МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

**ПОДОЛЯН МИКОЛА ОЛЕКСАНДРОВИЧ**

Допускається до захисту:
завідувач кафедри
інформаційних технологій,
д. т. н., доцент
_____ Т. В. Нескородєва
«_____» _____ 2022 р.

# Формування семантичної бази українських науковців на основі опрацювання їх публічних профілів

Спеціальність 122 «Комп'ютерні науки»
**Кваліфікаційна (магістерська) робота**

Науковий керівник:
Штовба С.Д., професор кафедри
інформаційних технологій,
д.т.н, професор
_____
(підпис)

Оцінка: _____/_____/_____
(бали за шкалою ЄКТС/за національною шкалою)

Голова ЕК: _____
(підпис)

Вінниця – 2022

# АНОТАЦІЯ

**Подолян М. О.**

**Формування семантичної бази українських науковців на основі опрацювання їх публічних профілів.** Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні технології обробки даних (Data Science)». Донецький національний університет імені Василя Стуса, Вінниця, 2022.

У кваліфікаційній (магістерській) роботі досліджено методи пошуку та обробки інформації розміщеної у відкритому доступі, в мережі інтернет, та розроблено прикладне програмне забезпечення для вирішення поставленої задачі. Робота містить 49 рисунків та 8 таблиць. Загальний обсяг роботи становить 81 сторінок.

Ключові слова: пошук та вилучення інформації, аналіз та зіставлення даних.

## ABSTRACT

**Podolyan M. O.**

**Creation of the semantic database of Ukrainian scientists based on the processing of their public profiles.** Specialty 122 "Computer science", educational program "Computer technologies of data processing (Data Science)". Vasyl Stus Donetsk National University, Vinnytsia, 2022.

In the qualification (master's) thesis, the methods of searching and processing information placed in open access, on the Internet, were investigated, and applied software was developed to solve the given problem. The work contains 49 figures and 8 tables. The total volume of work is 81 pages.

The relevance of the research topic is considered in the introduction. The goal and stages of the work are formulated, the object, subject and tasks of the research are defined.

Keywords: information search, information extraction, data analysis and comparison.

**ЗМІСТ**

# ВСТУП

Автоматизація присутня у всіх аспектах діяльності людини і може бути представлена у різних формах та застосована для вирішення різних типів задач, від найпростіших побутових до складних та багатоетапних процесів.

Актуальність теми наукової роботи визначається тим, що на даний момент, пошук та отримання специфічної, потрібної інформації з ресурсів у мережі інтернет є нагальним питанням, яке потребує вирішення, а інформаційна база знань є невід'ємною частиною будь-якого підприємства та продукту. Для сучасного бізнесу важливо мати можливість знаходити, виокремлювати, зберігати та оперувати інформацією, яка пов'язана з веденням діяльності певного суб'єкта. При цьому пошук та отримання необхідної інформації часто потребує великих вкладень ресурсів та часу. Задача автоматизації таких процесів стає все більш актуальною через збільшення об'ємів та варіантів форм представлення даних. Одним з можливих рішень є автоматизація процесу збору та обробки інформації з використанням інструментів автоматичної обробки ресурсів у мережі інтернет і збереження інформації, у потрібному форматі, у базу даних, робота з якою буде потребувати менше часу та вкладень.

Об'єктом дослідження в рамках магістерської атестаційної роботи є процес аналізу підходів до автоматизації збору та обробки інформації з відкритих джерел мережі інтернет.

Предметом дослідження є підходи до автоматизації роботи з браузерами та веб-сторінками, методи роботи з їх елементами, які використовуються у процесах автоматизації збору інформації, та що дозволять досягти поставленої мети з формування бази даних науковців у автоматичному режимі.

За мету роботи визначено розробити програмне рішення для формування бази даних науковців, використовуючи отримані знання.

Завданнями наукової роботи є:

1. Дослідити існуючі методи та підходи до автоматизації збору інформації
2. Визначити існуючі способи взаємодії з браузерами, веб-сторінками та їх елементами
3. Розглянути існуючі рішення-аналоги
4. Обрати технології та інструменти для реалізації поставленої мети
5. Розробити програмне рішення з використанням здобутих знань та обраних технологій

Науковою новизною дослідження, що проводиться, є використання технології автоматизації процесів тестування якості програмних продуктів, для досягнення поставленої мети з формування семантичної бази даних науковців. Також новизною є вдала комбінація використання багатопотокового підходу до виконання задач у межах асинхронних викликів до веб-додатку, що дозволяє прискорити процес автоматизованого збору інформації.

Практичним значення отриманого результату є потенційне значне скорочення матеріальних витрат, а також часу необхідних для отримання інформації.

У вступі розглянуто актуальність теми дослідження. Сформульована мета та етапи виконання роботи, визначено об’єкт, предмет та задачі дослідження.

У першому розділі розглянуто постановку задачі роботи, а також проведено аналіз вимог для досягнення результату відповідно до даної тематики.

У другому розділі описано обрані інструменти та технології, які використано під час розробки програмного продукту.

У третьому розділі йдеться про огляд створеного додатку, його особливостей та можливостей.

У висновку розглянуто результат виконаної роботи, а також проведено аналіз доцільності використання визначеного підходу до вирішення питання та створеного програмного продукту.

# РОЗДІЛ 1
# ВИЗНАЧЕННЯ ЦІЛІ, МЕТИ ТА ПОСТАНОВКА ЗАДАЧ, ОГЛЯД ІСНУЮЧИХ ПІДХОДІВ ТА РІШЕНЬ-АНАЛОГІВ

Ціллю роботи є створення програмного рішення для формування семантичної бази даних науковців, що буде наповнена на основі даних отриманих з відкритих джерел, зокрема профілів науковців у соціальних мережах.

Метою є створення повноцінного програмного продукту, функціонал якого дозволятиме збирати дані, а також аналізувати, зберігати та маніпулювати отриманою інформацією. Крім того, розроблений програмний продукт повинен мати можливість до легкого масштабування та зміни у майбутньому, а також бути відмовостійким та безпечним. Створене рішення повинне бути гнучким та зрозумілим, документованим та актуальним. Для досягнення поставленої мети, необхідно досягти виняткових результатів у низці поставлених задач.

Задачі:

1. Аналіз існуючих підходів та інструментів отримання, аналізу, обробки та збереження інформації;
2. Аналіз альтернативних способів досягнення поставленої мети та цілі;
3. Проектування архітектури бази даних;
4. Вибір технологій, інструментів, методів та підходів для створення програмного рішення;
5. Проектування архітектури програмного рішення;
6. Створення спроектованої бази даних, відповідно до поставлених вимог та цілей;
7. Реалізація спроектованого програмного рішення;
8. Створити опис проведеної наукової роботи та створення програмного рішення;

Програмний продукт повинен виконувати наступні функції та відповідати зазначеним вимогам:

1. Надавати можливість адміністрування, оновлення та робота з базою даних;
2. Реалізувати збір інформації з відкритих джерел мережі інтернет;
3. Проводити попередній аналіз та обробку знайденої інформації;
4. Реалізувати можливість збереження підготовленої інформації у спроектовану та створену базу даних;
5. Надавати доступ до існуючої у базі даних інформації, а також її модифікації, вилучення та отримання;
6. Реалізувати можливість взаємодії зі створеним програмним продуктом через виклики кінцевих створених кінцевих точок, або через наданий інтерфейс Swagger;
7. Мати можливість легкої та швидкої зміни підходу до розробки, структури або ж архітектури додатку, завдяки використанню абстракцій та низькій зв’язності компонентів створеного програмного рішення.

Безумовно, не все повинно бути автоматизовано, і більше того не все може бути автоматизовано, а в першу чергу автоматизації підлягають процеси, які містять багато рутинної та однотипної роботи. Процеси автоматизації та автоматизовані процеси уже давно стали частиною нашого життя, користуватись деякими з них ми уже звикли настільки, що не думаємо про те, як би це працювало без створених рішень автоматизації. Україна є чудовим прикладом держави, яка використовує автоматизацію процесів для цифровізації суспільства та державних установ, зокрема уже зараз велика кількість адміністративних послуг є автоматизованими і не потребують людського втручання. Для прикладу, отримання довідки про місце проживання або ж реєстрація шлюбу – уже є автоматизованими процесами, і здавалось би, що в цих автоматизаціях немає нічого незвичайного, але вони є

можливими лише завдяки цифровізації та автоматизації збору та збереження інформації.

Саме вдосконалення процесу автоматизації збору та збереження інформації, для отримання можливості швидкого доступу до проаналізованих та структурованих даних є ціллю, що поставлена у науковій роботі, адже наявність інформаційної бази знань є передумовою до створення будь якого програмного рішення.

Безумовно, для вирішення такої глобальної та актуальної проблеми уже існує багато рішень, то ж давайте розглянемо кілька з них.

Одним з лідерів серед рішень для збору інформації у мережі інтернет є Web Scraper - це універсальний простий у використанні застосунок для сканування довільних веб-сторінок і вилучення з них структурованих даних за допомогою кількох рядків коду JavaScript. Додаток завантажує веб-сторінки в браузер Chromium і відображає динамічний вміст. Web Scraper можна налаштувати та запустити вручну в інтерфейсі користувача або програмно за допомогою API. Витягнуті дані зберігаються в наборі даних, звідки їх можна експортувати в різні формати, наприклад JSON, XML або CSV. WebScraper є чудовим представником рішення для отримання інформації з веб-сторінок, проте його використання є обмеженим за рахунок недостатньої гнучкості у можливостях налаштування, а також незадовільних результатів часу виконання, оскільки додаток повертає всю доступну на сторінці інформацію, що може бути надзвичайно вибагливою до часу виконання операцією. При цьому, це рішення є хорошим вибором для автоматизації збору інформації з простих веб-сторінок, або ж інформації яка не потребує обробки чи аналізу.

Ще одним представником ринку є Google Search Results Scraper, на відміну від уже розглянутого WebScraper, не має опцій для аналізу та збору інформації з різних джерел мережі інтернет, а натомість пропонує функціонал для отримання результатів пошуку виконаного у пошуковому процесорі Google. Перевагами додатку є простота використання, відмовостійкість та швидкість виконання, при цьому основними недоліками є обмеженість сфер

використання та неможливість адаптації під конкретні задачі. Використання Google Search Results Scraper є ледь не єдиним способом отримання результатів пошуку у Google автоматизованих шляхом, оскільки починаючи з 2011 року Google не надає відкритих офіційних web API для використання їх сервісів.

Отже, основною проблемою існуючих рішень є їх обмеженість, а також частковість, що зумовлена прагненням створення уніфікованого програмного продукту, через це результати мають значно гірші можливості ніж ті, що вирішують одну з поставлених задач. Проаналізувавши представлені рішення, можна дійти висновку що в процесі автоматизації важливо не намагатись вирішити кілька задач одним рішенням, оскільки тоді жодна з них не принесе бажаного результату, зосередитись варто на одній задачі.

Для досягнення найкращого результату варто, за можливості, комбінувати кілька підходів для отримання інформації, безумовно надаючи перевагу отриманню інформації з використанням доступних кінцевих точок ресурсів та сервісів.

# РОЗДІЛ 2
# ОГЛЯД ТЕХНОЛОГІЙ ДЛЯ СТВОРЕННЯ ПРОГРАМНОГО ПРОДУКТУ

## 2.1 Огляд середовища розробки

Microsoft Visual Studio — лінійка продуктів компанії Microsoft, що включають інтегроване середовище розробки програмного забезпечення і ряд інших інструментальних засобів. Дані продукти дозволяють розробляти як консольні додатки, так і додатки з графічним інтерфейсом, в тому числі з підтримкою технології Windows Forms, а також веб-сайти, веб-додатки, веб-служби як в рідному, так і в керованому кодах для великої кількості платформ: Windows, Windows Mobile, Linux, MacOS, що робить Visual Studio чудовим вибором для між платформної розробки додатків різних рівнів.

Visual Studio включає в себе редактор вихідного коду з підтримкою технології IntelliSense і можливостями та інструментами для простого рефакторингу коду. Вбудований відладчик може працювати як відладчик рівня вихідного коду, так і відладчик машинного рівня. Решта вбудованих інструментів включають в себе редактор форм для спрощення створення графічного інтерфейсу додатку, веб-редактор, дизайнер класів і дизайнер схеми бази даних. Visual Studio дозволяє створювати і підключати сторонні додатки (розширення) для покращення функціональності практично на кожному рівні, включаючи додавання підтримки систем контролю версій вихідного коду (як, наприклад, Git), додавання нових наборів інструментів (наприклад, для редагування і візуального проектування коду на предметно-орієнтованих мовах програмування) або інструментів для інших аспектів процесу розробки програмного забезпечення (наприклад, клієнт Team Explorer та Git Changes для роботи з Git).

Visual Studio надає широкі можливості та велику кількість інструментів для розробки складних та масштабних додатків рівня, а також для злагодженої роботи великих команд. Однак, варто зауважити, що базова

безкоштовна версія Visual Studio Community Edition має обмежений фунціонал і не повинна використовуватись для розробки комерційних додатків та великими командами.

Visual Studio Community Edition – базова та безкоштовна версія IDE, що уже містить у собі усі необхідні інструменти та фунціонал для розробки додатків високої складності.

Visual Studio Professional Edition – професійне видання IDE, що включає розширені можливості для розробки та відладки коду а також взаємодії команди.

Visual Studio Enterprise Edition – видання, що відкриває найширші можливості відладки коду (.NET Memory Dump Analysis, Code Map Debugger Integration, IntelliTrace та ін.) та роботу у командах з необмеженою кількістю учасників.

Отже, у підсумку, перевагами і особливостями Visual Studio будуть наступні пункти:

a) середовище розробки підтримує роботу з багатьма мовами програмування, до яких відносяться найпопулярніші – C, C++, C#, F#, CLI, Visual Basic .NET, XML, XSLT, HTML, JavaScript, TypeScript та CSS.
b) Зручний та функціональний редактор коду, з підтримкою сучасного синтаксису;
c) IntelliSense – розумний помічник написання програмного коду.
d) Можливість розробки додатків для різних операційних систем, зокрема:
    a. Windows
    b. Linux
    c. MacOS
    d. Android
    e. IOS

а також веб додатків.

e) Можливість розробки додатків для різних платформ, зокрема:
    a. PC
    b. Mobile
    c. Xbox
    d. PlayStation
f) Можливість тестування роботи додатків; тестування продуктивності додатків, в будь-якій системі, відбувається в емуляторі без необхідності у встановленні додаткового програмного забезпечення;
g) рефакторинг готового коду;
h) попередня перевірка створеного додатку на наявність тих чи інших помилок в коді;
i) великий набір засобів інструментів для тестування кожного елемента додатку;
j) Велика спільнота та детальна документація, що постійно оновлюється та доповнюється;
k) Велика кількість готових шаблонів для додатків;

## 2.2 Огляд використаного фреймворку

Програмне рішення розроблене з використання фреймворку ASP.NET Core, який є нащадком все ще підтримуваного та використовуваного ASP.NET, на відміну від ASP.NET, Core версія підтримує кросплатформність і дає можливість розробляти додатки для різних операційних систем без необхідності внесення змін до кодової бази.

Фреймворк ASP.NET Core є інструментом для розробки веб-додатків на для платформи .Net, що в свою чергу дає низку переваг, зокрема вихідний код може бути написаний на різних, .Net сумісних, мовах програмування, і при цьому всі вони без проблем зможуть злагоджено працювати, наче всі частини додатку були розроблені з використанням однієї мови. Це стало можливим завдяки використанню загальномовного середовища Common Language Runtime, або ж CLR, скорочено, в якому код написаний на будь якій з .Net

сумісних мов представлено у вигляді коду написаного на мові проміжного рівня CIL (Common Intermediate Language), або ж MSIL (Microsoft Intermediate Language). Окрім вищезгаданої переваги, це також пришвидшує виконання програм, оскільки під час запуску програми необхідно значно менше часу для компіляції коду проміжного рівня на мові CIL у машинний код, ніж того б вимагала компіляція коду на мові високого рівня, зокрема C#. Таким чином, варто відзначити, що формат CIL значно краще підходить для компіляції у машинний код, оскільки конструкцій які використовуються уже є компільованими, і приведеними до вигляду значно ближчому до байт-коду ніж, для прикладу, будь яка мова високого рівня.

Саме через цю особливість отримав свою назву і компілятор, який виконує конвертацію коду написаного на CIL у машинний код, а саме JIT (Just in time), що дослівно має значення «миттєво».

На рисунку (див. Рисунок 3.1) продемонстровано взаємозв'язок середовища CLR, бібліотеки класів разом з відповідними додатками користувача і всією її системою. На рисунку також показано, як саме працює керований код у межах ширшої архітектури.

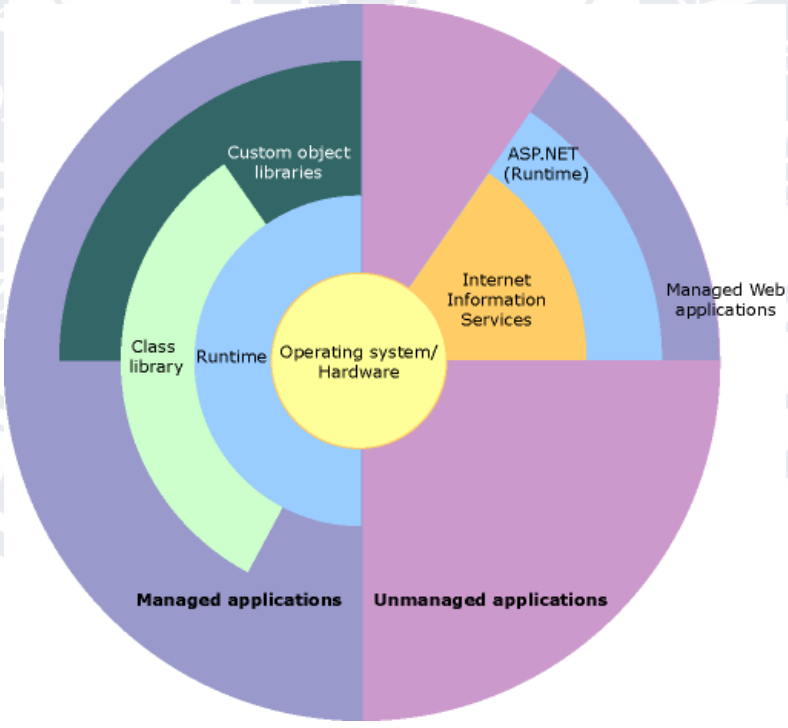



Рисунок 3.1 - Взаємозв'язок середовища CLR і бібліотеки класів

Повертаючись до кросплатформності, варто відмітити що остання версія .Net 7 підтримується більшістю версій та збірок операційних систем Windows, MacOs та Linux, що в свою чергу дозволяє розробляти програмні продукти і на обраній мові програмування C#, і вони будуть доступні для використання під різними операційними системами, зокрема на уже згаданих вище, а також на Tizen, Android, IOS та інших.

.Net рясніє кількістю бібліотек та інструментів, що дозволяє не просто обрати ту, що надає необхідний функціонал, але і вибрати серед різних варіантів ту, яка найбільше підходить для конкретного завдання, більше того, знову ж таки завдяки використанню загальномовного середовища, інструменти та бібліотека написані на одній з мов є доступними та повністю сумісними з будь якою іншою .Net сумісною мовою.

Під час розробки програмного рішення у рамках цієї наукової роботи, також було використано низку відомих та потужних інструментів доступних у .Net, зокрема EntityFramework – ORM система для взаємодії з базою даних та Dependency Injection, що використовується для ін’єкції залежностей.

На момент написання наукової роботи остання доступна версія .Net це сьома, проте в даній роботі буде використовуватися попередня, 6 версія, адже наразі вона є  найстабільнішою з останній випущених версій, разом з тим використано і передостанню, десяту, версію мови програмування C#.

## 2.3 Опис обраної мови програмування

C# є сучасною, об’єктно-орієнтованою та типо-безпечною мовою програмування, мова надає можливість розробникам створювати різноманітні типи безпечних та водночас надійних програм, які працюють у середовищі .NET. C# починав свій розвиток з мови C і є простим для розуміння програмістам на C, C++ та Java.

C# —компонентно-орієнтована мова програмування, яка пропонує набір конструкцій для підтримки більшості базових концепцій розробки програм, що робить C# мовою природнього походження для створення та використання

програмних компонентів. C# є об’єктно-орієнтованою мовою, суворої типізації, це значить, що ви самі визначаєте типи даних та їх подальшу поведінку. З часу появи, C# отримав досить велику кількість функцій для підтримки нових робочих навантажень і найновіших практик проектування програмного забезпечення.

Основні функції C# допоможуть розробнику створити надійні та довговічні програми. Наприклад, збирач сміття автоматично звільняє пам’ять, зайняту недоступними або невикористовуваними об’єктами. Обробка винятків надає структурований та розширений підхід до виявлення та виправлення знайдених помилок. Лямбда-вирази, або ж анонімні функції, забезпечують підтримку методів функціонального програмування, і навіть більше того, останні версії мови додали підтримку функторів та загальним методів. Синтаксис Language Integrated Query (LINQ) може створити базовий шаблон для роботи з даними з будь-якого джерела. Підтримка асинхронних операцій забезпечує синтаксисом, що потрібен для побудови розподілених систем, якою і є створене програмне рішення. Варто зазначити, C# використовує уніфіковану систему типів. Усі типи C#, включаючи примітивні типи, такі як int і double, унаслідуються від кореневого типу object. Усі типи мають загальний набір базових операцій. Крім того, C# підтримує визначені користувачем типи посилань та типи значень, що дає можливість в повній мірі реалізувати об’єктно-орієнтований підхід у створенні програмного рішення. Мова дозволяє динамічним чином розміщувати об’єкти та зберігати структуру даних у пам’яті, що, знову ж таки, покращує швидкість виконання коду та зменшує кількість використовуваної пам’яті. C# підтримує загальні методи та типи, що здатні забезпечити підвищену безпеку та високу продуктивність, а також реалізує ітератори, що в свою чергу дозволяють розробникам класів та колекцій визначати користувальницьку поведінку для масивів даних.

C# активно використовує версійність, щоб програми та бібліотеки могли розвиватися і це не призводило до порушення користувацьких контрактів. Аспекти дизайну C#, що змінилися з часом, щодо керування версіями,

доповнюються окремими віртуальними модифікаторами та модифікаторами перевизначення, правилами вирішення перевантаження методів і підтримкою декларацій елементів інтерфейсу.

Синтаксис мови є близьким до мов C++ та Java, та все ж дещо відрізняється.

Під час роботи над створенням C#, однією з цілей була відповідність до стандарту ECMA-334, яка наразі повною мірою не реалізується жодною з існуючих мов програмування.

### 2.4 Інструмент взаємодії з користувацьким інтерфейсом Selenium

Selenium — це інструмент тестування веб-автоматизації з відкритим кодом, який підтримує кілька браузерів і операційних систем. Це дозволяє тестувальникам використовувати кілька мов програмування, таких як Java, C#, Python, .Net, Ruby, PHP і Perl для кодування автоматизованих тестів.

За допомогою Selenium ми маємо можливість працювати з веб-сторінками імітуючи дії реального користувача, отримувати доступ до елементів та блоків сторінок використовуючи кілька можливих опцій доступу, зокрема використовуючи:

1. Унікальний ідентифікатор HTML елемента
2. LinkText HTML елемента
3. Назву HTML елемента
4. Абсолютний XPath HTML елемента
5. Відносний XPath HTML елемента
6. Набір класів HTML елемента
7. Частковий LinkText HTML елемента
8. TagName HTML елемента
9. CssSelector HTML елемента

Selenium надає інструменти для пошуку, доступу, та взаємодії з DOM елементами веб-сторінки, для прикладу, використовуючи Selenium ми маємо можливість отримати доступ до необхідної кнопки та натиснути на неї, або ж обійти обмеження визначене для поля.

Selenium використовує технологію WebDriver, яка підтримує більшість модерних браузерів, саме використання WebDriver надає можливість взаємодії з веб-сторінками, оскільки тільки отримавши доступ до сторінки – у нас буде можливість отримати доступ до її елементів.

На додачу, ми маємо можливість виконувати підготовлений JavaScript код для взаємодії з html сторінкою, що стане у пригоді для вирішення найскладніших ситуацій.

### 2.5 Система управління базами даних

Для проектування та роботи з базою даних у даній роботі було обрано систему керування базами даних PostgreSQL , адже вона є надає всі необхідні можливості для реалізації поставлених вимог, а також має чудову інтеграцію з використовуваним EntityFramework, що дозволяє здійснити інтеграцію без додаткових налаштувань. Вона є чудовою альтернативою комерційним СКБД, наприклад: Oracle Database, Microsoft SQL Server, IBM DB2, та СКБД з відкритим кодом: Firebird , MySQL, SQLite.

Також, варто відмітити, що на відміну від інших проектів з загальнодоступним вихідним кодом, зокрема MySql та Apache, PostgreSql справді розробляється колективно, ентузіастами, які хочуть працювати з системою керування базами даних, яка матиме всі найновіші і необхідні можливості, і це є можливим тому що PostgreSql не контролюється жодною компанією, а всі етапи розробки та впровадження нового функціоналу виконуються виключно реальними користувачами, які проявляють ініціативу та впроваджують найновіші підходи.

На додачу PostgreSql дає доступ до кількох унікальних, не притаманних жодній іншій системі, особливостей, зокрема власний тип даних JsonB, та можливості роботи з ним, який дозволяє зберігати дані у форматі Json.

Для розробки серверу PostgreSQL використано мову C, що дає беззаперечні переваги у швидкодії та функціоналі, проте значно сповільнює та ускладнює підтримку існуючого та розробку нового функціоналу, на додачу

вихідний код серверу може бути знайдений у відкритому доступі, і зазвичай має представлення текстових файлів, що містять початковий код серверу.

Доступ до обширного набору вбудованих функцій, сильно спрощує процес розробки додатків і створення баз даних, а адміністраторам дозволяє зберегти цілісність даних, а також керувати ними, незалежно від того, який об’єм даних, що оброблюються. PostgreSQL є безкоштовним і має у відкритому доступі вихідний код, також він має можливості до масштабування та досить легкий для розширення. Наприклад, ви маєте можливість створювати свої унікальні типи, функції.

У випадку, якщо зміни не мають під собою невдалих архітектурних рішень, і не суперечать базовому функціоналу, PostgreSQL прагне реалізувати функціонал відповідно до стандарту SQL. Більша частина функцій, які є обов’язковими для відповідності стандарту уже реалізовані, хоча деякі з них мають відмінний синтаксис. Скоріше за все, з часом, розроблена система буде близька до ідеалу та відповідатиме стандарту. Від п’ятнадцятої версії випущеної у жовтні 2022 року, PostgreSQL відповідає SQL:2016 Core реалізуючи 170 із 179 обов’язкових функцій. Зараз жодна реляційна база даних так і не відповідає цьому стандарту повністю.

## 2.5 Висновок

У цьому розділі розглянуто характеристики інструментів та технологій, що були використані для розробки веб орієнтованого додатку, а саме:

1. Середовище розробки VisualStudio
2. Платформа .Net.
3. Мова програмування C#.
4. Інструмент взаємодії з користувацьким інтерфейсом Selenium.
5. Система для управління базами даних PostgreSQL.

У наступному розділі буде розглянуто сам процес розробки, особливості та можливості створеного програмного рішення.

# РОЗДІЛ 3
# ОГЛЯД РОЗРОБЛЕНОГО ПІДХОДУ ДЛЯ ЗБОРУ ТА АГРЕГАЦІЇ ІНФОРМАЦІЇ ТА СТВОРЕННЯ ПРОГРАМНОГО ПРОДУКТУ

Під час виконання наукової роботи, та як її результат, було розроблено програмне рішення для збору, агрегації та збереження інформації про науковців, отриманої з відкритих джерел в мережі інтернет. Створене програмне рішення не є кінцевим та представляє собою концепцію системи, яка може бути розширена та модифікована. Зокрема функціонал може бути розширено завдяки інтеграції процесу збору даних з додаткових джерел, що дозволить збільшити об'єм та наповненість інформації про науковця. Розроблена частина є програмним продуктом у якому реалізовано найбільш цінний та потрібний функціонал.

## 3.1 Опис підходу до збору та агрегації інформації

Першопричиною у виборі підходу стала відсутність відкритих кінцевих точок для отримання доступу до інформації а також різний формат представлення даних у джерелах.

Ідея методу полягає в тому, що незалежно від способу збереження та маніпуляції даними у системах відкритих джерел мережі інтернет, в кінцевому варіанти всі вони представлені у вигляді HTML, що дає можливість виділити загальний підхід для збору інформації, який працюватиме незалежно від внутрішньої реалізації джерела інформації.

Таким підходом було обрано обробку HTML сторінок за допомогою бібліотеки, з відкритим вихідним кодом, для автоматизації тестування веб додатків – Selenium.

Selenium дозволяє взаємодіяти з даними розміщеними у веб-ресурсах імітуючи дії реального користувача, що відкриває можливості для розробки універсального підходу до збору та агрегації різнотипної інформації і при

розроблений додаток абстраговано від деталей реалізації оброблюваних веб-додатків.

Оскільки жоден з розглянутих веб-ресурсів не містить необхідної інформації у повно обсязі, розроблений додаток проводить агрегацію інформацію з різних джерел та зберігає отримані дані у спроектованій реляційній базі даних.

## 3.2 Опис спроектованої бази даних

### 3.2.1 Загальні відомості про обрану СУБД

Для проектування та роботи з базою даних було обрано PostgreSQL - це об'єктно-реляційна система керування базами даних (СКБД). Є альтернативою як комерційним СКБД (Oracle Database, Microsoft SQL Server, IBM DB2 та інші), так і СКБД з відкритим кодом (MySQL, Firebird, SQLite).

Порівняно з іншими проектами з відкритим кодом, такими як Apache, FreeBSD або MySQL, PostgreSQL не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення.

Сервер PostgreSQL написаний на мові C. Зазвичай розповсюджується у вигляді набору текстових файлів із вихідним кодом.

PostgreSQL має багато функцій, які допомагають розробникам створювати додатки, адміністраторам захищати цілісність даних і створювати відмовостійке середовище, а також допомагають вам керувати своїми даними, незалежно від того, наскільки великий чи малий набір обсяг даних. Крім того, що PostgreSQL є безкоштовним і поширюється у вигляді відкритого вихідного коду, він дуже розширюваний та має можливості масштабування. Наприклад, ви можете визначати власні типи даних, створювати власні функції, навіть писати код з різних мов програмування без перекомпіляції бази даних!

PostgreSQL намагається відповідати стандарту SQL, якщо така відповідність не суперечить традиційним функціям та не призводить до невдалих архітектурних рішень. Багато функцій, необхідних стандарту SQL, підтримуються, хоча іноді з дещо відмінним синтаксисом або функціями. З часом можна очікувати подальших кроків у напрямку відповідності. Починаючи з випуску версії 15 у жовтні 2022 року, PostgreSQL відповідає принаймні 170 із 179 обов'язкових функцій для відповідності SQL:2016 Core. На даний момент жодна реляційна база даних не відповідає цьому стандарту цілком.

### 3.2.2 Вимоги до розроблюваної бази даних

Створена база даних повинна відповідати вимогам до типу та змісту даних що будуть зберігатись.

За мету роботи поставлено збір та збереження даних науковців, тож є необхідним щоб спроектована база даних могла в повному обсязі зберігати та маніпулювати з такими набором даних.

Дані:

1. Науковець:
    a. Унікальний ідентифікатор запису
    b. Ім'я науковця
    c. Рейтинг науковця
    d. Організації, до яких науковець належить
2. Організація:
    a. Унікальний ідентифікатор запису
3. Наукова робота
    a. Унікальний ідентифікатор запису
    b. Назва
    c. Рік

4. Наукова робота науковця
    a. Унікальний ідентифікатор запису
    b. Унікальний ідентифікатор науковця
    c. Унікальний ідентифікатор наукової роботи
5. Концепт
    a. Унікальний ідентифікатор запису
    b. Назва
6. Соціальна мережа науковця
    a. Унікальний ідентифікатор запису
    b. Посилання
    c. Унікальний ідентифікатор науковця
    d. Унікальний ідентифікатор науковця у соціальній мережі
    e. Тип соціальної мережі
7. Напрям або галузь наукових робіт
    a. Унікальний ідентифікатор запису
    b. Австралійсько-новозеландський код напряму наукових робіт
    c. Назва
    d. Напрям до якого відноситься
8. Напрям або галузь наукових робіт науковця
    a. Унікальний ідентифікатор запису
    b. Унікальний ідентифікатор науковця
    c. Унікальний ідентифікатор напряму чи галузі наукових робіт

### 3.2.3 Спроектована база даних

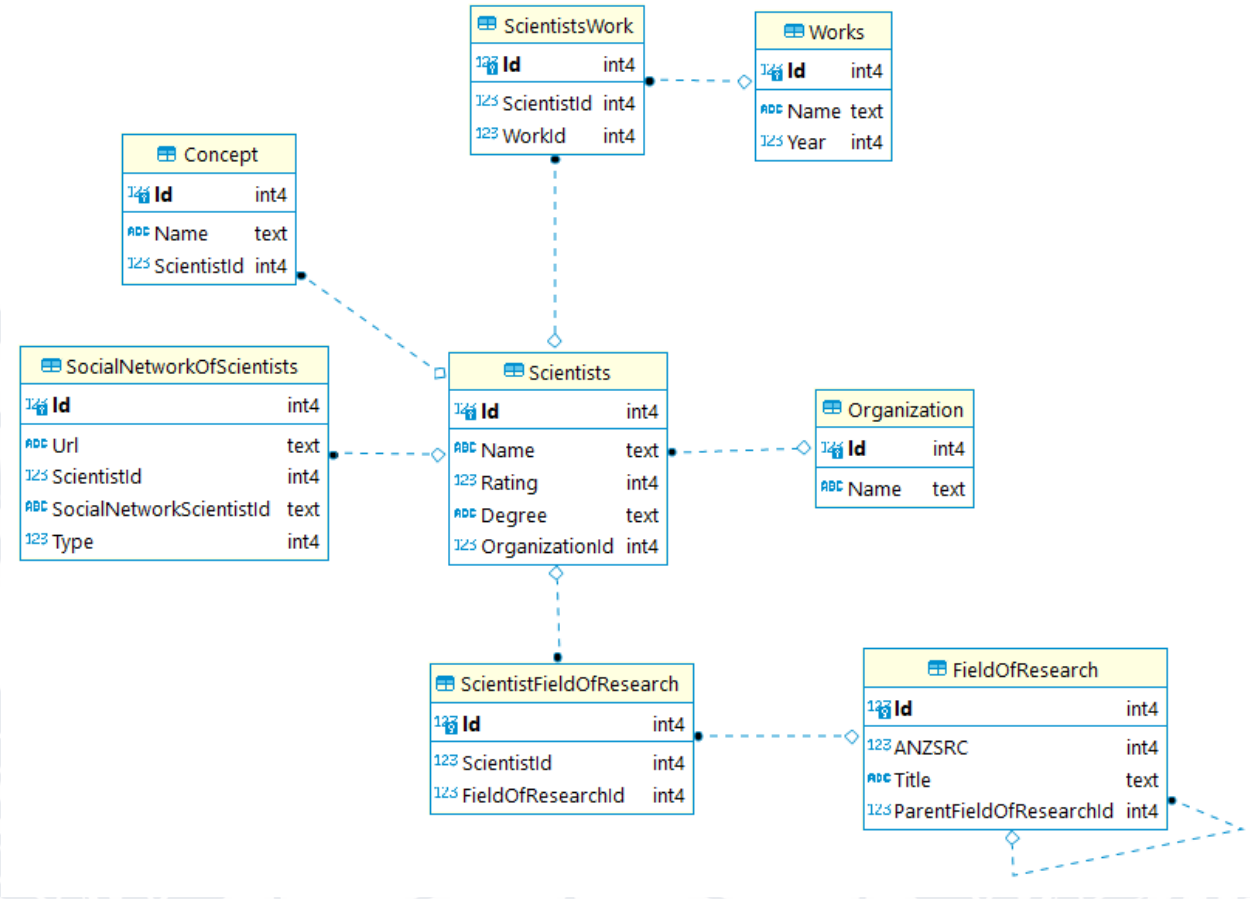



Рисунок 3.0 – Схема спроектованої бази даних

Опис зв’язків між сутностями бази даних:

1. Науковець знає свою організацію через ключ зовнішнього зв’язку OrganizationId, що є унікальним ідентифікатором організації.
2. Соціальна мережа знає науковця, до якого відноситься, через ключ зовнішнього зв’язку ScientistId, що є унікальним ідентифікатором науковця.
3. Концепт знає науковця, до якого відноситься, через ключ зовнішнього зв’язку ScientistId, що є унікальним ідентифікатором науковця.
4. Науковець знає свої напрями/галузі наукових робіт через таблицю посередника ScientistFieldOfResearch
5. Напрям/галузь наукових робіт знає науковця, до якого відноситься, через ключ зовнішнього зв’язку ScientistId, що є унікальним ідентифікатором науковця.

6. Напрям/галузь наукових робіт науковця знає напрям/галузь наукових робіт, до якого відноситься, через ключ зовнішнього зв’язку FieldOfResearchId, що є унікальним ідентифікатором напряму/галузі наукових робіт.
7. Науковець знає свої роботи через таблицю посередника ScientistWork
8. Робота науковця знає науковця, до якого відноситься, через ключ зовнішнього зв’язку ScientistId, що є унікальним ідентифікатором науковця.
9. Робота науковця знає роботу, до якої відноситься, через ключ зовнішнього зв’язку WorkId, що є унікальним ідентифікатором роботи (розробки/наукової публікації та ін.).

Створення таблиць посередників ScientistWork та ScientistFieldOfResearch зумовлена необхідністю створення зв’язку багато до багатьох, для уникнення повторюваності інформації у базі даних:

1. Над однією роботою можуть працювати декілька науковців
2. Необмежена кількість науковців можуть вести дослідження у одній і тій же ж галузі наукових робіт.

Кожній сутності присвоєно унікальний ідентифікатор запису, який генерується базою даних на етапі створення запису.

Таблиця Scientist є ключовою сутністю бази даних, в ній зберігаються основні дані науковців, такі як ім’я, рейтинг, наукова ступінь та організація. Колонка Scientist.Id використовується як унікальний ідентифікатор запису науковця і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.1 структура таблиці Scientist

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| Name | text | Name of the scientist | Required |
| Rating | Int | HRating of the scientist | |
| Degree | text | Degree of the scientist | |
| OrganizationId | Int | Scientist organization | Foreign key to the Organization table |

Таблиця Organization використовується для збереження інформації про організацію науковця, крім ідентифікатора в нії створено єдину колонку – назва організації.

Колонка Organization.Id використовується як унікальний ідентифікатор запису і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.2 структура таблиці Organization

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| Name | text | Name of the organization | Required |

Таблиця FieldOfResearch використовується для збереження інформації про напрям наукової діяльності відповідно до австралійсько-новозеландської класифікації наукової діяльності.

Крім ідентифікатора в нії створено колонки:

1. Title – повна назва напряму роботи
2. ANZRC – код класифікації напряму наукової діяльності
3. ParentFieldOfResearchId – зовнішній ключ до поля FieldOfReserach.Id який заповнюється, коли напрям наукової діяльності є дочірнім.

Колонка FieldOfReserach.Id використовується як унікальний ідентифікатор запису і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.3 структура таблиці FieldOfResearch

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| Title | text | Title of the field of research | Required |
| ANZSRC | Int | Australian and New Zealand Standard Research Classification (ANZSRC) | |
| ParentFieldOfResearchId | int | Used to create a relation between sub field of research and parent field of research | Nullable |

Таблиця ScientistFieldOfResearch використовується реалізації зв’язку багато до багатьох між таблицями Scientist та FieldOfResearch.

Крім ідентифікатора в нії створено колонки:

1. ScientistId – зовнішній ключ до Scientist.Id
2. FieldOfResearchId – зовнішній ключ до FieldOfResearch.Id

Колонка ScientistFieldOfResearch.Id використовується як унікальний ідентифікатор запису і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.4 структура таблиці ScientistFieldOfResearch

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| ScientistId | Int | Id of the scientist | Required, foreign key to Scientist.Id |
| FieldOfResearchId | Int | Id of the FieldOfResearch | Required, foreign key to the FieldOfResearch.Id |

Таблиця Concept використовується для збереження інформації про концепції роботи науковця, крім ідентифікатора в нії створено колонки:

1. Name – назва
2. ScientistId – зовнішній ключ до Scientist.Id

Колонка Organization.Id використовується як унікальний ідентифікатор запису і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.5 структура таблиці Concept

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| Name | text | Concept title | Required |
| ScientistId | Int | Id of the scientist | Required, foreign key to Scientist.Id |

Таблиця SocialNetworkOfScientist використовується для збереження інформації про соціальні мережі науковця, крім ідентифікатора в нії створено колонки:

1. Url – посилання на аккаунт науковця в соціальній мережі
2. ScientistId – зовнішній ключ до Scientist.Id
3. SocialNetworkScientistId – унікальний ідентифікатор науковця у системі соціальної мережі
4. Type – тип соціальної мережі (Google Scholar/WOS/Scopus/Orcid)

Колонка SocialNetworkOfScientist.Id використовується як унікальний ідентифікатор запису.

Таблиця 3.6 структура таблиці SocialNetworkOfScientist

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| Url | text | URL of the social network | Required |
| ScientistId | Int | Id of the scientist | Required, foreign key to Scientist.Id |
| SocialNetworkScientistId | text | Unique identifier of the scientist in the social network | |
| Type | Int | Social network type | Required |

Таблиця Work використовується для збереження інформації про роботи науковця (наукові дослідження, винаходи, видана література), крім ідентифікатора в нії створено колонки:

1. Name – назва роботи
2. Year – рік створення/публікації роботи

Колонка Work.Id використовується як унікальний ідентифікатор запису і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.7 структура таблиці Work

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Record unique identifier | Unique, required |
| Name | text | Title of the work | Required |
| Year | Int | Year of the creation | |

Таблиця ScientistWork використовується реалізації зв’язку багато до багатьох між таблицями Scientist та Work.

Крім ідентифікатора в нії створено колонки:

1. ScientistId – зовнішній ключ до Scientist.Id
2. WorkId – зовнішній ключ до Work.Id

Колонка ScientistWork.Id використовується як унікальний ідентифікатор запису і слугує для побудови зв’язків з іншими таблицями.

Таблиця 3.8 структура таблиці ScientistWork

| Column title | Column type | Column meaning | Constraints |
|---|---|---|---|
| Id | Int | Unique identifier | Unique, required |
| ScientistId | Int | Id of the Scientist | Required, foreign key to Scientist.Id |
| WorkId | Int | Id of the Work | Required, foreign key to Work.Id |

## 3.3 Опис розробленої програми

### 3.3.1 Загальні відомості про розроблений додаток

Розроблене ПЗ є WebApi додатком розробленим на мові програмування C# на платформі .Net. Взаємодія з додатком відбувається з використанням Swagger. В якості джерел обрано:

1. http://nbuviap.gov.ua - Бібліометрика української науки від Google Scholar і Scopus - джерело отримання загальних даних про науковців, першоджерело інформації, початкова точка для збору інформації (рисунок 3.1)

   Інформація, що збирається:

   a) Ім’я науковця
   b) H-rating науковця
   c) Установа науковця
   d) Галузь науки
   e) Соціальні мережі науковця (Scholar, Scopus, Orcid, WOS)

Знайдено 56656: google - 56251, scopus - 12300, WoS - 2455

| Рейтинг | № з/п | П. І. Б. | h-index | | | Галузь науки<br>Рубрика Google Scholar | Установа |
|---|---|---|---|---|---|---|---|
| | | | Google Scholar | Scopus | WoS | | |
| 1 | 1 | Єрьоменко Едуард Анатолійович | 182 | - | - | Педагогіка<br>Physical Education & Sports Medicine | Університет Державної фіскальної служби України |
| 2 | 2 | Гриньов Борис Вікторович | 167 | 105 | - | Фізика та математика<br>Chemical & Material Sciences | Інститут сцинтиляційних матеріалів |
| 3 | 3 | Пугач Валерій Михайлович | 130 | 83 | - | Фізика та математика<br>High Energy & Nuclear Physics | Інститут ядерних досліджень |
| 4 | 4 | Зінов’єв Геннадій Михайлович | 112 | 83 | - | Фізика та математика<br>High Energy & Nuclear Physics | Інститут теоретичної фізики ім.М.М.Боголюбова |
| 5 | 5 | Мартинов Євген Сергійович | 80 | 72 | 69 | Фізика та математика<br>High Energy & Nuclear Physics | Інститут теоретичної фізики ім.М.М.Боголюбова |
| 6 | 6 | Боголюбов Микола Миколайович (1909-1992) | 73 | - | - | Фізика та математика<br>Mathematical Physics | Інститут теоретичної фізики ім.М.М.Боголюбова |
| 7 | 7 | Вернадський Володимир Іванович (1863-1945) | 72 | - | - | Науки про Землю<br>Earth Sciences | Президія Національної академії наук України |
| 8 | 8 | Тимошенко Степан Прокопович (1878-1972) | 71 | - | - | Фізика та математика<br>Materials Engineering | Інститут механіки ім.С.П.Тимошенка |

Рисунок 3.1 Nbuviap

2. https://app.dimensions.ai/ - Найбільший у світі набір даних про дослідників та вчених (рисунок 3.2).

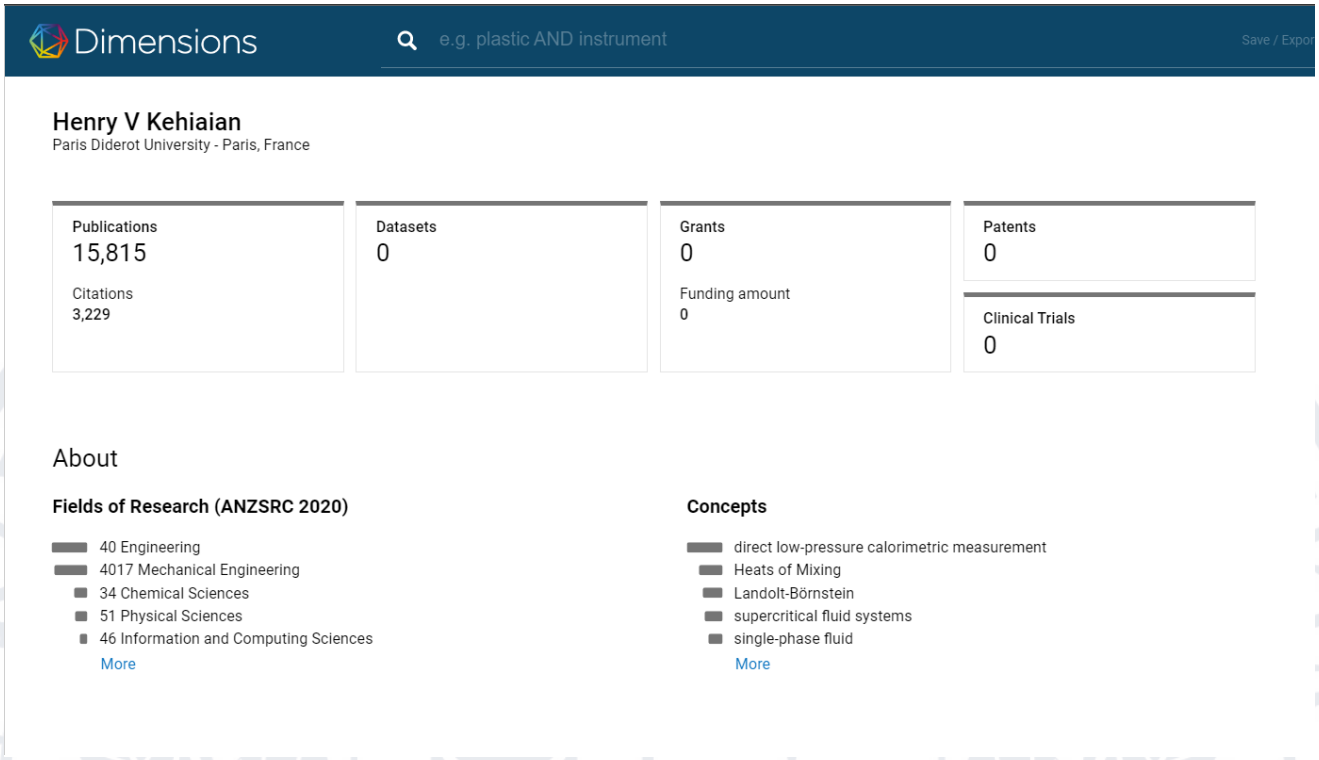



Рисунок 3.2 Dimensions

Використовується як джерело для отримання розширеної інформації про науковців. Пошук вченого виконується використовуючи транслітеровану та перекладені варіації імені, додатково перевіряється співпадіння профілів соціальної мережі Scopus.

Інформація, що збирається:

f) Галузі наукових робіт
g) Концепти
h) Роботи

### 3.3.2 Огляд API розробленого додатку

Swagger – користувацький інтерфейс для документації та взаємодії з Rest API. Робота програми розпочинається з сторінки Swagger (Рис 3.3).

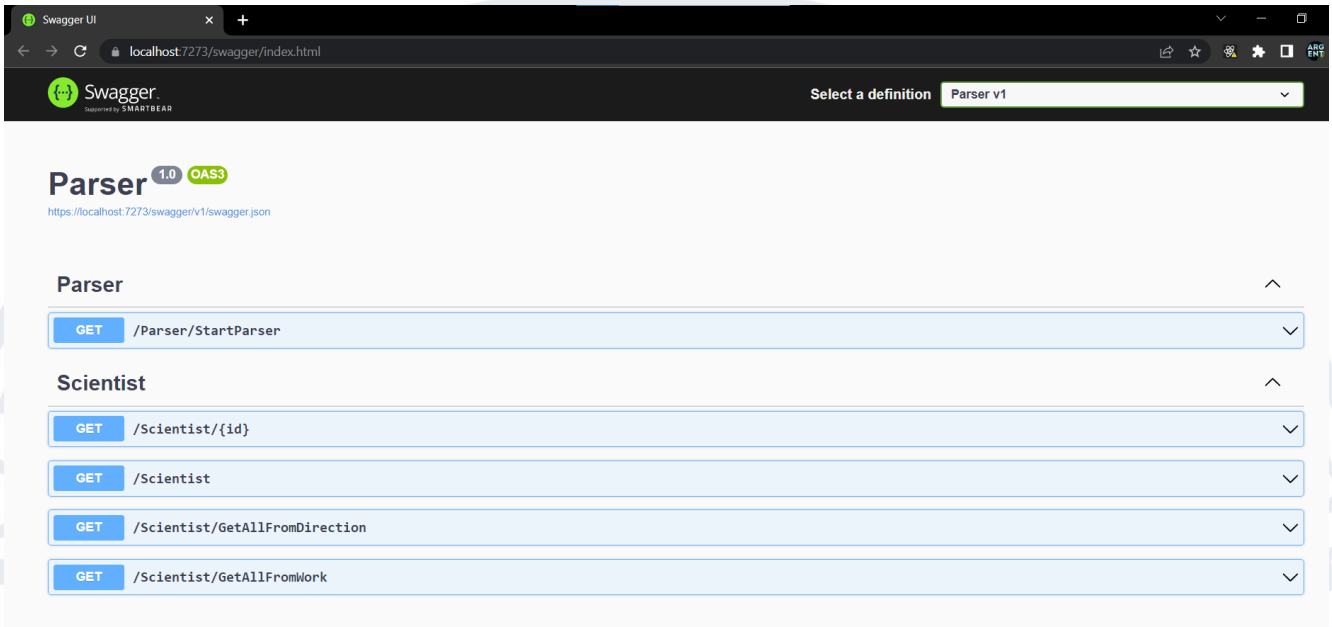



Рисунок 3.3 - Стартова сторінка

Графічний інтерфейс розділено на логічні блоки, що відповідають створеним контроллерам.

Веб додаток має 2 контроллери:

1. ParserController
2. ScientistController

ParserController – містить реалізацію кінцевої точки StartParser, яка використовується для запуску процесу збору даних. Бізнес логіка використовувана кінцевою точкою виконує збір, аналіз та збереження інформації у базі даних.

ScientistController – містить реалізацію кінцевих точок для взаємодії з наявними у базі даних сутностями scientist та пов’язаними з ними даними. Використовується для отримання необхідних даних.

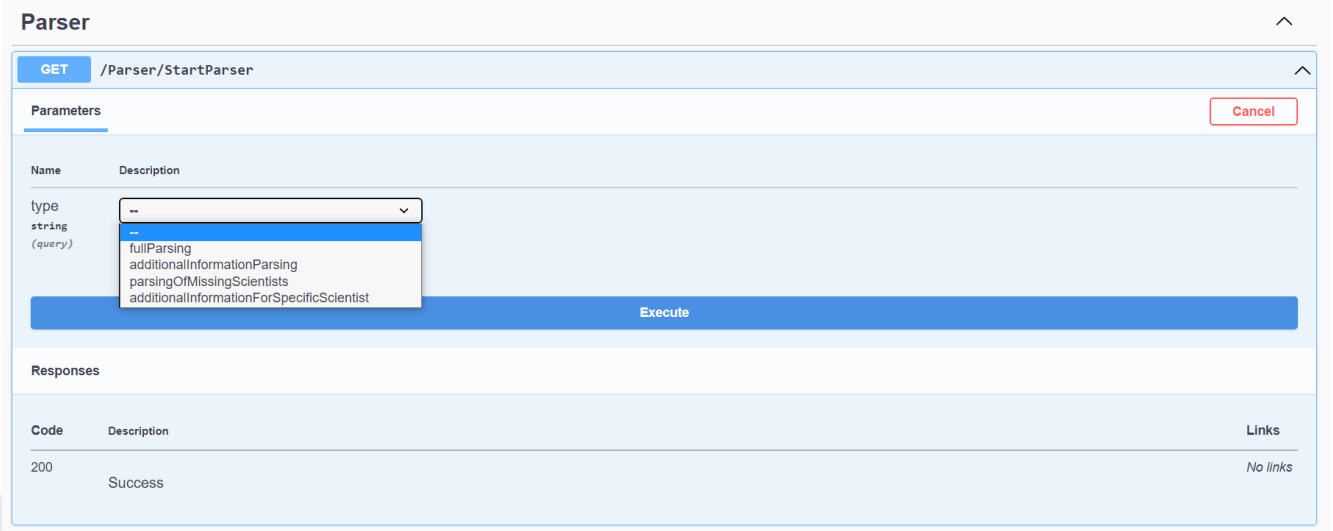



Рисунок 3.4 –кінцева точка StartParser

Кінцева точка StartParsing (рисунок 3.4) є точкою входу в додаток і використовується для запуску процесу збору інформації. Кінцева точка має один необхідний параметр type, який визначає тип виконуваного процесу. Type може бути задано однією з кількох можливих опцій:

1. FullParsing
2. AdditionalInformationParsing
3. ParsingOfMissingScientists
4. AdditionalInfromationParsingForSpecificScientist

FullParsing – при передачі вхідного параметра type зі значенням FullParsing процес збору інформації буде запущено у відповідному режимі, який виконає повний та послідовний аналіз всіх доступних даних для всіх знайдених вчених, при цьому уже проаналізованих раніше вчених буде пропущено.

AdditionalInformationParsing – при передачі вхідного параметра type зі значенням AdditionalInformationParsing процес збору інформації буде запущено у відповідному режимі, який виконає частковий аналіз всіх доступних даних для уже існуючих у базі даних записів про вчених, при цьому уже дані проаналізованих раніше вчених буде оновлено.

ParsingOfMissingScientists - при передачі вхідного параметра type зі значенням ParsingOfMissingScientists процес збору інформації буде запущено

у відповідному режимі, який виконає повний та послідовний аналіз всіх доступних даних для всіх вчених яких не було проаналізовано при попередніх запусках.

AdditionalInfromationParsingForSpecificScientist – при передачі вхідного параметра type зі значенням AdditionalInfromationParsingForSpecificScientist процес збору інформації буде запущено у відповідному режимі, який виконає частковий аналіз всіх доступних даних для уже існуючого у базі даних запису про конкретного вченого, знайденого за іменем, при цьому його дані буде оновлено.

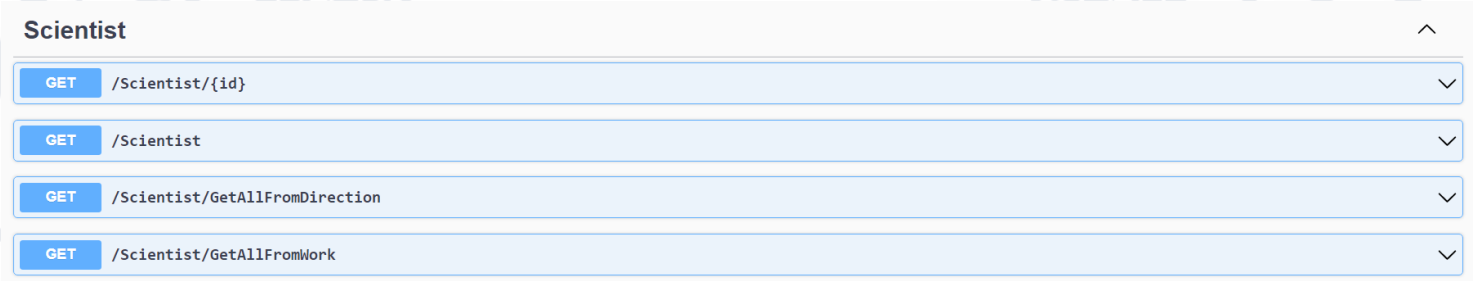



Рисунок 3.5 Scientist controller

Контроллер науковців надає доступ до 4 кінцевих точок, всі вони використовуються для отримання інформації про науковців, але різним чином.

Кінцеві точки контроллера науковців:

1. Scientist/{id} – отримання інформації про існуючого у базі даних науковця, з використанням унікального ідентифікатору запису.
2. Scientist – отримання вибірки науковців з наявної у базі даних інформації, з використанням можливостей створеного об’єкту-фільтра.
3. Scientist/GetAllFromDirection – отримання вибірки науковців з наявної у базі даних інформації, використовуючи напрям наукової роботи науковця як ключ для пошуку.
4. Scientist/GetAllFromWork - отримання вибірки науковців з наявної у базі даних інформації, використовуючи існуючи в базі даних роботу як ключ для пошуку.

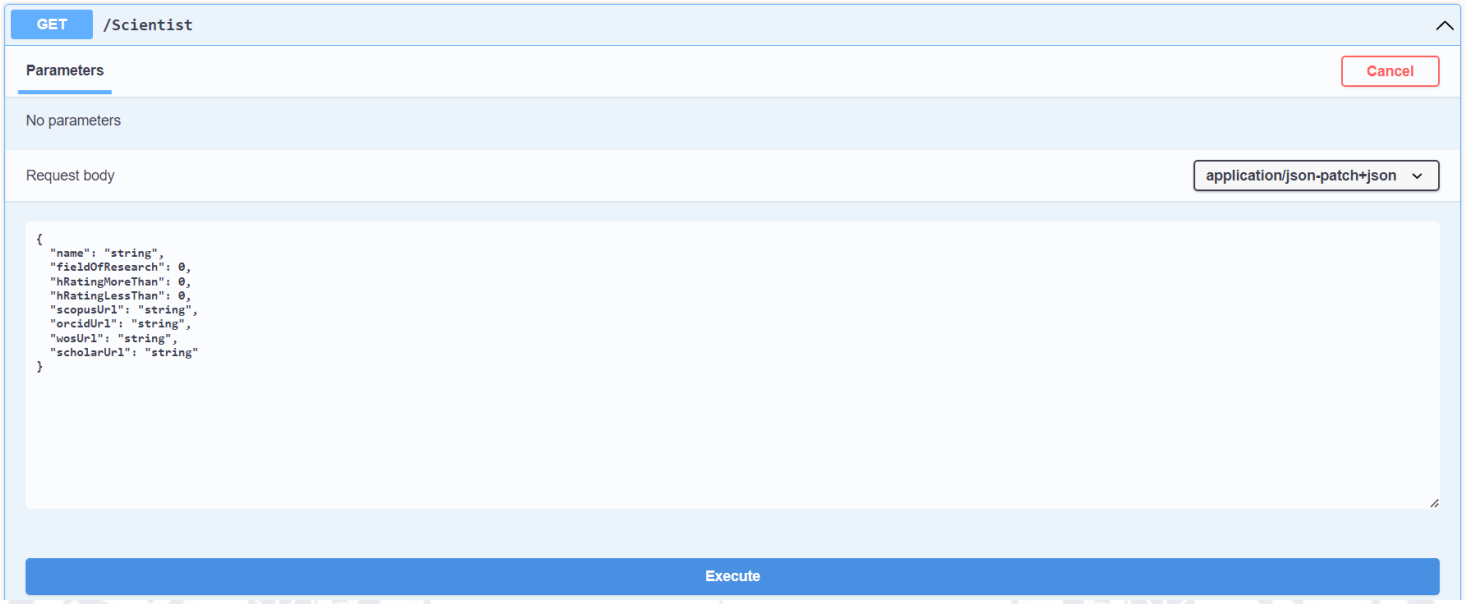



Рисунок 3.6 Кінцева точка Scientist

Кінцева точка Scientist використовує модель фільтра для передачі параметрів пошуку, і надає найширші можливості для пошуку науковців у базі даних.

```
3   public class ScientistFilter
4   {
        2 references
5       public string? Name { get; set; }
        2 references
6       public int? FieldOfResearchId { get; set; }
        2 references
7       public int? WorkId { get; set; }
        2 references
8       public int? HRatingMoreThan { get; set; }
        2 references
9       public int? HRatingLessThan { get; set; }
        2 references
10      public string? ScopusUrl { get; set; }
        2 references
11      public string? OrcidUrl { get; set; }
        2 references
12      public string? WosUrl { get; set; }
        2 references
13      public string? ScholarUrl { get; set; }
14  }
```

Рисунок 3.7 Модель ScientistFilter

Використовуючи можливості, що реалізовані у кінцевій точці Scientist, можна виконувати пошук науковців за наступними параметрами:

1. Name – пошук науковців за іменем
2. FieldOfResearchId – пошук науковців за напрямом наукової діяльності

3. WorkId – пошук науковців за ідентифікатором наукової роботи
4. HRatingMoreThan – пошук науковців HRating яких більше за вказане значення
5. HRatingLessThan – пошук науковців HRating яких менше за вкащане значення
6. ScopusUrl – пошук науковців за посиланням на профіль у соціальній мережі Scopus
7. OrcidUrl – пошук науковців за посиланням на профіль у соціальній мережі Orcid
8. WosUrl – пошук науковців за посиланням на профіль у соціальній мережі WOS
9. ScholarUrl – пошук науковців за посиланням на профіль у соціальній мережі Google Scholar

Приклад відповіді сервера:

```
39 references
public class Scientist
{
    7 references
    public int Id { get; set; }
    6 references
    public string Name { get; set; }
    1 reference
    public string? Degree { get; set; }
    3 references
    public int Rating { get; set; }

    2 references
    public int? OrganizationId { get; set; }
    1 reference
    public Organization? Organization { get; set; }

    [JsonIgnore]
    2 references
    public ICollection<ScientistFieldOfResearch> ScientistFieldsOfResearch { get; set; }
    [JsonIgnore]
    1 reference
    public ICollection<ScientistWork> ScientistsWorks { get; set; }
    [JsonIgnore]
    1 reference
    public ICollection<Concept> Concepts { get; set; }
    [JsonIgnore]
    7 references
    public ICollection<ScientistSocialNetwork> ScientistSocialNetworks { get; set; }

}
```

Рисунок 3.8 модель представлення науковця

## 3.4 Програмне рішення

### 3.4.1 Загальна структура додатку

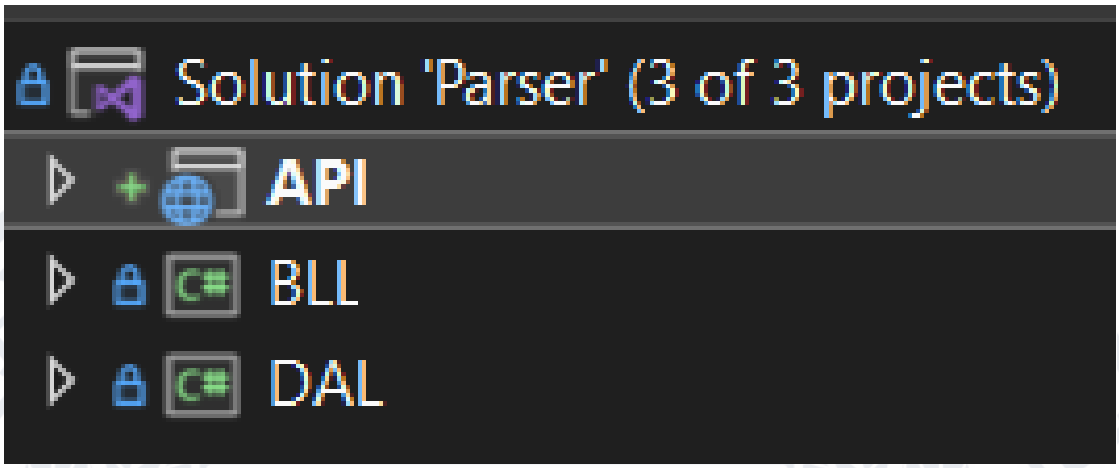



Рисунок 3.9 Модулі верхнього рівня

Структуру проекту побудовано за класичним трирівневим підходом, що складається з файлу рішення та 3 проектів у ньому, кожен з яких відповідає за відведену йому частин реалізації програмного рішення.

Файл Parser.sln – коріневий файл створеного рішення, відповідає за зв'язки між існуючими у ньому проектами, за їх взаємодію та залежності. Окрім того, рівень рішення визначає опції роботи зі проектами та включає в себе файли системи керування версіями GIT.

Трирівнева архітектура побудови додатку є класичним підходом для створення монолітних рішень малого та середнього розміру, за умови правильної структуризації, використання абстракцій для реалізації ін'єкції залежностей та низької зв'язності структурних елементів додатку, може бути з легкістю розділена на окремі сервіси та приведена до мікросервісної архітектури додатку.

Саме задля збереження можливості модифікації додатку та переходу до мікросервісної акрхітектури, підхід до ін'єкції залежностей реалізовано за допомогою інтерфейсів – абстрактних сутностей, що лише визначають необхідні елементи (методи та поля) для майбутніх конкретних реалізацій, що уже міститимуть тіла методів та відповідні поля класів.

Для зменшення зв’язності складових частин рішення було прийнято рішення не створювати залежність між верхнім (API) та нижнім (DAL) рівнями додатку, їх взаємодія відбувається виключно через проміжний шар бізнес логіки (BLL). При цьому шар бізнес логіки має чітко визначену залежність від шару доступу до бази даних, але не має залежності від шару зовнішнього доступу. Приклад взаємодії між складовими частинами додатку зображено на рисунку 3.10 у вигляді діаграми зв’язків.

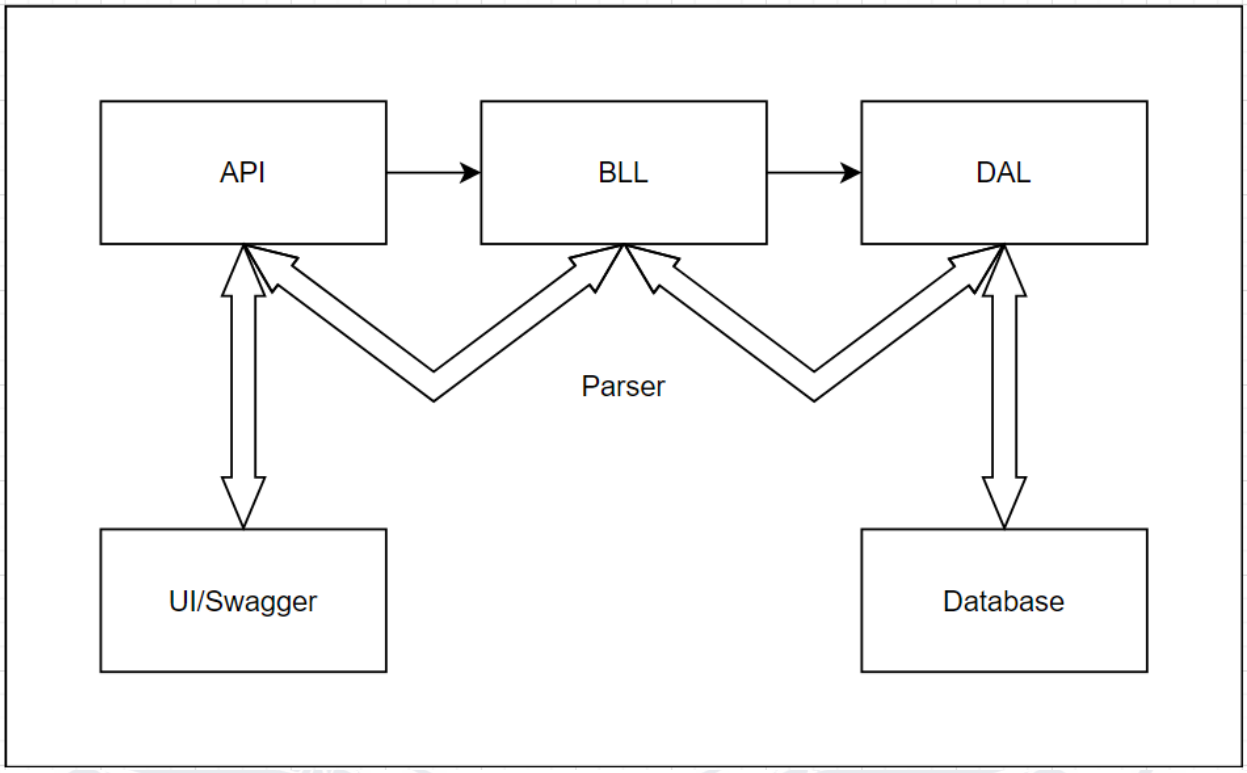



Рисунок 3.10 Діаграма зв’язків додатку

### 3.4.2 Взаємодія з базою даних

Для взаємодії з базою даних з коду було обрано Entity Framework ORM, що надає велику кількість можливостей для роботи з даними, структурою, версіями та налаштуваннями баз даних, і при цьому не потребує внесення змін напряму до бази даних. Entity Framework є реалізацією ORM об’єктно-реляційного відображення бази даних у коді – виступає в ролі ланки зв’язку між кодом та базою даних і дає можливість розробнику працювати зі звичними об’єктами та синтаксисом обраної мови програмування. Крім того, обрана ORM надає функціонал контролю версій бази даних, і зберігає всі впроваджені

структурні зміни у вигляді SQL скриптів, що називаються міграціями (рисунок 3.11) і зберігаються на рівні DAL в окремій директорії Migrations.

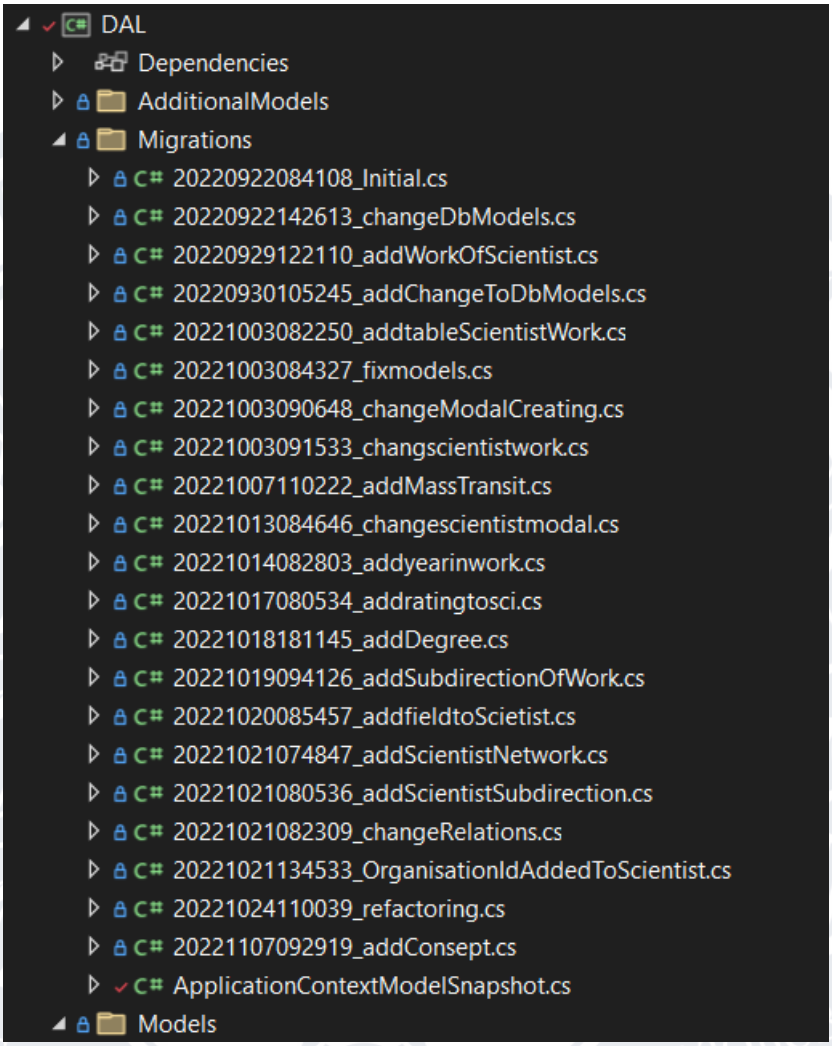



Рисунок 3.11 Файли міграцій бази даних

Це дає змогу безпроблемного розгортання бази у додаткових середовищах, проведення відновлення попередніх версій, або глобального перестворення у випадку невідворотних змін, що спричинили критичні проблеми. Міграції – файли, що використовуються EntityFramework для відслідковування змін у структурі бази даних та можуть бути створеними за допомогою консольної команди «add-migration {name of the migration}». Файли міграцій містять код на мові C#, та інтерпретуються EntityFramework у SQL скрипти, що і використовуються для взаємодії з базою даних.

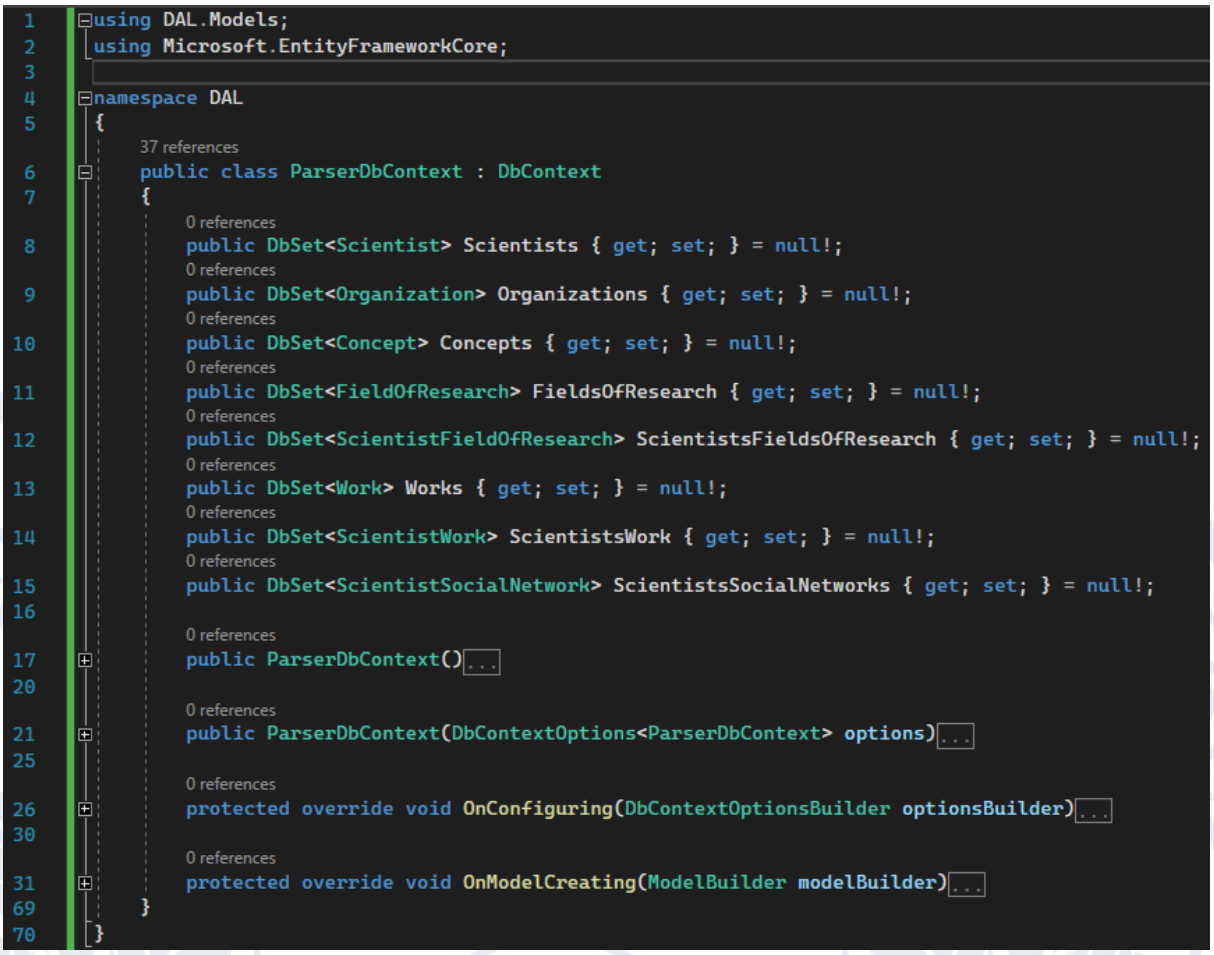
Entity Framework надає два альтернативні підходи до роботи з базою даних, Code First – який було використано при розробці додатку та Database First:

1. Code first – базу даних буде створено на основі існуючих моделей об’єктів, та визначених між ними зв’язків. Для визначення зв’язків та обмежень таблиць використовується перевизначення методу OnModelCreating в класі контексту бази даних (рисунок 3.12).
2. Database first – код буде згенеровано на основі існуючої бази даних.

```
0 references
protected override void OnModelCreating(ModelBuilder modelBuilder)
{

    modelBuilder.Entity<ScientistWork>()
        .HasOne(bc => bc.Scientist)
        .WithMany(b => b.ScientistsWorks)
        .HasForeignKey(bc => bc.ScientistId);

    modelBuilder.Entity<ScientistWork>()
        .HasOne(bc => bc.Work)
        .WithMany(c => c.ScientistsWorks)
        .HasForeignKey(bc => bc.WorkId);

    modelBuilder.Entity<ScientistSocialNetwork>()
        .HasOne(s => s.Scientist)
        .WithMany(g => g.ScientistSocialNetworks)
        .HasForeignKey(s => s.ScientistId);

    modelBuilder.Entity<Scientist>()
        .HasOne(s => s.Organization)
        .WithMany(g => g.Scientists)
        .HasForeignKey(s => s.OrganizationId);

    modelBuilder.Entity<Concept>()
        .HasOne(s => s.Scientist)
        .WithMany(g => g.Concepts)
        .HasForeignKey(s => s.ScientistId);

    modelBuilder.Entity<ScientistFieldOfResearch>()
        .HasOne(s => s.Scientist)
        .WithMany(g => g.ScientistFieldsOfResearch)
        .HasForeignKey(s => s.ScientistId);

    modelBuilder.Entity<ScientistFieldOfResearch>()
        .HasOne(s => s.FieldOfResearch)
        .WithMany(g => g.ScientistsFieldsOfResearch)
        .HasForeignKey(s => s.FieldOfResearchId);
}
```

Рисунок 3.12 перевизначений метод OnModelCreating

Безпосередня взаємодія з базою даних відбувається з використанням класу ParserDbContext, який наслідує базовий клас DbContext з простору імен Microsoft.EntityFrameworkCore і ініціалізує поля, що відповідають створеним класам таблиць бази даних (Рисунок 3.13).



```
using DAL.Models;
using Microsoft.EntityFrameworkCore;

namespace DAL
{
    37 references
    public class ParserDbContext : DbContext
    {
        0 references
        public DbSet<Scientist> Scientists { get; set; } = null!;
        0 references
        public DbSet<Organization> Organizations { get; set; } = null!;
        0 references
        public DbSet<Concept> Concepts { get; set; } = null!;
        0 references
        public DbSet<FieldOfResearch> FieldsOfResearch { get; set; } = null!;
        0 references
        public DbSet<ScientistFieldOfResearch> ScientistsFieldsOfResearch { get; set; } = null!;
        0 references
        public DbSet<Work> Works { get; set; } = null!;
        0 references
        public DbSet<ScientistWork> ScientistsWork { get; set; } = null!;
        0 references
        public DbSet<ScientistSocialNetwork> ScientistsSocialNetworks { get; set; } = null!;

        0 references
        public ParserDbContext()...

        0 references
        public ParserDbContext(DbContextOptions<ParserDbContext> options)...

        0 references
        protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)...

        0 references
        protected override void OnModelCreating(ModelBuilder modelBuilder)...
    }
}
```

Рисунок 3.13 клас ParserDbContext

Як можна побачити на рисунку 3.13, у класі ParserDbContext визначено 8 полів, кожне з яких є представленням відповідної таблиці у базі даних (рисунок 3.1), а також конструктори та перевизначені методи OnConfiguring та OnModelCreating (рисунок 3.12).

Сам же клас доступу до бази даних ParserDbContext зареєстровано у впроваджувачі залежностей (більш детально – дивитись у розділі 3.4.3 – впровадження залежностей) та ініціалізовано стрічкою підключення до бази даних, що містить необхідні для підключення параметри та зберігається у файлі конфігурації проекту (більш детально – дивитись у розділі 3.4.6 – конфігурація проекту).

Класи, що використовуються як представлення таблиць, створено у просторі імен DAL.Models (Рисунок 3.14)

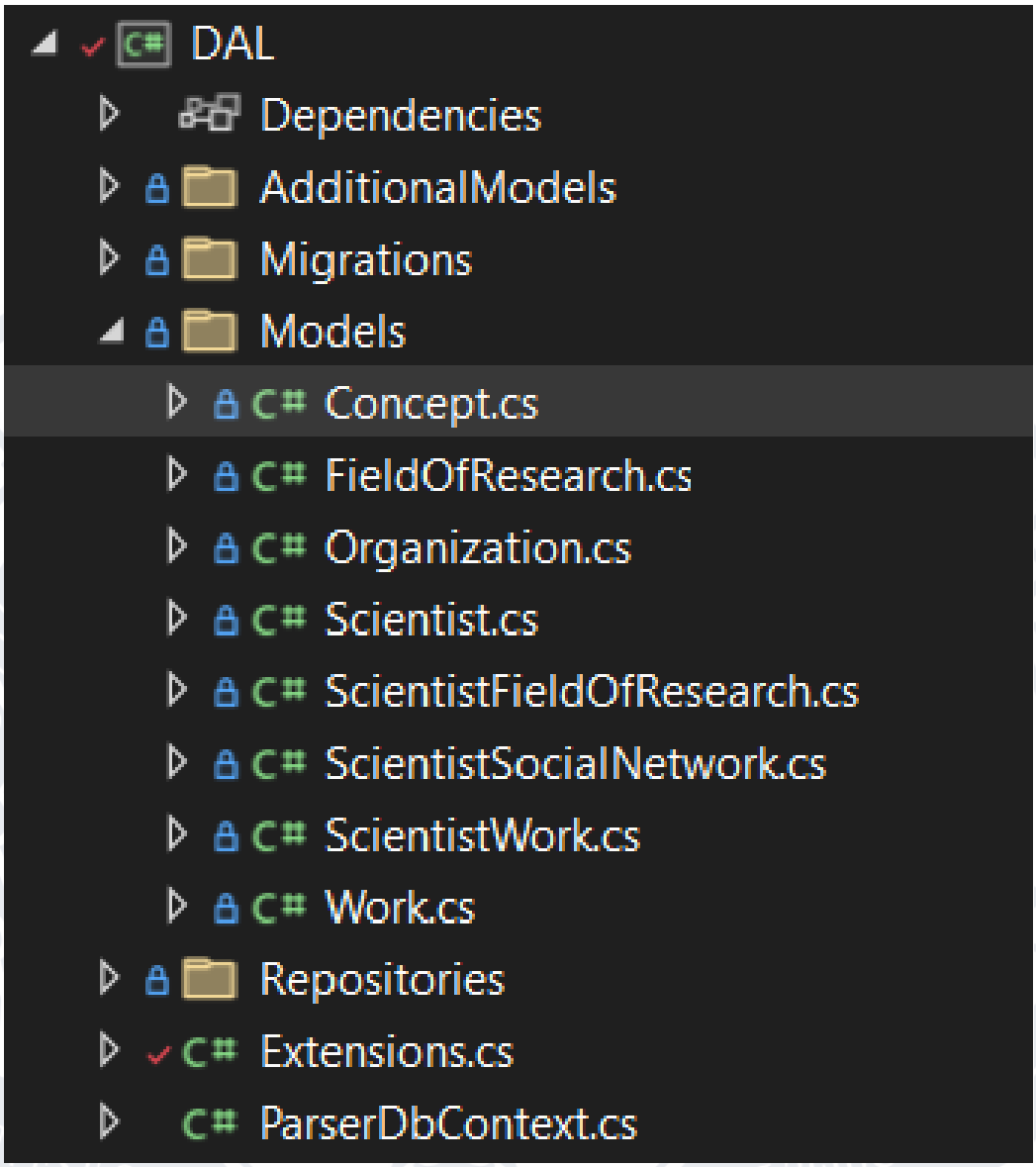



Рисунок 3.14 моделі представлення таблиць

### 3.4.3 Впровадження залежностей

Для кращого контролю над використанням ресурсів системи, зокрема пам’яті, використано інструмент впровадження залежностей, який контролює життєві цикли та використання пам’яті зареєстрованих сервісів-залежностей. Залежності – поняття, що застосовується до сервісів які використовуються під час виконання коду, і є невід’ємною частиною додатку. Таким чином, для прикладу, залежностями для проекту API є сервіси проекту BLL, а для проекту BLL, відповідно, залежностями є сервіси проекту DAL.

Для реєстрації залежностей у DAL і BLL створено класи Extensions, що містять логіку впровадження залежностей і визначення типів життєвих циклів для них (Рисунок 3.15).

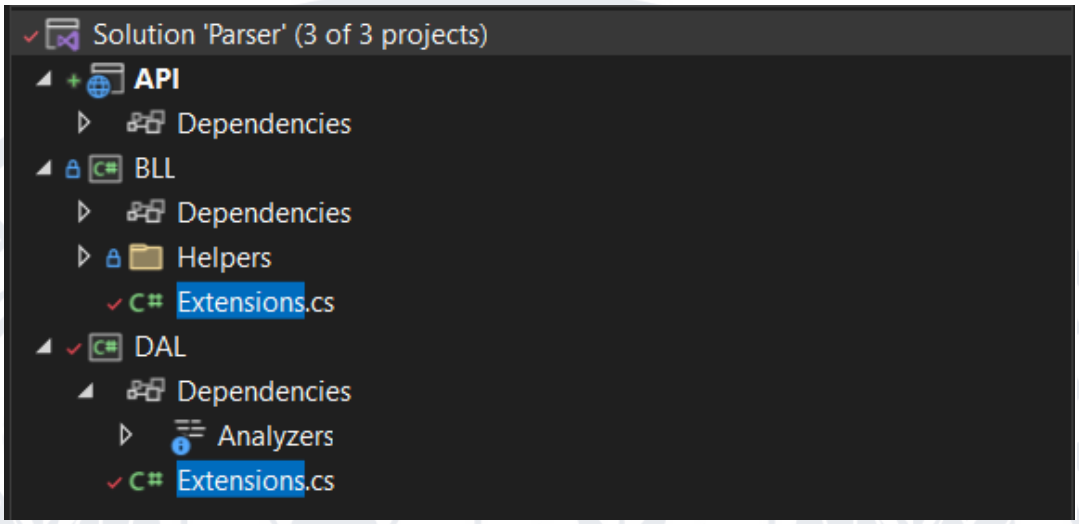



Рисунок 3.15 Extensions класи

Клас Extensions на рівні DAL відповідає за реєстрацію сервісів створених на цьому рівні (рисунок 3.16)

```
namespace DAL
{
    0 references
    public static class Extensions
    {
        1 reference
        public static void AddDataLayer(this IServiceCollection services, IConfiguration configuration)
        {
            services.AddDbContext<ParserDbContext>(options => options.UseNpgsql(configuration.GetConnectionString("ParserDB")));

            using (var serviceScope = services.BuildServiceProvider().GetRequiredService<IServiceScopeFactory>().CreateScope())
            {
                using (var context = serviceScope.ServiceProvider.GetService<ParserDbContext>())
                {
                    context.Database.Migrate();
                }
            }

            services.AddScoped(typeof(IRepository<>), typeof(Repository<>));

            services.AddScoped(typeof(IFieldOfResearchRepository), typeof(FieldOfResearchRepository));

            services.AddScoped(typeof(IScientistFieldOfResearchRepository), typeof(ScientistFieldOfResearchRepository));

            services.AddScoped(typeof(IScientistRepository), typeof(ScientistRepository));

            services.AddScoped(typeof(IScientistSocialNetworkRepository), typeof(ScientistSocialNetworkRepository));

            services.AddScoped(typeof(IScientistWorkRepository), typeof(ScientistWorkRepository));

            services.AddScoped(typeof(IWorkRepository), typeof(WorkRepository));

            services.AddScoped(typeof(IOrganizationRepository), typeof(OrganizationRepository));

            services.AddScoped(typeof(IConceptRepository), typeof(ConceptRepository));
        }
    }
```

Рисунок 3.16 клас Extensions рівня DAL

Клас Extensions надає реалізацію єдиного статичного методу AddDataLayer, що згодом буде використаний при реєстрації BLL, який є залежним від DAL.

Оскільки основною задачею DAL є робота з базою даних, реєстрація класу доступу до бази даних DbContext, реалізацією якого є ParserDbContext, є першою, це можна побачити у стрічці 13 класу Extension на рисунку 3.16, у цій же стрічці відбувається ініціалізація класу стрічкою підключення до бази даних configuration.GetConnectionString("ParserDB"). Після цього у стрічках 15-21 відбувається процес міграції бази даних, що виконується EntityFramework, і полягає в оновленні бази даних до останньої версії, відповідно до існуючих файлів міграцій (рисунок 3.11). Цей самий процес може бути виконаний у ручному режимі з використанням консольної команди «update-database», що дає додаткові можливості для перевірки правильності застосованих міграцій.

У стрічках 23-39 решта сервісів, які містять реалізацію методів для роботи з базою даних для відповідних таблиць, реєструються з використанням типу життєвого циклу Scoped, що значить що буде створено один єдиний об’єкт класу і його буде повернуто для всіх запитів отримання залежності, у межі одного звернення до додатку (виклику кінцевої точки).

Додатково, варто відмітити, що всі залежності зареєстровано як реалізації відповідних інтерфейсів, що дає змогу використання залежностей не як конкретних реалізацій, а як абстрактних сутностей, які можуть мати необмежену кількість різних реалізацій, для прикладу, всі зареєстровані repository сервіси реалізують загальний IRepository інтерфейс, і можуть представляти його в повній мірі.

Клас Extensions на рівні BLL відповідає за реєстрацію сервісів створених на цьому рівні (рисунок 3.17).

```
11  namespace BLL
12  {
        0 references
13      public static class Extensions
14      {
            1 reference
15          public static void AddBusinessLayer(this IServiceCollection services, IConfiguration configuration)
16          {
17              services.AddDataLayer(configuration);
18
19              services.AddScoped<IScientistService, ScientistService>();
20              services.AddScoped<IParsingHandler, ParcingHandler>();
21              services.AddScoped<IDimensionsParser, DimensionsParser>();
22              services.AddScoped<INbuviapParser, NbuviapParser>();
23
24              services.AddScoped<IWebDriver, ChromeDriver>();
25          }
26      }
27  }
```

Рисунок 3.17 клас Extensions рівня BLL

Клас Extensions надає реалізацію єдиного статичного методу AddBusinessLayer, що згодом буде використаний під час налаштування веб-додатку на рівні API, який є залежним від BLL.

Оскільки проект BLL є залежним від проекту DAL, першою операцією є виклик методу AddDataLayer класу Extensions рівня DAL (рисунок 3.16) для реєстрації всіх необхідних залежностей.

Наступним кроком, у стрічках 19-22, відбувається реєстрація сервісів реалізованих на рівні BLL, всі вони зареєстровані як реалізації відповідних інтерфейсів, і будуть використовуватись виключно як інтерфейси, задля збереження низької зв’язності коду.

Остання дія у методі – реєстрація ChromeDriver як реалізація IWebDriver, що є класом з простору імен WebDriver і використовуватиметься для взаємодії з браузером Google Chrome.

Як логічне завершення роботи над ін’єкцією залежностей, метод AddBusinessLayer викликається у класі Program на рівні API, який є точкою входу в додаток, виконує налаштування і конфігурацію а також ініціалізує всі необхідні сервіси.

### 3.4.4 Рівень DAL

Data Access Layer є найнижчим рівнем у ієрархії проектів створеного програмного рішення, тут створено інтерфейси (рисунок 3.18), які визначають методи необхідні для взаємодії з базою даних, а також класи (рисунок 3.21), що реалізують вищезгадані інтерфейси, названі репозиторіями.

Інтерфейси при об’явленні позначаються ключовим словом interface, що значить що методи цієї структурної одиниці не можуть мати реалізацій, а також немає можливості створити об’єкт інтерфейсу. Інтерфейси є абстрактними елементами системи, що повинні бути реалізовані у класах нащадках, а їх використання зумовлене прагненням до зменшення зв’язності та дублювання коду.

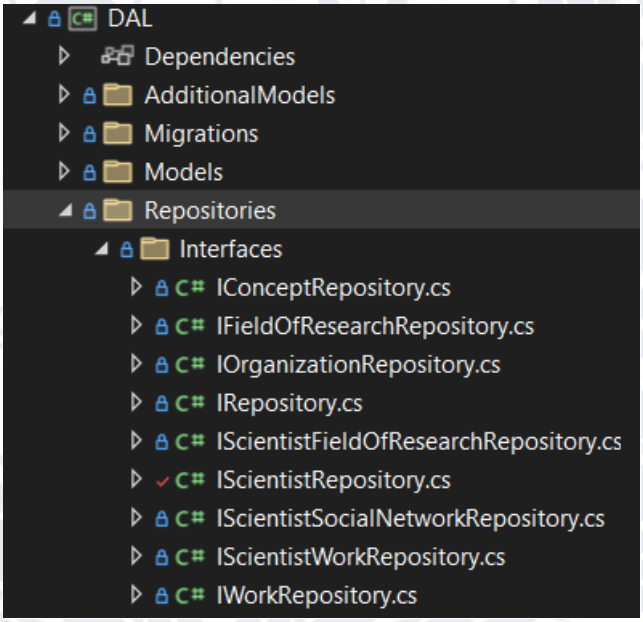



Рисунок 3.18 Інтерфейси репозиторіїв DAL

Найнижчим у ієрархії інтерфейсів є інтерфейс IRepository (рисунок 3.19), у якому визначено методи, які повинні бути реалізовані для всіх без виключення моделей. Крім того, інтерфейс IRepository є загальним інтерфейсом, що дозволяє застосовувати його до різних типів, за умови, що типи мають однакову поведінку – і у нашому випадку це саме те, що потрібно. Це і є чудовим прикладу використання інтерфейсу для уникнення дублювання коду, оскільки наш інтерфейс є загальним, а тип з яким працює інтерфейс можна вказувати явно у кожному окремому випадку – нам не потрібно

надавати окрему реалізацію для кожного з типів, отже багато типів – одна реалізація.

```
1   namespace DAL.Repositories.Interfaces
2   {
        18 references
3       public interface IRepository<EntType>
4       {
            32 references
5           IQueryable<EntType> GetAll();
            3 references
6           Task<int> CreateAsync(EntType entity);
            2 references
7           Task<int> CreateAsync(IEnumerable<EntType> entities);
            2 references
8           Task<int> UpdateAsync(EntType entity);
            7 references
9           Task<int> UpdateAsync(IEnumerable<EntType> entities);
            1 reference
10          Task<int> DeleteAsync(EntType entity);
            1 reference
11          Task<int> DeleteAsync(IEnumerable<EntType> entities);
12      }
13  }
```

Рисунок 3.19 IRepository інтерфейс

Як можна побачити на рисунку 3.19 інтерфейс IRepository ініціалізує низку загальних методів, які є базовими та необхідними для всіх типів, окрім базових методів Create, Update та Delete додатково визначено такі ж методи, але для роботи з масивами даних, їх реалізація дозволить працювати з даними використовуючи так званий «Bulk Operation» підхід, що значно пришвидшує виконання логіки, оскільки звернення до бази даних – процес який потребує використання значної кількості ресурсів та часу, і крім того кількість одночасно можливих з’єднань з базою обмежена.

На останок, в інтерфейсі об’явлено метод GetAll, що повертає особливий тип даних IQueryable, завдяки цьому ми уникаємо конкретної реалізації методу, яка повинна бути різною для кожного з типів, і залишаємо можливість для реалізації конкретного підходу для кожного з них. Оскільки IQueryable є абстрактним типом запиту і не повертає справжніх даних, а його реальне представлення це запит до бази даних, який може бути модифікований згодом, відповідно до вимог конкретної задачі, та виконаний після цього – ми надаємо можливість для створення конкретної реалізації для кожної з моделей.

Окрім базового інтерфейсу IRepository, який безпосередньо не використовується для доступу до бази даних, а є лише абстрактним представленням базових операцій, створено ще низку інтерфейсів, по одному для кожної з моделей.

Для прикладу розглянемо інтерфейс IScientistRepository (рисунок 3.20) створений для моделі Scientist, яка є представленням таблиці Scientist у базі даних.

```
3   namespace DAL.Repositories.Interfaces
4   {
        8 references
5       public interface IScientistRepository : IRepository<Scientist>
6       {
            2 references
7           Task<Scientist> GetAsync(int id);
            3 references
8           Task<Scientist> GetAsync(string name);
9       }
10  }
```

Рисунок 3.20 IScientistRepository

Інтерфейс IScientistRepository є наслідником базового інтерфейсу IRepository і явно передає тип моделі з якою він працює, тобто Scientist. На додачу інтерфейс IScientistRepository створює нові, унікальні для моделі Scientist методи GetAsync використовуючи id вченого, та GetAsync використовуючи ім’я вченого. Створення цих методів на рівні інтерфейсу репозиторія конкретної сутності Scientist зумовлено їх конкретною задачею та типами вхідних даних, для прикладу вчений має ім’я, а інші моделі можуть не мати такого ж поля, теж саме стосується і ідентифікатора запису, у випадку моделі науковця це тип даних int, але для інших моделей може бути використано і інші типи, зокрема bigint або GUID.

Оскільки інтерфейс IScientistRepository є наслідником інтерфейсу IRepository<Scientist> - його реалізація буде зобов’язана надати конкретну імплементацію для всіх методів IScientistRepository та IRepository<Scientist>. Пізніше ми розглянемо, як вдалось уникнути дублювання реалізації для базових методів, які створено в інтерфейсі IRepository.

Тепер, коли ми розглянули інтерфейси, які були створені на рівні DAL, можна перейти до реалізацій (рисунок 3.21).

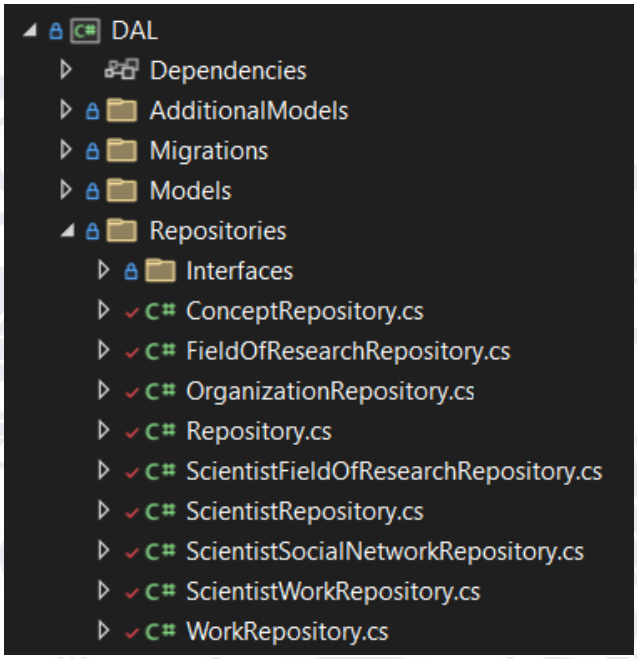



Рисунок 3.21 Реалізації репозиторіїв

Почати ж варто з загального класу Repository, який реалізує найнижчий в ієрархії інтерфейс IRepository (рисунок 3.22), який де і визначено реалізації методів відповідно до переданого типу даних.

```
namespace DAL.Repositories
{
    18 references
    public abstract class Repository<EntType> : IRepository<EntType> where EntType : class
    {
        private readonly ParserDbContext _context;

        8 references
        public Repository(ParserDbContext context)
        {
            _context = context;
        }

        32 references
        public IQueryable<EntType> GetAll()
        {
            return _context.Set<EntType>().AsQueryable();
        }

        3 references
        public Task<int> CreateAsync(EntType entity)
        {
            _context.Add(entity);
            return _context.SaveChangesAsync();
        }

        2 references
        public Task<int> CreateAsync(IEnumerable<EntType> entities)
        {
            _context.AddRange(entities);
            return _context.SaveChangesAsync();
        }

        2 references
        public Task<int> UpdateAsync(EntType entity)
        {
            _context.Update(entity);
            return _context.SaveChangesAsync();
        }
```

Рисунок 3.22 Клас Repository, що реалізує IRepository

Як можна бачити на рисунку 3.22, клас Repository використовує об’єкт класу ParserDbContext, який отримано з системи впровадження залежностей (дивитись розділ 3.4.3), і метод Set<EntType> для явного вказання на тип, з яким ми працюємо. Оскільки клас Repository є реалізацію інтерфейсу IRepository саме тут реалізовано всі об’явлені у інтерфейсі методи, а завдяки використанні дженерік підходу нам не потрібно створювати окремі реалізації для кожної з моделей.

На рисунку також 3.22 можна побачити реалізації деяких методів інтерфейсу, зокрема:

1. GetAll – що використовується для отримання IQueryable
2. CreateAsync – створення запису у базі даних
3. CreateAsync(IEnumerable) – створення низки записів у базі даних
4. UpdateAsync – оновлення даних запису у базі даних

Решта методів містять аналогічні реалізації, їх можна переглянути у кодовій базі.

Проте робота з базою відбувається не через клас Repository, оскільки він не представляє конкретну сутність, а також відмічений ключовим словом abstract, що значить, що він потребує конкретної реалізації.

Конкретну реалізацію також розглянемо на прикладі моделі Scientist, що і було зроблено під час огляду інтерфейсів. Отож, репозиторій для моделі Scientist, ScientistRepository (рисунок 3.23), є наслідником класу Repository<Scientist>, де знаходяться реалізації базових методів, а також інтерфейсу IScientistRepository, де створено, але не реалізовано, унікальні для моделі Scientist методи. Виходячи з цього, клас ScientistRepository містить реалізації лише для унікальних методів моделі Scientist, зокрема GetAsync використовуючи унікальний ідентифікатор, та GetAsync використовуючи ім’я науковця.

```
2 references
public class ScientistRepository : Repository<Scientist>, IScientistRepository
{
    0 references
    public ScientistRepository(ParserDbContext dbContext) : base(dbContext)
    {
    }

    2 references
    public async Task<Scientist> GetAsync(int id)
    {
        return await GetAll().FirstOrDefaultAsync(scientist => scientist.Id == id);
    }

    3 references
    public async Task<Scientist> GetAsync(string name)
    {
        return await GetAll().FirstOrDefaultAsync(scientist => scientist.Name.Equals(name));
    }
}
```

Рисунок 3.23 ScientistRepository

GetAsync(int id) використовує метод GetAll, реалізований в базовому класі Repository, для отримання IQueryable для таблиці Scientist, і модифікує його для виконання пошуку за унікальним ідентифікатором запису, для цього використано асинхронний метод FirstOrDefaultAsync з простору імен Microsoft.EntityFrameworkCore, який повертає об’єкт, якщо такий знайдено, або нульове значення, у випадку, коли об’єкт не знайдено. Порівняння відбувається між переданим ідентифікатором, і тим, що наявний у базі даних.

Аналогічна операція відбувається і у реалізації методу GetAsync(string name), за винятком того, що для порівняння двох значень використано базовий метод Equals типу даних string.

Також, варто відмітити, що всі операції з базою даних виконуються асинхронно, для уникнення простою потоків, недоцільного використання ресурсів, та покращення швидкодії додатку.

Таким чином реалізовано загальновідомий архітектурний паттерн для роботи з базою даних IRepository.

### 3.4.5 Рівень BLL

Business Logic Layer є середнім рівнем у ієрархії проектів створеного програмного рішення, методи, що виконують основну бізнес логіку проекту реалізовано саме тут, крім того BLL є рівнем посередником для зв'язку між API та DAL.

Шар бізнес логіки (рисунок 3.24) надає низку інтерфейсів та їх реалізацій, що розширюють операції рівня доступу до бази даних, а також реалізують основну логіку проекту – збір інформації про науковців з відкритих джерел мережі інтернет. Окрім того, BLL містить методи помічники для роботи зі стрічками та WebDriver, які реалізовано у StringHelper та WebDriverHelper класах відповідно, та додаткові класи необхідні для роботи додатку, зокрема фільтри для моделей.

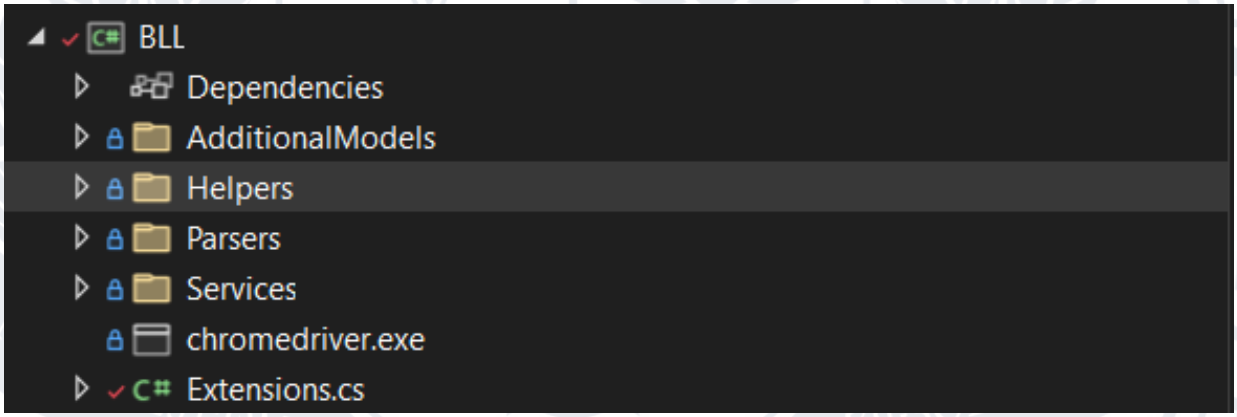



Рисунок 3.24 Структура BLL

На верхньому рівні проекту розташовано єдиний клас Extensions, який відповідає за реєстрацію сервісів у системі ін'єкції залежностей і уже був розглянутий у розділі 3.4.3, а також директорії, що містять додаткові класи, класи-помічники та бізнес-логіку проекту.

Решту класів розглянемо згідно їх порядку (рисунок 3.24) і почнемо з директорії AdditionalModels (рисунок 3.25).

Як видно з рисунка, директорія включає в себе низку класів, більшість з яких є класами фільтрів для конкретних моделей, інші ж є допоміжними класами, так званими Domain Models, які використовуються виключно в

середині системи для виконання певних операцій або ж виступають у ролі посередників при передачі даних.

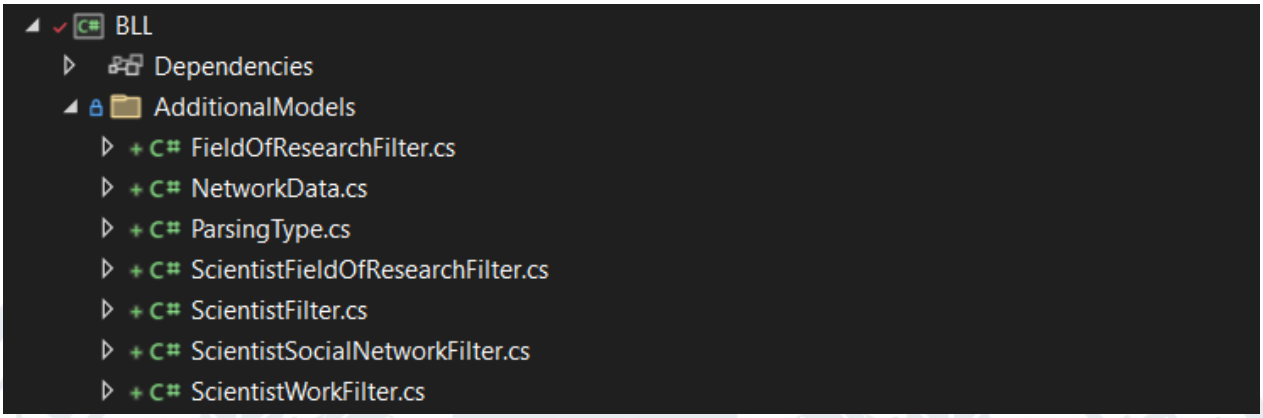



Рисунок 3.25 Директорія BLL.AdditionalModels

Отож, умовно контент директорії можна розділити на 2 типи класів, перші – класи фільтри (рисунок 3.26) та друга класи-помічники (рисунок 3.27).

```
3   public class ScientistFilter
4   {
        2 references
5       public string? Name { get; set; }
        2 references
6       public int? FieldOfResearchId { get; set; }
        2 references
7       public int? WorkId { get; set; }
        2 references
8       public int? HRatingMoreThan { get; set; }
        2 references
9       public int? HRatingLessThan { get; set; }
        2 references
10      public string? ScopusUrl { get; set; }
        2 references
11      public string? OrcidUrl { get; set; }
        2 references
12      public string? WosUrl { get; set; }
        2 references
13      public string? ScholarUrl { get; set; }
14  }
15  }
```

Рисунок 3.26 Приклад класу-фільтра ScientistFilter

Розглянемо тип клас-фільтр на прикладі ScientistFilter, як можна побачити на рисунку 3.26 клас надає доступ до низки полів, всі з яких відмічено як «ті, що можуть бути пустими» або ж nullable, це зроблено для того щоб жодна з опцій фільтрації не несла обов’язкового характеру.

Тип клас-помічник буде розглянуто на прикладі класу NetworkData (рисунок 3.27), який використовується при роботі з соціальними мережами вчених.


Рисунок 3.27 клас-помічник NetworkData

Клас NetworkData визначає низку полів:

1. XPath – шлях до елемента в HTML розмітці сторінки.
2. Scientist – об’єкт науковця, до якого відноситься конкретний об’єкт NetworkData.
3. Value – посилання на соціальну мережу науковця, отримане під час виконання логіки.
4. NetworkType – тип соціальної мережі, що визначається одним зі значень визначеного SocialNetworkType (рисунок 3.28) enum.

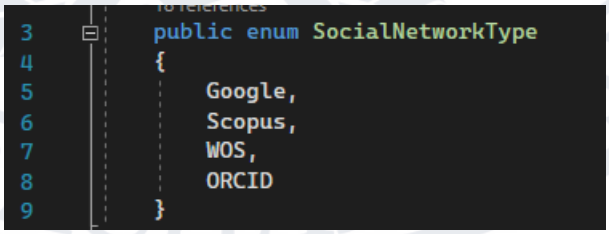



Рисунок 3.28 SocialNetworkType enum

Також визначено конструктор, який є єдиним способом створити об’єкт класу, і приймає обов’язкові вхідні параметри Scientist та NetworkType.

Додатково клас містить низку методів, які використовуються як помічники для роботи з інформацією класу та вченого. Для прикладу, розглянемо метод GetScientistSocialNetworkAccountId (рисунок 3.29), який використовується для отримання ідентифікатору профіля науковця в системі соціальної мережі.

```
2 references
private string GetScientistSocialNetworkAccountId()
{
    return NetworkType switch
    {
        SocialNetworkType.Google => new Uri(Value).Query.Split("&")
            .FirstOrDefault(parameter => parameter.Split("=")[0].Equals("?user")).Split("=")[1],
        SocialNetworkType.Scopus => new Uri(Value).Query.Split("&")
            .FirstOrDefault(parameter => parameter.Split("=")[0].Equals("?authorId")).Split("=")[1],
        SocialNetworkType.WOS => new Uri(Value).AbsolutePath.Split("/").Last(),
        SocialNetworkType.ORCID => new Uri(Value).AbsolutePath.Split("/").Last(),
        _ => throw new Exception(),
    };
}
```

Рисунок 3.29 метод GetScientistSocialNetworkAccountId

Метод використовує конструкцію мови switch case, для надання конкретної реалізації відносно кожного з існуючих типів соціальних мереж. На прикладі соціальної мережі WOS, можна побачити, що значення унікального ідентифікатору науковця отримується як останній елемент шляху посилання на соціальну мережу. Реальний приклад – «https://www.webofscience.com/wos/author/record/1149062» де шляхом є «/wos/author/record/1149062» а унікальним ідентифікатором користувача «1149062», тобто останній з елементів шляху.

Решта методів-помічників може бути переглянута у кодовій базі створеного програмного рішення. На цьому огляд директорії AdditionalModels завершено, тож перейдемо до розгляду директорії Helpers (рисунок 3.30).

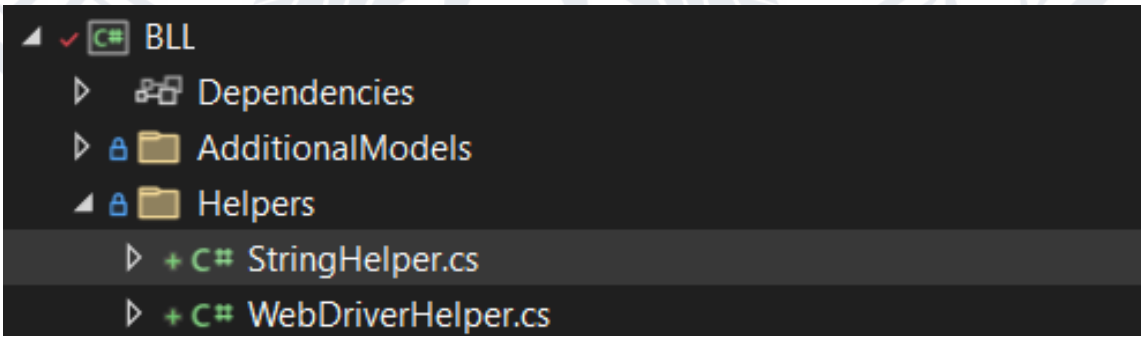



Рисунок 3.30 директорія BLL.Helpers

Директорія Helpers містить реалізації для класів-помічників базових типів розробленого додатку, зокрема базового типу string та інтерфейсу IWebDriver з простору імен WebDriver. Розглянемо клас WebDriverHelper (рисунок 3.31) як приклад.

```
public static class WebDriverHelper
{
    10 references
    public static IWebElement FindElement(this IWebDriver driver, By by, int timeoutInSeconds)
    {
        if (timeoutInSeconds > 0)
        {
            var wait = new WebDriverWait(driver, TimeSpan.FromSeconds(timeoutInSeconds));
            return wait.Until(drv => drv.FindElement(by));
        }
        return driver.FindElement(by);
    }

    6 references
    public static ReadOnlyCollection<IWebElement> FindElements(this IWebDriver driver, By by, int timeoutInSeconds)
    {
        if (timeoutInSeconds > 0)
        {
            var wait = new WebDriverWait(driver, TimeSpan.FromSeconds(timeoutInSeconds));
            return wait.Until(drv => drv.FindElements(by));
        }
        return driver.FindElements(by);
    }
}
```

Рисунок 3.31 клас-помічник WebDriverHelper

Клас реалізує два методи розширення для інтерфейсу IWebDriver, які розширюють базовий функціонал наданий бібліотекою WebDriver, додаючи логіку очікування при пошуку елементу/елементів, для уникнення помилок викликаних часом завантаження сторінки. Методи FindElement та FindElements містять однакову реалізацію, за виключенням того, що метод FindElements виконує множинний пошук низки елементів, а метод FindElement використовується для пошуку одного конкретного елементу.

Як можна бачити на рисунку 3.31 методи створено з ключовим словом static, а першим параметром є звернення до IWebDriver як до конкретного об'єкту класу, що є реалізацією методу розширення для класу, і дозволяє доповнити базовий функціонал класу додатковими методами, які будуть викликатись так само як і рідні методи класу.

На цьому розгляд директорії BLL.Helpers завершено, переглянути решту методів, та клас StringHelper зокрема, можна у кодовій базі створеного програмного рішення.

Наступною директорією у проекті BLL є BLL.Parsers (рисунок 3.32), яка містить інтерфейси та реалізації які використовуються для пошуку, аналізу, обробки та збору інформації про науковців.

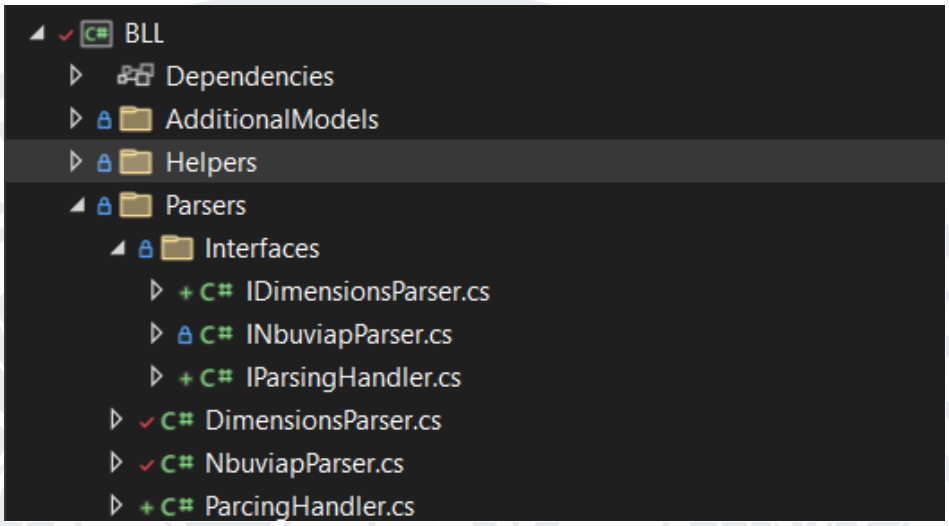


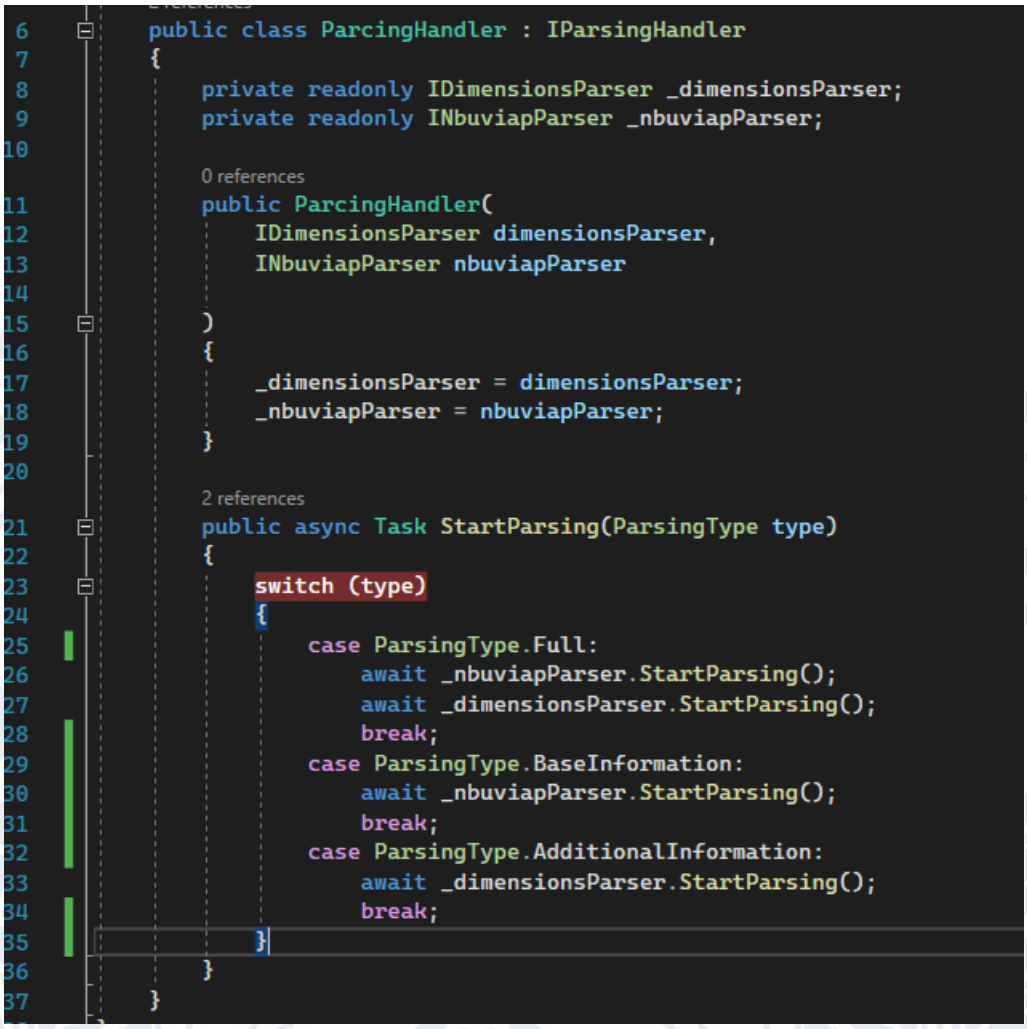

Рисунок 3.32 Директорія BLL.Parsers

Оскільки у директорії створено реалізації для сервісів, що використовуватимуться поза межами проекту BLL, для кожної з них створено інтерфейс. Отож почнемо розгляд з директорії BLL.Parsers.Interfaces, яка містить в собі три інтерфейси:

1. IParsingHandler – точка входу в процес парсингу, ініціалізований метод StartParsing приймає ParsingType як вхідний параметр, і починає виконання відповідного процесу парсингу.
2. INbuviapParser – створює єдиний метод, який буде доступний ззовні, для запуску процесу парсингу сайту http://nbuviap.gov.ua/bpnu/
3. IDimensionsParser – створює єдиний метод, який буде доступний ззовні, для запуску процесу парсингу сайту https://app.dimensions.ai/discover

Для кожного з перелічених інтерфейсів створено реалізації, які розміщено у корені директорії BLL.Parsers.

Отож розглянемо реалізації деяких з перелічених інтерфейсів, і почнемо з огляду основної точки взаємодії з процесом парсингу, а саме реалізації методу IParsingHandler.StartParsing у класі ParcingHandler (рисунок 3.33).



```
6   public class ParcingHandler : IParsingHandler
7   {
8       private readonly IDimensionsParser _dimensionsParser;
9       private readonly INbuviapParser _nbuviapParser;
10
        0 references
11      public ParcingHandler(
12          IDimensionsParser dimensionsParser,
13          INbuviapParser nbuviapParser
14
15      )
16      {
17          _dimensionsParser = dimensionsParser;
18          _nbuviapParser = nbuviapParser;
19      }
20
        2 references
21      public async Task StartParsing(ParsingType type)
22      {
23          switch (type)
24          {
25              case ParsingType.Full:
26                  await _nbuviapParser.StartParsing();
27                  await _dimensionsParser.StartParsing();
28                  break;
29              case ParsingType.BaseInformation:
30                  await _nbuviapParser.StartParsing();
31                  break;
32              case ParsingType.AdditionalInformation:
33                  await _dimensionsParser.StartParsing();
34                  break;
35          }
36      }
37  }
```

Рисунок 3.33 клас ParsingHandler

Клас ParsingHandler реалізує інтерфейс IParsingHandler визначаючи тіло для методу StartParsing, у якому використовуються дві залежності. Оскільки клас ParcingHandler використовує сервіси NbuviapParser та DimensionsParser – він є залежним від них і отримує їх реалізації з системи ін’єкції залежностей, що можна бачити в конструкторі класу в стрічках 11-19 на рисунку 3.33. Згодом методи визначених сервісів-залежностей використовуються у реалізації методу StartParsing(ParsingType type). Для кожного з визначених типів парсингу метод StartParsing виконує виклики певних методів класів-залежностей. Для виконання парсингу з типом Full відбувається виклик асинхронного методу StartParsing() сервісу NbuviapParser для збору базової інформації науковців з бібліометрики української науки, а після того як виконання методу завершилось, викликається аналогічний метод але уже для

сервісу DimensionsParser, який виконує парсинг додаткової інформації про науковців з ресурсу Dimensions.

Також визначено окремі реалізації для інших типів парсингу, зокрема BaseInformation виконує частковий парсинг, і збирає інформацію тільки з ресурсу Nbuviap, а AdditionalInfromation виконує парсинг інформації, для існуючих у базі даних науковців, з ресурсу Dimensions, пропускаючи етап збору базової інформації науковців.

Тепер же перейдемо до розгляду конкретної реалізації на прикладі сервісу NbuviapParser (рисунок 3.34) який реалізує інтерфейс INbuviapParser і його метод StartParsing.

```
11  public class NbuviapParser : INbuviapParser
12  {
13      private readonly IWebDriver _driver;
14      private readonly IFieldOfResearchRepository _fieldOfResearchRepository;
15      private readonly IScientistRepository _scientistRepository;
16      private readonly IOrganizationRepository _organizationRepository;
17
18      //bibliometrics - we are going to take scientist names + social networks + organization
19      private const string NbuviapURL = @"http://nbuviap.gov.ua/bpnu/index.php?page=search";
20
21      private const string ScientistsNamesElementsXPath = "//main/div/table/tbody/tr/td[3]";
22      private const string ScientistsOrganizationsXPath = "//table/tbody/tr/td[8]";
23      private const string ResearchElementXPath = "//table//tr/td[7]";
24      private const string SearchButtonXPath = "//input[@class='btn btn-primary mb-2']";
25      private const string NextPageButtonXPath = "//a[contains(.,'>>')]";
26      private const string FieldsOfResearchXPath = "//*[@id=\"galuz1\"]/option";
27
        0 references
28      public NbuviapParser(
29          IWebDriver driver,
30          IFieldOfResearchRepository fieldOfResearchRepository,
31          IScientistRepository scientistRepository,
32          IOrganizationRepository organizationRepository
33      )...
40
        3 references
41      public async Task StartParsing()...
84
85      /// <summary>
86      /// Creates missing fields of research and all found
87      /// </summary>
88      /// <returns></returns>
        1 reference
89      private async Task<List<string>> GetFieldsOfReserach()...
108
        1 reference
109     private async Task AddScientistData((string ScientistName, string FieldOfResearchTitle, string OrganizationTitle) scientistData)...
139
        1 reference
140     private async Task ParseSocialNetworsAndRating(Scientist scientist)...
168
        1 reference
169     private string GetSocialUrl(string socialNetworkXPath)...
189
        1 reference
190     private async Task<Organization> GetOrCreateOrganization(string organizationTitle)...
202
        1 reference
203     private async Task<FieldOfResearch> GetOrCreateFieldOfResearch(string currentFieldOfResearch)...
222 }
```

Рисунок 3.34 клас NbuviapParser

На рисунку 3.34 можна побачити структуру класу NbuviapParser, без конкретних реалізацій методів, які будуть розглянуті більш детально пізніше. Загальна структура класу включає визначення приватних полів доступних

тільки для читання, які є сервісами-залежностями, і ініціалізуються в конструкторі класу (рисунок 3.35)

```
13      private readonly IWebDriver _driver;
14      private readonly IFieldOfResearchRepository _fieldOfResearchRepository;
15      private readonly IScientistRepository _scientistRepository;
16      private readonly IOrganizationRepository _organizationRepository;
17
        0 references
18      public NbuviapParser(
19          IWebDriver driver,
20          IFieldOfResearchRepository fieldOfResearchRepository,
21          IScientistRepository scientistRepository,
22          IOrganizationRepository organizationRepository
23      )
24      {
25          _driver = driver;
26          _fieldOfResearchRepository = fieldOfResearchRepository;
27          _scientistRepository = scientistRepository;
28          _organizationRepository = organizationRepository;
29      }
```

Рисунок 3.35 Сервіси-залежності та конструктор класу NbuviapParser

Додатково у класі визначено низку полів-констант (рисунок 3.36), одне з яких є посиланням на ресурс, а інші є представленнями XPath до необхідних елементів на сторінці веб-ресурсу.

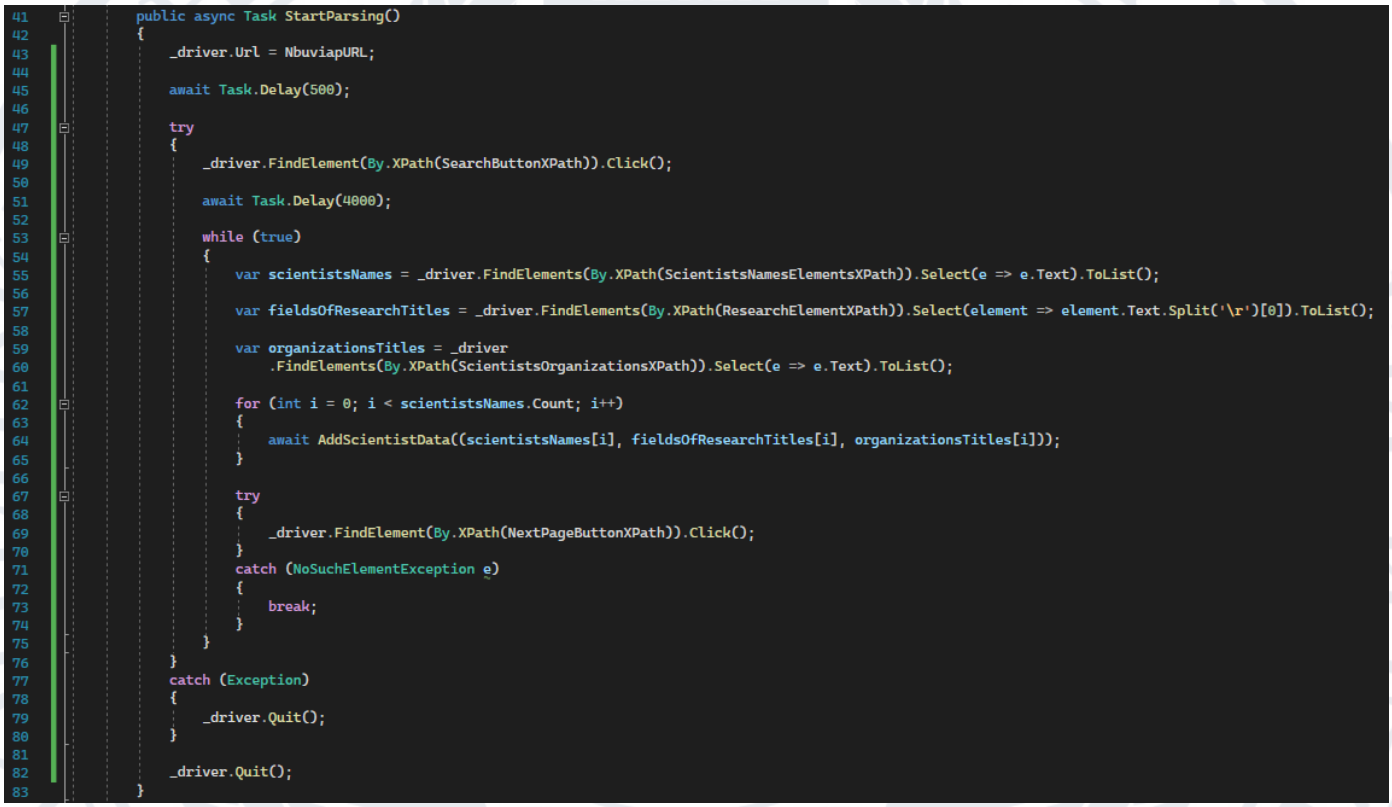
```
18      //bibliometrics - we are going to take scientist names + social networks + organization
19      private const string NbuviapURL = @"http://nbuviap.gov.ua/bpnu/index.php?page=search";
20
21      private const string ScientistsNamesElementsXPath = "//main/div/table/tbody/tr/td[3]";
22      private const string ScientistsOrganizationsXPath = "//table/tbody/tr/td[8]";
23      private const string ResearchElementXPath = "//table//tr/td[7]";
24      private const string SearchButtonXPath = "//input[@class='btn btn-primary mb-2']";
25      private const string NextPageButtonXPath = "//a[contains(.,'>>')]";
26      private const string FieldsOfResearchXPath = "//*[@id=\"galuz1\"]/option";
```

Рисунок 3.36 Поля-константи класу NbuviapParser

Окрім того, клас містить реалізацію для методу StartParsing (рисунок 3.37) наслідуваного від інтерфейсу INbuviapParser та низку приватних методів, які використовуються у процесі парсингу:

1. GetFieldsOfResearch – отримує напрями наукової роботи з інтернет ресурсу та створює ті з них, яких не існує в базі даних.
2. AddScientistData – перевіряє чи існує запис про науковця у базі даних, і створює його, якщо запису не знайдено.
3. ParseSocialNetworsAndRating – отримує соціальні мережі та HRating науковця та доповнює дані науковця цією інформацією

4. GetSocialUrl – метод-помічник для отримання посилання на профіль соціальної мережі науковця
5. GetOrCreateOrganization – повертає об’єкт організації, якщо такий існує в базі даних, інакше створює новий запис та повертає його.
6. GetOrCreateFieldOfResearch – повертає об’єкт напряму наукової роботи, якщо такий існує в базі даних, інакше створює новий запис та повертає його.



```
41      public async Task StartParsing()
42      {
43          _driver.Url = NbuviapURL;
44
45          await Task.Delay(500);
46
47          try
48          {
49              _driver.FindElement(By.XPath(SearchButtonXPath)).Click();
50
51              await Task.Delay(4000);
52
53              while (true)
54              {
55                  var scientistsNames = _driver.FindElements(By.XPath(ScientistsNamesElementsXPath)).Select(e => e.Text).ToList();
56
57                  var fieldsOfResearchTitles = _driver.FindElements(By.XPath(ResearchElementXPath)).Select(element => element.Text.Split('\r')[0]).ToList();
58
59                  var organizationsTitles = _driver
60                      .FindElements(By.XPath(ScientistsOrganizationsXPath)).Select(e => e.Text).ToList();
61
62                  for (int i = 0; i < scientistsNames.Count; i++)
63                  {
64                      await AddScientistData((scientistsNames[i], fieldsOfResearchTitles[i], organizationsTitles[i]));
65                  }
66
67                  try
68                  {
69                      _driver.FindElement(By.XPath(NextPageButtonXPath)).Click();
70                  }
71                  catch (NoSuchElementException e)
72                  {
73                      break;
74                  }
75              }
76          }
77          catch (Exception)
78          {
79              _driver.Quit();
80          }
81
82          _driver.Quit();
83      }
```

Рисунок 3.37 Метод StartParsing класу NbuviapParser

Метод StartParsing є єдиним доступним ззовні методом, оскільки його створено з модифікатором доступу public. Саме тут відбувається початок процесу парсингу інформації, зокрема у стрічкці 43 ми можемо бачити присвоєння посилання на необхідний ресурс до властивості Url об'єкта класу WebDriver, та очікування завантаження початкової сторінки у стрічці 45.

Після цього у блокові try відбувається натиск на кнопку пошуку, у стрічці 49, на головній сторінці веб-ресурсу і очікування завантаження інформації, у стрічці 51. Коли інформація є доступною, а сторінка повністю завантажилась, відбувається ітеративний процес проходження по всім

доступним сторінкам за допомогою циклу while, останньою операцією у якому є переключення на наступну сторінку, якщо така доступна. У стрічках 55-59 відбувається отримання інформації з поточної сторінки, зокрема отримуються елементи, що містять імена науковців, їх напрями наукової діяльності та організації. Коли дані отримано, вони обробляються та зберігаються у методі AddScientistData (рисунок 3.38).

```
109     private async Task AddScientistData((string ScientistName, string FieldOfResearchTitle, string OrganizationTitle) scientistData)
110     {
111         var scientistName = StringHelper.GetScientistName(scientistData.ScientistName);
112         var fieldOfResearch = await GetOrCreateFieldOfResearch(scientistData.FieldOfResearchTitle);
113         var organization = await GetOrCreateOrganization(scientistData.OrganizationTitle);
114
115         var scientist = new Scientist()
116         {
117             Name = scientistName,
118             Organization = organization,
119             ScientistFieldsOfResearch = new List<ScientistFieldOfResearch>
120                 {
121                     new ScientistFieldOfResearch()
122                     {
123                         FieldOfResearch = fieldOfResearch
124                     }
125                 }
126         };
127
128         var foundResult = await _scientistRepository.GetAsync(scientistName);
129
130         if (foundResult is null)
131         {
132             await ParseSocialNetworsAndRating(scientist);
133
134             await _scientistRepository.CreateAsync(scientist);
135         }
136     }
```

Рисунок 3.38 Реалізація методу AddScientistData

У стрічці 111 метод-помічник GetScientistName, з раніше розглянутого класу StringHelper, використовується для обробки тексту, що містить ім’я науковця, і отримання лише імені. У стрічках 112 та 113 викликаються методи GetOrCreateFieldOfResearch та GetOrCreateOrganization, що повертають відповідні існуючі або новостворені об’єкти цих типів.

Коли всі необхідні дані отримано та оброблено, у стрічках 115-126, відбувається ініціалізація нового об’єкту класу Scientist, та присвоєння йому отриманих даних. Використовуючи новостворений об’єкт та метод GetAsync класу ScientistRepository, що був розглянутий раніше, відбується пошук існуючого запису про науковця по його імені, якщо такого запису не знайдено – у стрічці 132 викликається метод ParseSocialNetworksAndRating який розширює новостворений об’єкт та додає у нього дані про соціальні мережі науковця та його HRating. Наостанок, використовуючи метод CreateAsync

класу ScientistRepository відбувається збереження створеного об’єкту науковця до бази даних.

Всі ці дії повторюються ітеративно для кожного зі знайдених науковців, на всіх доступних сторінках виконаного пошуку на ресурсі Nbuviap.

Реалізація парсингу для ресурсу Dimensions використовує схожий підхід до збору даних, але є модифікованою відповідно до поставлених задач та особливостей ресурсу, її можна переглянути у кодовій базі створеного програмного рішення.

На цьому розгляд сервісів що реалізують логіку парсингу інформації з веб-ресурсів завершено, і ми переходимо до наступної і останньої Services (рисунок 3.39) директорії у проекті BLL.

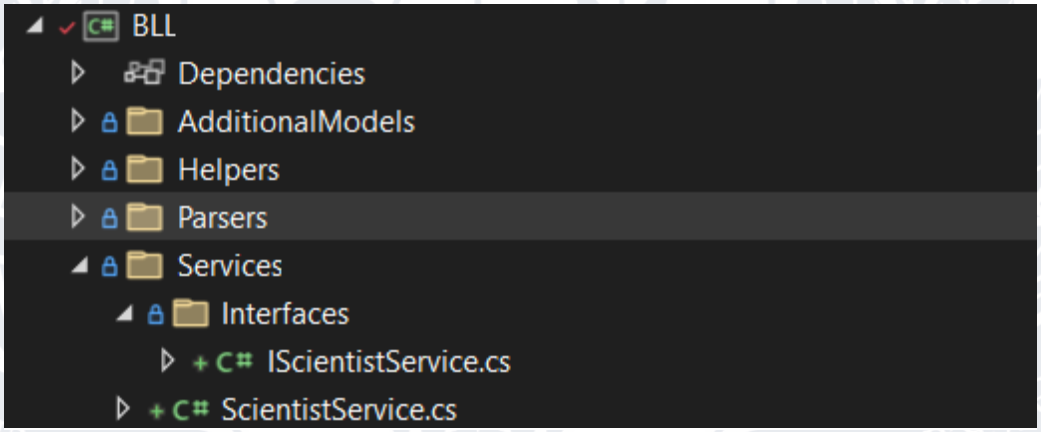
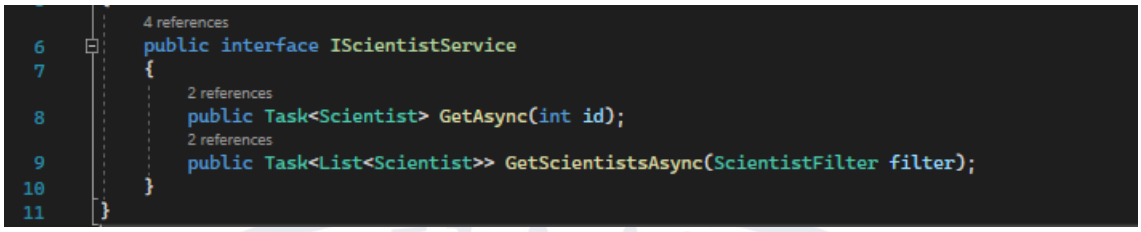



Рисунок 3.39 Директорія BLL.Services

Як можна бачити на рисунку 3.39, структура директорії є такою ж як і структура директорії BLL.Parsers (рисунок 3.32), і включає в себе реалізації сервісів у корені директорії, та вкладену директорію BLL.Servies.Interfaces в якій знаходяться інтерфейси для створених сервісів.

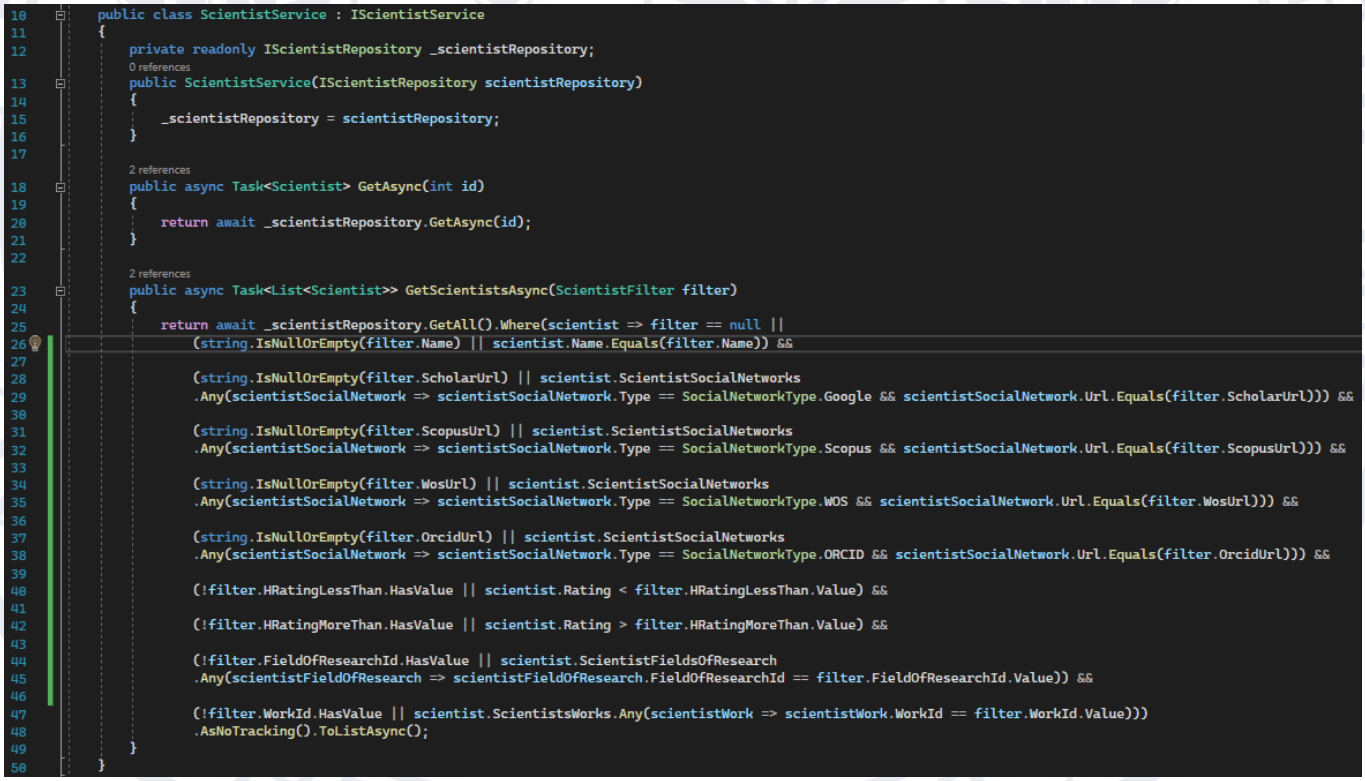
У директорії створено інтерфейси та сервіси які покликані надавати конкретну реалізацію для домейн логіки проекту, зокрема розширюючи можливості для роботи з базою даних, а також виступаючи у ролі проксі між проектами API та DAL, отож розглянемо реалізацію такого сервісу на прикладі ScientistService і почнемо зі створеного для нього інтерфейсу IScientistService (рисунок 3.40).



```
4 references
public interface IScientistService
{
    2 references
    public Task<Scientist> GetAsync(int id);
    2 references
    public Task<List<Scientist>> GetScientistsAsync(ScientistFilter filter);
}
```

Рисунок 3.40 Інтерфейс IScientistService

У інтерфейсі визначено два методи, які будуть реалізовані у класі нащадкові ScientistService (рисунок 3.41), та використовуватимуться у проекті API. Перший метод – отримання науковця використовуючи унікальний ідентифікатор запису в базі даних, другий метод – отримання масиву об’єктів науковців використовуючи об’єкт фільтра для моделі науковців (рисунок 3.7). Ці методи виконують логіку яка використовується кінцевими точками визначеними у ScientistController на рівні API.



```
public class ScientistService : IScientistService
{
    private readonly IScientistRepository _scientistRepository;
    0 references
    public ScientistService(IScientistRepository scientistRepository)
    {
        _scientistRepository = scientistRepository;
    }

    2 references
    public async Task<Scientist> GetAsync(int id)
    {
        return await _scientistRepository.GetAsync(id);
    }

    2 references
    public async Task<List<Scientist>> GetScientistsAsync(ScientistFilter filter)
    {
        return await _scientistRepository.GetAll().Where(scientist => filter == null ||
            (string.IsNullOrEmpty(filter.Name) || scientist.Name.Equals(filter.Name)) &&

            (string.IsNullOrEmpty(filter.ScholarUrl) || scientist.ScientistSocialNetworks
            .Any(scientistSocialNetwork => scientistSocialNetwork.Type == SocialNetworkType.Google && scientistSocialNetwork.Url.Equals(filter.ScholarUrl))) &&

            (string.IsNullOrEmpty(filter.ScopusUrl) || scientist.ScientistSocialNetworks
            .Any(scientistSocialNetwork => scientistSocialNetwork.Type == SocialNetworkType.Scopus && scientistSocialNetwork.Url.Equals(filter.ScopusUrl))) &&

            (string.IsNullOrEmpty(filter.WosUrl) || scientist.ScientistSocialNetworks
            .Any(scientistSocialNetwork => scientistSocialNetwork.Type == SocialNetworkType.WOS && scientistSocialNetwork.Url.Equals(filter.WosUrl))) &&

            (string.IsNullOrEmpty(filter.OrcidUrl) || scientist.ScientistSocialNetworks
            .Any(scientistSocialNetwork => scientistSocialNetwork.Type == SocialNetworkType.ORCID && scientistSocialNetwork.Url.Equals(filter.OrcidUrl))) &&

            (!filter.HRatingLessThan.HasValue || scientist.Rating < filter.HRatingLessThan.Value) &&

            (!filter.HRatingMoreThan.HasValue || scientist.Rating > filter.HRatingMoreThan.Value) &&

            (!filter.FieldOfResearchId.HasValue || scientist.ScientistFieldsOfResearch
            .Any(scientistFieldOfResearch => scientistFieldOfResearch.FieldOfResearchId == filter.FieldOfResearchId.Value)) &&

            (!filter.WorkId.HasValue || scientist.ScientistsWorks.Any(scientistWork => scientistWork.WorkId == filter.WorkId.Value)))
            .AsNoTracking().ToListAsync();
    }
}
```

Рисунок 3.41 Клас ScientistService

Клас ScientistService є нащадком інтерфейсу IScientistService і реалізує обидва створені у ньому методи для отримання даних GetAsync(int id) та GetScientistsAsync(ScientistFilter filter). Реалізація методу GetAsync(int id) використовує метод GetAsync сервісу ScientistRepository, який було отримано

з системи ін’єкції залежностей у конструкторі сервісу, для пошуку і повернення моделі представлення науковця (рисунок 3.8), даний метод є проксі методом між API та DAL сервісами і в перспективі може мати розширену реалізацію, що включатиме перевірку вхідних параметрів, конвертацію між внутрішніми (domain) та зовнішніми (DTO) моделями, реалізацію повернення даних за сторінками (pagination).

Основний же функціонал, що реалізований у класі ScientistService, знаходиться у методі GetScientistsAsync, який використовує метод GetAll класу ScientistRepository для отримання IQueryable типу, що відноситься до моделі Scientist. У методі відбувається модифікація IQueryable об’єкту з використанням технології LINQ (language integrated query) та включення умов отримання даних. Для прикладу, у стрічці 26 на рисунку 3.41 можна побачити умову виконання фільтрації науковців за іменем, у випадку якщо поле ім’я було передано зі значенням в об’єкті фільтра. На додачу розглянем ще кілька реалізованих фільтрацій:

1. Фільтрація за посиланням на соціальну мережу – у випадку передачі значення у полі фільтра для конкретної соціальної мережі відбувається перевірка співпадіння посилання на конкретну соціальну мережу науковця зі значення переданим у об’єкті фільтра. Фільтрації за соціальною мережею є подібними одна до одної і розташовані у стрічках 28-35.
2. Фільтрація за значення HRating має дві доступні опції:
    a. HRatingMoreThan – отримання науковців чий рейтинг більший за передане значення
    b. HRatingLessThan – отримання науковців чий рейтинг менший за передане значення

    Ці опції можна комбінувати, для отримання науковців чий рейтинг знаходиться у певному проміжку, для прикладу при передачі HRatingMoreThan = 100 та HRatingLessThan = 150 ми отримаємо

масив об’єктів науковців чий рейтинг знаходиться у проміжку між 100 та 150 одиниць.

3. Фільтрація за значенням FieldOfResearchId дозволяє виконати пошук науковців за переданим унікальним ідентифікатором напряму наукової діяльності, що наявний у базі даних
4. Фільтрація за значенням WorkId дозволяє виконати пошук науковців за переданим унікальним ідентифікатором наукової роботи, яка наявна у базі даних.

Після виконання фільтрацій відбувається виклик методу AsNoTracking, який зумовлений прагненням пришвидшити роботу системи, і вимикає відслідковування змін у повернених об’єктах, оскільки метод не призначений для використання при модифікації даних, а слугує лише для отримання даних. Останнім же викликається метод ToListAsync, виклик якого і розпочинає процес конвертації IQueryable до запиту у базу даних, а також виконання самого запиту та повернення даних.

На цьому розгляд рівня BLL завершено, наступним буде розглянуто останній проект розробленого програмного рішення API.

### 3.4.6 Рівень API

Рівень, який реалізує кінцеві точки, та слугує точкою входу в додаток, а також є проектом який ініціалізує та конфігурує всі інші сервіси та залежності – рівень API (рисунок 3.42).

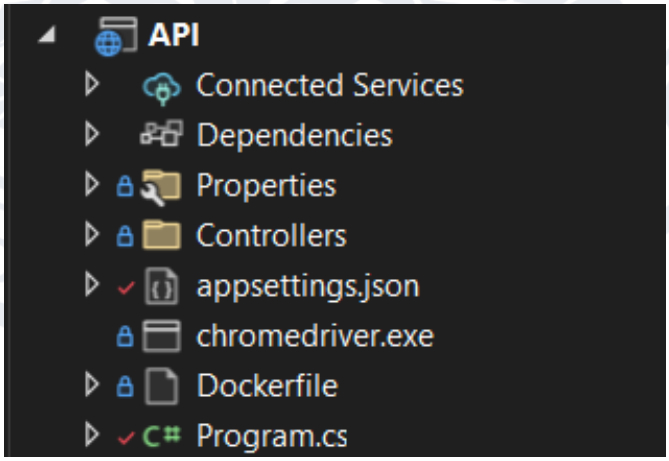



Рисунок 3.42 Рівень API

Структура проекту API включає в себе:

1. Директорію Controllers, яка містить реалізації контроллерів та кінцевих точок, використовуючи які відбувається взаємодія з додатком.
2. Файл appsettings.json, де зберігається конфігурація проекту у форматі json.
3. Chromedriver.exe – файл драйверу для роботи з браузером Google Ghrome
4. Dockerfile – файл конфігурації контейнера Docker, який необхідний для розгортання створеного програмного рішення у контейнері Docker.
5. Клас Program.cs – який є точкою початку виконання додатку, а також проводить підключення, реєстрацію та налаштування всіх необхідних сервісів та залежностей.

Почати розгляд проекту API варто з файлу конфігурації проекту appettings.json (рисунок 3.43), що включає в себе базові налаштування логування, стрічку підключення до бази даних, яка є унікальною і залежить від середовища де було розгорнуто рішення, а також конфігурацію AllowedHost, яка контролює доступ до додатку ззовні.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "ConnectionStrings": {
    "ParserDB": "Host=localhost;Port=5432;Database=parserDB;Username=postgres;Password=[redacted]"
  },
  "AllowedHosts": "*"
}
```

Рисунок 3.43 Файл конфігурації додатку appsettings.json

Наступним кроком буде розглянуто клас Program.cs (рисунок 3.44), в якому виконується реєстрація сервісів та конфігурація додатку, цей файл також називають файлом Startup, оскільки він викликається під час запуску додатку. Більшість налаштувань, які були зроблені, є стандартними та виконують реєстрацію та налаштування базових сервісів.

Реалізацію методу Main можна умовно розділити на 2 частини:

1. Реєстрація сервісів, яка відбувається у стрічках 10-26
2. Конфігурація сервісів, яка відбувається у стрічках 31-41

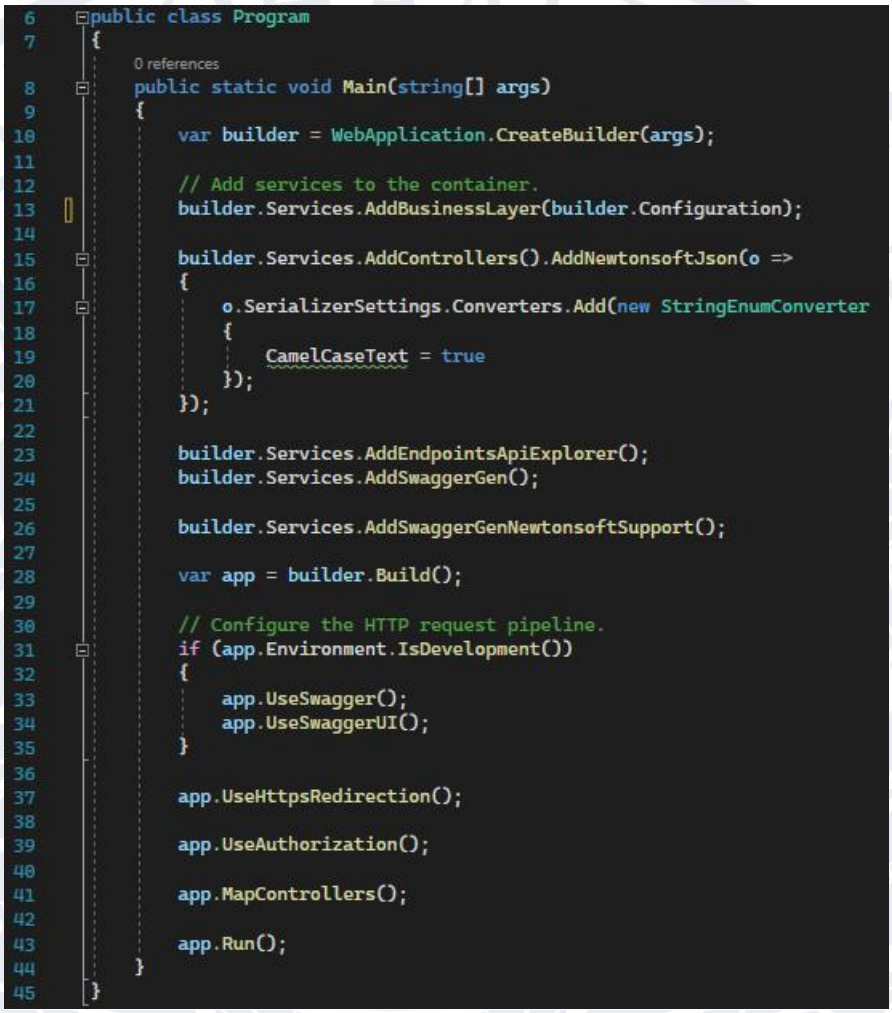



Рисунок 3.44 Клас Program.cs

У десятій стрічці відбувається виклик методу CreateBuilder класу WebApplication, який ініціалізує базові сервіси та повертає об’єкт класу WebApplicationBuilder, використовуючи який ми можемо зареєструвати власні сервіси та додаткові базові сервіси. Зокрема, об’єкт builder використовується для виклику методу розширення AddBusinessLayer, розглянутого раніше у розділі 3.4.3. У стрічках 15-26 реєструються додаткові базові сервіси, зокрема контроллери, що створені у проекті API, та Swagger – інтерфейс взаємодії з додатком. У стрічці 28 відбувається виклик методу Build класу WebApplicationBuilder, який повертає об’єкт класу WebApplication, який в

подальшому використовується для додаткового налаштування зареєстрованих сервісів у стрічках 31-41. Коли налаштування завершено відбувається запуск додатку викликом методу Run класу WebApplication, у стрічці 43.

На останок розглянемо створені контроллери(рисунок 3.45), які включають у себе налаштування шляхів та створення кінцевих точок веб-додатку.

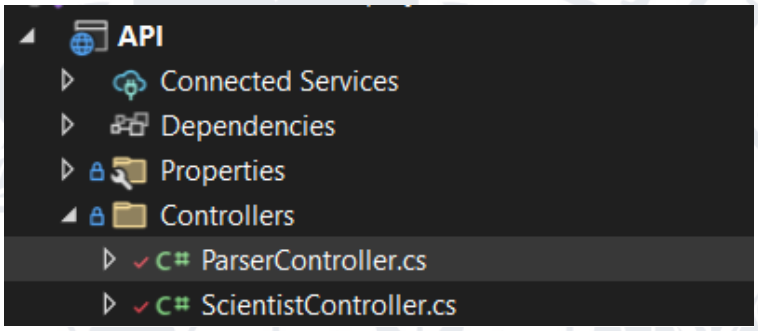



Рисунок 3.45 Директорія Controllers на рівні API

Директорія Controllers містить два реалізованих контроллери, які уже були розглянуті зі сторони користувача у розділі 3.3.2, тепер же ми розглянемо їх зі сторони розробника.

Створені контроллера можна умовно розділити на два логічні блоки:

1. Виконання бізнес логіки проекту – за цю частину відповідає ParserController.
2. Маніпуляції з даними – кінцеві точки для цих операцій знаходяться у ScientistController.

Процес парсингу інформації запускається викликом кінцевої точки StartParser у класі ParserController (рисунок 3.46), з передачею обов’язкового параметра ParsingType type, який згодом використовується для запуску конкретного процесу парсингу. Окрім того, всі контроллери наслідують базовий клас ContollerBase, де реалізується базова логіка і конфігурація класу типу контроллер. У стрічці дев’ять використано атрибут рівню класу для визначення шляху, за яким контроллер буде доступним, у розробленому додатку шлях до контроллера це його назва без слова Controller, то ж для ParserController – шляхом є “{baseUrl}{Parser}”.

```
8       [ApiController]
9       [Route("[controller]")]
        1 reference
10      public class ParserController : ControllerBase
11      {
12          private readonly IParsingHandler _parsingHandler;
13
            0 references
14          public ParserController(IParsingHandler parsingHandler)
15          {
16              _parsingHandler = parsingHandler;
17          }
18
19
20          [HttpGet("StartParser")]
            0 references
21          public async Task<IActionResult> StartParser(ParsingType type)
22          {
23              await _parsingHandler.StartParsing(type);
24              return Ok("Parsing succesfully started");
25          }
26      }
```

Рисунок 3.46 клас ParserController

Для вказання шляху до конкретної кінцевої точки використано атрибут рівня методу у стрічці сорок три, який вказує на тип використовуваного HTTP методу та, як параметр, приймає шлях за яким кінцева точка буде доступна, то ж на прикладі кінцевої точки StartParser шляхом буде GET “{baseUrl}{Parser}{StartParser}”.

Окрім того ParserController використовує розглянутий раніше у розділі 3.4.5 сервіс IParsingHandler для взаємодії з BLL та виклику метода StartParsing, що реалізує логіку перебору типів парсингу та виклик необхідної логіки.

Контроллер ScientistController (3.47) реалізовано за тим же принципом, що і ParserController, за відміною використовуваних залежностей та реалізованих кінцевих точок.

На відміну від ParserController у контроллері ScientistController єдиною залежністю є сервіс ScientistService, реалізація якого була розглянута у розділі 3.4.5, і саме його методи викликаються для виконання необхідної бізнес логіки. Крім того, кінцева точка GetScientistsAsync створена з використанням атрибуту рівня методу HttpPost, що вказує на використовуваний тип HTTP методу Post, який дозволяє передавати вхідні параметри у вигляді body методу.

```
[ApiController]
[Route("[controller]")]
1 reference
public class ScientistController : ControllerBase
{
    private readonly IScientistService _scientistService;

    0 references
    public ScientistController(IScientistService scientistService)
    {
        _scientistService = scientistService;
    }

    /// <summary>
    /// Get specific scientist using Scientist.Id to search
    /// </summary>
    /// <param name="id"></param>
    /// <returns></returns>
    [HttpGet("{id}")]
    0 references
    public async Task<IActionResult> Get(int id)
    {
        var user = await _scientistService.GetAsync(id);

        return user is null ? BadRequest($"User with Id = {id} not found") : Ok(user);
    }

    /// <summary>
    /// Get scientists list using filtering options in the filter object
    /// </summary>
    /// <param name="filter">Used to provide filtering options</param>
    /// <returns>List of scientists that were found</returns>
    [HttpPost]
    0 references
    public async Task<ActionResult<IEnumerable<Scientist>>> GetScientistsAsync(ScientistFilter filter)
    {
        return Ok(await _scientistService.GetScientistsAsync(filter));
    }
}
```

Рисунок 3.47 Клас ScientistController

Як уже було сказано, клас ScientistController використовує клас ScientistService для роботи з бізнес логікою програмного рішення, змінна створюється з типом інтерфейсу IScientistService і ініціалізується реалізацією інтерфейсу, яка була зареєстрована раніше, отриманою з сервісу ін’єкції залежностей .

Контроллер надає реалізацію для двох кінцевих точок:

1. GetAsync(int id) – що використовується для отримання конкретного науковця за унікальним ідентифікатором запису у базі даних, та використовує метод GetAsync класу ScientistService для виконання запиту.
2. GetScientistsAsync(ScientistFilter filter) – що використовується для отримання масиву науковців використовуючи дані передані у вхідному параметрі як опції фільтрації, та використовує метод GetScientistsAsync класу ScientistService для виконання запиту.

## 3.5 Висновок

У розділі розглянуто створений, під час виконання наукової роботи, додаток, який дозволяє збирати, зберігати, аналізувати та отримувати інформації про науковців.

Розглянуто архітектуру створеної бази даних, таблиці та зв’язки між сутностями, а також типи даних та встановлені обмеження. Описано тип даних, що зберігаються а також призначення створених таблиць.

Надано детальний опис створеного інтерфейсу взаємодії з додатком, розглянуто створені контроллери, їх призначення та надані можливості. Також описано створені кінцеві точки, а саме їх назви, вхідні та вихідні дані, обмеження та використані шляхи.

Проведено детальний розгляд всіх складових системи створеного програмного рішення, розглянуто використані технології та підходи, реалізовані патерни програмування та загальновизнані практики створення програмних продуктів типу веб-додатків.

Розглянуто підхід до роботи з базою даних, з використанням інструменту EntityFramework, який використовується для взаємодії з базою даних, внесення змін а також контролю версій.

Розглянуто підхід використаний підхід реєстрації та ін’єкції залежностей у системі, а також поняття життєвого циклу.

Розглянуто рівень DAL, який є реалізує логіку взаємодії з базою даних та надає інтерфейси для використання реалізованих методів у інших частинах додатку, а також відповідає за контроль версій бази даних та її структуру.

Розглянуто рівень BLL, який є зв’язною ланкою між рівнями API та DAL та реалізує основну бізнес логіку розробленого програмного рішення, а також розширює реалізацію методів рівня DAL.

Розглянуто рівень API, який є найвищим рівнем системи і в той же час точкою входу і старту додатку, реалізує контракти та інтерфейси взаємодії з ззовні, а також виконує функції створення, ініціалізації та налаштування всіх складових створеної системи.

Розроблений додаток є гнучким та легко піддається внесенню змін та масштабуванню. При цьому, варто відмітити, що обраний варіант реалізації не є відмовостійким у довгостроковій перспективі, оскільки створена логіка працює з чітко визначеним користувацьким інтерфейсом джерел, що може призвести до очікуваних проблем у майбутньому, за умови змін у користувацькому інтерфейсі джерела.

# ВИСНОВКИ

Під час виконання магістерської роботи було розроблено програмний продукт, що реалізує можливість збору, аналізу, збереження та маніпулювання даними про науковців, які отримані з відкритих джерел мережі інтернет.

Для досягнення поставленої мети було:

1. здійснено аналіз існуючих варіантів рішень задач пошуку, аналізу, збору та збереження інформації;
2. проаналізовано літературні джерела на відповідну тематику;
3. обрано засоби та інструменти розробки програмного продукту;
4. обрано методи розробки;
5. розроблено архітектуру програмного забезпечення та бази даних;
6. створено програмну реалізацію продукту з використанням обраних технологій програмування та підходів.

Розроблений програмний продукт включає в себе наступний функціонал:

1. адміністрування, оновлення, та робота з базою даних;
2. збір інформації з відкритих джерел мережі інтернет;
3. попередній аналіз та обробка знайденої інформації;
4. збереження підготовленої інформації у спроектовану та створену базу даних;
5. доступ до існуючої у базі даних інформації, а також її модифікація, вилучення та отримання;
6. можливість взаємодії зі створеним програмним продуктом через виклики кінцевих створених кінцевих точок, або через наданий інтерфейс Swagger;
7. можливість легкого та швидкого зміни підходу до розробки, структури або ж архітектури додатку, завдяки використанню абстракцій та низькій зв’язності компонентів створеного програмного рішення.

По завершенню наукової роботи, отримано програмний продукт, побудований з використанням модерних технологій, сучасних підходів та практик, який дозволяє здійснювати пошук, збереження та маніпулювання даними. Розроблений продукт відповідає усім поставленим вимогам та реалізує всі поставлені задачі.

## СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ ТА ЛІТЕРАТУРИ


1. C# documentation [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/
2. .Net documentation [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/
3. The Selenium Browser Automation Project [Електронний ресурс] – Режим доступу до ресурсу: https://www.selenium.dev/documentation/
4. The Selenium Browser Automation Project [Електронний ресурс] – Режим доступу до ресурсу: https://www.selenium.dev/documentation/
5. PostgreSQL documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.postgresql.org/docs/
6. Entity Framework documentation [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/ef/
7. Dependency injection in ASP.NET Core [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection?view=aspnetcore-7.0
8. Design Patterns [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/design-patterns
9. Repository Design Pattern In ASP.NET MVC [Електронний ресурс] – Режим доступу до ресурсу: https://www.c-sharpcorner.com/article/repository-design-pattern-in-asp-net-mvc/
10. Entity Framework documentation [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/ef/
11. Scraping Dynamic Web Pages Using Selenium And C# [Електронний ресурс] – Режим доступу до ресурсу: https://www.lambdatest.com/blog/scraping-dynamic-web-pages/

12. Code First to a New Database [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database
13. Npgsql Entity Framework Core Provider [Електронний ресурс] – Режим доступу до ресурсу: https://www.npgsql.org/efcore/
14. ASP.NET Core Performance Best Practices [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/performance/performance-best-practices?view=aspnetcore-7.0
15. SOLID Principles In C# [Електронний ресурс] – Режим доступу до ресурсу: https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/
16. C# clean code [Електронний ресурс] – Режим доступу до ресурсу: https://refactoring.guru/refactoring/what-is-refactoring

Подолян Микола Олександрович
Прізвище, ім’я, по-батькові
Факультет інформаційних і прикладних технологій
Факультет
122 «Комп’ютерні науки»
Шифр і назва спеціальності
Комп’ютерні технології обробки даних
Освітня програма

## ДЕКЛАРАЦІЯ АКАДЕМІЧНОЇ ДОБРОЧЕСНОСТІ

Усвідомлюючи свою відповідальність за надання неправдивої інформації, стверджую, що подана кваліфікаційна (магістерська) робота на тему: «Формування семантичної бази українських науковців на основі опрацювання їх публічних профілів» 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 отримувалась іншими особами, а також дані та інформація не отримувалась у недозволений спосіб.

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

_____ _____

(дата) (підпис здобувача освіти)