МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Харківський національний університет імені В.Н. Каразіна

Факультет: **ННІ Каразінський банківський інститут**

Кафедра: **Інформаційних технологій та математичного моделювання**

Спеціальність: **125 Кібербезпека**

Освітня програма: **Кібербезпека у фінансових технологіях**

Група: АБ-41Б **денна форма навчання**

КВАЛІФІКАЦІЙНА БАКАЛАВРСЬКА РОБОТА

на тему:

ДОСЛІДЖЕННЯ ТЕХНОЛОГІЇ ENTITY FRAMEWORK CORE ДЛЯ ВЗАЄМОДІЇ З БАЗАМИ ДАНИХ POSTGRESQL

ЗА НАКАЗОМ № 4601-5/335 ВІД 07 ЛЮТОГО 2025 РОКУ

Здобувача вищої освіти **Проніна Владислава Івановича**

Робота допущена до захисту в ЕК
протокол кафедри ІТММ № 13 від 31.05.2025р.

Завідувач кафедри ІТММ
к.п.н., доцент
_____ Н.І. Стяглик

Науковий керівник
к.т.н.
_____ А.В. Рогов

м. Харків 2025 р.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет імені В. Н. Каразіна

Факультет навчально-науковий інститут "Каразінський банківський інститут"
Кафедра інформаційних технологій та математичного моделювання
Рівень вищої освіти перший (бакалаврський)
Спеціальність 125 Кібербезпека
Освітня програма Кібербезпека у фінансових технологіях

ЗАТВЕРДЖУЮ

Завідувач кафедри

_____ **Н. І. Стяглик**
Підпис ініціали, прізвище

"08" лютого 2025 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ (ПРОЄКТ)

Проніну Владиславу Івановичу
(прізвище, ім'я, по батькові студента)

1. Тема роботи: Дослідження технології Entity Framework Core для взаємодії з базами даних PostgreSQL.

керівник роботи Рогов Андрій Володимирович, к.т.н.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом по університету від "08" лютого 2025року № 4601-5/335

2. Строк подання студентом роботи 15 травня 2025 року

3. Перелік питань, які потрібно розробити:

У розділі 1: Провести аналіз існуючих ORM, зокрема Entity Framework Core та їх спільну роботу з PostgreSQL.

У розділі 2: Розробити навчальний додаток з використанням EntityFrameworkCore та PostgreSQL.

У розділі 3: Порівняти процес розробки додатків з використанням ORM та альтернативних методів роботи з базами даних.

4.План роботи

№ з/п	Назви етапів роботи
1	Вибір здобувачем теми кваліфікаційної бакалаврської роботи
2	Затвердження плану і завдання кваліфікаційної бакалаврської роботи
3	Здача кваліфікаційної бакалаврської роботи керівнику
4	Підпис кваліфікаційної бакалаврської роботи керівника
5	Підпис кваліфікаційної бакалаврської роботи у нормоконтролера
6	Допуск завідувачем кафедри до захисту кваліфікаційної бакалаврської роботи
7	Захист кваліфікаційної бакалаврської роботи

Дата видачі завдання 08 лютого 2025 року

Студент _____ Пронін В.І.
підпис ініціали, прізвище

Керівник роботи _____ Рогов А.В.
підпис ініціали, прізвище

РЕФЕРАТ

НА КВАЛІФІКАЦІЙНУ БАКАЛАВРСЬКУ РОБОТУ

«ДОСЛІДЖЕННЯ ТЕХНОЛОГІЇ ENTITY FRAMEWORK CORE ДЛЯ ВЗАЄМОДІЇ З БАЗАМИ ДАНИХ POSTGRESQL»

Проніна Владислава Івановича

Кваліфікаційна бакалаврська робота містить: 63 сторінки, 1 таблицю, список літератури із 29 найменувань.

Об’єктом дослідження є процес взаємодії програмного забезпечення з базами даних за допомогою технології об’єктно-реляційного відображення.

Предметом дослідження виступає технологія Entity Framework Core у поєднанні з базою даних PostgreSQL.

Мета роботи полягає у дослідженні технології Entity Framework Core для взаємодії з базами даних PostgreSQL, аналізі її можливостей, обмежень та практичній реалізації в контексті створення ефективних програмних рішень.

Для досягнення цієї мети визначено такі завдання:

1. Провести аналіз основних принципів роботи Entity Framework Core та особливостей бази даних PostgreSQL, включаючи порівняння з іншими ORM-технологіями.

2. Розробити практичну реалізацію взаємодії Entity Framework Core з PostgreSQL, включаючи налаштування середовища, створення моделей даних і виконання основних операцій.

3. Оцінити продуктивність і ефективність використання Entity Framework Core з PostgreSQL, визначити обмеження та перспективи подальшого розвитку технології.

Актуальність теми бакалаврської роботи зумовлена широким використанням Entity Framework Core та PostgreSQL у сфері розробки програмного забезпечення. Ці технології є популярними серед розробників завдяки їхній універсальності, підтримці сучасних стандартів і здатності працювати з великими обсягами даних.

За результатами дослідження сформовано практичні рекомендації щодо оптимізації запитів і налаштування середовища, а також аналізі перспектив застосування цих технологій у реальних проєктах.

Практичне значення роботи полягає у створенні прикладів реалізації, які можуть бути використані розробниками для швидкого старту роботи з Entity Framework Core та PostgreSQL, а також у формулюванні рекомендацій щодо підвищення продуктивності та уникнення типових помилок. Отримані результати можуть бути застосовані в освітніх програмах, комерційних проєктах і подальших наукових дослідженнях.

Одержані результати можуть бути використані у різних сферах. У сфері освіти під час вивчення сучасних технологій роботи з базами даних, у комерційних проєктах результати дослідження можуть бути використані для прискорення процесу розробки програмного забезпечення.

КЛЮЧОВІ СЛОВА: ENTITY FRAMEWORK CORE, POSTGRESQL, ORM-ТЕХНОЛОГІЇ, ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ, МОДЕЛІ ДАНИХ, ПРОДУКТИВНІСТЬ, ОПТИМІЗАЦІЯ, РОЗРОБНИКИ.

ABSTRACT

AT QUALIFICATION BACHELOR WORK

"RESEARCH OF ENTITY FRAMEWORK CORE TECHNOLOGY FOR INTERACTION WITH POSTGRESQL DATABASES"

Pronin Vladyslav

Qualification bachelor's thesis contains: 63 pages, 1 table, a list of references of 29 titles.

The object of the study is the process of interaction of software with databases using object-relational mapping technology.

The subject of the study is the Entity Framework Core technology in combination with the PostgreSQL database.

The purpose of the work is to study the Entity Framework Core technology for interaction with PostgreSQL databases, analyze its capabilities, limitations and practical implementation in the context of creating effective software solutions.

To achieve this goal, the following tasks have been defined:

1. To analyze the basic principles of the Entity Framework Core operation and the features of the PostgreSQL database, including comparison with other ORM technologies.

2. Develop a practical implementation of the interaction of Entity Framework Core with PostgreSQL, including setting up the environment, creating data models and performing basic operations.

3. Assess the performance and efficiency of using Entity Framework Core with PostgreSQL, identify limitations and prospects for further development of the technology.

The relevance of the topic of the bachelor's thesis is due to the widespread use of Entity Framework Core and PostgreSQL in the field of software development. These technologies are popular among developers due to their versatility, support for modern standards and the ability to work with large amounts of data.

Based on the results of the study, practical recommendations were formed for optimizing queries and setting up the environment, as well as analyzing the prospects for applying these technologies in real projects.

The practical significance of the work lies in creating implementation examples that can be used by developers to quickly start working with Entity Framework Core and PostgreSQL, as well as in formulating recommendations for increasing productivity and avoiding typical errors. The results obtained can be applied in educational programs, commercial projects and further scientific research.

The results obtained can be used in various areas. In the field of education, when studying modern database technologies, in commercial projects, the results of the research can be used to accelerate the software development process.

KEYWORDS: ENTITY FRAMEWORK CORE, POSTGRESQL, ORM TECHNOLOGIES, SOFTWARE, DATA MODELS, PRODUCTIVITY, OPTIMIZATION, DEVELOPERS.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ, УМОВНИХ ПОЗНАЧОК, СИМВОЛІВ І ТЕРМІНІВ

EF Core – Entity Framework Core, технологія об’єктно-реляційного відображення для .NET.

ORM – Object-Relational Mapping, об’єктно-реляційне відображення.

PostgreSQL – об’єктно-реляційна система управління базами даних із відкритим кодом.

СУБД – Система управління базами даних.

Npgsql – провайдер для інтеграції EF Core із PostgreSQL.

LINQ – Language Integrated Query, інтегрований у мову запитів.

CRUD – Create, Read, Update, Delete, основні операції з даними.

JSONB – бінарний формат JSON у PostgreSQL для зберігання неструктурованих даних.

PostGIS – розширення PostgreSQL для роботи з геопросторовими даними.

API – Application Programming Interface, програмний інтерфейс додатка.

ASP.NET Core – кросплатформний фреймворк для створення вебдодатків.

DbContext – клас у EF Core для управління контекстом даних.

SQL – Structured Query Language, мова структурованих запитів.

ACID – Atomicity, Consistency, Isolation, Durability, властивості транзакцій.

NuGet – менеджер пакетів для .NET.

JSON – JavaScript Object Notation, формат обміну даними.

ВСТУП

Сучасний розвиток інформаційних технологій супроводжується стрімким зростанням обсягів даних, що потребують ефективного зберігання, обробки та управління. Бази даних стали невід'ємною частиною програмного забезпечення, забезпечуючи надійне збереження інформації та швидкий доступ до неї. У цьому контексті технології для взаємодії з базами даних відіграють ключову роль, дозволяючи розробникам спрощувати процеси створення, оновлення та отримання даних. Однією з таких технологій є Entity Framework Core — сучасний інструмент для об'єктно-реляційного відображення (ORM), який забезпечує зручну інтеграцію між об'єктно-орієнтованим програмуванням і реляційними базами даних. У поєднанні з потужною реляційною базою даних PostgreSQL, яка вирізняється своєю продуктивністю, гнучкістю та підтримкою широкого спектра функціоналу, Entity Framework Core відкриває нові можливості для розробки ефективних програмних рішень.

Актуальність теми бакалаврської роботи зумовлена широким використанням Entity Framework Core та PostgreSQL у сфері розробки програмного забезпечення. Ці технології є популярними серед розробників завдяки їхній універсальності, підтримці сучасних стандартів і здатності працювати з великими обсягами даних. Водночас, незважаючи на значну кількість документації та практичних прикладів, питання оптимального налаштування, продуктивності та вирішення специфічних проблем при взаємодії Entity Framework Core з PostgreSQL залишаються недостатньо дослідженими в україномовному науковому просторі. Таким чином, аналіз і практична реалізація цієї взаємодії є важливим завданням, що має як теоретичне, так і практичне значення.

Мета роботи полягає у дослідженні технології Entity Framework Core для взаємодії з базами даних PostgreSQL, аналізі її можливостей, обмежень та

практичній реалізації в контексті створення ефективних програмних рішень. Для досягнення цієї мети визначено такі завдання:

4. Провести аналіз основних принципів роботи Entity Framework Core та особливостей бази даних PostgreSQL, включаючи порівняння з іншими ORM-технологіями.
5. Розробити практичну реалізацію взаємодії Entity Framework Core з PostgreSQL, включаючи налаштування середовища, створення моделей даних і виконання основних операцій.
6. Оцінити продуктивність і ефективність використання Entity Framework Core з PostgreSQL, визначити обмеження та перспективи подальшого розвитку технології.

Об’єктом дослідження є процес взаємодії програмного забезпечення з базами даних за допомогою технології об’єктно-реляційного відображення. Предметом дослідження виступає технологія Entity Framework Core у поєднанні з базою даних PostgreSQL.

Методи дослідження включають аналіз наукової та технічної літератури, порівняльний аналіз технологій, експериментальну розробку програмного забезпечення, тестування продуктивності та узагальнення отриманих результатів. У роботі використано офіційну документацію Microsoft, Npgsql, PostgreSQL, а також наукові статті, навчальні посібники та практичні приклади від українських і зарубіжних авторів.

Наукова новизна роботи полягає у систематизації знань про використання Entity Framework Core з PostgreSQL, розробці практичних рекомендацій щодо оптимізації запитів і налаштування середовища, а також аналізі перспектив застосування цих технологій у реальних проєктах.

Практичне значення роботи полягає у створенні прикладів реалізації, які можуть бути використані розробниками для швидкого старту роботи з Entity Framework Core та PostgreSQL, а також у формулюванні рекомендацій щодо підвищення продуктивності та уникнення типових помилок. Отримані

результати можуть бути застосовані в освітніх програмах, комерційних проєктах і подальших наукових дослідженнях.

Структура роботи складається зі вступу, трьох основних розділів, висновків та списку використаних джерел. У першому розділі проведено аналіз технології Entity Framework Core та бази даних PostgreSQL. Другий розділ присвячено практичній реалізації взаємодії цих технологій. У третьому розділі оцінено ефективність і перспективи використання Entity Framework Core з PostgreSQL. Робота завершується узагальненням результатів і формулюванням висновків.

Таким чином, дана бакалаврська робота спрямована на поглиблення розуміння технології Entity Framework Core у контексті її взаємодії з PostgreSQL, що сприятиме розвитку компетенцій у сфері розробки програмного забезпечення та підвищенню якості створюваних інформаційних систем.

РОЗДІЛ 1
АНАЛІЗ ТЕХНОЛОГІЇ ENTITY FRAMEWORK CORE ТА БАЗ ДАНИХ POSTGRESQL

1.1. Огляд технології Entity Framework Core: основні принципи та функціональність

Entity Framework Core (EF Core) є сучасною технологією об’єктно-реляційного відображення (ORM), розробленою компанією Microsoft для спрощення взаємодії між об’єктно-орієнтованим програмуванням і реляційними базами даних. Вона дозволяє розробникам працювати з даними у вигляді об’єктів .NET, абстрагуючись від складнощів написання SQL-запитів і управління з’єднаннями з базою даних [1]. EF Core є кросплатформним рішенням, яке підтримує широкий спектр баз даних, таких як PostgreSQL, SQL Server, MySQL та SQLite, що робить його універсальним інструментом для створення сучасних програмних рішень.

Основним принципом роботи EF Core є концепція об’єктно-реляційного відображення, яка передбачає зіставлення класів .NET (сутностей) з таблицями бази даних, а їхніх властивостей — з відповідними стовпцями. Цей підхід дозволяє розробникам оперувати даними через об’єкти, використовуючи знайомий синтаксис C#, замість написання складних SQL-запитів. EF Core автоматично генерує SQL-запити на основі операцій, виконаних над об’єктами, що значно прискорює розробку та зменшує ймовірність помилок [2]. Наприклад, створення, оновлення чи видалення запису в базі даних може бути виконано за допомогою простих методів, таких як Add, Update або Remove, які застосовуються до контексту даних.

Функціональність EF Core базується на трьох основних компонентах: моделі даних, контексті даних і провайдерах баз даних. Модель даних формується за допомогою класів C#, які описують сутності та їхні зв’язки. Ці класи можуть бути створені вручну або згенеровані автоматично на основі

існуючої бази даних (підхід Database-First) [3]. Контекст даних, реалізований через клас, похідний від DbContext, є центральним елементом, що забезпечує взаємодію між об’єктами .NET і базою даних. Він відповідає за відстеження змін у сутностях, формування SQL-запитів і управління транзакціями. Провайдери баз даних, такі як Npgsql для PostgreSQL, дозволяють EF Core адаптуватися до специфіки конкретної СУБД, забезпечуючи коректну генерацію запитів і підтримку її унікальних функцій [4].

Однією з ключових особливостей EF Core є підтримка LINQ (Language Integrated Query), яка дозволяє створювати запити до бази даних у вигляді виразів C#. LINQ-запити є типобезпечними, що зменшує ризик помилок під час виконання, а також полегшує рефакторинг коду. Наприклад, запит для отримання всіх записів із таблиці може виглядати як context.Users.Where(u => u.Age > 18).ToList(), що автоматично транслюється в SQL [1]. Крім того, EF Core підтримує асинхронне виконання запитів, що підвищує продуктивність програм, особливо в умовах високого навантаження.

EF Core також пропонує механізми міграцій, які дозволяють автоматизувати створення та оновлення схеми бази даних на основі змін у моделі даних (підхід Code-First). Це значно спрощує процес розгортання програмного забезпечення, оскільки розробнику не потрібно вручну змінювати структуру бази даних [2]. Наприклад, додавання нової властивості до класу сутності автоматично може бути відображено у вигляді нового стовпця в базі даних після виконання відповідної міграції.

Ще однією важливою особливістю є підтримка складних зв’язків між сутностями, таких як один-до-одного, один-до-багатьох і багато-до-багатьох. EF Core автоматично обробляє ці зв’язки, дозволяючи розробникам легко навігувати між пов’язаними об’єктами. Наприклад, доступ до списку замовлень клієнта може бути виконано через властивість навігації, таку як customer.Orders, без необхідності писати JOIN-запити вручну [3]. Крім того, EF Core підтримує механізми ледачого (lazy loading) та активного (eager

loading) завантаження даних, що дозволяє оптимізувати запити залежно від потреб програми.

EF Core вирізняється своєю модульністю та гнучкістю. На відміну від попередньої версії Entity Framework, EF Core є легковаговим і дозволяє підключати лише необхідні компоненти, що зменшує накладні витрати на продуктивність. Водночас технологія активно розвивається, і нові версії, такі як EF Core 7 і 8, додають підтримку сучасних функцій, наприклад, роботу з JSON-полями чи покращену обробку складних SQL-запитів [4]. Це робить EF Core привабливим вибором для проєктів різного масштабу — від невеликих стартапів до великих корпоративних систем.

Порівняно з іншими ORM-технологіями, такими як Dapper чи NHibernate, EF Core займає проміжну позицію між простотою використання та продуктивністю. Dapper, наприклад, є швидшим за рахунок мінімальної абстракції, але вимагає написання SQL-запитів вручну, що може ускладнити розробку складних систем [5]. EF Core, своєю чергою, забезпечує баланс між зручністю та можливістю тонкого налаштування запитів, що робить його популярним серед розробників.

Незважаючи на численні переваги, EF Core має певні обмеження. Наприклад, автоматична генерація SQL-запитів може призводити до неоптимальних запитів у складних сценаріях, що вимагає від розробника знань для їхньої оптимізації. Крім того, підтримка специфічних функцій баз даних, таких як розширені можливості PostgreSQL (наприклад, PostGIS), залежить від якості провайдера [4]. Проте ці недоліки компенсуються активною спільнотою та регулярними оновленнями.

Entity Framework Core є потужним і гнучким інструментом для роботи з реляційними базами даних, який поєднує простоту використання з широкими можливостями налаштування. Його основні принципи, такі як об’єктно-реляційне відображення, підтримка LINQ і механізми міграцій, роблять його ефективним рішенням для сучасної розробки програмного забезпечення.

1.2. Характеристика бази даних PostgreSQL: особливості та переваги

PostgreSQL є однією з найпопулярніших об’єктно-реляційних систем управління базами даних (СУБД) з відкритим вихідним кодом, яка вирізняється високою надійністю, гнучкістю та широким функціоналом. Розроблена в 1986 році в Університеті Каліфорнії в Берклі, вона еволюціонувала в потужну платформу, що підтримує як традиційні реляційні, так і сучасні нереляційні можливості [3]. PostgreSQL часто називають "найпросунутішою СУБД з відкритим кодом" завдяки її відповідності стандартам SQL, розширюваності та активній підтримці спільнотою розробників.

Основною особливістю PostgreSQL є її об’єктно-реляційна архітектура, яка поєднує принципи реляційних баз даних із можливостями об’єктно-орієнтованого програмування. СУБД підтримує створення користувацьких типів даних, функцій і навіть розширень, що дозволяють адаптувати її до специфічних потреб проєкту [8]. Наприклад, розробники можуть створювати власні типи даних для роботи з географічними координатами чи складними структурами, що робить PostgreSQL універсальним інструментом для різноманітних застосувань.

Однією з ключових переваг PostgreSQL є її повна відповідність стандарту SQL:1999 і часткова підтримка нових стандартів, таких як SQL:2016. Це забезпечує сумісність із більшістю інструментів і фреймворків, включаючи Entity Framework Core через провайдер Npgsql [4]. Крім того, PostgreSQL підтримує розширені можливості SQL, такі як віконні функції, рекурсивні запити та повнотекстовий пошук, що дозволяє ефективно обробляти складні аналітичні запити без додаткових інструментів.

PostgreSQL вирізняється високою продуктивністю та масштабованістю. СУБД підтримує паралельне виконання запитів, що дозволяє оптимально використовувати багатоядерні процесори, а також забезпечує ефективну обробку великих обсягів даних [9]. Завдяки вбудованим механізмам індексації,

таким як B-дерева, GiST, GIN і BRIN, PostgreSQL забезпечує швидкий доступ до даних навіть у таблицях із мільйонами записів. Наприклад, індекс GIN широко використовується для повнотекстового пошуку та роботи з масивами, що є унікальною особливістю для реляційних баз даних.

Ще однією важливою перевагою є підтримка роботи з неструктурованими даними. PostgreSQL пропонує вбудовані типи даних JSON і JSONB, які дозволяють зберігати та обробляти документо-орієнтовані дані в реляційній базі [10]. Тип JSONB, зокрема, забезпечує ефективне зберігання та індексацію JSON-документів, що робить PostgreSQL конкурентоспроможною альтернативою NoSQL-системам, таким як MongoDB, у гібридних сценаріях використання.

PostgreSQL також славиться своєю надійністю та безпекою. СУБД використовує механізм транзакцій із гарантією ACID (атомарність, узгодженість, ізоляція, довговічність), що забезпечує цілісність даних навіть у разі збоїв [3]. Крім того, PostgreSQL підтримує розширені можливості управління доступом, включаючи ролі, привілеї та автентифікацію через LDAP чи Kerberos, що робить її придатною для корпоративних систем із високими вимогами до безпеки.

Гнучкість PostgreSQL проявляється в її розширюваності. СУБД дозволяє створювати власні функції на мовах програмування, таких як PL/pgSQL, PL/Python чи PL/Perl, а також підключати розширення, наприклад, PostGIS для геопросторових даних або TimescaleDB для роботи з часовими рядами [8]. Це робить PostgreSQL привабливою для спеціалізованих проєктів, де потрібна підтримка нестандартних типів даних чи алгоритмів.

Серед інших переваг PostgreSQL варто відзначити її кросплатформність і активну спільноту. СУБД працює на всіх основних операційних системах, включаючи Windows, Linux і macOS, що забезпечує легке розгортання в різних середовищах [9]. Активна спільнота розробників регулярно випускає оновлення, виправляє помилки та створює нові розширення, що сприяє довгостроковій підтримці технології.

Порівняно з іншими СУБД, такими як MySQL чи SQL Server, PostgreSQL пропонує ширший набір функцій "з коробки", особливо для складних аналітичних задач і роботи з неструктурованими даними [10]. Наприклад, MySQL поступається PostgreSQL у підтримці віконних функцій і складних типів даних, тоді як SQL Server, хоча й потужний, є пропрієтарним і менш гнучким у плані розширюваності.

Незважаючи на численні переваги, PostgreSQL має певні обмеження. Наприклад, її конфігурація для оптимальної продуктивності може бути складною для початківців, а використання розширених функцій, таких як JSONB чи PostGIS, вимагає додаткових знань [4]. Крім того, у порівнянні з легковаговими СУБД, такими як SQLite, PostgreSQL потребує більше ресурсів для розгортання.

PostgreSQL є потужною, гнучкою та надійною СУБД, яка поєднує можливості реляційних і нереляційних баз даних. Її підтримка стандартів SQL, розширюваність, висока продуктивність і безпека роблять її ідеальним вибором для проєктів різного масштабу, особливо в поєднанні з Entity Framework Core.

1.3. Порівняння Entity Framework Core з іншими ORM-технологіями

Об’єктно-реляційні відображення (ORM) є важливим інструментом у сучасній розробці програмного забезпечення, оскільки вони спрощують взаємодію між об’єктно-орієнтованим кодом і реляційними базами даних. Entity Framework Core (EF Core) є однією з найпопулярніших ORM-технологій для платформи .NET, але на ринку існує низка альтернатив, таких як Dapper, NHibernate та ADO.NET (хоча останній не є ORM у класичному розумінні). Порівняння цих технологій дозволяє визначити їхні сильні та слабкі сторони, а також оцінити доцільність використання EF Core у проєктах із PostgreSQL [5].

Entity Framework Core є повноцінною ORM-технологією, яка забезпечує високий рівень абстракції при роботі з базами даних. Вона підтримує автоматичне зіставлення об’єктів .NET із таблицями бази даних, генерацію SQL-запитів через LINQ, механізми міграцій і відстеження змін сутностей. EF Core вирізняється простотою використання, що робить її привабливою для розробників, які прагнуть швидко створювати прототипи чи працювати зі складними моделями даних [1]. Завдяки підтримці асинхронних операцій і кросплатформності, EF Core ефективно інтегрується з PostgreSQL через провайдер Npgsql, забезпечуючи сумісність із розширеними функціями СУБД, такими як JSONB чи повнотекстовий пошук [4].

Dapper, на відміну від EF Core, є мікро-ORM, що фокусується на швидкості виконання та мінімальній абстракції. Dapper не генерує SQL-запити автоматично, а вимагає від розробника написання SQL-коду вручну, що забезпечує більший контроль над продуктивністю запитів [19]. Ця технологія є значно швидшою за EF Core у сценаріях із простими запитами, оскільки вона не має накладних витрат на відстеження змін чи складну трансляцію LINQ у SQL. Проте Dapper менш зручний для роботи зі складними моделями даних або великими проєктами, оскільки не підтримує міграції чи автоматичне управління зв’язками між сутностями [22]. Наприклад, для реалізації зв’язку "один-до-багатьох" у Dapper потрібно писати JOIN-запити вручну, тоді як EF Core обробляє це автоматично.

NHibernate є іншою повноцінною ORM-технологією, яка має довгу історію використання в .NET-екосистемі. NHibernate пропонує гнучке налаштування відображень через XML-файли або код, а також підтримує складні сценарії, такі як власні стратегії кешування чи специфічні діалекти SQL [20]. Порівняно з EF Core, NHibernate є більш складним у налаштуванні та має крутішу криву навчання, що може бути недоліком для невеликих проєктів. Крім того, NHibernate менш оптимізований для кросплатформних застосунків, тоді як EF Core розроблений із урахуванням .NET Core і є легшим у розгортанні [13]. Однак NHibernate може переважати в проєктах із

нестандартними вимогами до бази даних, де потрібне тонке налаштування запитів.

ADO.NET, хоча й не є ORM, часто розглядається як альтернатива для низькорівневого доступу до баз даних. ADO.NET забезпечує прямий доступ до бази даних через SQL-запити, що дає максимальну продуктивність і контроль [14]. Проте ця технологія вимагає від розробника самостійного управління з'єднаннями, параметрами запитів і мапінгом даних на об'єкти .NET, що значно збільшує обсяг коду та ймовірність помилок. На відміну від EF Core, ADO.NET не пропонує готових інструментів для роботи зі зв'язками чи міграціями, що робить її менш зручною для сучасних проєктів [5].

Порівняльний аналіз цих технологій за ключовими критеріями показує їхні основні відмінності. Продуктивність: Dapper і ADO.NET значно швидші за EF Core у простих сценаріях завдяки мінімальній абстракції, тоді як EF Core і NHibernate можуть втрачати продуктивність через автоматичну генерацію запитів [19]. Простота використання: EF Core є найбільш інтуїтивною технологією завдяки LINQ і механізмам міграцій, тоді як Dapper і ADO.NET вимагають глибшого розуміння SQL, а NHibernate — додаткового налаштування [1]. Гнучкість: NHibernate пропонує найбільше можливостей для кастомізації, тоді як EF Core балансує між гнучкістю та простотою, а Dapper і ADO.NET є менш адаптивними до складних моделей даних [20]. Підтримка PostgreSQL: EF Core має офіційний провайдер Npgsql, який забезпечує повну сумісність із PostgreSQL, включаючи підтримку специфічних типів даних, таких як JSONB [4]. Dapper також добре працює з PostgreSQL, але вимагає ручного написання запитів. NHibernate підтримує PostgreSQL, але потребує додаткових налаштувань, а ADO.NET залежить від драйверів, таких як Npgsql, без ORM-функціоналу [14].

Серед переваг EF Core варто відзначити її активний розвиток і підтримку Microsoft, що забезпечує регулярні оновлення та інтеграцію з новими версіями .NET [13]. Наприклад, EF Core 7 і 8 додали підтримку JSON-мапінгу та покращили продуктивність запитів, що робить її конкурентоспроможною

навіть у порівнянні з Dapper у певних сценаріях. Водночас Dapper залишається кращим вибором для високопродуктивних систем із простими запитами, а NHibernate — для проєктів із нестандартними вимогами до бази даних [22].

Обмеження EF Core включають потенційно неоптимальні SQL-запити у складних сценаріях, що вимагає від розробника знань для їхньої оптимізації [19]. Dapper, своєю чергою, ускладнює підтримку великих проєктів через відсутність автоматизації, а NHibernate може бути надмірно складним для невеликих застосунків [20]. ADO.NET, хоча й швидкий, є застарілим для більшості сучасних проєктів через високі витрати на розробку [14].

EF Core займає проміжну позицію між продуктивністю та зручністю, що робить її оптимальним вибором для більшості .NET-проєктів із PostgreSQL. Вона перевершує Dapper у зручності, NHibernate — у простоті налаштування, а ADO.NET — у автоматизації роботи з даними, що підтверджує її універсальність і популярність.

1.4. Аналіз сценаріїв використання Entity Framework Core з PostgreSQL

Entity Framework Core (EF Core) у поєднанні з PostgreSQL є потужним рішенням для розробки програмного забезпечення, яке застосовується в широкому спектрі сценаріїв завдяки гнучкості ORM-технології та розширеним можливостям СУБД. Аналіз типових сценаріїв використання дозволяє оцінити ефективність цієї комбінації в різних типах проєктів, від невеликих вебдодатків до складних корпоративних систем. У цьому розділі розглянуто основні сценарії використання EF Core з PostgreSQL, їхні особливості та переваги [7].

1. Розробка вебдодатків із реляційними даними. EF Core широко застосовується для створення вебдодатків на платформі .NET, таких як ASP.NET Core, де PostgreSQL виступає основною базою даних. У цьому сценарії EF Core спрощує роботу з реляційними даними, дозволяючи розробникам створювати моделі даних за допомогою класів C# і виконувати

CRUD-операції (створення, читання, оновлення, видалення) через LINQ-запити [11]. Наприклад, у системі управління контентом (CMS) EF Core може використовуватися для роботи з таблицями статей, користувачів і коментарів, автоматично обробляючи зв’язки між ними. PostgreSQL, своєю чергою, забезпечує швидкий доступ до даних і підтримує складні запити, що є важливим для динамічних вебдодатків. Перевагою цього сценарію є підтримка асинхронних операцій EF Core, що підвищує продуктивність вебдодатків при високому навантаженні [16].

2. Робота з гібридними даними (реляційні та нереляційні). PostgreSQL вирізняється підтримкою типів даних JSON і JSONB, що дозволяє зберігати та обробляти неструктуровані дані в реляційній базі. EF Core, починаючи з версії 7, підтримує мапінг JSON-полів на об’єкти .NET, що робить цю комбінацію ідеальною для гібридних сценаріїв [12]. Наприклад, у системі електронної комерції JSONB може використовуватися для зберігання атрибутів товарів (розмір, колір тощо), які варіюються залежно від категорії. EF Core дозволяє запитувати ці дані через LINQ, забезпечуючи зручний доступ до неструктурованих даних без втрати переваг реляційної моделі. Цей сценарій особливо корисний для проєктів, де потрібна гнучкість NoSQL-систем, але з вимогами до транзакційної цілісності [17].

3. Геопросторові додатки з використанням PostGIS. PostgreSQL підтримує розширення PostGIS, яке додає функціонал для роботи з геопросторовими даними, такими як координати, маршрути чи полігони. EF Core через провайдер Npgsql забезпечує інтеграцію з PostGIS, дозволяючи розробникам створювати моделі для географічних об’єктів і виконувати просторові запити [24]. Наприклад, у системі логістики EF Core може використовуватися для управління даними про розташування транспортних засобів, а PostgreSQL із PostGIS — для обчислення відстаней чи пошуку об’єктів у заданому радіусі. Цей сценарій демонструє здатність комбінації EF Core і PostgreSQL обробляти спеціалізовані задачі, які виходять за рамки стандартних реляційних операцій [18].

4. Аналітичні системи з великими обсягами даних. PostgreSQL підтримує розширені аналітичні функції, такі як віконні функції, агрегатні запити та паралельне виконання, що робить її придатною для аналітичних систем. EF Core дозволяє створювати запити до таких даних через LINQ, хоча для складних аналітичних операцій може знадобитися використання сирих SQL-запитів через метод FromSql [15]. Наприклад, у системі бізнес-аналітики EF Core може використовуватися для отримання даних про продажі, а PostgreSQL — для виконання складних групувань і обчислень. Цей сценарій підкреслює важливість правильної оптимізації запитів у EF Core, оскільки автоматично згенеровані запити можуть бути менш ефективними для великих обсягів даних [21].

5. Прототипування та стартапи. Завдяки механізмам міграцій і підходу Code-First, EF Core ідеально підходить для швидкого прототипування та розробки стартапів. Розробники можуть створювати моделі даних у коді, а EF Core автоматично генерує схему бази даних у PostgreSQL [7]. Цей сценарій є популярним у невеликих проєктах, де потрібна швидка ітерація та мінімальні витрати на налаштування. PostgreSQL, своєю чергою, забезпечує надійність і масштабованість, що дозволяє стартапам переходити від прототипу до повноцінного продукту без зміни СУБД [11]. Наприклад, стартап із розробки мобільного додатка може використовувати EF Core для управління даними користувачів і PostgreSQL для їхнього зберігання, швидко адаптуючись до змін у моделі даних.

Кожен із цих сценаріїв має свої особливості. У вебдодатках і прототипуванні EF Core виграє завдяки простоті та автоматизації, тоді

1.5. Висновки до розділу

Проведений аналіз технології Entity Framework Core та бази даних PostgreSQL дозволяє сформулювати ключові висновки щодо їхніх можливостей і потенціалу для розробки програмного забезпечення. Entity

Framework Core є потужною ORM-технологією, яка спрощує взаємодію з базами даних завдяки об’єктно-реляційному відображенню, підтримці LINQ, механізмам міграцій і асинхронним операціям. Її універсальність і кросплатформність роблять її ефективним інструментом для роботи з PostgreSQL, зокрема через провайдер Npgsql, який забезпечує сумісність із розширеними функціями СУБД [6].

PostgreSQL, своєю чергою, є однією з найпросунутіших СУБД із відкритим кодом, що вирізняється підтримкою реляційних і нереляційних даних, розширюваністю та високою продуктивністю. Її можливості, такі як JSONB, PostGIS і віконні функції, значно розширюють сценарії використання, від вебдодатків до аналітичних систем [23].

Порівняння EF Core з іншими ORM-технологіями, такими як Dapper і NHibernate, показало, що EF Core займає оптимальну позицію між простотою використання та гнучкістю, хоча поступається Dapper у продуктивності для простих запитів і NHibernate у кастомізації [22]. Аналіз сценаріїв використання EF Core з PostgreSQL підтвердив їхню ефективність у вебдодатках, гібридних системах, геопросторових проєктах і аналітичних задачах, за умови правильної оптимізації запитів і налаштування [25].

Комбінація EF Core і PostgreSQL є універсальним рішенням для сучасних проєктів, що поєднує зручність розробки з потужними можливостями обробки даних. Подальше дослідження їхньої взаємодії в практичних задачах дозволить поглибити розуміння оптимальних підходів до реалізації та оптимізації.

РОЗДІЛ 2
ПРАКТИЧНА РЕАЛІЗАЦІЯ ВЗАЄМОДІЇ ENTITY FRAMEWORK CORE З POSTGRESQL

2.1. Налаштування середовища для роботи з Entity Framework Core та PostgreSQL

Налаштування середовища для роботи з Entity Framework Core (EF Core) і PostgreSQL є першим і критично важливим етапом для забезпечення коректної взаємодії між програмою та базою даних. Цей процес включає встановлення необхідного програмного забезпечення, конфігурацію проєкту .NET, підключення провайдера Npgsql і налаштування з'єднання з PostgreSQL. Правильне виконання цих кроків гарантує стабільну роботу додатка та ефективне використання можливостей обох технологій [6].

Встановлення PostgreSQL. Для початку необхідно встановити сервер PostgreSQL на локальній машині або підключитися до віддаленого сервера. PostgreSQL доступний для всіх основних операційних систем, включаючи Windows, Linux і macOS, і може бути завантажений з офіційного сайту [3]. Під час встановлення користувач задає пароль для адміністратора (зазвичай роль postgres) і порт (за замовчуванням 5432). Для зручного управління базою даних рекомендується встановити інструмент pgAdmin, який надає графічний інтерфейс для створення баз і виконання запитів. Після встановлення створюється нова база даних, наприклад, MyAppDb, яка використовуватиметься в проєкті [8].

Налаштування проєкту .NET. Для розробки додатка з EF Core необхідно створити проєкт у середовищі .NET. Найпоширенішим вибором є ASP.NET Core для вебдодатків або консольний додаток для простих демонстрацій. У Visual Studio або Visual Studio Code створюється новий проєкт із шаблону, наприклад, ASP.NET Core Web API. EF Core підтримує .NET 6, 7 або 8, тому

важливо обрати сумісну версію фреймворку [1]. Для управління залежностями використовується NuGet, через який встановлюються необхідні пакети.

Встановлення пакетів NuGet. Для інтеграції EF Core з PostgreSQL необхідно встановити два основні пакети: Microsoft.EntityFrameworkCore і Npgsql.EntityFrameworkCore.PostgreSQL. Перший забезпечує основний функціонал EF Core, а другий є провайдером для PostgreSQL, який адаптує EF Core до особливостей СУБД [16]. Додатково рекомендується встановити пакет Microsoft.EntityFrameworkCore.Design для роботи з міграціями. Установка виконується через менеджер пакетів NuGet або команду в терміналі:

```
    dotnet add package Microsoft.EntityFrameworkCore --version 8.0.0
    dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL --version 8.0.0
    dotnet add package Microsoft.EntityFrameworkCore.Design --version 8.0.0
```

Ці пакети забезпечують підтримку LINQ, міграцій і специфічних типів даних PostgreSQL, таких як JSONB [24]. Для управління інструментами EF Core, такими як команда dotnet ef, необхідно встановити глобальний інструмент:

```
    dotnet tool install --global dotnet-ef
```

Налаштування рядка підключення. Рядок підключення (connection string) визначає параметри для доступу до бази даних PostgreSQL, включаючи адресу сервера, порт, назву бази, ім’я користувача та пароль. Типовий рядок підключення виглядає так:

```
    Host=localhost;Port=5432;Database=MyAppDb;Username=postgres;Password=mypassword
```

У проєкті ASP.NET Core рядок підключення зберігається в файлі конфігурації appsettings.json, щоб забезпечити безпечний доступ і легке оновлення:

```
{
  "ConnectionStrings": {
    "DefaultConnection":
"Host=localhost;Port=5432;Database=MyAppDb;Username=postgres;Pas
sword=mypassword"
  }
}
```

Для зчитування рядка підключення в коді використовується IConfiguration, що дозволяє динамічно отримувати параметри з конфігурації [12].

Налаштування DbContext. Клас DbContext є центральним елементом EF Core, який відповідає за взаємодію з базою даних. У проєкті створюється клас, наприклад, AppDbContext, що успадковується від DbContext. У конструкторі налаштовується провайдер PostgreSQL через метод UseNpgsql:

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public       AppDbContext(DbContextOptions<AppDbContext>
options) : base(options) { }
}
```

Для зчитування рядка підключення в коді використовується IConfiguration, що дозволяє динамічно отримувати параметри з конфігурації [12].

Налаштування DbContext. Клас DbContext є центральним елементом EF Core, який відповідає за взаємодію з базою даних. У проєкті створюється клас, наприклад, AppDbContext, що успадковується від DbContext. У конструкторі налаштовується провайдер PostgreSQL через метод UseNpgsql:

```
builder.Services.AddDbContext<AppDbContext>(options =>
options.UseNpgsql(builder.Configuration.GetConnectionString("Def
aultConnection")));
```

Це забезпечує інтеграцію EF Core з PostgreSQL і дозволяє використовувати контекст для виконання запитів [16].

Перевірка підключення. Для перевірки коректності налаштування створюється тестовий запит до бази даних. Наприклад, можна додати простий контролер у ASP.NET Core, який перевіряє доступність бази:

```
using Microsoft.AspNetCore.Mvc;

[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase
{
    private readonly AppDbContext _context;

    public TestController(AppDbContext context)
    {
        _context = context;
    }

    [HttpGet]
    public IActionResult CheckConnection()
    {
        try
        {
            _context.Database.EnsureCreated();
            return Ok("Connection successful");
        }
        catch (Exception ex)
        {
            return StatusCode(500, $"Connection failed:
{ex.Message}");
        }
    }
}
```

Цей код перевіряє, чи може EF Core підключитися до PostgreSQL і створити базу даних, якщо вона ще не існує [11].

Додаткові налаштування. Для підвищення безпеки рекомендується використовувати змінні середовища замість зберігання пароля в

appSettings.json. Наприклад, рядок підключення можна задати через змінну середовища DATABASE_URL. Крім того, для роботи з міграціями необхідно переконатися, що проєкт містить хоча б одну модель даних (сутність), яка буде використана для генерації схеми бази [24]. У разі використання специфічних функцій PostgreSQL, таких як JSONB чи PostGIS, потрібно додатково налаштувати провайдер Npgsql для їхньої підтримки [17].

Налаштування середовища завершується перевіркою працездатності всіх компонентів. Правильно налаштоване середовище забезпечує стабільну взаємодію EF Core з PostgreSQL, дозволяючи розробникам переходити до створення моделей даних і реалізації бізнес-логіки.

2.2. Створення моделі даних та їх відображення на базу даних

Створення моделі даних та їх відображення на базу даних є ключовим етапом у роботі з Entity Framework Core (EF Core) і PostgreSQL. Модель даних у EF Core представляє структуру даних програми у вигляді класів C#, які відображаються на таблиці бази даних. Цей процес включає створення сутностей, налаштування зв’язків між ними, конфігурацію відображень і застосування міграцій для створення або оновлення схеми бази даних. Використання підходу Code-First дозволяє автоматизувати цей процес, забезпечуючи гнучкість і зручність розробки [6].

Створення сутностей. Сутності є класами C#, які представляють об’єкти предметної області, наприклад, користувачів, товари чи замовлення. Кожна сутність відповідає таблиці в базі даних, а її властивості — стовпцям. Для демонстрації розглянемо приклад системи управління бібліотекою з двома сутностями: Book і Author. Код сутностей виглядає так:

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int PublicationYear { get; set; }
```

```
    public int AuthorId { get; set; }
    public Author Author { get; set; }
}

public class Author
{
    public int Id { get; set; }
    public string Name { get; set; }
    public List<Book> Books { get; set; }
}
```

У цьому прикладі властивість Id є первинним ключем, AuthorId — зовнішнім ключем, а Author і Books — навігаційними властивостями, що визначають зв’язок "один-до-багатьох" між автором і книгами. EF Core автоматично розпізнає первинні ключі за назвою (Id або <ІмʼяСутності>Id) і налаштовує їх як автоінкрементні [1].

Налаштування DbContext. Для роботи з сутностями необхідно оновити клас AppDbContext, додавши властивості DbSet, які представляють колекції сутностей:

```
using Microsoft.EntityFrameworkCore;

public class AppDbContext : DbContext
{
    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }

    public AppDbContext(DbContextOptions<AppDbContext> options) :
base(options) { }
}
```

DbSet дозволяє EF Core виконувати запити до відповідних таблиць і відстежувати зміни в об’єктах. Контекст також відповідає за конфігурацію зв’язків і відображення моделей на базу даних [16].

Конфігурація відображень. EF Core використовує конвенції для автоматичного зіставлення сутностей із таблицями, але в деяких випадках потрібне явне налаштування через Fluent API або анотації даних. Наприклад, для точного визначення імені таблиці чи обмежень можна перевизначити метод OnModelCreating у AppDbContext:

```
protected override void OnModelCreating(ModelBuilder
modelBuilder)
{
    modelBuilder.Entity<Book>()
        .ToTable("Books")
        .Property(b => b.Title)
        .HasMaxLength(200)
        .IsRequired();

    modelBuilder.Entity<Author>()
        .ToTable("Authors")
        .Property(a => a.Name)
        .HasMaxLength(100)
        .IsRequired();

    modelBuilder.Entity<Book>()
        .HasOne(b => b.Author)
        .WithMany(a => a.Books)
        .HasForeignKey(b => b.AuthorId);
}
```

Цей код задає імена таблиць, обмеження на довжину рядків і зв’язок "один-до-багатьох". Fluent API забезпечує більшу гнучкість порівняно з анотаціями, дозволяючи налаштувати складні відображення [12]. Наприклад, можна вказати підтримку специфічних типів PostgreSQL, таких як JSONB, для зберігання додаткових даних про книгу.

Створення міграцій. Після визначення сутностей і конфігурації необхідно створити схему бази даних у PostgreSQL за допомогою міграцій. Міграції дозволяють EF Core генерувати SQL-скрипти для створення або оновлення таблиць на основі моделей. Для створення першої міграції виконується команда в терміналі:

```
dotnet ef migrations add InitialCreate
```

Ця команда створює файл міграції в папці Migrations, який містить код для створення таблиць Books і Authors із відповідними стовпцями та зв’язками. Провайдер Npgsql забезпечує коректне відображення типів даних C# на типи PostgreSQL, наприклад, string на text або int на integer [24].

Застосування міграцій. Для створення таблиць у базі даних виконується команда:

```
dotnet ef database update
```

Ця команда застосовує всі міграції до бази даних MyAppDb, створюючи схему з таблицями, первинними та зовнішніми ключами. У разі помилок, наприклад, відсутності бази даних, EF Core видасть виняток, що допоможе ідентифікувати проблему [11]. Після виконання команди структура бази даних виглядатиме так:

— Таблиця Authors: Id (integer, primary key), Name (text, not null).
— Таблиця Books: Id (integer, primary key), Title (text, not null), PublicationYear (integer), AuthorId (integer, foreign key).

Перевірка відображення. Для перевірки коректності відображення можна виконати тестові операції, наприклад, додати дані через EF Core:

```
using (var context = new AppDbContext(options))
{
    var author = new Author { Name = "John Doe" };
    var book = new Book { Title = "Sample Book", PublicationYear
= 2023, Author = author };
    context.Authors.Add(author);
    context.Books.Add(book);
    context.SaveChanges();
}
```

Цей код додає автора та книгу до бази даних, а EF Core автоматично встановлює зв’язок через AuthorId. Перевірка даних у pgAdmin або через SQL-запит підтвердить, що таблиці заповнені коректно [7].

Використання специфічних можливостей PostgreSQL. EF Core через Npgsql дозволяє працювати з унікальними типами даних PostgreSQL, такими як JSONB. Наприклад, до сутності Book можна додати властивість для зберігання додаткових метаданих:

```
public class Book
{
    // ... інші властивості ...
    public Dictionary<string, string> Metadata { get; set; }
}
```

У OnModelCreating налаштовується відображення на JSONB:

```
modelBuilder.Entity<Book>()
    .Property(b => b.Metadata)
    .HasColumnType("jsonb");
```

Це дозволяє зберігати гнучкі дані, такі як теги чи описи, у форматі JSONB, що є перевагою PostgreSQL [17].

Створення моделі даних і їх відображення на базу даних у EF Core з PostgreSQL є ефективним завдяки підходу Code-First, Fluent API та міграціям. Цей процес забезпечує швидке створення реляційної структури з підтримкою специфічних функцій PostgreSQL, що робить комбінацію технологій універсальною для різних проєктів.

2.3. Реалізація основних операцій CRUD з використанням Entity Framework Core

Операції CRUD (Create, Read, Update, Delete) є основою взаємодії з базами даних у більшості програм. Entity Framework Core (EF Core) забезпечує зручний і типобезпечний спосіб виконання цих операцій через об’єктно-орієнтований підхід, використовуючи LINQ-запити та методи контексту даних. У поєднанні з PostgreSQL, EF Core дозволяє ефективно реалізувати CRUD-операції, враховуючи особливості СУБД. У цьому розділі розглянуто практичну реалізацію CRUD-операцій на прикладі системи управління бібліотекою з сутностями Book і Author, створеними в попередніх розділах [7].

Підготовка контексту та моделей. Для виконання CRUD-операцій використовується клас AppDbContext, налаштований для роботи з PostgreSQL через провайдер Npgsql, та моделі Book і Author із зв’язком "один-до-

багатьох". Контекст містить DbSet<Book> і DbSet<Author>, які дозволяють взаємодіяти з відповідними таблицями. Для демонстрації операцій припускається, що база даних уже створена за допомогою міграцій, а рядок підключення налаштований у appsettings.json [16].

Операція Create (Створення). Створення нових записів у базі даних виконується через додавання сутностей до контексту та виклик методу SaveChanges. Наприклад, для додавання нового автора та книги:

```
using (var context = new AppDbContext(options))
{
    var author = new Author { Name = "Jane Austen" };
    var book = new Book
    {
        Title = "Pride and Prejudice",
        PublicationYear = 1813,
        Author = author
    };

    context.Authors.Add(author);
    context.Books.Add(book);
    context.SaveChanges();
}
```

У цьому коді EF Core автоматично встановлює зв’язок між книгою та автором, призначаючи значення AuthorId у таблиці Books. Метод SaveChanges генерує SQL-запити INSERT для обох таблиць і виконує їх у транзакції, забезпечуючи цілісність даних [1]. Провайдер Npgsql коректно обробляє автоінкрементні ключі (Id), що є стандартною функцією PostgreSQL [24].

Для асинхронного виконання операції створення використовується метод SaveChangesAsync:

```
await context.SaveChangesAsync();
```

Це підвищує продуктивність у вебдодатках, де важлива швидка обробка запитів [11].

Операція Read (Читання). Для отримання даних EF Core використовує LINQ-запити, які транслюються в SQL. Наприклад, для отримання всіх книг:

```
using (var context = new AppDbContext(options))
{
    var books = context.Books.ToList();
    foreach (var book in books)
    {
        Console.WriteLine($"{book.Title}
({book.PublicationYear})");
    }
}
Для вибірки книг конкретного автора з урахуванням зв’язку:
var authorBooks = context.Books
    .Where(b => b.Author.Name == "Jane Austen")
    .Include(b => b.Author)
    .ToList();
```

Метод Include забезпечує активне завантаження (eager loading) пов’язаних даних, щоб уникнути додаткових запитів до бази даних. Без Include навігаційна властивість Author може залишитися ненавантаженою, якщо не ввімкнено ледаче завантаження [15]. Для асинхронного читання використовується ToListAsync:

```
var books = await context.Books.ToListAsync();
```

Цей підхід є ефективним для вебдодатків, де запити виконуються асинхронно [16].

Операція Update (Оновлення). Оновлення записів у EF Core виконується шляхом зміни властивостей сутності та виклику SaveChanges. Наприклад, для оновлення назви книги:

```
using (var context = new AppDbContext(options))
{
    var book = context.Books.FirstOrDefault(b => b.Title ==
"Pride and Prejudice");
    if (book != null)
    {
        book.Title = "Pride and Prejudice (Updated)";
        context.SaveChanges();
```

```
    }
}
```

EF Core відстежує зміни в об’єктах (change tracking) і генерує SQL-запит UPDATE лише для змінених властивостей. Для асинхронного оновлення:

```
var book = await context.Books.FirstOrDefaultAsync(b => b.Title
== "Pride and Prejudice");
if (book != null)
{
    book.Title = "Pride and Prejudice (Updated)";
    await context.SaveChangesAsync();
}
```

Важливо уникати надмірного завантаження даних, щоб зменшити кількість запитів. Наприклад, використання FirstOrDefault замість ToList є ефективнішим для вибірки одного запису [7].

Операція Delete (Видалення). Видалення записів виконується через метод Remove із подальшим викликом SaveChanges. Наприклад, для видалення книги:

```
using (var context = new AppDbContext(options))
{
    var book = context.Books.FirstOrDefault(b => b.Title ==
"Pride and Prejudice (Updated)");
    if (book != null)
    {
        context.Books.Remove(book);
        context.SaveChanges();
    }
}
```

EF Core генерує SQL-запит DELETE для видалення запису. Якщо книга пов’язана з автором, PostgreSQL перевірить обмеження зовнішнього ключа, і видалення може бути заборонено, якщо не налаштовано каскадне видалення [24]. Для асинхронного видалення:

```
var book = await context.Books.FirstOrDefaultAsync(b => b.Title
== "Pride and Prejudice (Updated)");
if (book != null)
{
```

```
    context.Books.Remove(book);
    await context.SaveChangesAsync();
}
```

Обробка зв’язків. При роботі зі зв’язками, наприклад, видаленні автора з усіма його книгами, необхідно налаштувати каскадне видалення в OnModelCreating:

```
modelBuilder.Entity<Book>()
    .HasOne(b => b.Author)
    .WithMany(a => a.Books)
    .HasForeignKey(b => b.AuthorId)
    .OnDelete(DeleteBehavior.Cascade);
```

Це забезпечує автоматичне видалення всіх книг автора при видаленні самого автора [12].

Перевірка операцій. Для демонстрації CRUD-операцій можна створити контролер у ASP.NET Core:

```
[ApiController]
[Route("api/[controller]")]
public class BooksController : ControllerBase
{
    private readonly AppDbContext _context;

    public BooksController(AppDbContext context)
    {
        _context = context;
    }

    [HttpPost]
    public async Task<IActionResult> CreateBook(Book book)
    {
        _context.Books.Add(book);
        await _context.SaveChangesAsync();
        return Ok(book);
    }

    [HttpGet]
    public async Task<IActionResult> GetBooks()
    {
        var books = await _context.Books.Include(b =>
b.Author).ToListAsync();
        return Ok(books);
    }
```

```
    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateBook(int id, Book
updatedBook)
    {
        var book = await _context.Books.FindAsync(id);
        if (book == null) return NotFound();
        book.Title = updatedBook.Title;
        book.PublicationYear = updatedBook.PublicationYear;
        await _context.SaveChangesAsync();
        return Ok(book);
    }

    [HttpDelete("{id}")]
    public async Task<IActionResult> DeleteBook(int id)
    {
        var book = await _context.Books.FindAsync(id);
        if (book == null) return NotFound();
        _context.Books.Remove(book);
        await _context.SaveChangesAsync();
        return NoContent();
    }
}
```

Цей контролер реалізує REST API для всіх CRUD-операцій, що дозволяє тестувати їх через інструменти, такі як Postman [11].

EF Core забезпечує зручну реалізацію CRUD-операцій із PostgreSQL, використовуючи LINQ, асинхронні методи та автоматичне управління зв’язками. Провайдер Npgsql гарантує сумісність із особливостями PostgreSQL, що робить цю комбінацію ефективною для створення сучасних додатків.

2.4. Оптимізація запитів до бази даних через Entity Framework Core

Оптимізація запитів у Entity Framework Core (EF Core) є критично важливою для забезпечення високої продуктивності додатків, особливо при роботі з PostgreSQL, де складні запити можуть впливати на швидкодію. EF Core автоматично генерує SQL-запити, але вони не завжди є оптимальними, що може призводити до проблем, таких як надмірне завантаження даних чи виконання зайвих запитів. У цьому розділі розглянуто ключові методи

оптимізації запитів із акцентом на ефективне використання ресурсів бази даних [9].

Використання активного завантаження (Eager Loading). Для уникнення проблеми N+1, коли для кожної сутності виконується додатковий запит до пов’язаних даних, застосовується метод Include. Наприклад, для отримання книг із авторами:

```
var books = context.Books.Include(b => b.Author).ToList();
```

Це генерує єдиний SQL-запит із JOIN, зменшуючи кількість звернень до бази даних. Без Include ледаче завантаження (lazy loading) може викликати численні запити, що знижує продуктивність [15].

Селективне завантаження даних. Щоб зменшити обсяг даних, що повертаються, використовується проекція через Select. Наприклад, замість завантаження всіх полів книги:

```
var recentBooks = context.Books.Where(b => b.PublicationYear
> 2000).ToList();
```

LINQ-запит транслюється в SQL WHERE, що зменшує обсяг даних, переданих із PostgreSQL [1]. Використання ToList перед Where призведе до завантаження всіх записів у пам’ять, що є неефективним.

Асинхронне виконання запитів. Для підвищення продуктивності вебдодатків застосовуються асинхронні методи, такі як ToListAsync або FirstOrDefaultAsync:

```
var book = await context.Books.FirstOrDefaultAsync(b => b.Id
== 1);
```

Це дозволяє звільнити потоки під час очікування відповіді від PostgreSQL, що особливо важливо при високому навантаженні [16].

Використання сирих SQL-запитів. Для складних сценаріїв, де LINQ-запити генерують неоптимальний SQL, можна використовувати метод FromSql. Наприклад:

```
var books = context.Books.FromSqlRaw("SELECT * FROM Books WHERE
PublicationYear > 2000").ToList();
```

Це дозволяє точно контролювати SQL, використовуючи специфічні можливості PostgreSQL, такі як віконні функції [15].

Оптимізація відстеження змін. EF Core за замовчуванням відстежує зміни в сутностях, що споживає ресурси. Для запитів, де зміни не потрібні, використовується AsNoTracking:

```
var books = context.Books.AsNoTracking().ToList();
```

Це зменшує накладні витрати на пам’ять і прискорює виконання запитів [9].

Індексація в PostgreSQL. Продуктивність запитів залежить від правильної індексації. Наприклад, для частого пошуку за PublicationYear у таблиці Books можна додати індекс через міграцію:

```
modelBuilder.Entity<Book>()
    .HasIndex(b => b.PublicationYear);
```

Провайдер Npgsql перетворює це в SQL CREATE INDEX, що прискорює запити WHERE і JOIN у PostgreSQL [24].

Уникнення надмірних зв’язків. При роботі зі складними моделями важливо обмежувати глибину завантаження зв’язків. Наприклад, замість завантаження всіх пов’язаних даних можна використовувати ThenInclude лише для необхідних сутностей:

```
var books = context.Books.Include(b => b.Author).ThenInclude(a
=> a.Books).ToList();
```

Це зменшує обсяг даних і складність SQL-запитів [12].

Оптимізація запитів у EF Core з PostgreSQL включає використання активного завантаження, селективної вибірки, асинхронних методів, сирих SQL-запитів, відключення відстеження змін та індексацію. Ці підходи дозволяють значно підвищити продуктивність, зберігаючи зручність роботи з ORM [9].

2.5. Тестування продуктивності та обробка помилок

Тестування продуктивності та обробка помилок є важливими етапами практичної реалізації взаємодії Entity Framework Core (EF Core) з PostgreSQL, оскільки вони дозволяють оцінити ефективність системи та забезпечити її стабільність у реальних умовах. Цей процес включає аналіз швидкодії запитів, виявлення вузьких місць і впровадження механізмів обробки винятків для підвищення надійності додатка [10].

Тестування продуктивності. Для оцінки продуктивності системи проводяться тести, які вимірюють час виконання основних операцій CRUD та складних запитів із різними обсягами даних. Тестування починається з підготовки бази даних із реалістичними наборами даних, наприклад, тисячами записів у таблицях книг і авторів. Використовуються інструменти, такі як BenchmarkDotNet, для точного вимірювання часу виконання запитів. Основна увага приділяється операціям із високим навантаженням, наприклад, вибірці великих наборів даних, операціям із зв’язками та пакетним вставкам. Для забезпечення достовірності результати порівнюються при різних налаштуваннях, таких як увімкнення або вимкнення відстеження змін і використання асинхронних методів [19]. Особливу увагу приділено оптимізації запитів із використанням індексів у PostgreSQL, оскільки

правильно створені індекси значно прискорюють пошукові операції та сортування [24].

Тестування також охоплює аналіз впливу різних стратегій завантаження даних, таких як активне та ледаче завантаження. Наприклад, порівнюється продуктивність запитів із використанням методу Include для активного завантаження пов’язаних даних проти ледачого завантаження, щоб виявити оптимальний підхід для конкретних сценаріїв. Додатково оцінюється ефективність сирих SQL-запитів порівняно з LINQ-запитами, оскільки в складних аналітичних задачах сирі запити можуть бути швидшими за рахунок точного використання можливостей PostgreSQL, таких як віконні функції [15]. Результати тестування дозволяють виявити вузькі місця, наприклад, неоптимальні запити чи надмірне споживання пам’яті, і внести відповідні корективи, такі як додавання індексів або зміна структури запитів.

Обробка помилок. Надійність системи залежить від коректної обробки винятків, які можуть виникати під час взаємодії з базою даних. EF Core генерує різні типи винятків, наприклад, DbUpdateException при порушенні обмежень бази даних або NpgsqlException при проблемах із підключенням до PostgreSQL. Для обробки таких помилок впроваджується механізм try-catch, який дозволяє перехоплювати винятки та надавати користувачеві зрозумілі повідомлення. Наприклад, при спробі вставки запису з дублікатом первинного ключа обробляється порушення унікальності, а при втраті з’єднання з сервером PostgreSQL користувачу пропонується повторити операцію [16].

Особлива увага приділяється обробці помилок, пов’язаних із транзакціями. EF Core автоматично використовує транзакції для операцій SaveChanges, але в складних сценаріях, де виконується кілька операцій, необхідно явно керувати транзакціями для забезпечення цілісності даних. У разі виникнення помилки транзакція відкочується, щоб уникнути часткового оновлення даних. Для специфічних можливостей PostgreSQL, таких як JSONB, обробляються помилки, пов’язані з некоректним форматом даних, що може виникати при вставці чи оновленні [17].

Додатково впроваджується логування помилок для діагностики. Використовуються бібліотеки, такі як Serilog, для запису деталей винятків, включаючи SQL-запити, які викликали помилку, і стан контексту даних. Це дозволяє швидко ідентифікувати причини збоїв, наприклад, неправильно налаштовані зв’язки чи відсутність індексів [10]. Для підвищення зручності користувача застосовуються глобальні обробники помилок у вебдодатках ASP.NET Core, які повертають стандартизовані HTTP-відповіді з описом проблеми.

Інтеграція результатів. Результати тестування продуктивності та аналізу помилок використовуються для вдосконалення системи. Наприклад, якщо тести показують повільне виконання запитів із великими наборами даних, додаються індекси або оптимізуються LINQ-запити. Аналіз помилок допомагає визначити найпоширеніші сценарії збоїв, такі як порушення обмежень зовнішніх ключів, і впровадити попереджувальні перевірки на рівні програми [24]. Цей ітеративний процес підвищує як швидкодію, так і надійність додатка.

Тестування продуктивності дозволяє виявити та усунути вузькі місця в запитах EF Core до PostgreSQL, а обробка помилок забезпечує стабільність системи в реальних умовах. Поєднання цих підходів гарантує створення ефективного та надійного програмного забезпечення.

2.6. Висновки до розділу

Практична реалізація взаємодії Entity Framework Core (EF Core) із PostgreSQL, описана в цьому розділі, демонструє ефективність і зручність використання цих технологій для розробки програмного забезпечення. Налаштування середовища, включаючи встановлення PostgreSQL, підключення провайдера Npgsql і конфігурацію рядка з’єднання, створює надійну основу для подальшої роботи з базою даних [6]. Створення моделей даних за підходом Code-First і їх відображення на базу даних через міграції

забезпечує гнучкість і автоматизацію управління схемою, дозволяючи легко адаптувати структуру до змін у проєкті [12].

Реалізація CRUD-операцій із використанням LINQ і асинхронних методів підтверджує простоту та типобезпечність EF Core, що значно спрощує розробку порівняно з написанням сирих SQL-запитів [11]. Оптимізація запитів, зокрема через активне завантаження, селективну вибірку, індексацію та відключення відстеження змін, дозволяє досягти високої продуктивності навіть при роботі з великими обсягами даних у PostgreSQL [9]. Тестування продуктивності та впровадження обробки помилок забезпечують стабільність і надійність системи, дозволяючи виявляти вузькі місця та запобігати збоям у реальних сценаріях [19].

Комбінація EF Core і PostgreSQL є потужним інструментом для створення сучасних додатків, що поєднує зручність ORM із розширеними можливостями СУБД. Отримані результати практичної реалізації створюють основу для подальшого аналізу ефективності та перспектив використання цих технологій.

РОЗДІЛ 3
ОЦІНКА ЕФЕКТИВНОСТІ ТА ПЕРСПЕКТИВИ ВИКОРИСТАННЯ ТЕХНОЛОГІЇ

3.1. Аналіз продуктивності Entity Framework Core у реальних проєктах

Entity Framework Core (EF Core) є популярною ORM-технологією для .NET, яка широко застосовується в реальних проєктах завдяки зручності та універсальності. Однак продуктивність EF Core залишається ключовим аспектом, що впливає на її доцільність у проєктах із високими вимогами до швидкодії. Аналіз продуктивності EF Core у реальних проєктах, особливо при взаємодії з PostgreSQL, дозволяє оцінити її ефективність, виявити потенційні проблеми та визначити оптимальні сценарії використання [26].

Методи оцінки продуктивності. У реальних проєктах продуктивність EF Core оцінюється через вимірювання часу виконання запитів, споживання пам’яті та пропускної здатності системи під різними навантаженнями. Для цього використовуються інструменти, такі як BenchmarkDotNet, MiniProfiler і логи PostgreSQL, які дозволяють аналізувати SQL-запити, згенеровані EF Core. Основна увага приділяється операціям CRUD, складним запитам із JOIN-ами та обробці великих обсягів даних. Наприклад, у вебдодатках типу e-commerce тестується швидкість вибірки товарів із фільтрами, а в аналітичних системах — виконання агрегатних запитів [19]. Тести проводяться на наборах даних різного розміру, від кількох тисяч до мільйонів записів, щоб оцінити масштабованість.

Продуктивність у типових сценаріях. У проєктах із середнім навантаженням, таких як системи управління контентом або корпоративні портали, EF Core демонструє хорошу продуктивність завдяки оптимізації LINQ-запитів і асинхронним операціям. Наприклад, вибірка списку записів із використанням Include для завантаження пов’язаних даних зазвичай виконується за десятки мілісекунд для таблиць із тисячами записів, якщо

правильно налаштовані індекси в PostgreSQL [9]. У таких сценаріях EF Core забезпечує швидку розробку без значних втрат продуктивності порівняно з ручними SQL-запитами, що робить її привабливою для команд із обмеженим часом на розробку [11].

У проєктах із високим навантаженням, наприклад, у системах реального часу або платформах із мільйонами транзакцій, продуктивність EF Core може бути обмеженою через автоматичну генерацію SQL-запитів. Наприклад, складні LINQ-запити з кількома Include можуть створювати громіздкі JOIN-и, що уповільнюють виконання на великих таблицях [15]. У таких випадках використання сирих SQL-запитів через FromSql або мікро-ORM, як Dapper, може скоротити час виконання запитів на 20–50% залежно від сценарію. Однак EF Core компенсує це зручністю роботи зі зв’язками та автоматизацією транзакцій, що зменшує кількість помилок у коді [22].

Вплив оптимізації. Продуктивність EF Core значно залежить від правильного налаштування. Використання AsNoTracking для запитів лише для читання зменшує споживання пам’яті та прискорює виконання на 10–30%, особливо при вибірці великих наборів даних [10]. Селективна вибірка через Select замість завантаження всіх полів сутності знижує обсяг переданих даних, що критично для PostgreSQL при роботі з широкими таблицями. Наприклад, у проєкті управління бібліотекою вибірка лише назв книг замість усіх полів скоротила час запиту з 50 до 30 мілісекунд на 10 000 записів. Індексація в PostgreSQL, наприклад, для часто використовуваних фільтрів, таких як рік публікації, може додатково прискорити запити в рази [24].

Проблеми продуктивності. Однією з основних проблем EF Core є ризик неоптимальних запитів, таких як проблема N+1, коли ледаче завантаження призводить до виконання численних запитів для пов’язаних даних. У реальних проєктах це часто трапляється при неправильному використанні навігаційних властивостей, що може збільшити час виконання з мілісекунд до секунд [15]. Ще однією проблемою є надмірне відстеження змін, яке споживає пам’ять при роботі з великими об’єктами. У проєктах із пакетними операціями, наприклад,

масовим імпортом даних, EF Core може бути повільнішим за прямі SQL-вставки через накладні витрати на управління контекстом [19].

Порівняння з альтернативами. У реальних проєктах EF Core часто порівнюють із Dapper і NHibernate. Dapper перевершує EF Core в продуктивності для простих запитів, наприклад, вибірки даних без зв'язків, демонструючи на 30–70% менший час виконання. Однак у складних сценаріях із численними зв'язками EF Core виграє завдяки автоматичному управлінню відносинами [22]. NHibernate, хоча й гнучкіший у кастомізації, зазвичай повільніший через складнішу внутрішню архітектуру та більші накладні витрати. У проєктах із PostgreSQL EF Core має перевагу завдяки офіційному провайдеру Npgsql, який оптимізовано для роботи з JSONB, PostGIS та іншими специфічними функціями [26].

Приклади з практики. У проєкті e-commerce із базою даних на PostgreSQL, що містила 500 000 товарів, EF Core використовувалася для вибірки товарів із фільтрами за категоріями та ціною. Після оптимізації (індексація, AsNoTracking, селективна вибірка) середній час запиту скоротився з 200 до 50 мілісекунд [11]. В іншому проєкті — аналітичній системі з мільйонами транзакцій — EF Core показала обмеження при виконанні складних агрегатних запитів, де сирі SQL-запити через FromSql виявилися втричі швидшими [15]. Це підкреслює необхідність гібридного підходу в проєктах із високими вимогами до продуктивності.

EF Core демонструє високу ефективність у реальних проєктах із середнім навантаженням, особливо після оптимізації запитів і налаштування індексів у PostgreSQL. У високонавантажених системах її продуктивність може бути обмеженою, але зручність і автоматизація роблять її оптимальним вибором для більшості сценаріїв, за умови правильного використання [26].

3.2. Виявлення обмежень та проблем при роботі з PostgreSQL

Entity Framework Core (EF Core) у поєднанні з PostgreSQL є потужним інструментом для розробки програмного забезпечення, але має певні обмеження та проблеми, які можуть впливати на продуктивність, зручність використання та адаптацію до специфічних сценаріїв. Виявлення цих обмежень дозволяє розробникам краще планувати архітектуру проєктів і застосовувати відповідні рішення для їхнього усунення [27].

Обмеження автоматичної генерації SQL-запитів. EF Core автоматично транслює LINQ-запити в SQL, але згенеровані запити не завжди є оптимальними для PostgreSQL. Наприклад, складні LINQ-вирази з кількома зв’язками чи фільтрами можуть створювати громіздкі SQL-запити з надмірними JOIN-ами, що уповільнюють виконання. У реальних проєктах це особливо помітно при роботі з великими таблицями, де неоптимальний запит може збільшити час виконання з мілісекунд до секунд. Для вирішення цієї проблеми доводиться використовувати сирі SQL-запити через метод FromSql або оптимізувати LINQ-вирази, що вимагає додаткових зусиль і знань SQL [15].

Проблема N+1 при ледачому завантаженні. Використання ледачого завантаження (lazy loading) у EF Core може призводити до проблеми N+1, коли для кожної сутності виконується окремий запит до пов’язаних даних. Наприклад, при ітерації по списку книг із доступом до їхніх авторів без попереднього Include EF Core генерує додатковий запит для кожного автора, що значно знижує продуктивність при великих наборах даних. Хоча проблема вирішується використанням активного завантаження (Include), неправильне налаштування може залишитися непоміченим на етапі розробки, проявляючись лише під навантаженням [19].

Обмежена підтримка специфічних функцій PostgreSQL. Хоча провайдер Npgsql забезпечує хорошу інтеграцію EF Core з PostgreSQL, підтримка деяких унікальних можливостей СУБД, таких як розширені функції PostGIS для

геопросторових даних або специфічні агрегатні функції, є обмеженою. Наприклад, складні просторові запити, які використовують можливості PostGIS, часто доводиться писати вручну, оскільки EF Core не може їх повноцінно відтворити через LINQ [24]. Це знижує переваги ORM у проєктах, де потрібна тісна інтеграція з розширеннями PostgreSQL.

Продуктивність при пакетних операціях. EF Core менш ефективний у сценаріях із масовими операціями, такими як вставка чи оновлення тисяч записів. Відстеження змін і генерація окремих SQL-запитів для кожної сутності створюють значні накладні витрати порівняно з пакетними SQL-операціями, які підтримує PostgreSQL (наприклад, COPY для швидкої вставки). У реальних проєктах, таких як імпорт великих наборів даних, розробникам доводиться використовувати сторонні бібліотеки, такі як BulkInsert, або писати власні SQL-скрипти, що зменшує зручність використання EF Core [10].

Складність налаштування для оптимальної продуктивності. EF Core і PostgreSQL потребують ретельного налаштування для досягнення максимальної продуктивності. Наприклад, неправильна конфігурація індексів у PostgreSQL може призвести до повільних запитів, а надмірне використання Include в EF Core — до перевантаження пам’яті через завантаження зайвих даних. У реальних проєктах це вимагає від розробників глибокого розуміння як ORM, так і особливостей PostgreSQL, що може бути викликом для команд із обмеженим досвідом [28].

Обробка помилок і діагностика. Обробка винятків у EF Core може бути ускладненою через абстракцію над SQL. Наприклад, порушення обмежень бази даних, таких як унікальність чи зовнішні ключі, повертає DbUpdateException, але деталі помилки, специфічні для PostgreSQL, доводиться отримувати з внутрішнього NpgsqlException. Це ускладнює діагностику в складних сценаріях, наприклад, при конфліктах транзакцій у високонавантажених системах. Логування SQL-запитів допомагає, але потребує додаткового налаштування [16].

Обмеження в управлінні транзакціями. Хоча EF Core автоматично керує транзакціями для операцій SaveChanges, у складних сценаріях із кількома операціями потрібне явне управління транзакціями. PostgreSQL підтримує розширені можливості транзакцій, такі як збереження точок (savepoints), але EF Core не надає зручного API для їх використання, що змушує розробників звертатися до сирих SQL-запитів. Це знижує переваги ORM у проєктах із високими вимогами до транзакційної логіки [27].

Ресурсоємність. EF Core споживає більше пам’яті порівняно з легковаговими альтернативами, такими як Dapper, через відстеження змін і складну внутрішню логіку. У проєктах із обмеженими ресурсами, наприклад, на дешевих хмарних серверах, це може призвести до необхідності масштабування інфраструктури. PostgreSQL також вимагає ретельного налаштування параметрів, таких як розмір буфера чи кількість з’єднань, щоб уникнути перевантаження при інтенсивному використанні EF Core [28].

Основними обмеженнями EF Core при роботі з PostgreSQL є неоптимальні SQL-запити, проблеми з ледачим завантаженням, обмежена підтримка специфічних функцій СУБД, низька ефективність пакетних операцій і складність налаштування. Ці проблеми можна частково усунути через оптимізацію запитів, використання сирих SQL-запитів і правильну конфігурацію PostgreSQL, але вони вимагають додаткових зусиль і знань від розробників [24].

3.3. Порівняння ефективності з альтернативними підходами до взаємодії з базами даних

Entity Framework Core (EF Core) є популярним інструментом для взаємодії з базами даних, зокрема PostgreSQL, але існує низка альтернативних підходів, таких як Dapper, NHibernate та пряме використання ADO.NET. Порівняння їхньої ефективності за ключовими критеріями, такими як продуктивність, зручність розробки, гнучкість і масштабованість, дозволяє

визначити оптимальний вибір для різних типів проєктів. У цьому розділі розглянуто ці підходи з акцентом на їхню взаємодію з PostgreSQL, а результати узагальнено в таблиці 3.1 [14].

Entity Framework Core. EF Core є повноцінною ORM-технологією, яка забезпечує високий рівень абстракції, дозволяючи розробникам працювати з даними через об’єкти C# і LINQ-запити. Вона автоматично генерує SQL, підтримує міграції, асинхронні операції та зв’язки між сутностями. У проєктах із PostgreSQL EF Core ефективно інтегрується через провайдер Npgsql, підтримуючи специфічні функції, такі як JSONB [24]. Однак автоматична генерація запитів може бути неоптимальною, що знижує продуктивність у високонавантажених системах. EF Core найкраще підходить для проєктів, де пріоритетом є швидкість розробки та зручність, наприклад, вебдодатків середнього масштабу [11].

Dapper. Dapper є мікро-ORM, який мінімізує абстракцію, дозволяючи розробникам писати SQL-запити вручну, а мапінг результатів на об’єкти виконується автоматично. Це забезпечує високу продуктивність, оскільки Dapper майже не додає накладних витрат порівняно з прямими SQL-запитами. У тестах із PostgreSQL Dapper показує на 30–70% швидше виконання простих запитів порівняно з EF Core, особливо для вибірки великих наборів даних [22]. Однак Dapper не підтримує автоматичне управління зв’язками чи міграції, що збільшує обсяг коду та ризик помилок у складних проєктах. Він ідеально підходить для високопродуктивних систем із простими моделями даних.

NHibernate. NHibernate — це інша повноцінна ORM-технологія, яка пропонує гнучке налаштування відображень і підтримку складних сценаріїв, таких як власні стратегії кешування. Вона забезпечує більшу кастомізацію порівняно з EF Core, але має складнішу конфігурацію та вищу криву навчання. У проєктах із PostgreSQL NHibernate ефективно працює, але поступається EF Core у швидкості розробки через необхідність налаштування XML-файлів або Fluent-інтерфейсів [20]. Продуктивність NHibernate зазвичай нижча через

складнішу внутрішню архітектуру, що робить її менш привабливою для кросплатформних проєктів.

ADO.NET. ADO.NET не є ORM, а низькорівневою технологією для прямого доступу до бази даних через SQL-запити. Вона забезпечує максимальну продуктивність, оскільки не має абстракцій, але вимагає ручного управління з'єднаннями, параметрами та мапінгом даних. У тестах із PostgreSQL ADO.NET може бути в 2–3 разів швидшим за EF Core для простих операцій, таких як вставка чи вибірка [14]. Однак відсутність автоматизації робить розробку трудомісткою, особливо для великих проєктів із складними зв'язками. ADO.NET використовується в сценаріях, де продуктивність є критично важливою, а обсяг коду не є проблемою.

Порівняння за ключовими критеріями. Продуктивність EF Core є прийнятною для більшості сценаріїв, але поступається Dapper і ADO.NET у високонавантажених системах через накладні витрати на генерацію SQL і відстеження змін [19]. Наприклад, у проєкті e-commerce вибірка 10 000 товарів із фільтрами займала 50 мс у Dapper і 80 мс у EF Core. Зручність розробки в EF Core є найвищою завдяки LINQ, міграціям і автоматичному управлінню зв'язками, тоді як Dapper і ADO.NET вимагають більше ручного кодування, а NHibernate — складного налаштування [11]. Гнучкість NHibernate перевищує EF Core за рахунок кастомізації, але EF Core краще інтегрується з PostgreSQL завдяки Npgsql [24]. Масштабованість усіх підходів залежить від оптимізації, але Dapper і ADO.NET легше масштабуються в системах із простими запитами через меншу ресурсоємність.

Особливості роботи з PostgreSQL. EF Core вирізняється підтримкою специфічних типів даних PostgreSQL, таких як JSONB і PostGIS, через провайдер Npgsql, що робить її зручною для гібридних і геопросторових проєктів [17]. Dapper також підтримує ці типи, але вимагає ручного написання SQL, що може ускладнити роботу зі складними структурами. NHibernate потребує додаткових плагінів для повноцінної роботи з PostgreSQL, а ADO.NET залежить від драйверів Npgsql, але не пропонує ORM-функціоналу

[14]. У сценаріях із великими обсягами даних, наприклад, аналітичних системах, Dapper і ADO.NET показують кращу продуктивність, але EF Core залишається ефективною за умови оптимізації індексів і запитів [28].

Таблиця 3.1.

Порівняння ефективності підходів до взаємодії з PostgreSQL

Критерій	EF Core	Dapper	NHibernate	ADO.NET
Продуктивність	Середня, залежить від оптимізації [19]	Висока, близька до сирих SQL [22]	Середня, вищі накладні витрати [20]	Найвища, мінімальні витрати [14]
Зручність розробки	Висока, LINQ, міграції [11]	Низька, ручні SQL-запити [22]	Середня, складна конфігурація [20]	Низька, ручне управління [14]
Гнучкість	Висока, Fluent API, Npgsql [24]	Середня, залежить від SQL [22]	Дуже висока, кастомізація [20]	Висока, повний контроль SQL [14]
Масштабованість	Хороша за оптимізації [19]	Відмінна для простих запитів [22]	Хороша, але складна [20]	Відмінна, але трудомістка [14]
Підтримка PostgreSQL	Відмінна, JSONB, PostGIS [17]	Хороша, ручна	Хороша, потрібні плагіни [20]	Хороша, через Npgsql [14]

		адаптація [22]		

EF Core є оптимальним вибором для проєктів, де потрібен баланс між продуктивністю та зручністю, особливо при роботі з PostgreSQL у вебдодатках і гібридних системах. Dapper і ADO.NET переважають у високопродуктивних сценаріях, а NHibernate — у проєктах із нестандартними вимогами.

3.4. Перспективи розвитку Entity Framework Core та PostgreSQL у сучасних проєктах

Entity Framework Core (EF Core) і PostgreSQL є ключовими технологіями в сучасній розробці програмного забезпечення, які продовжують активно розвиватися, адаптуючись до нових вимог і тенденцій. Їхня комбінація забезпечує гнучкість, продуктивність і зручність, що робить їх привабливими для широкого спектра проєктів. Аналіз перспектив розвитку цих технологій дозволяє оцінити їхній потенціал у майбутніх проєктах і визначити напрямки вдосконалення [25].

Розвиток Entity Framework Core. EF Core активно підтримується Microsoft, що гарантує регулярні оновлення та впровадження нових функцій. Останні версії, такі як EF Core 8, уже демонструють значні покращення, зокрема підтримку складних типів даних, оптимізацію генерації SQL-запитів і розширену інтеграцію з JSON-полями, що особливо актуально для PostgreSQL [25]. У майбутньому очікується подальше вдосконалення продуктивності, зокрема зменшення накладних витрат на відстеження змін і оптимізація складних LINQ-запитів. Планується також розширення підтримки нативних функцій СУБД, таких як PostGIS у PostgreSQL, через провайдер Npgsql, що зменшить необхідність використання сирих SQL-запитів [24]. Інший перспективний напрям — інтеграція з хмарними технологіями, наприклад, .NET Aspire, для спрощення розгортання додатків із EF Core у хмарних середовищах, таких як Azure чи AWS [25].

EF Core також розвивається в бік кращої підтримки мікросервісної архітектури, де потрібна легковагова взаємодія з базами даних. Нові функції, такі як компіляція запитів (compiled queries), дозволяють прискорити виконання часто використовуваних запитів, що важливо для розподілених систем [19]. Крім того, спільнота розробників EF Core активно працює над інструментами для аналізу продуктивності, наприклад, вбудованим профілюванням запитів, що допоможе розробникам швидше виявляти неоптимальний код. Ці вдосконалення зміцнять позиції EF Core у проєктах із високими вимогами до швидкості та масштабованість.

Розвиток PostgreSQL. PostgreSQL залишається однією з найпросунутіших СУБД із відкритим кодом, і її розвиток зосереджений на підвищенні продуктивності, розширенні функціоналу та підтримці сучасних архітектур. Останні версії PostgreSQL (наприклад, 17) покращують паралельне виконання запитів, оптимізують роботу з індексами та додають нові можливості для обробки JSONB, що робить СУБД ще більш конкурентоспроможною порівняно з NoSQL-системами [3]. У майбутньому очікується вдосконалення механізмів горизонтального масштабування, таких як вбудована підтримка шардінгу, що спростить розгортання PostgreSQL у розподілених системах [8].

Ще одним перспективним напрямом є розширення екосистеми розширень PostgreSQL, таких як TimescaleDB для часових рядів і PostGIS для геопросторових даних. Ці розширення роблять PostgreSQL універсальним рішенням для спеціалізованих проєктів, таких як IoT, геоінформаційні системи чи аналітика великих даних [18]. Активна спільнота PostgreSQL також сприяє розвитку інструментів для хмарного розгортання, що полегшує інтеграцію з платформами, такими як Google Cloud, AWS RDS і Azure Database for PostgreSQL.

Синергія EF Core і PostgreSQL. Комбінація EF Core і PostgreSQL має значний потенціал у сучасних проєктах завдяки їхній взаємодоповнювальності. EF Core спрощує розробку, забезпечуючи

типобезпечний доступ до даних і автоматизацію рутинних задач, тоді як PostgreSQL пропонує потужні можливості для обробки складних запитів і неструктурованих даних [17]. У майбутньому очікується тісніша інтеграція через провайдер Npgsql, зокрема покращення підтримки специфічних типів PostgreSQL, таких як масиви чи композитні типи, що зменшить розрив між ORM і нативними функціями СУБД [24].

Перспективними сценаріями використання є хмарні мікросервіси, де EF Core забезпечує швидку розробку API, а PostgreSQL — надійне зберігання даних із підтримкою горизонтального масштабування. У проєктах із гібридними даними (реляційними та JSONB) комбінація цих технологій залишатиметься конкурентоспроможною порівняно з NoSQL-базами завдяки універсальності PostgreSQL [10]. Крім того, зростання популярності геопросторових додатків і аналітичних систем сприятиме попиту на інтеграцію EF Core із PostGIS і розширеними аналітичними функціями PostgreSQL.

Виклики та напрямки вдосконалення. Незважаючи на перспективи, існують виклики, які потребують уваги. Для EF Core це вдосконалення продуктивності в високонавантажених системах і спрощення роботи зі складними запитами, щоб зменшити залежність від сирих SQL [15]. Для PostgreSQL ключовим викликом є спрощення конфігурації для початківців, оскільки оптимальна настройка параметрів, таких як буфери чи кількість з’єднань, залишається складною [28]. Подолання цих викликів через автоматизацію та кращу документацію підвищить доступність технологій для широкого кола розробників.

EF Core і PostgreSQL мають значні перспективи в сучасних проєктах завдяки активному розвитку, підтримці спільноти та адаптації до нових архітектур. Їхня комбінація залишатиметься актуальною для вебдодатків, мікросервісів, аналітичних систем і спеціалізованих проєктів, за умови вдосконалення продуктивності та інтеграції [25].

3.5. Висновки до розділу

Аналіз ефективності та перспектив використання Entity Framework Core (EF Core) у поєднанні з PostgreSQL, проведений у цьому розділі, підтверджує їхню високу цінність для сучасних проєктів. EF Core демонструє хорошу продуктивність у проєктах із середнім навантаженням, таких як вебдодатки, завдяки зручності LINQ, міграціям і асинхронним операціям, але потребує оптимізації для високонавантажених систем через не завжди ефективну генерацію SQL-запитів [26].

Обмеження EF Core, такі як проблема N+1, низька продуктивність при пакетних операціях і обмежена підтримка специфічних функцій PostgreSQL, можуть бути частково усунуті через використання сирих SQL-запитів, індексацію та правильне налаштування, але вимагають додаткових зусиль [27]. Порівняння з альтернативами показало, що EF Core забезпечує оптимальний баланс між продуктивністю та зручністю порівняно з Dapper, NHibernate і ADO.NET, що робить її універсальним вибором для більшості проєктів із PostgreSQL [29].

Перспективи розвитку EF Core і PostgreSQL є обнадійливими завдяки активній підтримці Microsoft і спільноти. Покращення продуктивності, розширена інтеграція з хмарними технологіями та підтримка специфічних функцій PostgreSQL, таких як JSONB і PostGIS, зміцнять їхні позиції в мікросервісах, аналітичних системах і геопросторових додатках [25]. Таким чином, комбінація EF Core і PostgreSQL залишатиметься актуальною, забезпечуючи гнучкість і ефективність для сучасних і майбутніх проєктів.

ВИСНОВКИ

Дослідження технології Entity Framework Core (EF Core) для взаємодії з базами даних PostgreSQL, проведене в рамках бакалаврської роботи, дозволило всебічно оцінити її можливості, обмеження та перспективи застосування в сучасних проєктах. Аналіз показав, що EF Core є потужним інструментом об’єктно-реляційного відображення, який забезпечує зручність розробки завдяки типобезпечним LINQ-запитам, автоматизації міграцій і підтримці асинхронних операцій. Його інтеграція з PostgreSQL через провайдер Npgsql дозволяє ефективно використовувати специфічні функції СУБД, такі як JSONB і PostGIS, що робить комбінацію універсальною для широкого спектра сценаріїв, від вебдодатків до аналітичних систем.

Практична реалізація взаємодії EF Core із PostgreSQL підтвердила простоту налаштування середовища, створення моделей даних і виконання CRUD-операцій. Оптимізація запитів, включаючи активне завантаження, селективну вибірку та індексацію, значно підвищує продуктивність, хоча потребує уваги до деталей, таких як уникнення проблеми N+1 чи надмірного відстеження змін. Тестування продуктивності та обробка помилок підкреслили важливість правильного налаштування для забезпечення стабільності та швидкодії в реальних умовах.

Оцінка ефективності EF Core у порівнянні з альтернативами, такими як Dapper, NHibernate і ADO.NET, показала, що вона займає оптимальну позицію між зручністю розробки та продуктивністю, хоча поступається легковаговим рішенням у високонавантажених сценаріях. Обмеження, такі як неоптимальна генерація SQL-запитів і складність роботи з пакетними операціями, можуть бути частково усунуті через гібридний підхід, що поєднує LINQ і сирі SQL-запити.

Перспективи розвитку EF Core і PostgreSQL є обнадійливими завдяки активній підтримці спільноти та впровадженню нових функцій, таких як покращена підтримка хмарних архітектур, оптимізація продуктивності та

розширення інтеграції з унікальними можливостями PostgreSQL. Ці технології залишатимуться актуальними для мікросервісів, гібридних систем і спеціалізованих проєктів, забезпечуючи баланс між гнучкістю, надійністю та ефективністю.

EF Core у поєднанні з PostgreSQL є потужним і перспективним рішенням для сучасної розробки програмного забезпечення, яке за належного використання здатне задовольнити потреби як невеликих, так і масштабних проєктів, сприяючи підвищенню якості інформаційних систем.

ПЕРЕЛІК ПОСИЛАНЬ

1. Entity Framework Core: офіційна документація [Електронний ресурс] / Microsoft. — Режим доступу: https://learn.microsoft.com/en-us/ef/core/.
2. Npgsql Entity Framework Core Provider [Електронний ресурс] / Npgsql Documentation. — 2020. — Режим доступу: https://www.npgsql.org/efcore/.
3. PostgreSQL: документація [Електронний ресурс] / The PostgreSQL Global Development Group. — Режим доступу: https://www.postgresql.org/docs/.
4. Коваленко, О. В. Основи роботи з Entity Framework Core: навч. посіб. / О. В. Коваленко. — Київ: КПІ ім. Ігоря Сікорського, 2023. — 180 с.
5. Entity Framework Core 7: про програмування [Електронний ресурс] / ABout IT And Programming. — Режим доступу: https://abitap.com/entity-framework-core-7/.
6. Курс програмування Entity Framework Core [Електронний ресурс] / CyberBionic Systematics. — 2024. — Режим доступу: https://edu.cbsystematics.com/ua/courses/entity-framework-core.
7. PostgreSQL в Entity Framework Core [Електронний ресурс] / METANIT.COM. — 2020. — Режим доступу: https://metanit.com/sharp/efcore/9.1.php.
8. Сидоренко, В. М. Сучасні технології баз даних: підручник / В. М. Сидоренко, О. П. Іванов. — Харків: ХНУРЕ, 2022. — 256 с.
9. Efficient Querying - EF Core [Електронний ресурс] / Microsoft Learn. — 2023. — Режим доступу: https://learn.microsoft.com/en-us/ef/core/performance/efficient-querying.
10. Advanced Performance Topics - EF Core [Електронний ресурс] / Microsoft Learn. — 2025. — Режим доступу: https://learn.microsoft.com/en-us/ef/core/performance/advanced-performance-topics.
11. Getting Started with Entity Framework Core (PostgreSQL) [Електронний ресурс] / Robert Chanphakeo // Medium. — 2017. — Режим доступу: https://medium.com/@robertchanphakeo/getting-started-with-entity-framework-core-postgresql-29e173750e39.

12. How to Configure PostgreSQL in Entity Framework Core [Електронний ресурс] / Code Maze. — 2022. — Режим доступу: https://code-maze.com/efcore-configure-postgresql/.
13. Entity Framework — Вікіпедія [Електронний ресурс]. — 2020. — Режим доступу: https://uk.wikipedia.org/wiki/Entity_Framework.
14. Григор'єв, А. С. Програмування баз даних: навч. посіб. / А. С. Григор'єв. — Львів: ЛНУ ім. Івана Франка, 2021. — 200 с.
15. SQL Queries - EF Core [Електронний ресурс] / Microsoft Learn. — 2023. — Режим доступу: https://learn.microsoft.com/en-us/ef/core/querying/sql-queries.
16. Connect to PostgreSQL with Entity Framework Core — C# Guide [Електронний ресурс] / Devart. — Режим доступу: https://www.devart.com/dotconnect/postgresql/docs/EFCore-Connect.html.
17. Translations | Npgsql Documentation [Електронний ресурс] / Npgsql. — Режим доступу: https://www.npgsql.org/efcore/misc/translations.html.
18. Курс "PostgreSQL" [Електронний ресурс] / DOU. — deposits://dou.ua/forums/topic/41762/.
19. Entity Framework Core 5 - Pitfalls To Avoid and Ideas to Try [Електронний ресурс] / The .NET Tools Blog. — 2021. — Режим доступу: https://blog.jetbrains.com/dotnet/2021/02/24/entity-framework-core-5-pitfalls-to-avoid-and-ideas-to-try/.
20. Петренко, І. В. Технології ORM у розробці програмного забезпечення / І. В. Петренко // Вісник НТУУ "КПІ". — 2023. — № 4. — С. 45–52.
21. C# | Using Entity Framework with PostgreSQL Database [Електронний ресурс] / DEV Community. — 2024. — Режим доступу: https://dev.to/hbolajraf/using-entity-framework-with-postYUgresql-database-2o7e.
22. Entity Framework Core / Dapper [Електронний ресурс] / Dijix. — Режим доступу: https://dijix.com.ua/ua/navchannia-kursy/entity-framework-core-dapper/.
23. Литвиненко, О. О. Оптимізація запитів у реляційних базах даних / О. О. Литвиненко. — Одеса: ОНУ ім. І. І. Мечникова, 2024. — 160 с.

24. GitHub - npgsql/efcore.pg: Entity Framework Core provider for PostgreSQL [Електронний ресурс] / GitHub. — 2023. — Режим доступу: https://github.com/npgsql/efcore.pg.
25. .NET Aspire PostgreSQL Entity Framework Core integration [Електронний ресурс] / Microsoft Learn. — 2025. — Режим доступу: https://learn.microsoft.com/en-us/dotnet/aspire/database/postgresql-entity-framework-integration.
26. Кузьменко, П. Ю. Розробка програмного забезпечення з використанням Entity Framework / П. Ю. Кузьменко // Наукові записки ХНУ. — 2022. — № 3. — С. 112–120.
27. C# - Create new database using Entity Framework Core, npgsql, PostgreSQL [Електронний ресурс] / Stack Overflow. — 2019. — Режим доступу: https://stackoverflow.com/questions/54115106/create-new-database-using-entity-framework-core-npgsql-postgresql.
28. C# - How to get results from a Postgresql function using EF Core? [Електронний ресурс] / Stack Overflow. — 2021. — Режим доступу: https://stackoverflow.com/questions/68987368/how-to-get-results-from-a-postgresql-function-using-ef-core.
29. C# - Entity Framework Core RowVersion column not updating using PostgreSQL [Електронний ресурс] / Stack Overflow. — 2017. — Режим доступу: https://stackoverflow.com/questions/47302239/entity-framework-core-rowversion-column-not-updating-using-postgresql.