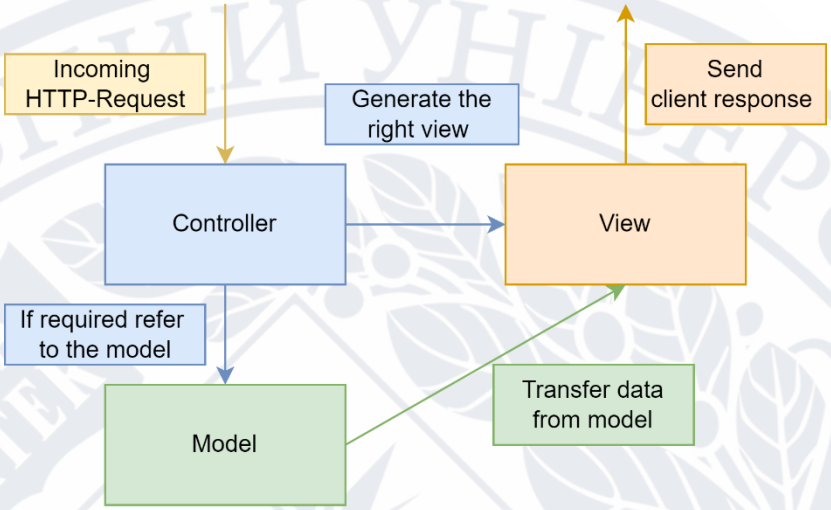


Рис. 9 – Принцип роботи MVC

- ASP.NET Core Web API реалізує патерн REST, де кожному типу HTTP-запиту (GET, POST, PUT, DELETE) призначений окремий ресурс. Ресурси визначаються за допомогою методів контролера Web API. Ця модель особливо підходить для розробки односторінкових додатків, але може бути використана і в інших випадках.

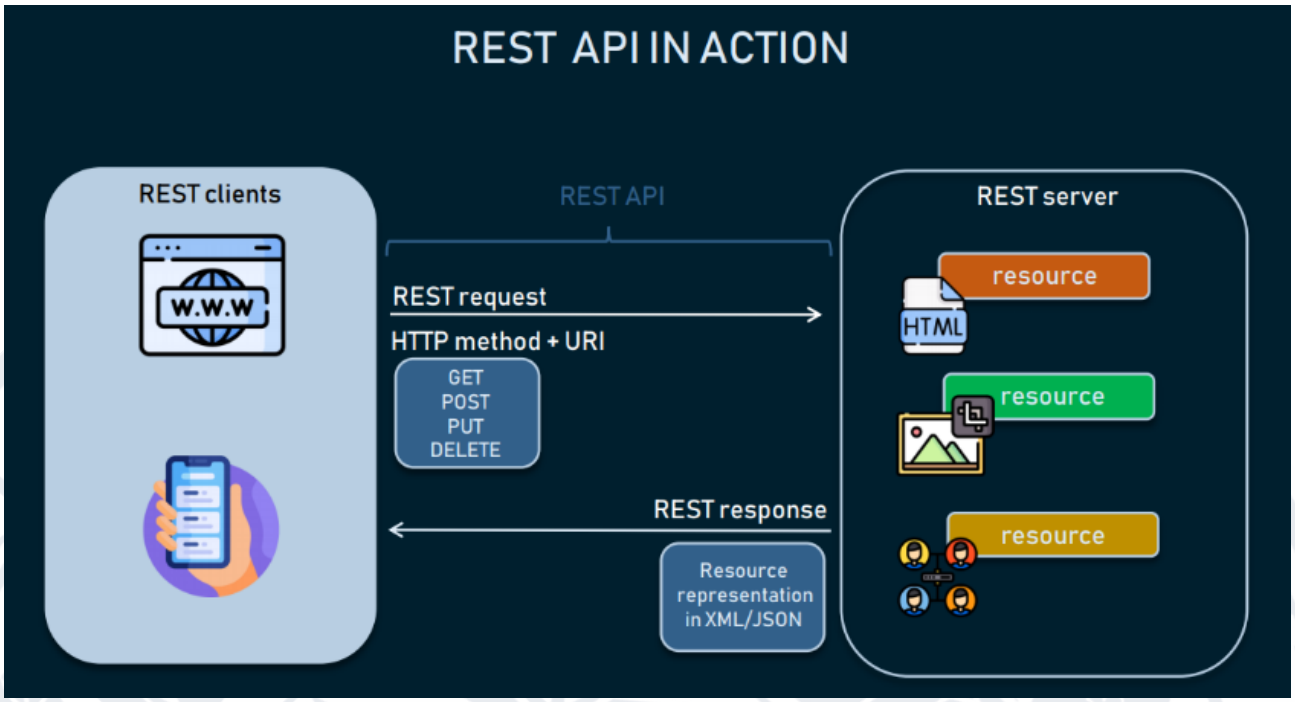


Рис. 10 – Принцип роботи REST Web API

Особливості платформи [28]

• ASP.NET Core використовує платформу .NET і надає доступ до всього її функціоналу.

• Для розробки проектів ASP.NET Core використовуються мови програмування, які підтримуються платформою .NET. Офіційно вбудована підтримка мовами для проектів ASP.NET Core - C# і F#.

• ASP.NET Core є крос-платформним фреймворком, який дозволяє розгортати додатки на операційних системах Windows, Mac OS і Linux. Це означає, що ви можете створювати крос-платформні програми для Windows, Linux і Mac OS за допомогою ASP.NET Core, а також запускати їх на цих операційних системах.

• Завдяки модульності фреймворку, всі компоненти веб-програми можуть завантажуватися як окремі модулі через пакетний менеджер NuGet.

• ASP.NET Core підтримує роботу з багатьма поширеними системами управління базами даних, такими як MS SQL Server, MySQL, Postgres і MongoDB.

• Фреймворк складається з незалежних компонентів, які можна використовувати вбудовану реалізацію, розширити їх за допомогою механізму спадкування або створити власні компоненти з власним функціоналом.

• Інструментарій для розробки програм на ASP.NET Core є дуже розширеним. Можна використовувати такі інструменти, як Visual Studio від Microsoft, яке має багатий функціонал, або середовище розробки Rider від компанії JetBrains.

• Крім того, .NET CLI дозволяє створювати і запускати проекти ASP.NET Core з консолі. Це означає, що для написання коду можна використовувати звичайний текстовий редактор, наприклад, Visual Studio Code.

2.2 Обрана IDE

Microsoft Visual Studio [29-30]

Microsoft Visual Studio є інтегрованою середовищем розробки, яке не лише дозволяє розробляти, писати та редагувати код, але й надає ряд інших функцій.

Воно підтримує використання практично будь-якої мови програмування, включаючи:

• C, C++, C++/CLI

• .NET

• JavaScript

• TypeScript

• XML

• XSLT

• HTML

• CSS

Крім того, деякі інші мови, такі як Python, Ruby, Node.js і N, доступні через плагіни. Цей широкий вибір мов програмування означає, що ви маєте значну гнучкість у виборі інструментів. Microsoft Visual Studio прагне бути повним і всеосяжним середовищем для всіх ваших потреб у кодуванні.

Для досягнення цієї мети воно пропонує чотири основні функції: розробка, налагодження, тестування та спільна робота. Ви можете використовувати Visual Studio для створення програмного забезпечення, відлагодження коду, проведення тестування та спільної роботи з іншими розробниками.

Розробка

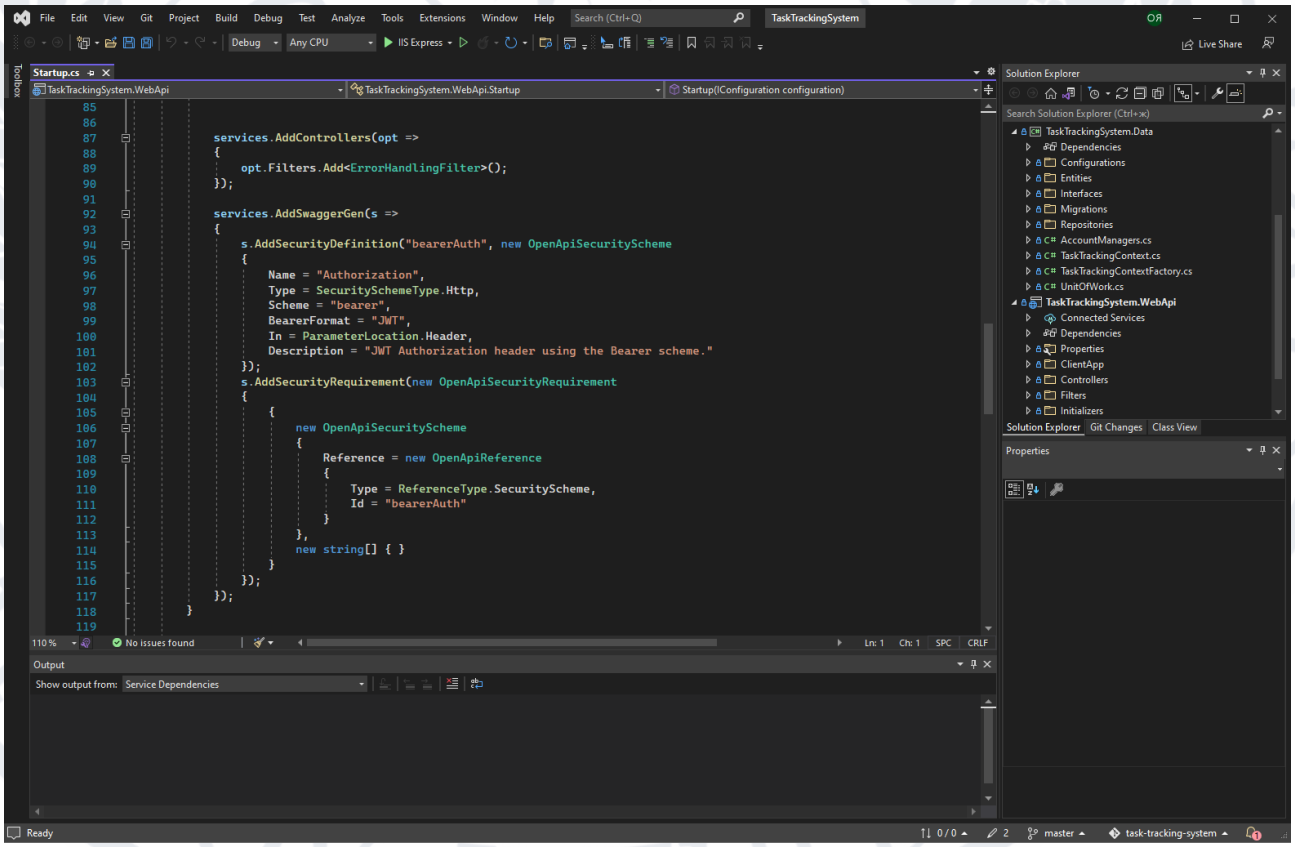


Рис. 11 – Розробка за допомогою Visual Studio

• Visual Studio надає підказки та допомогу під час написання коду, незалежно від використовуваної мови програмування.

• IntelliSense надає описи API під час введення тексту та автоматично завершує код, що прискорює процес розробки.

• Програма відстежує ваше місце у процесі написання коду, навіть якщо ви переглядаєте інші частини коду.

• Функція "Знайти всі посилання" дозволяє групувати, фільтрувати та шукати посилання в результатах пошуку.

• Code Lens допомагає зрозуміти структуру виклику вашого коду та перейти до пов'язаних функцій, а також вказує, хто останній редагував код.

• Піктограми-лампочки повідомляють вас про поширені помилки кодування та пропонують виправлення, навіть при введенні коду вперше.

• Список помилок об'єднує всі проблеми з кодуванням в одному місці, що дозволяє легко вирішувати проблеми.

• Code Link може знаходити потенційні рішення для складних проблем.

• Visual Studio виконує багато рутинних завдань з рефакторингу під час розвитку вашого проекту, що спрощує роботу з кодом.

Налагодження (Debugging)

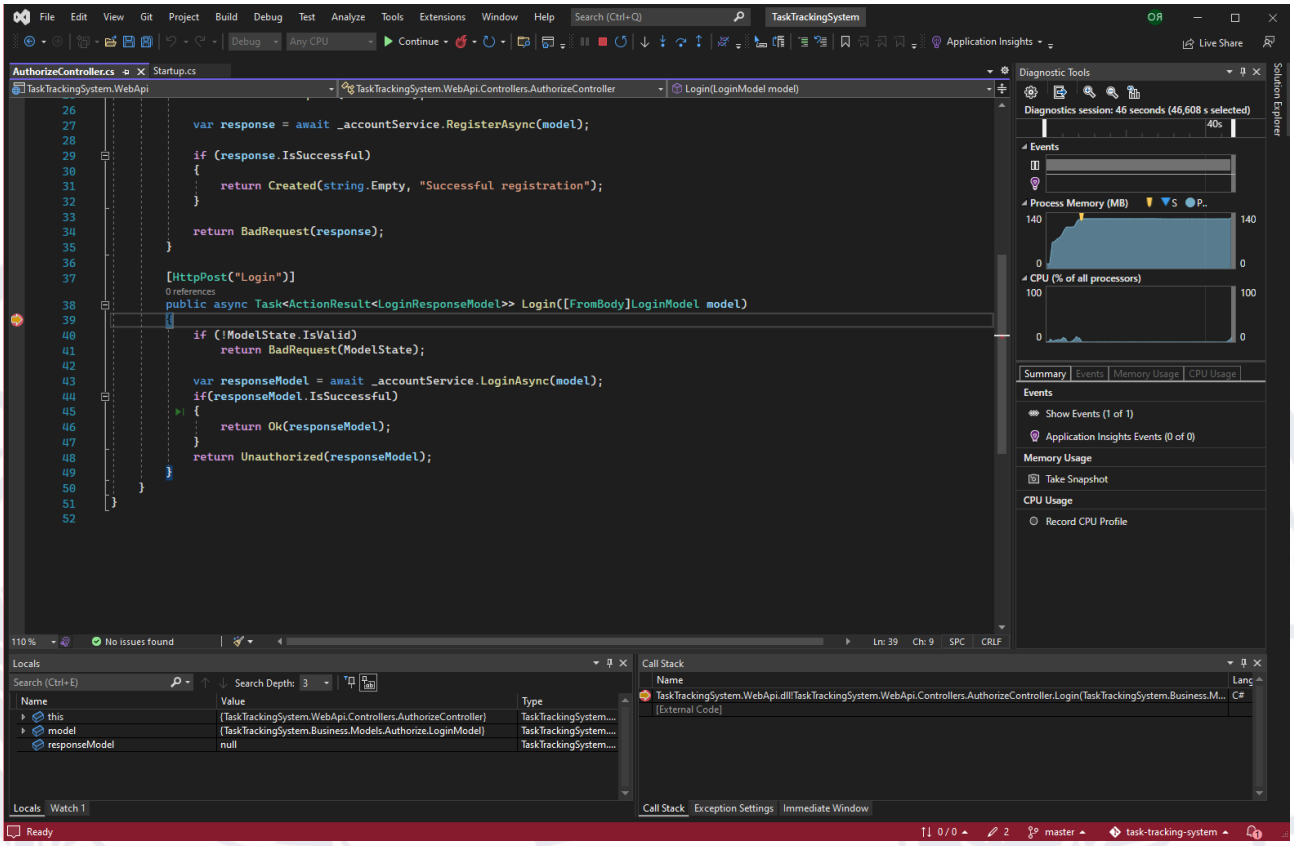


Рис. 12 – Процес налагодження використовуючи Visual Studio

• Налагодження працює навіть при розробці коду на декількох мовах програмування одночасно.

• Ви можете налагоджувати код незалежно від платформи, на якій працює ваша програма (Windows, Android, Azure, iOS і т.д.).

• Visual Studio надає повний контроль над процесом налагодження і дозволяє вам вибирати місця для зупинки потоків та перевірки коду.

• Програма також надає гнучкість у перевірці коду, дозволяючи переглядати значення змінних та складних виразів в будь-якому місці коду.

• Ви отримуєте повідомлення про помилки, які виникають під час налагодження.

• Програма дозволяє легко перевіряти код у багатьох потоках.

• Функції PerfTips та інструменти діагностики дозволяють отримати більше інформації про продуктивність вашого коду та використання пам'яті.

Тестування

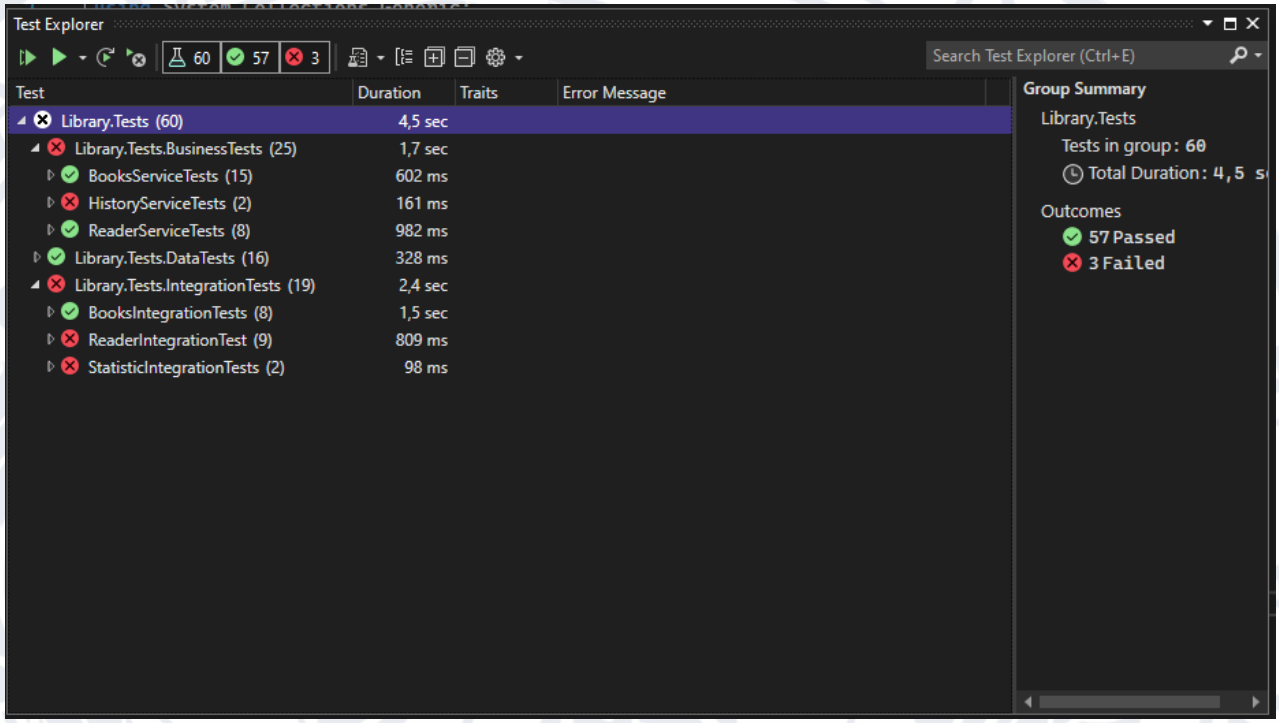


Рис. 13 – Інструмент для тестування у Visual Studio

• Visual Studio пропонує широкий вибір шаблонів і фреймворків для написання, виконання та налагодження модульних тестів.

• За допомогою IntelliTest зменшуються зусилля, необхідні для створення і підтримки модульних тестів.

• Модульне тестування в реальному часі дозволяє переконатися, що ваші зміни не порушують наявні тести.

• Тестування користувацького інтерфейсу дозволяє управляти вашим додатком через його інтерфейс.

• Ви можете проводити тестування вашого коду великого масштабу для сотень тисяч одночасних користувачів з усього світу.

• Результати тестування можна переглядати навіть під час організації, виконання та налагодження тесту. Ви можете автоматично запускати тести після кожної збірки.

2.3 База даних

Microsoft SQL Server[31-32] був обраний як сервер бази даних. Microsoft SQL Server, також відомий як MS SQL Server, є реляційною базою даних, розробленою компанією Microsoft. Сервер бази даних - це програмне забезпечення, яке використовується для зберігання даних, а інші програми отримують доступ до цих даних за допомогою мови запитів, яка в разі MS SQL Server називається SQL (Structured Query Language).

Основним інструментом інтерфейсу для MS SQL Server є SQL Server Management Studio (SSMS), який підтримує як 64-розрядні, так і 32-розрядні операційні системи. MS SQL Server підтримує стандарт ANSI SQL, який є стандартною мовою структурованих запитів. Однак MS SQL Server також використовує власну мову, відому як Transact-SQL або T-SQL, яка надає додаткові можливості, такі як збережені процедури, обробка винятків та змінні.

До переваг MS SQL Server можна віднести наступне:

- Просте встановлення: Встановлення продуктів Microsoft, включаючи MS SQL Server, є легким завдяки графічному інтерфейсу та детальним інструкціям.

- Покращена продуктивність: MS SQL Server має потужні можливості стиснення та шифрування, що покращує функції зберігання та пошуку даних.

- Безпека: MS SQL Server відомий своєю високою безпекою завдяки складним алгоритмам шифрування, що робить його вразливим на мінімальний ризик атак.

- Різні випуски і варіанти цін: MS SQL Server доступний в різних версіях, що відповідає потребам корпоративних організацій і домашніх користувачів з різними бюджетами.

- Механізм відновлення та резервного копіювання даних: MS SQL Server має потужні функції для відновлення та резервного копіювання даних, що дозволяє відновлювати втрачені або пошкоджені дані. SQL Server Database Engine керує зберіганням даних і виконує запити, а також керує транзакціями, індексами та іншими аспектами обробки даних.

Висновок до розділу 2

У цьому розділі будуть розглянуті інструменти та технології, які сприяють сучасній та ефективній розробці Web API. Також будуть розглянуті ключові інструменти та переваги обраного сервера баз даних.

РОЗДІЛ 3
РОЗРОБКА ТА АНАЛІЗ ПРОГРАМНОГО ПРОДУКТУ

3.1 Архітектура

REST (Representational State Transfer) [33-36]

REST є найпопулярнішим архітектурним стилем сучасності, який встановлює набір правил для доступу до інформаційних ресурсів через мережеві протоколи. Щоб веб-апі можна було вважати RESTful, воно повинно дотримуватися кількох ключових принципів:

1. **Клієнт-серверна архітектура:** Це означає взаємодію між клієнтами і серверами через запити та відповіді. У цій моделі клієнт (наприклад, мобільний застосунок) відправляє HTTP запити на сервер, який містить необхідні ресурси, і отримує від нього відповіді на основі запиту.
2. **Уніфікований інтерфейс:** Цей принцип полягає в чіткому визначенні формату обміну даними між клієнтом і сервером. Зазвичай це включає в себе використання JSON, XML або HTML для обміну інформацією. Клієнт має заздалегідь визначений набір запитів для взаємодії з ресурсами сервера та отримання чітких і зрозумілих відповідей на свої запити. Взаємодія з REST API здійснюється через гіперпосилання.
3. **Відсутність стану (Statelessness):** Під час взаємодії між сервером і клієнтом, клієнт повинен надсилати усю необхідну інформацію для обробки запиту, не очікуючи, що сервер пам'ятає попередні запити.
4. **Багатошарова архітектура:** Це передбачає розділення складних компонентів на кілька простих проміжних слоїв з певною ієрархією.

Чиста архітектура (Clean Architecture)[37-39]

Чиста архітектура (Clean Architecture) - це архітектурний підхід до розробки програмного забезпечення, який наголошує на відокремленні рівнів абстракції та залежності між компонентами. Ця архітектура сприяє створенню гнучких, розширюваних та тестованих систем. Основна ідея полягає в тому, щоб розділити програму на чотири основні рівні, які ізолюють бізнес-логіку від деталей інтерфейсу користувача та технічних аспектів реалізації. Ось опис кожного з цих рівнів:

- *Рівень Домену (Domain Layer):*

Рівень Домену - це серце системи, де знаходиться бізнес-логіка програми. Він включає в себе об'єкти, сутності та правила, які визначають ядро додатку. Цей рівень повинен бути абсолютно незалежним від будь-яких технічних деталей, таких як база даних чи фреймворки. Всі правила бізнесу повинні бути визначені саме тут.

- *Рівень Застосування (Application Layer):*

Рівень Застосування містить логіку, яка відповідає за взаємодію між рівнем Домену та рівнем Інфраструктури. Це місце, де відбувається оркестрація операцій, обробка запитів користувачів і виконання команд з рівня інтерфейсу користувача. Рівень Застосування слідкує за тим, щоб дані з рівня Домену оброблялися правильно та відповідно до бізнес-правил.

- *Рівень Інфраструктури (Infrastructure Layer):*

Рівень Інфраструктури забезпечує зовнішні залежності системи, такі як бази даних, зовнішні сервіси, фреймворки та інші інструменти. Він містить код, який взаємодіє з технічними деталями і дозволяє програмі взаємодіяти з зовнішніми системами. Цей рівень забезпечує імплементацію інтерфейсів та реалізацію зберігання даних.

- *Рівень Представлення (Presentation Layer):*

Рівень Представлення відповідає за взаємодію із користувачем. Це може бути веб-інтерфейс, мобільний застосунок чи інше зовнішнє засіб взаємодії з системою. Його головна мета - приймати введення користувача, відображати дані та передавати команди до рівня Застосування.

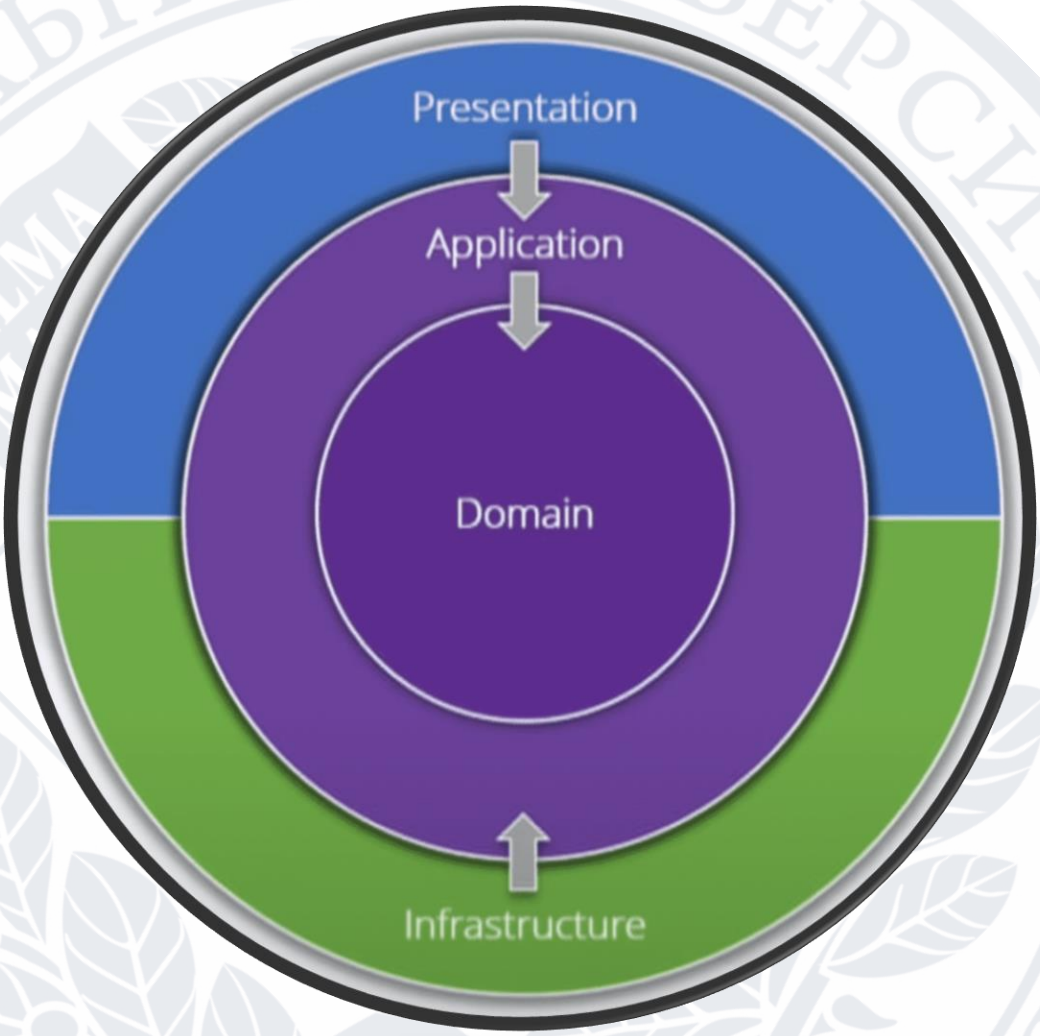


Рис. 14 – Чиста архітектура [40]

Важливо відзначити, що ці рівні повинні мати однобічні залежності. Це означає, що рівень Представлення залежить від Рівня Застосування, Рівень Застосування залежить від Рівня Домену, а Рівень Домену абсолютно незалежний від інших рівнів. Ця структура забезпечує високу модульність, тестируваність та легкість змін у програмній системі.

Ця архітектурна модель допомагає зберігати бізнес-логіку чистою та незалежною від технічних деталей, що робить систему більш гнучкою та легко розширюваною в майбутньому.

Принцип Command Query Responsibility Segregation [41-42]

Принцип CQRS (Command Query Responsibility Segregation) — це концепція проектування програмного забезпечення, яка розділяє операції читання (запити) і запису (команди) в системі. Замість того, щоб використовувати одну модель для обох операцій, CQRS рекомендує використовувати дві різні моделі:

- **Модель команд (Command Model):** Використовується для обробки команд, які змінюють стан системи. Команди вказують системі на виконання певної дії, наприклад, створення нового запису або зміну існуючого.
- **Модель запитів (Query Model):** Використовується для виконання операцій читання. Запити не змінюють стан системи, а лише повертають дані. Це може бути зручно для отримання швидкого доступу до даних без обробки логіки команд.

Розглянемо детальніше модель команд і модель запитів в контексті CQRS:

Модель команд (Command Model):

1. **Операції запису:**
    - **Команди (Commands):** Це структуровані об'єкти, що представляють інструкції для внесення змін у стан системи. Команди можуть представляти операції створення, оновлення або видалення даних.
    - **Валідація та бізнес-логіка:** В моделі команд зазвичай містяться механізми валідації, які перевіряють правильність команди перед внесенням змін. Тут також розміщається бізнес-логіка, пов'язана з операціями запису.
2. **Асинхронність:**
    - **Асинхронні операції:** Оскільки запис може вимагати багато часу або ресурсів, обробка команд часто відбувається асинхронно. Це дозволяє системі швидше реагувати на команди, а користувачам отримувати відгук без чекання завершення довгих операцій запису.
3. **Моделювання стану:**
    - **Зміни в стані системи:** Модель команд моделює, як система змінює свій стан відповідно до отриманих команд. Це може включати в себе збереження або оновлення даних в базі даних та інші дії, спрямовані на забезпечення консистентності даних.

Відповідно структура **моделі команди** може складатись з:

1. **Команди (Commands):** Це структуровані об'єкти або класи, які містять дані інструкції для внесення змін у стан системи. Кожна команда визначає, що і як потрібно змінити стан системи.
2. **Обробники команд (Command Handlers):** Це класи або компоненти, які відповідають за виконання конкретних команд. Обробники команд

обробляють команди, валідують їх, взаємодіють з доменною логікою та вносять зміни у стан системи.

3. **Модель домену (Domain Model):** Це представлення бізнес-логіки та правил, які визначають, як система повинна здійснювати операції запису. Модель домену включає в себе сутності, значення та агрегати.
4. **Механізми валідації:** Це компоненти, які перевіряють коректність та прийнятність команд перед їх обробкою. Валідація може включати перевірку формату даних, прав доступу та інші бізнес-правила.

Розберемо структуру моделі команди на основні функціоналу сторення нової моделі ролі на проєкті:

```
15 usages  Michael Tarnorutsky +1
public record CreateTodoItemCommand : IRequest<int>
{
    8 usages
    public int ListId { get; init; }

    9 usages
    public string? Title { get; init; }
}

Jason Taylor +1
public class CreateTodoItemCommandHandler : IRequestHandler<CreateTodoItemCommand, int>
{
    private readonly IApplicationDbContext _context;

    Jason Taylor
    public CreateTodoItemCommandHandler(IApplicationDbContext context)
    {
        _context = context;
    }

    Jason Taylor +1
    public async Task<int> Handle(CreateTodoItemCommand request, CancellationToken cancellationToken)
    {
        var entity = new TodoItem
        {
            ListId = request.ListId,
            Title = request.Title,
            Done = false
        };

        _context.TodoItems.Add(entity);

        await _context.SaveChangesAsync(cancellationToken);

        return entity.Id;
    }
}
```

Рис. 15 Приклад імплементації «команди» в CQRS [37]

Основні аспекти CQRS, які можна виділити в поданому коді:

1. **Команда CreateTodoItemCommand:**
    - Відповідає за внесення змін у систему, конкретно, за створення нової ролі проекту.

- Параметри команди (наприклад, ім'я ролі) вказують, які зміни потрібно внести.

2. **Обробник команди CreateTodoItemCommandHandler:**
    - Відповідає за реальне виконання команди створення ролі проекту.
    - Взаємодіє з додатковими сервісами, такими як база даних (представлено **IApplicationDbContext**), для збереження нової ролі.
    - Містить домену модель **TodoItem**, яка є сутністю ролі на проєкті.
3. **Повернення ідентифікатора (int) з обробника:**
    - Зазвичай ідентифікатор повертається після успішного виконання команди. Це може бути важливим для подальшого використання чи відстеження створеного об'єкта.
4. **Відсутність запитів в даному коді:**
    - Поданий код фокусується на командах та їх обробці, не включаючи запитів для отримання даних. Це може вказувати на розділення відповідальностей для змін та читань.

Модель запитів (Query Model):

1. **Операції читання:**
    - Запити (Queries): Це запити на отримання даних з системи, які не призводять до змін стану. Зазвичай ці запити використовуються для відображення даних користувачам або виконання різноманітних аналітичних операцій.
    - Оптимізація запитів: Модель запитів спрямована на оптимізацію шляху отримання даних і може використовувати різні засоби для

швидкого доступу до інформації, такі як матеріалізовані види або кешування.

2. **Швидкість і ефективність:**
    - Оптимізовані структури даних: Оскільки операції читання часто виконуються набагато частіше, ніж операції запису, модель запитів може використовувати оптимізовані структури даних, що дозволяє швидко отримувати необхідну інформацію.
3. **Денормалізація даних:**
    - Матеріалізовані види (Materialized Views): Щоб зменшити час відповіді на запити, дані можуть бути денормалізовані та збережені у вигляді матеріалізованих видів, які містять готові до використання результати запитів.

Відповідно структура **моделі команди** може складатись з:

1. **Запити (Queries):** Це структуровані об'єкти або класи, які містять дані для запиту на отримання інформації з системи. Кожен запит визначає, яку інформацію потрібно отримати і як її відфільтрувати.
2. **Обробники запитів (Query Handlers):** Це компоненти, які відповідають за виконання конкретних запитів. Обробники запитів отримують запити, взаємодіють з читальною моделлю та повертають результати.
3. **Читальна модель (Read Model):** Це представлення даних, оптимізоване для швидкого виконання операцій читання. Читальна модель може використовувати денормалізовані дані або матеріалізовані види для покращення продуктивності запитів.
4. **Механізми оптимізації запитів:** Це компоненти, які використовуються для оптимізації читальної моделі та прискорення виконання запитів. Це

може включати в себе кешування, використання індексів та інші методи оптимізації.

Розберемо структуру моделі запитів на основні функціоналу сторення нової моделі ролі на проєкті:

```
4 usages  Michael Tarnorutsky +1
public record GetTodoItemsWithPaginationQuery : IRequest<PaginatedList<TodoItemBriefDto>>
{
    2 usages
    public int ListId { get; init; }
    2 usages
    public int PageNumber { get; init; } = 1;
    2 usages
    public int PageSize { get; init; } = 10;
}

Ilker Aytı +2 *
public class GetTodoItemsWithPaginationQueryHandler : IRequestHandler<GetTodoItemsWithPaginationQuery,
                                                                      PaginatedList<TodoItemBriefDto>>
{
    private readonly IApplicationDbContext _context;
    private readonly IMapper _mapper;

    Ilker Aytı
    public GetTodoItemsWithPaginationQueryHandler(IApplicationDbContext context, IMapper mapper)
    {
        _context = context;
        _mapper = mapper;
    }

    Ilker Aytı +2 *
    public async Task<PaginatedList<TodoItemBriefDto>> Handle(GetTodoItemsWithPaginationQuery request,
                                                              CancellationToken cancellationToken)
    {
        return await _context.TodoItems // DbSet<TodoItem>
            .Where(x :TodoItem => x.ListId == request.ListId) // IQueryable<TodoItem>
            .OrderBy(x :TodoItem => x.Title) // IOrderedQueryable<TodoItem>
            .ProjectTo<TodoItemBriefDto>(_mapper.ConfigurationProvider) // IQueryable<TodoItemBriefDto>
            .PaginatedListAsync(request.PageNumber, request.PageSize); // Task<PaginatedList<...>>
    }
}
```

Рис. 16 – Приклад запиту в CQRS [37]

Розберемо структуру моделі команди на основні функціоналу сторення нової моделі ролі на проєкті:

1. **Запит (GetProjectRolesQuery):**
    - Це структурований об'єкт, що представляє дані для запиту на отримання інформації.
    - Визначає, яку інформацію потрібно отримати (список ролей проекту).
2. **Обробник запиту (GetProjectRolesQueryHandler):**
    - Це компонент, який відповідає за виконання конкретного запиту.
    - Отримує запит, взаємодіє з читальною моделлю (в даному випадку - базою даних через Entity Framework Core) та повертає результати.
3. **Читальна модель (Read Model):**
    - Читальна модель у цьому контексті представлена у базі даних (**_context.ProjectPositions**).
    - Використовується **AsNoTracking()** для отримання данних без відстеження змін, що може покращити продуктивність для операцій читання.
4. **Проекція (ProjectTo<ProjectRoleDto>(_mapper.ConfigurationProvider)):**
    - Використовується AutoMapper для проекції даних із сутності (**ProjectPosition**) у DTO (**ProjectRoleDto**).
5. **Сортування та отримання результатів (OrderBy(p => p.Name).ToListAsync(cancellationToken)):**
    - Додаткова обробка результатів, в даному випадку - сортування за іменем та отримання списку ролей у вигляді списку.

Ці компоненти взаємодіють між собою, створюючи комплексну систему, яка забезпечує ефективні операції запису та читання відповідно до принципів CQRS.

3.2 Реалізація Domain Layer

Рівень домейну (domain layer) буде містити основні сутності з якими буде працювати наша програма. Цей рівень повинен бути абсолютно незалежним та містити основу нашої бізнес логіки.

```
using System.ComponentModel.DataAnnotations;                                    ⚠ 2
using System.ComponentModel.DataAnnotations.Schema;
using TaskManagementSystem.Domain.Common;

namespace TaskManagementSystem.Domain.Entities
{
    5 usages  Oleh Yarosh *  2 exposing APIs
    public class Executor : BaseEntity
    {
        [Key]
        [ForeignKey( name: "Account")]
        public new string Id { get; set; }

        1 usage
        public Account? Account { get; set; }

        1 usage
        public ExecutorProfile? ExecutorProfile { get; set; }

        public ICollection<ExecutorWork> ExecutorWorks { get; set; } = new List<ExecutorWork>();
    }
}
```

Рис. 17 – Сутність Executor (Виконавець)

```
using TaskManagementSystem.Domain.Common;

namespace TaskManagementSystem.Domain.Entities
{
    8 usages  Oleh Yarosh *  2 exposing APIs
    public class ExecutorWork : BaseEntity
    {
        [Required]
        2 usages
        public string ExecutorId { get; set; }

        [Required]
        1 usage
        public int ProjectId { get; set; }

        [Required]
        2 usages
        public int ProjectPositionId { get; set; }

        2 usages
        public ProjectPosition? ProjectPosition { get; set; }

        2 usages
        public Project? Project { get; set; }

        2 usages
        public Executor? Executor { get; set; }

        2 usages
        public ICollection<ProjectTask> ProjectTasks { get; private set; }
    }
}
```

Рис. 18 – Сутність ExecutorWork (Робота виконавця)

BaseEntity - це абстрактна сутність, що містить в собі спільну логіку всіх сутностей. Всі інші сутності, які зберігаються в базі даних, будуть успадковувати від BaseEntity.

Сутність Executor успадковує властивості сутності BaseEntity. Її унікальний ідентифікатор представляє собою зовнішній ключ сутності Account. Для забезпечення відповідності типів ідентифікаторів (string для Account і int для BaseEntity), ми використовуємо механізм «приховання». Це означає, що тип ідентифікатора сутності Executor стає таким самим, як у сутності Account (тобто string), хоча в сутності BaseEntity він має тип int. Щоб забезпечити можливість поліморфізму і взаємодії з базою даних, всі сутності, що взаємодіють з базою даних, повинні успадковувати сутність BaseEntity.

У сутності Executor також знаходиться сутність ExecutorProfile і колекція сутностей ExecutorWork. Сутність ExecutorWork виконує роль сполучної ланки для різноманітних сутностей. Ця сутність обладнана унікальним ідентифікатором, успадкованим від сутності BaseEntity, оскільки вона слугує описом конкретного користувача на певному проєкті. Основною інформацією є перелік завдань, який представляє собою колекцію сутностей ProjectTask, які прив'язані до даного користувача та його позиції в рамках проєкту, а саме сутності ProjectPosition. Додатковими параметрами є посилання на сутності Executor та Project, які Entity Framework Core використовує для побудови зв'язків в базі даних.

По такому ж принципу були створенні всі сутності, які необхідні для функціонування програми та які будуть зберігатись в базі даних.

3.3 Реалізація Application Layer

В рівні застосування знаходиться вся основна бізнес-логіка програми та всі обчислення, які вкионуються. Він повинен бути незалежним від усіх рівнів, окрім домейну, адже він відповідає за взаємодії інших рівнів з рівнем домейну.

Задля забезпечення взаємодії, без впровадження залежностей між рівнями, були використанні так звані Data Transfer Object (DTO)[43], що перекладається як об’єкт для передачі даних. Він виконує функцію відповідно до своєї назви. Задля зручності трансформації таких об’єктів від сутності домейну до DTO та навпаки, ми використовуємо інструмент під назвою “AutoMapper”[44]. Це зручний інструмент, за допомогою якого ми гнучко можемо налаштовувати перетворення об’єктів з одного відповідно до заданих налаштувань.

```
10 usages  Oleh Yarosh *
public class ProjectDto
{
    public int Id { get; set; }

    1 usage
    public string Name { get; set; }

    1 usage
    public string Description { get; set; }

    1 usage
    public DateTime CreationDate { get; set; }

    public DateTime ClosingDate { get; set; }

    public bool IsClosed { get; set; }

    1 usage
    public ICollection<int> ProjectTasksIds { get; set; }

    1 usage
    public ICollection<int> ExecutorWorksIds { get; set; }
```

Рис. 19 – Приклад DTO проекту

```
public class AutomapperProfile : Profile
{
    1 usage  Oleh Yarosh *
    public AutomapperProfile()
    {
        CreateMap<ExecutorProfile, ExecutorProfileDto>().ReverseMap();

        CreateMap<ExecutorWork, ExecutorWorkDto>().ReverseMap();

        CreateMap<Project, ProjectDto>()
            .ForMember( destinationMember: pm :ProjectDto => pm.ExecutorWorksIds, memberOptions: l :IMer
                l.MapFrom( mapExpression: ew :Project => ew.ExecutorWorks.Select(w :ExecutorWork => v
            .ForMember( destinationMember: pm :ProjectDto => pm.ProjectTasksIds, memberOptions: i :IMem
                i.MapFrom( mapExpression: p :Project => p.ProjectTasks.Select(t :ProjectTask => t.Id)

        CreateMap<Executor, ExecutorDto>()
            .ForMember( destinationMember: em :ExecutorDto => em.Id, memberOptions: i :IMemberConfigurationI
                .MapFrom( mapExpression: m :Executor => m.Id))
            .ForMember( destinationMember: em :ExecutorDto => em.Email, memberOptions: i :IMemberConfigura
                .MapFrom( mapExpression: m :Executor => m.Account.Email))
            .ForMember( destinationMember: em :ExecutorDto => em.Name, memberOptions: n :IMemberConfigurati
                .MapFrom( mapExpression: e :Executor => e.ExecutorProfile.Name))
            .ForMember( destinationMember: em :ExecutorDto => em.Surname, memberOptions: n :IMemberConfig
                .MapFrom( mapExpression: e :Executor => e.ExecutorProfile.Surname))
            .ForMember( destinationMember: em :ExecutorDto => em.Phone, memberOptions: p :IMemberConfigura
                .MapFrom( mapExpression: m :Executor => m.ExecutorProfile.Phone)) // IMappingExpression<
            .ForMember( destinationMember: em :ExecutorDto => em.ExecutorWorksIds, memberOptions: l :IM
                l.MapFrom( mapExpression: ew :Executor => ew.ExecutorWorks.Select(w :ExecutorWork =>

        CreateMap<ProjectTask, ProjectTaskModel>()
```

Рис. 20 – Приклад конфігурації AutoMapper

Також на цьому рівні в нас повинен знаходитись інтерфейс використовуючи який, в рівні інфраструктури ми повинні будемо отримувати доступ до даних з нашої БД.

```
namespace TaskManagementSystem.Application.Common.Interfaces;

6 usages  Oleh Yarosh *
public interface ITaskManagementSystemDbContext
{
    1 usage  new *
    public DbSet<Executor> Employees { get; }

    1 usage  new *
    public DbSet<Project> Projects { get; }

    1 usage  new *
    public DbSet<ProjectTask> ProjectTasks { get; }

    1 usage  new *
    public DbSet<ExecutorWork> ExecutorWorks { get; }

    1 usage  new *
    public DbSet<ProjectTaskStatus> ProjectTaskStatuses { get; }

    3 usages  new *
    public DbSet<ProjectPosition> ProjectPositions { get; }

    2 usages  Oleh Yarosh
    Task<int> SaveChangesAsync(CancellationToken cancellationToken);
```

Рис. 21 – Приклад опису інтерфейсу використовуючи DbSet

Посередництво між рівнями відбувається за допомогою бібліотеки MediatR[45].

MediatR - це бібліотека для підтримки паттерну проектування "Посередник" (Mediator) в програмуванні. У контексті ASP.NET її зазвичай використовує для вирішення питань обробки запитів та операцій над даними. За допомогою цього інструменту ми можемо доволі легко застосувати принцип CQRS в нашій програмі.

Основними компонентами бібліотеки «MediatR» можна виділити[46]:

Запити (Requests):

Це класи, які представляють запити, що будуть виконуватися. Зазвичай це об'єкти, які містять необхідні дані для виконання операції. Такі класи імплементують інтерфейс “IRequest<ResultType>”, де ResultType є типом, яким ми бажаємо мати в результаті обробки запиту.

```
using ...

namespace TaskManagementSystem.Application.ProjectRole.Commands;

3 usages
public class CreateProjectRoleCommand : IRequest<int>
{
    public int Id { get; set; }
    1 usage
    public string Name { get; set; }
}

public class CreateProjectRoleCommandHandler : IRequestHandler<CreateProjectRoleCommand, int>
{
    private readonly ITaskManagementSystemDbContext _context;

    public CreateProjectRoleCommandHandler(ITaskManagementSystemDbContext context)
    {
        _context = context;
    }

    public async Task<int> Handle(CreateProjectRoleCommand request, CancellationToken cancellationToken)
    {
        var entity = new ProjectPosition
        {
            Name = request.Name
        };

        _context.ProjectPositions.Add(entity);

        await _context.SaveChangesAsync(cancellationToken);

        return entity.Id;
    }
}
```

Рис. 22 – Приклад створення команди та її обробника з використанням MediatR

Обробники (Handlers):

Це класи, які виконують реальну логіку обробки запитів. Вони використовуються для обробки конкретного типу запиту і повертають результат. Такі класни імплементують інтерфейс IRequestHandler<MyQuery/MyCommand, ResultType>, де MyQuery/MyCommand повинні бути класами в яких містяться дані, необхідні для виконання нашого запиту або команди відповідно, а ResultType визначає бажаний тип, який буде повертати операція після її виконання.

```
using AutoMapper;
using AutoMapper.QueryableExtensions;
using MediatR;
using Microsoft.EntityFrameworkCore;
using TaskManagementSystem.Application.Common.Interfaces;

namespace TaskManagementSystem.Application.ProjectRole.Queries;

3 usages  new *
public record GetProjectRolesQuery : IRequest<IList<ProjectRoleDto>>;

Kelariax *
public class GetProjectRolesQueryHandler : IRequestHandler<GetProjectRolesQuery, IList<ProjectRoleDto>>
{
    private readonly ITaskManagementSystemDbContext _context;
    private readonly IMapper _mapper;

    Kelariax *
    public GetProjectRolesQueryHandler(ITaskManagementSystemDbContext context, IMapper mapper)
    {
        _context = context;
        _mapper = mapper;
    }

    Kelariax *
    public async Task<IList<ProjectRoleDto>> Handle(GetProjectRolesQuery request, CancellationToken canc
    {
        return await _context.ProjectPositions.AsNoTracking() // IQueryable<ProjectPosition>
            .ProjectTo<ProjectRoleDto>(_mapper.ConfigurationProvider) // IQueryable<ProjectRoleDto>
            .OrderBy(p :ProjectRoleDto => p.Name) // IOrderedQueryable<ProjectRoleDto>
            .ToListAsync(cancellationToken); // Task<List<...>>
    }
}
```

Рис 23 – Приклад створення запиту та його обробника з використанням MediatR

3.4 Реалізація Infrastructure Layer

В рівні інфраструктури ми визначаємо основні зовнішні залежності системи, в нашому випадку бази даних.

Було обрано реляційну модель бази даних, яка ґрунтується на визначених зв'язках між наборами даних. У цій моделі кожен набір даних представлений у вигляді таблиці, де дані зберігаються у стовпцях і рядках. Кожен стовпець має найменування та тип даних, а рядки містять відповідні значення. В кожній таблиці один зі стовпців використовується як унікальний ідентифікатор, що є первинним ключем. Цей первинний ключ виступає як зовнішній ключ, забезпечуючи зв'язок між таблицями.

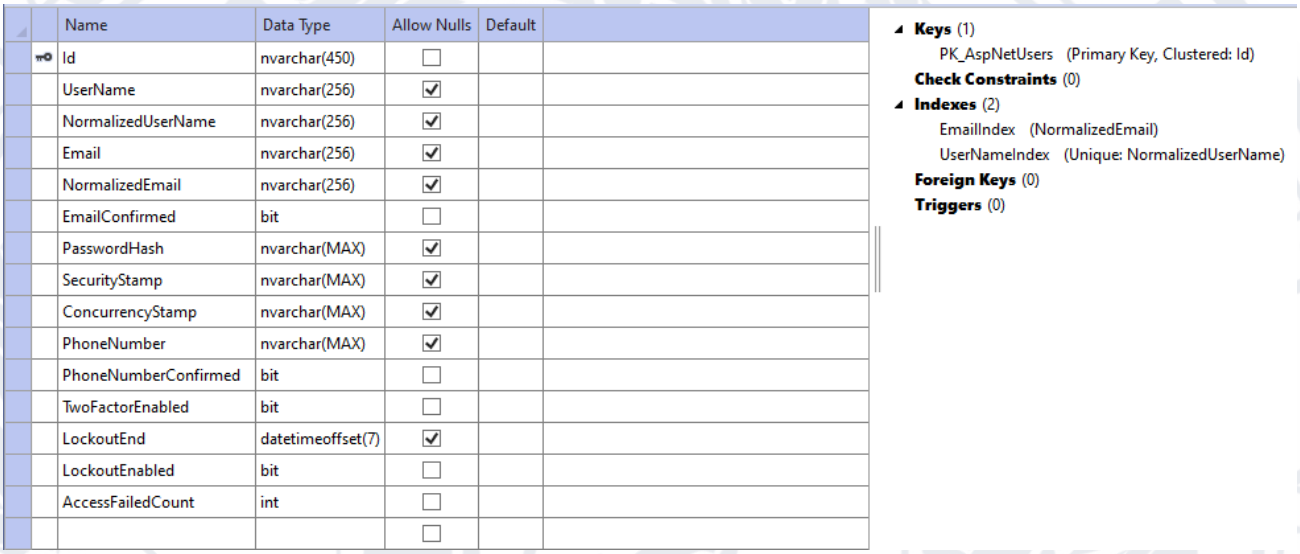
Розробка та створення бази даних виконані за принципом Code First[47] (Спершу код). Його суть полягає в тому, що ми формуємо моделі об'єктів, написуючи код у вигляді класів. Після цього ORM (Object Relational Mapping), зокрема Entity Framework Core[48], автоматично створює необхідні таблиці та налаштовує зв'язки в базі даних. Для використання цього підходу у додатку встановлюємо відповідні пакети за допомогою NuGet Package Manager:

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

Ці пакети дозволять вам використовувати Entity Framework Core для створення та взаємодії з базою даних, а також надають інструменти для розробки, такі як дизайнер контексту та міграції бази даних.

Опишемо детальніше цей процес на прикладі сутності Account:

Клас Account є нащадком класу IdentityUser із простору імен Microsoft.AspNetCore.Identity. Він автоматично успадковує всі необхідні дані для облікового запису, такі як електронна пошта, пароль і унікальний ідентифікатор користувача. Згідно з цим, Entity Framework Core автоматично створив таблицю для цього класу.



	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	☐	
			☐	

Рис. 24 – Структура таблиці сутності Account

Entity Framework Core генерує SQL для таблиць з класу відповідно до підходу Code First. Основний принцип - це використання атрибутів та конвенцій для визначення структури бази даних на основі класів об'єктів.

Основні принципи:

1. **Code First Convention:** Entity Framework Core використовує конвенції для визначення імен та типів стовпців в базі даних на основі властивостей класу. Наприклад, властивість з ім'ям "Id" може бути визначена як первинний ключ.

2. **Атрибути та анотації:** Розмітка коду атрибутами або анотаціями може явно вказувати EF Core, як взаємодіяти з базою даних. Наприклад, використання атрибута **[Key]** для визначення первинного ключа.
3. **Fluent API:** Entity Framework Core надає Fluent API, яка дозволяє вам детально налаштовувати модель даних в коді. Вона використовується для визначення відносин, обмежень та інших аспектів бази даних.
4. **Migrations:** Під час розробки з підходом Code First використовуються міграції, щоб зберегти та оновлювати схему бази даних відповідно до змін в коді класів.

Під час виклику методу **dbContext.Database.Migrate()**, EF Core аналізує модель даних та застосовує будь-які зміни до схеми бази даних, необхідні для відображення цієї моделі.

```
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. 25 – Скрипт створення таблиці на основі сутності Account

Також в рівні інфрасруктури знаходяться всі наобхідні визначенні нами конфігурації для зовнішніх залежностей систем.

Приклад конфігурації для бази даних:

```
using Microsoft.AspNetCore.Identity;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace TaskManagementSystem.Infrastructure.Data.Configurations
{
    1 usage  Oleh Yarosh *
    public class RoleConfiguration : IEntityTypeConfiguration<IdentityRole>
    {
        Oleh Yarosh *
        public void Configure(EntityTypeBuilder<IdentityRole> builder)
        {
            builder.HasData(
                new IdentityRole
                {
                    Name = "Admin",
                    NormalizedName = "ADMIN"
                },
                new IdentityRole
                {
                    Name = "Manager",
                    NormalizedName = "MANAGER"
                },
                new IdentityRole
                {
                    Name = "Executor",
                    NormalizedName = "EXECUTOR"
                }
            );
        }
    }
}
```

Рис. 26 – Конфігурація для початкової ініціалізації ролей для таблиці в БД

3.5 Реалізація Presentation Layer

Реалізувавши основну логіку програми на рівні застосування (application layer), все що нам залишається це імплементувати обробку клієнтських запитів до WebApi. На цьому рівні ми будемо просто приймати запити і передавати їх на виконання до рівня застосування, від якого він відповідно матиме залежність. Також тут впроваджуються залежності та задаються всі необхідні налаштування, такі як: дані для підключення бази даних, дані для обробки токенів на авторизацію і т.п.

```
Oleh Yarosh *
public void ConfigureServices(IServiceCollection services)
{
    services.AddApplicationServices();
    services.AddInfrastructureServices(Configuration);
    services.AddWebServices();


    services.AddControllers( configure: opt :MvcOptions =>
    {
        opt.Filters.Add<ErrorHandlingFilter>();
    });

    services.AddSwaggerGen(s :SwaggerGenOptions =>
    {
        s.AddSecurityDefinition( name: "bearerAuth", new OpenApiSecurityScheme
        {
            Name = "Authorization",
            Type = SecuritySchemeType.Http,
            Scheme = "bearer",
            BearerFormat = "JWT",
            In = ParameterLocation.Header,
            Description = "JWT Authorization header using the Bearer scheme."
        });
        s.AddSecurityRequirement(new OpenApiSecurityRequirement
        {
            {
                new OpenApiSecurityScheme
                {
                    Reference = new OpenApiReference
                    {
                        Type = ReferenceType.SecurityScheme,
                        Id = "bearerAuth"
                    }
                },
                new string[] { }
            }
        });
    });
}
```

Рис. 27 – впровадження залежностей через метод ConfigureServices

Обробка запитів в ASP.NET Core[49] будується на принципі конвейера, де дані запиту пройшовши через різні етапи обробки, подаються через послідовні компоненти, відомі як middleware. Ці middleware розташовані в конвейері, а кожен з них відповідає за певний аспект обробки запиту. Підключення middleware відбувається за допомогою методу Configure, який визначений у класі 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. 28 – Налаштування middleware за допомогою метода Configure

Клієнтський запит взаємодіє з додатком шляхом взаємодії з контролером, який має визначені точки доступу, відомі як ендпоінти. Ендпоінти представляють собою методи, через які відбувається пряме взаємодії з додатком. У відповідності з принципами REST існують різні типи HTTP-запитів:

- GET – запит на отримання даних

- POST – запит на введення даних або авторизацію
- PUT – запит на зміну існуючих даних
- PATCH – запит на часткову зміну існуючих даних
- DELETE – запит на видалення даних

Застосунок генерує відповідь на основі цих запитів і повертає її клієнту. Для обробки винятків був створений ExceptionFilter, який відповідає за обробку помилок (Exceptions) та повертає визначений результат в залежності від типу помилки.

```
[Route( template: "api/[controller]")]
[ApiController]
Oleh Yarosh +1 *
public class AuthController : ControllerBase
{
    private readonly IMediator _mediator;

    Oleh Yarosh +1
    public AuthController(IMediator mediator)
    {
        _mediator = Guard.Against.Null(mediator, nameof(mediator));
    }

    [HttpPost( template: "registration")]
    Oleh Yarosh +1
    public async Task<ActionResult<RegistrationResponse>> RegistrationAsync([FromBody] Registrat
    {
        return Ok(await _mediator.Send(request));
    }

    [HttpPost( template: "login")]
    Oleh Yarosh +1
    public async Task<ActionResult<LoginResponse>> LoginAsync([FromBody] LoginRequest request)
    {
        return Ok(await _mediator.Send(request));
    }
```

Рис. 29 – Приклад контролера

Посередник (Mediator):

Це сервіс, який використовується для відправлення запитів до відповідних обробників. Його ми будемо використовувати на рівні представляння для того, щоб викликати потрібний нам обробник запиту.

3.6 Використання Swagger

Swagger [50] – це інструмент, розроблений для документування REST API. Основна його функція полягає в тому, що він надає можливість взаємодії з описом API в режимі реального часу, дозволяючи переглядати специфікацію, відправляти запити та переглядати отримані результати в інтерактивному вигляді.

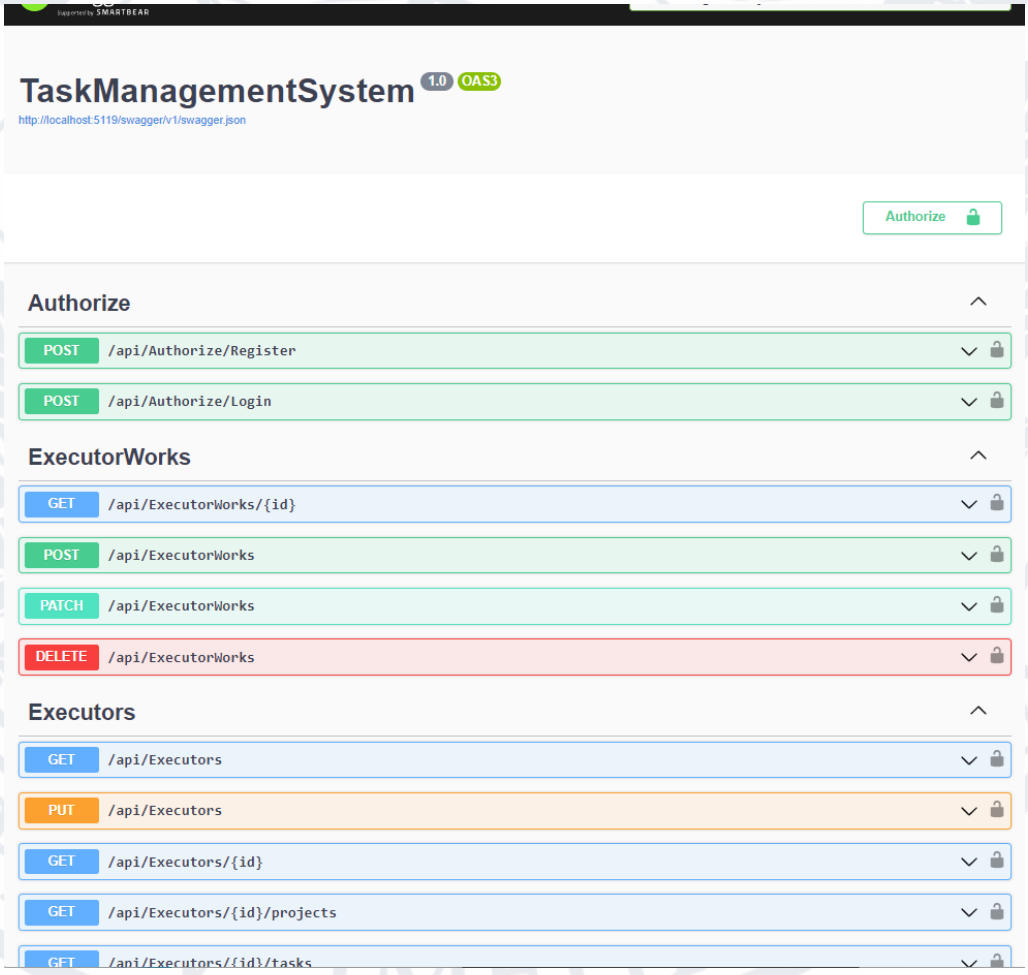


Рис. 30 – Приклад перегляду запитів з використанням Swagger

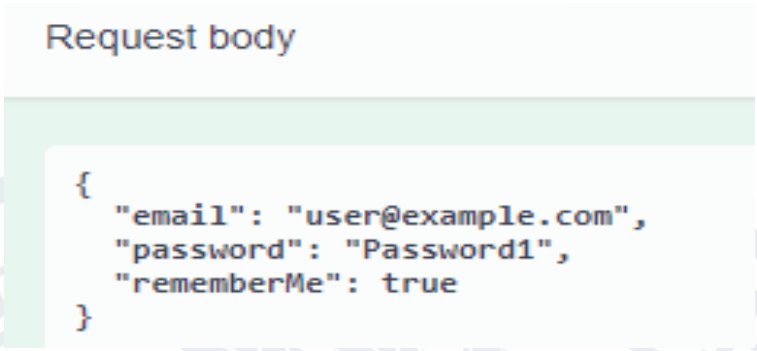


Рис. 31 – Тіло запиту на авторизацію

Рис. 32 – Результат запиту на авторизацію

3.7 Аналіз продуктивності виконавців

Задля аналізу продуктивності виконавця, була виведена формула на основі різноманітних факторів, які впливають на ефективність виконання завдань. Назвемо це коефіцієнтом продуктивності, скорочено КП.

Як вираховується КП:

$$КП = \frac{w_T \cdot T + w_Q \cdot Q + w_N \cdot N}{w_W \cdot W + w_C \cdot C}$$

Де:

1. **Час виконання завдань (T)**: Чим менше часу виконавець витрачає на виконання завдань, тим продуктивніше він працює.
2. **Якість виконання завдань (Q)**: Оцінка якості виконання завдань може бути внесена в формулу як важливий фактор, особливо якщо завдання включає етапи перевірки або оцінки якості.
3. **Кількість завдань (N)**: Цей параметр враховує обсяг роботи, яку виконавець здатен виконати за певний період часу.
4. **Співвідношення завдань до робочого часу (W)**: Це відношення обсягу роботи до часу, доступного для виконання завдань.
5. **Рівень складності завдань (C)**: Чим більші вимоги та складність завдань, тим менший коефіцієнт продуктивності може бути.
6. ***wT,wQ,wN,wW,wC*** - це преорітет для часу виконання, якості роботи, кількості завдань, робочого часу та рівня складності відповідно.

Ці преорітети можуть бути виражені як відсоткові частки, сума яких дорівнює 1, або як будь-які інші значення, які відображають важливість кожного фактору для вашої конкретної системи оцінки продуктивності.

Застосувавши цю формулу, отримуємо на виході дані, які можна використати для побудування графіків використовуючи зовнішній API, як наприклад google charts[51].

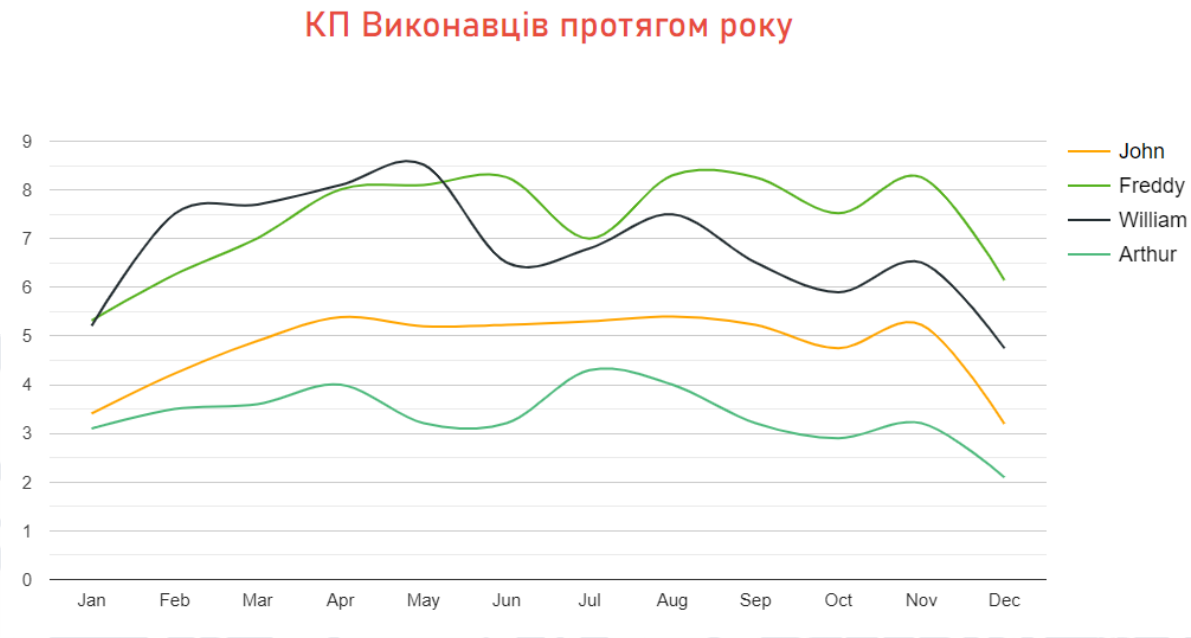


Рис. 33 – Графік побудований на основі вирахуваного КП

Також окрім вирахування КП, ми можемо отримувати інші дані про продуктивність виконавців:

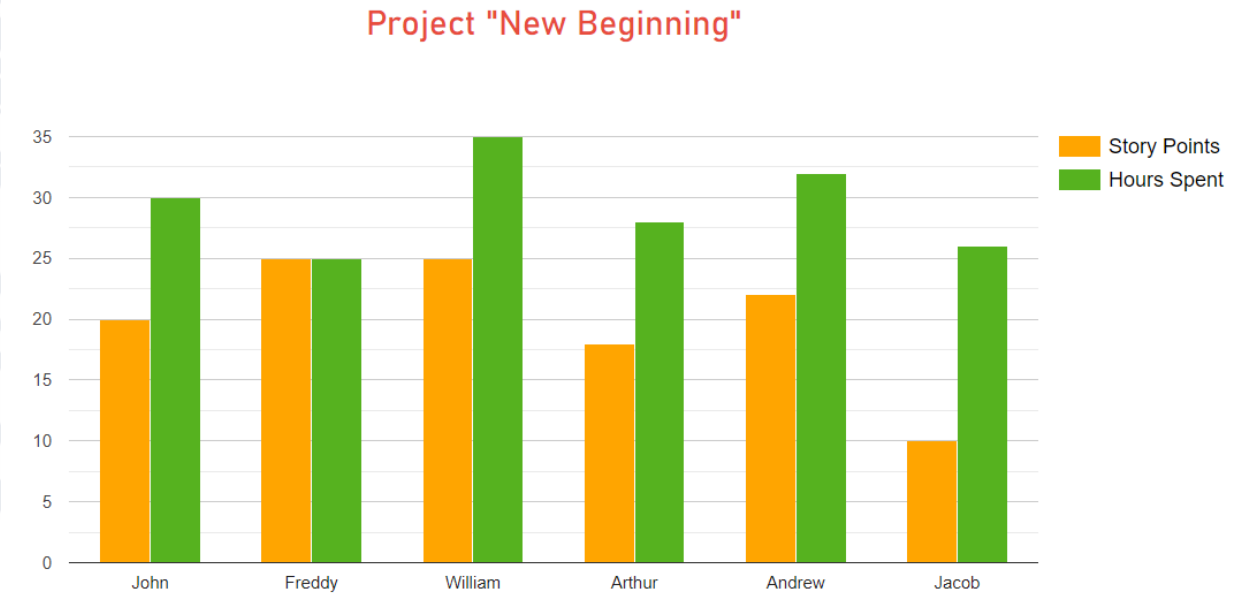


Рис. 34 – Порівння продуктивності виконавців на основі об’єму виконаних задач та затраченого часу

Висновок до розділу 3

Було оглянуто, проаналізовано та використано сучасні архітектурні принципи REST та «Чистої архітектури», а також підхід Command and Query Responsibility Segregation. Описано реалізацію кожного рівня чистої архітектури. Виведено і описано формулу коефіцієнту продуктивності виконавців, а також наведено приклади її використання. Також були поясненні основні принципи взаємодії з API.

ВИСНОВКИ

У рамках даної роботи було розглянуто актуальність систем розподілу задач та аналізу продуктивності виконавців, їх поширення в сьогодення.

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

Було створено та протестовано API відповідно до принципів REST. Реалізовано чисту архітектуру з використанням підходу Command and Query Responsibility Segregation. Набуті та вдосконаленні навички роботи з сучасними інструментами розробки Web Api та використання сучасних архітектурних підходів.

СПИСОК ЛІТЕРАТУРИ

1 – Як пандемія та війна вплинули на дистанційну роботу: аналіз ринку 2019-2022 [Електронний ресурс]. Режим доступу до ресурсу: https://www.work.ua/news/ukraine/2172/

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 – 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/

4 - What Is Task Management Software? [Електронний ресурс]. Режим доступу до ресурсу: https://peoplemanagingpeople.com/articles/what-is-task-management-software/

5 – Trello [Електронний ресурс]. Режим доступу до ресурсу: https://trello.com/

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

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

8 – Jira [Електронний ресурс]. Режим доступу до ресурсу: https://www.atlassian.com/software/jira

9 – "CLR via C# (4th Edition)" by Jeffrey Richter [Текст]

10 – "Pro C# 9 with .NET 5" by Andrew Troelsen and Philip Japikse [Текст]

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

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

13 – "C# 9 and .NET 5 – Modern Cross-Platform Development" by Mark J. Price [Текст]

14 – "Pro C# 7: With .NET and .NET Core" by Andrew Troelsen and Philip Japikse [Текст]

15 – "C# in Depth" by Jon Skeet [Текст]

16 – C# 9.0 in a Nutshell: The Definitive Reference" by Joseph Albahari and Ben Albahari [Текст]

18 – "C# Programming Yellow Book" by Rob Miles [Текст]

19 – "Head First C#" by Andrew Stellman and Jennifer Greene [Текст]

20 – "Programming C# 8.0" by Ian Griffiths [Текст]

21 – "C# 8.0 and .NET Core 3.0 – Modern Cross-Platform Development" by Mark J. Price

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

23 – "ASP.NET Core in Action" by Andrew Lock [Текст]

24 – "Programming ASP.NET Core" by Dino Esposito [Текст]

25 – "ASP.NET Core Application Development: Building an application in four sprints" by James Chambers, David Paquette, and Simon Timms [Текст]

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

27 – 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

28 – 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

29 – 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

30 – 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

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

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

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

34 – "RESTful Web APIs" by Leonard Richardson and Mike Amundsen [Текст]

35 – "RESTful API Design: Best Practices in API Design with REST" by Matthias Biehl [Текст]

36 – "Building RESTful Web Services with .NET Core" by Gaurav Aroraa, Ambily Jos, and Fanie Reynders [Текст]

37 – Clean Architecture [Електронний ресурс]. Режим доступу до ресурсу: https://github.com/jasontaylordev/CleanArchitecture

38 – "Clean Architecture: A Craftsman's Guide to Software Structure and Design" by Robert C. Martin [Текст]

39 – "Architecting Modern Web Applications with ASP.NET Core and Microsoft Azure" by Steve Smith and Shawn Wildermuth [Текст]

40 – Clean Architecture - An Introduction [Електронний ресурс]. Режим доступу до ресурсу: https://www.dandoescode.com/blog/clean-architecture-an-introduction

41 – "CQRS Journey" by Microsoft Patterns & Practices [Текст]

42 – "CQRS: The Example" by Mark Nijhof [Текст]

43 – Data Transfer Object DTO Definition and Usage [Електронний ресурс]. Режим доступу до ресурсу: https://www.okta.com/identity-101/dto/#:~:text=A%20data%20transfer%20object%20(DTO,without%20potentially%20exposing%20sensitive%20information.

44 – 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/

45 – MediatR NuGet [Електронний ресурс]. Режим доступу до ресурсу: https://www.nuget.org/packages/MediatR/

46 – MediatR — Beyond the basics [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/@cristian_lopes/mediatr-beyond-the-basics-8ab90841a732

47 – Entity Framework Code First: with Migrations [Електронний ресурс]. Режим доступу до ресурсу: https://medium.com/@josiahmahachi/entity-framework-code-first-with-migrations-8d1a197d2bd4

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

49 – ASP.NET Core Middleware [Електронний ресурс]. Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-8.0

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

51 – Google Charts [Електронний ресурс]. Режим доступу до ресурсу: https://developers.google.com/chart

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, _____

_____

Повністю вказується ПІБ та статус (посада для працівників, освітня (освітньо-наукова) програма – для здобувачів вищої освіти)

що нижче підписалась/підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)