МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

**ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІМ. Ю.М. ПОТЕБНІ
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ**

**КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА
ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ**

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET**

Виконав: студент 2 курсу, групи 8.1212-пзс
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
А. Л. Сєров
(ініціали та прізвище)
Керівник доцент, к.ф.-м.н. Г. П.Коломоєць
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ «Алтер Віжн Груп»

В.С. Тряпічко
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІМ. Ю.М. ПОТЕБНІ
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра електроніки, інформаційних систем та програмного забезпечення
Рівень вищої освіти другий (магістерський)
Спеціальність 121 Інженерія програмного забезпечення
(код та назва)
Освітня програма Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри Т.В. Критська
"_02_" __вересня_____2023_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Сєров Арсеній Леонідович
(прізвище, ім'я, по батькові)

1.Тема роботи Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET

керівник роботи доцент, к.ф.-м.н. Коломоєць Геннадій Павлович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом ЗНУ від 09.10. 2023 р. №1577-с

2. Строк подання студентом кваліфікаційної роботи 30.11.2023
3. Вихідні дані магістерської роботи
- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень та аналогів;
- дослідження проблеми розпізнавання мов та розробка методів її вирішення;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- дослідження поставленої проблеми та розробка висновків та пропозицій.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
16 слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 02.109.2023

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником.	12.10-19.10.22	виконано
2	Літературний огляд. Перший розділ. Об'єктно-реляційне відображення та його значення для розробки програмного забезпечення.	19.10.22-31.12.22	виконано
3	Літературний огляд. Другий розділ. Огляд фреймворків ORM для платформи .NET.	31.12.22-01.02.23	виконано
4	Третій розділ. Вибір технологій проектування та особливості архітектури програмного забезпечення..	01.02.23-20.04.23	виконано
5	Представлення теоретичних результатів на конференції «Молода наука - 2023».	20.04.23	виконано
6	Розробка застосунка на мові C# для дослідження продуктивності ORM фреймворків: Dapper, NHibernate, Entity Framework Core.	21.04.23-30.09.23	виконано
7	Збір аналітичних даних за допомогою застосунка та програми Postman. Аналіз даних, побудова графічного представлення результатів. Формування висновків за результатами дослідження.	30.09.2023-20.10.23	виконано
8	Представлення практичних результатів дослідження на «Конференція молодих вчених - 2023».	21.10.23	виконано
9	Четвертий розділ. Розробка та використання ПЗ для проведення досліджень.	21.10.23 – 20.11.23	виконано
10	Оформлення звіту.	20.11.23-27.11.23	виконано

Студент _____ А.Л. Сєров
(підпис) (прізвище та ініціали)

Керівник роботи _____ Г.П. Коломоєць
(підпис) (прізвище та ініціали)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Сторінок: 130

Рисунків: 33

Таблиць: 18

Джерел: 33

Сєров А. Л. Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник Г. П. Коломоєць. Запоріжжя : ЗНУ, 2023. 130 с.

Мета і завдання дослідження полягають у порівняння фреймворків об'єктно-реляційних відображень (ORM), доступних для платформи Microsoft .NET, за чітко визначеними критеріями та надання рекомендацій щодо оптимального використання відповідних інструментів.

У процесі дослідження була розглянута проблема вибору ORM фреймворку на платформі Microsoft .NET, зроблено порівняння трьох найпоширеніших ORM фреймворків для цієї платформи: Dapper, NHibernate, Entity Framework Core за таким критеріями як за продуктивність, інструментарій, масштабованість, функціональність, документація та спільнота. Для оцінки та порівняння продуктивності ORM фреймворків було створено застосунок, на мові C#, на платформі .NET 6, з відкритим програмним інтерфейсом (API), який дозволив зібрати аналітичні дані для різних сценаріїв роботи з ORM фреймворками. За результатом аналізу зібраних даних - сформульовані і представлені рекомендації щодо використання досліджених ORM фреймворків.

Ключові слова: Об'єктно-реляційне відображення, мапінг, фреймворк, програмний інтерфейс, Microsoft .NET, Dapper, NHibernate, Entity Framework Core

SUMMARY

Pages: 130

Figures: 33

Tables: 18

Sources: 33

Sierov A.L. Comparative study of object-relational mapping frameworks for the Microsoft .Net platform: qualification work of the master of specialty 121 "Software Engineering" / Science head H.P. Kolomoiets. Zaporizhzhia: ZNU, 2023. 130 p.

The aim and objectives of the research are to compare object-relational mapping (ORM) frameworks available for the Microsoft .NET platform using clearly defined criteria, and to provide recommendations for the optimal use of the respective tools.

During the research process, the problem of selecting an ORM framework on the Microsoft .Net platform was addressed. A comparison was made between three of the most common ORM frameworks for this platform: Dapper, NHibernate, and Entity Framework Core, based on criteria such as performance, tooling, scalability, functionality, documentation and community support. To assess and compare the performance of the ORM frameworks, a test application was developed in C# on the .NET 6 platform with an open API, allowing for the collection of analytical data for various scenarios of working with ORM frameworks. Based on the analysis of the collected data, recommendations for the use of the researched ORM frameworks were formulated and presented.

Keywords: Object-relational mapping, mapping, framework, application programming interface, Microsoft .NET, Dapper, NHibernate, Entity Framework Core.

ЗМІСТ

ВСТУП

Актуальність теми

Об'єктно-реляційне відображення (ORM) є важливим елементом сучасного програмування, особливо в контексті високорівневої розробки. Як інструмент, що спрощує взаємодію з базами даних, ORM дозволяє розробникам зосередитися на основній бізнес-логіці своїх програм, забезпечуючи гнучкий і ефективний доступ до даних. Фреймворки ORM пропонують абстрактний підхід до маніпуляції даними, замість того, щоб прямо написати SQL-код, тим самим вони дозволяють розробникам працювати на більш високому рівні абстракції.

Тим не менше, важливим питанням, з яким стикаються розробники, є вибір конкретного фреймворку ORM. Для платформи Microsoft .NET існує широкий вибір таких фреймворків, включаючи три найрозповсюджених [3], але не обмежуючись, Entity Framework Core, NHibernate, Dapper. Кожен з цих фреймворків має свої особливості, переваги та недоліки. Наприклад, одні фреймворки пропонують більші можливості для налаштування та персоналізації, тоді як інші зосереджуються на простоті використання та швидкості розробки.

Актуальність теми полягає в тому, що вибір правильного ORM фреймворку може значно вплинути на продуктивність розробки та функціональність кінцевого продукту. Враховуючи, що програмна індустрія є ключовим елементом сучасної економіки, та важливою частиною економіки України, ця тема є особливо актуальною. Дослідження, спрямоване на аналіз та порівняння популярних ORM-фреймворків для платформи Microsoft .NET, може допомогти визначити, який з них найкраще підходить для конкретних вимог та обставин. Таке дослідження має на меті дати детальну оцінку кожного з цих фреймворків, розглядаючи такі аспекти, як швидкість виконання запитів до бази даних, можливості налаштування, зручність використання, сумісність та гнучкість. Важливо не тільки розглянути технічні аспекти кожного фреймворка, але і вра-

хувати їх вплив на процес розробки в цілому. Чи полегшує фреймворк розробку? Чи підвищує він продуктивність команди? Який вплив він має на якість кінцевого продукту?

Результати такого дослідження не тільки допоможуть розробникам при виборі найкращого фреймворка для їхніх потреб, але і сприятимуть розвитку програмної індустрії в Україні в цілому, підвищуючи рівень знань та розуміння цієї важливої області. Це є основною метою і основною мотивацією для проведення даного дослідження.

Мета і завдання дослідження

Мета цього дослідження - порівняння фреймворків об'єктно-реляційних відображень, доступних для платформи Microsoft .NET, та надання рекомендацій щодо оптимального використання відповідних інструментів.

Для досягнення цієї мети, дослідження має наступні задачі:

– визначення та основи об'єктно-реляційного відображення. Розуміння основ ORM є важливим для дослідження, оскільки це становить основу для розуміння, як фреймворки виконують свою роботу;

– огляд фреймворків ORM для платформи .NET. Включає огляд і порівняння ключових ORM фреймворків, доступних для платформи .NET, зокрема Entity Framework Core, NHibernate, Dapper;

– вибір технологій та особливості архітектури програмного забезпечення, розробленого у рамках роботи для проведення досліджень. Передбачає визначення і обґрунтування вибору технологій для дослідження, а також визначення вимог до програмного забезпечення, яке буде використано для аналізу фреймворків;

– створення моделі та запитів для порівняльного аналізу. Включає проектування бази даних і розробку основних запитів для використання з різними ORM фреймворками;

– розробка та використання програмного забезпечення. Завдання включає розробку алгоритмів і програмних модулів, а також використання програмного забезпечення з різними ORM фреймворками;

– розробка рекомендацій щодо використання досліджених фреймворків. Після аналізу результатів, будуть сформульовані і представлені рекомендації щодо використання досліджених ORM фреймворків.

Об’єкт дослідження

Процес розробки програмного забезпечення на платформі Microsoft .NET з використанням об'єктно-реляційного відображення.

Предмет дослідження

Предметом дослідження є фреймворки ORM для платформи Microsoft .NET, включаючи Entity Framework Core, NHibernate, Dapper. Використання та порівняння цих фреймворків в контексті поставлених вимог та сценаріїв розробки програмного забезпечення.

Методи дослідження

Методи, що були спрямовані на вирішення поставлених задач та досягнення цілей дослідження:

– аналіз особливостей об'єктно-реляційного відображення. Було використано дослідження суті ORM, його ролі в програмуванні, стандартів ORM в .NET, та переваг та недоліків їх використання;

– аналіз ORM фреймворків для платформи .NET. Здійснено огляд популярних серед розробників ORM фреймворків доступних для .NET: Entity Framework Core, NHibernate та Dapper;

– аналіз технологій та архітектурного проектування програмного забезпечення. Включаючи обґрунтування вибору мови програмування, середовища розробки, системи управління базами даних (СУБД), а також визначення вимог до програмного забезпечення та ORM фреймворків для порівняння;

– синтез у процесі дослідження. Використовуючи отримані знання та інформацію, була створена модель бази даних і запитів для порівняльного аналізу;

– експериментування з розробкою та використанням програмного забезпечення. Включаючи розробку алгоритмів, програмних модулів системи, а також апробацію застосунку з використанням різних ORM фреймворків;

– експериментування з аналізом результатів та вибором найкращих методів. Оцінка отриманих результатів і вибір найефективніших ORM фреймворків з урахуванням продуктивності, легкості використання, гнучкості та інших критеріїв.

Наукова новизна одержаних результатів

Наукова новизна одержаних в даному дослідженні результатів характеризується такими основними аспектами:

– створення універсальної моделі для порівняльного аналізу. Була розроблена універсальна модель, що дозволяє проектувати бази даних та формувати запити для аналізу продуктивності ORM фреймворків. Модель була сконструйована із урахуванням множини факторів, які впливають на роботу ORM, що забезпечує глибоке дослідження та порівняння фреймворків;

– впровадження комплексного підходу до оцінювання фреймворків. Дослідження пропонує новий метод оцінювання ORM фреймворків, який поєднує в собі аналіз продуктивності, легкості використання, гнучкості, а також таких компонентів як набір інструментів, масштабованість, документація та підтримка спільноти. Такий підхід дозволяє отримати найбільш повноцінні та об'єктивні результати оцінювання.

– розробка рекомендацій щодо використання ORM фреймворків. На основі отриманих результатів дослідження розроблено рекомендації стосовно вибору та застосування ORM фреймворків в різних контекстах розробки програмного забезпечення на платформі .NET.

Практичне значення одержаних результатів

Практичне значення одержаних результатів в процесі проведення дослідження включає:

– підвищення продуктивності роботи програмістів. Проведені в даному дослідженні аналізи та порівняння допоможуть розробникам .NET обрати найбільш підходящий ORM фреймворк для конкретних задач та проектів. Це, в свою чергу, підвищує ефективність їхньої роботи та знижує витрати часу на виконання задач;

– підвищення ефективності проектів з розробки програмного забезпечення. Наявність чітких рекомендацій щодо використання ORM фреймворків дозволяє збільшити ефективність розробки та масштабування проектів на платформі .NET;

– використання у навчальному процесі. Результати дослідження можуть бути використані в навчальних матеріалах та курсах для студентів, які вивчають .NET та розробку баз даних;

– застосування результатів дослідження в реальних проектах. Висновки та рекомендації з даного дослідження можуть бути використані в реальних проектах розробки програмного забезпечення.

Апробація одержаних результатів

Результати дослідження були представлені на XVI науково-практичній конференції студентів, аспірантів, докторантів і молодих вчених Запорізького національного університету «Молода наука-2023» [1], а також на науково-технічній конференції студентів, аспірантів, магістрантів і викладачів Інженерного навчально-наукового інституту Запорізького національного університету «Конференція молодих вчених» [2]**Помилка! Джерело посилання не знайдено.**.

Глосарій

Фреймворк (англ. Framework) — це структурований набір концепцій, практик, стандартів інструментів та коду, який визначає архітектуру програмної системи або проекту та надає загальні функції та можливості для спрощення розробки програмного забезпечення. Фреймворки надають готові до використання компоненти та інтерфейси, що дозволяють розробникам швидше створювати програми, орієнтовані на конкретні завдання чи платформи.

Платформа .NET (англ. .NET platform) — це фреймворк для розробки та виконання програмного забезпечення. Вона була розроблена компанією Microsoft і забезпечує серію інструментів та ресурсів для створення різноманітних типів програм, від десктопних до веб- та мобільних додатків.

Об'єктно-реляційне відображення (англ. Object-relational mapping) — це техніка програмування, яка дозволяє зв'язувати об'єктно-орієнтовану модель даних в програмі з реляційною базою даних.

Реляційна база даних (англ. Relational Database) — це тип бази даних, який використовує модель даних, базовану на теорії множин та представлення відносин між різними типами даних. Реляційні бази даних використовуються для зберігання та управління даними, що можуть бути організовані у вигляді таблиць.

Транзакція (англ. Transaction) — це група дій або операцій, які виконуються в базі даних. Ці операції виконуються так, що вони або виконуються повністю, або жодна з них не виконується.

Ліниве завантаження (англ. Lazy loading) — вказує на те, що дані завантажуються лише в той момент, коли вони дійсно потрібні. Це означає, що програма не завантажує всі дані одразу, а відкладає завантаження до того часу, коли дані будуть запитані або використані.

Жадібне завантаження (англ. Eager loading) — протилежне лінивому завантаженню. Дані завантажуються наперед, до того, як вони будуть факти-

чно потрібні. Це забезпечує швидший доступ до даних, оскільки вони вже наявні в пам'яті, коли програма потребує їх використати.

Явне завантаження (англ. Explicit loading) — передбачає те, що розробник вручну вказує, які саме дані потрібно завантажити. Замість автоматичного завантаження даних, розробник сам визначає, які об'єкти чи частини даних мають бути завантажені в пам'ять.

Кешування (англ. Caching) — це техніка зберігання копій обчислених або отриманих раніше результатів операцій, звертання до даних чи запитів, для подальшого використання замість повторного обчислення чи отримання даних із джерела.

Прикладний програмний інтерфейс (англ. Application Programming Interface або API) — це набір правил та інструментів, які дозволяють різним програмам взаємодіяти між собою. API визначає, які запитання можна робити до програми або сервісу, як вони повинні бути сформульовані, і які дані можна отримати відповідно до цих запитань.

Впровадження залежності (англ. Dependency Injection або DI) — це принцип проектування програмного забезпечення, який включає в себе передачу залежностей об'єктам, замість того, щоб вони створювали їх самостійно. Цей підхід робить код більш гнучким, об'єкти більш перевикористовуваними та сприяє зменшенню зв'язності між компонентами системи.

Час користувача ЦП (англ. CPU User Time) — це кількість часу, яку центральний процесор (ЦП) витрачає на виконання інструкцій користувацьких програм або задач, які працюють в користувацькому просторі.

Час ядра ЦП (англ. CPU Kernel Time) — це кількість часу, яку центральний процесор (ЦП) витрачає на виконання системних операцій та служб ядра операційної системи.

CRUD — це абревіатура, яка визначає чотири основні операції, які можна виконувати з даними в системі управління базами даних або в контексті розробки програмного забезпечення. Ці операції включають створення (Create), читання (Read), оновлення (Update) та видалення (Delete) даних. Вони

є ключовими для управління та обробки інформації в базах даних та забезпечують основні функції для взаємодії з даними в різних типах систем баз даних

LINQ (Language-Integrated Query) — це мовно-інтегрований механізм запитів, який дозволяє виконувати різноманітні операції обробки та фільтрації даних безпосередньо в коді C#. LINQ дозволяє розробникам писати більш зрозумілий, зручний та ефективний код для роботи з колекціями даних, базами даних, XML-документами та іншими джерелами даних.

POCO об'єкти (англ. Plain Old CLR Objects) — це звичайні класи у мові програмування, які не мають жодних залежностей від конкретного фреймворку чи технології. У контексті роботи з базами даних, POCO класи можуть виступати в якості моделей даних, які можна використовувати з ORM системами для спрощення взаємодії з базою даних. POCO підходить для ситуацій, де простота та універсальність є ключовими вимогами.

Swagger — це набір інструментів для документування та взаємодії з RESTful API. Основним елементом Swagger є специфікація OpenAPI, яка визначає стандарти для опису REST API. Це дозволяє розробникам та іншим зацікавленим сторонам легко розуміти функціонал API та спрощує його використання.

Postman — це клієнтський інструмент для формування запитів до API. Він дозволяє розробникам взаємодіяти з різними видами API, такими як RESTful, SOAP, інші веб-сервіси та HTTP-запити.

Microsoft SQL Server — це реляційна система управління базами даних, розроблена компанією Microsoft. SQL Server є однією з найпопулярніших баз даних серед корпоративних користувачів та розробників.

SQL Server Profiler — це інструмент, який надається Microsoft SQL Server для моніторингу та аналізу діяльності бази даних. Він дозволяє адміністраторам баз даних та розробникам стежити за запитами, які виконуються на сервері бази даних, а також для вивчення різних аспектів їх виконання.

РОЗДІЛ 1 ОБ'ЄКТНО-РЕЛЯЦІЙНЕ ВІДОБРАЖЕННЯ ТА ЙОГО ЗНАЧЕННЯ ДЛЯ РОЗРОБКИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Визначення та основи об'єктно-реляційного відображення

Об'єктно-реляційне відображення – це техніка програмування, яка спрямована на перетворення даних між двома несумісними системами типів в об'єктно-орієнтованих програмних мовах. В основу ORM покладена ідея використання об'єктів програмування, щоб створити віртуальну базу даних, яка може бути використана в межах програмного коду.

Основна ідея ORM полягає у відображенні реляційних таблиць на об'єкти програмування, а також відображенні операцій SQL на методи цих об'єктів. ORM дозволяє розробникам працювати з базами даних, використовуючи об'єктно-орієнтовані парадигми.

ORM забезпечує значні переваги, зокрема, забезпечення високого рівня абстракції при використанні баз даних, зменшення кількості коду, що написаний вручну, а також збільшення швидкості розробки за рахунок використання готових шаблонів і наборів функцій [4].

Основні концепції ORM включають:

– відображення (англ. mapping). Процес встановлення відповідності між об'єктами в коді та рядами в таблицях бази даних;

– сесія. Це об'єкт, який реалізує шаблон проектування "Unit of Work" і забезпечує API для створення, читання, зміни та видалення об'єктів;

– запити. ORM надає API для виконання SQL-запитів та отримання результатів;

– транзакції. ORM дозволяє робити взаємозалежні операції з базою даних атомарними, усунувши таким чином проблеми невідповідності станів.

ORM найчастіше використовується для розробки:

– веб-додатків. ORM фреймворки широко використовуються в ASP.NET додатках для взаємодії з базами даних. Це включає все, від простих веб-сайтів до складних веб-додатків з багатими функціями [5];

– серверних додатків. .NET додатки, що працюють на сервері, також часто використовують ORM для здійснення CRUD операцій з базами даних. Це може включати в себе все, від API серверів до бекграундових сервісів та систем обробки даних [6];

– корпоративних додатків. Великі корпоративні додатки, які вимагають взаємодії з різними базами даних та обміну даними між різними системами, часто використовують ORM для спрощення цих процесів.

– мобільних додатків. Хоча .NET традиційно асоціюється з веб- і серверною розробкою, з появою Xamarin та .NET MAUI, ORM фреймворки також можуть бути використані в контексті мобільної розробки.

У всіх цих сценаріях, ORM фреймворки дозволяють розробникам сконцентруватися на бізнес-логіці та вимогах до додатків, замість того, щоб бути зайнятими низькорівневими деталями реалізації взаємодії з базами даних.

1.2 Роль ORM у сучасному програмуванні

Об'єктно-реляційні відображення відіграють визначальну роль в сучасному програмуванні, служачи стратегічним механізмом зв'язку між реляційними базами даних та об'єктно-орієнтованими мовами програмування. Основне завдання ORM полягає в тому, щоб ізолювати розробника від низькорівневих деталей реалізації SQL, дозволяючи зосередитися на бізнес-логіці та об'єктно-орієнтованому дизайну.

Відсутність ORM могла б перетворити процес розробки на складний набір операцій: написання обширного SQL коду, переклад між форматами бази даних і об'єктно-орієнтованої моделі, подвійна робота при зміні структури бази даних або коду. В той же час, застосування ORM дозволяє розробникам працювати з більш зрозумілими об'єктами і колекціями, обходячи непосередній SQL.

Однак, функціонал ORM не обмежується лише зручностями для розробників. Вони також забезпечують оптимізацію, роблячи код більш структуро-

ваним, чистим і читабельним. До того ж, багато ORM фреймворків підтримують велику кількість реляційних систем управління базами даних, надаючи розробникам гнучкість у виборі підходящої технології.

Щодо безпеки, ORM фреймворки мають вбудовані механізми захисту від загроз, наприклад, SQL ін'єкцій, через автоматизовану генерацію і виконання SQL запитів.

Крім того, ORM можуть сприяти підвищенню якості коду через підтримку модульного тестування. Це дає змогу розробникам створювати тестові сценарії для баз даних та проводити автоматичні тести, що сприяє покращенню надійності коду та ефективності циклу розробки.

Отже, ORM є важливим інструментом у сучасному програмуванні, що надає можливість розробникам зосереджуватися на ключових аспектах розробки, при цьому отримуючи багато переваг, включаючи ефективність, безпеку, тестування та гнучкість у виборі технологій.

1.3 Стандарти об'єктно-реляційного відображення у .NET

Стандарти ORM для .NET визначають зв'язок між об'єктами в програмному коді і структурами даних у реляційній базі даних. Ці стандарти допомагають забезпечити узгодженість, ефективність і безпеку при роботі з базами даних.

Взаємодія з базами даних є одним з основних завдань при розробці багатьох видів програмного забезпечення. Microsoft .NET, як потужна та гнучка платформа для розробки, надає ряд стандартів і фреймворків для взаємодії з базами даних, зокрема через технологію об'єктно-реляційного відображення [7]. Основні стандарти ORM в .NET включають:

– стандарти та конвенції. ORM фреймворки повинні дотримуватися загальних стандартів та конвенцій .NET, що включає роботу з типами даних .NET, виключеннями, подіями та іншими особливостями мови. Вони також повинні інтегруватися з іншими частинами екосистеми .NET, такими як

ASP.NET Core для веб-розробки і ADO.NET для низькорівневої взаємодії з базами даних;

– з'єднання з базою даних. ORM фреймворки повинні використовувати стандартні .NET інтерфейси для роботи з базами даних, такі як IDbConnection, IDbCommand, IDbTransaction і IDataReader. Це дозволяє ORM фреймворкам взаємодіяти з будь-якою базою даних, яка підтримує ці інтерфейси;

– відображення даних. ORM фреймворки повинні відображати рядки бази даних на об'єкти .NET. Це часто досягається за допомогою атрибутів, які визначають, як поля бази даних відповідають властивостям об'єкта, або за допомогою API налаштування, які дозволяють декларативно визначати це відображення;

– транзакції. ORM фреймворки повинні підтримувати створення та керування транзакціями бази даних. Це часто включає в себе роботу з транзакціями за допомогою об'єктів IDbTransaction;

– LINQ. Багато ORM фреймворків для .NET підтримують LINQ, що дозволяє розробникам виконувати запити до бази даних, використовуючи синтаксис C# або VB.NET. Це дозволяє розробникам використовувати знайомий синтаксис для взаємодії з базою даних і спрощує процес написання складних запитів;

– підтримка міграцій. Не всі, але деякі ORM фреймворки, такі як Entity Framework Core, включають підтримку міграцій бази даних, що дозволяє розробникам змінювати схему бази даних з часом, зберігаючи при цьому історію цих змін;

– робота з відношеннями. ORM фреймворки повинні підтримувати відображення відношень між таблицями бази даних на відношення між об'єктами .NET. Це включає в себе підтримку "один до одного", "один до багатьох", "багато до багатьох" відношень.

Усі ці стандарти разом формують базовий набір вимог до ORM фреймворків в .NET, що сприяє ефективності, надійності та продуктивності в розробці програмного забезпечення.

Для платформи .NET існує ряд ORM-фреймворків, серед яких Entity Framework Core, Dapper, NHibernate, тощо. Вони дозволяють розробникам виконувати CRUD операції з базами даних без прямого написання SQL-запитів, підтримують LINQ, що забезпечує безпечний і зручний спосіб роботи з базами даних.

Стандарти ORM для .NET важливі з декількох причин:

– сумісність. Якщо ORM фреймворк дотримується стандартів .NET, він може легко взаємодіяти з іншими компонентами .NET. Наприклад, він може використовувати стандартні .NET інтерфейси для виконання операцій з базою даних;

– переносимість. Якщо ORM фреймворк дотримується стандартів, його можна використовувати з різними базами даних. Це означає, що можна змінити базу даних без необхідності зміни коду, що працює з ORM;

– продуктивність. Дотримання стандартів може підвищити продуктивність. Наприклад, використання стандартних інтерфейсів для виконання SQL запитів може поліпшити продуктивність за рахунок використання оптимізацій, вбудованих в базу даних;

– безпека. Дотримання стандартів може забезпечити безпеку. Наприклад, використання параметризованих запитів, які є стандартом, може допомогти запобігти SQL ін'єкціям;

– масштабування. Стандарти також допомагають з масштабуванням. Якщо код написаний з урахуванням стандартів, він буде більш гнучким і масштабованим [8].

1.4 Переваги та недоліки використання об'єктно-реляційного відображення

Об'єктно-реляційне відображення стало одним з основних інструментів, які використовують розробники для взаємодії з базами даних в контексті платформи Microsoft .NET. Проте, прийняття рішення щодо використання ORM в конкретному проекті вимагає зваженого аналізу як переваг, так і можливих недоліків цього підходу. У цьому розділі ми детально розглянемо основні переваги та недоліки, пов'язані з використанням ORM фреймворків, зокрема на платформі .NET.

Переваги використання об'єктно-реляційного відображення:

– автоматизація коду. ORM фреймворки можуть автоматизувати багато коду, який б розробникам довелося писати вручну. Зокрема, вони можуть автоматично генерувати код для взаємодії з базою даних, які включають CRUD (створення, читання, оновлення, видалення) операції;

– менше помилок. Автоматизований код, зазвичай, менш схильний до помилок, ніж код, написаний вручну, тому що він базується на відповідних стандартах і тестується на великому числі сценаріїв;

– збільшена продуктивність. Розробники можуть працювати ефективніше, зосереджуючись на бізнес-логіці програми, а не на деталях роботи з базою даних;

– відмова від SQL. Використовуючи ORM, розробникам не потрібно напряму працювати з SQL. Це може бути корисним для розробників, які не знайомі з SQL або віддають перевагу об'єктно-орієнтованому програмуванню.

– сумісність з базами даних. ORM фреймворки зазвичай сумісні з багатьма різними типами баз даних, що дозволяє розробникам легко переключатися між різними СУБД.

Недоліки використання об'єктно-реляційного відображення:

– зниження продуктивності. ORM може сповільнити продуктивність програми, оскільки вони додають додатковий шар між програмою та базою даних;

– складність. ORM фреймворки можуть бути складними і вимагати часу для вивчення. Вони також можуть вносити свої власні обмеження та нюанси, які можуть ускладнити розробку;

– низька гнучкість. ORM фреймворки можуть бути менш гнучкими, ніж пряма робота з SQL. Це може бути проблематичним для деяких застосунків, які вимагають складних або нестандартних запитів до бази даних;

– проблеми з проектуванням. ORM фреймворки можуть примушувати розробників приймати певні рішення проектування, які не обов'язково є оптимальними для їх конкретного застосунку;

– абстракція бази даних. ORM намагається абстрагувати роботу з базою даних, але іноді розробники все ще потребують глибокого знання SQL та бази даних для оптимізації продуктивності та управління транзакціями.

Отже, під час вибору використання ORM для проекту, розробники повинні розглянути ці переваги та недоліки в контексті вимог до їх конкретного застосунку.

Висновки з розділу 1

У даному розділі проведено дослідження концепції об'єктно-реляційного відображення. Визначення ORM як фундаментального інструменту для високорівневої взаємодії розробників з базами даних, використовуючи об'єктно-орієнтовані моделі, стало центральною темою. Поняття, концепції та основи ORM були ретельно вивчені, з урахуванням того, як ORM впливає на забезпечення гнучкості та ефективності у розробці програмного забезпечення, що є одним із ключових факторів успіху в сучасному програмуванні.

За допомогою дослідження стандартів ORM для платформи .NET було підняте ключове питання про дотримання загальних стандартів та конвенцій .NET. До цього включається розуміння і робота з типами даних .NET, виключеннями, подіями та іншими особливостями мови. Оцінка цих стандартів не лише допомогла в зрозумінні принципів ORM, але і підкреслила їх значущість для високоефективного програмування.

Питання переваг та недоліків ORM були розглянуті з багатьох точок зору, наводячи на приклади про збільшення продуктивності розробників та спрощення взаємодії з базами даних. Однак, наряду з цими перевагами, було важливо підкреслити і потенційні виклики, пов'язані з продуктивністю та управлінням складними запитами до бази даних. Ці нюанси допомагають виявити більш глибоке розуміння контексту та використання ORM.

В цілому, цей розділ став підґрунтям до наступного етапу дослідження, який буде присвячений детальному порівнянню ORM фреймворків для платформи .NET, таких як Entity Framework Core, Dapper, NHibernate. Подальший аналіз забезпечить не лише більш глибоке розуміння ORM, але і допоможе визначити, який фреймворк найкраще підходить для конкретних задач та сценаріїв розробки. Подальший аналіз допоможе виявити, який фреймворк найкраще підходить для конкретних задач та сценаріїв розробки, що значно приблизить до досягнення фінальної мети дослідження.

РОЗДІЛ 2 ОГЛЯД ФРЕЙМВОРКІВ ORM ДЛЯ ПЛАТФОРМИ .NET

2.1 Entity Framework Core

2.1.1 Основні особливості Entity Framework Core

Entity Framework Core (EFC) є відкритим ORM фреймворком від Microsoft, який інтенсивно використовується у програмуванні на платформі .NET. Його розробка та підтримка активно ведуться, що робить його сучасним та відповідним вимогам ринку. Основні особливості Entity Framework Core зосереджуються на високорівневій абстракції роботи з базами даних, що значно спрощує розробку та підтримку додатків [9].

Ключові особливості Entity Framework Core, які відображають його потенціал і гнучкість включають:

– підтримка моделі "Code First" та "Database First". EFC надає гнучкість вибору підходу до моделювання даних. За допомогою "Code First" програмісти можуть визначати структуру бази даних безпосередньо у коді, а EFC сам генерує необхідні SQL запити для створення або модифікації структури бази даних. З іншого боку, "Database First" дозволяє розробникам генерувати модель даних на основі вже існуючої структури бази даних;

– інтеграція з Language Integrated Query. EFC підтримує LINQ, що дозволяє розробникам використовувати мову C# для написання складних запитів до бази даних. Це зменшує необхідність в ручному написанні SQL запитів та полегшує процес розробки [10];

– автоматичне відслідковування змін. EFC відслідковує всі зміни, внесені в об'єкти моделі, і автоматично генерує відповідні SQL запити при збереженні цих змін. Це підвищує ефективність розробки та спрощує управління даними;

– підтримка відносин між таблицями. EFC вміє обробляти відношення між таблицями, такі як "один до багатьох", "багато до багатьох" та

"один до одного". Така підтримка відношень між об'єктами у моделі дозволяє легко і ефективно працювати з взаємозв'язаними даними;

– кешування. EFC використовує кешування першого рівня, що забезпечує більш ефективне використання ресурсів під час виконання повторюваних запитів до одних і тих же даних в межах одного контексту даних;

– розширене управління помилками. EFC надає засоби для обробки помилок, що дозволяє розробникам керувати помилками, що виникають під час виконання SQL запитів, використовуючи стандартні засоби C#;

– підтримка пакетних операцій. EFC підтримує виконання пакетних операцій, що дозволяє зменшувати навантаження на базу даних та підвищувати продуктивність додатків за рахунок зменшення кількості SQL запитів до бази.

В цілому, особливості Entity Framework Core роблять його потужним, гнучким та ефективним інструментом для розробки на платформі .NET.

2.1.2 Особливості створення моделі

EFC є потужним інструментом для взаємодії з базами даних в .NET. Він складається з ряду компонентів, які працюють разом, щоб створити абстракцію рівня бази даних та забезпечити об'єктно-орієнтований інтерфейс для маніпуляцій з даними [11]. Особливості моделі Entity Framework Core представлено на рисунку 1.

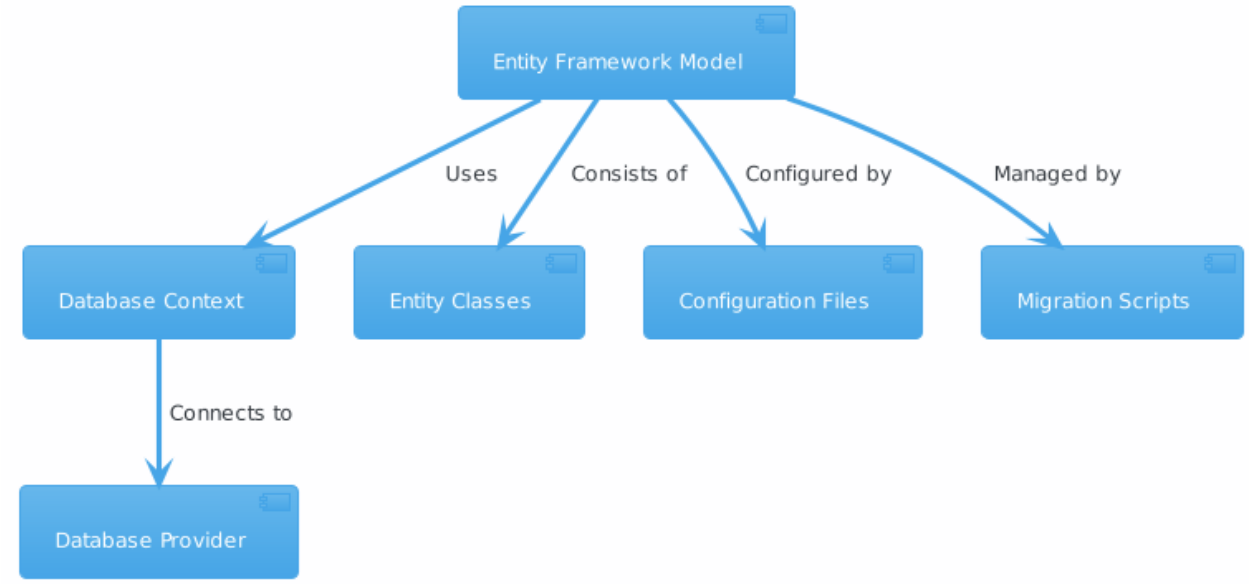

Рис. *1 Взаємодія компонентів при створенні моделі EFC*

Модель Entity Framework Core складається із таких компонентів:

– Entity Framework Model (EFModel). Це основний компонент, який використовує всі інші компоненти для створення та управління базою даних.

– Database Context (DbContext). EFModel використовує DbContext для встановлення з'єднання з базою даних. DbContext є мостом між вашою моделлю EFC (сутностями) та базою даних;

– Database Provider (DbProvider). DbContext використовує DbProvider для взаємодії з конкретною базою даних, наприклад SQL Server, MySQL тощо;

– Entity Classes (Entities). EFModel складається з сутностей, які представляють таблиці в базі даних. Кожна сутність має відображення на таблицю в базі даних;

– Configuration Files (Config). EFModel налаштовується за допомогою файлів конфігурації, де ви можете вказати різні параметри, такі як рядок підключення, провайдер бази даних, налаштування міграції тощо;

– Migration Scripts (Migrations). EFModel керує міграціями, які є скриптами, що автоматично генеруються для внесення змін до схеми бази даних. Міграції дозволяють вам вносити зміни в модель та синхронізувати ці зміни з базою даних.

Отже, використання Entity Framework Core дозволяє розробникам зосередитися на бізнес-логіці, замість написання багато коду для роботи з базою даних. EFC автоматизує багато рутинних завдань, пов'язаних з доступом до даних, що забезпечує більш ефективний та продуктивний процес розробки.

2.1.3 Робота з даними

Для роботи з даними в Entity Framework Core використовуються дві ключові компоненти: DbContext і DbSet [12]. DbContext є основним класом, що представляє сесію з базою даних та дозволяє виконувати операції CRUD. Кожна сутність, яка відображає таблицю в базі даних, представляється через DbSet в DbContext.

EFC надає декілька способів зчитування даних. Запити можна формувати за допомогою LINQ, який інтегрований в C# і дозволяє писати запити безпосередньо в коді. EFC також підтримує зчитування даних за допомогою SQL-запитів.

Для змінення даних розробник повинен спочатку зчитати дані, внести зміни в об'єкти, а потім викликати метод DbContext.SaveChanges() або асинхронний метод DbContext.SaveChangesAsync(). Entity Framework автоматично визначить, які об'єкти були змінені, і згенерує відповідні SQL-команди для оновлення бази даних.

Для додавання нових записів в базу даних розробник створює новий об'єкт, додає його до відповідного DbSet та викликає метод DbContext.SaveChanges(). Для видалення записів процес подібний: спочатку зчитуються дані, потім вони видаляються з DbSet, а потім знову викликається DbContext.SaveChanges().

EFC автоматично управляє транзакціями при виконанні операцій CRUD. Однак розробник також може контролювати транзакції вручну, якщо це необхідно.

Також EFC має вбудовані механізми для запобігання таким загрозам безпеки, як SQL Injection. За замовчуванням всі запити, що виконуються через

Entity Framework Core, використовують параметризовані SQL запити, які забезпечують безпечну роботу з даними.

Отже, EFC надає потужний набір інструментів для роботи з даними в об'єктно-орієнтованому стилі, що забезпечує високий рівень абстракції і гнучкість при розробці програмного забезпечення.

2.1.3.1 Організація CRUD-запитів

Entity Framework Core забезпечує потужні та гнучкі механізми для виконання цих операцій в контексті об'єктно-реляційного відображення.

Основні функції, які використовуються в персистентних сховищах даних включають:

– створення (Create). Для створення нового запису в базі даних розробник спочатку створює новий екземпляр моделі даних. Після цього новий об'єкт додається до відповідного DbSet в DbContext. Після виклику методу SaveChanges() на DbContext, Entity Framework Core згенерує відповідний SQL запит для вставки нового запису в базу даних;

– читання (Read). Entity Framework Core надає багато способів зчитування даних з бази даних. Можна використовувати LINQ-запити, які можна писати безпосередньо в коді C#, або використовувати SQL запити. Для ефективного зчитування даних Entity Framework Core також підтримує різні операції, такі як відкладене завантаження, швидке завантаження та явне завантаження;

– оновлення (Update). Для оновлення запису в базі даних, розробник спочатку має зчитати відповідний запис з бази даних, потім змінити властивості цього об'єкта в коді, а потім викликати метод SaveChanges() на DbContext. Entity Framework автоматично визначить, які об'єкти були змінені, і згенерує відповідні SQL-команди для оновлення бази даних;

– видалення (Delete). Процес видалення запису з бази даних досить схожий на процес оновлення. Розробник спочатку має зчитати відповідний запис, потім видалити цей об'єкт з DbSet, а потім викликати метод SaveChanges()

на DbContext. Entity Framework Core згенерує SQL запит на видалення для видалення запису з бази даних [13].

Використовуючи об'єктно-орієнтований підхід, Entity Framework Core дозволяє розробникам працювати з базами даних так, як би вони працювали з об'єктами в коді, що забезпечує високу продуктивність та зручність.

2.1.3.2 Запити до пов'язаних таблиць

Однією з сильних сторін Entity Framework Core є здатність ефективно обробляти запити до пов'язаних таблиць, використовуючи концепцію навігаційних властивостей. Ці властивості дозволяють розробникам створювати зв'язки між моделями у вигляді об'єктних відношень, відображаючи таким чином зв'язки між таблицями в базі даних.

EFC дозволяє встановлювати наступні зв'язки:

– "один до одного" та "один до багатьох". За допомогою навігаційних властивостей. Для цього розробникам потрібно визначити навігаційну властивість в класі, який відображає "одну" сторону зв'язку, яка може містити колекцію об'єктів "багатьох" сторін. На "багатьох" сторонах, навігаційна властивість буде вказувати на "один" об'єкт;

– "багато до багатьох". Для цього розробникам потрібно створити додаткову модель, яка відображатиме проміжну таблицю в базі даних, і визначити дві навігаційні властивості, які вказують на кожну сторону зв'язку;

– запити до декількох таблиць. З використанням навігаційних властивостей, запити до пов'язаних таблиць стають простими та інтуїтивно зрозумілими. Наприклад, можна використовувати метод Include() для явного завантаження пов'язаних даних, або розробники можуть використовувати LINQ-запити для створення складних запитів, які включають дані з декількох таблиць.

EFC також надає різні стратегії завантаження пов'язаних даних, такі як "ліниве завантаження", "жадібне завантаження" та "явне завантаження", дозволяючи розробникам вибрати найбільш ефективний підхід для їх конкретних потреб.

У цілому, робота з пов'язаними таблицями в EFC є гнучкою та ефективною, що дозволяє розробникам використовувати повний потенціал об'єктно-реляційного відображення для створення складних запитів до бази даних.

2.1.3.3 Організація транзакцій

Транзакції відіграють суттєву роль у роботі з базами даних, оскільки вони гарантують, що низка операцій з даними буде виконана як одна цілісна одиниця роботи, тобто або всі операції будуть успішно завершені, або жодна з них не буде виконана, в разі виникнення будь-яких помилок. Entity Framework Core надає вбудовану підтримку для роботи з транзакціями.

EFC автоматично використовує транзакції при виконанні операцій з базою даних за допомогою метода SaveChanges() чи його асинхронного аналога SaveChangesAsync(). Коли викликається SaveChanges(), EFC автоматично відкриває транзакцію, виконує всі необхідні SQL-операції, а потім закриває транзакцію. Якщо всі операції пройшли успішно, транзакція фіксується; у разі помилки транзакція відкочується.

У деяких випадках може бути необхідно контролювати транзакції вручну. EFC дозволяє це робити за допомогою методів BeginTransaction(), Commit() та Rollback(). Розробник може створити нову транзакцію за допомогою BeginTransaction(), фіксувати зміни за допомогою Commit(), або відкотити транзакцію за допомогою Rollback().

EFC також підтримує транзакції в розподілених системах за допомогою TransactionScope. Це дозволяє групувати кілька операцій, які можуть виконуватися на різних ресурсах, в одну транзакцію. Якщо будь-яка з операцій зазнає невдачі, всі операції в транзакції відкочуються.

Отже, EFC надає гнучке та потужне керування транзакціями, що дозволяє розробникам писати надійний і ефективний код для взаємодії з базами даних.

2.1.4 Інтеграція з .NET

EFC базується на ADO.NET, що є стандартом доступу до даних в .NET. Це означає, що EFC співпрацює з усіма основними провайдерами даних ADO.NET, включаючи SQL Server, MySQL, Oracle та інші. Ця тісна інтеграція дає розробникам можливість використовувати всю потужність ADO.NET, в той же час працюючи з вищим рівнем абстракції, що надається EFC.

EFC повністю інтегрований з LINQ, що є вбудованим механізмом запитів в .NET. LINQ дозволяє розробникам виконувати запити до даних, використовуючи синтаксис C# або Visual Basic, замість написання SQL коду. Це спрощує розробку, зменшує помилки і підвищує продуктивність.

Даний фреймворк також тісно інтегрований з ASP.NET, що є основним фреймворком для розробки веб-додатків в .NET. Завдяки цій інтеграції розробники можуть легко створювати веб-додатки, які взаємодіють з базами даних, використовуючи EFC для виконання всіх операцій з даними.

Entity Framework Core є новою версією Entity Framework (EF), спеціально розробленою для .NET Core. Вона включає всі ключові особливості EF, а також додаткові вдосконалення, такі як підтримка крос-платформеності, висока продуктивність та гнучкість.

Отже, EFC є цінним інструментом для розробки баз даних в екосистемі .NET, завдяки своїй глибокій інтеграції з ключовими компонентами платформи.

2.1.5 Сценарії використання Entity Framework Core

До найбільш ключових сценаріїв використання EFC можна віднести:

– розробка веб-додатків. EFC широко використовується при розробці веб-додатків на ASP.NET. Він дозволяє розробникам працювати з базами даних за допомогою об'єктно-орієнтованого інтерфейсу, замість використовувати SQL запити. Це може включати виконання CRUD операцій, обробку відносин між таблицями, використання транзакцій і багато іншого.

– створення мікросервісів. Entity Framework Core надає потужні можливості для створення мікросервісів. Він підтримує крос-платформеність, високу продуктивність та гнучкість, що робить його ідеальним вибором для створення мікросервісних архітектур;

– розробка настільних додатків. EFC також може бути використаний для розробки настільних додатків на .NET. Він може спростити взаємодію з базою даних, виконуючи всю важку роботу з SQL, дозволяючи розробникам зосередитися на бізнес-логіці додатку;

– робота з великими даними. EFC також підтримує роботу з великими даними, дозволяючи розробникам використовувати LINQ-запити для обробки великих наборів даних. Однак, для оптимізації продуктивності, може бути необхідним дотримання деяких особливостей інструменту, таких як використання "лінивого завантаження" замість "жадібного завантаження" при роботі з великими об'ємами даних.

Отже, EFC - це багатогранний інструмент, який може бути ефективно використаний в різних сценаріях розробки, від веб-додатків і мікросервісів до настільних додатків і обробки великих даних.

2.1.6 Переваги та недоліки Entity Framework Core

При вивченні будь-якого фреймворку важливо зрозуміти його переваги та обмеження. EFC, як і будь-яка інша технологія, має свої власні переваги та недоліки.

Переваги Entity Framework Core включають:

– високий рівень абстракції. EFC надає високий рівень абстракції при виконанні операцій з базами даних. Це означає, що розробники можуть працювати з базами даних, використовуючи об'єктно-орієнтований підхід, що зменшує необхідність написання вручну складних SQL-запитів;

– підтримка LINQ-запитів. EFC дозволяє використовувати LINQ (Language Integrated Query) для формування і виконання запитів до бази даних.

LINQ дозволяє розробникам використовувати стандартний синтаксис мови C# для написання запитів, що робить код більш зрозумілим та читабельним;

– автоматичне відстеження змін. EFC автоматично відстежує зміни в об'єктах, що дозволяє легко виконувати операції зі зміною даних (Update, Delete);

– підтримка міграцій. EFC має вбудовану підтримку міграцій, що спрощує процес зміни структури бази даних відповідно до змін у моделях даних.

– кросплатформеність. EFC може використовуватись не лише для Windows систем, а також для інших платформ, як, наприклад, Linux.

До недоліків Entity Framework відносяться такі аспекти:

– продуктивність. EFC може бути менш продуктивним, ніж деякі інші ORM, особливо при виконанні складних запитів або при роботі з великими наборами даних. Це пов'язано з тим, що він генерує SQL-запити автоматично, що може призвести до менш оптимальних запитів, ніж ті, що були б написані вручну [14];

– складність. EFC може бути складним для вивчення, особливо для нових розробників. Це включає розуміння його внутрішньої архітектури, принципів роботи з об'єктно-реляційним відображенням, міграцій, а також використання LINQ;

– відсутність контролю над SQL. Хоча автоматичне генерування SQL є перевагою EFC з точки зору зручності, це може бути недоліком для розробників, які хотіли б контролювати точний SQL-запит, що виконується.

Таким чином, EFC представляє собою потужний інструмент для роботи з базами даних.

2.2 NHibernate

2.2.1 Основні особливості NHibernate

NHibernate є одним з найбільш широко використовуваних фреймворків об'єктно-реляційного відображення в екосистемі .NET. Цей фреймворк забезпечує велику гнучкість і масштабованість при роботі з реляційними базами даних [15] Основні особливості NHibernate включають:

– робота з об'єктно-орієнтованими доменами. NHibernate забезпечує відображення об'єктів в реляційні таблиці і навпаки, дозволяючи розробникам працювати з об'єктно-орієнтованим доменом;

– гнучке відображення. NHibernate підтримує різні стилі відображення, включаючи XML, атрибути .NET та Fluent NHibernate. Ця гнучкість дозволяє розробникам вибирати стиль відображення, який найкраще підходить для їх потреб;

– ліниве завантаження. NHibernate підтримує ліниве завантаження об'єктів. Це означає, що об'єкти завантажуються тільки при першому зверненні до них, що може значно покращити продуктивність застосунків, що працюють з великими об'ємами даних;

– кешування. NHibernate має вбудовану підтримку кешування, що дозволяє зберігати результати запитів та відтворювати їх при подальших зверненнях, що в свою чергу зменшує навантаження на базу даних;

– розширене формування запитів. NHibernate підтримує розширене формування запитів, включаючи LINQ, HQL (Hibernate Query Language) та критерії. Це дозволяє розробникам використовувати найкращий підхід до формування запитів, залежно від їх потреб;

– підтримка транзакцій. NHibernate підтримує транзакції, дозволяючи розробникам виконувати ряд операцій над базою даних як єдину атомарну операцію;

– розширення. NHibernate має вбудовану підтримку розширень, що дозволяє розробникам налаштовувати його поведінку та додавати нові функції.

Отже, NHibernate є потужним та гнучким фреймворком ORM, який використовується для створення застосунків .NET, що працюють з РБД.

2.2.2 Особливості створення моделі

NHibernate забезпечує велику гнучкість і масштабованість при роботі з реляційними базами даних. Особливості моделі NHibernate представлено на рисунку 2.

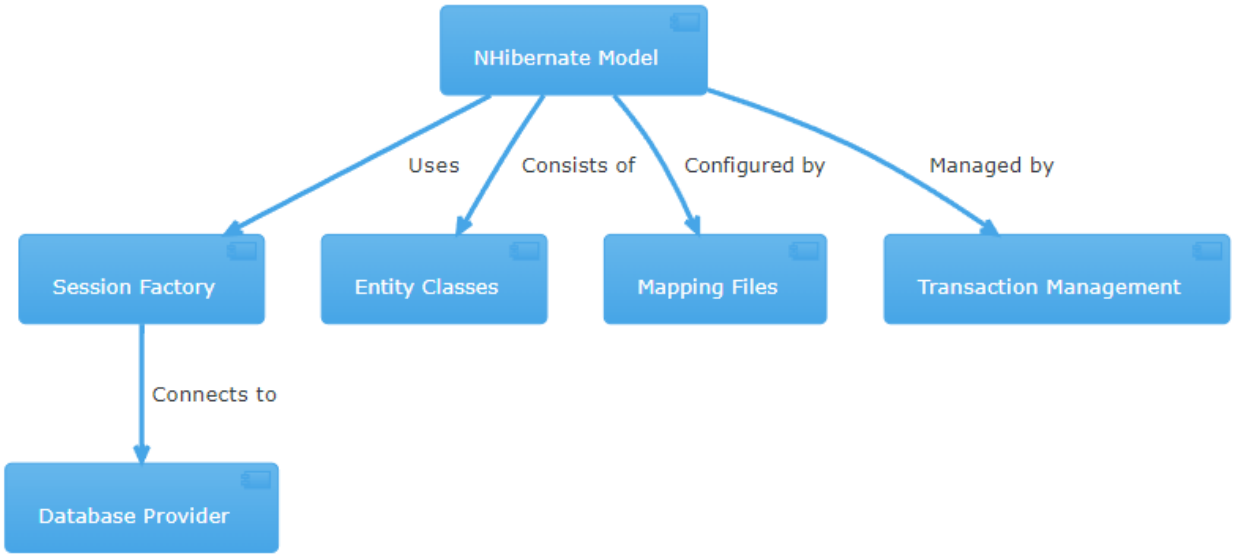

Рис. *2 Взаємодія компонентів при створенні моделі NHibernate*

Модель NHibernate складається із таких компонентів:

– NHibernate Model (NHModel). Це основний компонент, який використовує всі інші компоненти для створення та управління базою даних;

– Session Factory (SessionFactory). NHModel використовує SessionFactory для встановлення сесій, які є основними точками взаємодії з базою даних;

– Database Provider (DbProvider). SessionFactory використовує DbProvider для взаємодії з конкретною базою даних, наприклад SQL Server, MySQL тощо;

– Entity Classes (Entities). NHModel складається з сутностей, які представляють таблиці в базі даних. Кожна сутність відображається на таблицю в базі даних;

– Mapping Files (MappingFiles). NHModel налаштовується за допомогою файлів відображення, де ви можете вказати, як сутності відображаються на таблиці бази даних;

– Transaction Management (Transactions). NHModel керує транзакціями, які є важливим механізмом для забезпечення консистентності даних під час виконання операцій з базою даних.

2.2.3 Робота з даними

Для роботи з даними в NHibernate використовуються дві ключові компоненти: ISessionFactory і ISession. ISessionFactory є основним інтерфейсом, що представляє сесію з базою даних та дозволяє виконувати операції CRUD. Кожна сутність, яка відображає таблицю в базі даних, обробляється через ISession в ISessionFactory.

NHibernate надає декілька способів зчитування даних. Запити можна формувати за допомогою HQL (Hibernate Query Language), який інтегрований в NHibernate і дозволяє писати запити безпосередньо в коді. NHibernate також підтримує зчитування даних за допомогою SQL запитів.

Для змінення даних розробник повинен спочатку зчитати дані, внести зміни в об'єкти, а потім викликати метод ISession.Flush(), чи асинхронний аналог ISession.FlushAsync(). NHibernate автоматично визначить, які об'єкти були змінені, і згенерує відповідні SQL команди для оновлення бази даних.

Це забезпечує гнучкість і контроль над процесом роботи з даними, дозволяючи розробникам вільно вибирати стратегії збереження і відтворення даних, що найкраще підходять для їх конкретних потреб та сценаріїв.

2.2.3.1 Організація CRUD-запитів

Центральною частиною виконання CRUD операцій у NHibernate є інтерфейс ISession. ISession, який надає методи для здійснення всіх необхідних операцій з даними. Особливості CRUD операції NHibernate включають:

– створення. Для створення нового об'єкта в NHibernate використовується метод Save() або Persist(), або їх асинхронні аналоги SaveAsync() та PersistAsync(). Об'єкт, який необхідно зберегти, передається в якості аргумента цим методам. Після виконання цих методів, NHibernate генерує SQL запит INSERT і зберігає об'єкт в базі даних;

– читання. Для читання даних в NHibernate використовується метод Get() або Load(), або асинхронні аналоги GetAsync() та LoadAsync(). Ці методи приймають тип об'єкта і його ідентифікатор в якості аргументів і повертають об'єкт з бази даних;

– оновлення. Для оновлення об'єкта в NHibernate використовується метод Update() або UpdateAsync(). Спочатку потрібно завантажити об'єкт, виконати необхідні зміни, а потім викликати метод Update(). NHibernate автоматично генерує SQL запит UPDATE для оновлення об'єкта в базі даних;

– видалення. Для видалення об'єкта в NHibernate використовується метод Delete() або DeleteAsync(). Об'єкт, який потрібно видалити, передається в якості аргумента цьому методу. NHibernate автоматично генерує SQL запит DELETE і видаляє об'єкт з бази даних.

У всіх цих операціях важливо викликати метод ISession.Flush() або асинхронний аналог ISession.FlushAsync(), щоб забезпечити, що всі зміни, які були зроблені в об'єктах бази даних. Також варто згадати, що NHibernate використовує транзакції для забезпечення консистентності даних.

2.2.3.2 Запити до пов'язаних таблиць

NHibernate надає можливість встановлювати наступні зв'язки між таблицями:

– "один до одного" та "один до багатьох". Ці зв'язки реалізуються за допомогою навігаційних властивостей. Для їх встановлення, розробнику потрібно визначити навігаційну властивість в класі, який відображає "одну" сторону зв'язку, яка буде містити колекцію об'єктів "багатьох" сторін. З іншого боку, на "багатьох" сторонах, навігаційна властивість вказуватиме на "один" об'єкт;

– "багато до багатьох". Для встановлення цього типу зв'язків, розробникам потрібно створити додаткову модель, яка відображатиме проміжну таблицю в базі даних, та визначити дві навігаційні властивості, які вказують на кожну сторону зв'язку;

– запити до декількох таблиць. NHibernate дозволяє використовувати навігаційні властивості для виконання запитів до пов'язаних таблиць, що робить цей процес простим та інтуїтивно зрозумілим. Наприклад, можна використовувати різні стратегії завантаження, або використовувати LINQ-to-NHibernate для створення складних запитів, які включають дані з декількох таблиць.

NHibernate надає різні стратегії завантаження пов'язаних даних, такі як "ліниве завантаження", "жадібне завантаження" та "явне завантаження". Це дозволяє розробникам вибрати найбільш ефективний підхід для їх конкретних потреб.

Всі ці особливості роблять NHibernate потужним інструментом для виконання запитів до пов'язаних таблиць в базі даних, надаючи розробникам велику гнучкість і контроль над процесом завантаження та маніпуляції даними.

2.2.3.3 Організація транзакцій

Транзакції в NHibernate реалізовані за допомогою інтерфейсу ITransaction. Цей інтерфейс надає можливість контролювати процес виконання транзакцій, надаючи розробникам велику гнучкість.

Початок транзакції відбувається з викликом методу ISession.BeginTransaction(). Після виконання цього методу, всі подальші операції з даними в межах цієї сесії будуть виконуватися в межах даної транзакції. Це означає, що усі зміни даних не будуть фізично збережені в базі даних до моменту підтвердження транзакції.

Підтвердження транзакції відбувається за допомогою методу ITransaction.Commit(). Цей метод зберігає всі зміни в базі даних і закриває транзакцію. У випадку, якщо під час виконання транзакції виникли якісь помилки, транзакцію можна відмінити за допомогою методу ITransaction.Rollback().

NHibernate також підтримує роботу з розподіленими транзакціями. Для цього використовується інтерфейс ISession.Transaction, який дозволяє працювати з транзакцією як з окремим об'єктом.

Окрім того, NHibernate надає можливість автоматичного керування транзакціями за допомогою механізму контексту сесії (session context). Цей механізм дозволяє автоматично створювати нову сесію та транзакцію для кожного потоку виконання.

Отже, NHibernate надає потужні інструменти для керування транзакціями, що дозволяє розробникам ефективно керувати процесом зміни даних в базі даних.

2.2.4 Інтеграція з .NET

NHibernate, будучи одним з перших ORM-фреймворків для платформи .NET, розроблений на основі Hibernate для Java, відрізняється тісною інтеграцією з екосистемою .NET, що забезпечує гнучкість, потужність і стабільність для розробки.

Даний фреймворк використовує ADO.NET для всіх операцій з базою даних, що забезпечує сумісність з усіма провайдерами даних, що підтримуються ADO.NET. Серед яких можна виділити: SQL Server, MySQL, Oracle та інші;

За допомогою LINQ to NHibernate, розробники можуть виконувати запити до даних, використовуючи синтаксис C# або Visual Basic, замість написання HQL (Hibernate Query Language) або SQL-коду. LINQ to NHibernate є розширенням LINQ, що використовується для підтримки запитів до бази даних NHibernate.

NHibernate може бути використаний в ASP.NET-додатках для роботи з базами даних. Завдяки своїй потужності та гнучкості, NHibernate часто використовується в складних веб-додатках, які потребують високого рівня керування даними [16].

В цілому, NHibernate є потужним інструментом для роботи з базами даних в екосистемі .NET, завдяки своїй глибокій інтеграції з ключовими компонентами платформи.

2.2.5 Сценарії використання NHibernate

До найбільш ключових сценаріїв використання NHibernate можна віднести:

– розробка веб-додатків. NHibernate широко використовується при розробці веб-додатків на ASP.NET. Він надає розробникам засоби працювати з базами даних за допомогою об'єктно-орієнтованого інтерфейсу, відстежуючи зміни у сесії та дозволяючи розробникам фокусуватись на бізнес-логіці, замість використання SQL запитів;

– сервіс-орієнтована архітектура (SOA). NHibernate є вибором для розробки сервіс-орієнтованих архітектур та розподілених систем, де потрібно гарантувати цілісність даних у розподілених транзакціях;

– розробка настільних додатків. NHibernate також може бути використаний для розробки настільних додатків на .NET. Він може спростити взаємодію з базою даних, здійснюючи більшу частину роботи з SQL і дозволяючи розробникам зосередитися на бізнес-логіці додатку;

– робота з великими даними. NHibernate також підтримує роботу з великими даними. Він дозволяє розробникам використовувати HQL-запити

для обробки великих наборів даних. Оптимізація продуктивності може бути досягнута за допомогою застосування стратегій "ліниве завантаження", кешування другого рівня і пакетних операцій.

Таким чином, NHibernate може слугувати як потужний інструмент в різноманітних сценаріях розробки, включаючи веб-додатки, сервіс-орієнтовані архітектури, настільні додатки, а також у сценаріях, що вимагають роботи з великими даними.

2.2.6 Переваги та недоліки NHibernate

NHibernate, як один з ключових фреймворків об'єктно-реляційного відображення (ORM) для платформи Microsoft .NET, має ряд значних переваг та викликів, які потрібно враховувати при його використанні.

До переваг цього фреймворку належать:

– високий рівень гнучкості. NHibernate є дуже гнучким рішенням, яке дозволяє використовувати багато стратегій відображення, включаючи наслідування, компоненти та збірки;

– підтримка багатьох СУБД. NHibernate підтримує велику кількість систем управління базами даних;

– кешування другого рівня. NHibernate включає кешування другого рівня, яке дозволяє зменшити навантаження на базу даних, покращуючи продуктивність застосунку;

– багато можливостей для оптимізації. Він має багато стратегій оптимізації, таких як "ліниве завантаження", пакетні операції, SQL запити і HQL.

До недоліків можна віднести:

– складність конфігурації. NHibernate є складним в налаштуванні, особливо при роботі з складними структурами даних або бізнес-логікою;

– високий поріг входу. NHibernate має високий поріг входу через свою складність. Його гнучкість призводить до більшої складності в порівнянні з іншими фреймворками ORM, що робить його важким для початківців;

– продуктивність. Хоча NHibernate має багато можливостей для оптимізації продуктивності, він може бути повільнішим, ніж деякі інші ORM, особливо при виконанні складних запитів;

– недостатня інтеграція з .NET. У порівнянні з Entity Framework, NHibernate має менш тісну інтеграцію з екосистемою .NET, що може ускладнити його використання у розробці деяких додатків.

У вирішенні деяких завдань NHibernate може виявитися найкращим вибором завдяки своїй гнучкості, але слід враховувати, що він вимагає високого рівня розуміння для ефективного використання.

2.3 Dapper

2.3.1 Основні особливості Dapper

Dapper, розроблений командою Stack Overflow, є мікро ORM для .NET, який забезпечує швидку обробку об'єктно-реляційного відображення (ORM) з мінімальною накладною витратою. Хоча він не пропонує такої ж широкої функціональності, як повноцінні ORM, такі як Entity Framework Core або NHibernate, він надає ряд ключових особливостей, серед яких можна виділити:

– мінімальна накладна витрата. Однією з головних відмінностей Dapper є його мінімальна накладна витрата. Він зосереджений на простоті і швидкості, надаючи тільки основний функціонал ORM без додаткових можливостей, які могли б збільшити накладну витрату;

– швидкість. Dapper відомий своєю високою швидкістю в порівнянні з більш важкими ORM. Завдяки своїй простоті і мінімальній накладній витраті, він може виконувати операції з даними значно швидше за деякі інші фреймворки;

– пряма робота з SQL. Dapper дозволяє розробникам прямо працювати з SQL, надаючи більше контролю над тим, як виконуються запити до бази даних. Це може допомогти оптимізувати продуктивність і дозволяє використовувати весь спектр SQL операторів;

– гнучкість відображення. Dapper дозволяє використовувати декілька стратегій відображення, включаючи пряме відображення результатів SQL запитів на POCO об'єкти, а також відображення через використання багатьох різних типів результатів;

– підтримка багатьох провайдерів баз даних. Dapper підтримує велику кількість провайдерів баз даних, включаючи SQL Server, SQLite, MySQL, PostgreSQL та Oracle;

– вбудовані можливості обробки транзакцій. Dapper включає вбудовану підтримку обробки транзакцій, що дозволяє розробникам легко використовувати цю важливу особливість баз даних в своїх застосунках.

Хоча Dapper не пропонує такого ж обсягу функціональності, як більш важкі ORM, його основні особливості роблять його ефективним інструментом для певних сценаріїв, де важливі швидкість і мінімальна накладна витрата [17].

2.3.2 Особливості створення моделі

Dapper надає розробникам можливість працювати безпосередньо з SQL, одночасно надаючи потужні засоби для простого відображення результатів SQL запитів на об'єкти. Особливості моделі Dapper представлено на рисунку 3.

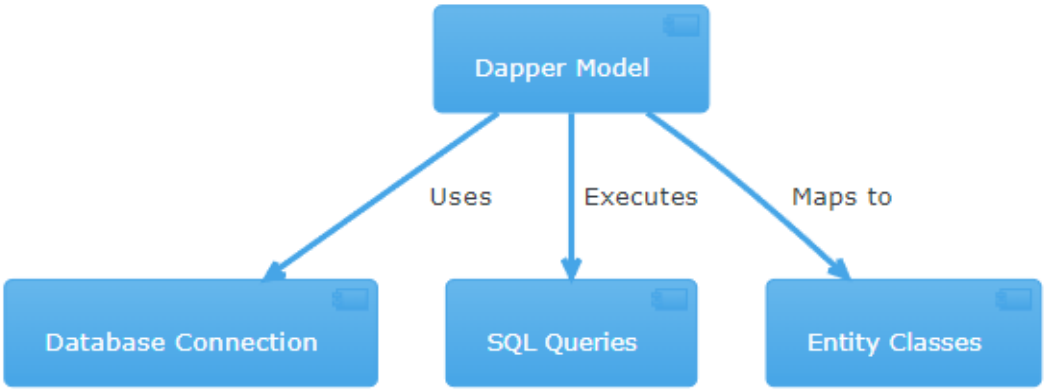

Рис. *3 Взаємодія компонентів при створенні моделі Dapper*

Модель Dapper складається із таких компонентів:

– Dapper Model (DapperModel). Це основний компонент, який використовує всі інші компоненти для створення та управління базою даних;

– Database Connection (DbConnection). DapperModel використовує DbConnection для встановлення з'єднання з базою даних. DbConnection є мостом між вашою моделлю Dapper та базою даних;

– SQL Queries (SQLQueries). DapperModel виконує SQL запити для маніпуляцій з даними в базі даних. Запити можуть бути на створення, читання, оновлення або видалення даних;

– Entity Classes (Entities). DapperModel використовує сутності для відображення результатів SQL запитів на об'єкти в коді. Кожна сутність відображається на рядок в результаті SQL запиту.

Отже, Dapper є мікро-ORM, який надає простий і гнучкий спосіб взаємодії з базою даних. Він не надає багато високорівневих функцій, які надають повнофункціональні ORM, такі як Entity Framework Core або NHibernate, але забезпечує високу продуктивність та гнучкість для виконання SQL запитів та мапінгу результатів на об'єкти.

2.3.3 Робота з даними

Організація запитів в Dapper відрізняється від більш традиційних ORM, таких як Entity Framework Core, і це пов'язано з тим, що Dapper не надає абстракцій високого рівня для моделювання або запитів до бази даних. Замість цього, він зосереджений на наданні ефективних і гнучких інструментів для роботи з SQL.

Однією з ключових особливостей Dapper є те, що створюється SQL запити безпосередньо як рядки в коді C#. Немає потреби використовувати LINQ або інший ORM-специфічний механізм для створення запитів. Це може зробити код більш доступним для розуміння тим, хто вже знайомий з SQL, і може забезпечити більший контроль над виконанням запитів.

Dapper підтримує параметризацію запитів, що дозволяє безпечно вставляти значення в SQL запити. Ви можете передати об'єкт, властивості якого відповідають параметрам запиту, а Dapper автоматично замінить параметри в SQL запиті відповідними значеннями.

Запити виконуються за допомогою методів Query, або асинхронний аналог QueryAsync (для запитів, які повертають результати) або Execute, чи асинхронний аналог ExecuteAsync (для запитів, які не повертають результати, таких як INSERT, UPDATE, DELETE). Ці методи виконують SQL запити, які ви передали, і повертають результати (якщо вони є).

Хоча Dapper не керує транзакціями автоматично, як деякі інші ORM, він надає підтримку для ручного управління транзакціями. Ви можете використовувати об'єкти IDbTransaction для створення транзакцій, виконання операцій в рамках цих транзакцій, і потім підтвердження або скасування транзакцій [18].

2.3.3.1 Організація CRUD-запитів

Хоча Dapper не має вбудованих механізмів для автоматичного створення CRUD запитів, як це мають деякі інші ORM, його простота та гнучкість дозволяють легко імплементувати ці операції вручну. Особливості CRUD операції Dapper включають:

– створення. Для створення нових записів в базі даних розробник може написати SQL запити використовуючи вставку INSERT. Запит може включати параметри, які відповідають властивостям об'єкту, що додається. Метод Execute, або асинхронний аналог ExecuteAsync, у Dapper використовується для виконання цього запиту;

– читання. Dapper надає метод Query, або асинхронний аналог QueryAsync, для виконання SQL запитів SELECT і повернення результатів. Можна використовувати цей метод для отримання даних від бази даних і проекції їх на об'єкти вашого домену;

– оновлення. Оновлення даних в базі даних можна виконати, використовуючи SQL запит UPDATE. Як і з іншими типами запитів, можна використовувати параметри для передачі даних, що будуть використовуватись в запиті;

– видалення. Видалення даних можна виконати за допомогою SQL запиту DELETE. Параметри можуть бути використані для вказівки критеріїв видалення.

Отже, Dapper дозволяє розробникам використовувати знайомий SQL синтаксис для виконання CRUD операцій, надаючи їм повний контроль над взаємодією з базою даних.

2.3.3.2 Запити до пов'язаних таблиць

Dapper, будучи мікро-ORM, не надає вбудованих механізмів для опису відношень між таблицями в схемі бази даних. На практиці це означає, що зв'язки між об'єктами потрібно створювати вручну, використовуючи SQL-запити [19]. Встановлювати наступні зв'язки між таблицями можна наступним чином:

– відношення "один до одного" та "один до багатьох" в Dapper можна створювати вручну. Наприклад, можна виконати окремі SQL запити для отримання "одного" об'єкта та його "багатьох" пов'язаних об'єктів. Потім можна вручну заповнити властивості "одного" об'єкта, що представляють "багатьох" об'єктів.

– відношення "багато до багатьох" в Dapper також потребують ручного управління. Вам потрібно виконати SQL запит, що включає JOIN між трьома таблицями (двома основними таблицями та проміжною таблицею), і потім вручну об'єднати результати в об'єкти.

– запити до декількох таблиць. Dapper використовує концепцію multiple mapping для виконання запитів до декількох таблиць. Це дозволяє вам здійснювати JOIN-операції та отримувати результати з кількох таблиць, які потім можуть бути відображені в складні об'єкти або колекції об'єктів.

Оскільки Dapper не використовує схему бази даних, стратегії завантаження, такі як "ліниве", "жадібне" або "явне" завантаження, не є прямо застосовними. Завантаження даних виконується тоді, коли виконується SQL запит, і все відбувається одразу. Якщо потрібно ліниве завантаження, програмісту потрібно буде самостійно реалізувати це, можливо, використовуючи деякі принципи засновані на шаблонах проектування.

Організація роботи з відношеннями між таблицями в Dapper вимагає більшого контролю і розуміння SQL, але надає більшу гнучкість і можливість оптимізації.

2.3.3.3 Організація транзакцій

Dapper, як мікро-ORM, не має вбудованого механізму для автоматичного управління транзакціями, на відміну від повноцінних ORM, таких як Entity Framework Core або NHibernate. Проте це не означає, що Dapper не може працювати з транзакціями. Насправді, він пропонує досить гнучкий підхід до їхньої організації, використовуючи засоби, надані самою платформою .NET.

Основний клас для роботи з транзакціями в .NET - це DbTransaction. Цей клас використовується Dapper для створення, виконання та керування транзакціями.

При роботі з Dapper, необхідно самостійно створювати та керувати транзакціями. Після відкриття з'єднання з базою даних, можна викликати метод BeginTransaction, щоб почати нову транзакцію. Тоді, всі операції з базою даних, які виконуються через це з'єднання, будуть виконуватися в контексті цієї транзакції. Після виконання всіх потрібних операцій, можна викликати метод Commit, щоб підтвердити транзакцію, або метод Rollback, щоб скасувати транзакцію.

Хоча цей підхід потребує більше коду порівняно з автоматичним управлінням транзакціями в повноцінних ORM, він надає більшу гнучкість та контроль над процесом. Оскільки можна точно визначити, коли транзакція має

бути відкрита, зобов'язана або скасована, що може бути корисним в ситуаціях, коли потрібно оптимізувати роботу з базою даних.

2.3.4 Інтеграція з .NET

Dapper, будучи одним з найшвидших ORM-фреймворків для платформи .NET, є дуже легким і гнучким, забезпечуючи при цьому високу продуктивність і ефективність роботи з базами даних.

Даний фреймворк активно використовує ADO.NET для всіх операцій з базою даних, що забезпечує сумісність з усіма провайдерами даних, що підтримуються ADO.NET, включаючи SQL Server, MySQL, Oracle та інші.

Він також надає розробникам можливість виконувати запити до баз даних, використовуючи звичний для них SQL-синтаксис, без потреби у вивченні специфічних мов запитів, як це може бути в інших ORM-фреймворках. Додатково, Dapper надає механізм для виконання складних запитів і мапінг результатів на об'єкти моделі, що дозволяє легко інтегрувати його з існуючими системами.

Dapper може бути використаний в ASP.NET-додатках для роботи з базами даних. Завдяки своїй легкості та продуктивності, Dapper часто використовується в додатках, які вимагають швидкого доступу до даних і високої продуктивності.

В загальному, Dapper є потужним інструментом для роботи з базами даних в екосистемі .NET, завдяки своїй простоті використання та гнучкості, а також глибокій інтеграції з ключовими компонентами платформи.

2.3.5 Сценарії використання Dapper

До найбільш ключових сценаріїв використання Dapper можна віднести:

– розробка веб-додатків. Dapper широко використовується при розробці веб-додатків на ASP.NET. Його гнучкість та швидкість роблять його від-

мінним вибором для веб-додатків, де потрібна висока продуктивність при взаємодії з базою даних, і де розробники хочуть мати більше контролю над SQL запитами;

– мікросервісна архітектура. Dapper ідеально підходить для використання в сервіс-орієнтованих архітектурах і мікросервісних додатках, завдяки своїй легкості і високій продуктивності. Він дозволяє ефективно виконувати операції з даними в розподілених системах, забезпечуючи високу продуктивність і стабільність;

– розробка настільних додатків. Dapper може бути використаний для розробки настільних додатків на .NET. Його простота та гнучкість можуть спростити взаємодію з базами даних, дозволяючи розробникам зосередитися на бізнес-логіці додатку;

– робота з великими даними. Dapper дозволяє розробникам ефективно виконувати операції з великими даними. Він надає можливість використовувати параметризовані SQL-запити для обробки великих наборів даних і прямо контролювати процес їх виконання, що допомагає оптимізувати продуктивність та забезпечує гнучкість для виконання складних операцій з даними.

Отже, Dapper виявляється високоефективним та гнучким рішенням для широкого спектру сценаріїв використання, від веб-додатків до роботи з великими даними, пропонуючи розробникам більший контроль та гнучкість.

2.3.6 Переваги та недоліки Dapper

У процесі дослідження ORM в рамках платформи Microsoft .NET, важливо розглянути і переваги, і недоліки різних фреймворків. До них належить Dapper, мікро-ORM, що надає розробникам можливість використовувати низькорівневі SQL запити з автоматичним відображенням результатів на об'єкти [20].

Переваги Dapper включають:

– висока продуктивність. Однією з найочевидніших переваг Dapper є його висока продуктивність. Dapper виконує SQL запити швидше за багато

інших ORM, включаючи Entity Framework і NHibernate, завдяки своєму прямому підходу до взаємодії з базами даних;

– простота використання. Dapper не вимагає від розробників налаштування метаданих або навчання новим запитам. Замість цього, Dapper дозволяє розробникам використовувати звичайний SQL, роблячи його легким для вивчення і використання;

– гнучкість. Dapper не накладає обмежень на схему бази даних і не вимагає від розробників використання певного стилю кодування. Це дозволяє розробникам мати більше контролю над структурою бази даних і архітектурою додатку;

– підтримка багатьох баз даних. Dapper підтримує велику кількість різних баз даних, включаючи SQL Server, MySQL, SQLite і PostgreSQL. Це робить його універсальним інструментом для розробки.

Недоліки Dapper складають:

– обмежена підтримка автоматичної міграції схеми бази даних. Відмінно від багатьох інших ORM, Dapper не надає автоматичної підтримки міграції схеми бази даних. Розробники мають самостійно керувати змінами в структурі бази даних;

– відсутність підтримки "ліниве завантаження". Dapper не підтримує "ліниве завантаження", техніку завантаження даних на запит, яка часто використовується в інших ORM для оптимізації продуктивності;

– необхідність написання більше коду. Оскільки Dapper не надає автоматичних засобів для створення, оновлення, видалення або отримання записів, розробникам часто потрібно написати більше SQL-коду, ніж при використанні інших ORM;

– відсутність підтримки LINQ. Dapper не підтримує LINQ, популярного запитного мови, яка інтегрована в C# і дозволяє розробникам виконувати запити до даних безпосередньо з коду.

З урахуванням всіх цих переваг та недоліків, Dapper є відмінним вибором для сценаріїв, де важлива висока продуктивність, простота використання і контроль над взаємодією з базою даних.

Висновки з розділу 2

В процесі дослідження особливостей фреймворків об'єктно-реляційного відображення для платформи .NET було проаналізовано три основних представника цієї категорії: Entity Framework Core, NHibernate та Dapper.

Кожний із цих фреймворків має свою унікальну сферу застосування, використовуючи свої власні переваги та унікальні функції. Entity Framework Core, в основному, використовується для розробки великих застосунків з великою кількістю бізнес-логіки, завдяки своїм можливостям створення складних моделей і впровадження потужних механізмів для роботи з даними. NHibernate зазначається своєю гнучкістю та можливістю інтеграції в різноманітні архітектури, включаючи веб-додатки, настільні додатки та сервіс-орієнтовані архітектури. Dapper же відрізняється своєю простотою та продуктивністю, будучи ідеальним вибором для задач, де потрібен максимальний контроль над SQL-запитами.

Отже, при виборі ORM-фреймворка для платформи .NET розробникам слід враховувати не лише потреби конкретного проекту, але й особливості роботи кожного з цих фреймворків. Вибір певного рішення відбувається на основі балансу між функціональністю, продуктивністю, простотою використання та іншими критеріями, які важливі для конкретного проекту.

РОЗДІЛ 3 ВИБІР ТЕХНОЛОГІЙ ПРОЕКТУВАННЯ ТА ОСОБЛИВОСТІ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Вибір технологій

На етапі попередньої підготовки до розробки програмного забезпечення, особливо важливим стає процес детального дослідження потенційних технологій для створення додатків. Цей процес має на меті визначення технології, що найкраще відповідає вимогам поставленої задачі та здатна найефективніше вирішити виниклу проблему.

Проведення аналізу потенційних технологій для розробки додатків включає в себе вивчення та оцінювання широкого спектру важливих факторів. Серед них варто виділити: обсяг функціональних можливостей, продуктивність, масштабованість, зручність у процесі розробки та подальшої підтримки, наявність розроблених інструментів та їх екосистема, доступ до необхідних ресурсів, та інші аспекти, які можуть суттєво впливати на успішність та ефективність розробки.

Особливої уваги в контексті даної теми заслуговують об'єктно-реляційні відображення для платформи Microsoft .NET. Вибір конкретного ORM фреймворка може значно вплинути на всі вищезгадані параметри, тому його варто розглядати як ключовий етап підготовчої фази проекту.

3.1.1 Обґрунтування вибору мови програмування

Селекція відповідної мови програмування є вирішальним етапом у процесі проектування програмного забезпечення.

Python — це мова програмування високого рівня, інтерпретована, з динамічною типізацією, розроблена Гвідо ван Россумом у 1991 році. Основою її популярності стали простота і лаконічність синтаксису, що робить Python ідеальним вибором для тих, хто лише знайомиться з програмуванням. Python ши-

роко застосовується у веб-розробці, автоматизації процесів, наукових дослідженнях, аналітиці даних, розробці штучного інтелекту та машинного навчання [21].

Java — це мова програмування високого рівня, об'єктно-орієнтована, яку винайшла компанія Sun Microsystems у 1995 році. Завдяки принципу "Напиши одного разу, запускай будь-де" та застосуванню віртуальної машини Java (JVM), Java працює на численних платформах. Ця мова широко використовується для розробки корпоративних і веб-додатків, мобільних додатків на платформі Android, а також в системах вбудованих пристроїв і Інтернету речей [22].

C# — це об'єктно-орієнтована мова програмування, розроблена компанією Microsoft у 2000 році як складова платформи .NET. C# втілює переваги мов програмування C++ та Java. Її використовують для створення різноманітних додатків, включаючи веб-додатки, мобільні додатки, додатки для операційної системи Windows, ігри (особливо з використанням движка Unity) та корпоративні програмні рішення та хмарні сервіси. C# працює в середовищі .NET, яке пропонує великий асортимент бібліотек та фреймворків для швидкої та ефективної розробки програмного забезпечення [23].

Таблиця 1 представляє порівняльний аналіз функціональних характеристик кожної з розглянутих мов програмування.

Таблиця *1*

Порівняльний аналіз мов програмування

Характеристика	Python	Java	C#
Типізація	Динамічна	Статична	Статична
Парадигма програмування	Об'єктно-орієнтована, процедурна	Об'єктно-орієнтована	Об'єктно-орієнтована, компонентна
Модель виконання	Інтерпретація	Компіляція до байт-коду, JVM	Компіляція до проміжного коду, CLR

Стиль синтаксису	Мінімалістичний, відступи для блоків коду	Вплив C++, формалізований	Вплив C++, Java, формалізований
Фреймворки для веб-розробки	Django, Flask, FastAPI	Jakarta, Spring, JSF	ASP.NET, Blazor
Розробка мобільних додатків	Kivy, BeeWare, Chaquopy (Android)	Android SDK	Xamarin, .NET MAUI
Підтримка крос-платформності	Висока	Висока	Висока (через .NET)
Фреймворки для роботи з СУБД	SQLAlchemy, Peewee, Django ORM	JPA, Hibernate, TopLink, EclipseLink, MyBatis	Entity Framework, NHibernate, Dapper

Аналіз характеристик трьох мов програмування - Python, Java та C# - в контексті потреб проекту показав, що кожна з цих мов має свої особливості та переваги. Проте в рамках даного дослідження, особливо з огляду на тему "Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .Net", мова C# є найбільш відповідним вибором.

Вибір C# обґрунтовується наступними чинниками:

– об'єктно-орієнтований та компонентний стилі програмування, що притаманні C#, відповідають потребам роботи з об'єктно-реляційним відображенням, що стоїть в центрі нашого дослідження;

– підтримка фреймворків для роботи з СУБД, таких як Entity Framework Core, NHibernate та Dapper, дозволяє відповідно використовувати та аналізувати фреймворки об'єктно-реляційного відображення в контексті C#;

– крос-платформність. Завдяки платформі .NET, C# може бути використана для розробки додатків для різних платформ, включаючи Windows, Linux, macOS, а також для мобільних платформ через Xamarin або .NET MAUI;

– спільнота та підтримка. C# має велику та активну спільноту розробників, велику кількість ресурсів для навчання та широку підтримку від Microsoft. Це забезпечує доступ до постійно оновлюваних бібліотек та фреймворків, а також до кваліфікованої підтримки.

Таким чином, з урахуванням цих факторів, мова програмування C# є оптимальним вибором для проведення даного дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET.

3.1.2 Обґрунтування вибору середовища розробки

Відбір відповідного інструментарію для розробки є критичним етапом, який впливає на ефективність і швидкість реалізації проекту. Для конструювання системи, яка буде розроблена з допомогою мови C#, можна розглянути такі інтегровані середовища розробки (IDE), як Microsoft Visual Studio [24], Visual Studio Code [25] і JetBrains Rider [26].

Microsoft Visual Studio є потужним інтегрованим середовищем розробки, яке підтримує широкий спектр мов програмування, включаючи C++, C#, Visual Basic, F# та інші. Дане середовище надає розробникам зручні інструменти для створення додатків для Windows, мобільних платформ та веб-сервісів. Visual Studio включає продумані механізми для редагування коду, дебагінга, профілювання, тестування та візуального проектування.

Visual Studio Code представляє собою легкий і продуктивний редактор коду з укомплектованими засобами для редагування, дебагу та підтримки великого числа мов програмування та фреймворків. Редактор має вбудовану підтримку Git, віртуальних середовищ, додатків, а також інших інструментів, що значно полегшують роботу над проектами для програмістів.

JetBrains Rider – це інтегроване середовище розробки, яке підтримує велику кількість мов програмування, зокрема C#, VB.NET, F#, JavaScript, TypeScript, CSS та HTML. JetBrains Rider пропонує широкий спектр засобів для відладки, профілювання та тестування, а також містить численні корисні функції, такі як автоматичне визначення типів, підсвічування синтаксису

тощо. Це середовище відмінно інтегрується з іншими інструментами JetBrains, як-от ReSharper, dotTrace і dotMemory, що забезпечує зростання продуктивності та покращення якості коду.

У таблиці 2 наведено порівняльний аналіз вищезазначених інтегрованих середовищ розробки (IDE).

Таблиця *2*

Порівняльний аналіз середовищ розробки

Критерій	Microsoft Visual Studio	Visual Studio Code	JetBrains Rider
Розробник	Microsoft	Microsoft	JetBrains
Підтримувані мови програмування	C#, VB.NET, C++, F#, тощо	Багатомовний	C#, VB.NET, F#
Темплейти проектів	Повна підтримка	Обмежена підтримка через розширення	Повна підтримка
Інтелектуальне завершення коду	Так	Так	Так
Можливості для відлагодження	Так	Так	Так
Підтримка рефакторинга	Так	Обмежена	Так
Інструменти для тестування	Так	Обмежена підтримка	Так
Інтеграція з Git	Так	Так	Так
Підтримка фреймворків для СУБД	Entity Framework Core, Dapper та ін.	В основному через розширення	Entity Framework Core, Dapper та ін.

Аналізуючи розглянуті інтегровані середовища розробки, важливо зазначити, що кожне з них має свої переваги та особливості. Втім, в контексті порівняльного дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET, Microsoft Visual Studio виявляється найбільш підходящим варіантом.

Основні аргументи на користь Microsoft Visual Studio полягають у наступному:

– підтримка мови C# та .NET. Як офіційний продукт Microsoft, Visual Studio пропонує оптимізовану та інтегровану підтримку для мови програмування C# та фреймворку .NET;

– повна підтримка шаблонів проектів. Visual Studio має широку бібліотеку шаблонів проектів, яка значно спрощує ініціалізацію нових проектів;

– робота з фреймворками СУБД. Visual Studio підтримує використання таких популярних фреймворків, як Entity Framework Core та Dapper, що забезпечують потужні можливості для об'єктно-реляційного відображення;

– графічні інструменти для дизайну. Вбудовані графічні дизайнери, такі як Windows Forms та WPF, можуть бути корисними для розробки користувацького інтерфейсу;

– розширюваність. Visual Studio підтримує велику кількість додаткових розширень і плагінів, що дозволяє індивідуалізувати середовище розробки під специфічні вимоги проекту;

– інтеграція з іншими продуктами Microsoft. Це включає безпосередню інтеграцію з такими сервісами, як Azure, Team Foundation Server, SQL Server тощо.

Тому, на основі цих характеристик, Microsoft Visual Studio є оптимальним вибором для розробки даного проекту на платформі .NET, використовуючи фреймворки об'єктно-реляційного відображення.

3.1.3 Обґрунтування вибору СУБД

Правильно вибрана система управління базами даних відіграє ключову роль у успіху будь-якого проекту, який залежить від ефективного зберігання та обробки даних. Враховуючи різноманітність СУБД, доступних на ринку, критично важливим є розуміння специфічних потреб проекту і можливостей, що надає кожна система.

MS SQL Server — це система управління реляційними базами даних, створена Microsoft, яка пропонує ряд важливих переваг. Вона підтримує стандартні мови запитів, як-то SQL та T-SQL, що сприяє гнучкій та продуктивній роботі з даними. Основні характеристики MS SQL Server включають високу процесорну потужність та здатність обробляти значні обсяги даних, що робить її відмінним вибором для проектів великого масштабу [27].

MySQL, яка є відкритою реляційною СУБД, забезпечує високу продуктивність та надійність. Вона є частим вибором для веб-додатків та проектів, що обробляють великі набори даних. MySQL підтримує різні типи баз даних, включаючи стовпчиково-орієнтовані та NoSQL. Однак, для початківців її може бути важко освоїти, і вона має деякі обмеження щодо обробки специфічних типів даних [28].

Oracle — одна з найбільш поширених комерційних СУБД, яка також використовує мову запитів SQL. Вона відзначається масштабованістю та потужністю, що дозволяє обробляти великі набори даних та велику кількість запитів. Oracle пропонує широкий спектр функцій, включаючи захист даних та підтримку транзакцій. Проте, її висока складність та вартість можуть бути перешкодою для менш масштабних проектів [29].

Таблиця 3 наводить порівняльний аналіз характеристик згаданих СУБД, що допоможе при розробці стратегії зберігання даних та дослідження роботи фреймворків.

Таблиця *3*

Порівняльний аналіз сховищ БД

Функція / Властивість	MS SQL Server	MySQL	Oracle
Тип ліцензії	Комерційна	Відкрита (GPL) або Комерційна	Комерційна
Доступні версії	Express, Standard, Enterprise	Community, Standard, Enterprise	Standard, Enterprise, Express
Підтримувані платформи	Windows	Windows, Linux, macOS	Windows, Linux, Unix, macOS
Продуктивність	Висока	Висока	Надзвичайно висока
Масштабування	Відмінне	Високе	Відмінне
Підтримка розподіленої обробки	Так	Так	Так
Відкритість API	Відсутня	Присутня	Відсутня
Підтримка резервного копіювання	Так	Так	Так
Використання ORM фреймворків	Entity Framework, Dapper, NHibernate	MySQL Connector/NET, Dapper, NHibernate	ODP.NET, NHibernate, Dapper

Порівняльний аналіз систем управління базами даних демонструє, що кожна система має свої особливості. У контексті розробки проекту, який заснований на платформі Microsoft .NET і використовує об'єктно-реляційне відображення, MS SQL Server виявляється найбільш придатною СУБД.

Перш за все, MS SQL Server має глибоку інтеграцію з платформою .NET, що спрощує розробку та впровадження. За допомогою T-SQL та .NET він надає широкий спектр можливостей для виконання складних запитів та обробки даних.

Другим важливим аспектом є масштабованість MS SQL Server. Він здатний ефективно обробляти великі обсяги даних, що є критичним для багатьох сучасних застосунків.

Третім фактором є підтримка ORM фреймворків, як-от Entity Framework Core, Dapper і NHibernate. Це дозволяє розробникам працювати з базами даних на більш високому рівні абстракції, що сприяє продуктивності та якості коду.

Таким чином, враховуючи вимоги до даного проекту та об'єктивні переваги MS SQL Server, ця система управління базами даних є оптимальним вибором для його реалізації.

3.2 Вимоги до програмного забезпечення для порівняльного аналізу

3.2.1 Функціональні вимоги

Для достовірного порівняльного аналізу фреймворків об'єктно-реляційного відображення (ORM) на платформі Microsoft .NET викладено ряд функціональних вимог, які необхідно виконати в рамках даного дослідження:

– розробка концептуальної моделі даних. Потрібно створити універсальну модель даних, яка б максимально чітко демонструвала особливості роботи з різними типами відношень між сутностями в ORM-фреймворках. Модель повинна включати в себе відношення типу "один-до-одного", "один-до-багатьох", "багато-до-багатьох". Від цієї моделі будуть відправлятися усі наступні етапи дослідження;

– реалізація моделі в ORM-фреймворках. Необхідно відобразити створену концептуальну модель даних на реляційну базу даних за допомогою кожного з вивчених ORM-фреймворків. Таким чином, буде можливість прямо

порівняти як роботу різних фреймворків з одними й тими ж даними, так і їхню можливість відображати різні типи відношень;

– розробка набору типових операцій з даними. Потрібно створити набір типових CRUD-операцій (створення, читання, оновлення, видалення даних), які будуть виконуватися за допомогою кожного з ORM-фреймворків. Це дозволить оцінити не тільки продуктивність, але і зручність роботи з ORM-фреймворками для виконання базових операцій з даними;

– вимірювання продуктивності. Виконання розроблених CRUD-операцій в кожному з ORM-фреймворків та збір метрик продуктивності. Метрики продуктивності мають включати час виконання операцій, використання пам'яті, CPU-навантаження. Ці дані будуть використані для об'єктивного порівняння ORM-фреймворків;

– розробка інтерфейсу користувача. Потрібно розробити графічний інтерфейс користувача, який би дозволяв демонструвати роботу ORM-фреймворків та відображати результати вимірювання продуктивності. Інтерфейс має бути інтуїтивно зрозумілим і зручним для користування, що дозволить використовувати дослідження не тільки фахівцям у даній області, але і широкому колу зацікавлених осіб.

3.2.2 Вимоги до проектування бази даних

Під час проведення порівняльного аналізу фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET, вимоги до проектування бази даних відіграють критичну роль. Для якісного виконання аналізу та отримання достовірних результатів, необхідно дотримуватися наступних вимог:

– модельність. Модель даних, яка буде використовуватися для порівняльного аналізу, має бути універсальною та добре структурованою. Вона повинна включати різні типи відношень, включаючи "один до одного", "один до багатьох", та "багато до багатьох". Використання різних типів відношень допоможе адекватно оцінити можливості та гнучкість кожного з ORM-фреймворків;

– нормалізація. Для забезпечення ефективності роботи з базою даних та мінімізації дублювання даних, модель бази даних має бути нормалізованою. Це означає, що кожна таблиця має містити лише пов'язані дані та має унікальний ключ, який ідентифікує кожний запис;

– інтеграція. Кожен з ORM-фреймворків повинен мати змогу відображати створену модель бази даних без будь-яких обмежень або додаткових модифікацій;

– продуктивність. База даних має бути спроектована та оптимізована таким чином, щоб максимізувати продуктивність. Це включає правильне використання індексів для поліпшення швидкості запитів та використання відповідних типів даних для зменшення використання ресурсів;

– масштабованість. Модель бази даних має бути масштабованою, тобто мати змогу ефективно працювати з великими обсягами даних. Це важливо для адекватного вимірювання продуктивності ORM-фреймворків.

Дотримання цих вимог допоможе створити базу даних, яка буде ефективною, гнучкою та здатною до адаптації до різних ORM-фреймворків, що дозволить провести об'єктивний і дієвий аналіз.

3.2.3 Вимоги до ORM фреймворків для порівняння

Для проведення об'єктивного порівняння фреймворків об'єктно-реляційного відображення на платформі Microsoft .NET необхідно визначити відповідні вимоги. Вимоги до ORM фреймворків можуть включати наступне:

– повнота відображення. Фреймворк має надавати змогу відображення всіх ключових аспектів моделі даних у реляційну базу даних. Це включає відображення різних типів відношень, таких як "один до одного", "один до багатьох" та "багато до багатьох";

– підтримка CRUD операцій: Фреймворк має надавати механізми для виконання основних операцій з даними: створення (Create), читання (Read), оновлення (Update) та видалення (Delete);

– продуктивність. Фреймворк повинен мати високу продуктивність при виконанні CRUD операцій, зокрема при великих обсягах даних;

– легкість використання. Фреймворк має бути зручним у використанні, із інтуїтивно зрозумілими механізмами відображення та запитів;

– сумісність. Фреймворк повинен бути сумісний з платформою Microsoft .NET та СУБД MS SQL Server, що використовується в даному дослідженні;

– підтримка транзакцій. Фреймворк повинен надавати можливість управління транзакціями, що дозволяє забезпечити цілісність та надійність даних;

– підтримка кешування. Фреймворк має надавати можливість кешування даних, що може підвищити продуктивність за рахунок зменшення кількості запитів до бази даних;

– масштабованість. Фреймворк має підтримувати масштабованість, дозволяючи ефективно обробляти збільшення обсягу даних та користувацького навантаження;

– підтримка асинхронних запитів; Фреймворк має забезпечити максимальну продуктивність за рахунок багатопотокового виконання.

3.3 Створення моделі та запитів для порівняльного аналізу

Проведення об'єктивного порівняльного аналізу ORM-фреймворків потребує створення стандартної моделі даних, яка використовуватиметься в рамках кожного дослідження. Ця модель даних має бути досить складною, щоб демонструвати можливості відображення різних відношень між об'єктами ("один до одного", "один до багатьох", "багато до багатьох"). Деталі моделі, такі як типи даних, обмеження, індекси та інші параметри, мають бути чітко визначені та консистентні в усіх сценаріях дослідження.

3.3.1 Проектування бази даних. ERD діаграма

Проектування бази даних – це сукупність процесів, спрямованих на визначення структури даних, які будуть зберігатися в базі даних. Це включає в себе аналіз доменної області, визначення ключових сутностей та відношень між ними, вибір відповідних типів даних для атрибутів та розробку схеми, яка відображає всю цю інформацію в структурованому та легко зрозумілому вигляді.

У контексті цього проекту, завдання проектування бази даних включає розробку моделі, яка відображає доменну область системи управління персоналом. Модель бази даних складається з п'яти основних таблиць: Departments, EmployeeDetails, EmployeeProject, Employees, Projects та LogEntries.

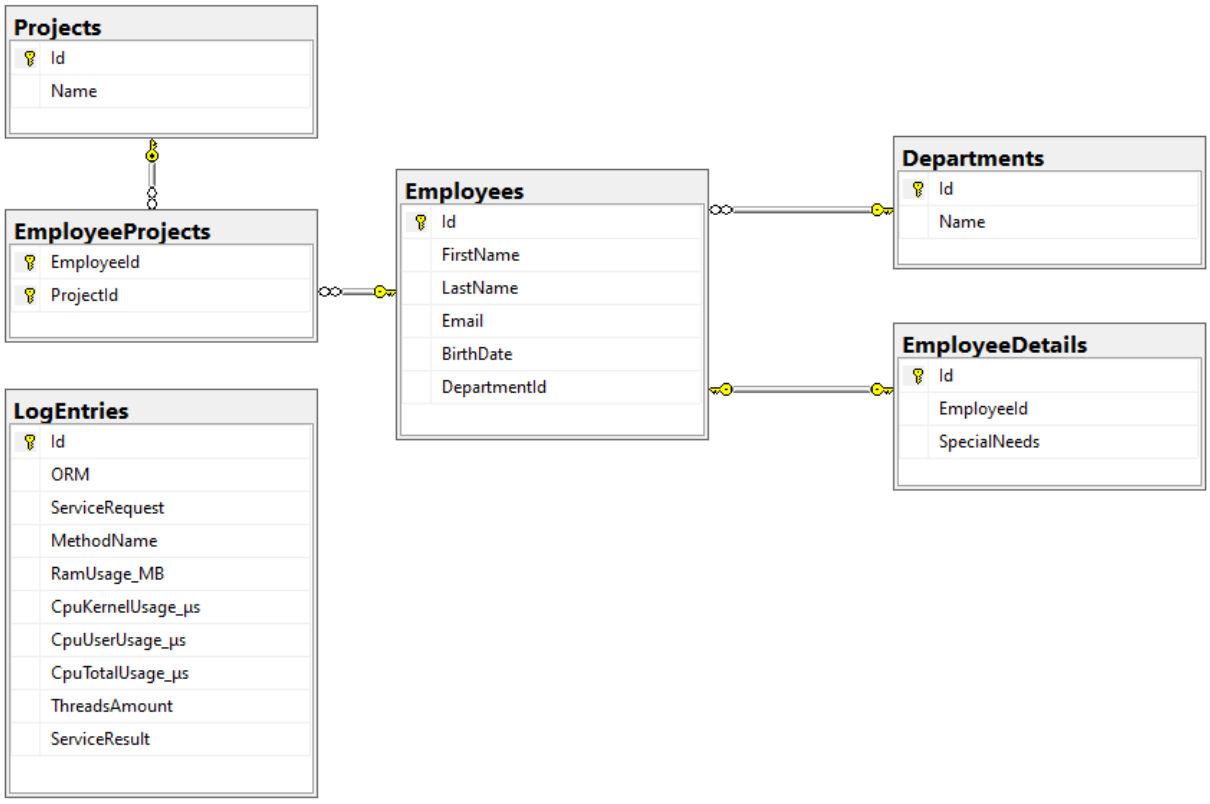

Рис. *4 Діаграма бази даних*

Кожна з цих таблиць представляє окрему сутність і містить ряд атрибутів, що відображають властивості цієї сутності:

– таблиця "Departments" містить два поля: "Id" (первинний ключ, автоінкремент) та "Name" (назва відділу, тип даних - nvarchar(100));

– таблиця "EmployeeDetails" включає поля "Id" (первинний ключ, автоінкремент), "EmployeeId" (ідентифікатор співробітника, тип даних - int), "SpecialNeeds" (національний ідентифікаційний номер, тип даних - nvarchar(20));

– таблиця "EmployeeProject" визначає відношення багато-до-багатьох між співробітниками та проектами. Вона включає два поля: "EmployeeId" та "ProjectId", які разом формують первинний ключ;

– таблиця "Employees" містить поля "Id" (первинний ключ, автоінкремент), "FirstName" (ім'я співробітника, тип даних - nvarchar(50)), "LastName" (прізвище співробітника, тип даних - nvarchar(50)), "Email" (електронна адреса співробітника, тип даних - nvarchar(max)), "BirthDate" (дата народження, тип даних - datetime) та "DepartmentId" (ідентифікатор департаменту, тип даних - int);

– таблиця "Projects" містить поля "Id" (первинний ключ, автоінкремент) та "Name" (назва проекту, тип даних - nvarchar(100)).

– таблиця "LogEntries" містить поля "Id" (первинний ключ, автоінкремент), "ORM" (назва ORM яка була задіяна при обробці запиту, тип даних - nvarchar(20)), "ServiceRequest" (модель запиту, тип даних - nvarchar(max)), "MethodName" (виконавчий метод, який був задіяний, для обробки запиту, тип даних - nvarchar(100)), "RamUsage_MB" (зарезервована оперативна пам'ять під виконання функціоналу ORM, тип даних - nvarchar(100)), "CpuKernelUsage_μs" (час ядра, який процесор витратив на виконання операцій, які здійснюються в просторі ядра операційної системи, під час виконання функціоналу ORM, тип даних - bigint), "CpuUserUsage_μs" (час користувача, який процесор витрачає на виконання програм, які працюють у просторі користувача. Це включає в себе виконання коду самої програми, а також взаємодію з системними службами через системні виклики, тип даних - bigint), "CpuTotalUsage_μs" (загальний процесорний час на виконання ORM, час ядра

+ час користувача,тип даних - bigint), "ThreadsAmount" (кількість активних потоків, які задіяв застосунок, тип даних - int), "ServiceResult" (модель, яку вертає застосунок після виконання запиту, якщо така існує, тип даних - nvarchar(max)). Ця таблиця не є частиною моделі доменну області системи управління персоналом, натомість вона зберігає метрики витрачених ресурсів на виконання функціоналу ORM фреймворків під час обробки різних сценаріїв запитів користувача.

Відношення між сутностями представлені у вигляді ліній з маркерами, які вказують на тип відношення. В даному проекті організовані наступні відношення:

– сутності "Employees" і "Departments" зв'язані відношенням "багато-до-одного", яке показує, що кожен співробітник може належати лише до одного відділу, але відділ може містити багато співробітників;

– сутності "Employees" і "EmployeeDetails" пов'язані відношенням "один-до-одного", що відображає, що кожен співробітник має лише одну детальну інформацію;

– сутності "Employees" і "Projects" пов'язані відношенням "багато-до-багатьох" через допоміжну таблицю "EmployeeProject", яка відображає факт того, що співробітники можуть бути включені в декілька проектів і проекти можуть мати декілька учасників.

Після завершення етапу проектування, була розроблена діаграма відношень сутностей для бази даних. Ця діаграма візуалізує структуру та зв'язки між окремими таблицями та їх полями (рис. 4). Створена модель впроваджується як суттєвий інструмент у процес проектування та подальшої розробки функціональності додатку, а також у процесі порівняльного аналізу об'єктно-реляційних фреймворків для платформи Microsoft .NET.

У додатку "А" наведено скрипт створення бази даних з усіма таблицями та зв’язками.

3.3.2 Розробка основних запитів для ORM фреймворків

Для проведення ефективного порівняльного аналізу роботи об'єктно-реляційних відображень, трійка вибраних фреймворків - EFC, NHibernate та Dapper - була ретельно досліджена з огляду на реалізацію основних CRUD операцій (створення, читання, оновлення, видалення). Цей аналіз надає важливу інформацію щодо зручності написання коду, продуктивності, а також ступеня оптимізації, яку забезпечують розглянуті ORM-фреймворки.

Для цього було реалізовано наступні запити:

– створення (Create). Досліджено, як кожен з трьох фреймворків дозволяє створювати нові записи в таблицях. Для цього використовувалася процедура додавання нового співробітника, відділу, проекту або інших об'єктів, представлених в базі даних. Головною метою було визначити простоту і зручність створення нового об'єкта і його збереження в базі даних за допомогою EFC, NHibernate та Dapper;

– читання (Read). Для оцінки зручності виконання операцій читання даних, були розроблені запити, які дозволяють отримувати інформацію про конкретний об'єкт за його ідентифікатором, а також виконувати комплексний пошук об'єктів за різними параметрами за допомогою кожного з ORM-фреймворків;

– оновлення (Update). При оновленні даних кожний фреймворк повинен забезпечувати зміну атрибутів об'єкта, а також оновлення зв'язків між об'єктами. Було проведено аналіз, наскільки просто та зручно оновлювати дані за допомогою EFC, NHibernate та Dapper;

– видалення (Delete). Останнім етапом дослідження було оцінювання процедур видалення об'єктів і їх залежностей у кожному з вибраних ORM-фреймворків.

Виконання цих операцій у EFC, NHibernate та Dapper дозволило провести безпосереднє порівняння їх роботи, продуктивності, а також визначити, який з фреймворків надає оптимальний баланс між продуктивністю та зручністю використання.

3.3.2.1 Розробка запитів для фреймворку Entity Framework Core

В рамках проекту, під час роботи з фреймворком Entity Framework Core, основна увага приділялася створенню різних типів запитів, які необхідні для реалізації CRUD операцій та роботи з різними типами зв'язків між сутностями. Цей фреймворк використовує LINQ для формування запитів, що надає гнучкість та високу продуктивність при виконанні різноманітних операцій з даними.

Для створення нових записів, на прикладі типу Employee, було використано асинхронний метод AddRangeAsync() до DbSet із DbContext, після чого були застосовані зміни за допомогою асинхронного методу SaveChangesAsync() (лістинг 1).

Лістинг *1 Реалізація додавання нових працівників у БД*

```
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    await context.Employees.AddRangeAsync(employees);
    await context.SaveChangesAsync(cancellationToken);
    await context.DisposeAsync();
}
```

Для читання записів з однієї таблиці, на прикладі Employees, було використано метод Select() до відповідного DbSet та асинхронний метод ToListAsync (), для перетворення отриманої послідовності у список (лістинг 2).

Лістинг *2 Реалізація запиту працівників з БД (запит з однієї таблиці)*

```
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    employees = await context.Employees
        .Select(_ => new EmployeeModel()
        {
            EmployeeId = _.Id,
            BirthDate = _.BirthDate,
```

```
                Email = _.Email,
                FirstName = _.FirstName,
                LastName = _.LastName
            }).ToListAsync(cancellationToken);
    await context.DisposeAsync();
}
```

Для читання записів з таблиць, які мають відношення один до одної, Employees та EmployeeDetails, було також використано метод Select() до DbSet Employees, створено додатковий підзапит до зв’язаної сутності EmployeeDetail та викликано асинхронний метод ToListAsync() для перетворення отриманої послідовності у список (лістинг 3).

Лістинг *3 Реалізація запиту працівників та додаткової їх інформації з БД (запит до таблиць з відношенням "один до одного")*

```
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    employeesWithDetails = await context.Employees
        .Include(_ => _.EmployeeDetail)
        .Select(e => new EmployeeWithDetails()
        {
            Id = e.Id,
            FirstName = e.FirstName,
            LastName = e.LastName,
            Email = e.Email,
            BirthDate = e.BirthDate,
            SpecialNeeds = e.EmployeeDetail == null ?
"" : e.EmployeeDetail.SpecialNeeds
        }).ToListAsync(cancellationToken);

    await context.DisposeAsync();
}
```

Для читання записів з таблиць, які мають відношення один до багатьох, Departments та Employees, було створено запит до DbSet Departments, створено підзапит за допомогою метода Include() до DbSet Employees, викликано метод

Select() та асинхронний метод ToListAsync() для перетворення отриманої послідовності у список (лістинг 4).

Лістинг *4 Реалізація запиту відділів за працівниками з БД (запит до таблиць з відношенням "один до багатьох")*

```
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    departmentsWithEmployees = await
context.Departments
        .Include(_ => _.Employees)
        .Select(d => new DepartmentWithEmployeesModel()
        {
            DepartmentId = d.Id,
            DepartmentName = d.Name,
            Employees = d.Employees.Select(e => new
EmployeeModel()
            {
                EmployeeId = e.Id,
                FirstName = e.FirstName,
                LastName = e.LastName,
                Email = e.Email,
                BirthDate = e.BirthDate
            })
        }).ToListAsync(cancellationToken);

    await context.DisposeAsync();
}
```

Для читання записів з таблиць, які мають відношення багато до багатьох, Projects та Employee, було створено запит до DbSet Employees, створено підзапит за допомогою метода Include() до DbSet EmployeeProjects, та ще один підзапит до зв’язаної сутності Project, за допомогою метода ThenInclude(), далі методом Select() та асинхронним методом ToListAsync() утворюється необхідна колекція об’єктів (лістинг 5).

Лістинг 5 *Реалізація запиту працівників та їх проектів з БД (запит до таблиць з відношенням "багато до багатьох")*

```
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    employeesWithProjects = await context.Employees
        .Include(e => e.EmployeeProjects)
            .ThenInclude(ep => ep.Project)
        .Select(e => new EmployeeWithProjectsModel
        {
            EmployeeId = e.Id,
            FirstName = e.FirstName,
            LastName = e.LastName,
            Projects = e.EmployeeProjects.Select(ep =>
new ProjectModel
            {
                ProjectId = ep.Project.Id,
                ProjectName = ep.Project.Name
            })
        })
        .ToListAsync(cancellationToken);

    await context.DisposeAsync();
}
```

Для оновлення записів у таблицях, на прикладі Departments, було використано метод ExecuteUpdateAsync який оновлює записи у таблиці Departments без необхідності їх отримання з бази, перед оновленням (лістинг 6).

Лістинг 6 *Реалізація оновлення назв відділів у БД*

```
Dictionary<string, int> response = new();
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    var departmentsCount = await context.Departments
        .ExecuteUpdateAsync(_ => _.SetProperty(__ =>
__.Name, __ => prefix + ": " + __.Name));
    response.Add("Amount of updated departments: ",
departmentsCount);
    await context.DisposeAsync();
```

```
}

return response;
```

Для видалення об'єктів у таблицях, на прикладі Employees, було створено запит до DbSet Employees, за допомогою методу RemoveRange() ініційовано видалення вказаних співробітників з бази даних та методом SaveChangesAsync() зберігаються зміни у БД (лістинг 7).

Лістинг *7 Реалізація видалення працівників у БД*

```
using (var context = await
_contextFactory.CreateDbContextAsync(cancellationToken)
)
{
    context.Employees.RemoveRange(context.Employees);
    await context.SaveChangesAsync(cancellationToken);
    await context.DisposeAsync();
}
```

3.3.2.2 Розробка запитів для фреймворку NHibernate

В ході розробки запитів до бази даних за допомогою фреймворку NHibernate було зосереджено увагу на виконанні базових CRUD-операцій, а також роботі з відношеннями між сутностями. NHibernate використовує API сесії для управління станом об'єктів, що надає гнучкість та ефективність при роботі з базою даних.

Для створення нового об'єкта, на прикладі типу Employee, використовується метод Save() або SaveAsync() із об'єкта сесії. За збереження даних у базі відповідає метод сесії Flush() або FlushAsync().

Лістинг *8 Реалізація додавання нових працівників у БД*

```
using var session = _sessionService.OpenSession();
foreach (var employee in employees)
{
    await session.SaveAsync(employee,
cancellationToken);
    }
```

```
    await session.FlushAsync(cancellationToken);
await
_sessionService.CloseSessionAsync(cancellationToken);
```

Для читання записів з однієї таблиці, на прикладі Employees, було використано метод сесії Query() для відповідної сутності та асинхронний метод ToListAsync(), для перетворення отриманої послідовності у список (лістинг 9).

Лістинг *9 Реалізація запиту працівників з БД (запит з однієї таблиці)*

```
using var session = _sessionService.OpenSession();
IEnumerable<EmployeeModel> employees = await
session.Query<Employee>()
    .Select(_ => new EmployeeModel()
    {
        EmployeeId = _.Id,
        BirthDate = _.BirthDate,
        Email = _.Email,
        FirstName = _.FirstName,
        LastName = _.LastName
    }).ToListAsync(cancellationToken);
await
_sessionService.CloseSessionAsync(cancellationToken);
return employees;
```

Для читання записів з таблиць, які мають відношення один до одної, Employees та EmployeeDetails, було використано метод сесії Query() для сутності Employee та метод Fetch() для підзапиту до сутності EmployeeDetail, асинхронний метод ToListAsync() використовується для перетворення отриманої послідовності у список (лістинг 10).

Лістинг *10 Реалізація запиту працівників та додаткової їх інформації з БД (запит до таблиць з відношенням "один до одного")*

```
IEnumerable<EmployeeWithDetails> employeesWithDetails;
using var session = _sessionService.OpenSession();
employeesWithDetails = await session.Query<Employee>()
    .Fetch(e => e.EmployeeDetail)
    .Select(e => new EmployeeWithDetails()
```

```
    {
        Id = e.Id,
        FirstName = e.FirstName,
        LastName = e.LastName,
        Email = e.Email,
        BirthDate = e.BirthDate,
        SpecialNeeds = e.EmployeeDetail == null ? "" :
e.EmployeeDetail.SpecialNeeds
    })
    .ToListAsync(cancellationToken);
await
_sessionService.CloseSessionAsync(cancellationToken);
return employeesWithDetails;
```

Для читання записів з таблиць, які мають відношення один до багатьох, Departments та Employees, було використано метод сесії Query() для сутності Department та метод Fetch() для підзапиту до сутностей Employees, асинхронний метод ToListAsync() використовується для перетворення отриманої послідовності у список (лістинг 11).

Лістинг *11 Реалізація запиту відділів за працівниками з БД (запит до таблиць з відношенням "один до багатьох")*

```
IEnumerable<DepartmentWithEmployeesModel>
departmentsWithEmployees;
using var session = _sessionService.OpenSession();
departmentsWithEmployees = await
session.Query<Department>()
    .Fetch(e => e.Employees)
    .Select(d => new DepartmentWithEmployeesModel()
    {
        DepartmentId = d.Id,
        DepartmentName = d.Name,
        Employees = d.Employees.Select(e => new
EmployeeModel()
        {
            EmployeeId = e.Id,
            FirstName = e.FirstName,
            LastName = e.LastName,
            Email = e.Email,
            BirthDate = e.BirthDate
```

```
        })
    }).ToListAsync(cancellationToken);

await
_sessionService.CloseSessionAsync(cancellationToken);
return departmentsWithEmployees;
```

Для читання записів з таблиць, які мають відношення багато до багатьох, Employees та Projects, було використано метод сесії Query() для сутності Employee, метод Fetch() для підзапиту до сутностей EmployeeProjects та метод ThenFetch() для підзапиту до сутності Project, асинхронний метод ToListAsync() використовується для перетворення отриманої послідовності у список (лістинг 12).

Лістинг *12 Реалізація запиту працівників та їх проектів з БД (запит до таблиць з відношенням "багато до багатьох")*

```
IEnumerable<EmployeeWithProjectsModel>
employeesWithProjects;
using var session = _sessionService.OpenSession();
employeesWithProjects = await session.Query<Employee>()
    .FetchMany(e => e.EmployeeProjects)
    .ThenFetch(ep => ep.Project)
    .Select(e => new EmployeeWithProjectsModel
    {
        EmployeeId = e.Id,
        FirstName = e.FirstName,
        LastName = e.LastName,
        Projects = e.EmployeeProjects.Select(ep => new
ProjectModel
        {
            ProjectId = ep.Project.Id,
            ProjectName = ep.Project.Name
        })
    }).ToListAsync(cancellationToken);

await
_sessionService.CloseSessionAsync(cancellationToken);
return employeesWithProjects;
```

Для оновлення записів у таблицях, на прикладі Departments, викликано метод сесії Query() для сутності Departments, далі викликаються методи UpdateBuilder та UpdateAsync, які дозволяють оновити дані у базі, без необхідності їх попереднього отримання, за допомогою метода сесії FlushAsync() зберігаються зміни у БД (лістинг 13).

Лістинг *13 Реалізація оновлення назв відділів у БД*

```
Dictionary<string, int> response = new();
using var session = _sessionService.OpenSession();
var departmentsCount = await
session.Query<Department>()
    .UpdateBuilder().Set(_ => _.Name, _ => prefix + ":
" + _.Name).UpdateAsync();
response.Add("Amount of updated departments: ",
departmentsCount);
await session.FlushAsync(cancellationToken);
await
_sessionService.CloseSessionAsync(cancellationToken);
return response;
```

Для видалення об'єктів у таблицях, на прикладі Employees, було викликано два методи сесії, а саме Query() та DeleteAsync(), а за допомогою метода сесії FlushAsync() зберігаються зміни у БД (лістинг 14).

Лістинг *14 Реалізація видалення працівників у БД*

```
using var session = _sessionService.OpenSession();
await
session.Query<Employee>().DeleteAsync(cancellationToken
);
await session.FlushAsync(cancellationToken);
await
_sessionService.CloseSessionAsync(cancellationToken);
```

3.3.2.3 Розробка запитів для фреймворку Dapper

Для розробки запитів до бази даних з використанням мікро ORM фреймворку Dapper було реалізовано набір типових CRUD-операцій і враховано різні типи відношень між сутностями.

Для створення нового об'єкта типу Employee використовується метод QueryAsync(), який приймає SQL-запит і об'єкт як параметри та повертає унікальний ідентифікатор створеного об’єкту (лістинг 15).

Лістинг *15 Реалізація додавання нових працівників у БД*

```
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    await Parallel.ForEachAsync(employees, async
(employee, cancellationToken) =>
    {
        (await connection.QueryAsync<int>(
            "INSERT INTO Employees (FirstName,
LastName, Email, DepartmentId, BirthDate) OUTPUT
INSERTED.Id VALUES (@FirstName, @LastName, @Email,
@DepartmentId, @BirthDate)",
            new { employee.FirstName,
employee.LastName, employee.Email,
employee.DepartmentId, employee.BirthDate })).Single();
    });

    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

Для читання записів з однієї таблиці, на прикладі Employees, використано метод QueryAsync(), який приймає SQL-запит типу SELECT до відповідної таблиці та повертає результат вибірки (лістинг 16).

Лістинг *16 Реалізація запиту працівників з БД (запит з однієї таблиці)*

```
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    var query = @"SELECT Id as EmployeeId, FirstName,
LastName, Email, BirthDate FROM Employees";
```

```
    employees = await
connection.QueryAsync<EmployeeModel>(query);
    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

Для читання записів з таблиць, які мають відношення один до одної, Employees та EmployeeDetails, використано метод QueryAsync(), який приймає SQL-запит типу SELECT до відповідних таблиць, об'єднаних за допомогою LEFT JOIN, та повертає результат вибірки (лістинг 17).

Лістинг *17 Реалізація запиту працівників та додаткової їх інформації з БД (запит до таблиць з відношенням "один до одного")*

```
IEnumerable<EmployeeWithDetails> employeesWithDetails =
new List<EmployeeWithDetails>();
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    var query = @"
        SELECT e.*, ED.SpecialNeeds
        FROM Employees e
        LEFT JOIN EmployeeDetails ed ON e.Id =
ed.EmployeeId";
    employeesWithDetails = await
connection.QueryAsync<EmployeeWithDetails>(query);
    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

Для читання записів з таблиць, які мають відношення один до багатьох, Departments та Employees, використано метод QueryAsync(), який приймає SQL-запит типу SELECT до відповідних таблиць, об'єднаних за допомогою LEFT JOIN, та повертає результат вибірки (лістинг 18).

Лістинг *18 Реалізація запиту відділів за працівниками з БД (запит до таблиць з відношенням "один до багатьох")*

```
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    var query = @"
        SELECT d.Id AS DepartmentId, d.Name AS
DepartmentName, e.Id AS EmployeeId, e.FirstName,
e.LastName, e.Email, e.BirthDate
        FROM Departments d
        LEFT JOIN Employees e ON d.Id =
e.DepartmentId";

    await
connection.QueryAsync<DepartmentWithEmployeesModel,
EmployeeModel, DepartmentWithEmployeesModel>(
        query,
        (department, employee) =>
        {
            if
(!departments.TryGetValue(department.DepartmentId, out
var dep))
            {
                dep = department;
                dep.Employees = new
List<EmployeeModel>();

    departments.Add(department.DepartmentId, dep);
            }

            dep.Employees =
dep.Employees.Append(employee);
            return dep;
        },
        splitOn: "EmployeeId"
    );

    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

Для читання записів з таблиць, які мають відношення багато до багатьох, Projects та Employee, використано метод QueryAsync(), який приймає SQL-запит типу SELECT до відповідних таблиць, об’єднаних за допомогою LEFT JOIN, та повертає результат вибірки (лістинг 19).

Лістинг *19 Реалізація запиту працівників та їх проектів з БД (запит до таблиць з відношенням "багато до багатьох")*

```
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    var query = @"
        SELECT e.Id AS EmployeeId, e.FirstName,
e.LastName, p.Id AS ProjectId, p.Name AS ProjectName
        FROM Employees e
        LEFT JOIN EmployeeProjects ep ON e.Id =
ep.EmployeeId
        LEFT JOIN Projects p ON ep.ProjectId = p.Id";
    await
connection.QueryAsync<EmployeeWithProjectsModel,
ProjectModel, EmployeeWithProjectsModel>(
        query,
        (employee, project) =>
        {
            if
(!employeeDictionary.TryGetValue(employee.EmployeeId,
out var emp))
            {
                emp = employee;
                emp.Projects = new
List<ProjectModel>();

    employeeDictionary.Add(employee.EmployeeId, emp);
            }

            emp.Projects =
emp.Projects.Append(project);
            return emp;
        },
        splitOn: "ProjectId"
    );
```

```
    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

Для оновлення записів у таблицях, на прикладі Departments, використано метод ExecuteAsync(), який приймає SQL-запит типу UPDATE до відповідної таблиці, та повертає кількість оновлених рядків (лістинг 20).

Лістинг *20 Реалізація оновлення назв відділів у БД*

```
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    string query = $"UPDATE Departments SET Name =
'{prefix}: ' + Name";
    int count = await connection.ExecuteAsync(query,
cancellationToken);
    response.Add("Amount of updated departments: ",
count);
    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

Для видалення об'єктів у таблицях, на прикладі Employees, використано метод ExecuteAsync(), який приймає SQL-запит типу DELETE до відповідної таблиці (лістинг 21).

Лістинг *21 Реалізація видалення працівників у БД*

```
using (var connection = new
SqlConnection(_connectionString))
{
    await connection.OpenAsync(cancellationToken);
    string query = "DELETE FROM Employees";
    await connection.ExecuteAsync(query);
    await connection.CloseAsync();
    await connection.DisposeAsync();
}
```

3.4 Архітектура програмного забезпечення для порівняльного аналізу

Для ефективного порівняльного аналізу фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET важливо використовувати ретельно сплановану архітектуру програмного забезпечення. Це дасть можливість чітко визначити різницю в продуктивності, можливостях та використанні кожного з розглянутих фреймворків.

Проведення порівняльного аналізу вимагає, щоб кожен модуль використовував однакові методи доступу до даних. Це можна здійснити за допомогою створення єдиного репозиторію для кожного фреймворку, який повинен реалізувати однаковий набір операцій (створення, читання, оновлення, видалення - CRUD).

Для розробки проекту було створено одне рішення, FrameworksTesterApp, у середовищі Microsoft Visual Studio 2022 (рис. 5).

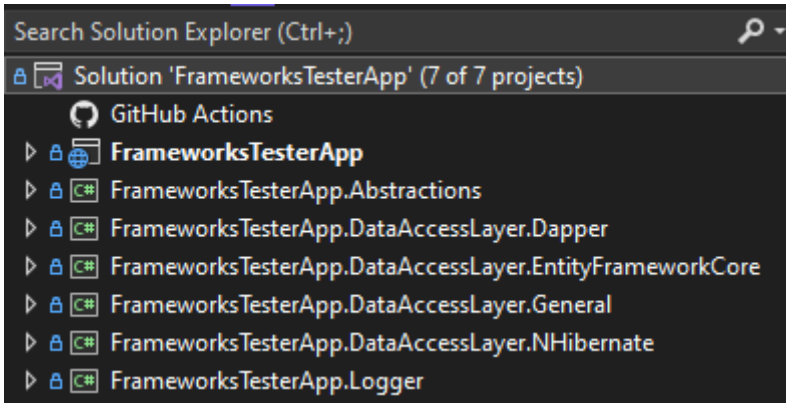

Рис. 5 *Список проектів рішення FrameworksTesterApp*

У цьому рішенні якому реалізовано 7 проектів:

– FrameworksTesterApp – головний проект в рішенні, який успадковує назву рішення. Має посилання на усі інші проекти рішення. До складку проекту входять контроллер TesterController, сервіс OrmService, валідаційний сервіс ValidationService та файл Program.cs.

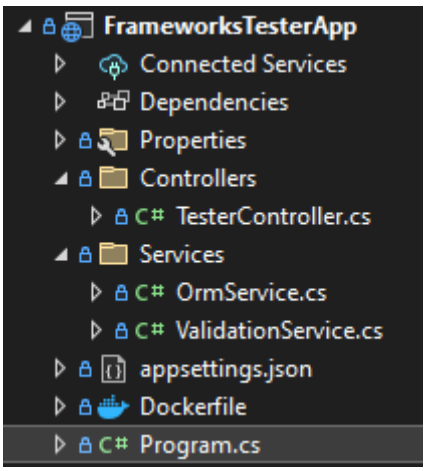

Рис. 6 *Структура проекту FrameworksTesterApp*

Контроллер TesterController містить у собі набір кінцевих точок (відкритий API), які здатні приймати запити для дослідження функціоналу ORM. Перелік наявних кінцевих точок та їх опис наведено на рисунку 7. Візуалізація API зроблена завдяки бібліотеці Swashbuckle.AspNetCore.Swagger [30].

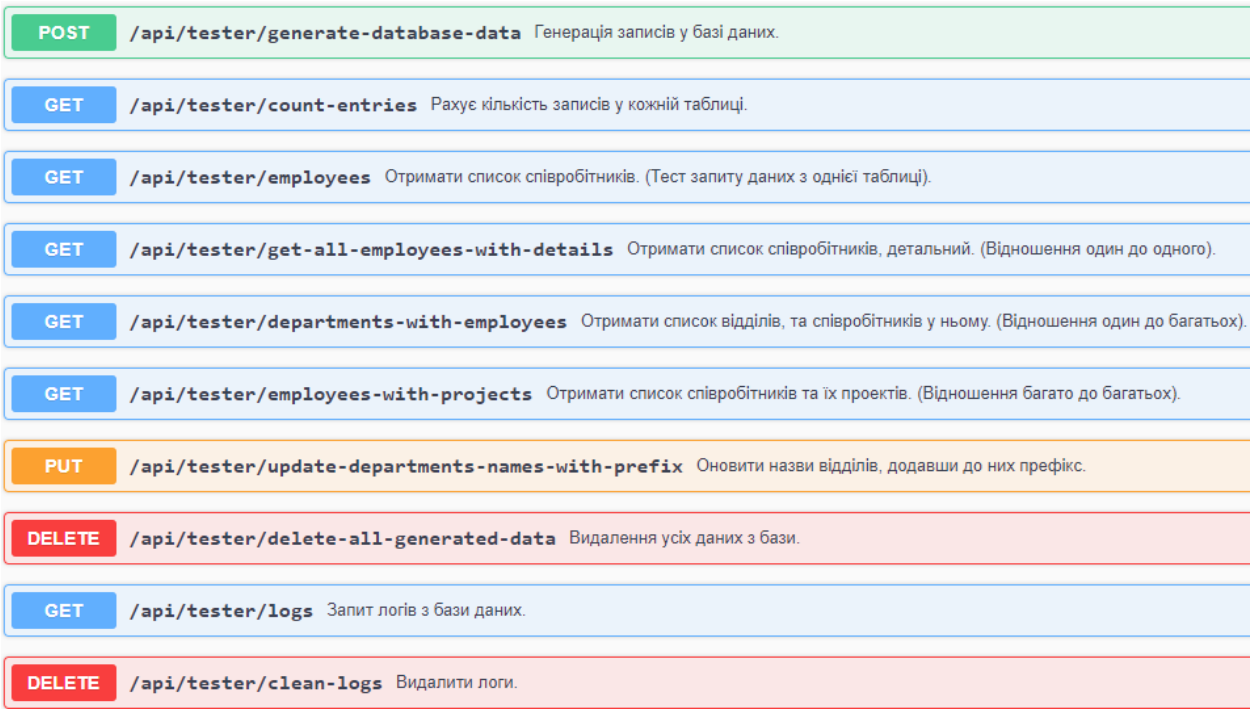

Рис. 7 *API проекту FrameworksTesterApp*

Сервіс OrmService це головний сервіс логіки проекту, який отримує від API інформацію про те, який запит було зроблено та який саме ORM фреймворк

треба задіяти. За допомогою Dependency Injection реалізується потрібний клас провайдер, вказаного у хедері запиту ORM фреймворку, та оброблює запит. Після оброблення запиту – у базу записуються метрики, а після вертається результат запиту та зібрані метрики до користувача. ValidationService виконує роль валідатора для запиту на генерацію даних. Файл Program.cs містить у собі реєстрацію усіх сервісів, які задіяні у рішенні;

– FrameworksTesterApp.Abstractions – проект містить у собі моделі та переліки (enumerations) які використовуються для запитів та відповідей на API;

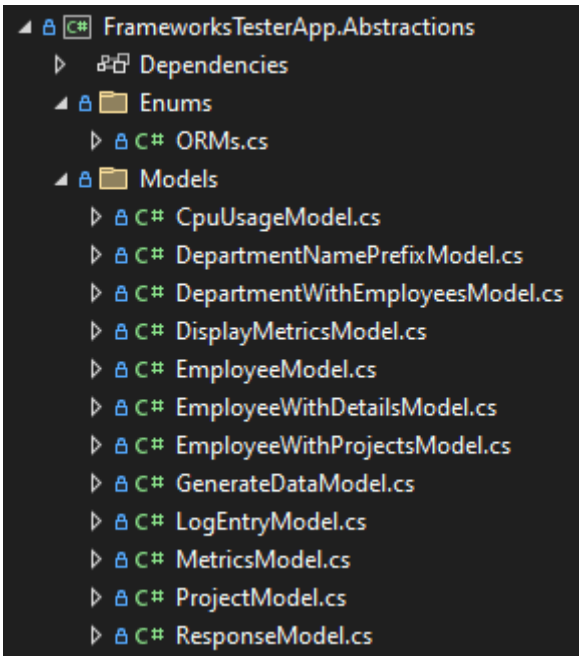

Рис. *8 Структура проекту FrameworksTesterApp.Abstractions*

– FrameworksTesterApp.DataAccessLayer.General – проект містить сутності, які описують таблиці в базі даних і використовуються усіма трьома ORM фреймворками. Також у проекті є генератор даних, який заповнює сутності значеннями, як наприклад назви відділів, ім’я та прізвище співробітників, назви проектів, тощо, для подальшого запису їх у базу даних. Генератор даних використовує бібліотеку Bogus [31], яка надає можливість швидко створити фейкові дані для сутностей;

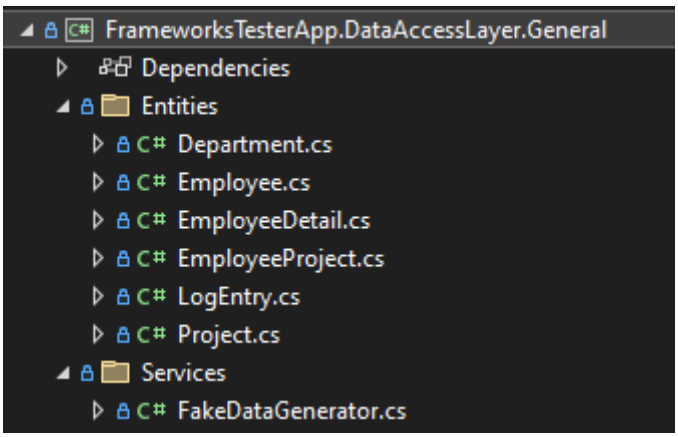

Рис. *9 Структура проекту FrameworksTesterApp.DataAccessLayer.General*

– FrameworksTesterApp.DataAccessLayer.Dapper – проект містить класи провайдери які мають можливість доступу до даних відповідних таблиць в базі даних через функціонал Dapper ORM. Також проект містить сервіси які генерують дані в базі, видаляють дані з усіх таблиць та рахують кількість записів у кожній таблиці;

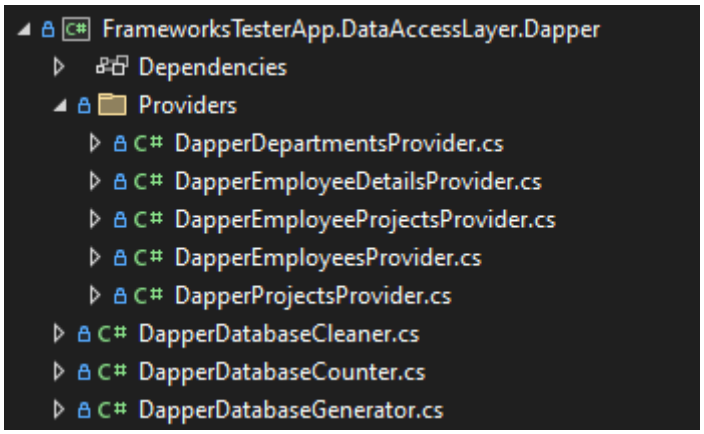

Рис. *10 Структура проекту FrameworksTesterApp.DataAccessLayer.Dapper*

– FrameworksTesterApp.DataAccessLayer.EntityFrameworkCore – проект містить такі ж класи провайдери які мають можливість доступу до даних відповідних таблиць в базі даних через, але через функціонал Entity Framework Core. Також проект містить сервіси які генерують дані в базі, видаляють дані з усіх таблиць, рахують кількість записів у кожній таблиці та модель ApplicationDbContext яка містить конфігурацію та мапінг сутностей

пов’язаних з базою даних. Кожен провайдер та інший сервіс, цього проекту, має свій екземпляр ApplicationDbContext для кожного запиту, що створюється фабрикою IDbContextFactory;

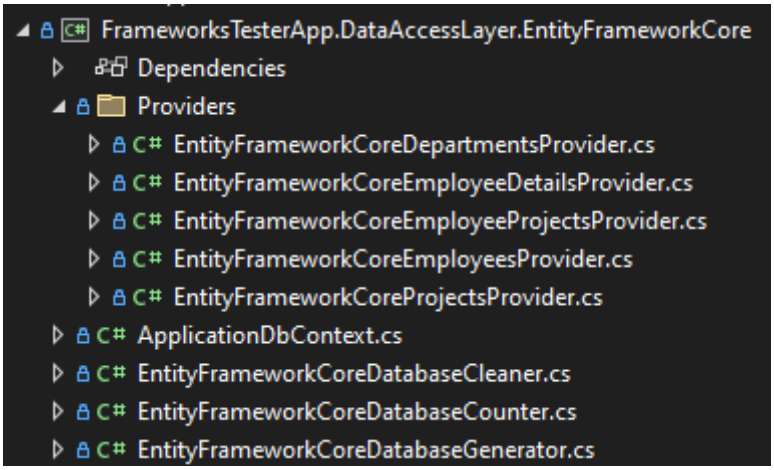

Рис. *11 Структура проекту FrameworksTesterApp.DataAccessLayer. EntityFrameworkCore*

– FrameworksTesterApp.DataAccessLayer.NHibernate – проект містить третю реалізацію класів провайдерів які мають можливість доступу до даних відповідних таблиць в базі даних через функціонал NHibernate. Також проект містить сервіси які генерують дані в базі, видаляють дані з усіх таблиць та рахують кількість записів у кожній таблиці. В окремих файлах для кожної сутності зазначено її мапінг до відповідної таблиці в базі даних. У сервісі NHibernateSessionService створено функціонал взаємодії з сесіями, що дозволяє встановлювати підключення до бази даних та виконувати CRUD операції. Цей сервіс реалізується у кожному провайдері та іншому сервісі цього проекту. Кожен запит має свою сесію, що створюється завдяки фабриці ISessionFactory;

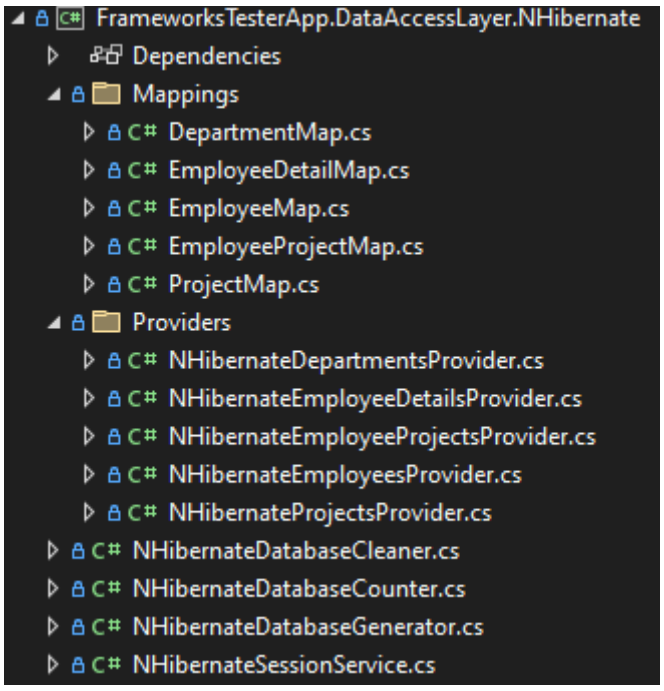

Рис. *12 Структура проекту FrameworksTesterApp.DataAccessLayer.NHibernate*

– FrameworksTesterApp.Logger – проект відповідає за запис метрик до таблиці LogEntries. Кожен CRUD запит, який оброблюється будь якою ORM, додається до цієї таблиці, разом з метриками та відповіддю, якщо така має місце бути.

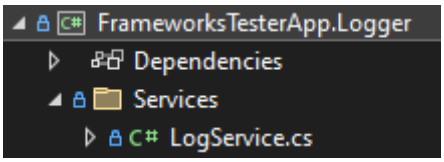

Рис. *13 Структура проекту FrameworksTesterApp.Logger*

Висновки з розділу 3

В даному розділі основний акцент ставився на вибір технологій та проектування архітектури ПЗ для порівняльного аналізу ORM-фреймворків.

В процесі аналізу були порівняні та обрані основні технології для розробки. Серед розглянутих мов програмування Python, Java та C#, було обрано C# через його відмінну інтеграцію з фреймворками .NET та SQL Server. Серед

середовищ розробки були порівняні Microsoft Visual Studio, Visual Studio Code та JetBrains Rider, проте було вибрано Microsoft Visual Studio через її потужність, інтуїтивність та широкий спектр функціональності. З урахуванням швидкості та безпеки, було обрано СУБД MS SQL Server.

Для забезпечення об'єктивного порівняльного аналізу були розроблені функціональні вимоги, вимоги до проектування бази даних та вимоги до ORM фреймворків. Такий підхід дозволив створити єдині умови для дослідження різних ORM-фреймворків.

Під час проектування бази даних була побудована ERD діаграма, яка допомогла визначити структуру даних і відношення між ними. Основні запити для ORM фреймворків були розроблені з урахуванням цієї структури.

Архітектура ПЗ була розроблена таким чином, щоб забезпечити уніфікований доступ до даних для різних ORM-фреймворків. Використання класів провайдерів дозволило забезпечити однаковий набір операцій (CRUD) для кожного фреймворку, а впровадження залежностей сприяло в забезпеченні гнучкості використання різних ORM-фреймворків.

Отже, розроблена архітектура та обрані технології створюють основу для наступного етапу дослідження, що включає в себе виконання порівняльного аналізу і виведення результатів.

РОЗДІЛ 4 РОЗРОБКА ТА ВИКОРИСТАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Розробка алгоритму роботи застосунку

Розробка алгоритму роботи застосунку є ключовим етапом дослідження, який включає в себе створення алгоритмічних рішень для основних функцій, які були визначені в попередньому розділі, а саме: створення, читання, оновлення та видалення даних (CRUD), збір метрик, запис результатів обробки запиту до відповідної таблиці.

Блок-схема, представлена на рис. 14, візуально демонструє алгоритм роботи застосунка. Алгоритм складається з наступних кроків:

1. Відкрите API застосунка отримує запит на здійснення одної з операцій, пов’язаної з функціоналом ORM фреймворків. У запиті є заголовок (англ. header) з назвою "orm", який зберігає назву ORM фреймворка, для якого виконується запит.
2. Перевіряється заголовок запиту. У заголовку має бути вказано тип ORM фреймворку (Dapper, NHibernate чи Entity Framework Core) для якого треба виконати запит. Якщо тип не вказано, або вказано невідомий тип – генерується виключення типу NotImplementedException.
3. В залежності від типу запиту проходить його валідація. Якщо запит не валідний, то генерується виключення.
4. Для вказаного у заголовку запита ORM фреймворку, за допомогою впровадження залежності, реалізується сервіс, необхідний для обробки запиту, що надійшов.
5. Створюється змінна типу завдання (англ. Task). Цій змінній присвоюється завдання на виконання методу реалізованого сервісу у попередньому кроці алгоритму. Усі методи для обробки запитів є асинхронними.
6. За допомогою бібліотеки Universe.CpuUsage [32] фіксується процесорний час, який вже витрачений на поточний процес.

7. За допомогою службового слова await та створеної, у п’ятому кроці алгоритму, змінної запускається виконання завдання по обробці запиту. Застосунок очікує на завершення завдання.
8. Після виконання завдання, ще раз, за допомогою бібліотеки Universe.CpuUsage, фіксується процесорний час, який вже витрачений на поточний процес.
9. За допомогою вбудованої бібліотеки System.Diagnostics фіксується зарезервована оперативна пам'ять під виконання поточного процесу.
10. За допомогою вбудованої бібліотеки System.Threading фіксується кількість активних потоків, задіяних у застосунку.
11. Вираховується процесорний час витрачений на виконання завдання, як різниця між значенням до початку завдання, та одразу після. До уваги береться процесорний час користувача.
12. Записується модель запиту, назва ORM фреймворка який опрацьовував запит, назва виконавчого метода, значення зарезервованої оперативної пам’яті, метрики процесорного часу, кількість активних потоків та модель відповіді до таблиці LogEntries.
13. Примусово викликається збирач сміття (англ. garbage collector) для звільнення задіяних ресурсів для обробки запиту.
14. API повертає модель відповіді користувачу, який зробив запит.

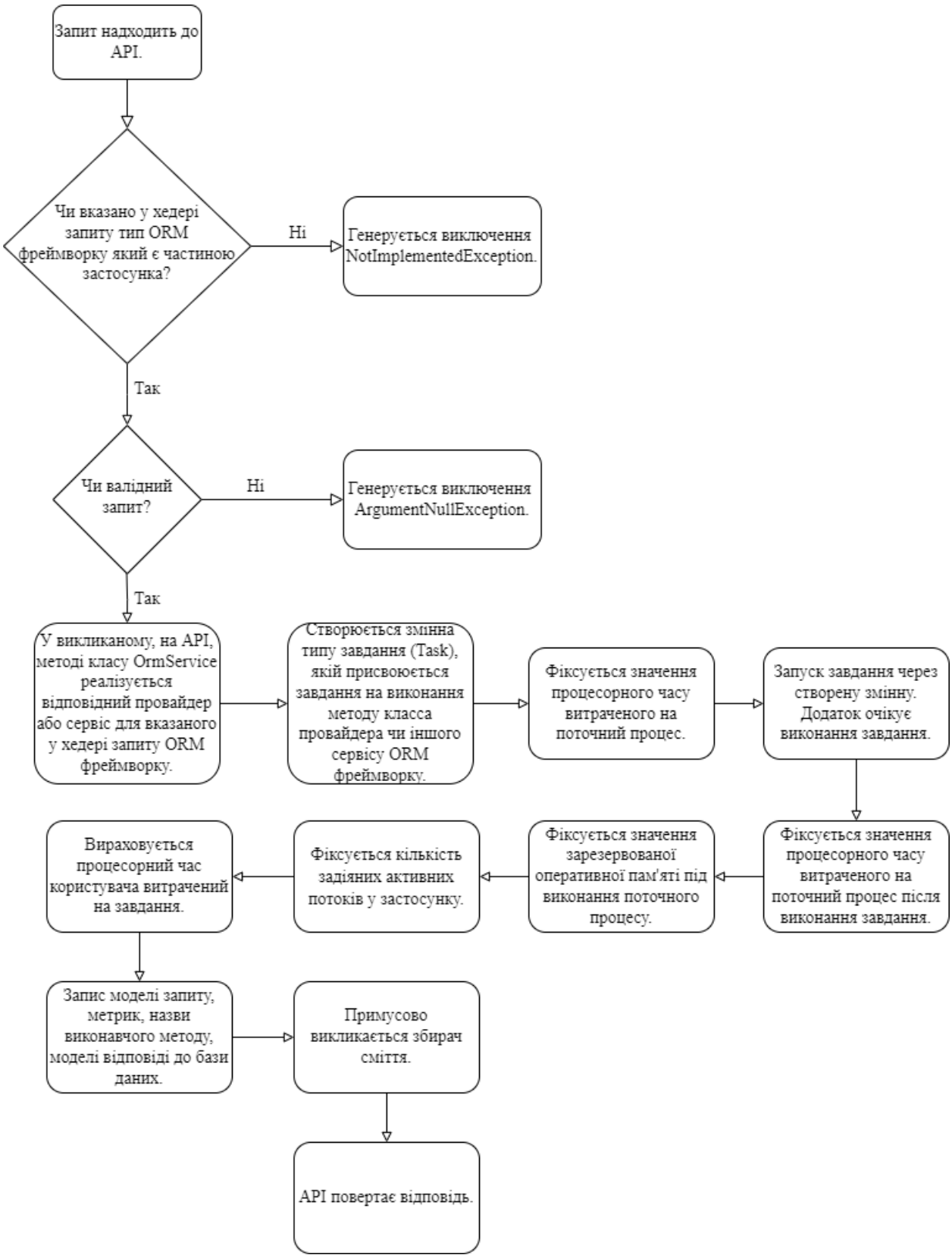

Рис. *13 Діаграма алгоритму роботи застосунку*

4.2 Апробація роботи застосунку

Апробація роботи застосунку забезпечує підтвердження правильності функціонування програмного забезпечення та його відповідності встановленим вимогам. Апробація розробленого застосунку була проведена на різних етапах його розробки.

У першу чергу було проведено одиничне випробування окремих компонентів системи з метою виявлення та усунення помилок на ранніх стадіях розробки. Це допомогло уникнути значної кількості помилок у подальшому та забезпечити стабільність роботи всіх компонентів системи.

При проведенні порівняльного аналізу ORM фреймворків для платформи Microsoft .NET важливо визначити об'єктивні критерії оцінки. Вони допомагають структурувати аналіз та забезпечують можливість вимірювання показників, що можуть бути порівняні між різними фреймворками. В цьому розділі представлені основні критерії, що були використані при порівнянні фреймворків Dapper, Entity Framework Core та NHibernate.

Для оцінки продуктивності кожного ORM фреймворку було розроблено дослідницькі сценарії, що включають виконання основних CRUD-операцій (створення, читання, оновлення та видалення даних). Ці сценарії було виконано для кожного з вибраних фреймворків, і були зібрані наступні метрики продуктивності:

– час виконання операцій на процесорі. Це задіяний процесорний час, потрібний для виконання кожної CRUD операції. Чим менше часу потрібно для виконання операції, тим вищою є продуктивність фреймворку. До уваги береться процесорний час користувача;

– резервування оперативної пам'яті. Обсяг оперативної пам'яті, що резервується під час виконання CRUD операцій. Ефективне використання пам'яті може покращити продуктивність застосунку і зменшити його вплив на системні ресурси;

– кількість активних потоків. Кількість задіяних активних потоків додатком під час виконання CRUD операцій. Ця метрика допоможе зрозуміти,

чи є, при роботі з ORM фреймворком, ситуація, коли програмний код не вірно керується пам'яттю, виділеною для потоків, що може призвести до витіку пам'яті (англ. “Memory leak”). Пам'ять витікає, коли об'єкти створюються в процесі виконання програми, але не звільняються з пам'яті після того, як вони вже не потрібні.

Враховуючи ці критерії, можна отримати об'єктивне уявлення про продуктивність кожного з вибраних ORM фреймворків. Це дає можливість зробити виважене рішення при виборі найбільш підходящого фреймворку для конкретних потреб та умов.

4.2.1 Метрики продуктивності виконання запитів до бази даних.

Результати проведених вимірювань булу записані у таблиці та представлені у вигляді графічної інтерпретації, що дає змогу проаналізувати їх в наступному розділі цієї роботи. Отримані дані дозволяють порівняти ORM фреймворки з точки зору їх продуктивності та ефективності використання системних ресурсів. Метрики були зібрані на робочій станції, яка містить наступні характеристики: Windows 11 Pro 64 bit, об’єм оперативної пам’яті 32 гігабайти, серія процесора 11th Gen Intel(R) Core(TM) i7-11800H. Актуальні версії фреймворків, для шостої версії .NET, які були задіяні у досліджені: Dapper версія 2.1.24, Microsoft.EntityFrameworkCore версія 7.0.11, NHibernate версія 5.4.6.

4.2.1.1 Метрики при виконанні запису даних до бази даних.

Збір метрик при записі до бази даних зроблено на прикладі запиту, у якому надходить кількість працівників, для цих працівників треба згенерувати дані та записати у базу даних. Для уніфікації запиту – усі працівники мають відношення до одного відділу та працюють на над одним проектом. Нижче наведені таблиці та відповідні графіки, які показують залежність процесорного часу користувача, витраченого на функціонал запису ORM фреймворку до бази даних, зарезервованої оперативної пам’яті та кількості активних потоків від кількості працівників, яких треба занести до бази даних.

Таблиця *4*

Залежність процесорного часу користувача, витраченого на запис даних, від кількості працівників

	Час обробки ЦП (мікросекунди)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	58750	82500	80000
2000	101875	161250	192187
4000	217500	325000	240000
8000	435546	713750	359375
16000	804375	1486875	1098437
32000	1563750	3431875	2089062
64000	3088125	6937500	4128125
128000	5795000	16200000	9435937

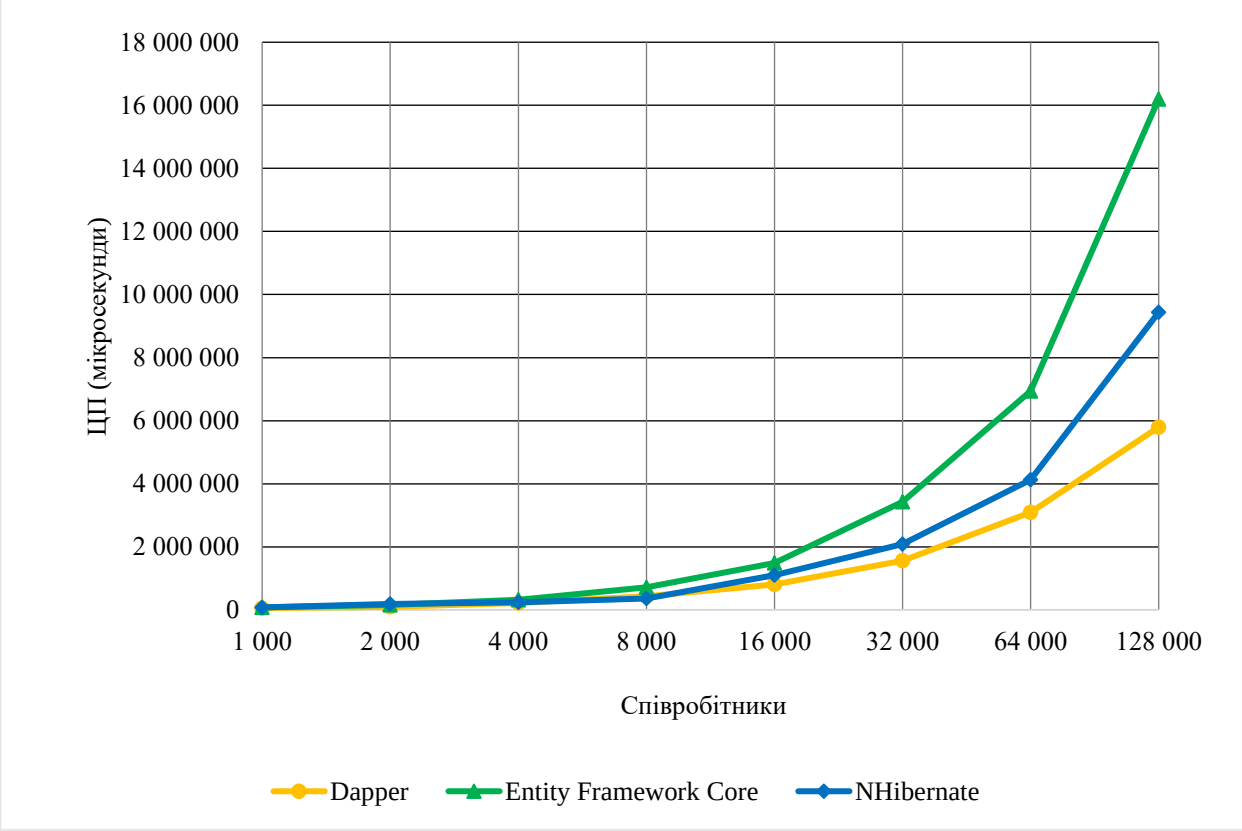

Рис *14. Графік залежності процесорного часу користувача, витраченого на запис даних, від кількості працівників*

Таблиця 5

Залежність зарезервованої оперативної пам'яті при записі даних, від кількості працівників

	ОЗП (МБ)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	156	225	200
2000	207	289	259
4000	275	340	321
8000	314	585	352
16000	329	951	359
32000	349	2052	411
64000	377	2183	704
128000	413	2840	1077

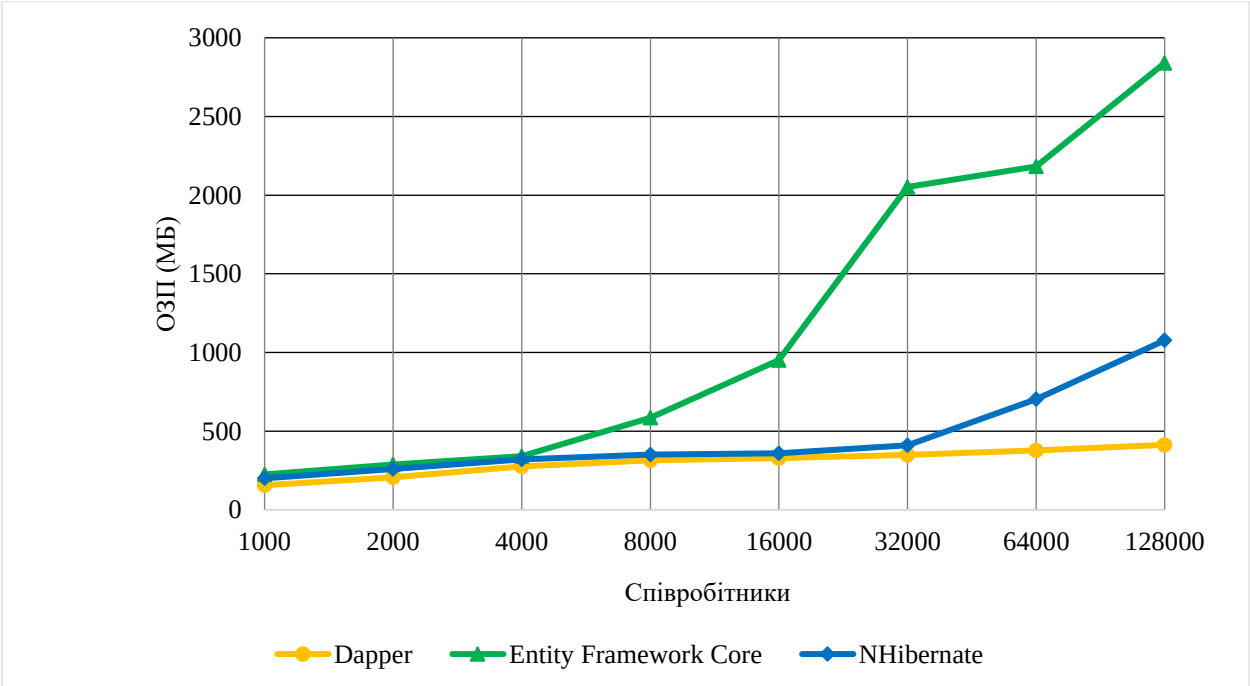

Рис 15. *Графік залежності зарезервованої оперативної пам'яті при записі даних, від кількості працівників*

Таблиця *6*

Залежність кількості задіяних активних потоків застосунком від кількості працівників під час запису даних

	Потоки (кількість)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	25	13	10
2000	25	11	12
4000	24	11	10
8000	24	10	11
16000	23	10	10
32000	20	10	10
64000	19	10	10
128000	18	9	10

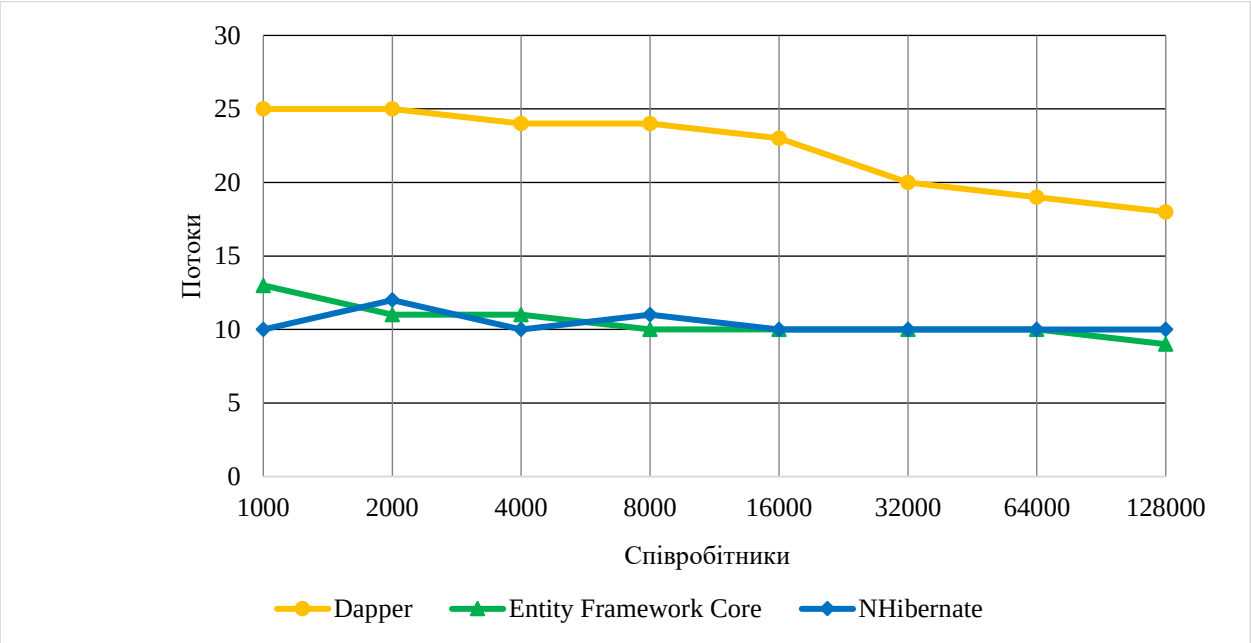

Рис *16. Графік залежності кількості задіяних активних потоків застосунком від кількості працівників під час запису даних*

З отриманих метрик можна констатувати, що, при записі даних, найшвидшим виявився Dapper, а найповільнішим Entity Framework Core. Dapper задіяв найменший об’єм оперативної пам’яті, а Entity Framework Core задіяв найбільший. NHibernate зайняв проміжне місце між показниками Dapper та Entity Framework Core. Усі три фреймворки показали зменшення кількості задіяних активних потоків, при збільшенні кількості даних для запису. Особливо це помітно у сценаріях з Dapper. Зменшення кількості активних потоків при збільшенні кількості записів та збільшенні часу, необхідного на запис даних, свідчить про те, що система оптимізує керування потоками шляхом збільшення навантаження на потік, замість збільшення кількості задіяних потоків, тим самим звільняє потоки для інших задач.

4.2.1.2 Метрики при виконанні запиту на отримання даних з однієї таблиці.

Збір метрик при отриманні даних зі однієї таблиці зроблено на прикладі запиту, у якому запрошується певна кількість записів з таблиці Employees. Нижче наведені таблиці та відповідні графіки, які показують залежність процесорного часу користувача, витраченого на функціонал читання ORM фреймворку з бази даних, зарезервованої оперативної пам’яті та кількості активних потоків від кількості записів співробітників, яких треба отримати з бази даних.

Таблиця *7*

Залежність процесорного часу користувача, витраченого на отримання даних з однієї таблиці, від кількості працівників

	Час обробки ЦП (мікросекунди)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	15625	62500	15625
2000	15625	93750	15625
4000	15625	31250	46875

8000	46875	15625	46875
16000	78125	31250	46875
32000	62500	31250	62500
64000	93750	31250	125000
128000	78125	78125	296875
256000	312500	328125	703125
512000	375000	359375	1343750
1024000	421875	437500	2671875

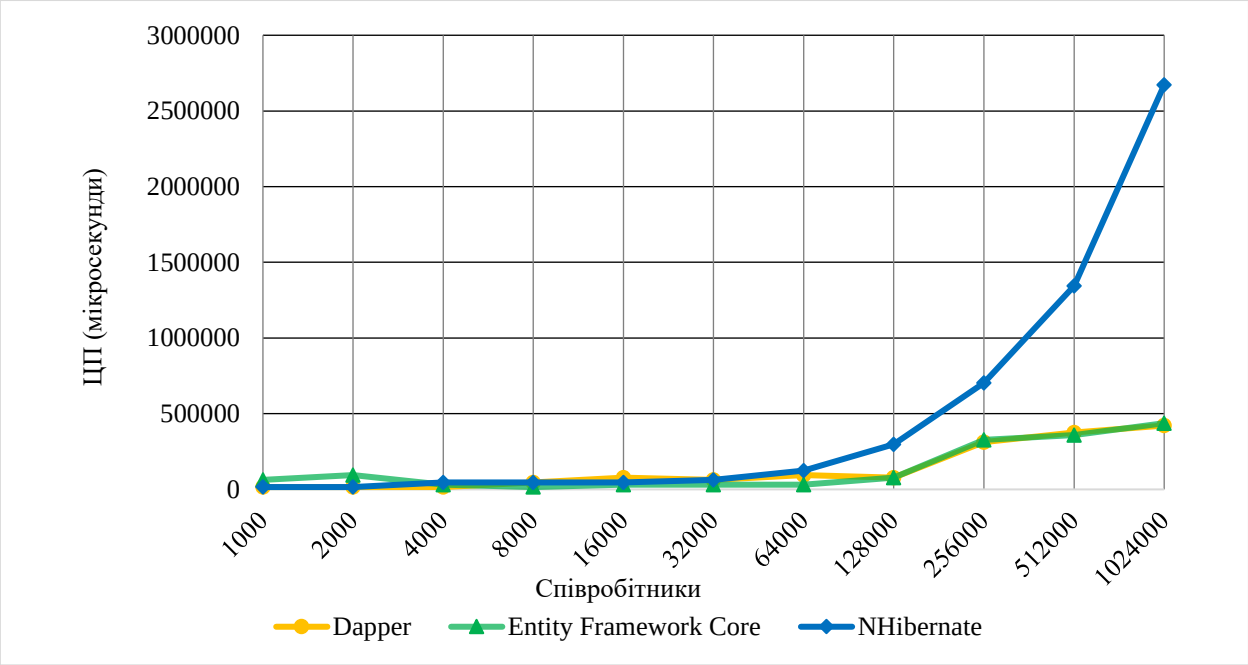

Рис *17. Графік залежності процесорного часу користувача, витраченого на отримання даних з однієї таблиці, від кількості працівників*

Таблиця *8*

Залежність зарезервованої оперативної пам'яті на отримання даних з однієї таблиці, від кількості працівників

	ОЗП (МБ)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	65	92	111

2000	65	88	91
4000	64	90	98
8000	66	93	111
16000	68	99	137
32000	75	111	190
64000	88	136	243
128000	114	186	355
256000	167	276	610
512000	225	369	1068
1024000	414	755	1782

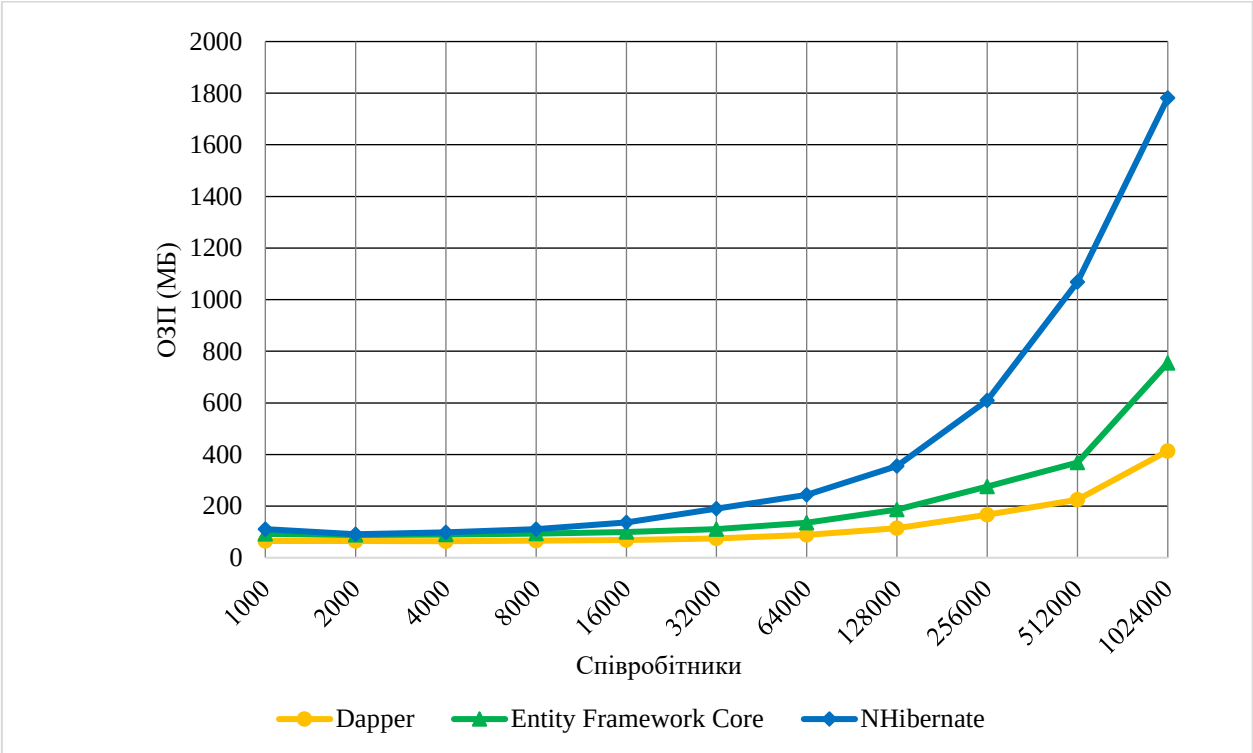

Рис *18. Графік залежності зарезервованої оперативної пам'яті на отримання даних з однієї таблиці, від кількості працівників*

Таблиця *9*

Залежність кількості задіяних активних потоків застосунком від кількості працівників під час запиту даних з однієї таблиці

	Потоки (кількість)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	7	8	8
2000	7	8	8
4000	7	8	8
8000	8	9	9
16000	7	9	9
32000	7	8	9
64000	7	7	8
128000	6	8	9
256000	8	12	8
512000	7	9	8
1024000	7	10	12

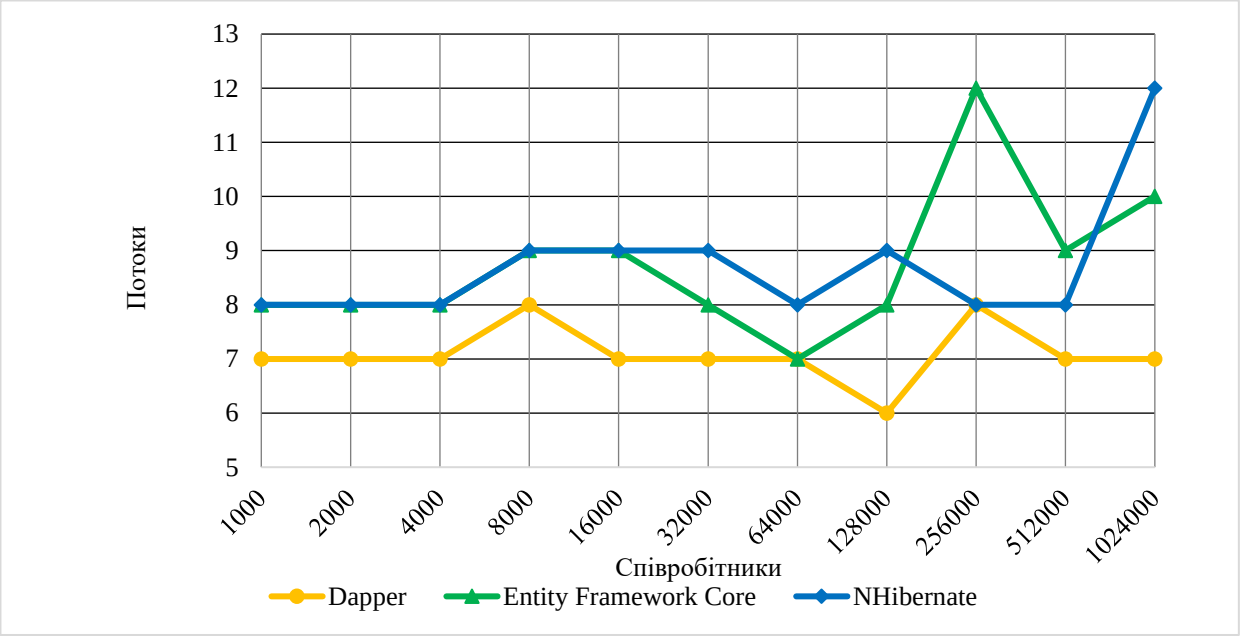

Рис *19. Графік залежності кількості задіяних активних потоків застосунком від кількості працівників під час запиту даних з однієї таблиці*

З отриманих метрик можна констатувати, що, при запиті даних з однієї, Dapper та Entity Framework Core показали майже однаковий результат швидкості читання, NHibernate показав значно гірший результат. Dapper задіяв найменший об’єм оперативної пам’яті, а NHibernate задіяв найбільший. Entity Framework Core та NHibernate показали невелике збільшення задіяних активних потоків, при збільшенні кількості даних, а показники потоків у сценаріях з Dapper майже не змінювались.

4.2.1.3 Метрики при виконанні запиту на отримання даних з таблиць пов’язаних відношенням "один до одного".

Збір метрик при отриманні даних з таблиць, з відношенням "один до одного", зроблено на прикладі запиту, у якому запрошується певна кількість записів з таблиць Employees та EmployeeDetails. Для того, щоб переконатися, що для усіх трьох ORM фреймворків код оброки запиту написано вірно, використано інструмент MS SQL Server Profiler [33], завдяки цьому інструменту можна відобразити у який SQL запит транслюється код запиту на тому чи іншому фреймворку. Нижче наведено скріншоти отриманих SQL запитів під час збору метрик.

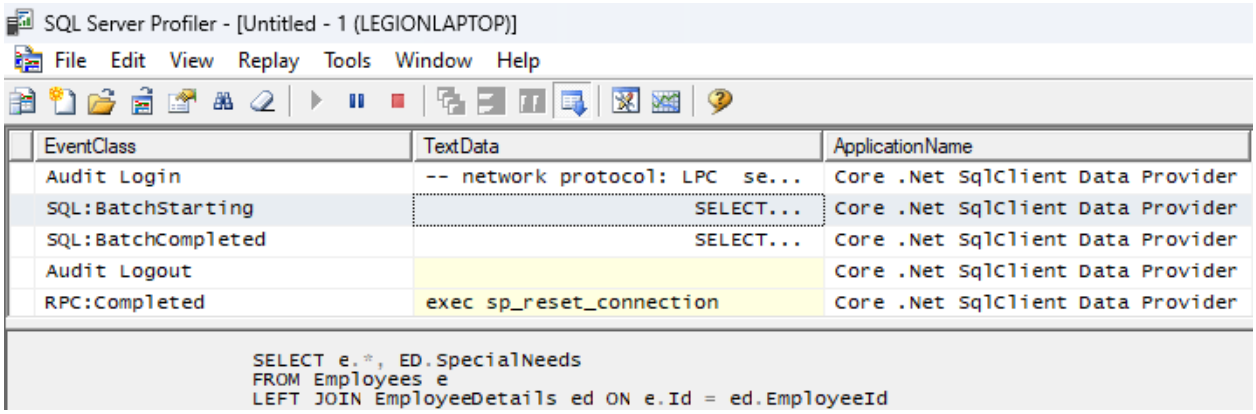

Рис *20. Dapper, трансльований запит до таблиць зі зв'язком "один до одного"*

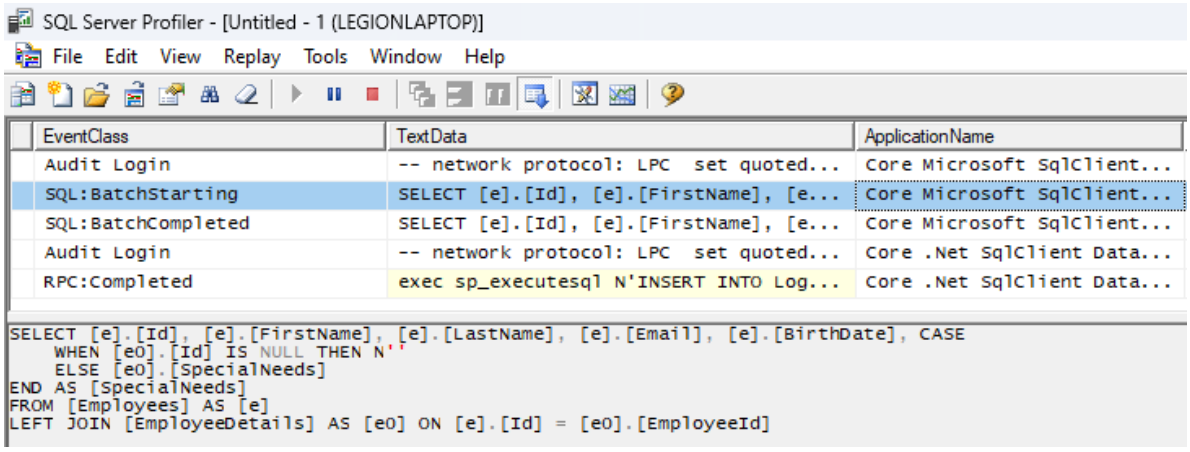

Рис *21*. *Entity Framework Core, трансльований запит до таблиць зі зв'язком "один до одного"*

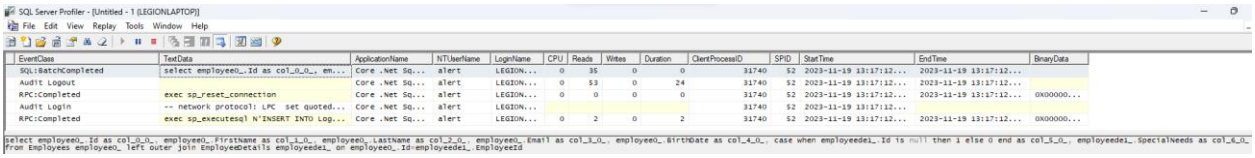

Рис *22*. *NHibernate, трансльований запит до таблиць зі зв'язком "один до одного"*

Виходячи з отриманих даних можна зробити висновок, що усі фреймворки утворюють запити однакової складності і код їх запитів написано вірно.

Нижче наведені таблиці та відповідні графіки, які показують залежність процесорного часу користувача, витраченого на функціонал читання ORM фреймворку з бази даних, зарезервованої оперативної пам’яті та кількості активних потоків від кількості записів співробітників, яких треба отримати з бази даних.

Таблиця *10*

Залежність процесорного часу користувача, витраченого на отримання даних з таблиць пов’язаних відношенням "один до одного"

	Час обробки ЦП (мікросекунди)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	15625	31250	31250

2000	46875	46875	31250
4000	15625	46875	46875
8000	46875	46875	46875
16000	15625	78125	31250
32000	78125	31250	93750
64000	62500	15625	187500
128000	140625	250000	468750
256000	343750	375000	984375
512000	437500	478125	2250000
1024000	500000	484375	3906250

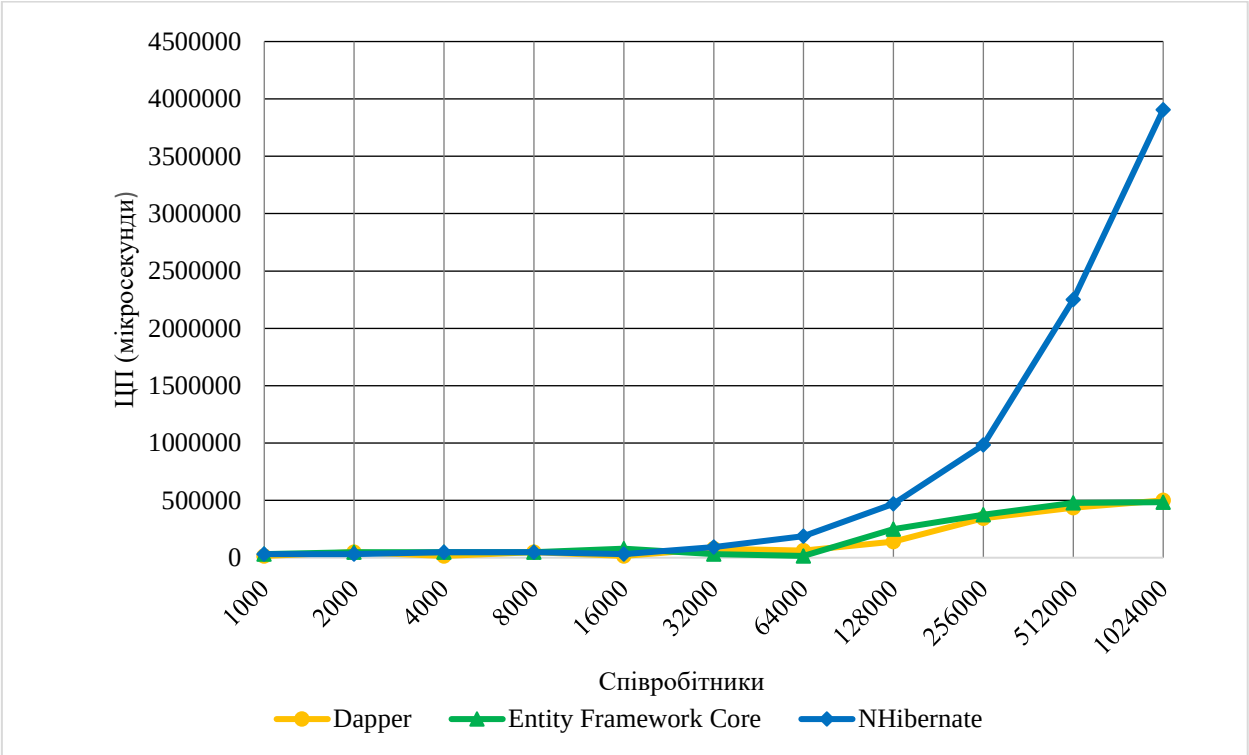

Рис *23. Графік процесорного часу користувача, витраченого на отримання даних з таблиць пов'язаних відношенням "один до одного"*

Таблиця *11*

Залежність зарезервованої оперативної пам'яті на отримання даних з таблиць пов'язаних відношенням "один до одного"

	ОЗП (МБ)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	68	88	89
2000	63	89	93
4000	64	91	102
8000	66	94	119
16000	69	101	153
32000	76	111	221
64000	91	140	261
128000	120	195	331
256000	178	275	690
512000	294	396	1199
1024000	522	813	1981

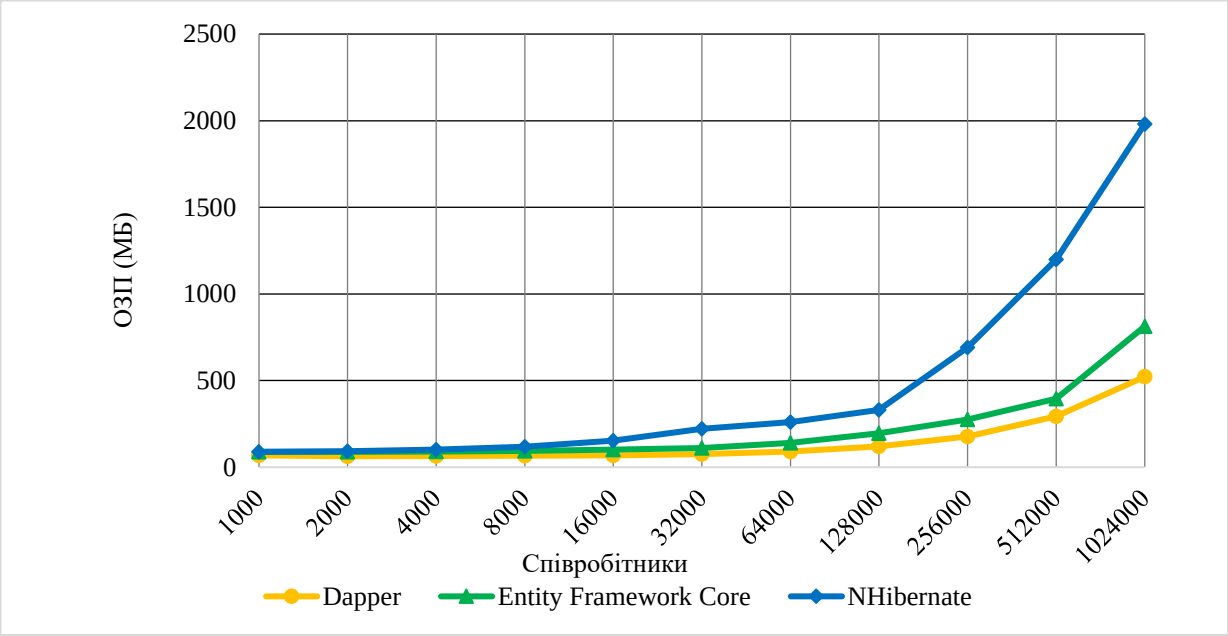

Рис *24. Графік залежності зарезервованої оперативної пам'яті на отримання даних з таблиць пов'язаних відношенням "один до одного"*

Таблиця *12*

Залежність кількості задіяних активних потоків застосунком від кількості працівників під час запиту даних з таблиць пов'язаних відношенням "один до одного"

	Потоки (кількість)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
1000	12	7	8
2000	7	7	8
4000	7	8	8
8000	7	9	8
16000	7	8	9
32000	7	8	9
64000	8	9	9
128000	8	9	9
256000	8	9	10
512000	8	9	12
1024000	9	9	12

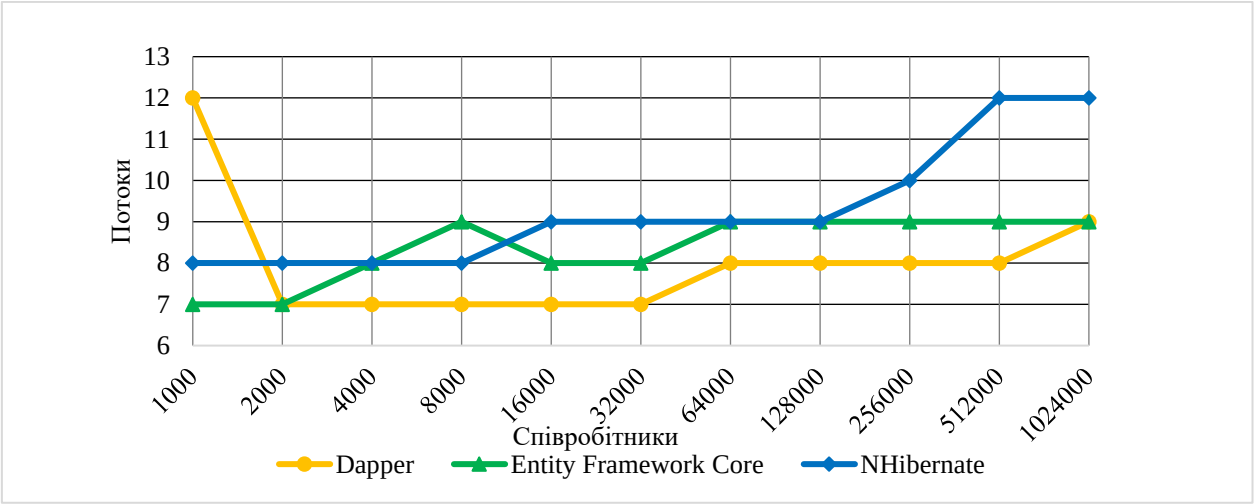

Рис *25. Графік залежності кількості задіяних активних потоків застосунком від кількості працівників під час запиту даних з таблиць пов'язаних відношенням "один до одного"*

З отриманих метрик можна констатувати, що, при запиті даних з таблиць пов’язаних відношенням "один до одного", Dapper показав найкращій результат швидкості читання при малої та середньої кількості даних, до півмільйона, а при збільшенні даних до мільйона Entity Framework Core майже зрівнявся з Dapper у швидкості читання, NHibernate показав значно гірший результат. Dapper задіяв найменший об’єм оперативної пам’яті, а NHibernate задіяв найбільший. Entity Framework Core та NHibernate показали невелике збільшення задіяних активних потоків, при збільшенні кількості даних для читання, а показники задіяних активних потоків у сценаріях з Dapper майже не змінювались.

4.2.1.4 Метрики при виконанні запиту на отримання даних з таблиць пов’язаних відношенням "один до багатьох".

Збір метрик при отриманні даних з таблиць, з відношенням "один до багатьох", зроблено на прикладі запиту, у якому запрошується певна кількість записів з таблиць Departments та Employees. Нижче наведено скріншоти отриманих SQL запитів за допомогою MS SQL Server Profiler під час збору метрик.

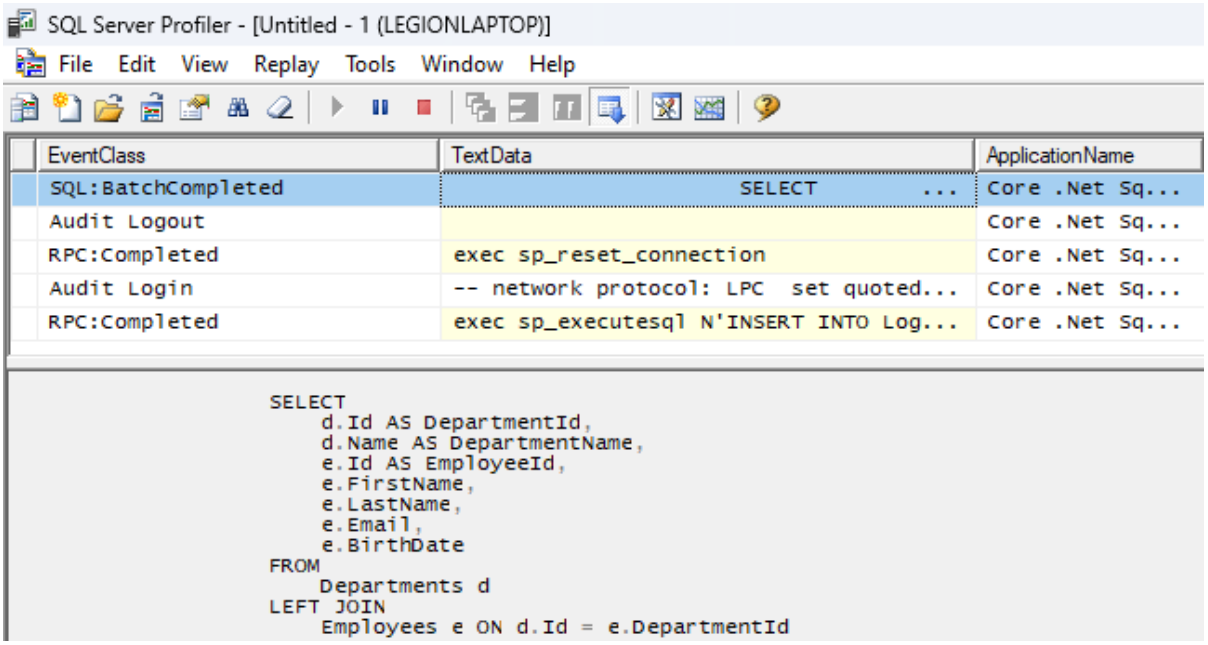

Рис *26. Dapper, трансльований запит до таблиць зі зв'язком "один до багатьох"*

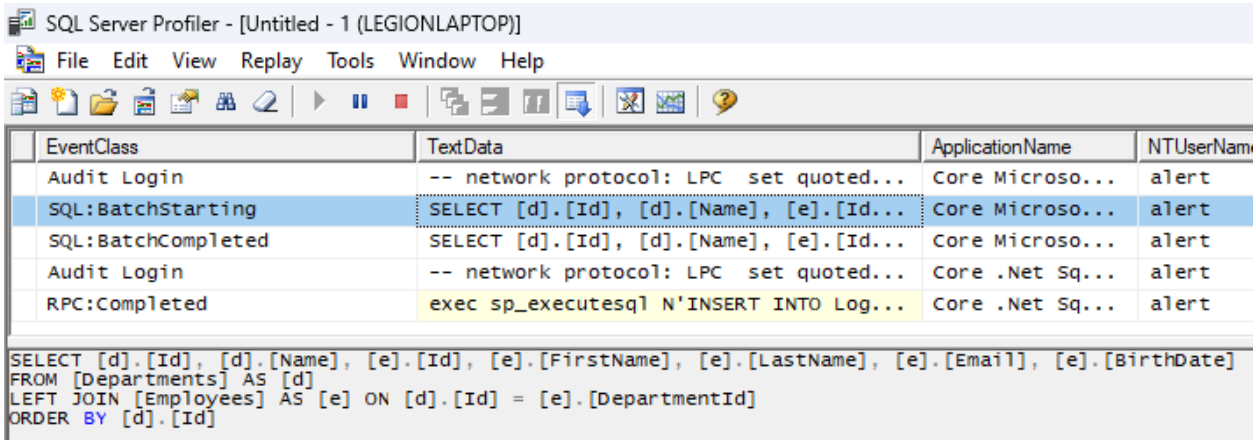

Рис *27. Entity Framework Core, трансльований запит до таблиць зі зв'язком "один до багатьох"*

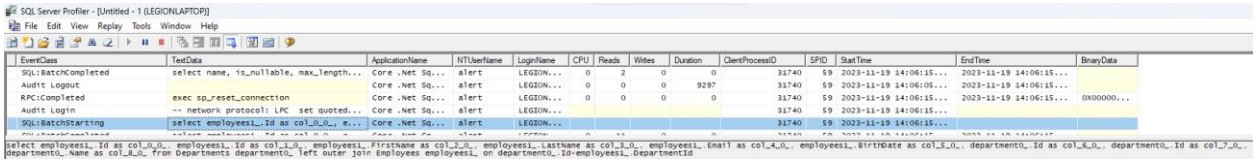

Рис *28. NHibernate, трансльований запит до таблиць зі зв'язком "один до багатьох"*

Виходячи з отриманих даних можна зробити висновок, що усі фреймворки утворюють запити однакової складності.

Нижче наведені таблиці та відповідні графіки, які показують залежність процесорного часу користувача, витраченого на функціонал читання ORM фреймворку з бази даних, зарезервованої оперативної пам’яті та кількості активних потоків від кількості записів відділів та співробітників (відношення записів відділів до записів співробітників 1 до 100) яких треба отримати з бази даних.

Таблиця *13*

Залежність процесорного часу користувача, витраченого на отримання даних з таблиць пов'язаних відношенням "один до багатьох"

	Час обробки ЦП (мікросекунди)		
Кількість відділів	Dapper	Entity Framework Core	NHibernate
10	15625	15625	31250
20	31250	46875	31250
40	31250	46875	46875
80	15625	15625	31250
160	46875	31250	109375
320	62500	281250	140625
640	93750	250000	140625
1280	171875	218750	437500
2560	187500	375000	718750
5120	421875	390625	2203125
10240	484375	531250	4718750

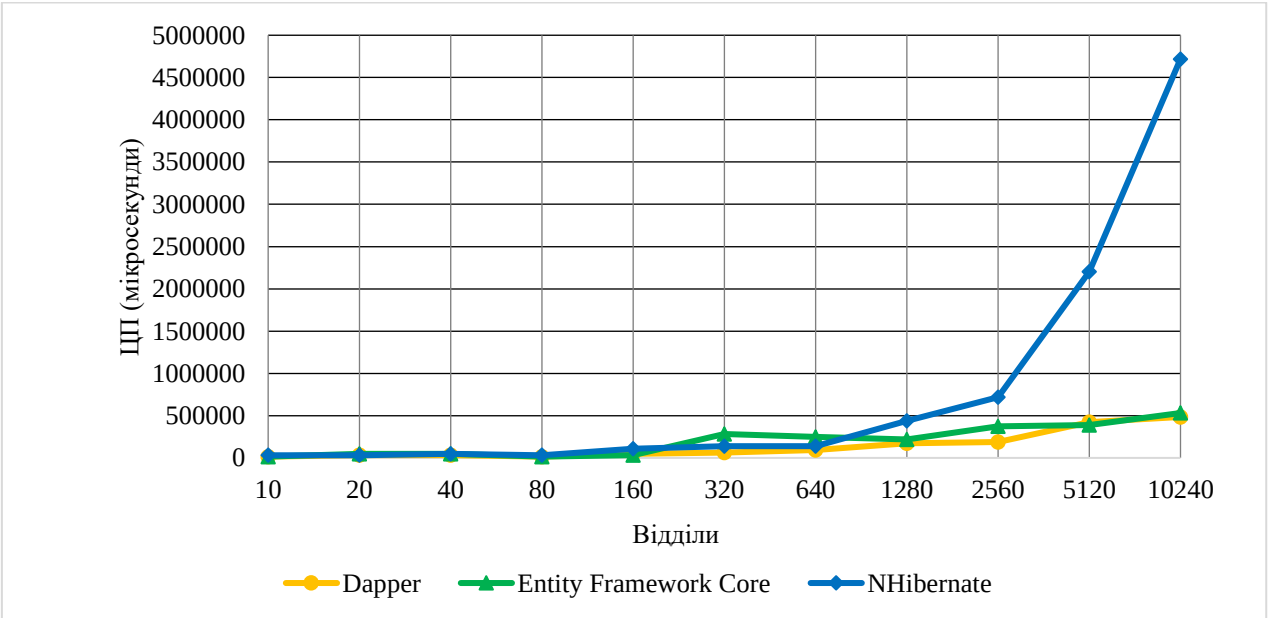

Рис *25*. *Графік залежності процесорного часу користувача, витраченого на отримання даних з таблиць пов'язаних відношенням "один до багатьох"*

Таблиця *14*

Залежність зарезервованої оперативної пам'яті на отримання даних з таблиць пов'язаних відношенням "один до багатьох"

	ОЗП (МБ)		
Кількість відділів	Dapper	Entity Framework Core	NHibernate
10	63	89	91
20	63	90	96
40	64	91	107
80	67	94	128
160	72	100	172
320	81	116	258
640	100	141	293
1280	139	190	609
2560	216	254	883
5120	284	376	1365
10240	575	771	2276

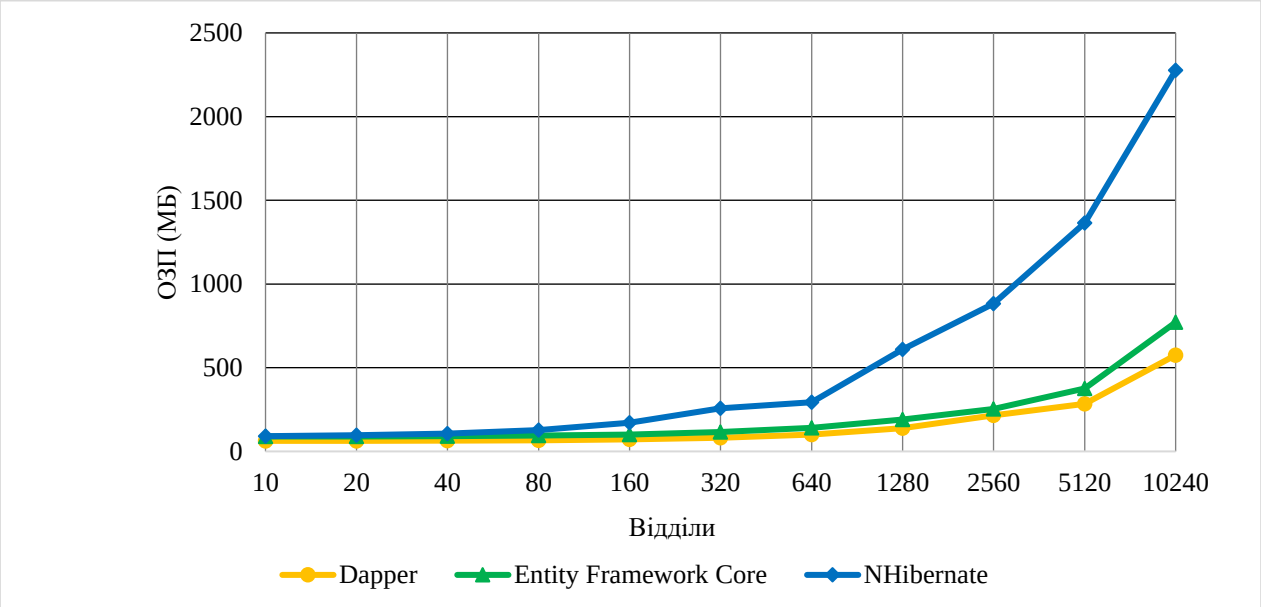

Рис *26. Графік залежності зарезервованої оперативної пам'яті на отримання даних з таблиць пов'язаних відношенням "один до багатьох"*

Таблиця *15*

Залежність кількості задіяних активних потоків застосунком від кількості відділів під час запиту даних з таблиць пов'язаних відношенням "один до багатьох"

	Потоки (кількість)		
Кількість співробітників	Dapper	Entity Framework Core	NHibernate
10	7	8	8
20	7	8	8
40	7	8	8
80	8	9	9
160	7	9	9
320	7	8	9
640	7	7	8
1280	6	8	9
2560	8	12	8
5120	7	9	8
10240	7	10	12

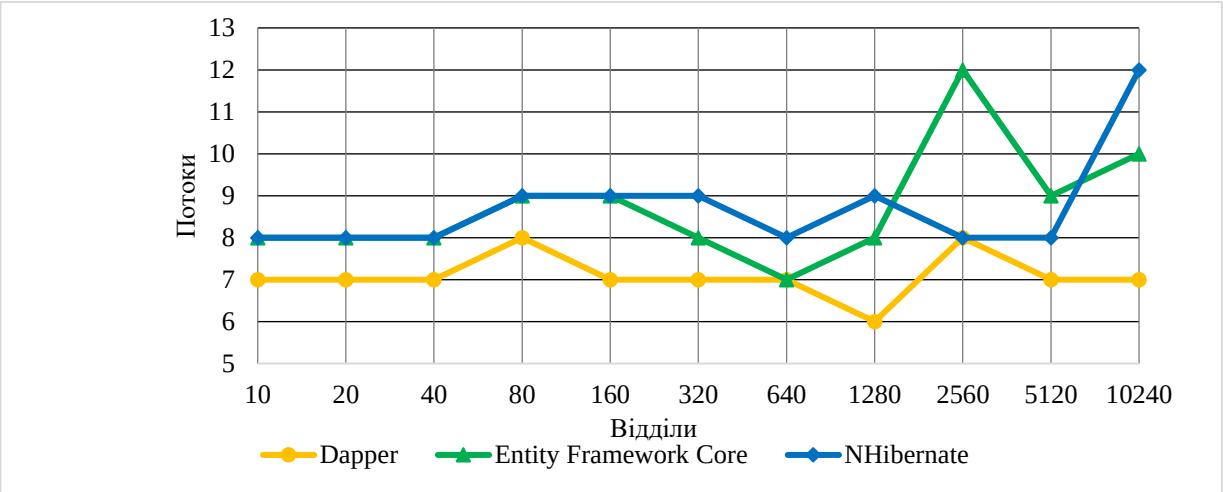

Рис *27*. *Графік залежності кількості задіяних активних потоків застосунком від кількості відділів під час запиту даних з таблиць пов'язаних відношенням "один до багатьох"*

З отриманих метрик можна констатувати, що, при запиті даних з таблиць пов’язаних відношенням "один до багатьох", Dapper продемонстрував найкращий результат при невеликій та середній кількості даних, до 2560 відділів та 256000 співробітників, а Entity Framework Core показав найкращий результат читання даних при їх великій кількості, від 5120 відділів та 512000 співробітників, трохи випередивши Dapper. NHibernate показав значно гірший результат. Dapper задіяв найменший об’єм оперативної пам’яті, а NHibernate задіяв найбільший. Entity Framework Core та NHibernate показали невелике збільшення задіяних активних потоків, при збільшенні кількості даних для читання, а показники задіяних активних потоків у сценаріях з Dapper майже не змінювались.

4.2.1.5 Метрики при виконанні запиту на отримання даних з таблиць пов’язаних відношенням "багато до багатьох".

Збір метрик при отриманні даних з таблиць, з відношенням "багато до багатьох", зроблено на прикладі запиту, у якому запрошується певна кількість записів з таблиць Employees та Projects через третю таблицю EmployeeProjects. Нижче наведено скріншоти отриманих SQL запитів за допомогою MS SQL Server Profiler під час збору метрик.

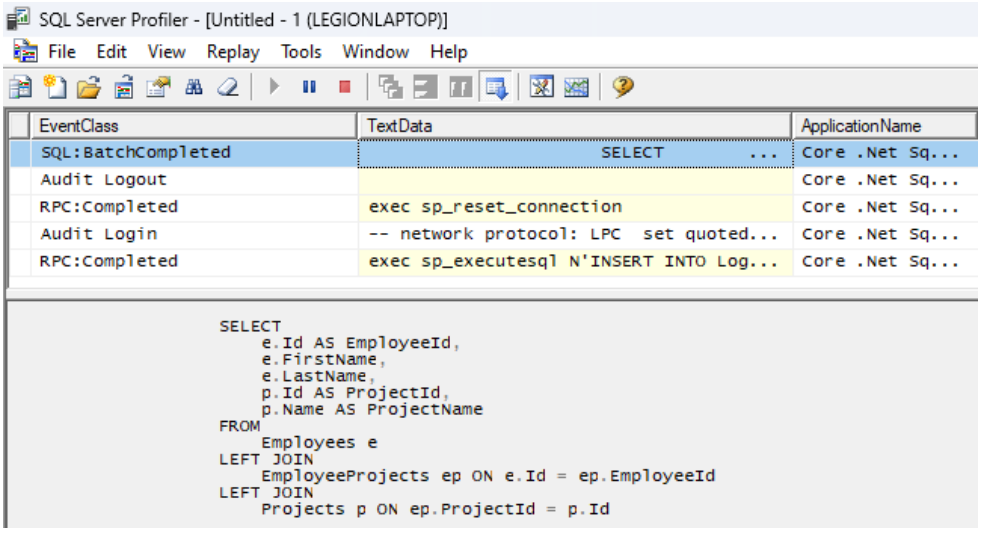

Рис *28. Dapper, трансльований запит до таблиць зі зв'язком "багато до багатьох"*

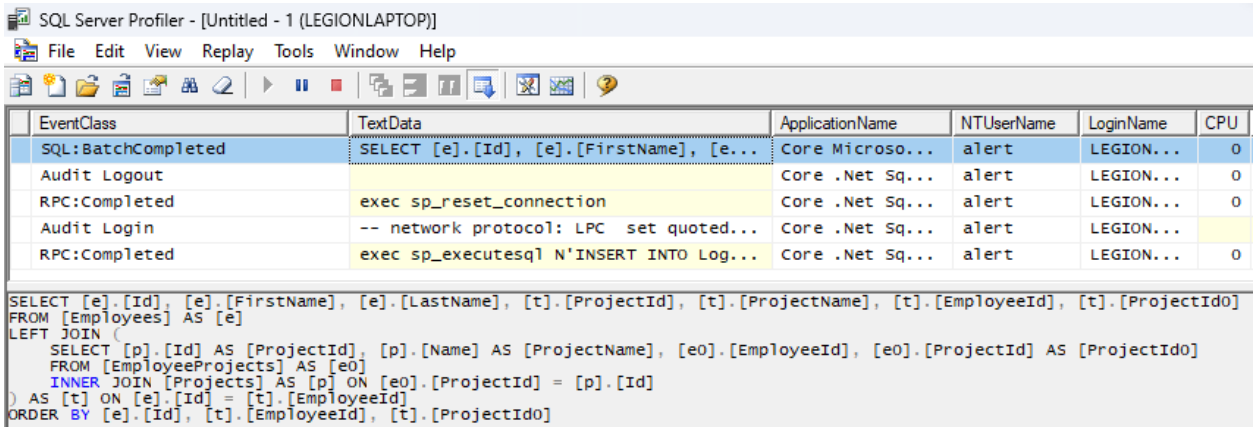

Рис *29. Entity Framework Core, трансльований запит до таблиць зі зв'язком "багато до багатьох"*

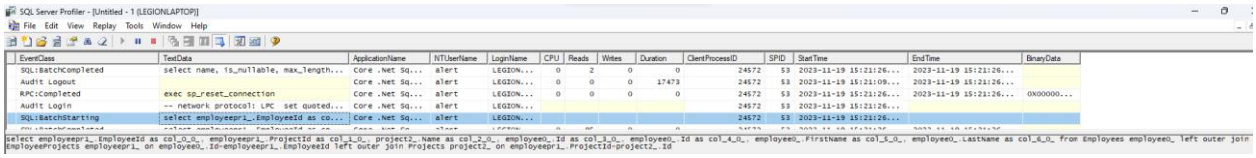

Рис *30. NHibernate, трансльований запит до таблиць зі зв'язком "багато до багатьох"*

Виходячи з отриманих даних можна зробити висновок, що Entity Framework Core, на відміну від Dapper та NHibernate, використовує підзапит при транслюванні запиту до таблиць зі зв’язком "багато до багатьох".

Нижче наведені таблиці та відповідні графіки, які показують залежність процесорного часу користувача, витраченого на функціонал читання ORM фреймворку з бази даних, зарезервованої оперативної пам’яті та кількості активних потоків від кількості записів проектів та співробітників (відношення записів проектів до записів співробітників 2 до 1, розподілення 10 проектів на співробітника) яких треба отримати з бази даних.

Таблиця *16*

Залежність процесорного часу користувача, витраченого на отримання даних з таблиць пов'язаних відношенням "багато до багатьох"

	Час обробки ЦП (мікросекунди)		
Кількість проектів	Dapper	Entity Framework Core	NHibernate
2000	31250	46875	46875
4000	46875	62500	62500
8000	31250	78125	93750
16000	31250	62500	296875
32000	109375	156250	437500
64000	296875	343750	609375
128000	359375	781250	765625
256000	1140625	1171875	3078125
512000	2437500	1937500	8546875
1024000	3562500	2343750	36718750

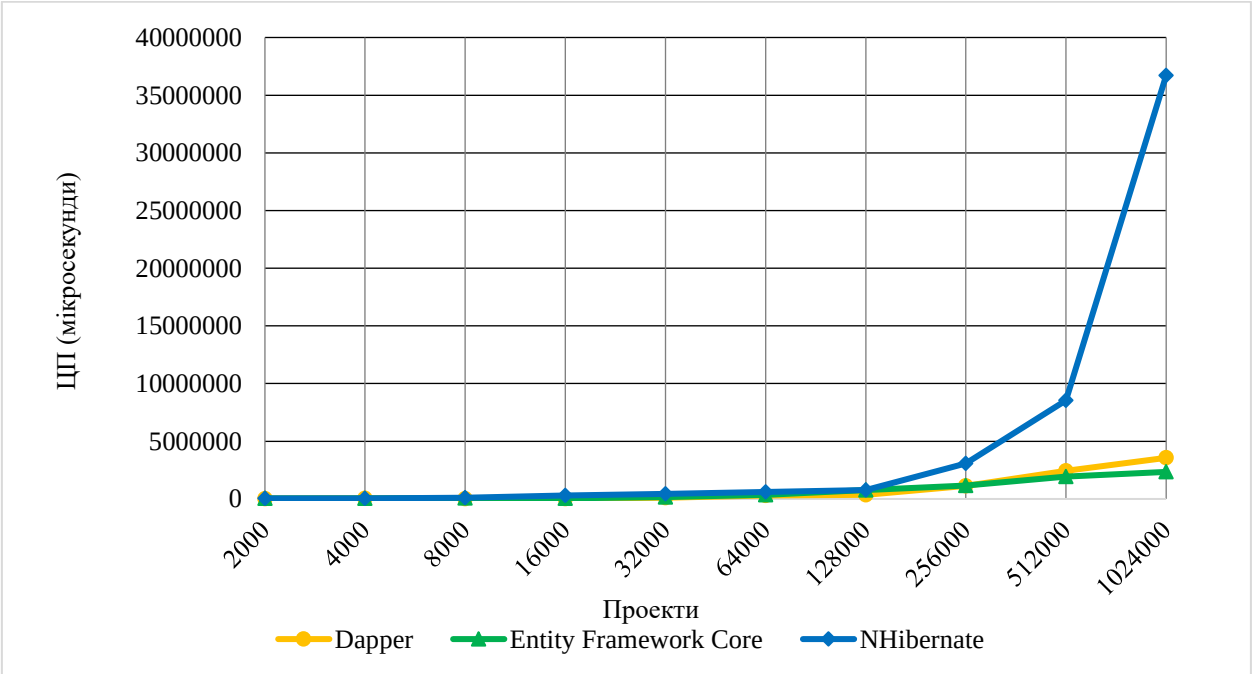

Рис *31*. *Графік залежності процесорного часу користувача, витраченого на отримання даних з таблиць пов'язаних відношенням "багато до багатьох"*

Таблиця *17*

Залежність зарезервованої оперативної пам'яті на отримання даних з таблиць пов'язаних відношенням "багато до багатьох"

	ОЗП (МБ)		
Кількість проектів	Dapper	Entity Framework Core	NHibernate
2000	65	93	103
4000	69	98	121
8000	77	108	157
16000	91	129	231
32000	121	171	282
64000	180	253	540
128000	256	297	1112
256000	387	619	1426
512000	898	1169	2486
1024000	1819	1781	4189

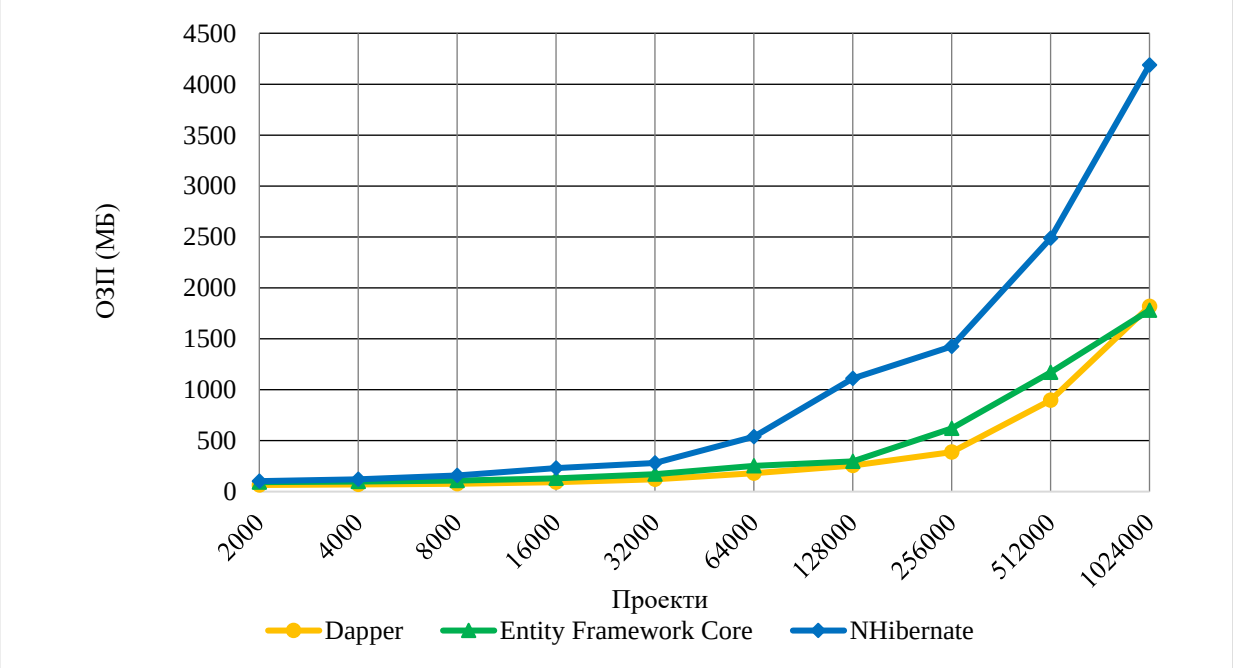

Рис *32. Графік залежності зарезервованої оперативної пам'яті на отримання даних з таблиць пов'язаних відношенням "багато до багатьох"*

Таблиця *18*

Залежність кількості задіяних активних потоків застосунком від кількості відділів під час запиту даних з таблиць пов'язаних відношенням " багато до багатьох"

	Потоки (кількість)		
Кількість проектів	Dapper	Entity Framework Core	NHibernate
2000	7	8	9
4000	7	8	9
8000	7	8	8
16000	6	9	8
32000	7	9	9
64000	7	8	10
128000	7	9	9
256000	7	10	12
512000	8	12	12
1024000	7	10	14

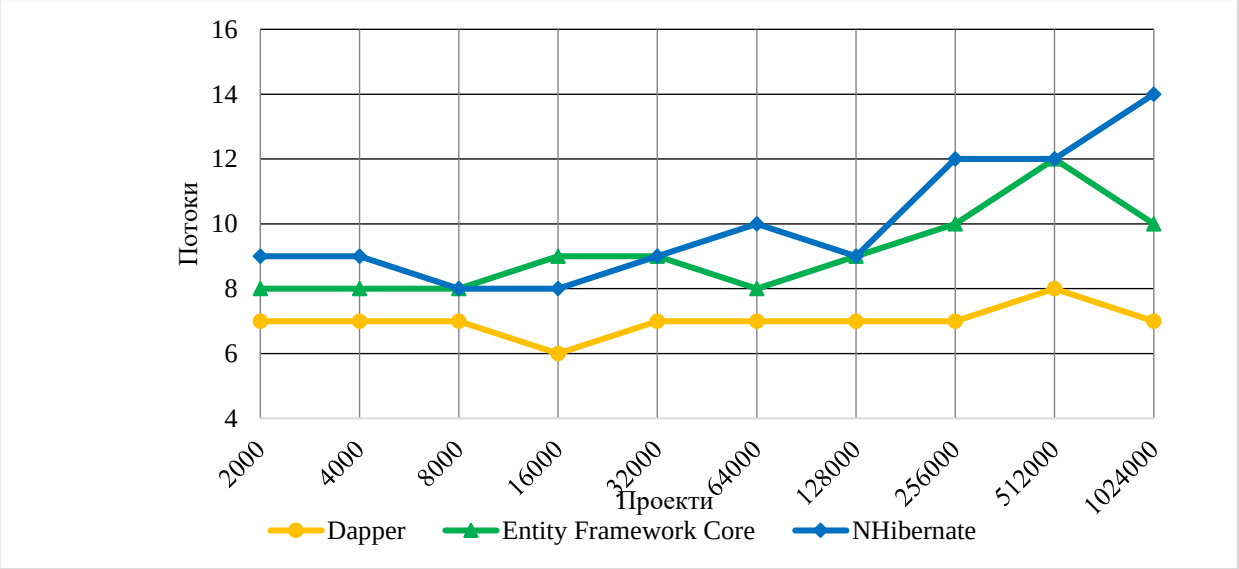

Рис *33. Графік залежності кількості задіяних активних потоків застосунком від кількості відділів під час запиту даних з таблиць пов'язаних відношенням "багато до багатьох"*

З отриманих метрик можна констатувати, що, при запиті даних з таблиць пов’язаних відношенням "багато до багатьох", Dapper продемонстрував найкращий результат при невеликій та середній кількості даних, до чверті мільйона записів, а Entity Framework Core показав найкращий результат читання даних при їх великій кількості, випередивши Dapper. NHibernate показав значно гірший результат. Dapper та Entity Framework Core задіяли майже однаковий об’єм оперативної пам’яті, а NHibernate задіяв найбільший. Entity Framework Core та NHibernate показали невелике збільшення задіяних активних потоків, при збільшенні кількості даних для читання, а показники задіяних активних потоків у сценаріях з Dapper майже не змінювались.

4.2.1.6 Метрики при виконанні запиту на оновлення даних в таблиці.

Збір метрик при оновлені даних у таблиці, зроблено на прикладі запиту, у якому оновлюється назва усіх відділів, у таблиці Departments, шляхом додавання до назви певного префіксу. Кількість записів у таблиці Departments, які фігурували у дослідженні, змінювалась від 1000 до 1000000. Усі три фреймворки продемонстрували швидке виконання запитів на оновлення даних, не маючи суттєвих відмінностей між собою у використаному процесорному часі, кількості задіяних активних потоків та об’єму зарезервованої оперативної пам’яті. Це обумовлюється тим, що усі досліджувані фреймворки мають функціонал, який дозволяє оновлювати дані в базі, без необхідності попереднього завантаження даних з бази та зберігання їх у пам’яті. На кількості записів у один мільйон – середні показники для кожного фреймворку становили: до 175 мілісекунд використаного процесорного часу користувача, до 120 мегабайт зарезервованої оперативної пам’яті, до 9 задіяних активних потоків.

4.2.1.7 Метрики при виконанні запиту на видалення даних в таблиці.

Збір метрик при видалені даних у таблиці, зроблено на прикладі запиту, який видаляє усі дані, у усіх таблицях в базі даних.

Кількість записів у таблицях, які фігурували у дослідженні, змінювалась від тисяч до сотень тисяч. Усі три фреймворки, як і у сценарії з оновленням даних, продемонстрували швидке виконання запитів на видалення даних, не маючи суттєвих відмінностей між собою у використаному процесорному часі, кількості задіяних активних потоків та об’єму зарезервованої оперативної пам’яті, знов завдяки тому, що фреймворки мають функціонал видалення даних з бази, який не потребує їх попереднього завантаження. На кількості записів у 512000 співробітників, 4000 відділів, 10000 проектів, та розподілу 2 співробітника на проект – середні показники для кожного фреймворку становили: до 200 мілісекунд використаного процесорного часу користувача, до 100 мегабайт зарезервованої оперативної пам’яті, до 8 задіяних активних потоків.

4.2.2 Інструментарій

У процесі порівняння ORM фреймворків важливою є не тільки об'єктивність критеріїв, але і їхнє коректне та всебічне застосування. Від вибору правильних інструментів залежить кінцева якість системи і її ефективність. Основні інструменти і функції, які надаються вибраними ORM фреймворками:

– Dapper. Відрізняється своєю простотою та високою продуктивністю. На відміну від повноцінних ORM, Dapper не надає можливостей автоматичного мапінгу бази даних та генерації SQL запитів, проте він відмінно виконує свою основну задачу - відображення SQL запитів на .NET об'єкти. Зокрема, Dapper підтримує роботу з параметризованими запитами та автоматичну конвертацію результатів SQL запитів в .NET об'єкти;

– Entity Framework Core. EFC - це більш повноцінний ORM фреймворк, який надає багатий набір інструментів для роботи з реляційними базами даних. Засоби EFC дозволяють автоматично генерувати SQL запити на основі LINQ запитів, виконувати tracking changes (відслідковування змін у даних), а також мігрувати бази даних;

– NHibernate. На відміну від Dapper та EFC, NHibernate не є вбудованим у .NET, проте він надає багатий набір інструментів та функцій. Це включає підтримку мапінгу через XML файли або Fluent API, кешування на рівні сесії, виконання пакетних операцій та додаткові можливості для оптимізації продуктивності.

Отже, кожен з розглянутих фреймворків має свої переваги та особливості. Незважаючи на їх відмінності, всі три фреймворки здатні забезпечити потреби більшості розробників у побудові надійних та ефективних систем, що працюють з реляційними базами даних. У подальшому аналізі ці інструменти будуть оцінюватися за критеріями, викладеними у попередньому розділі.

4.2.3 Масштабованість

Масштабованість, як критерій, є важливим для визначення того, наскільки добре ORM фреймворк може працювати в умовах великого обсягу даних або високого навантаження. Це включає, але не обмежується, визначенням продуктивності фреймворку при збільшенні обсягу даних і аналізом можливостей фреймворку в управлінні пам'яттю. Результат порівняння масштабованості трьох ORM фреймворків:

– Dapper. Оскільки Dapper є мікро ORM, він зосереджений на високій продуктивності і невеликому використанні ресурсів. Він не надає розширених можливостей управління пам'яттю або мапінгу об'єктів, що знижує його навантаження на систему. Це робить Dapper відмінним вибором для систем з великим обсягом даних або високим навантаженням, де основна вимога - швидкість і ефективність;

– EFC є повноцінним ORM фреймворком, який надає багатий набір можливостей, але може бути менш продуктивним при великому обсягу даних або високому навантаженні. Однак, EFC надає можливості для оптимізації продуктивності, такі як ліниве завантаження, кешування запитів, використання виглядів, що можуть допомогти покращити масштабованість;

– NHibernate є багатофункціональним ORM фреймворком з можливістю кешування на рівні сесії, батчінгу запитів та оптимізації запитів, що можуть допомогти збільшити масштабованість. Однак, подібно до EFC, він може виявитися менш продуктивним при великих обсягах даних або високому навантаженні в порівнянні з Dapper.

Всі три фреймворки мають свої переваги та недоліки в контексті масштабованості, і вибір між ними залежить від конкретних вимог проекту.

4.2.4 Функціональність

Функціональність є важливим критерієм порівняння, який визначає набір можливостей, що надаються ORM фреймворками. Це включає такі аспекти, як підтримка різних типів відношень між об'єктами, способи виконання запитів, підтримка транзакцій, кешування, тощо. Розглянемо функціональність трьох ORM фреймворків:

– Dapper. Як мікро ORM, Dapper пропонує обмежений, але дуже ефективний набір функцій. Він в основному зосереджений на виконанні SQL запитів і мапінгу результатів на об'єкти. Dapper не надає вбудовану підтримку для виконання CRUD операцій (доведеться писати власні SQL запити), трекінгу змін, кешування або роботи з транзакціями;

– EFC - це повнофункціональний ORM фреймворк, який надає багатий набір можливостей. Він підтримує виконання CRUD операцій, трекінг змін, ліниве та жадібне завантаження, кешування, використання транзакцій і багато іншого. EFC також має багатий API для формування запитів, який дозволяє створювати складні запити без прямого написання SQL;

– NHibernate. NHibernate, подібно до EFC, є повноцінним ORM фреймворком і надає широкий спектр функцій. Він підтримує CRUD операції, відстеження змін, "ліниве завантаження" та "жадібне завантаження", розширене кешування, роботу з транзакціями тощо. NHibernate також має потужний API для формування запитів, включаючи HQL (Hibernate Query Language) і Criteria API.

Отже, вибір фреймворку залежить від специфічних вимог до функціональності проекту. Якщо проект потребує багатофункціонального рішення з багатьма додатковими можливостями, то Entity Framework Core або NHibernate можуть бути відмінним вибором. Проте, якщо вимагається швидкість і ефективність з меншим набором функцій, то Dapper може бути більш доцільним варіантом.

4.2.5 Документація та спільнота

Чітка, повна документація допомагає швидко зрозуміти та ефективно використовувати фреймворк. Активна спільнота може забезпечити підтримку, рішення для специфічних проблем, а також забезпечити постійне оновлення та вдосконалення фреймворку. Розглянемо документацію та спільноту фреймворків:

Dapper має добре організовану документацію, яка детально описує основні аспекти використання фреймворку. Хоча обсяг документації Dapper може бути меншим в порівнянні з іншими, це пояснюється меншим обсягом функціональності, яку він надає. Спільнота Dapper активна і швидко реагує на запитання та проблеми;

EFC має велику та детальну документацію, яка покриває великий спектр тем, від основ до складних аспектів використання фреймворку. Спільнота EFC є однією з найбільших, з великим числом активних користувачів, які можуть надати підтримку та розробляти нові функції;

NHibernate має багатий набір документації, який декілька разів перебував у процесі переписування і вдосконалення. Він покриває велику кількість тем, проте є деякі скарги на його актуальність і складність. Спільнота NHibernate менш активна в порівнянні з EFC, але все ж достатньо велика для того, щоб надати підтримку користувачам.

Отже, при виборі ORM фреймворку необхідно враховувати не тільки його технічні характеристики, але й доступність і якість документації, а також

активність та розмір спільноти. Кожен з розглянутих фреймворків має свої сильні та слабкі сторони в цих аспектах, тому вибір має здійснюватися в залежності від специфічних потреб проекту.

4.2.6 Порівняння на основі легкості використання та гнучкості

У сфері програмування, легкість використання та гнучкість - це два ключових параметри, які мають велике значення при виборі технологій та інструментів для розробки. Вони визначають, наскільки просто розробник може розпочати роботу з фреймворком, а також наскільки легко внести зміни або налаштувати його для специфічних потреб.

Всі досліджувані фреймворки мають різний рівень легкості використання та гнучкості.

Dapper може бути віднесений до "легких" ORM фреймворків. Він не намагається автоматизувати все, а натомість дозволяє розробникам мати більший контроль над SQL запитами, що використовуються. Це може вважатися перевагою в термінах гнучкості, але в той же час це може зробити Dapper менш привабливим для розробників, які шукають більш автоматизовані рішення.

Entity Framework Core є більш "важким" ORM фреймворком, який намагається автоматизувати більше аспектів роботи з базою даних. Це робить його привабливим для розробників, які шукають ORM фреймворк з "батарейками". Він має розширену систему конфігурації, що дозволяє гнучко налаштовувати його роботу, але в той же час ця складність може зробити його менш привабливим для новачків.

NHibernate, на кшталт Entity Framework Core, також є "важким" ORM фреймворком. Він має велику кількість функцій та гнучку систему конфігурації. Незважаючи на це, NHibernate може бути важким для вивчення та використання, особливо для новачків, через його об'ємну документацію та високий рівень складності.

У загальному, Dapper виглядає більш простим для використання, але менш гнучким ніж Entity Framework Core і NHibernate. В той же час, Entity

Framework Core і NHibernate, маючи більші можливості, вимагають більше часу та зусиль для вивчення та налаштування.

4.3 Опрацювання рекомендацій щодо використання досліджених фреймворків

Беручи до уваги дані, представлені в таблицях 4 - 18, можна зробити декілька важливих висновків.

Першим критерієм для порівняння є продуктивність. Виконання CRUD операцій у кожному з ORM фреймворків показало, що вони відрізняються за показниками часу виконання операцій на процесорі, використання пам'яті та кількістю активних потоків.

Усі три досліджувані фреймворки показали відмінні результати у сценаріях з оновленням та видаленням даних, немаючі суттєвих відмінностей один від одного.

Dapper продемонстрував найкращі, порівнювальні, результати майже у усіх інших сценаріях за часом виконання запитів, резервуванням пам’яті та мінімумом задіяних активних потоків, особливо при роботі з великою кількістю даних, поступившись лише Entity Framework Core у сценарії запиту даних до таблиць з відношенням "багато до багатьох", при великій кількості даних для читання.

Entity Framework Core найгірші результати при записі даних, найкращі результати при читанні великої кількості даних за таблиць з відношенням "багато до багатьох", у усіх інших сценаріях показав результати, які наближені, або гірше да Dapper, за всіма показниками.

NHibernate випередив Entity Framework Core лише у сценарії запису даних, у усіх інших сценаріях його показники процесорного часу та використання оперативної пам’яті булі найгірші.

З погляду легкості використання та гнучкості, кожен фреймворк має свої відмінності. Dapper, хоч і виглядає простішим для використання, має обмежені

можливості налаштування. EFC та NHibernate, хоча і надають більше можливостей, вимагають більше часу та зусиль для освоєння.

Підсумовуючи дані проведеного порівняльного дослідження можна виставити кілька рекомендацій щодо використання досліджених ORM-фреймворків в контексті платформи Microsoft .NET.

Dapper виявився найкращим у контексті продуктивності, а також показав найменше використання ресурсів. Це робить його відмінним вибором для проектів, де необхідна висока продуктивність і мінімальне використання ресурсів, як наприклад мікросервіс. Це можуть бути проекти високого навантаження, такі як пошукові веб-сайти або мобільні програми, де швидкодія та ефективне використання ресурсів є критично важливими.

EFC показав середні результати в усіх випробуваннях, але він має додаткову вигоду в тому, що його легше освоїти та використовувати. Це робить його відмінним вибором для проектів, де важливішими є легкість використання та розробка, а не нескінченна оптимізація продуктивності. Це можуть бути менш масштабні проекти або проекти, де розробникам потрібно швидко виконувати зміни без занурення в складні деталі ORM.

NHibernate виявився найбільш витратним з точки зору використання ресурсів на базових налаштуваннях, його використання може бути рекомендовано для корпоративних великих проектів, коли є гарне розуміння у програмістів усіх налаштувань фреймворку, особливо кешування, для зменшення часу на обробку запитів, та потреба роботи з різними серверами баз даних, а не тільки MS SQL Server.

Висновки з розділу 4

У даному розділі було проведено детальний аналіз та порівняння трьох об'єктно-реляційних фреймворків для платформи Microsoft .NET: Dapper, Entity Framework Core та NHibernate. Кожен з них був оцінений за п'ятьма ос-

новними критеріями: швидкістю виконання запитів до бази даних, інструментарієм, масштабованістю, функціональністю, та рівнем підтримки у вигляді документації та спільноти.

Основними методами аналізу були розробка та використання програмного забезпечення для дослідження об'єктно-реляційних фреймворків, включаючи розробку специфічних алгоритмів та модулів для вимірювання кількісних результатів. Програмне забезпечення було використане для виконання серії CRUD операцій в кожному з ORM фреймворків і вимірювання різних показників продуктивності.

Результати аналізу показали, що кожен з фреймворків має свої сильні та слабкі сторони, що робить їх більш або менш придатними для різних типів проектів. Dapper виявився найбільш продуктивним та ефективним з точки зору використання ресурсів, що робить його ідеальним вибором для високонавантажуваних проектів. Entity Framework Core, з іншого боку, виявився найлегшим для використання, що робить його відмінним вибором для менш масштабних проектів або проектів, де швидкість розробки є важливішою за оптимізацію продуктивності. NHibernate показав невисоку продуктивність при великих об'ємах даних на базових налаштуваннях, що говорить про те, що цей фреймворк потребує додаткових налаштувань, щоб бути продуктивним.

ВИСНОВКИ

1. Було визначено концепцію ORM та її роль у сучасному програмуванні.
2. Досліджено стандарти ORM на платформі .NET та проаналізовано переваги та недоліки їх використання. Виявлено, що ORM фреймворки забезпечують високий рівень абстракції при роботі з базами даних, що може сприяти підвищенню продуктивності розробки, але також можуть призвести до зниження продуктивності виконання в певних ситуаціях, особливо при некоректному використанні.
3. Проведено детальний огляд та аналіз ключових ORM фреймворків, таких як Entity Framework Core, NHibernate, Dapper. Кожен з цих фреймворків був оцінений за рядом критеріїв, зокрема за особливостями створення моделей, роботи з даними, інтеграції з .NET, сценаріями використання, а також перевагами та недоліками.
4. Проведено огляд та вибір технологій для розробки. Обрані мова програмування C#, середовище розробки Microsoft Visual Studio та СУБД MS SQL Server були обґрунтовані на основі їх характеристик та вимог до програмного забезпечення. Були визначені вимоги до ORM фреймворків для порівняння, а також виконано проектування бази даних та розробку основних запитів.
5. Була проведена апробація розробленого застосунку з порівнянням ORM фреймворків за критеріями продуктивності, інструментарію, масштабованості, функціональності, документації та спільноти. В результаті було зроблено висновок, що кожен ORM фреймворк має свої переваги та недоліки, і вибір між ними повинен здійснюватися в залежності від конкретних вимог та контексту проекту.
6. Проведено аналіз результатів дослідження та дані рекомендації щодо використання досліджених фреймворків.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Сєров А.Л., Коломоєць Г.П. Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .Net. Збірник наукових праць студентів, аспірантів, докторантів і молодих вчених «Молода наука-2023» / Запорізький національний університет. – Запоріжжя : ЗНУ, 2023. Т.5. С. 121-122.

2. Сєров А.Л. – магістрант, Коломоєць Г.П. доцент, канд. фіз.-мат. наук – науковий керівник. Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .Net. Матеріали III Всеукраїнської науково-практичної конференції за участю молодих науковців «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України». Запорізький національний університет. Запоріжжя: ЗНУ, 2023. ??? с.

3. Medium. Mastering in ORM. URL: https://techbuddy.medium.com/orm-in-dotnet-4b87fbb3f117 (дата звернення: 02.08.2023).

4. Baeldung. What Is an ORM? How Does It Work? How Should We Use One? URL: https://www.baeldung.com/cs/object-relational-mapping (дата звернення: 02.08.2023).

5. Highload today. ASP.NET Що таке ASP.NET? Принцип функціонування та моделі розробки. URL: https://highload.today/uk/shho-take-asp-net-printsip-funktsionuvannya-ta-modeli-rozrobki/ (дата звернення: 03.08.2023).

6. Про Програмування. Операції з даними. URL: https://abitap.com/1-5-operacziyi-z-danymy-crud/ (дата звернення: 03.08.2023).

7. Telerik. .NET Basics: ORM. URL: https://www.telerik.com/blogs/dot-net-basics-orm-object-relational-mapping (дата звернення: 05.08.2023).

8. Openclassrooms. Identify Object-Relational Mapping (ORM) Tools for .NET. URL: https://openclassrooms.com/en/courses/5671811-implement-a-relational-database-with-asp-net-core/6588450-identify-object-relational-mapping-orm-tools-for-net (дата звернення: 05.08.2023).

9. Microsoft Learn. Entity Framework Core. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 07.08.2023).

10. Microsoft Learn. Language Integrated Query (LINQ). URL: https://learn.microsoft.com/en-us/dotnet/csharp/linq/ (дата звернення: 07.08.2023).

11. Microsoft Learn. Creating and Configuring a Model. URL: https://learn.microsoft.com/en-us/ef/core/modeling/ (дата звернення: 07.08.2023).

12. Microsoft Learn. DbContext Lifetime, Configuration, and Initialization. URL: https://learn.microsoft.com/en-us/ef/core/dbcontext-configuration/ (дата звернення: 07.08.2023).

13. C# corner. ASP.NET CORE - Learn CRUD Operations in Entity Framework Core from Zero to Hero. URL: https://www.c-sharpcorner.com/article/asp-net-core-learn-crud-operations-in-entity-framework-core-from-zero-to-hero/ (дата звернення: 07.08.2023).

14. Abdullah Eren Güvercin, Bilgin Avenoğlu. Performance Analysis of Object-Relational Mapping (ORM) Tools in .Net 6 Environment. 2022. URL: https://dergipark.org.tr/en/download/article-file/2198861 (дата звернення: 08.08.2023)

15. NHibernate. Reference Documentation. URL: https://nhibernate.info/doc/nh/en/nhibernate_reference.pdf (дата звернення: 10.08.2023)

16. Gunnar Peipman. NHibernate on ASP.NET Core. 2019. URL: https://gunnarpeipman.com/aspnet-core-nhibernate/ (дата звернення: 11.08.2023)

17. Medium. Simplify and Optimize: How Dapper, the Micro ORM, Transforms Data Retrieval into a Seamless and Efficient Process. URL: https://medium.com/@nirajranasinghe/simplify-and-optimize-how-dapper-the-micro-orm-

transforms-data-retrieval-into-a-seamless-and-d30b6f9799d1 (дата звернення: 13.08.2023)

18. Microsoft Learn. IDbTransaction Interface. URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 13.08.2023).

19. Learn Dapper. Using Relationships with Dapper. URL: https://www.learndapper.com/relationships (дата звернення: 13.08.2023)

20. Manoj Phadnis. EF VS DAPPER. URL: https://www.manojphadnis.net/entity-framework/ef-vs-dapper (дата звернення: 14.08.2023).

21. Cleveroad. Advantages of using python over other languages. URL: https://www.cleveroad.com/blog/python-vs-other-programming-languages/ (дата звернення: 15.08.2023).

22. Geeks for geeks. Introduction to Java. URL: https://www.geeksforgeeks.org/introduction-to-java/?ref=lbp (дата звернення: 16.08.2023).

23. Software Testing Help. C# Vs C++ And C# Vs Java – Explore the Key Differences. https://www.softwaretestinghelp.com/csharp-vs-cpp-vs-java/ (дата звернення: 16.08.2023).

24. Microsoft Learn. What is Visual Studio? URL: https://learn.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2022 (дата звернення: 16.08.2023).

25. Turing. Your Ultimate Guide to Visual Studio vs Visual Studio Code. URL: https://www.turing.com/kb/ultimate-guide-visual-studio-vs-visual-studio-code (дата звернення: 17.08.2023).

26. CodingSight. Visual Studio vs JetBrains Rider: A Detailed Comparison. URL: https://codingsight.com/visual-studio-vs-jetbrains-rider-a-detailed-comparison/ (дата звернення: 18.08.2023)

27. SqlServerTutorial.net. What is SQL Server. URL: https://www.sqlservertutorial.net/getting-started/what-is-sql-server/ (дата звернення: 20.08.2023)

28. Hostinger. MySQL vs SQL: Overview, Similarities, Differences. URL: www.hostinger.com/tutorials/difference-between-mysql-and-sql-server (дата звернення: 20.08.2023)

29. Neovera. MySQL vs. Microsoft SQL Server vs. Oracle: What Is the Right Relational Database Management System (RDBMS) for Your Business? URL: https://neovera.com/choosing-right-rdbms-oracle-mysql-sql-server/ (дата звернення: 20.08.2023)

30. GitHub. Swashbuckle.AspNetCore. URL: https://github.com/domaindrivendev/Swashbuckle.AspNetCore (дата звернення: 20.10.2023)

31. GitHub. Bogus. URL: https://github.com/bchavez/Bogus (дата звернення: 25.10.2023)

32. GitHub. Universe.CpuUsage. URL: https://github.com/devizer/Universe.CpuUsage (дата звернення: 01.11.2023)

33. Microsoft Learn. SQL Server Profiler. URL: https://learn.microsoft.com/en-us/sql/tools/sql-server-profiler/sql-server-profiler?view=sql-server-ver16 (дата звернення: 13.11.2023).

ДОДАТКИ

Додаток А. Скрипт створення бази даних

```
CREATE DATABASE [Frameworks]
GO
USE [Frameworks]
GO
CREATE TABLE [dbo].[Departments](
    [Id] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY
CLUSTERED,
    [Name] [nvarchar](100) NOT NULL)
GO
CREATE TABLE [dbo].[Projects](
    [Id] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY
CLUSTERED,
    [Name] [nvarchar](100) NOT NULL)
GO
CREATE TABLE [dbo].[Employees](
    [Id] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY
CLUSTERED,
    [FirstName] [nvarchar](50) NOT NULL,
    [LastName] [nvarchar](50) NOT NULL,
    [Email] [nvarchar](max) NOT NULL,
    [BirthDate] [datetime] NOT NULL,
    [DepartmentId] [int] NOT NULL)
GO
ALTER TABLE [dbo].[Employees]  WITH CHECK ADD
CONSTRAINT [FK_Employees_Departments] FOREIGN
KEY([DepartmentId])
REFERENCES [dbo].[Departments] ([Id])
GO
ALTER TABLE [dbo]. [Employees] CHECK CONSTRAINT
[FK_Employees_Departments]
GO
CREATE TABLE [dbo].[EmployeeProjects](
    [EmployeeId] [int] NOT NULL,
    [ProjectId] [int] NOT NULL,
PRIMARY KEY CLUSTERED
(
    [EmployeeId] ASC,
    [ProjectId] ASC
)) ON [PRIMARY]
GO
```

```
ALTER TABLE [dbo].[EmployeeProjects]  WITH CHECK ADD
CONSTRAINT [FK_EmployeeProjects_Employees] FOREIGN
KEY([EmployeeId])
REFERENCES [dbo].[Employees] ([Id])
GO
ALTER TABLE [dbo].[EmployeeProjects] CHECK CONSTRAINT
[FK_EmployeeProjects_Employees]
GO
ALTER TABLE [dbo].[EmployeeProjects]  WITH CHECK ADD
CONSTRAINT [FK_EmployeeProjects_Projects] FOREIGN
KEY([ProjectId])
REFERENCES [dbo].[Projects] ([Id])
GO
ALTER TABLE [dbo].[EmployeeProjects] CHECK CONSTRAINT
[FK_EmployeeProjects_Projects]
GO
CREATE TABLE [dbo].[EmployeeDetails](
    [Id] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY
CLUSTERED,
    [EmployeeId] [int] NOT NULL UNIQUE,
    [SpecialNeeds] [nvarchar](255) NULL)
GO
ALTER TABLE [dbo].[EmployeeDetails]  WITH CHECK ADD
CONSTRAINT [FK_EmployeeDetails_Employees] FOREIGN
KEY([EmployeeId])
REFERENCES [dbo].[Employees] ([Id])
GO
ALTER TABLE [dbo].[EmployeeDetails] CHECK CONSTRAINT
[FK_EmployeeDetails_Employees]
GO
CREATE TABLE [dbo].[LogEntries](
    [Id] [int] IDENTITY(1,1) NOT NULL PRIMARY KEY
CLUSTERED,
    [ORM] [nvarchar](20) NOT NULL,
    [ServiceRequest] [nvarchar](MAX) NULL,
    [MethodName] [nvarchar](100) NOT NULL,
    [RamUsage_MB] [bigint] NOT NULL,
    [CpuKernelUsage_µs] [bigint] NOT NULL,
    [CpuUserUsage_µs] [bigint] NOT NULL,
    [CpuTotalUsage_µs] [bigint] NOT NULL,
    [ThreadsAmount] [int] NOT NULL,
    [ServiceResult] [nvarchar](MAX) NULL)
GO
```

Декларація
академічної доброчесності
здобувача вищої освіти ЗНУ

Я Сєров Арсеній Леонідович, студент _2_курсу, форми здобуття освіти денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні ЗНУ, спеціальності 121 Інженерія програмного забезпечення, адреса електронної пошти se22m-22@stu.zsea.edu.ua

– підтверджую, що написана мною кваліфікаційна робота на тему **«Порівняльне дослідження фреймворків об'єктно-реляційного відображення для платформи Microsoft .NET»** відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст. 42 Закону України «Про освіту», зі змістом яких ознайомлений;

– заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

– згоден/згодна на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою Інтернет-системи, а також на архівування роботи в базі даних цієї системи.

Дата_30.11.2023_____ Підпис_____ А.Л. Сєров
(студент)

Дата__30.11.2023_____ Підпис_____ Г.П. Коломоєць
(науковий керівник)