Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження архітектурних моделей та методів для побудови ORM та їх легковагових варіантів на платформі .net

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-5

Шпорта А. О.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-наукова

Керівник доц., к.т.н. Мельнікова Р.В.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий(магістерський)

Спеціальність 121-Інженерія програмного забезпечення

(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ

Зав. Кафедри_____

(підпис)

«____»_____ 2024.

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Шпорта Артем Олексійович

(ПІБ)

1. Тема роботи «Дослідження архітектурних моделей та методів для побудови ORM та їх легковагових варіантів на платформі .net».

Затверджена наказом по університету від «29» березня 2024 р. №250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 25.06.2024.

3. Вихідні дані до роботи: аналіз досліджуваних існуючих ORM – Hibernate, EF Core, PetaPoco. Архітектурна модель стандартизованої компактної ORM, мови програмування C#, технології .NET Framework 4.8, .NET 6.0, .NET 7.0,.середовище розробки Visual Studio 2019.

4. Перелік питань, що потрібно опрацювати в роботі: дослідження предметної області для аналізу існуючих ORM, дослідження їх структурних особливостей, вибір необхідних архітектурних патернів та розробка архітектурного патерну для компактних ORM. Знайти оптимальну архітектурну модель та провести аналіз на її працездатність та універсальність для використання.

КАЛЕНДАРНИЙ ПЛАН

№	Назви етапів курсової роботи	Термін виконання етапів роботи	Примітка
1	Аналіз проблемної області та постановка задачі	23.01 – 14.02.24	виконано
2	Аналіз та вибір існуючих ORM для дослідження	15.02 – 24.02.24	виконано
3	Аналіз та моделювання предметної області	17.02 – 28.02.24	виконано
4	Аналіз методів роботи ORM	28.02- 10.03.2024	виконано
5	Планування експериментів та виведення стандартизованої архітектури ORM	10.03 – 15.03.2024	виконано
6	Програмна реалізація та проведення експериментів	15.03 – 20.04.2024	виконано
7	Аналіз результатів експериментальних досліджень	20.04 – 23.04.2024	виконано
8	Написання та оформлення статті та тез доповіді	17.04 – 23.04.24	виконано
9	Підготовка пояснювальної записки	01.05 – 31.05.24	виконано
10	Підготовка презентації та доповіді	31.05 – 03.06.2024	виконано
11	Нормоконтроль	9.06 – 11.06.2024	виконано
12	Рецензування	11.06 – 15.06.2024	виконано
13	Занесення диплома в електронний архів	18.06.2024	виконано
14	Попередній захист	20.06.2024	виконано
15	Допуск до захисту у зав. кафедри	25.06.2024	виконано

Дата видачі завдання 20 січня 2024 р.

Студент _____ Шпорта А. О.
(підпис) (прізвище, ініціали)

Керівник кваліфікаційної роботи _____ доц., к.т.н. Мельнікова Р.В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи магістра містить: 65 с., 9 рис., 1 таб., 11 джерел.

АРХІТЕКТУРА, ПАТЕРН, КОМПАКТНА ORM, СТАНДАРТИЗАЦІЯ, ORM.

Об’єктом дослідження – архітектура легковагових ORM на платформі .Net.

Метою роботи є дослідження існуючих ORM та їх структурних особливостей, вибір необхідних архітектурних патернів та розробка стандартизованої архітектури для компактних ORM.

В результаті спроектовано оптимальну архітектурну модель та протестовано її працездатність за допомогою програмного застосунку.

STANDARDIZATION, ORM, PATTERN, COMPACT ORM, ARCHITECTURE.

The object of the study is the architecture of lightweight ORMs on the .Net platform.

The purpose of the work is to study existing ORMs and their structural features, select the necessary architectural patterns, and develop a standardized architecture for compact ORMs.

As a result, the optimal architectural model will be found, and its performance and versatility for use will be analyzed.

Я, Шпорта Артем Олексійович, студент гр. ІПЗм-22-5, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота на тему «Дослідження архітектурних моделей та методів для побудови ORM та їх легковагових варіантів на платформі .net», що буде представлена в

екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений(на) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

У сучасному інформаційному суспільстві, яке прагне до стрімкого росту обсягів даних та технологічного розвитку, системи управління базами даних стають важливою складовою для ефективної організації та забезпечення доступу до інформації. Протягом багатьох десятиліть еволюції комп'ютерних технологій, питання зберігання та обробки даних завжди стояли на передньому плані. Виникла потреба у засобах, які дозволили б програмістам та розробникам працювати з даними більш ефективно та з меншими труднощами.

До появи об'єктно-реляційного відображення (ORM), взаємодія програмного забезпечення з реляційними базами даних вимагала від розробників написання складних SQL-запитів та обробки результатів, що робило процес розробки програм надзвичайно витратним у часі та зусиллях. Це ставало особливо проблематичним у великих та складних проектах, де управління даними стає справжньою складністю.

Розробка архітектурного патерну для компактних ORM є необхідною, оскільки вона сприяє створенню структурованих та оптимальних рішень для доступу до даних в невеликих проектах. Патерн допомагає визначити кращі практики, стандарти та методи, які спрощують розробку, підтримку та вдосконалення компактних ORM, забезпечуючи високу якість, ефективність та надійність рішень для роботи з базами даних.

В результаті це сприяє подальшому розвитку та поширенню компактних ORM, забезпечуючи їхню успішну інтеграцію в різноманітні проекти та сприяючи спільній базі знань для спільноти розробників.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Загальна характеристика ORM

Історія об'єктно-реляційного відображення (ORM) виходить з коренів розвитку реляційних баз даних (РБД) та змін в уявленнях про організацію даних у комп'ютерних системах. Поява ORM була зумовлена багатьма ключовими факторами та етапами, які також вплинули на її майбутній розвиток.

ORM виникли як реакція на появу реляційних баз даних у 1970-х роках(РБД), стали стандартом організації даних у базах даних. Ця модель передбачала зберігання даних у вигляді таблиць зі зв'язками між ними, що спростило зберігання та обробку великих обсягів інформації.

У той же час, коли виникли РБД, розвивалася і об'єктно-орієнтована парадигма програмування. Ця парадигма передбачає моделювання програмних об'єктів після реальних об'єктів та взаємодію з ними, що спрощує розробку програм та полегшує роботу з даними.

Однак, коли розробники спробували поєднати РБД і об'єктно-орієнтовану парадигму, виникли проблеми. Діапазони функцій, які пропонували РБД та ООП, були різними, і виникала несумісність між моделями даних в програмному коді та в базі даних. Це призводило до потреби вручну зберігати та оновлювати дані в обох системах, що було витратним та складним завданням.

Відповіддю на цю проблему стало створення ORM. Перші ORM-системи з'явилися у 1990-х роках та надали можливість автоматичного відображення даних між об'єктами в програмному коді та таблицями в РБД. ORM дозволило розробникам працювати з даними в об'єктно-орієнтованому стилі, спрощуючи роботу з РБД та забезпечуючи сумісність між двома системами.

З часом ORM стали популярними та отримали підтримку в різних програмних мовах. Також були розроблені стандарти, такі як Java Persistence API (JPA) для Java, які нормалізували та стандартизували роботу з ORM.

Загалом, поява ORM була результатом поєднання потреб розробників у спрощенні роботи з даними та несумісності між реляційними базами даних і об'єктно-орієнтованою парадигмою програмування. ORM стали важливим

інструментом для розробників, які бажають працювати з даними у вигляді об'єктів та забезпечити сумісність між програмним кодом та базами даних.

Сучасні об'єктно-реляційні відображення (ORM) є невід'ємною частиною розробки веб-сервісів та програмних застосунків. Вони забезпечують зручний та ефективний спосіб взаємодії з базами даних, що є важливою складовою сучасних інформаційних систем.

1.2 Особливості різновидів ORM

Об'єктно-реляційні відображення (ORM) мають кілька різновидів та підходів, які можуть бути використані в залежності від конкретних потреб розробки.

Активний запис (Active Record). Цей різновид ORM базується на ідеї, що об'єкт зберігає свої власні дані та вміє виконувати з ними операції. Активний запис активно використовується в Ruby on Rails та інших фреймворках. Об'єкти, які представляють записи в базі даних, мають методи для збереження, оновлення, видалення інформації. Цей підхід простий та інтуїтивно зрозумілий, але може бути недостатньо гнучким у складних випадках.

Активний запит (Active Query). Цей підхід розділяє логіку запитів до бази даних від моделей даних. Запити формуються у вигляді об'єктів та можуть бути складними та динамічно формованими. Цей стиль підходить для складних операцій з даними, де потрібна гнучкість у формулюванні запитів.

Активний об'єкт (Active Object). В цьому різновиді ORM об'єкти представляють асинхронні операції та можуть взаємодіяти з базою даних в асинхронному режимі. Це корисно для додатків, які працюють у високопродуктивних та розподілених середовищах.

Сховище даних (Data Mapper). У цьому підході об'єкти моделей не наділені логікою зберігання та вибірки даних. Замість цього, окремий компонент, відомий як "маппер даних," відповідає за відображення об'єктів на записи в базі даних та навпаки. Це дає більшу гнучкість у роботі з базою даних, але може бути складнішим для розуміння та використання.

Компактний ORM (Micro ORM). Цей клас ORM-систем призначений для малих проектів або там, де простота та ефективність є більшим пріоритетом, ніж повний функціонал ORM. Вони надають базовий набір функцій для роботи з базою даних та зазвичай мають невеликий розмір та об'єм[1].

Загалом, ORM представляють різні підходи до роботи з базами даних та мають свої переваги та недоліки в залежності від вимог та характеру проекту. Розробники можуть вибирати той різновид ORM, який найкраще відповідає їхнім потребам та особливостям розробки.

1.3 Актуальність проблеми

Компактні ORM (Object-Relational Mapping) є важливим компонентом розробки програмного забезпечення, особливо в сучасному світі, де доступ до даних із баз даних є однією з найбільш поширених задач. Компактні ORM створюють можливість розробникам працювати з даними у вигляді об'єктів програмного коду, а не SQL-запитів та рядків даних. Вони спрощують взаємодію з базами даних та дозволяють створювати більш читабельний та підтримуваний код.

Проте, на сьогоднішній день не існує конкретних стандартних інструкцій або загальної архітектурної моделі для розробки компактних ORM. Кожна бібліотека або фреймворк, які надають такий функціонал, може мати свою власну архітектурну концепцію та інтерфейс[2].

Ця ситуація створює необхідність розробити загальний підхід до архітектури компактних ORM. Дана необхідність зумовлена наступними факторами:

- відсутність загальних стандартів та рекомендацій з архітектури компактних ORM може призвести до великої різноманітності підходів, що ускладнює вибір правильної бібліотеки або фреймворку для проекту;
- розробка загальної архітектурної моделі може допомогти спростити розробку компактних ORM, забезпечуючи консистентність та стандартизацію;

- загальний підхід може полегшити підтримку і розширення існуючих компактних ORM. Розробники можуть бути більш обізнані з архітектурними концепціями та краще розуміти, як працює компонент;
- створення загального підходу до архітектури компактних ORM може сприяти розвитку спільноти та розширенню знань та ресурсів для цього напрямку розробки.

Отже, необхідність розробки загального підходу до архітектури компактних ORM визначається потребою в стандартизації, спрощенні розробки, підвищенні продуктивності та покращенні підтримки та розширення таких бібліотек.

1.4 Постановка задачі

В кваліфікаційній роботі буде проаналізовано предметну область дослідження та її об’єктів, також буде розібрано, які основні ORM існують, далі розберемо як вони працюють та їх особливості, розкладемо до архітектурних патернів усі архітектурні та складові елементи цих ORM.

Проведено порівняльний аналіз та аналіз призначення кожного архітектурного елементу обраної ORM, надалі планується створити загальну архітектурну модель, яка буде відображати ідеальну або рекомендовану основу для створення компактних ORM.

Робота спрямована на виявлення закономірностей в архітектурах сучасних ORM, на основі знайдених закономірностей буде сформовано стандартизовану архітектуру для компактних варіантів.

Реалізація наукового дослідження складається з наступних етапів:

- аналіз предметної області;
- аналіз архітектурних особливостей сучасних ORM;
- знаходження закономірностей та логічних висновків щодо стандартизованої архітектури компактних ORM;
- спроектовану архітектуру буде повторно проаналізовано та для її тестування буде створено програмний застосунок.

2 АНАЛІЗ ІСНУЮЧИХ ПІДХОДІВ ТА МЕТОДІВ

2.1 Аналіз архітектури Java ORM систем

Перед тим як перейти до аналізу ORM систем на Java, варто зазначити що при обранні архітектури для компактних ORM на платформу .Net важливо також аналізувати ORM на Java, оскільки це дозволяє зрозуміти та порівняти різні підходи та практики в ORM-технологіях. Оскільки Java та .Net є двома провідними платформами для розробки корпоративних додатків, ідеї та рішення, які добре працюють в одній екосистемі, часто можуть бути застосовані або адаптовані для іншої. Це також може допомогти розробникам вибрати найбільш оптимальне рішення, виходячи зі зрозуміння загальних принципів ORM та їх реалізації у різних мовах програмування[3].

2.1.1 Дослідження архітектурного підходу Hibernate

Hibernate – це потужний та популярний інструмент для роботи з базами даних у Java-додатках. Він використовується для взаємодії з реляційними базами даних, дозволяючи розробникам працювати з даними у вигляді об'єктів Java, замість прямого взаємодії з SQL-запитами та таблицями баз даних. Hibernate спрощує розробку, підтримку та розширення програм, забезпечуючи високий рівень абстракції над базою даних.

Однією з ключових функцій Hibernate є відображення об'єктів Java на таблиці бази даних та навпаки. Це дозволяє розробникам працювати з даними у вигляді об'єктів, що робить роботу з базою даних більш зрозумілою та приємною.

Hibernate надає розширений інтерфейс для взаємодії з базою даних, включаючи операції створення, оновлення, видалення та читання даних. Це дозволяє розробникам виконувати операції з даними з меншим обсягом коду та складності.

Ця ORM підтримує ліниве завантаження, що означає, що дані завантажуються тільки в той момент, коли вони дійсно потрібні. Це сприяє оптимізації продуктивності додатка та зменшенню навантаження на базу даних.

Також є можливість керувати транзакціями, що дозволяє виконувати групу операцій з базою даних як одну атомарну одиницю. Це важливо для забезпечення консистентності та цілісності даних.

Система підтримує різні стратегії відображення наслідування в базі даних. Ця система дозволяє використовувати поліморфізм у програмі та зберігати дані об'єктів різних класів у одній або окремих таблицях. Hibernate використовує оптимізовані механізми для взаємодії з базою даних, що дозволяє досягти високої продуктивності навіть в великих проектах[4].

Важливою складовою системи є кешування для збереження результатів запитів та об'єктів в оперативній пам'яті, що може значно покращити продуктивність.

Великим плюсом є те – що підтримуються різні системи управління базами даних, що дозволяє розробникам легко переносити додаток між різними базами даних.

Хоча Hibernate має багато переваг, він також може бути дещо складним у налаштуванні та використанні. Розробники повинні вивчити конфігурацію, анотації та правила маппінгу, щоб ефективно використовувати цей інструмент. Однак, коли він налаштований правильно, він може значно спростити роботу з базами даних та полегшити розробку надійних та швидких Java-додатків.

У підсумку очевидно що Hibernate – це високорівневий ORM-фреймворк, який дозволяє розробникам працювати з базами даних у вигляді об'єктів Java, надаючи абстракцію над рівнем SQL. Основною метою Hibernate – є зменшення кількість повторного та складного коду, який часто потрібен при роботі з реляційними базами даних.

Архітектура Hibernate складається з наступних основних компонентів:

−sessionFactory;

−session;

−transaction;

−connectionProvider;

−transactionFactory;

−sessionFactoryConfigurator;

−query;

−criteria;

−cache.

Кожна з цих складових відіграє важливу роль у загальному механізмі ORM (див. рис. 2.1).

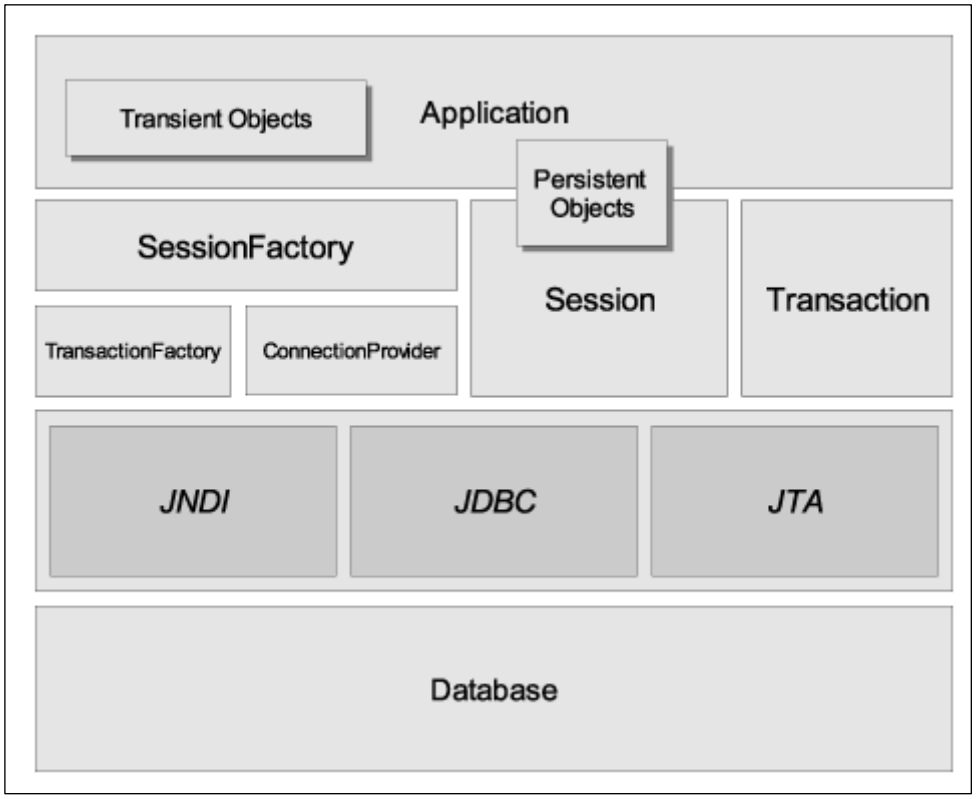

Рисунок 2.1 – архітектура Hibernate (рисунок створено самостійно)

Розглядаючи більш детально цю архітектуру нам необхідно вивести ключові по функціональності елементи щоб надалі їх можна було класифікувати для порівняння.

Session. Session є основним інтерфейсом для взаємодії між Java-додатком та базою даних. Цей інтерфейс виступає як контейнер для тривалості об'єктів та дозволяє виконувати створення, читання, оновлення та видалення (CRUD) операцій, а також запити та транзакції.

Кожен екземпляр Session створюється через SessionFactory і є прив'язаним до одного конкретного контексту тривалості. У контексті Hibernate, сесія не просто

місце для виконання операцій з базою даних, але й кеш на рівні першого рівня (first-level cache), де зберігається стан сутностей, керованих поточною сесією.

В Hibernate кожна сесія може мати одну або більше транзакцій, які представляють собою послідовність операцій з базою даних, що виконуються як одне ціле. Інтерфейс Transaction дозволяє управляти границями транзакцій на високому рівні абстракції.

Транзакції важливі для забезпечення консистентності даних, дозволяючи виконувати коміт або ролбек операцій у випадку помилки чи інших проблем. Hibernate дозволяє реалізувати ці можливості з допомогою простих методів begin, commit, та rollback.

ConnectionProvider. ConnectionProvider у Hibernate є ключовим інтерфейсом, який відповідає за управління з'єднаннями з базою даних. Цей компонент є частиною внутрішньої інфраструктури Hibernate та використовується внутрішніми механізмами для отримання доступу до бази даних.

Основна роль ConnectionProvider полягає в тому, щоб абстрагувати процес створення з'єднань з базою даних від програміста, дозволяючи Hibernate керувати пулом з'єднань та розподіляти їх по мірі необхідності. Це допомагає оптимізувати використання ресурсів, оскільки створення з'єднання з базою даних може бути ресурсоємним процесом.

Конфігурація ConnectionProvider зазвичай визначається в файлі hibernate.cfg.xml або через програмний API. Тут можна встановити різні параметри, такі як URL бази даних, ім'я користувача, пароль, максимальну кількість з'єднань у пулі, час ожидання з'єднання тощо.

ConnectionProvider може інтегруватися з зовнішніми пулами з'єднань, такими як c3p0 або HikariCP, що є стандартом де-факто у Java-світі. Пулінг з'єднань дозволяє підтримувати набір "живих" з'єднань, які можуть бути швидко передані програмі, коли вони потрібні, без необхідності створювати нове з'єднання кожного разу.

Ефективне використання цього елементу має вирішальне значення для продуктивності будь-якої програми, яка використовує Hibernate. Неправильно

сконфігурований пул з'єднань може призвести до затримок в роботі програми, витоку з'єднань або інших проблем з продуктивністю. Керування розміром пулу, часом життя з'єднань та поведінкою при витоку з'єднань є критичними аспектами, які потрібно ретельно планувати.

Також він відповідає за відновлення з'єднань у випадку їх втрати або помилок з'єднання. Сучасні пули з'єднань надають механізми для автоматичного відновлення та перепідключення, забезпечуючи стабільність додатка навіть у випадку непередбачених проблем з мережею або базою даних.

TransactionFactory. TransactionFactory в Hibernate є компонентом, який відповідає за створення та управління транзакціями в рамках сесій Hibernate. Ця фабрика є ключовою для імплементації управління транзакціями, яке є важливою частиною будь-якої системи, що забезпечує взаємодію з базою даних.

Головна функція TransactionFactory полягає в створенні об'єктів Transaction. Коли розробник викликає метод session.beginTransaction(), Hibernate через TransactionFactory створює новий екземпляр транзакції. Цей об'єкт транзакції використовується для управління життєвим циклом транзакції, включаючи її запуск, комміт та відкат (rollback).

У середовищах, де використовуються різні типи баз даних або різні стратегії управління транзакціями, TransactionFactory може бути налаштована таким чином, щоб відповідати специфічним потребам. Вона може підтримувати різні механізми транзакцій, наприклад JTA (Java Transaction API) або локальні транзакції баз даних.

У складних застосунках, де потрібна підтримка розподілених транзакцій або інтеграція з іншими ресурсами, TransactionFactory може бути налаштована для роботи з JTA. Це дозволяє Hibernate управляти транзакціями в рамках більш широкого контексту, що включає кілька різних джерел даних.

SessionFactoryConfigurator. SessionFactoryConfigurator в Hibernate є компонентом, який відповідає за налаштування та конфігурацію SessionFactory. Цей процес включає в себе аналіз метаданих сутностей, конфігураційних параметрів та властивостей з'єднання з базою даних.

Основні завдання SessionFactoryConfigurator включають визначення класів-сутностей, які будуть керуватися Hibernate, налаштування відносин між цими сутностями, а також встановлення параметрів кешування, логування та інших аспектів процесу взаємодії з базою даних.

Однією з ключових ролей SessionFactoryConfigurator є встановлення мапінгу між класами Java та таблицями бази даних. Це може включати в себе визначення анотацій на рівні класів або використання XML-файлів для мапінгу, забезпечуючи гнучкість та масштабованість в процесі розробки.

Також важливою функцією є налаштування параметрів підключення до бази даних, включаючи URL, ім'я користувача, пароль, властивості діалекту SQL та інші параметри, які забезпечують з'єднання та ефективну взаємодію з базою даних.

2.1.2 Дослідження архітектурного підходу ORMLite

ORMLite, як випливає з назви, є "легким" ORM (Object-Relational Mapping) фреймворком, призначеним для спрощення взаємодії між Java-об'єктами та реляційними базами даних. Цей фреймворк був розроблений з метою надати простішу та більш прямолінійну альтернативу більш великим і комплексним ORM-рішенням, таким як Hibernate.

Використання ORMLite дозволяє розробникам швидко мапувати Java-класи на таблиці бази даних, роблячи процес взаємодії з базою даних більш інтуїтивно зрозумілим і менш витратним з точки зору написання коду. Він підходить для розробників, які шукають простий у використанні фреймворк без необхідності заглиблюватися в складні конфігурації та розширену функціональність більш великих ORM-систем.

ORMLite підтримує ряд основних баз даних, таких як MySQL, PostgreSQL, Microsoft SQL Server, H2, HSQLDB та інші. Однією з його ключових особливостей є легкість інтеграції, особливо у проектах, де обсяг даних та вимоги до управління даними не є надто великими.

У цілому, ORMLite є відмінним вибором для проектів, де потрібна простота, швидкість розробки та менший обсяг вимог до управління даними. Це рішення

дозволяє розробникам залишатися продуктивними, зосереджуючись на бізнес-логіці, а не на технічних деталях взаємодії з базами даних.

Основні архітектурні елементи ORMLite включають:

– dao (Data Access Object);

– databaseTable;

– databaseField;

– queryBuilder;

– updateBuilder;

– connectionSource.

Далі проаналізуємо призначення кожного із цих елементів.

Dao (Data Access Object). Dao (Data Access Object) в ORMLite є фундаментальним інтерфейсом, який забезпечує абстракцію доступу до бази даних. Він дозволяє виконувати всі основні операції з базою даних для конкретного класу Java, такі як створення, читання, оновлення та видалення (CRUD).

Кожен Dao асоційований з певним класом, що представляє сутність бази даних, і містить методи для роботи з цією сутністю. Наприклад, якщо у вас є клас User, то ви створите UserDao для управління об'єктами User в базі даних.

Dao забезпечує високий рівень абстракції, що дозволяє розробникам працювати з об'єктами, не замислюючись про низькорівневі запити SQL. Особливість Dao в ORMLite полягає в тому, що його можна легко розширювати або налаштовувати, що робить його гнучким інструментом для вирішення різноманітних задач управління даними.

DatabaseTable. Анотація DatabaseTable в ORMLite використовується для позначення класу, який має бути відображений як таблиця в базі даних. Це ключова частина механізму ORM, оскільки вона дозволяє фреймворку ідентифікувати, які класи є сутностями бази даних і як їх слід обробляти.

За допомогою анотації DatabaseTable можна вказати назву таблиці, якщо вона відрізняється від назви класу. Також можна використовувати різні параметри анотації для налаштування поведінки таблиці, наприклад, вказати кастомні DAO класи.

Ця анотація спрощує процес мапінгу, оскільки вам не потрібно писати окремі SQL-скрипти для створення таблиць; ORMLite автоматично генерує необхідну структуру на основі класів і їх анотацій.

DatabaseField. DatabaseField – це анотація в ORMLite, яка використовується для вказівки, що поле класу має відповідати стовпцю в таблиці бази даних. Ця анотація дозволяє детально налаштовувати як кожне поле класу буде відображатися в базі даних, включаючи тип даних, індексацію, унікальність та інші властивості стовпців.

За допомогою DatabaseField можна вказати, чи поле є ідентифікатором (ID), чи має використовувати автоматичне генерування значення (autoincrement), встановити обмеження на стовпці, такі як not null, та інші SQL-обмеження. Це надає розробникам гнучкість у визначенні структури даних, забезпечуючи при цьому сильне зв'язування між об'єктною моделлю Java та схемою бази даних.

Використання DatabaseField разом з DatabaseTable створює повноцінну картину того, як об'єкти Java відображаються на реляційні таблиці, забезпечуючи ефективну та легкозрозумілу ORM-систему.

QueryBuilder. QueryBuilder у ORMLite – це потужний інструмент для створення складних SQL-запитів за допомогою об'єктно-орієнтованого підходу. Він дозволяє розробникам формувати запити, не пишучи вручну SQL-код, тим самим забезпечуючи більшу безпеку і зручність у роботі з даними.

З використанням QueryBuilder, можна легко фільтрувати дані, використовуючи різні критерії, з'єднувати таблиці, групувати результати, сортувати їх і так далі. Це робиться за допомогою ланцюжка викликів методів, кожен з яких додає певну частину до кінцевого запиту. Наприклад, можна використовувати методи where(), orderBy(), join(), щоб створити точно визначений запит.

QueryBuilder також підтримує побудову підзапитів та використання агрегатних функцій, що робить його дуже гнучким для створення різноманітних запитів до бази даних. Після того, як запит сформовано, QueryBuilder може

виконати його та повернути результати в формі об'єктів Java, автоматично здійснюючи мапінг даних з таблиць на об'єкти.

UpdateBuilder. UpdateBuilder в ORMLite використовується для побудови та виконання запитів на оновлення даних у базі даних. Цей інструмент дозволяє вносити зміни в записи, використовуючи об'єктно-орієнтований підхід, аналогічно до QueryBuilder.

За допомогою UpdateBuilder, розробники можуть вказувати, які поля і яким чином потрібно оновити, задавати умови для вибору записів, які підлягають оновленню, та використовувати різні SQL-операції, такі як інкрементування значень або встановлення нових даних.

UpdateBuilder включає методи для встановлення значень полів (updateColumnValue, updateColumnExpression) та умов оновлення (where). Після конфігурації запиту, його можна виконати для оновлення відповідних записів у базі даних. Цей інструмент є незамінним для ефективного управління даними, дозволяючи здійснювати оновлення без прямого написання SQL-запитів.

2.2 Аналіз архітектури ORM систем на платформі .Net

2.2.1 Дослідження архітектурного підходу EF Core

Entity Framework Core (EF Core) – це сучасна, компактна, розширювана та крос-платформенна версія Entity Framework, яка є ORM (Object-Relational Mapping) фреймворком від Microsoft. EF Core дозволяє розробникам працювати з даними у базах даних, використовуючи об'єкти .NET, тим самим зменшуючи необхідність вручну писати SQL код. Це рішення спрощує процес створення, читання, оновлення та видалення даних, автоматично мапуючи запити між об'єктною моделлю програми та схемою бази даних. EF Core підтримує різні бази даних та є популярним вибором для розробки додатків на платформі .NET, зокрема завдяки його гнучкості, продуктивності та зручності використання.

Далі наведено основні архітектурні елементи Entity Framework Core:

– dbContext;

– dbSet;

– modelBuilder;

– changeTracker;

– database provider;

– migrations;

– query types;

– entity types;

– LINQ Queries;

– data annotations and fluent API.

Схема взаємодії та розташування архітектурних елементів наведено на рисунку 2.2.

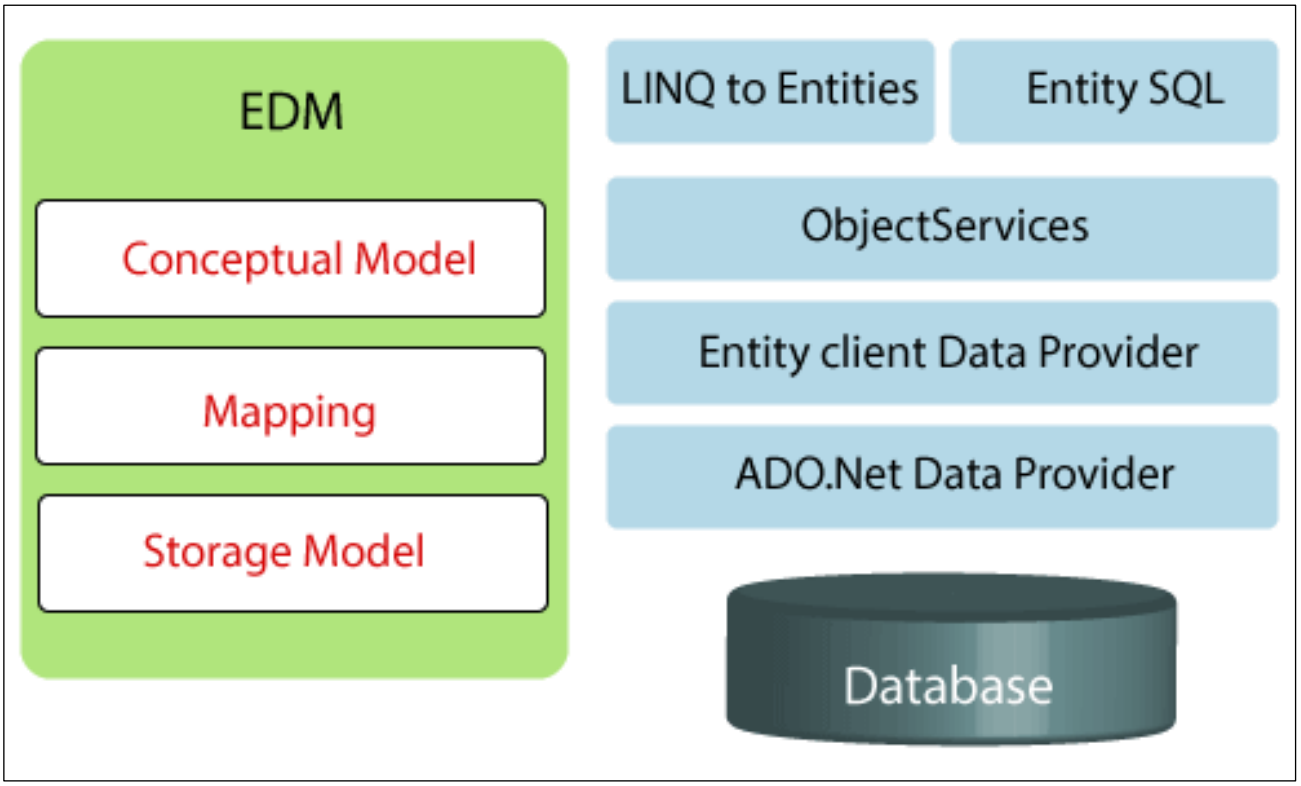

Рисунок 2.2 – Схема архітектури Entity Framework Core (рисунок створено самостійно)

Проаналізуємо усі ключові елементи архітектури Entity Framework Core.

DbContext. DbContext у Entity Framework Core виступає як основний клас, який спрощує взаємодію між вашим кодом та базою даних. Він представляє сесію з базою даних, дозволяючи запитувати та зберігати дані. В контексті EF Core, DbContext відіграє кілька ключових ролей: він зберігає інформацію про стан

сутностей, управляє з'єднаннями з базою даних, а також налаштовує модель даних і відносини між об'єктами[5].

Коли використовується DbContext, визначається властивості, які представляють колекції сутностей, пов'язаних з таблицями у базі даних. Ці властивості типу DbSet дозволяють вам виконувати операції CRUD (створення, читання, оновлення, видалення) на даних. DbContext також відповідає за відстеження змін у сутностях, що дозволяє EF Core автоматично генерувати відповідні SQL-операції при збереженні змін у базі даних.

DbSet. DbSet представляє колекцію всіх сутностей в контексті, що він у Entity Framework Core представляє колекцію сутностей певного типу, що відповідає таблиці в базі даних. Кожен DbSet в DbContext виступає як вхідна точка для запитів, які працюють з конкретним типом сутності, і дозволяє виконувати операції створення, читання, оновлення та видалення (CRUD) даних. Використання DbSet дозволяє розробникам взаємодіяти з даними на високому рівні абстракції, значно спрощуючи роботу з базою даних.

Коли виконуються операції через DbSet, Entity Framework Core автоматично перетворює ці дії на відповідні SQL-запити. Наприклад, додавання нової сутності до DbSet призводить до створення нового запису в таблиці бази даних після виклику методу SaveChanges() в DbContext. Також DbSet використовується для виконання запитів LINQ, які дають можливість фільтрувати, упорядковувати та групувати дані на стороні сервера.

ModelBuilder. ModelBuilder у Entity Framework Core використовується для налаштування моделі даних, яка визначає як ваші класи сутностей будуть відображені на базу даних. Цей клас використовується всередині OnModelCreating методу в DbContext для конфігурації сутностей, відносин між ними, ключів, індексів, обмежень та інших аспектів схеми бази даних.

З ModelBuilder розробники можуть детально визначати, як сутності мапляться на таблиці та стовпці бази даних, включно з використанням різних стратегій іменування, типів даних і т.д. Він дозволяє застосовувати конфігурації за допомогою Fluent API, яка надає більше можливостей та гнучкості порівняно з

анотаціями. Наприклад, за допомогою ModelBuilder можна визначити складні відносини, такі як один-до-багатьох або багато-до-багатьох, налаштовувати каскадне видалення, та багато іншого.

Загалом, ModelBuilder надає розробникам детальний контроль над тим, як об'єктна модель додатку відображається на реляційну модель бази даних, дозволяючи їм створювати ефективні, гнучкі та оптимізовані рішення для роботи з даними.

ChangeTracker. ChangeTracker у Entity Framework Core відіграє ключову роль у відстеженні змін, які вносяться в сутності під час їхнього життєвого циклу. Коли сутність отримується з бази даних, ChangeTracker відслідковує будь-які зміни, які згодом вносяться в цю сутність. При збереженні змін через DbContext, ChangeTracker аналізує всі відстежувані сутності та визначає, які з них були змінені. На основі цього визначення EF Core генерує відповідні SQL-операції для оновлення бази даних.

Цей компонент дозволяє розробникам оптимізувати взаємодію з базою даних, оскільки забезпечує збереження лише тих змін, які дійсно були зроблені. Він також надає розширений API для перевірки та управління станом сутностей у контексті.

Database Provider. У Entity Framework Core, Database Provider визначає, як саме EF Core буде взаємодіяти з конкретною базою даних. Різні провайдери існують для різних типів баз даних, таких як Microsoft SQL Server, PostgreSQL, MySQL, SQLite та інші. Кожен провайдер адаптує функціональність EF Core для специфіки конкретної системи управління базами даних.

Вибір відповідного провайдера є критично важливим, оскільки він впливає на спосіб виконання запитів, обробки даних та використання специфічних для бази даних функцій. Провайдери забезпечують мост між загальною логікою EF Core та конкретними особливостями кожної СУБД.

Migrations. Migrations у Entity Framework Core – це механізм, який дозволяє розробникам управляти змінами в схемі бази даних в процесі розвитку додатку. За

допомогою міграцій можна визначати, застосовувати та відкатувати зміни в схемі, такі як додавання або видалення таблиць, стовпців, обмежень тощо.

Коли розробник змінює модель даних, він може генерувати нову міграцію, яка містить код для оновлення схеми бази даних. Ці міграції дають можливість контролювати еволюцію бази даних, забезпечуючи її синхронізацію з моделлю даних у додатку.

Query Types. Query Types у EF Core – це функціональність, що дозволяє використовувати класи для читання даних, які не потрібно відстежувати або оновлювати. Ці типи запитів часто використовуються для роботи з табличними даними, які не відповідають сутностям у вашому додатку, наприклад, для запитів до результатів зіставлення або виконання запитів до SQL-виразів.

Використання Query Types дозволяє розробникам використовувати потужність LINQ для створення складних запитів до цих даних, але без створення сутностей, які потім потребують відстеження або управління змінами. Це корисно для оптимізації продуктивності та управління доступом до "тільки для читання" даних.

У висновку, варто зазначити що EF Core забезпечує широкий спектр функціональностей, включаючи підтримку складних запитів, міграцій, великого обсягу конфігурацій через Fluent API та Data Annotations, що робить його відмінним вибором для великих та складних додатків. Однак, ці особливості можуть бути надлишковими в контексті компактних застосунків, де основними вимогами є простота, легкість впровадження та мінімальне навантаження на продуктивність.

2.2.2 Дослідження архітектурного підходу PetaPoco

PetaPoco – це компактний ORM (Object-Relational Mapping) фреймворк для .NET, розроблений з метою простоти та швидкості. Він дозволяє розробникам здійснювати взаємодію між об'єктами .NET та реляційними базами даних без потреби в ручному написанні SQL-коду або складних конфігураціях. Хоча PetaPoco надає базовий функціонал ORM, він відрізняється своєю компактністю та

ефективністю, що робить його ідеальним для проектів, яким не потрібні всі можливості повнофункціональних ORM-систем.

PetaPoco розроблений як мікро-ORM, що означає, що він зосереджений на виконанні основних завдань з мінімальними вимогами до ресурсів та залежностей. Він підтримує POCO (Plain Old CLR Objects) для визначення моделей даних, що дозволяє розробникам використовувати звичайні класи .NET для представлення даних з бази. PetaPoco дозволяє швидко виконувати запити до бази даних та ефективно мапити результати на об'єкти.

Ще однією важливою особливістю PetaPoco є його підтримка SQL-блоків, які дозволяють динамічно створювати запити. Це допомагає уникнути жорсткого кодування SQL-запитів та забезпечує більшу гнучкість при роботі з даними[6].

Основні архітектурні елементи PetaPoco(див. рис. 2.3), компактний ORM для .NET, включають:

– database;

– poco;

– sql Builder;

– transaction;

– IMapper;

– IProvider;

– commandExecuted and CommandExecuting events.

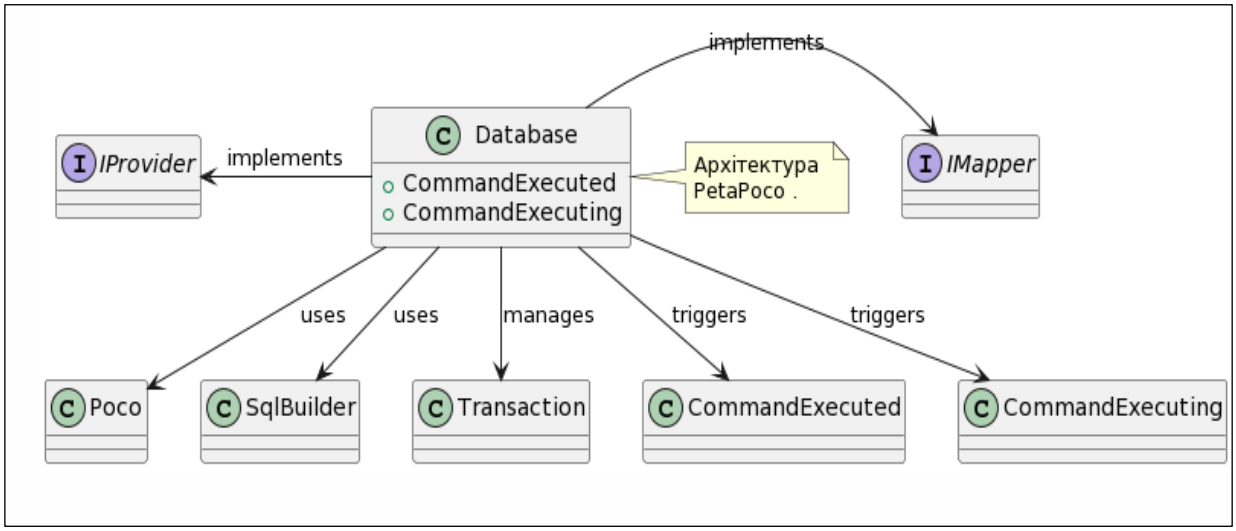

Рисунок 2.3 – Архітектура ORM PetaPoco (рисунок створено самостійно)

Розберемо більш детально архітектуру цієї ORM, та виведемо ключові елементи які ідентифікують її як компактної та швидкої ORM.

Database. Це основний клас у PetaPoco, що використовується для взаємодії з базою даних. Він надає методи для виконання SQL-запитів, управління транзакціями та мапінгу результатів на POCO об'єкти. Клас Database відіграє центральну роль в PetaPoco, об'єднуючи більшість функцій ORM.

Poco. У PetaPoco POCO (Plain Old CLR Object) використовуються для представлення даних, що витягуються з бази даних. Ці класи є простими і не мають залежностей від ORM, забезпечуючи легкість і гнучкість у роботі з даними.

Sql Builder. Це інструмент для динамічного конструювання SQL-запитів. Sql Builder дозволяє розробникам легко складати запити, не пишучи вручну рядки SQL, тим самим знижуючи ризик помилок і спрощуючи процес написання коду.

Transaction. У PetaPoco клас Database підтримує методи для управління транзакціями. Це дозволяє розробникам контролювати виконання операцій з даними, забезпечуючи консистентність та надійність в обробці даних.

IMapper. Цей інтерфейс використовується для кастомізації процесу мапінгу між базою даних і POCO класами. Він дозволяє розробникам визначити власні правила для відображення таблиць та стовпців на об'єкти.

IProvider. IProvider в PetaPoco визначає інтерфейс для специфічних для бази даних провайдерів. Це дозволяє PetaPoco підтримувати різні типи баз даних, надаючи специфічну для кожної СУБД функціональність.

CommandExecuted and CommandExecuting events. CommandExecuted and CommandExecuting events події дозволяють розробникам втручатися в процес виконання SQL-команд, що може бути корисно для логування, моніторингу або зміни поведінки виконання SQL-команд.

Ці елементи разом становлять гнучку та компактну архітектуру PetaPoco, надаючи розробникам потужні інструменти для роботи з базами даних, не вдаючись у складності та навантаження повнофункціональних ORM-систем[7].

3 ПРОЕКТУВАННЯ ТА МОДИФІКАЦІЯ АРХІТЕКТУРНОГО ПІДХОДУ

3.1 Виведення закономірності у порівнянні EF Core та PetaPoco

Щоб проаналізувати архітектурні особливості та вивести симтоматичність застосованих архітектурних підходів – нам необхідно проаналізувати EF Core та PetaPoco, у кінці визначити які практики, підходи та архітектурні елементи необхідні або бажані для розробки компактних та швидких ORM[8].

Надалі буде використовуватися метод з’єднання вибірок архітектурних та структурних елементів, на їх основі можна буде наглядно побачити та розділити спільні та відмінні даних у різних вибірках.

Спочатку давайте розглянемо спільні аспекти в архітектурних елементах Entity Framework Core і PetaPoco, які відображають схожі підходи або вирішують аналогічні завдання.

DbContext (EF Core) і Database (PetaPoco).

Обидва ці елементи служать як основні точки взаємодії між програмою і базою даних. Вони керують з'єднаннями, сесіями та операціями з даними.

DbSet (EF Core) і Poco (PetaPoco).

Ці елементи відіграють роль відображення об'єктів у базі даних. У EF Core DbSet представляє колекцію сутностей, тоді як у PetaPoco Poco використовується для мапінгу об'єктів на таблиці.

ModelBuilder (EF Core) і Sql Builder (PetaPoco).

Хоча ці інструменти служать різним цілям, обидва вони використовуються для побудови та налаштування запитів. ModelBuilder в EF Core використовується для налаштування моделей даних, тоді як Sql Builder в PetaPoco допомагає створювати SQL-запити.

Migrations (EF Core) і Transaction (PetaPoco).

Ці елементи використовуються для управління змінами. Migrations в EF Core управляють змінами схеми бази даних, тоді як Transaction в PetaPoco керує транзакційною логікою.

LINQ Queries (EF Core) і CommandExecuting events (PetaPoco).

Обидва фреймворки пропонують підтримку для формування запитів, хоча в різних формах. LINQ Queries дозволяє писати запити на високому рівні абстракції в EF Core, тоді як CommandExecuting events в PetaPoco дозволяють кастомізувати логіку виконання запитів.

Далі необхідно виявити відмінність архітектурних патернів більш компактної ORM.

IMapper та IProvider.

У більших ORM системах, мапінг між об'єктами та базою даних часто є більш автоматизованим і включає більш складні можливості, такі як налаштування відносин, lazy loading та інші ORM-функції[9]. IMapper у PetaPoco є більш простим і гнучким, але не надає такого рівня автоматизації та інтеграції.

IProvider в PetaPoco використовується для налаштування поведінки специфічної для конкретної бази даних, що є корисним у компактній ORM. У більших ORM системах такі налаштування зазвичай є частиною більш комплексного механізму конфігурації і провайдерів.

Sql Builder.

Sql Builder у PetaPoco дозволяє детально керувати створенням SQL-запитів, що важливо в системах, де не потрібен повний спектр ORM-функцій. У повнофункціональних ORM, як EF Core, подібний функціонал часто вбудований в більш високорівневі абстракції.

CommandExecuted та CommandExecuting events.

Ці події в PetaPoco дозволяють більш тонкий контроль над процесом виконання SQL-команд, що може бути не так необхідно у складніших ORM, де є більш розвинені механізми для логування, аудиту та дебагінгу.

Компактні транзакції.

У повнофункціональних ORM, управління транзакціями часто інтегроване з іншими аспектами системи, такими як управління сесіями та кешуванням. компактні транзакції у PetaPoco простіші та прямолінійніші, що є важливим у контексті компактного ORM. Візуальний свот-аналіз перетинів та закономірностей порівняння цих двох ORM зображено на рисунку 3.1.

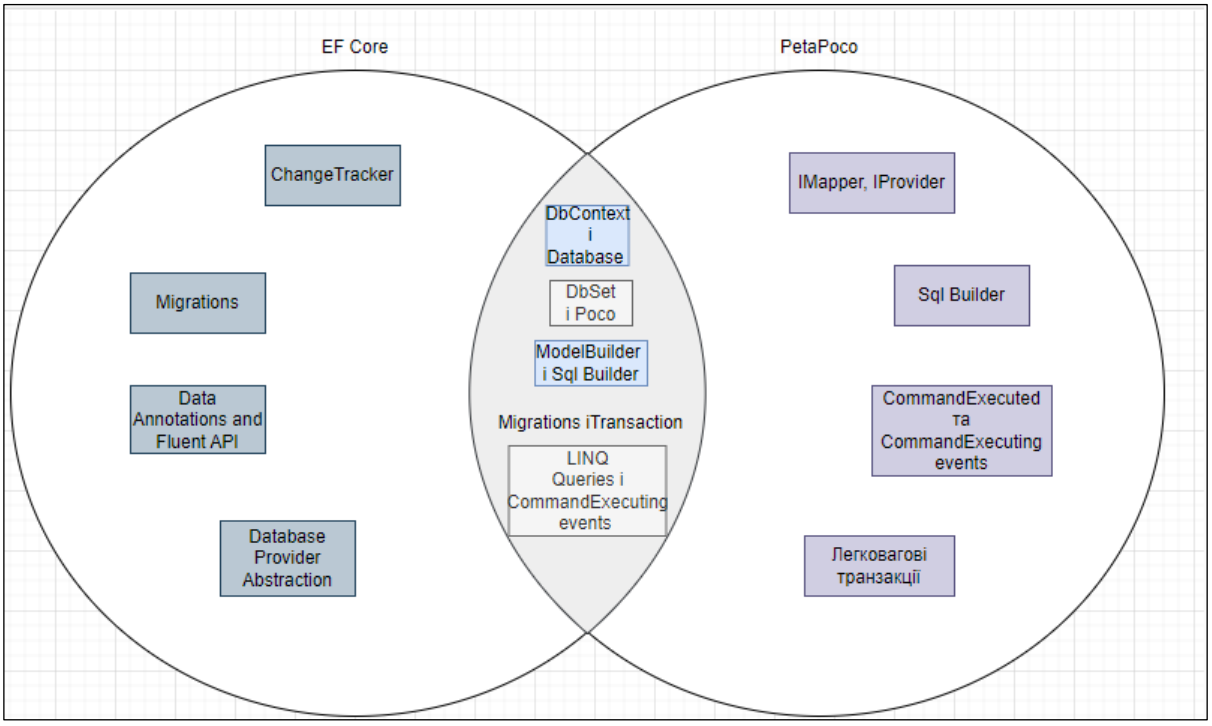

Рисунок 3.1 – Схема перетинаннь ORM PetaPoco та EF Core (рисунок створено самостійно)

Отже, ці елементи і підходи в PetaPoco вирішують завдання, які є центральними для компактних ORM, але можуть бути недостатньо комплексними або занадто простими для повнофункціональних ORM, які вимагають більш розширеного управління даними та відносинами.

Ці структурні патерни разом створюють функціональну основу для компактних ORM, вони роблять їх підходящим для проектів, де необхідний простий у реалізації та ефективний доступ до даних.

3.2 Підходи до створення міграції

Механізм міграції на основі версіонування схеми бази даних є фундаментальним підходом в розробці програмного забезпечення, особливо коли йдеться про управління змінами в структурі бази даних протягом життєвого циклу додатка[10]. Ця концепція дозволяє систематично вносити та відстежувати зміни у базі даних, забезпечуючи її сумісність з актуальною версією додатка. Розглянемо детально ключові аспекти цього механізму:

- версії міграцій. Кожна міграція асоціюється з унікальною версією (часто це послідовний номер або дата з часом). Версії дозволяють визначити порядок застосування міграцій та відстежувати історію змін схеми;
- міграційні скрипти. Для кожної версії створюється міграційний скрипт, який описує зміни в схемі бази даних, такі як додавання нових таблиць, зміна структури існуючих таблиць, додавання індексів тощо. Ці скрипти можуть застосовуватися автоматично або вручну;
- таблиця версій у базі даних. Система веде спеціальну таблицю у базі даних, де зберігається інформація про застосовані міграції. Це дозволяє системі визначити, які міграції вже були виконані та які ще потрібно застосувати;
- автоматизація застосування міграцій. При розгортанні або оновленні додатка система автоматично перевіряє версію схеми бази даних та застосовує необхідні міграційні скрипти для приведення бази даних у відповідність до поточної версії додатка;
- контроль версій. Забезпечує чітке розуміння змін в структурі бази даних та їх відповідності до версій додатка;
- послідовність застосування змін. Гарантує, що міграції застосовуються в правильному порядку, уникаючи конфліктів або втрати даних;
- відкат змін. Можливість відкоту до попередньої версії схеми у випадку виявлення проблем з новою версією;
- автоматизація. Мінімізація ручної роботи та помилок при розгортанні та оновленні додатка.

Реалізація механізму версіонування схеми може відрізнятися в залежності від конкретної ORM системи та мови програмування. Втім, більшість сучасних інструментів та фреймворків (наприклад, Django для Python, ActiveRecord для Ruby on Rails, Liquibase або Flyway для Java) надають вбудовані засоби для управління міграціями на основі версіонування схеми, спрощуючи цей процес.

Деталі архітектурної побудови версіонування зображено на рисунку 3.2.

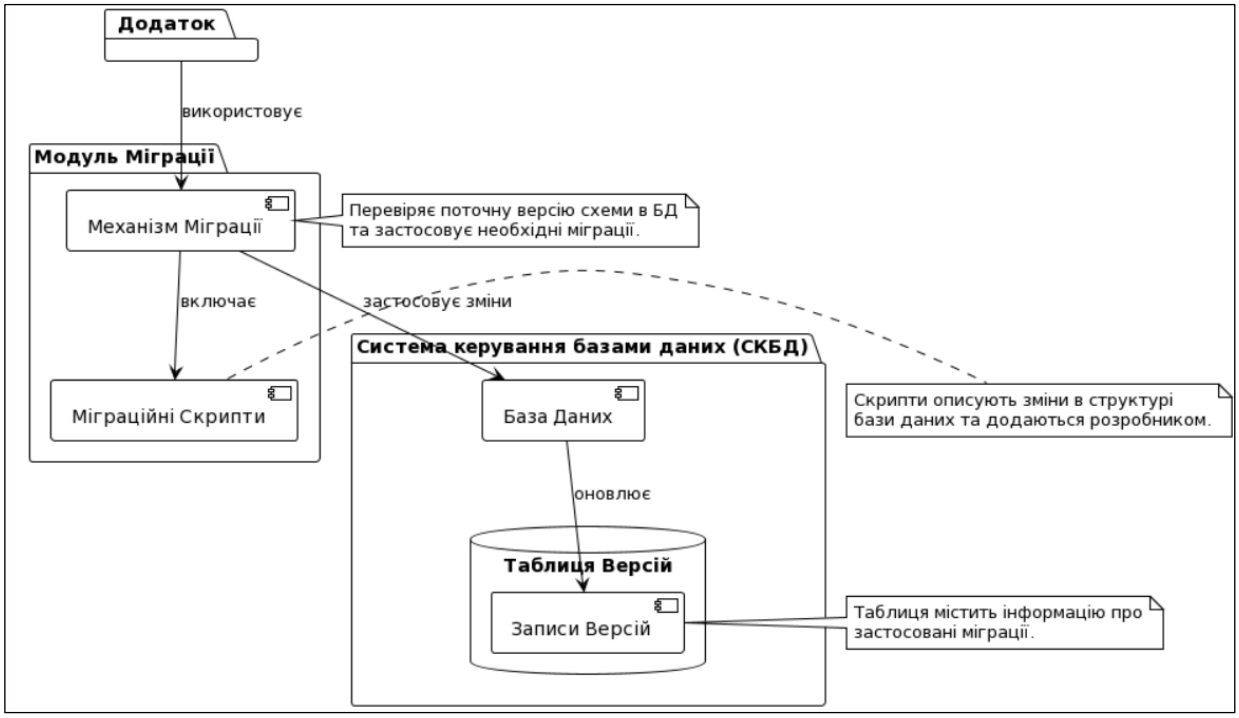

Рисунок 3.2 – Архітектура версіонування (рисунок створено самостійно)

Додаток – основне програмне забезпечення, яке взаємодіє з базою даних. Система керування базами даних (СКБД) – включає саму Базу Даних та Таблицю Версій, де зберігається інформація про застосовані міграції.

Модуль Міграції – містить Механізм Міграції для керування процесом застосування міграцій та Міграційні Скрипти, які описують зміни в схемі бази даних.

Взаємодія між цими компонентами показана через з'єднання, зокрема як Механізм Міграції застосовує зміни в Базу Даних та оновлює Записи Версій у Таблиці Версій.

Ця діаграма служить високорівневим описом процесу версіонування схеми бази даних і механізму міграції в контексті кастомної ORM системи.

Міграційні скрипти слугують основою для керування змінами в структурі бази даних у процесі розвитку програмного забезпечення. Вони визначають послідовність операцій, необхідних для модифікації схеми бази даних, таких як створення або видалення таблиць, зміна структури існуючих таблиць (додавання, видалення чи модифікація стовпців), створення або зміна обмежень (наприклад, зовнішні ключі), а також індексів для оптимізації запитів до бази даних.

Ключовою характеристикою міграційних скриптів є їх версіонування, що дозволяє відстежувати порядок застосування змін і забезпечує можливість відкату до попереднього стану в разі виявлення помилок або необхідності повернення до більш ранньої версії схеми бази даних. Кожен міграційний скрипт повинен мати унікальний ідентифікатор (часто це дата та час створення або послідовний номер), який використовується для контролю за порядком застосування міграцій.

Міграційні скрипти повинні бути ідемпотентними, тобто повторне їх застосування не повинно призводити до змін у базі даних після першого успішного виконання. Це гарантує, що система залишиться у консистентному стані незалежно від кількості спроб застосування міграцій.

Міграції зазвичай застосовуються автоматично під час розгортання нової версії додатку або оновлення існуючої. Багато сучасних фреймворків та ORM надають інструменти для генерації шаблонів міграційних скриптів, що спрощує процес їх створення. Також, деякі системи дозволяють автоматично генерувати міграційні скрипти на основі змін, внесених у моделі даних.

Важливим аспектом роботи з міграціями є наявність детальної документації та засобів моніторингу застосування міграцій, що дозволяє команді розробників мати чітке уявлення про поточний стан схеми бази даних і історію її змін.

4 ПРОВЕДЕННЯ ЕКСПЕРИМЕНТУ

4.1 Визначення фінального архітектурного складу ORM

Для початку розглянемо аналітику щодо фундаментальних складових ORM систем та їх складові.

Основний функціонал ORM систем часто описується в різних джерелах, які зосереджуються на тому, як ці системи спрощують роботу з базами даних для розробників. Ось кілька ключових функцій, які є фундаментальними для більшості ORM систем:

- мапінг об'єктів і таблиць. ORM дозволяє відображати об'єкти в програмному коді на таблиці в базі даних, що робить взаємодію з базою даних більш інтуїтивно зрозумілою для розробників, що працюють з об'єктно-орієнтованими мовами;
- автоматизація операцій CRUD. Створення, читання, оновлення та видалення (crud) даних в базі даних можуть бути автоматизовані за допомогою методів, які визначаються на рівні об'єктів, що робить код менш заплутаним і більш переносними;
- керування сесіями та транзакціями. Більшість orm інструментів надають механізми для управління сесіями з'єднань і транзакціями, що забезпечує більшу надійність і цілісність даних при взаємодії з базою даних;
- оптимізація запитів. ORM можуть оптимізувати запити до бази даних, що дозволяє підвищити продуктивність застосунків, особливо коли мова йде про складні запити, які вимагають обробки великих обсягів даних;
- кешування. Для підвищення продуктивності деякі orm системи включають механізми кешування, що дозволяє зберігати результати запитів для швидкого доступу.

Діаграма Вена (див. рис. 3.1), яку було розглянуто вище, зображує спільні та відмінні риси між EF Core та PetaPoco, двома системами Object-Relational Mapping (ORM). Із цієї діаграми можна виділити наступні елементи, які є спільними для обох систем і важливими для базової ORM системи:

- dbcontext і database - обидві системи використовують центральні класи для управління базою даних та контекстом даних. це ключовий компонент для координації роботи orm системи з базою даних;
- dbset і poco - вони представляють колекції сутностей, з якими можна працювати як з об'єктами, що дозволяє здійснювати операції crud (створення, читання, оновлення, видалення);
- modelbuilder і sql builder - ці інструменти дозволяють налаштовувати моделі даних і будувати sql-запити, відповідно, що забезпечує гнучкість в управлінні структурою бази даних та оптимізацію запитів;
- migrations і transaction - підтримка міграцій та транзакцій забезпечує можливість контролювати зміни в структурі бази даних та виконувати групу операцій як одну логічну одиницю;
- linq queries і commandexecuting events - можливість використання linq для формування запитів та події, пов'язані з виконанням команд, надають додатковий контроль над обробкою даних та їхньою взаємодією.

Ці особливості формують базовий набір вимог до ORM системи, які включають управління моделями даних, взаємодію з базою даних через абстрактні об'єкти, оптимізацію запитів, управління змінами і транзакціями та розширені можливості запитів. Такий підхід дозволяє розробникам зосередитись на логіці додатку, мінімізуючи ручне втручання в управління даними(див. рис. 4.1).

Пояснення базової архітектури:

- dbcontext є центральним класом, який управляє контекстом даних і базою даних;
- dbset використовується для управління набором сутностей в контексті (dbset містить poco);
- poco представляє сутність даних;
- modelbuilder використовується для конфігурації моделі та визначення взаємодій між класами і таблицями бази даних;
- sqlbuilder будує sql-запити для modelbuilder;
- migrations відповідає за управління змінами в схемі бази даних;

– transaction управляє транзакціями бази даних;

– linq queries використовується для формування запитів до бази даних через методи linq;

– commandexecuting events обробляє події, пов'язані з виконанням команд.

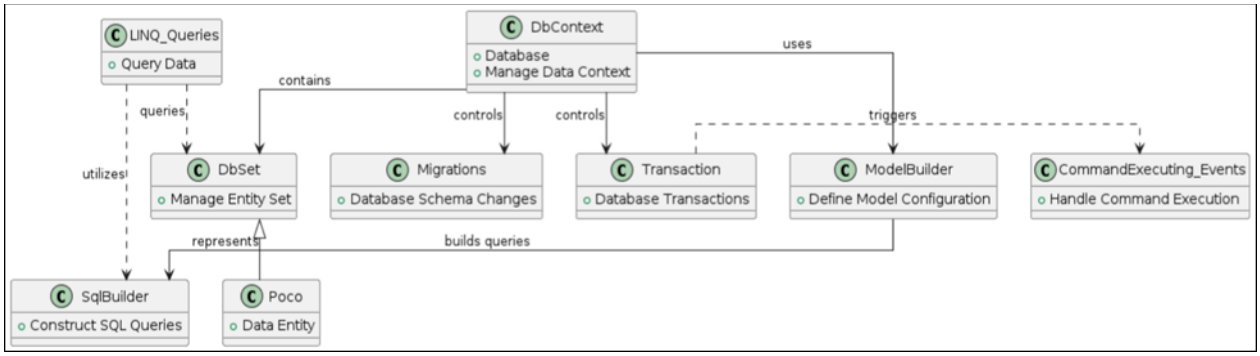

Рисунок 4.1 – Базова архітектура ORM (рисунок створено самостійно)

4.2 Програмна реалізація

Для проведення тестування працездатності встановленої архітектури – необхідно розробити відносно невелику ORM систему, яка б реалізувала всі елементи обрані раніше та стратегії використання, таким чином додатково буде створено стандартизовану архітектуру для розробки подібних систем.

Нижче буде описано файлову структуру, пояснено діаграму класів та розроблено журнал розробки із рекомендаціями, проблемами та вирішеннями, в кінці систему буде протестовано для підтвердження працездатності.

Отже нижче наведено приклад файлової системи нашого TinyORM додатку(див. рис. 4.2).

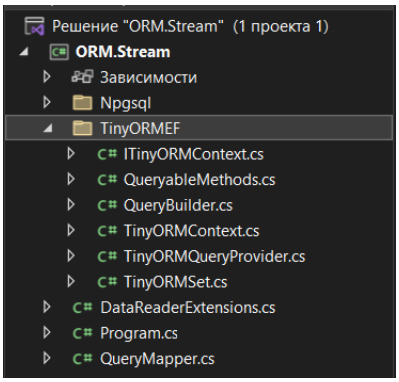

Рисунок 4.2 – Зображення файлової системи (рисунок створено самостійно)

Кореневий каталог має наступні елементи:

- TinyORMContext.cs. Визначає клас TinyORMContext, який є центральним контекстом для керування базою даних. Цей клас відповідає за створення сетів (TinyORMSet<T>) для різних типів сутностей та управління з'єднаннями з базою даних;
- QueryableMethods.cs. Ймовірно містить допоміжні методи для роботи з LINQ-запитами або розширення функціональності TinyORMQueryable<T>;
- QueryBuilder.cs. Забезпечує клас QueryBuilder, що використовується для побудови SQL-запитів на основі виразів, які вводяться через LINQ;
- TinyORMQueryProvider.cs: Визначає клас TinyORMQueryProvider, який реалізує IQueryProvider та відповідає за перетворення LINQ-виразів у SQL-запити;
- TinyORMSet.cs. Визначає клас TinyORMSet<T>, який представляє колекцію об'єктів певного типу і дозволяє виконувати на них операції CRUD;
- Program.cs. Входовий файл програми, який містить Main метод для запуску додатка;
- QueryMapper.cs. Цей клас може використовуватися для відображення результатів SQL-запитів у сутності або для трансформації SQL-результатів у об'єкти;
- DataReaderExtensions.cs. Може містити розширення для класу IDataReader, яке дозволяє простіше маппінг даних з бази у класи.

Далі буде розглянуто цікаві та корисні структурні елементи які були розроблені та будуть виокремлені рекомендації щодо застосування методів.

4.2.1 Розгляд маппінгу в системі

Було досліджено та розглянуто різні методи мапінгу для їх імпліментації у наш застосунок.

Насамперед, тестувалися стандартні підходи, які зазвичай використовуються в розробці на .NET, включаючи методи, засновані на делегатах, які є одними з

найбільш ефективних з точки зору продуктивності. Це дозволяє виміряти базову швидкість обробки без використання додаткових бібліотек чи інструментів.

Додатково було досліджено динамічні методи за допомогою DynamicMethod, які надають більшу гнучкість шляхом динамічного створення коду в рантаймі. Ці методи особливо корисні в ситуаціях, де стандартні підходи не ефективні або не підходять через специфіку даних чи вимог до обробки.

Також були розглянуті підходи на базі виразів, зокрема CompiledExpression та FastCompiledExpression. Вони використовують можливості .NET для компіляції лямбда-виразів в делегати на льоту, що може забезпечити кращу продуктивність порівняно з інтерпретацією цих виразів у рантаймі. Ці методи особливо підходять для складних мапінгів, де потрібна висока продуктивність.

Разом з тим, оцінено методи, які базуються на рефлексії – SlowReflection та FastReflection. Ці методи дозволяють здійснювати мапінг за допомогою рефлексії, що є менш продуктивним, але надзвичайно гнучким підходом, який може бути необхідним у деяких випадках. Результати тестування наведено на рисунку 4.3.

Консоль отладки Microsoft Visual Studio

```
[Host]             : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2 [AttachedDebugger]
.NET 6.0           : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2
.NET 7.0           : .NET 7.0.5 (7.0.523.17405), X64 RyuJIT AVX2
.NET Framework 4.8 : .NET Framework 4.8.1 (4.8.9241.0), X64 RyuJIT VectorSize=256
```

Method	Job	Runtime	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen0	Allocated	Alloc Ratio
Baseline	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
Delegate	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
DynamicMethod	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
CompiledExpression	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
FastCompiledExpression	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
SlowReflection	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
FastReflection	.NET 5.0	.NET 5.0	NA	NA	NA	NA	?	?	-	-	?
Baseline	.NET 6.0	.NET 6.0	6.654 ns	0.2013 ns	0.2067 ns	6.557 ns	0.81	0.05	0.0134	56 B	1.00
Delegate	.NET 6.0	.NET 6.0	7.045 ns	0.1645 ns	0.1285 ns	7.036 ns	0.84	0.03	0.0134	56 B	1.00
DynamicMethod	.NET 6.0	.NET 6.0	7.261 ns	0.1381 ns	0.1153 ns	7.342 ns	0.87	0.04	0.0134	56 B	1.00
CompiledExpression	.NET 6.0	.NET 6.0	6.787 ns	0.1112 ns	0.1040 ns	6.749 ns	0.81	0.05	0.0134	56 B	1.00
FastCompiledExpression	.NET 6.0	.NET 6.0	7.209 ns	0.1300 ns	0.1086 ns	7.191 ns	0.87	0.04	0.0134	56 B	1.00
SlowReflection	.NET 6.0	.NET 6.0	818.017 ns	15.6026 ns	13.0289 ns	810.704 ns	98.23	3.40	0.0458	192 B	3.43
FastReflection	.NET 6.0	.NET 6.0	579.835 ns	5.1821 ns	4.3273 ns	579.669 ns	69.64	2.73	0.0267	112 B	2.00
Baseline	.NET 7.0	.NET 7.0	8.172 ns	0.1185 ns	0.1108 ns	8.132 ns	0.98	0.05	0.0134	56 B	1.00
Delegate	.NET 7.0	.NET 7.0	9.079 ns	0.2348 ns	0.2704 ns	9.093 ns	1.10	0.07	0.0134	56 B	1.00
DynamicMethod	.NET 7.0	.NET 7.0	10.947 ns	0.2749 ns	0.6533 ns	10.813 ns	1.30	0.16	0.0134	56 B	1.00
CompiledExpression	.NET 7.0	.NET 7.0	9.509 ns	0.2473 ns	0.3702 ns	9.425 ns	1.15	0.09	0.0134	56 B	1.00
FastCompiledExpression	.NET 7.0	.NET 7.0	9.851 ns	0.2470 ns	0.3123 ns	9.823 ns	1.20	0.09	0.0134	56 B	1.00
SlowReflection	.NET 7.0	.NET 7.0	488.517 ns	7.9631 ns	7.8208 ns	486.134 ns	58.87	3.05	0.0458	192 B	3.43
FastReflection	.NET 7.0	.NET 7.0	253.621 ns	4.9668 ns	4.6460 ns	254.297 ns	30.42	1.75	0.0267	112 B	2.00
Baseline	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
Delegate	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
DynamicMethod	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
CompiledExpression	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
FastCompiledExpression	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
SlowReflection	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
FastReflection	.NET Core 3.1	.NET Core 3.1	NA	NA	NA	NA	?	?	-	-	?
Baseline	.NET Framework 4.8	.NET Framework 4.8	8.376 ns	0.3826 ns	1.0079 ns	8.059 ns	1.00	0.00	0.0134	56 B	1.00
Delegate	.NET Framework 4.8	.NET Framework 4.8	8.436 ns	0.2049 ns	0.2736 ns	8.370 ns	1.02	0.08	0.0134	56 B	1.00
DynamicMethod	.NET Framework 4.8	.NET Framework 4.8	11.009 ns	1.1505 ns	3.3922 ns	8.773 ns	1.41	0.50	0.0134	56 B	1.00
CompiledExpression	.NET Framework 4.8	.NET Framework 4.8	45.918 ns	0.9788 ns	1.9772 ns	45.435 ns	5.46	0.55	0.0134	56 B	1.00
FastCompiledExpression	.NET Framework 4.8	.NET Framework 4.8	8.655 ns	0.2408 ns	0.2958 ns	8.638 ns	1.06	0.05	0.0134	56 B	1.00
SlowReflection	.NET Framework 4.8	.NET Framework 4.8	1,929.166 ns	151.8744 ns	447.8051 ns	1,654.722 ns	235.73	52.17	0.0916	385 B	6.88
FastReflection	.NET Framework 4.8	.NET Framework 4.8	1,117.981 ns	7.4267 ns	6.2016 ns	1,116.260 ns	134.28	5.22	0.0725	305 B	5.45

```
Benchmarks with issues:
  Benchmark.Baseline: .NET 5.0(Runtime=.NET 5.0)
```

Рисунок 4.3 – Результати бенчмаркінгу маппінг механізмів (створенно самостійно)

Метод на базі рефлексії був обраний для подальшого використання в системі через його виняткову гнучкість і здатність адаптуватися до різноманітних моделей даних без необхідності змінювати код при зміні структури бази даних або об'єктних моделей. Цей підхід дозволяє автоматизувати процес мапінгу даних, що значно спрощує розробку та супровід коду, оскільки розробники можуть фокусуватися на бізнес-логіці, а не на технічних деталях обробки даних[11]. Навіть якщо рефлексія може мати певний вплив на продуктивність, цей недолік часто компенсується зменшенням часу на розробку та збільшенням гнучкості системи. Крім того, існують техніки для оптимізації рефлексивних операцій, такі як кешування метаданих, які можуть значно знизити вплив рефлексії на продуктивність. Використання рефлексії також надає можливість легко інтегрувати з іншими системами і технологіями, що використовують метадані для керування даними, як це часто робиться в сучасних розподілених та модульних архітектурах. Таким чином, незважаючи на потенційні виклики, переваги рефлексії як інструменту для динамічного мапінгу роблять його привабливим вибором для комплексних і адаптивних систем.

Розглянемо нашу реалізацію мапінгу:

```
using System.Collections.Generic;
using System.Data;
using System.Threading.Tasks;

public static class QueryMapper
{
    public static async Task<IEnumerable<T>>
QueryAsync<T>(IDbConnection connection, string sql, object parameters =
null)
    {
        using (var command = connection.CreateCommand())
        {
            command.CommandText = sql;
            // Here you would add parameters to the command if needed

            var result = new List<T>();
            using (var reader = await command.ExecuteReaderAsync())
            {
                while (await reader.ReadAsync())
                {
                    result.Add(reader.MapTo<T>());
                }
            }
```

```
                return result;
            }
        }

        public static async Task<object> QueryAsyncType(IDbConnection
connection, FormattableString sql, Type entityType, object parameters =
null)
        {
            using (var command = connection.CreateCommand())
            {
                command.CommandText = sql.ToString();
                // Here you would add parameters to the command if needed

                var result =
(IList)Activator.CreateInstance(typeof(List<>).MakeGenericType(entityType))
;
                using (var reader = await command.ExecuteReaderAsync())
                {
                    while (await reader.ReadAsync())
                    {
                        var item = Activator.CreateInstance(entityType);
                        foreach (var prop in entityType.GetProperties())
                        {
                            if (!reader.HasColumn(prop.Name) ||
reader[prop.Name] == DBNull.Value)
                                continue;

                            prop.SetValue(item, reader[prop.Name]);
                        }
                        result.Add(item);
                    }
                }
                return result;
            }
        }
    }
```

У класі QueryMapper реалізовані два методи, які відповідають за виконання SQL-запитів до бази даних та мапінг результатів на об'єкти .NET.

Перший метод, QueryAsync<T>, приймає з'єднання з базою даних, SQL-запит і параметри запиту, створює та виконує команду SQL, а потім використовує IDataReader для прочитання результатів запиту. Метод MapTo<T> з класу DataReaderExtensions використовується для мапінгу кожного рядка результату на об'єкт типу T. Результати зберігаються в списку і повертаються як асинхронне завдання.

Другий метод, QueryAsyncType, працює аналогічно, але може обробляти запити для будь-якого типу сутності, визначеного через параметр entityType. Цей

метод використовує рефлексію для створення інстансів сутностей і заповнює їх властивості значеннями з результатів запиту. Результати зберігаються в універсальному списку (IList) і повертаються як асинхронне завдання.

Обидва методи забезпечують асинхронне виконання запитів до бази даних, що дозволяє уникнути блокування основного потоку програми, підвищуючи її продуктивність і реактивність.

Наведемо нестандартні та цікаві рішення:

- методи-розширення для idatareader. Метод mapto<t> у datareaderextensions.cs дозволяє легко мапувати результати запитів на об'єкти .net. це рішення спрощує перетворення даних з бази даних у об'єкти і робить код більш читабельним і підтримуваним;
- динамічне створення сетів через рефлексію. Використання рефлексії у tinyormcontext для автоматичного створення інстансів tinyormset<t> на основі властивостей контексту. це рішення дозволяє уникнути ручного створення кожного набору сутностей і робить контекст гнучким і розширюваним;
- універсальний метод queryasynctype для мапінгу об'єктів різних типів. Метод queryasynctype у querymapper.cs дозволяє виконувати запити та маппінг для будь-якого типу сутності, використовуючи рефлексію. це забезпечує високу гнучкість у роботі з різними типами даних без необхідності написання спеціалізованих методів для кожного типу.

Expression Trees (дерева виразів) – це потужний механізм у .NET, який дозволяє програмно створювати та обробляти кодування виразів на кшталт LINQ. У нашій системі TinyORM, Expression Trees використовуються для динамічного формування запитів до бази даних на основі LINQ-виразів. Це дозволило нам писати запити в знайомому синтаксисі LINQ, які потім перетворюються на SQL-запити, що виконуються базою даних.

Клас TinyORMQueryProvider реалізує інтерфейс IQueryProvider, який визначає методи для створення і виконання запитів. Основні методи тут – CreateQuery та Execute. Код наведено нижче:

```
using System;
using System.Linq;
using System.Linq.Expressions;

public class TinyORMQueryProvider : IQueryProvider
{
    private readonly TinyORMContext _context;

    public TinyORMQueryProvider(TinyORMContext context)
    {
        _context = context;
    }

    public IQueryable CreateQuery(Expression expression)
    {
        var elementType = expression.Type.GetGenericArguments()[0];
        try
        {
            return
(IQueryable)Activator.CreateInstance(typeof(TinyORMQueryable<>).MakeGeneric
Type(elementType), new object[] { this, expression });
        }
        catch (Exception ex)
        {
            throw new InvalidOperationException("Could not create
query", ex);
        }
    }

    public IQueryable<TElement> CreateQuery<TElement>(Expression
expression)
    {
        return new TinyORMQueryable<TElement>(this, expression);
    }

    public object Execute(Expression expression)
    {
        return Execute<object>(expression);
    }

    public TResult Execute<TResult>(Expression expression)
    {
        var query = new QueryBuilder().BuildQuery(expression);
        return
_context.QueryAsync<TResult>(FormattableStringFactory.Create(query)).Result
;
    }
}
```

Продивимось методи і їх призначення в програмі:

– iqueryable createquery(expression expression). Цей метод створює запит на основі переданого дерева виразів (expression). він отримує тип елементів у запиті (elementtype), створює інстанс tinyormqueryable<> з відповідним

типом, використовуючи рефлексію, і передає йому дерево виразів. якщо виникає помилка під час створення запиту, генерується invalidoperationexception;

- метод використовує рефлексію для створення інстансу tinyormqueryable<> з типом елементів, отриманим із дерева виразів. це дозволяє створити конкретний тип запиту, який відповідає типу даних у виразі;
- iqueryable<telement> createquery<telement>(expression expression). Це узагальнений варіант методу createquery, який створює запит tinyormqueryable<telement> на основі переданого дерева виразів. метод повертає запит, який можна використовувати для подальшого виконання.
- метод приймає узагальнений тип telement, що дозволяє створювати запити для будь-якого типу сутності;
- executeobject execute(expression expression). Цей метод виконує запит, представлений деревом виразів, і повертає результат як об'єкт. він викликає узагальнений варіант методу execute<tresult>;
- метод обробляє дерево виразів і виконує запит, повертаючи результат як об'єкт;
- tresult execute<tresult>(expression expression). Узагальнений метод, що виконує запит і повертає результат типу tresult. він використовує querybuilder для побудови sql-запиту на основі переданого дерева виразів, а потім викликає метод queryasync з контексту _context для виконання запиту і отримання результату;
- метод аналізує дерево виразів і використовує querybuilder для створення sql-запиту;
- після побудови sql-запиту метод викликає асинхронний метод queryasync<tresult> у контексті бази даних для виконання запиту.
- результат запиту повертається як об'єкт типу tresult.

Використання дерев виразів в TinyORM забезпечує потужний та гнучкий механізм для створення і виконання запитів до бази даних. Це дозволяє розробникам використовувати знайомий синтаксис LINQ для динамічного

формування запитів, які потім перетворюються на SQL-запити для взаємодії з базою даних.

Узагальнений метод, що виконує запит і повертає результат типу tresult. він використовує querybuilder для побудови sql-запиту на основі переданого дерева виразів, а потім викликає метод queryasync з контексту _context для виконання запиту і отримання результату.

Загалом метод рефлексії є загальноприйнятим для реалізації маппінг механізмів у звичайних веб-застосунках та спеціалізованих, таких як наш на приклад для реалізації методів управління даними із базою даних.

Це метод використовується в багатьох подібних системах та програмних рішеннях.

Далі наведемо таблицю з описом функціоналу нашого застосунку та ролі в ORM системі яку він виконує. (див. табл. 4.1).

Таблиця 4.1 – Специфікація функціоналу (зроблена самостійно)

Фундаментальна Функція	Опис	Відповідні Класи/Методи
Управління контекстом бази даних	Центральний контекст для взаємодії з базою даних, створення наборів даних та управління з'єднаннями.	TinyORMContext - TinyORMContext(TinyORMConnection connection) - CreateSet(Type) - ResolveTableName(Type) - QueryAsync<TResult>(FormattableString)
Створення запитів	Динамічне створення запитів на основі LINQ-виразів.	TinyORMQueryProvider - CreateQuery(Expression) - CreateQuery<TElement>(Expression)

Продовження таблиці 4.1

Фундаментальна Функція	Опис	Відповідні Класи/Методи
Виконання запитів	Виконання запитів та повернення результатів.	TinyORMQueryProvider - Execute(Expression) - Execute<TResult>(Expression) TinyORMContext - QueryAsync<TResult>(FormattableString)
Побудова SQL-запитів	Перетворення дерев виразів у SQL-запити.	QueryBuilder - BuildQuery(Expression) SqlExpressionVisitor - Visit(Expression)
Маппінг результатів запитів	Маппінг рядків бази даних на об'єкти .NET.	DataReaderExtensions - MapTo<T>(IDataReader) QueryMapper - QueryAsync<T>(IDbConnection, string, object) - QueryAsyncType(IDbConnection, FormattableString, Type, object)
Підтримка LINQ	Інтеграція з LINQ для формування запитів.	TinyORMQueryable<T> - Expression - Provider TinyORMQueryProvider - CreateQuery(Expression) - CreateQuery<TElement>(Expression)
Створення наборів даних (Sets)	Створення і управління колекціями об'єктів (сетов) для операцій CRUD.	TinyORMSet<T> - Add(T) - Remove(T) - Find(object) TinyORMContext - CreateSet(Type) - CreateSetInternal<T>()

Кінець таблиці 4.1

Фундаментальна Функція	Опис	Відповідні Класи/Методи
Рефлексія для створення і ініціалізації	Використання рефлексії для створення інстансів і ініціалізації властивостей.	TinyORMContext - CreateSet(Type) - CreateSetInternal<T>()
Асинхронне виконання запитів	Асинхронне виконання запитів для підвищення продуктивності	TinyORMContext - QueryAsync<TResult>(FormattableString) QueryMapper - QueryAsync<T>(IDbConnection, string, object) - QueryAsyncType(IDbConnection, FormattableString, Type, object)

4.3 Аналіз результатів

Перевірка результатів в нашій системі здійснюється через асинхронне виконання запитів, маппінг результатів на об'єкти, використання рефлексії для динамічного створення інстансів, юніт-тести для валідації функціональності та обробку винятків для забезпечення надійності системи. Ці методи допомагають переконатися, що система працює коректно і відповідає очікуванням.

Отже після дебагінгу було отримано наступні результати(див. рис. 4.3).

На зображенні представлено результат дебагінгу, який показує масив об'єктів типу Document. Відкритий список у вікні дебагера демонструє масив results з 19 елементів, де кожен елемент є об'єктом Document.

Результат, який було отримано в дебагері, створюється за допомогою методу QueryAsync з TinyORMContext, який викликає метод QueryAsyncType з QueryMapper для виконання SQL-запиту і маппінгу результатів на об'єкти типу

Document. Ці методи забезпечують динамічне виконання запитів і маппінг результатів на відповідні об'єкти .NET.

Отже в ході дослідження архітектури мало функціональної ORM системи на платформі .NET було розглянуто та реалізовано кілька ключових аспектів, які формують фундамент для ефективної та гнучкої взаємодії з базами даних.

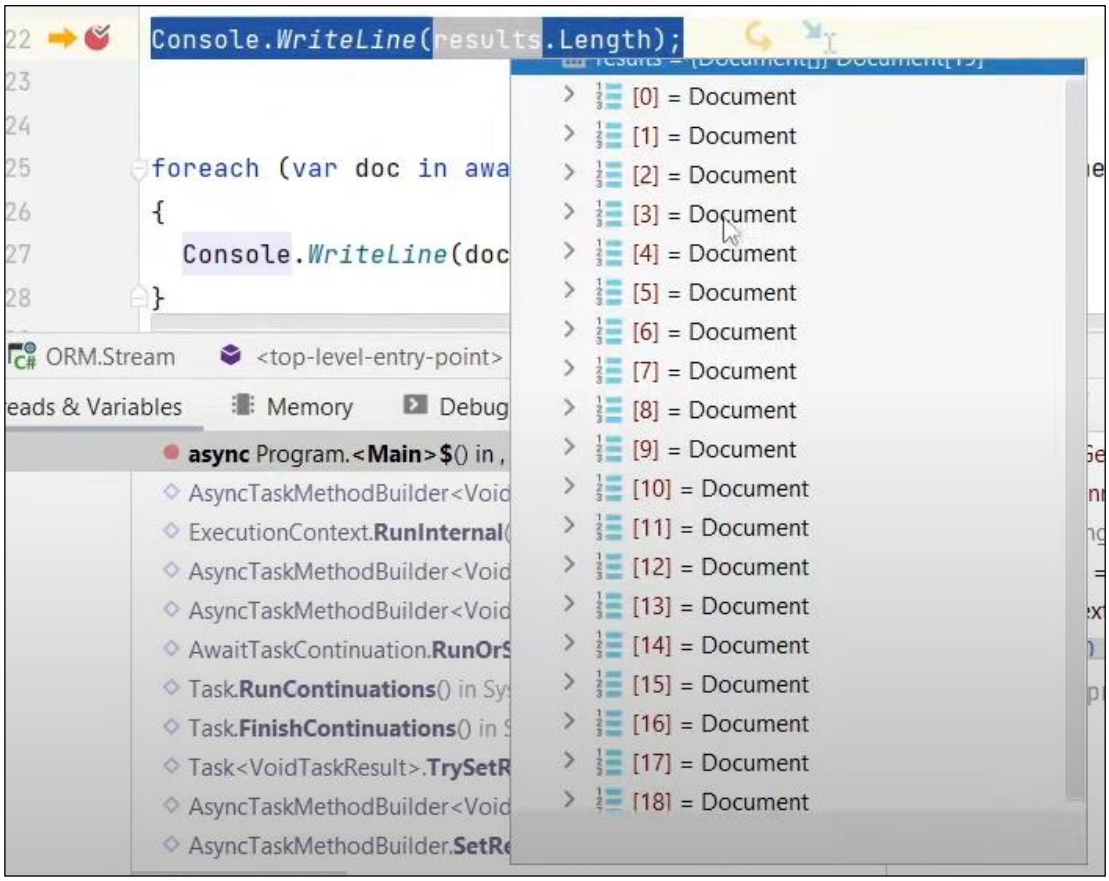

Рисунок 4.4 – Результат дебагінгу (рисунок створено самостійно)

У ході роботи визначено ключові компоненти та їхню взаємодію у рамках ORM системи:

– TinyORMContext. Керує з'єднанням з базою даних та створює набори даних;
– TinyORMQueryProvider. Використовує дерева виразів для створення та виконання запитів, побудови SQL-запитів;
– QueryMapper. Виконує SQL-запити та маппінгує результати на об'єкти;
– DataReaderExtensions. Забезпечує маппінг результатів на об'єкти через метод MapTo<T>.

Унікальні та цікаві рішення:

- динамічне створення сетів за допомогою рефлексії. Використання рефлексії для динамічного створення і ініціалізації сетів дозволяє уникнути ручного визначення кожного сету в контексті, роблячи систему більш гнучкою і легко масштабованою;
- побудова запитів за допомогою дерев виразів. Дерева виразів забезпечують динамічне формування запитів на основі linq, що дозволяє створювати запити програмно і динамічно змінювати їх під час виконання;
- асинхронне виконання запитів. Підтримка асинхронного виконання запитів підвищує продуктивність додатка, дозволяючи виконувати операції з базою даних без блокування головного потоку.

Результати нашого дослідження показують, що навіть малофункціональна ORM система може забезпечити ефективну та гнучку роботу з базами даних, використовуючи сучасні підходи, такі як дерева виразів, асинхронне програмування та динамічний маппінг. Подальші дослідження та вдосконалення можуть включати розширення функціональності системи, інтеграцію з міграційними інструментами та покращення механізмів оптимізації запитів.

ВИСНОВКИ

У кваліфікаційній роботі зосереджено увагу на аналізі області ORM (Object-Relational Mapping) з метою розробки універсального архітектурного патерну для компактних ORM систем. Основна мета дослідження полягала в ідентифікації ключових архітектурних елементів та підходів, які могли б бути застосовані при створенні ефективних, але компактних ORM інструментів, здатних задовольнити потреби сучасних розробників.

Було проаналізовано предметну область, та сферу застосування певних ORM, потім була поставлена задача та розроблено стратегію виявлення оптимальних архітектурних патернів ля компактних ORM.

Для досягнення цієї мети розглянуто та проаналізовано архітектури існуючих ORM фреймворків, які широко використовуються в Java та .NET середовищах. Дослідження охопило як великі, так і компактні ORM системи, включаючи Entity Framework Core для .NET та PetaPoco - як приклад компактного ORM. Цей аналіз дозволив нам виявити спільні та відмінні риси між обома категоріями ORM.

На основі аналізу було виявлено певні закономірності які характерні для систем не передбачаючи великих навантажень та довгого часу розробки, надалі ці напрацювання були реалізовані в бібліотеці для мапінгу яку згодом протестували та виявили абсолютну працездатність усіх розроблених елементів.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз предметної області застосування ORM, URL: https://medium.com/@masztalski/repository-architecture-with-ormlite-fcf7e08ad23e (дата звернення: 10.04.2024).

2. Шпорта А.О., Мельнікова Р.В. Дослідження архітектурних моделей та методів для побудови ORM та їх компактних варіантів на платформі .net. 28-й Міжнародний молодіжний форум «Радіоелектроніка та молодь у XXI столітті». Зб. матеріалів форуму. Т. 6., - Харків: ХНУРЕ. 2024. 529-531 с.

3. Аналіз існуючих рішень, URL: https://medium.com/@masztalski/repository-architecture-with-ormlite-fcf7e08ad23e (дата звернення: 10.04.2024).

4. Аналіз ORM на Java, URL: https://hibernate.org/orm/(дата звернення: 10.04.2024).

5. Аналіз EF Core, URL: https://learn.microsoft.com/ru-ru/ef/core/ (дата звернення: 10.04.2024).

6. Аналіз PetaPoco, URL: https://github.com/CollaboratingPlatypus/PetaPoco (дата звернення: 10.04.2024).

7. What Is ORM? A Comprehensive Guide to Object-Relational Mapping, URL: https://www.spiceworks.com/tech/data-management/articles/what-is-orm-a-comprehensive-guide-to-object-relational-mapping/ (дата звернення: 16.04.2024).

8. .NET Basics: ORM (Object Relational Mapping), URL: https://www.telerik.com/blogs/dotnet-basics-orm-object-relational-mapping (дата звернення: 16.04.2024).

9. Falatiuk H., Shirokopetleva M., Dudar Z. Investigation of Architecture and Technology Stack for e-Archive System. 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), Kyiv,Ukraine,8-11October2019.2019., URL: https://doi.org/10.1109/picst47496.2019.9061407 (дата звернення: 16.04.2024).

10. ORM Logic-Based English (OLE) and the ORM ReDesigner Tool: Fact-Based Reengineering and Migration of Relational Databases / Herman Balsters at Conference: OTM Confederated International Conferences "On the Move to Meaningful

Internet Systems" 2012.,no. 1.P. 46–74. URL: https://doi.org/10.1615/telecomradeng.v78.i10.30 (дата звернення: 16.04.2024).

11. Аналіз результатів бенчмаркіунгу URL: https://github.com/dotnet/BenchmarkDotNet (дата звернення: 16.04.2024)