Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження методів та засобів побудови архітектурних мапінг-механізмів на платформі .NET
(тема)

Виконав:
студент 2 курсу, групи ІПЗм-22-1

Шмельов О. Б.
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)

Тип програми освітньо-наукова

Керівник доц. Афанасьєва І. 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

студента *Шмельова Олега Борисовича*

(ПІБ)

1. Тема роботи «Дослідження методів та засобів побудови архітектурних мапінг-механізмів на платформі .NET».
   Затвердженя наказом по університету від 29.03.2024
2. Термін подання студентом роботи до екзаменаційної комісії 14.06.2024
3. Вхідні дані до роботи Дослідження методів та засобів побудови архітектурних мапінг-механізмів на платформі .NET.
4. Перелік питань, що потрібно опрацювати в роботі Метою роботи – є дослідження предметної області для методів та засобів побудови архітектурних мапінг-механізмів на платформі .NET, вдосконалення та детальний огляд методів та виведення патернів сценаріїв використання. В результаті буде знайдено оптимальну мапінг бібліотеку та встановлено ключові особливості підходів та методів для розробки персональних мапінг механізмів.

КАЛЕНДАРНИЙ ПЛАН

№	Назви етапів курсової роботи	Термін виконання етапів (роботи)	Примітка
1	Аналіз предметної галузі	23.01.2024 - 14.02.2024	*Виконано*
2	Розробка постановки задачі	14.02.2024 - 21.03.2024	*Виконано*
3	Розробка ПЗ	21.03.2024 - 02.04.2024	*Виконано*
4	Проведення експерименту	02.04.2024 - 23.04.2024	*Виконано*
5	Оформлення пояснювальної записки	01.05.2024 - 26.05.2024	*Виконано*
6	Підготовка презентації та доповіді	26.04.2024 - 02.05.2024	*Виконано*
7	Нормоконтроль, рецензування	11.06.2024	*Виконано*
8	Попередній захист	12.06.2024	*Виконано*
9	Здача роботи у електронний архів	13.06.2024	*Виконано*
10	Допуск до захисту у зав. кафедри	14.06.2024	*Виконано*

Дата видачі завдання 20 січня 2024р.

Студент (ка) _____ Шмельов О.Б.
(підпис)

Керівник роботи _____ доц. Афанасьєва І. В.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи магістра містить: 76 с., 16 рис., 13 джерел., 3 таблиці

АНАЛІТИКА, ПАТЕРН, ПРОГРАМНА СИСТЕМА, МАПІНГ, МАППЕР.

Об’єктом дослідження – методи та засоби побудови архітектурних мапінг-механізмів на платформі .NET

Метою роботи – є дослідження предметної області для методів та засобів побудови архітектурних мапінг-механізмів на платформі .NET, вдосконалення та детальний огляд методів та виведення патернів сценаріїв використання.

В результаті буде знайдено оптимальну мапінг бібліотеку та встановлено ключові особливості підходів та методів для розробки персональних мапінг механізмів.

ANALYTICS, PATTERN, SOFTWARE SYSTEM, MAPPING, MAPPER.

The object of the research is methods and means of building architectural mapping mechanisms on the .NET platform

The purpose of the work is to research the subject area for methods and means of building architectural mapping mechanisms on the .NET platform, improve and review the methods and derive patterns of usage scenarios.

As a result, the optimal mapping library will be found and the key features of approaches and methods for the development of personal mapping mechanisms will be established.

Я, Шмельов Олег Борисович, студент(ка) гр. ІПЗм-21-1, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Дослідження методів та засобів побудови архітектурних мапінг-механізмів 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

ПЕРЕЛІК СКОРОЧЕНЬ

ASCII – American Standard Code for Information Interchange

FIX – Financial Information Exchange

FAST – FIX Adapted for Streaming

FTP – File Transfer Protocol

SBE – Simple Binary Encoding

SOH – Simple Open Header

HTTP – Hypertext Transfer Protocol

MVD – Model-View-Delegate

SDK – Serial Development Kit

TCP – Transmission Control Protocol

WPF – Windows Presentation Foundation

XML – Extensible Markup Language

ВСТУП

У відповідності до динамічного розвитку технологій і зростаючих обсягів даних, інформаційні системи неперервно розвиваються, вводячи нові підходи та методології для ефективної роботи з даними. Особливу увагу в цьому контексті заслуговує платформа .NET, яка пропонує широкий спектр можливостей для розробки сучасного програмного забезпечення.

Маппінг даних стає важливою частиною багатьох проектів, дозволяючи ефективно інтегрувати, перетворювати та управляти інформацією між різними частинами системи. Це включає перенесення даних між різними рівнями аплікації, конвертацію між об'єктними моделями та реляційними базами даних, а також обробку даних для забезпечення відповідності бізнес-логіці.

Основна мета цього дослідження полягає у вивченні та аналізі різноманітних методів та інструментів, які можуть бути застосовані для побудови ефективних мапінг-механізмів на платформі .NET. Це включає розгляд сучасних технік об'єктно-реляційного відображення (ORM), а також інших інноваційних підходів, що можуть забезпечити гнучкість, швидкість, та надійність у процесах маппінгу.

Додатково необхідно дослідити різні версії платформи на предмет відповідності певній версії мапінг бібліотек, також важливо визначити патерни та рекомендації що до використання певної бібліотеки.

Важливо визначити принцип роботи всіх популярних методів мапінгу, особливості їх роботи та розробити бенчмарк який би допоміг розробникам з’ясувати кращий метод мапінгу відносно платформи яку вони використовують та особливості та потреби проекту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Загальна характеристика маппінгу

Історія появи маппінгу в інформаційних технологіях тісно пов'язана з розвитком комп'ютерних систем та баз даних. У початкові десятиліття комп'ютерної ери, коли основним завданням було зберігання та обробка даних, основну увагу приділяли ефективності зберігання та швидкості доступу до даних. Ранні системи баз даних були в основному ієрархічними або мережевими, і маппінг даних був справою технічної необхідності для трансформації даних з одного формату в інший.

З появою реляційних баз даних у 1970-х роках, виникла потреба у нових підходах до маппінгу. Реляційні моделі пропонували більш гнучку та універсальну систему зберігання даних, але водночас виникала потреба у нових інструментах для взаємодії між програмним забезпеченням та базами даних.

Зростання об'єктно-орієнтованого програмування в 1980-х та 1990-х роках привело до нових викликів у маппінгу. Розрив між об'єктними структурами у програмному коді та таблично-орієнтованими структурами в базах даних створив потребу в ефективних механізмах маппінгу, що могли б перекладати дані між цими двома світами. Так з'явилося об'єктно-реляційне відображення (ORM), яке дозволяло автоматизувати цей процес.

У 2000-х роках, з розвитком інтернет-технологій та веб-додатків, маппінг став ще більш важливим. Вимоги до швидкості розвитку та гнучкості додатків зросли, і тут ORM зайняло своє місце як ключовий компонент в архітектурі багатьох сучасних додатків. Системи, такі як Hibernate для Java та Entity Framework для .NET, стали стандартом у розробці програмного забезпечення.

Сьогодні, у епоху хмарних обчислень, мікросервісів та великих даних, маппінг продовжує еволюціонувати. Виникають нові виклики, пов'язані з розподіленою обробкою даних та інтеграцією різноманітних систем. Маппінг не лише перетворює дані, але й сприяє інтеграції різних джерел даних

Концепція маппінгу в контексті розробки програмного забезпечення є фундаментальною і має велике значення у створенні ефективних та гнучких систем. Маппінг, у широкому розумінні, включає процес перетворення та відображення даних з одного формату чи структури в інший. Цей процес є ключовим у багатьох аспектах програмування, від баз даних до інтерфейсів користувача.

Маппінг можна визначити як процес створення відповідностей між двома різними структурами даних. Наприклад, у контексті об'єктно-реляційного відображення (ORM) у розробці ПЗ, маппінг використовується для перетворення даних з формату, зрозумілого для реляційних баз даних, у формат, прийнятний для об'єктно-орієнтованих мов програмування, і навпаки[1].

Однією з основних проблем, з якою стикаються розробники при маппінгу, є необхідність у точному відображенні даних між системами, що мають різні моделі представлення. Це може стосуватися типів даних, відношень між об'єктами, управління ідентичністю об'єктів та виконання складних запитів до бази даних. Невірне відображення може призвести до втрати даних або до некоректної роботи програми.

Популярним рішенням є використання ORM бібліотек, таких як Entity Framework у .NET або Hibernate в Java, які надають готові засоби для маппінгу об'єктів програми на таблиці бази даних.

Інструменти автоматизованого маппінгу, такі як AutoMapper у .NET, дозволяють визначати маппінги за допомогою конфігураційних правил, мінімізуючи потребу у вручну написаному коді[2].

Деякі системи дозволяють автоматично відображати об'єкти на бази даних, використовуючи конвенції щодо іменування і структури.

Маппінг даних продовжує еволюціонувати з розвитком технологій. Наприклад, з появою хмарних технологій і мікросервісів, з'являються нові виклики та можливості для оптимізації маппінгу. Також, зростання інтересу до штучного інтелекту та машинного навчання створює потребу у більш розширених та адаптивних підходах до маппінгу даних.

Завдяки цьому, маппінг перетворюється з простої технічної необхідності в стратегічний інструмент, який може допомогти організаціям більш ефективно управляти своїми даними та використовувати їх для отримання цінних бізнес-інсайтів.

1.1.1 Існуючі дослідження

Дослідження – Object-Relational Mapping vs Entity Framework, зосереджено на порівняльному аналізі продуктивності запитів між двома популярними ORM (Object-Relational Mapping) інструментами для .NET: Entity Framework (EF) Core та Dapper. Сенс цього дослідження полягає у визначенні, який з цих інструментів краще підходить для різних сценаріїв використання на основі їх продуктивності при обробці різних типів баз даних.

Дослідження було реалізовано за допомогою .NET Web API, який використовує EF Core та Dapper для виконання операцій CRUD (створення, читання, оновлення, видалення) над базою даних PostgreSQL. Основними критеріями оцінки були час відповіді на запити та кількість оброблених запитів за визначений часовий проміжок. Для тестування продуктивності використовувався інструмент k6, що дозволяє проводити навантажувальне тестування веб-додатків.

Метою було не тільки порівняти продуктивність EF Core і Dapper, але й дати розробникам інформацію, яка допоможе їм вибрати найбільш підходящий інструмент для своїх проектів залежно від конкретних вимог до продуктивності та складності розробки.

Результати показали, що Dapper має кращу продуктивність порівняно з EF Core в тестах на створення та читання даних, в основному через його легковагову архітектуру та пряме використання SQL-запитів, що мінімізує накладні витрати на обробку даних. Однак, у тесті на читання даних за ідентифікатором EF Core показав кращі результати, що свідчить про те, що в певних сценаріях використання він може бути більш ефективним.

Загальний висновок дослідження полягає у тому, що вибір між Dapper та EF Core залежить від конкретних потреб проекту, вимог до продуктивності, та

переваг у розробці. Dapper рекомендується для проектів, де важлива висока продуктивність і гнучкість у використанні SQL-запитів, тоді як EF Core може бути кращим вибором для проектів, що вимагають швидкої розробки за рахунок більш високої абстракції. Це порівняння має на меті визначити, який інструмент краще підходить для різних задач розробки залежно від їх продуктивності при виконанні операцій з базами даних.

Головна мета полягає у визначенні, який із цих двох інструментів є більш ефективним з точки зору швидкості виконання запитів до бази даних і загальної продуктивності при роботі з даними. Це особливо важливо для розробників, які прагнуть оптимізувати продуктивність своїх додатків, забезпечуючи при цьому ефективне управління даними.

У дослідженні використовується .NET Web API, що дозволяє порівнювати продуктивність EF Core і Dapper через серію тестів продуктивності, зокрема, вставки (CREATE) і вибірки (READ) даних. Для тестування продуктивності використовується інструмент k6, який моделює навантаження на додаток і вимірює реакцію системи на це навантаження. В результаті дослідження зібрано дані про середній час обробки запитів, медіану, а також процентильні значення, що дозволяє отримати уявлення про розподіл часу відповіді системи на запити[4].

Цей аналіз допомагає розробникам ухвалювати обґрунтовані рішення при виборі технологій для взаємодії з базами даних в своїх проектах, засновуючись на конкретних потребах продуктивності та специфіці розробки.

Створення (CREATE): Dapper перевершив EF Core, вставивши більше рядків за той же часовий проміжок і показавши кращу середню швидкість обробки запитів. Це підтверджує, що Dapper може бути більш ефективним для операцій на великі обсяги даних, коли потрібна максимальна швидкість вставки[5].

Читання (READ): У тестах на читання Dapper також показав кращі результати, обробивши більше запитів за той самий проміжок часу порівняно з EF Core. Однак, у деяких сценаріях, наприклад, при вибірці даних за

ідентифікатором, EF Core може виявитися кращим варіантом, забезпечуючи швидший доступ до даних.

Загалом, дослідження показало, що вибір між Dapper і EF Core залежить від конкретних потреб проекту. Dapper виявився кращим варіантом для сценаріїв, що вимагають високої продуктивності та ефективності при роботі з великими обсягами даних, завдяки своїй легкій архітектурі та прямому використанню SQL-запитів. Це дозволяє зменшити обчислювальні витрати та покращити час відгуку додатка.

Однак, використання Dapper може збільшити складність розробки, оскільки вимагає від розробників глибшого розуміння SQL та ручного керування взаємодією з базою даних. Таким чином, хоча Dapper надає кращу продуктивність, він також вимагає більшого часу та зусиль на розробку.

EF Core, з іншого боку, пропонує більш високий рівень абстракції та автоматизації у взаємодії з базою даних, що може спростити розробку за рахунок зниження продуктивності. EF Core краще підходить для проектів, де важливі швидкість розробки та зручність управління даними, ніж крайня оптимізація продуктивності.

Вибір між Dapper і EF Core повинен базуватися на вагомому аналізі потреб проекту, рівні вмінь розробників та вимогах до продуктивності. Це дослідження надає цінні вказівки, що допоможуть у цьому виборі, висвітлюючи переваги та недоліки кожного інструменту в різних умовах використання.

1.2 Актуальність проблеми

Актуальність вибору маппінг бібліотеки та методу для створення особистого мапера на платформі .NET обумовлена кількома ключовими факторами, які впливають на ефективність, швидкість та якість розробки програмного забезпечення.

Обробка даних є фундаментальною частиною більшості сучасних програм, і вибір ефективного механізму маппінгу безпосередньо впливає на продуктивність

програми. Неправильний вибір може призвести до зайвого навантаження на систему та зниження продуктивності.

У великих та складних системах необхідна висока гнучкість маппінгу, щоб адаптуватися до змін у бізнес-логіці та структурі даних. Вибір методу маппінгу та бібліотеки повинен бути здатний задовольнити потреби масштабування та розвитку проекту.

Вибір між використанням готових бібліотек, таких як AutoMapper або Mapster, і створенням власного мапера за допомогою методів, як Source Generators або Reflection Emit, також впливає на складність коду, швидкість розробки та легкість підтримки програми.

Цей метод забезпечує високу продуктивність, оскільки код генерується на етапі компіляції, знижуючи навантаження в рантаймі. Він також підходить для сценаріїв, де потрібно чітке визначення типів та сильна типізація.

Хоча цей метод може бути більш гнучким, оскільки дозволяє динамічно створювати код в рантаймі, він може бути менш продуктивним у порівнянні з Source Generators. Reflection Emit вимагає глибшого розуміння внутрішньої роботи .NET та може бути складнішим у підтримці.

У виборі підходу до маппінгу також важливо враховувати сумісність з іншими технологіями та фреймворками, використовуваними в проекті, такими як ASP.NET, Entity Framework, або мікросервісна архітектура.

Враховуючи ці фактори, вибір між використанням готових маппінг бібліотек маппінгу або створенням власного мапера за допомогою Source Generators чи Reflection Emit стає особливо актуальним. Цей вибір має стратегічне значення, адже від нього залежить здатність проекту ефективно адаптуватися до змін, швидкість розробки, а також загальна продуктивність та підтримуваність системи.

Враховуючи ці вимоги, розробники стикаються з важливим вибором: використовувати готові маппінг бібліотеки, які пропонують швидке впровадження та легкість у використанні, але можуть бути обмежені у термінах налаштувань, або ж розробляти власні маппінг механізми, що дозволяють більшу гнучкість і оптимізацію, але вимагають більших зусиль у розробці та підтримці.

Таким чином, вибір між готовими бібліотеками та власними рішеннями є критичним моментом. Це рішення має базуватися на глибокому розумінні потреб проекту, а також на знаннях про сильні та слабкі сторони кожного з доступних підходів до маппінгу.

1.4 Постановка задачі

Уході даної роботи нами буде досліджено предметну область дослідження маппінг механізмів та маппінг бібліотек, ми розберемо які є бібліотеки, далі розберемо як вони працюють та їх особливості, розділимо до архітектурних патернів усі архітектурні та складові елементи бібліотек. Ми проведемо порівняльний аналіз методів та фундаментальних підходів для створення особистих маппінг механізмів. Ця робота спрямована на виявлення критеріїв для вибору способу маппінгу об’єктів на певному проекті.

Реалізація наукового дослідження складається з наступних етапів:

- аналіз предметної області;
- аналіз архітектурних особливостей сучасних маппінг бібліотек;
- знаходження фундаментальних закономірностей у мапінг механізмах;
- порівняльний аналіз та модифікація існуючих рішень.

2 АНАЛІЗ ІСНУЮЧИХ МЕТОДІВ

2.1 Аналіз маппінг бібліотек

З початку розберемо декілька най-популярніших бібліотек.

Mapster – це бібліотека для маппінгу даних в .NET, яка дозволяє легко та ефективно виконувати операції маппінгу між об'єктами даних, такими як класи POCO (Plain Old CLR Objects), DTO (Data Transfer Objects), та іншими. Mapster був створений для спрощення рутинних завдань маппінгу та пропонує кілька ключових особливостей і переваг.

Mapster спрощує створення маппінгу за допомогою інтуїтивного синтаксису. Вам не потрібно писати багато коду для визначення маппінгу між двома класами. Зазвичай вам просто потрібно викликати метод Adapt, і Mapster відповідатиме за решту.

Mapster підтримує різні сценарії маппінгу, що робить його вельми гнучким:

- один до одного (One-to-One): Можна визначити маппінг між об'єктами одного типу на об'єкти іншого типу;
- багато до одного (Many-to-One): Можна визначити маппінг списків об'єктів одного типу на об'єкти іншого типу;
- один до багатьох (One-to-Many): Можна визначити маппінг об'єкта одного типу на список об'єктів іншого типу;
- багато до багатьох (Many-to-Many): Можна визначити маппінг списків об'єктів одного типу на списки об'єктів іншого типу.

Це дозволяє легко вирішувати різноманітні завдання маппінгу, навіть якщо вони вимагають обробки великої кількості даних.

Mapster також має інші функціональні особливості, такі як вбудована підтримка проекцій, можливість налаштування глибинного копіювання об'єктів, можливість ігнорувати властивості та підтримка атрибутів для більшого контролю над маппінгом. Бібліотека також відома своєю високою продуктивністю завдяки генеруванню оптимізованого коду.

Що стосується плюсів і мінусів Mapster, то вони більш загальні і можуть залежати від конкретних потреб проекту та особливостей бібліотеки в порівнянні з іншими аналогічними інструментами, такими як AutoMapper. Важливо пам'ятати, що вибір бібліотеки для маппінгу даних повинен враховувати конкретні потреби, якість та продуктивність коду, а також зручність в роботі з нею.

Mapster також дозволяє легко вказати властивості, які повинні бути проігноровані під час маппінгу. Це корисно, коли ми маєте деякі властивості, які не потрібно мапити через певні причини.

Загалом, ці особливості Mapster роблять його дуже гнучким і зручним для роботи з маппінгом даних, а також дозволяють знижувати навантаження на базу даних та робити код більш зрозумілим і придатним до використання.

Далі на рисунку 2.1 наведено архітектурну структуру Mapster.

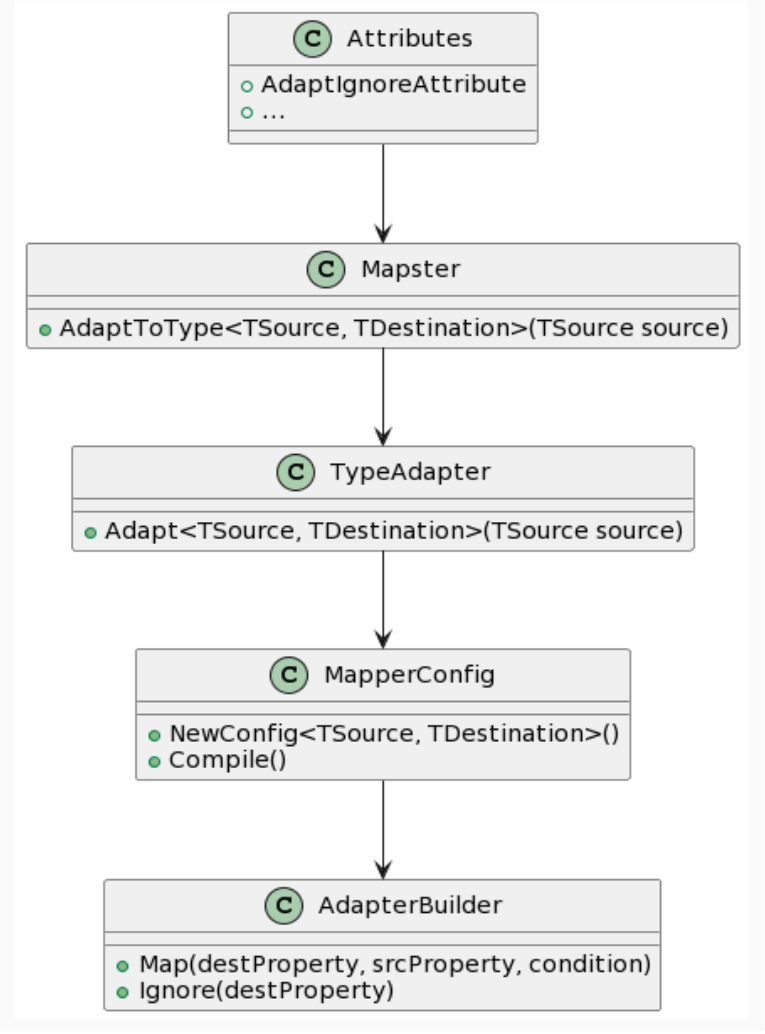


Рисунок 2.1 Архітектурна структура Mapster

Розберемо елементи більш детально:

- MapperConfig – це головний клас для конфігурації Mapster. Він використовується для налаштування загальних правил маппінгу та визначення специфічних маппінгів для пар типів. Основні методи включають NewConfig, який визначає правила маппінгу для конкретних типів, і Compile, який компілює конфігурацію та дозволяє виконувати маппінг;
- TypeAdapter – це клас, який виконує фактичний маппінг об'єктів між типами. Він використовується для створення нового об'єкта з використанням правил маппінгу, визначених у MapperConfig;
- AdapterBuilder – це клас дозволяє налаштовувати і керувати маппінгом між конкретними типами. Ми можете використовувати його, щоб визначити, які властивості мають бути маплені і як це слід робити;
- AdaptToType – це метод дозволяє створити новий об'єкт певного типу на основі існуючого об'єкта з використанням налаштованих правил маппінгу;
- Attributes – Mapster підтримує використання атрибутів, які можуть бути застосовані до властивостей класів, щоб вказати додаткові налаштування маппінгу. Наприклад, AdaptIgnoreAttribute використовується для ігнорування певних властивостей під час маппінгу.

Далі розберемо другого популярного інструменту маппінгу – Automaper.

AutoMapper – це популярна бібліотека для маппінгу даних в .NET, яка надає зручний спосіб визначити та виконати правила маппінгу між об'єктами різних типів. Вона дозволяє зменшити дублювання коду та спростити операції маппінгу, що полегшує розробку програмного забезпечення та забезпечує більшу читабельність коду.

Система дозволяє легко виконувати завдання, такі як зведення об'єктів до об'єктів (object-to-object mapping), проекції даних, обчислення обчислювальних властивостей та ігнорування певних властивостей під час маппінгу.

Вона широко використовується в програмуванні на платформі .NET для полегшення операцій обробки та трансформації даних в додатках.

Зручний інтерфейс конфігурації надає простий та декларативний інтерфейс для визначення правил маппінгу між об'єктами різних типів. Це дозволяє розробникам легко визначати, які властивості мають бути маплені.

Автоматичний маппінг може автоматично виконувати маппінг властивостей між об'єктами на основі їхніх назв та типів. Це полегшує рутинні операції маппінгу та зменшує необхідність вручну визначати правила маппінгу.

Незважаючи на автоматичний маппінг, AutoMapper надає можливість ручного налаштування маппінгу для складних сценаріїв. Ми можемо налаштовувати властивості окремо або визначати власні конвертери для більшої гнучкості.

AutoMapper дозволяє створювати проекції даних, вибираючи лише необхідні властивості для виходу. Це допомагає оптимізувати запити до бази даних та зменшити обсяг пересиланої інформації в додатках.

AutoMapper використовує комбінацію Reflection та Expressions для маппінгу об'єктів. Він починає з Reflection для виявлення властивостей, які потрібно відображати, а потім використовує скомпільовані Expressions для створення швидкого та ефективного коду маппінгу. Це дозволяє AutoMapper бути гнучким у визначенні відповідностей між типами, а також забезпечувати високу продуктивність маппінгу за рахунок уникнення використання Reflection в часі виконання.

Наведемо приклад підходу Expression Trees:

```
var propertyInfo = typeof(MyClass).GetProperty("MyProperty");
var value = propertyInfo.GetValue(myObject, null);
```

Цей код використовує Reflection для отримання значення властивості MyProperty з об'єкта myObject.

```
var parameter = Expression.Parameter(typeof(MyClass));
var property = Expression.Property(parameter, "MyProperty");
```

```
var lambda = Expression.Lambda<Func<MyClass, PropertyType>>(property, parameter).Compile();
var value = lambda(myObject);
```

Цей код використовує Expression Trees для створення динамічної функції, яка отримує значення властивості MyProperty[6].

Важливо відзначити, що AutoMapper оптимізує ці процеси, використовуючи кешування та інші техніки для підвищення продуктивності. Ці приклади є спрощеними версіями того, що відбувається в бібліотеці.

Загалом головна перевага AutoMapper полягає у його здатності до автоматичного маппінгу з мінімальною необхідністю конфігурації. Це дозволяє розробникам швидко налаштувати маппінг між об'єктними моделями, значно зменшуючи кількість ручного коду, який потрібно написати. AutoMapper також надає гнучкість для складних маппінгів завдяки своїм налаштуванням і розширенням.

2.2 Аналіз архітектурних підходів до створення механізмів маппінгу

Далі нам потрібно розібрати ключові, фундаментальні підходи які використовуючи свої специфічні методи – реалізують мапінг різноманітних бібліотек, ці концепції та підходи надалі будуть теж порівняні та виведені їх переваги та недоліки, також необхідно буде порівняти деякі методи цих підходів.

Source Generators – це одна з нововведень, яка була введена в C# 9.0, і ця технологія може бути використана в контексті маперів, включаючи AutoMapper, для автоматизації генерації коду маппінгу між об'єктами. Source Generators дозволяють генерувати додатковий код під час компіляції на основі анотованого вихідного коду, що може спростити рутинні завдання, такі як маппінг об'єктів.

Підхід має наступні архітектурні складові:

- атрибути та анотації: ми використовуємо атрибути та анотації в нашому коді, щоб вказати source generator, які класи або методи потребують генерації додаткового коду для маппінгу. наприклад, ми можемо створити

спеціальний атрибут automap, який вказує, які класи мають бути маплені, і які методи мають бути викликані для генерації маппінгу;

- source generator: ми створюємо source generator, який аналізує наш вихідний код та використовує інформацію з атрибутів і анотацій для створення додаткового коду маппінгу. source generator може генерувати методи маппінгу, які додаються до нашого додаткового коду;
- компіляція: коли ми компілюєте свій проект, source generator викликається під час компіляції. він аналізує наш код і генерує код маппінгу на основі атрибутів та анотацій, після чого цей згенерований код включається до нашого додаткового коду;
- готовий код маппінгу: згенерований код маппінгу стає частиною проекту. тепер ми можемо використовувати цей готовий код для виконання маппінгу між об'єктами;
- source generators дозволяють автоматизувати процес генерації коду маппінгу, що робить роботу з маперами більш ефективною та менше помилковою. вони дозволяють створювати більш чистий та підтримуваний код, особливо для великих проектів, де маппінг може стати складним та монотонним завданням.

Далі розберемо послідовність виконання цього підходу(див.рис.2.2).

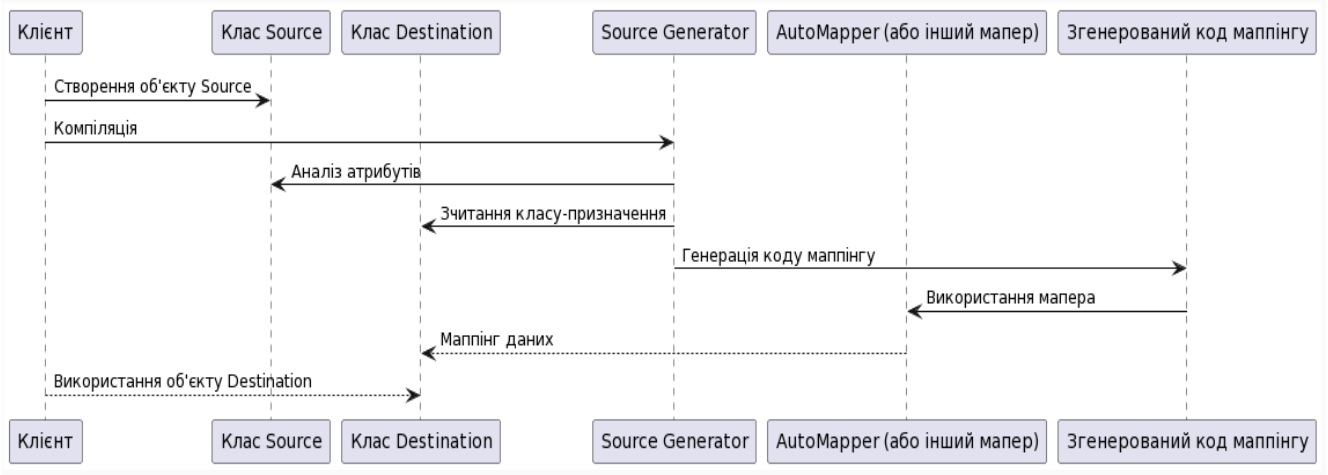


Рисунок 2.2 – Послідовність роботи Source Generators

Ця діаграма послідовності показує взаємодію між клієнтом, класами джерела (SourceClass) та призначення (DestinationClass), Source Generator,

AutoMapper та згенерованим кодом маппінгу. Клієнт створює об'єкт класу Source, а після компіляції Source Generator аналізує атрибути та генерує код маппінгу, який використовується AutoMapper для мапінгу даних на об'єкт Destination.

Основна особливість Source Generators полягає в тому, що вони дозволяють автоматично генерувати код на основі вихідного коду під час компіляції, на відміну від інших підходів, де код маппінгу пишеться вручну або генерується під час виконання програми. Source Generators дозволяють забезпечити автоматизований та ефективний процес генерації коду маппінгу, зменшуючи необхідність в рутинній роботі розробника.

IQueryable – це інтерфейс в .NET Framework, який представляє запит до джерела даних, який може бути виконаний для отримання результатів запиту. Цей інтерфейс дозволяє побудувати запит до джерела даних (наприклад, бази даних) на мові C# без його виконання, а потім виконати цей запит, коли результати будуть потрібні. IQueryable зазвичай використовується для роботи з базами даних і дозволяє оптимізувати та вибирати тільки потрібні дані під час виконання запиту.

IQueryable базується на технології LINQ (Language Integrated Query) і дозволяє використовувати мовні конструкції C# для побудови запитів до бази даних, що називається "LINQ to SQL".

Однією з ключових особливостей IQueryable є те, що він дозволяє відкладати виконання запиту до джерела даних до того моменту, коли результати фактично потрібні. Це дозволяє оптимізувати запити і вибирати лише необхідні дані.

При роботі з IQueryable, запити зазвичай компілюються в запити до бази даних. Це означає, що вони можуть бути оптимізовані та виконані на стороні бази даних, що призводить до покращення продуктивності та ефективності запитів.

IQueryable може використовуватися для роботи з різними джерелами даних, такими як реляційні бази даних, колекції об'єктів, XML-документи і інші.

Важливо розуміти, що IQueryable не виконує запит до бази даних без явного виклику методів, таких як ToList(), FirstOrDefault(), Count() і інші, які вимагають

отримання результатів запиту. Це робить IQueryable потужним інструментом для оптимізації запитів і вибору тільки необхідних даних для обробки.

Зазвичай IQueryable використовується в додатках, де потрібна робота з джерелами даних, такими як бази даних, і коли важливо оптимізувати та контролювати запити до цих джерел.

Він дозволяє створювати потужні і оптимізовані запити, зменшуючи навантаження на базу даних та забезпечуючи ефективну роботу з даними.

Підхід побудови маперів на expressions - це техніка, яка використовує вирази (expressions) в C# для автоматичної генерації коду маппінгу між об'єктами різних типів. Цей підхід дозволяє створювати маппери, які можуть динамічно аналізувати та мапити дані без явного написання коду маппінгу для кожного поля або властивості. Ось докладніше про цей підхід:

Вирази в C# дозволяють представляти код на рівні мови програмування. Підхід побудови маперів на expressions використовує expressions для створення динамічних функцій, які можуть визначати, як дані між об'єктами будуть мапитися.

Маппери на expressions створюють лямбда-вирази, які мають підпис функції для маппінгу властивостей одного об'єкта на інший. Ці лямбда-вирази виконуються під час виконання програми і виконують мапінг даних.

Під час створення маппера на expressions, код аналізує структуру типів об'єктів і властивостей, які потрібно змапити. Це може виконуватися використовуючи відображення (reflection) або аналіз коду.

Зберігання та кешування: Створені лямбда-вирази можуть бути збережені та кешовані для подальшого використання, що дозволяє уникнути зайвого аналізу структури об'єктів при кожному мапінгу.

Підхід побудови маперів на expressions є потужним інструментом для автоматичного мапінгу даних між об'єктами. Він дозволяє створювати динамічні мапери з гнучкими правилами маппінгу і зменшує зусилля, необхідні для розробки та підтримки коду маппінгу.

Далі йде найбільший та мабуть найважливіший підхід у створення маппінг механізмів, а саме – Expression Trees.

Expression Tree підхід у контексті маперів у .NET використовується для створення більш ефективних і гнучких механізмів мапінгу об'єктів.

Створення Виразів для Мапінгу: Використовуючи Expression Trees, можна динамічно створювати вирази, які визначають, як мапити дані з одного об'єкта в інший. Це може включати копіювання властивостей, перетворення типів даних або більш складні операції.

Динамічність та Гнучкість: Expression Trees дозволяють маперам бути гнучкими і адаптованими до різних сценаріїв. Мапінги можуть бути налаштовані в рантаймі без необхідності зміни вихідного коду.

Оптимізація Продуктивності: Порівняно з традиційними методами мапінгу, які використовують рефлексію, мапінг на основі Expression Trees може пропонувати кращу продуктивність, оскільки вирази можуть бути компільовані та оптимізовані.

Інтеграція з ORM Фреймворками: Цей підхід особливо корисний при роботі з ORM фреймворками, такими як Entity Framework. Expression Trees можуть бути перетворені в оптимізовані SQL-запити, що зменшує навантаження на базу даних.

Компіляція та Виконання: Expression Trees можуть бути скомпільовані та виконані, що дозволяє використовувати динамічно створені мапінги в реальному часі.

Підтримка Комплексних Сценаріїв: Цей підхід дозволяє реалізувати складні сценарії мапінгу, які можуть включати умовні логіку, цикли та інші програмні конструкції, недоступні в простих маперах.

Узагальнюючи, Expression Tree підхід у маперах є потужним інструментом для реалізації динамічних, ефективних і гнучких рішень мапінгу в програмах на .NET. Далі буде розглянуто сценарій використання цього підходу(див. рис. 2.3).

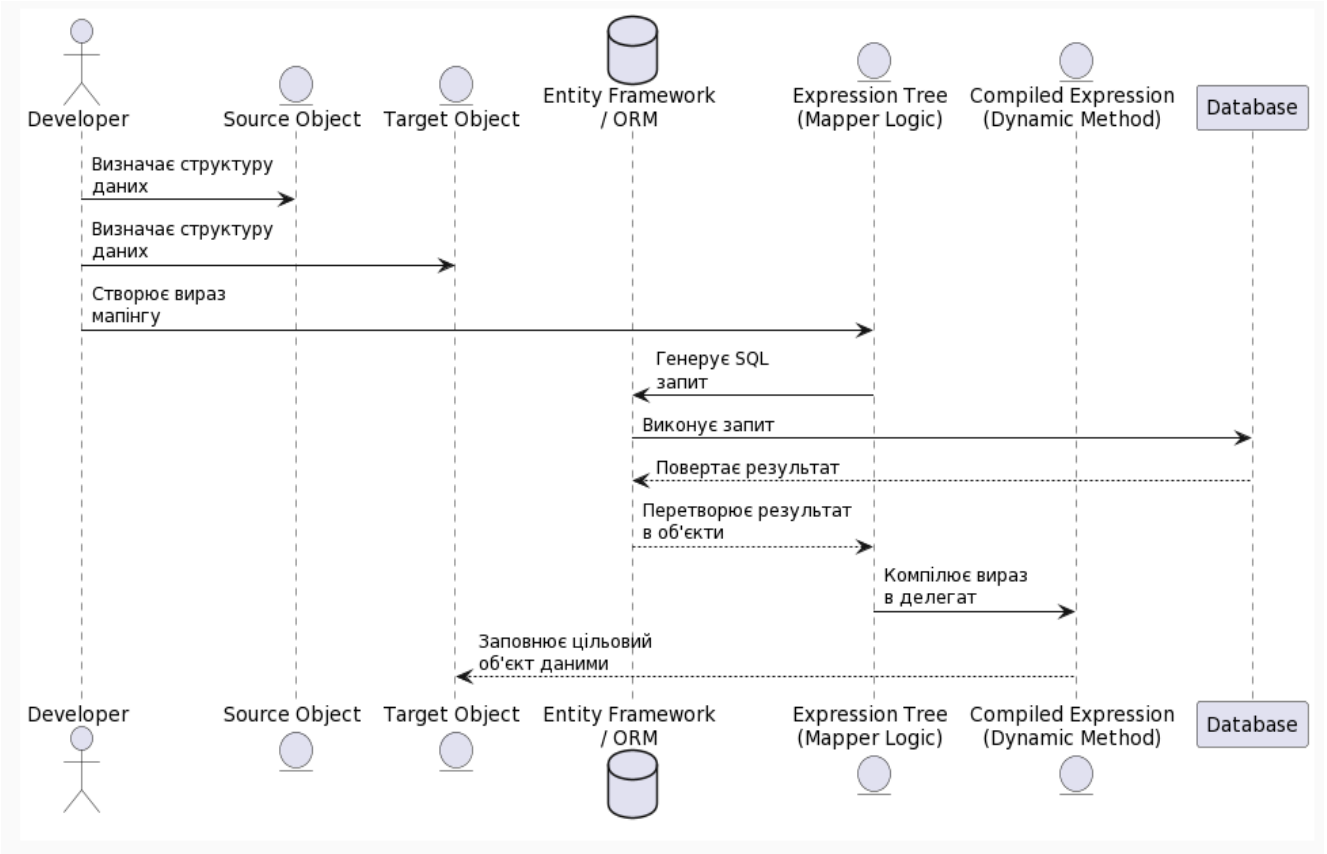


Рисунок 2.3 – Послідовність роботи Expression Trees

На малюнку ми бачимо наступні єтапи імплементації:

- розробник визначає структуру даних для вихідного та цільового об'єктів;
- створюється вираз мапінгу за допомогою Expression Trees;
- вираз мапінгу використовується для генерації SQL запиту у ORM (наприклад, Entity Framework);
- ORM виконує SQL запит до бази даних і отримує результати;
- результати перетворюються назад у об'єкти за допомогою ORM та передаються в Expression Tree;
- Expression Tree компілює вираз у делегат;
- цей делегат використовується для заповнення цільового об'єкта даними.

Expression Trees у .NET є потужним механізмом для представлення та маніпулювання кодом як структурованими даними в рантаймі.

Вони дозволяють динамічно створювати та компілювати запити, що можуть бути використані для мапінгу об'єктів, оптимізації запитів до баз даних та інших складних операцій обробки даних. Завдяки можливості компіляції в SQL та інтеграції з ORM-інструментами, Expression Trees забезпечують ефективність і гнучкість у розробці програмного забезпечення.

Рефлексія в .NET — це потужний механізм, який дозволяє програмам інспектувати і модифікувати самі себе в рантаймі. Ця можливість надає розробникам унікальні інструменти для динамічної взаємодії з кодом, який може бути невідомим на момент компіляції. Рефлексія широко використовується для таких завдань, як серіалізація об'єктів, клонування, інвокація методів, а також для розробки багатьох інструментів і фреймворків, що використовуються в сучасному програмуванні.

Одна з ключових особливостей рефлексії — можливість інспектувати типи. В .NET це реалізується за допомогою класу Type, який містить інформацію про класи, інтерфейси, перечислення та інші компоненти програми. Ми можемо дізнатися про методи, поля, властивості, делегати, які є у класі, використовуючи методи GetMethod, GetField, GetProperty, і так далі. Крім того, рефлексія дозволяє аналізувати атрибути, які застосовані до різних членів.

Рефлексія дозволяє не тільки переглядати атрибути і структуру класів, але й викликати методи та змінювати значення полів і властивостей в рантаймі. Це робить можливим динамічну модифікацію поведінки об'єктів. Наприклад, ми можемо динамічно викликати метод, ім'я якого отримуєте в якості вхідного параметра, або змінити значення приватного поля, що може бути корисним у складних сценаріях інтеграції або при написанні бібліотек, що взаємодіють з не дуже добре відомими або сторонніми типами даних.

Рефлексія дозволяє створювати екземпляри класів використовуючи Activator.CreateInstance або конструктори з ConstructorInfo. Це особливо корисно в системах, де типи даних можуть змінюватися або не бути відомими на етапі компіляції. Це також є основою для реалізації різноманітних фабрик об'єктів, IoC

контейнерів, які використовуються для управління залежностями в сучасних застосунках.

Незважаючи на потужні можливості, рефлексія приходить з ціною у вигляді накладних витрат на продуктивність.

Виклики через рефлексію відносно повільніші за прямі виклики методів, особливо коли вони виконуються велику кількість разів. Тому важливо використовувати рефлексію обачно і оптимізувати доступ до часто використовуваних членів через кешування або інші методи зменшення впливу на продуктивність.

Рефлексія може становити потенційні ризики безпеки, оскільки дозволяє доступ до приватних членів класів та методів. Це може призвести до непередбачуваних змін в стані об'єктів, якщо не контролювати доступ до рефлексивних операцій відповідними засобами безпеки.

Рефлексія залишається важливим інструментом в арсеналі розробника .NET, надаючи гнучкість та можливості, які важко досягнути іншими засобами. Втім, її використання вимагає розуміння і обережності, особливо у контексті продуктивності і безпеки.

Також важливо зазначити, що рефлексія дозволяє створювати більш гнучкі та розширювані системи, зокрема, при розробці плагінів або модульних додатків. Це забезпечує можливість динамічного завантаження та виконання компонентів, що підвищує адаптивність і масштабованість програмного забезпечення.

Однак, незважаючи на всі переваги рефлексії, її варто використовувати лише тоді, коли немає інших, більш ефективних шляхів досягнення необхідних результатів. Надмірне використання рефлексії може ускладнити відлагодження та супровід коду, а також призвести до погіршення продуктивності, тому баланс між гнучкістю і ефективністю завжди має бути під контролем розробника.

Хоча вона має недоліки, але згідно документації – розробники платформи невпинно її допрацьовують щоб вона нарощувала продуктивність відносно інших методів мапінгу(див. рис. 2.4).

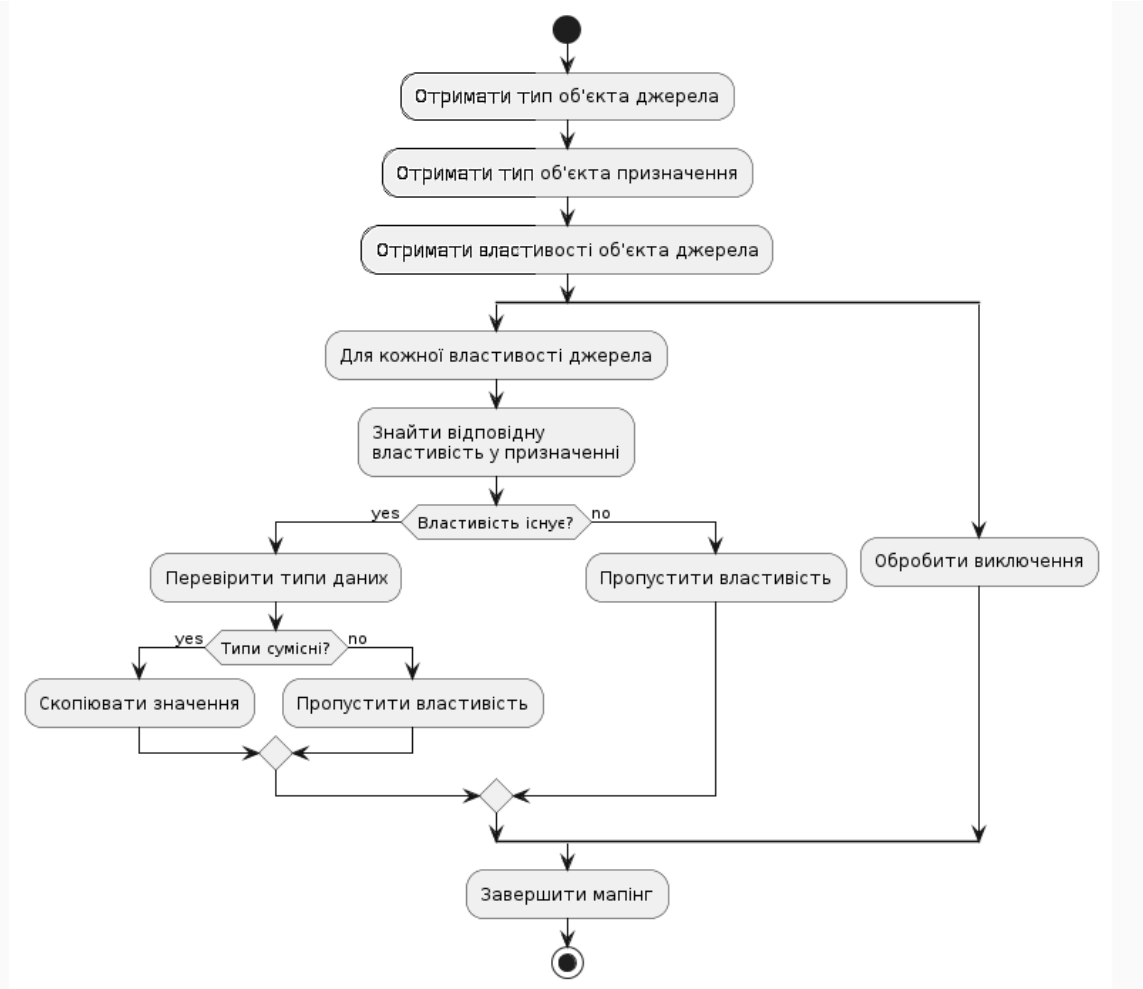


Рисунок 2.4 – принцип роботи методу через рефлексію

Ця модель ілюструє, як мапер на базі рефлексії може динамічно копіювати дані між об'єктами, що може бути особливо корисним у ситуаціях, де структури даних не повністю відомі на етапі компіляції.

Надалі необхідно буде розглянути способи модифікації підходів до побудови мапінг механізмів та порівняти розглянуті засоби для встановлення сценаріїв використання та рекомендацій по створенню мапінг механізмів з нуля.

3 ЕКСПЕРЕМЕНТАЛЬНЕ ПОРІВНЯННЯ БІБЛІОТЕК

3.1 Виведення оптимального методу та підходу до маппінгу

Після детального дослідження бібліотек та підходів, стало зрозуміло що загалом є два основних підходи до проектування мапперу – через source generators та через reflection ammit та expressions які використовуются у mapster та automaper[7].

Через Source Generators – цей підхід включає генерацію коду під час компіляції. Він ефективний для створення маперів, оскільки знижує навантаження у рантаймі та дозволяє створювати високопродуктивний код[8].

Через Reflection Emit та Expressions – цей підхід, що використовується у Mapster та AutoMapper, базується на рефлексії та Expression Trees. Він дозволяє динамічно створювати та компілювати запити, що робить його гнучким і потужним інструментом для мапінгу об'єктів, особливо у складних сценаріях.

Використання Source Generators у .NET рекомендується у таких випадках:

- коли потрібна висока продуктивність – Source Generators зменшують навантаження у рантаймі, оскільки генерують код під час компіляції;
- у великих та складних проектах – для проектів з великою кількістю мапінгу, де генерація коду відіграє ключову роль;
- для створення статично типізованого коду – це забезпечує меншу ймовірність помилок у рантаймі та кращу інтеграцію з IDE;
- коли необхідна сумісність з AoT (Ahead-of-Time) компіляцією – наприклад, у випадках розробки для платформ, які вимагають AoT;
- коли важлива читабельність та підтримуваність коду – Source Generators дозволяють легше розуміти та підтримувати код, оскільки він генерується автоматично, але є читабельним і відповідає стандартам проекту.

Також треба розібрати Fast Expression Compiler, він використовується для оптимізації продуктивності компіляції Expression Trees. Це забезпечує швидше створення делегатів, які використовуються для мапінгу, порівняно зі стандартним Expression.Compile()[9].

Source Generators, з іншого боку, забезпечують генерацію коду на етапі компіляції, що покращує загальну продуктивність, знижуючи навантаження у рантаймі. Обидва підходи використовуються для підвищення ефективності маперів, але діють на різних рівнях і можуть бути комбіновані для досягнення оптимальної продуктивності.

У цьому конексті необхідно буде проаналізувати наступні методи: BaseLine, DynamicMethod, Delegate, CompiledExpression, FustCompiledExpression, SlowReflection, FastReflection.

- BaseLine: Стандартний, базовий підхід або точка відліку для порівняння інших методів;
- DynamicMethod: Використовується для створення методів динамічно в рантаймі;
- Delegate: Стандартний делегат у .NET, що використовується для капсулювання методів;
- CompiledExpression: Використання Expression Trees, що компілюються в делегати;
- FastCompiledExpression: Оптимізована версія CompiledExpression для швидшої компіляції;
- SlowReflection: Мапінг за допомогою рефлексії, який може бути відносно повільним;
- FastReflection: Оптимізований підхід до рефлексії для покращення продуктивності.

3.2 Оптимальна бібліотека для маппінгу

Проаналізувавши дві бібліотеки для мапінгу mapster vs automaper, можна зробити загальний висновок що mapster – це повноцінна бібліотека, яка має функціональність, що можна порівняти з AutoMapper. У багатьох сценаріях Mapster простіше в налаштуванні через те, що немає необхідності явно задавати вихідний та цільовий типи для моделей доти, доки не потрібні додаткові налаштування мапінгу властивостей.

Але для детального виміру продуктивності – необхідно розробити невелику бенчмарк утиліту яка буде збирати статистику мапінгу із багатьох бібліотек та проектів з відкритим вихідним кодом[10].

3.2.1 Огляд файлової структури

Файлова структура бенчмарк проекту розподілена за відповідними директоріями. Також наявні тестові дані. На рисунку 3.1 наведено файлову структуру нашого бенчмарк OMBenchmark.

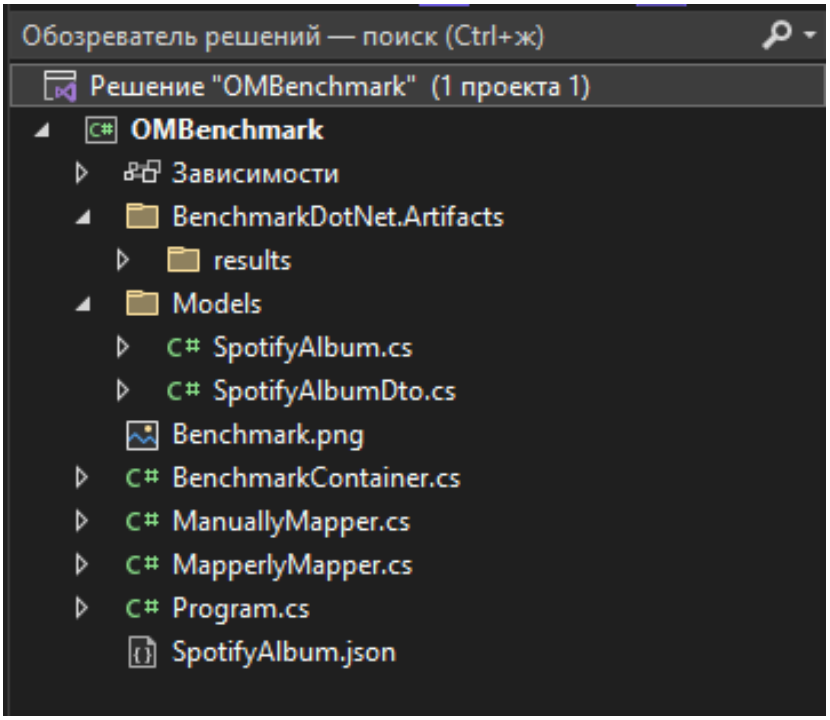


Рисунок 3.1 – Рузультати проведення бенчмарку для mapster vs. Automaper

Проект включає моделі даних для альбому Spotify, такі як SpotifyAlbum, SpotifyAlbumDto та відповідні дочірні класи (Artist, Image, Tracks тощо). Ці моделі використовуються для серіалізації/десеріалізації JSON даних, які представляють альбоми Spotify.

Наш проект містить кілька класів маперів:

- manuallymapper: клас для ручного мапінгу об'єктів з spotifyalbumdto в spotifyalbum;

- mapperlymapper: визначається як частковий клас, який, ймовірно, використовує деяку форму генерації коду для мапінгу.

Конфігурація бенчмарку:

- клас benchmarkcontainer включає анотації та методи для виконання бенчмарку різних маперів, включаючи automapper, tinymapper, expressmapper, mapster та інші;
- в globalsetup методі здійснюється ініціалізація та налаштування всіх маперів, які будуть тестуватися.

Використовуючи бібліотеку BenchmarkDotNet, проект вимірює продуктивність кожного мапера. Виходи цього бенчмарку можуть включати час виконання, використання пам'яті та інші метрики продуктивності.

Також варто відмітити що Program.cs виконує бенчмарк, візуалізує результати і зберігає їх у формі зображення.

Проект використовує приклад JSON даних, що представляють альбом Spotify (SpotifyAlbum.json), для тестування маперів.

3.2.2 Стандартизація даних

Для того щоб оцінити коректним чином мапінг на усіх бібліотеках – важливо забезпечити однакові умови проведення експерименту.

Стандартизація підходу до тестування в системі забезпечує, що всі мапери працюють у рівних і контрольованих умовах, дозволяючи об'єктивно оцінювати їхню продуктивність. Далі розглянемо основні кроки, які ми використовуємо у нашому проекті для досягнення стандартизації:

Єдиний джерело даних: усі мапери використовують однаковий вхідний набір даних. Це досягається за допомогою десеріалізації одного і того ж JSON файлу (SpotifyAlbum.json) до DTO (SpotifyAlbumDto), який потім використовується як вхід для кожного мапера. Це забезпечує, що всі мапери працюють з однаковою інформацією.

Конфігурація маперів: перед початком тестів кожен мапер налаштовується в методі GlobalSetup. Це включає ініціалізацію та налаштування маперів з усіма

потрібними відображеннями та конфігураціями, щоб забезпечити, що кожен мапер виконує мапінг згідно з однаковими правилами. Нижче наведемо код із системи:

```
    public void Setup()
    {
        var json = File.ReadAllText("spotifyAlbum.json");
        _spotifyAlbumDto = SpotifyAlbumDto.FromJson(json);

        // AutoMapper Configuration
        var mapperConfig = new MapperConfiguration(cfg =>
        {
            cfg.CreateMap<SpotifyAlbumDto, SpotifyAlbum>();
            // Configuration for other mappings...
        });
        _autoMapper = mapperConfig.CreateMapper();

        // TinyMapper Configuration
        Nelibur.ObjectMapper.TinyMapper.Bind<SpotifyAlbumDto,
SpotifyAlbum>();
        // Configuration for other mappings...

        // ExpressMapper Configuration
        global::ExpressMapper.Mapper.Register<SpotifyAlbumDto,
SpotifyAlbum>();
        // Configuration for other mappings...

        // Mapperly Configuration (if exists in your settings)
        _mapperlyMapper = new MapperlyMapper();


    }
```

Однакові умови виконання забезпечуються бібліотекою BenchmarkDotNet яка дозволяє контролювати умови виконання бенчмарку, включаючи керування процесорним часом та іншими системними ресурсами. Це допомагає зменшити зовнішні впливи на продуктивність тестів.

Перевірка налаштувань середовища: наш код перевіряє, чи запущено тести в режимі DEBUG або з прикріпленим дебагером, що може вплинути на результати продуктивності. У разі використання режиму DEBUG користувачеві буде вказано на це, та запропоновано змінити режим задля проведення експерименту відповідно до умов, які б задовольняли необхідні. Ці перевірки забезпечують, що

бенчмарки запускаються в релізному режимі для найточніших вимірювань. Нижче наведемо код із системи:

```
      static void ValidateRuntime()
      {
          if (BenchmarkAutoRunner.IsRunningInDebugMode())
          {
              Console.ForegroundColor = ConsoleColor.Red;
              Console.WriteLine("You are in DEBUG mode. To achieve accurate
results, set project configuration to RELEASE mode.");

              Console.ForegroundColor = ConsoleColor.Yellow;
              Console.WriteLine("Please wait 3 seconds to enter DEBUG
mode");

              Thread.Sleep(3000);
          }
          else if (Debugger.IsAttached)
          {
              Console.ForegroundColor = ConsoleColor.Red;
              Console.WriteLine("Debugger is Attached. To achieve accurate
results, run project without Debugger.");

              Console.ForegroundColor = ConsoleColor.Yellow;
              Console.WriteLine("Or if you want to Debug project, set
project configuration to DEBUG mode.");

              Console.ForegroundColor = ConsoleColor.White;
              Environment.Exit(0);
          }
      }
```

Таким чином гарантується проведення бенчмаркінгу у ідентичних умовах для найточніших вимірювань.

3.4 Співставлення маперів та методів мапінгу

Для того, щоб достеменно розібратись у продуктивності маперів та методах мапінгу – нам необхідно порівняти продуктивність маперів та відповідно їх методами, на яких вони побудовані[11].

Відповідно, для більш детального розуміння принципів роботи кожної бібліотеки мапінгу для початку треба співставити методи мапінгу до бібліотеки мапінгу. Порівняння бібліотек, із зазначенням використовуваного методу мапінгу та з коротким описом принципів роботи відповідних методів мапінгу наведено у

таблиці 3.4. Це порівняння дозволяє краще розуміти відмінності між бібліотеками, що покращить якість розуміння експерименту та його результатів.

Таблиця 3.4 Співставлення методів до маперів(таблиця виконана самостійно)

Бібліотека мапінгу	Метод мапінгу	Опис методу
AutoMapper	Рефлексія, скомпільовані вирази	Використовує рефлексію для налаштування мапінгу на етапі запуску і кешує скомпільовані вирази для викликів в рантаймі.
Mapster	Скомпільовані вирази, Source Generators	Генерує код в рантаймі або використовує Source Generators для створення коду на етапі компіляції.
TinyMapper	Рефлексія, скомпільовані вирази	Використовує рефлексію для ініціалізації мапінгу і кешує скомпільовані вирази для швидшого виконання.
ExpressMapper	Рефлексія	Залежить від рефлексії для динамічного мапінгу об'єктів.
ValueInjecter	Рефлексія	Використовує рефлексію для мапінгу значень між об'єктами, дозволяючи дуже налаштовувані мапінги.
AgileMapper	Рефлексія, скомпільовані вирази	Створює складні мапінги і використовує кешування для оптимізації виконання.

Зміни в методах мапінгу, таких як рефлексія, скомпільовані вирази (Expressions), швидкі вирази (Fast Expression Compiler), дерева виразів (Expression Trees), і Source Generators, часто відбуваються з оновленнями платформи .NET

через зусилля Microsoft щодо підвищення продуктивності, безпеки і гнучкості розробки.

Тож для більш коректно статистики – наведемо співставлення версій бібліотек до версій платформи у таблиці 3.5.

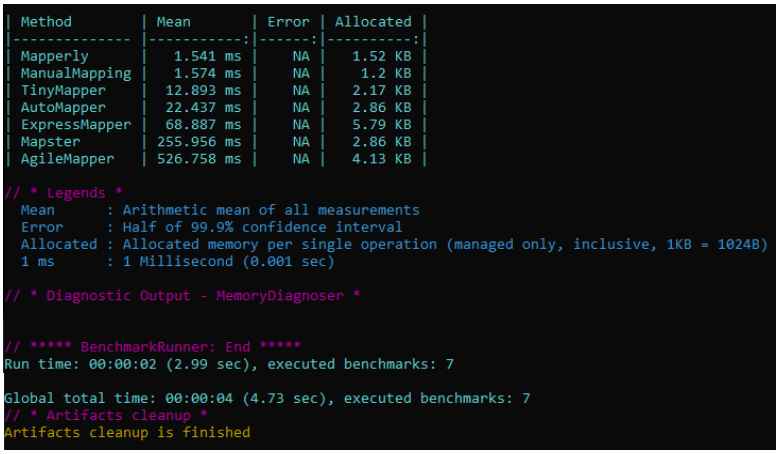
Таблиця 3.5 Співставлення методів до версій платформи(таблиця виконана самостійно)

Бібліотека мапінгу	.Net Framework	.Net Core	.Net 4.8	.Net 5	.Net 6	.Net 7	.Net 8
AutoMapper	4.6.1+	2.0+	2.0	4.6.1+	4.6.1+	4.6.1+	4.6.1+
Mapster	4.5+	2.0+	2.0-4.5	4.5+	4.5+	4.5+	4.5+
TinyMapper	4.0+	Not support	Not support	Not support	Not support	Not support	Not support
AgileMapper	Not specified	2.0+	2.0+	2.0+	2.0+	2.0+	2.0+

Із цієї таблиці – ми робимо важливий та очевидний висновок – мапінг механізми а із ними і бібліотеки змінювались протягом певних версій системи, далі важливо з’ясувати чи змінювалась продуктивність.

3.5 Проведення тестування та аналіз результатів

Процес тестування буде включати аналіз ефективності кожної бібліотеки відносно її версії, нами буде протестовано бібліотеки версії яких відрізняються функціональними можливостями версії платформі .Net для якої вона назначалась, таким чином окрім продуктивності нами буде протестовано продуктивність механізмів мапінгу для кожної версії платформи. Було прийнято рішення запустити програму на різних версіях .Net, а саме на 4.8(див.рис.3.2)



```
| Method        | Mean       | Error | Allocated |
|-------------- |-----------:|------:|----------:|
| Mapperly      |   1.541 ms |    NA |   1.52 KB |
| ManualMapping |   1.574 ms |    NA |    1.2 KB |
| TinyMapper    |  12.893 ms |    NA |   2.17 KB |
| AutoMapper    |  22.437 ms |    NA |   2.86 KB |
| ExpressMapper |  68.887 ms |    NA |   5.79 KB |
| Mapster       | 255.956 ms |    NA |   2.86 KB |
| AgileMapper   | 526.758 ms |    NA |   4.13 KB |

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ms      : 1 Millisecond (0.001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
Run time: 00:00:02 (2.99 sec), executed benchmarks: 7

Global total time: 00:00:04 (4.73 sec), executed benchmarks: 7
// * Artifacts cleanup *
Artifacts cleanup is finished
```

Рисунок 3.2 – Результат тесту на 4.8

А також на .NET 6 версії(див.рис.3.3).

```
Job=Dry  Toolchain=InProcessEmitToolchain  IterationCount=1
LaunchCount=1  RunStrategy=ColdStart  UnrollFactor=1
WarmupCount=1

| Method        | Mean       | Error | Allocated |
|-------------- |-----------:|------:|----------:|
| ManualMapping |   1.557 ms |    NA |   2.13 KB |
| Mapperly      |   1.581 ms |    NA |    1.9 KB |
| TinyMapper    |  10.659 ms |    NA |   2.17 KB |
| AutoMapper    |  22.344 ms |    NA |   2.86 KB |
| ExpressMapper |  51.880 ms |    NA |   5.79 KB |
| Mapster       |  88.496 ms |    NA |   1.92 KB |
| AgileMapper   | 216.943 ms |    NA |   4.13 KB |

// * Legends *
  Mean      : Arithmetic mean of all measurements
  Error     : Half of 99.9% confidence interval
  Allocated : Allocated memory per single operation (managed only, inclusive, 1KB = 1024B)
  1 ms      : 1 Millisecond (0.001 sec)

// * Diagnostic Output - MemoryDiagnoser *


// ***** BenchmarkRunner: End *****
Run time: 00:00:01 (1.67 sec), executed benchmarks: 7

Global total time: 00:00:02 (2.32 sec), executed benchmarks: 7
// * Artifacts cleanup *
Artifacts cleanup is finished
```

Рисунок 3.3 – Результат тесту на .NET 6

Аналіз результатів:

- manualmapping та mapperly показують найкращі часові показники, приблизно 1.55 мс, що свідчить про їх високу ефективність і мінімальні накладні витрати;
- tinymapper та automapper відносно швидкі, але значно повільніші за найшвидші методи;

- expressmapper та mapster показують середні результати, що можуть бути пов'язані з додатковою логікою обробки в цих бібліотеках;
- agilemapper має найгірший результат, що може вказувати на складність операцій, які він виконує при мапінгу;
- manualmapping знову є одним із найшвидших, але злегка повільніший порівняно з .net 6;
- tinymapper, automapper, та expressmapper показують гірші часові показники, ніж на .net 6, що може бути пов'язано з менш ефективним управлінням пам'яттю або іншими оптимізаціями в новіших версіях .net;
- mapster і agilemapper також мають гірші показники, особливо agilemapper, що вказує на високі витрати на виконання мапінгу.

Основні висновки:

- загалом, продуктивність маперів здається кращою на .net 6 порівняно з .net 4.8, що може свідчити про загальні покращення в рантаймі та jit компіляторі в новіших версіях .net;
- мапери, які використовують простіші або більш прямі методи мапінгу (наприклад, ручне мапінг або використання легковажних бібліотек;
- використання складніших бібліотек, таких як automapper та agilemapper, може призводити до більш високих затримок, на старіших .net.

Цей аналіз підкреслює важливість вибору відповідної бібліотеки мапінгу з огляду на конкретні потреби застосунку та середовище виконання, що впливає на загальну продуктивність та ефективність застосунку.

4 ЕКСПЕРЕМЕНТАЛЬНЕ ПОРІВНЯННЯ ПІДХОДІВ

Отже на даний момент ми проаналізували швидкість роботи наявних бібліотек та з’ясували певні патерни, ми з’ясували що продуктивність деяких бібліотек змінювалась відносно версій платформи, отже у роботі самої платформи – змінювалась робота із певним мапінг механізмом.

У зв’язку з цим – варто провести повторне дослідження але вже більш детально дослідити підходи до мапінгу, а саме дослідити швидкість роботи кожного методу мапінгу відносно наявних версій платформ, для цього нами було реалізовано програмну систему бенчмаркінгу яка збере необхідну для аналізу та висновків статистику.

4.1 Вхідні дані

Тестування різних механізмів мапінгу для порівняння їх ефективності вимагає стандартизованого підходу, де кожен механізм має бути протестований у схожих умовах. Використання одних і тих же даних для всіх мапінг механізмів є критично важливим з наступних причин:

Використання однакових даних гарантує, що всі мапінг механізми мають однакову "вихідну точку". Це дозволяє точно оцінювати виключно ефективність механізмів мапінгу, мінімізуючи вплив зовнішніх або несподіваних факторів, таких як різниця в об'ємах даних чи їх структурі.

Якщо використовувати різні набори даних для кожного механізму, можливе виникнення зміщення результатів. Деякі мапери можуть краще справлятися з певними типами даних або структурами. Стандартизація даних допомагає забезпечити, що всі механізми мапінгу тестуються рівномірно.

На рисунку 4.1 зображено приклад двох тестових моделей. Моделі мають подібну структуру та типи даних кожного з властивостей тестових класів, що необхідно для проведення експеременту приведення між двома типами. Використано такі типи даних як int, decimal та string.

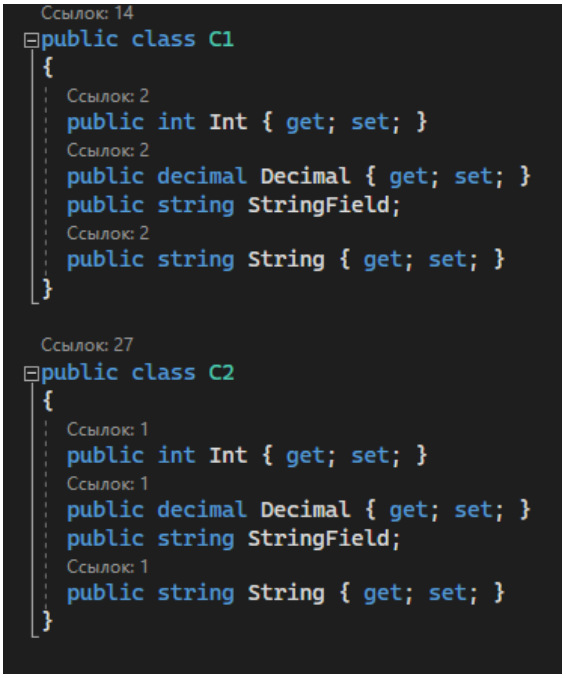


Рисунок 4.1 – Тестові моделі

Далі також зазначимо участь BenchmarkDotNet у класі нашого бенчмарку.

Наш проект використовує BenchmarkDotNet для тестування та порівняння різних методів мапінгу, включаючи вручну імплементацію, динамічні методи, компільовані та швидко компільовані вирази, а також повільні та швидкі методи рефлексії. Це дозволяє оцінити продуктивність кожного методу мапінгу в різних версіях .NET.

Розглянемо як було використано BenchmarkDotNet в нашому проекті:

- методи мапінгу: різні методи мапінгу анотовані як [benchmark], що дозволяє benchmarkdotnet автоматично тестувати і зібрати продуктивність кожного методу;

- глобальна підготовка: використання методу, анотованого як [globalsetup], для ініціалізації об'єктів та налаштування, необхідних для мапінгу перед запуском тестів. це забезпечує, що всі мапінги виконуються з тими ж умовами.

- конфігурація benchmarkdotnet: визначені різні рантайми для тестування, включаючи .net framework 4.8, .net core 3.1, .net 5.0, .net 6.0, і .net 7.0.

це робиться за допомогою атрибутів, таких як [simplejob], що дозволяє вказати, які рантайми використовувати для кожного тесту(див.рис.4.2);

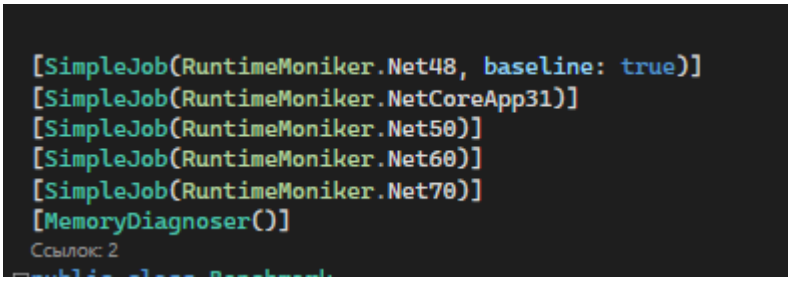


Рисунок 4.2 – Використання атрибутів BenchmarkDotNet

Наша імплементація в Program.cs використовує BenchmarkDotNet для забезпечення об'єктивних, повторюваних, і надійних результатів порівняння продуктивності різних методів мапінгу, що дозволяє глибше зрозуміти як оптимально використовувати ці методи в різних умовах.

Далі наведемо метрики які він зібрав, методи метрик включають:

- ручний мапінг (mapmethod);
- динамічні методи (dynamicmethod);
- компільовані вирази (compiledexpression);
- швидко компільовані вирази (fastcompiledexpression);
- методи рефлексії (slowreflection та fastreflection).

Використання атрибуту [MemoryDiagnoser] для збору даних про використання пам'яті кожним методом мапінгу. Це забезпечує інформацію не тільки про час виконання, але і про ресурси пам'яті, що використовуються.

Наступним важливим кроком було – забезпечити роботу нашого бенчмарку одразу із декількаами версіями платформи, для отримання результатів на однаковій продуктивності системи та обладнання, але звісно нас сам перед для порівняння продуктивності підходів відносно декількох версій.

Нижче наведено код використані атрибути для збору статистики та тестування на різних версіях платформи:

```
[SimpleJob(RuntimeMoniker.Net48, baseline: true)]
[SimpleJob(RuntimeMoniker.NetCoreApp31)]
```

```
[SimpleJob(RuntimeMoniker.Net50)]
[SimpleJob(RuntimeMoniker.Net60)]
[SimpleJob(RuntimeMoniker.Net70)]
[MemoryDiagnoser()]
```

Кожен атрибут використовується для конфігурації специфічних аспектів бенчмаркінгу. Далі наведемо детальний опис кожного з них:

Атрибут SimpleJob використовується для запуску тестів продуктивності на певних версіях .NET. Кожен із наших SimpleJob атрибутів націлений на різну версію .NET:

- RuntimeMoniker.Net48: Вказує, що бенчмарк має виконуватися на .NET Framework 4.8;
- RuntimeMoniker.NetCoreApp31: Вказує на .NET Core 3.1;
- RuntimeMoniker.Net50: Вказує на .NET 5;
- RuntimeMoniker.Net60: Вказує на .NET 6;
- RuntimeMoniker.Net70: Вказує на .NET 7.

Опція baseline: true для .NET 4.8 вказує, що результати для цієї платформи будуть використані як базові (baseline) для порівняння з результатами інших платформ. Це дуже корисно для візуалізації різниці в продуктивності між різними версіями .NET.

Атрибут MemoryDiagnoser включає збір даних про використання пам'яті тестованими методами. Це дозволяє отримати інформацію не тільки про час виконання тестів, але й про кількість алокованої пам'яті під час виконання.

Це особливо важливо, коли ми хочемо зрозуміти вплив нашого коду на ресурси системи і можливі проблеми зі збором сміття.

4.1.1 Додання методу Delegate та Baseline

Метод, позначений як Baseline, використовується для створення базової лінії у порівняльному аналізі продуктивності різних методів мапінгу. У коді, метод Baseline() викликає MapMethod(C1 c), який виконує мапінг об'єктів між класами C1 і C2, використовуючи пряме присвоєння значень полів і властивостей. Цей

метод служить як стандарт, до якого будуть порівнюватися всі інші методи мапінгу, оцінювані у бенчмарках. Цей метод діє напряму отже є найшвидшим.

Було вирішено додатково для порівняння обрати метод делегатів для порівняння різних підходів мапінгу, оскільки він є одним із найпростіших і найбільш ефективних способів виконання цієї операції в .NET. Делегати дозволяють капсулювати методи в змінні, що можна викликати динамічно, що робить їх ідеальними для реалізації гнучких рішень мапінгу, які можуть легко адаптуватися до змін у бізнес-логіці або в структурі даних без необхідності змінювати код.

Делегати в .NET – це типи, які безпосередньо представляють сигнатури методів і можуть використовуватися для зберігання посилань на методи. В контексті мапінгу, делегати можуть бути використані для створення загальних маперів, які виконують специфічні операції копіювання або трансформації даних між об'єктами різних типів.

Особливості мапінгу за допомогою делегатів:

- висока продуктивність: мапінг через делегати часто є швидшим, ніж мапінг через рефлексію або динамічне викликання, оскільки делегати компілюються і можуть бути оптимізовані clr (common language runtime). вони мінімізують накладні витрати, що зазвичай асоціюються з рефлексією;
- гнучкість: мапери, реалізовані через делегати, можуть легко адаптуватися під зміни у вхідних даних або вимогах до обробки даних, оскільки змінити логіку мапінгу можна, просто змінивши метод, на який вказує делегат;
- зниження залежності від рефлексії: хоча рефлексія дозволяє дуже гнучко керувати типами та членами класу в рантаймі, вона може призводити до зниження продуктивності.

Мапінг через делегати дозволяє уникнути цих витрат, оскільки делегати налаштовуються і використовуються без викликів рефлексії після їх компіляції.

У нашому бенчмарку відповідно реалізовано такий метод(див. рис. 4.3).

```
[MethodImpl(MethodImplOptions.NoInlining)]
Ссылок: 2
private C2 MapMethod(C1 c)
{
  C2 result = new C2();
  result.Int = c.Int;
  result.Decimal = c.Decimal;
  result.String = c.String;
  result.StringField = c.StringField;

  return result;
}
```

Рисунок 4.3 – Приклад реалізації методу делегату

Вважається що подібний метод мапінгу є найшвидшим. В MapMethod, де прямо копіюються значення властивостей з одного об'єкта до іншого, вважається одним із найшвидших через кілька ключових причин.

Метод мапінгу використовує прямий доступ до полів та властивостей, що мінімізує витрати на виклик методів і обходить потребу у рефлексії або більш складних методах отримання та встановлення значень. Цей тип доступу до даних є найшвидшим можливим у .NET, оскільки він відбувається без додаткових обчислень або перевірок.

Такий метод не вимагає використання виразів або інших форм компіляції в рантаймі, що може бути часо затратним. Відсутність необхідності у компіляції коду перед виконанням також сприяє його швидкодії.

Загальний код бенчмарку наведено у додатку Б.

4.1.2 Інші методи

Метод CompiledExpression використовує LINQ Expression Trees для побудови і компіляції лямбда-виразів у делегати, які потім можуть виконувати мапінг. Expression Trees дозволяють побудувати складні вирази, які включають умовні оператори, цикли та багато іншого, а потім компілюють ці вирази в код, який можна виконувати в рантаймі. Цей метод дуже гнучкий і потужний, але часто має деякі накладні витрати, пов'язані з процесом компіляції.

FastCompiledExpression також використовує Expression Trees, але з оптимізованим процесом компіляції. Це досягається за допомогою бібліотеки, такої як FastExpressionCompiler, яка є значно швидшою за стандартний метод компіляції, використовуваний у .NET. Цей метод спрямований на зменшення витрат часу та ресурсів, пов'язаних із компіляцією виразів, надаючи більш ефективну продуктивність у сценаріях, де необхідно часто використовувати динамічно згенеровані функції.

DynamicMethod – це доволі новий та нестандартний метод мапінгу, він використовує більш низькорівневий підхід до динамічного виконання коду. Він дозволяє програмістам створювати методи в рантаймі за допомогою IL (Intermediate Language), який є асемблером для віртуальної машини .NET. Це надзвичайно потужний інструмент, який дозволяє точно контролювати поведінку виконання коду, максимізувати продуктивність і мінімізувати виклики до високорівневих API. Однак, робота з IL може бути складнішою і вимагає глибшого розуміння внутрішньої структури .NET.

Але окрім складності написання подібного методу – варто пам’ятати що оскільки DynamicMethod генерує код в рантаймі, компілятор не може перевірити типи або інші аспекти безпеки до моменту виконання. Це може призвести до помилок типів, які будуть виявлені тільки під час виконання.

Кожен із цих викликів потребує уважного розгляду при виборі DynamicMethod для реалізації мапінгу в проектах на .NET. Важливо зважити потенційні переваги в продуктивності проти можливих ризиків і складностей.

У нашій системі використання DynamicMethod було реалізовано через наступні кроки:

- створення dynamicmethod: ми створили динамічні методи для мапінгу об'єктів, використовуючи il код, щоб точно визначити, як дані мають копіюватися між об'єктами. це включало визначення методів для читання значень властивостей з одного об'єкта і їх запису в інший;

- оптимізація через sigil: для полегшення роботи з il і забезпечення кращої продуктивності ми використали бібліотеку sigil, яка дозволяє більш зручно генерувати il код.

Код методу наведено нижче:

```
private Func<C1, C2> CreateMethod()
  {
    var emiter = Emit<Func<C1, C2>>.NewDynamicMethod("MyMethod");
    using (var local = emiter.DeclareLocal<C2>())
    {
      emiter.NewObject<C2>();
      emiter.StoreLocal(local);

      emiter.LoadLocal(local);
      emiter.LoadArgument(0);
      emiter.CallVirtual(typeof(C1).GetProperty("Int")?.GetMethod);
      emiter.CallVirtual(typeof(C2).GetProperty("Int")?.SetMethod);

      emiter.LoadLocal(local);
      emiter.LoadArgument(0);

emiter.CallVirtual(typeof(C1).GetProperty("Decimal")?.GetMethod);

emiter.CallVirtual(typeof(C2).GetProperty("Decimal")?.SetMethod);

      emiter.LoadLocal(local);
      emiter.LoadArgument(0);

emiter.CallVirtual(typeof(C1).GetProperty("String")?.GetMethod);

emiter.CallVirtual(typeof(C2).GetProperty("String")?.SetMethod);

      emiter.LoadLocal(local);
      emiter.LoadArgument(0);
      emiter.LoadField(typeof(C1).GetField("StringField"));
      emiter.StoreField(typeof(C2).GetField("StringField"));

      emiter.LoadLocal(local);
      emiter.Return();
    }
    return emiter.CreateDelegate();
  }
```

У нашому методі CreateMethod, Emit<Func<C1, C2>> використовується для створення динамічного методу, який трансформує об'єкт типу C1 в об'єкт типу C2. Це здійснюється шляхом прямого використання IL (Intermediate Language), який є машинним кодом для .NET CLR (Common Language Runtime).

Використання IL дозволяє оптимізувати продуктивність шляхом уникнення додаткових витрат на виклик методів через рефлексію. IL код виконується безпосередньо CLR, що забезпечує набагато швидше виконання порівняно з високорівневими операціями, які вимагають інтерпретації чи додаткових перевірок.

Цей підхід зокрема корисний у сценаріях, де необхідно виконувати велику кількість мапінгів, або коли мапінг має бути виконаний якомога швидше, як наприклад у реальному часі або в системах з високим навантаженнямх[12].

Також дуже важливо відзначити що цей метод повинен бути ідентичним до нашого базового мапінгу. Адже під час виконання динамічного методу ми брали IL код із методу нашого базового підходу, адже він найшвидший.

4.2 Аналіз результатів

Далі ми провели тестування, заміри та бенчмаркінг системи для встановлення результатів швидкості роботи та використання ресурсів системи кожного методу мапінгу. На рисунку 4.2 наведено результати відпрацювання системи[13].

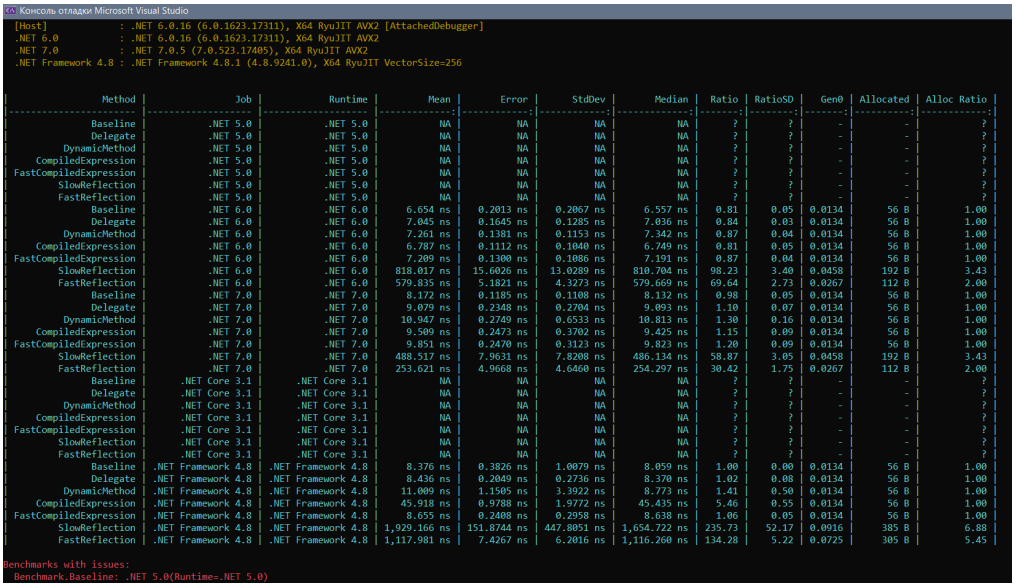


Рисунок 4.2 – Результат відпрацювання бенчмарку

Для структурованого відображення, результати було впорядковано. Впорядковані результати наведено на рисунку 4.3.

Method	Job	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen0	Allocated	Alloc Ratio
Baseline	Framework 4.8	7.794 ns	0.4898 ns	1.3571 ns	7.496 ns	1.00	0.00	0.0089	56 B	1.00
DynamicMethod	Framework 4.8	8.103 ns	0.2260 ns	0.4768 ns	8.054 ns	1.12	0.17	0.0089	56 B	1.00
Delegate	Framework 4.8	7.974 ns	0.2216 ns	0.4910 ns	7.940 ns	1.09	0.19	0.0089	56 B	1.00
CompiledExpression	Framework 4.8	39.070 ns	0.8423 ns	1.7016 ns	39.063 ns	5.45	0.88	0.0089	56 B	1.00
FastCompiledExpression	Framework 4.8	7.705 ns	0.2189 ns	0.4520 ns	7.730 ns	1.07	0.18	0.0089	56 B	1.00
SlowReflection	Framework 4.8	1,610.503 ns	32.0085 ns	63.9245 ns	1,611.465 ns	224.01	32.66	0.0610	385 B	6.88
FastReflection	Framework 4.8	1,051.206 ns	20.8693 ns	44.0204 ns	1,042.916 ns	144.90	21.24	0.0477	305 B	5.45
Baseline	.NET Core 3.1	8.708 ns	0.2633 ns	0.7682 ns	8.588 ns	1.16	0.22	0.0089	56 B	1.00
DynamicMethod	.NET Core 3.1	8.193 ns	0.2266 ns	0.5600 ns	8.254 ns	1.08	0.21	0.0089	56 B	1.00
Delegate	.NET Core 3.1	8.108 ns	0.2269 ns	0.4736 ns	8.101 ns	1.12	0.18	0.0089	56 B	1.00
CompiledExpression	.NET Core 3.1	9.232 ns	0.5360 ns	1.5805 ns	9.259 ns	1.21	0.25	0.0089	56 B	1.00
FastCompiledExpression	.NET Core 3.1	9.285 ns	0.3705 ns	1.0924 ns	9.443 ns	1.21	0.20	0.0089	56 B	1.00
SlowReflection	.NET Core 3.1	1,396.654 ns	94.5429 ns	278.7618 ns	1,250.371 ns	189.04	52.35	0.0610	384 B	6.86
FastReflection	.NET Core 3.1	846.239 ns	16.9672 ns	38.9849 ns	846.052 ns	114.68	20.01	0.0477	304 B	5.43
Baseline	.NET 5.0	6.064 ns	0.1778 ns	0.4775 ns	6.003 ns	0.80	0.14	0.0089	56 B	1.00
DynamicMethod	.NET 5.0	6.892 ns	0.1918 ns	0.3098 ns	6.831 ns	0.93	0.15	0.0089	56 B	1.00
Delegate	.NET 5.0	8.224 ns	0.2436 ns	0.7028 ns	8.164 ns	1.08	0.19	0.0089	56 B	1.00
CompiledExpression	.NET 5.0	7.738 ns	0.2609 ns	0.7610 ns	7.791 ns	1.02	0.20	0.0089	56 B	1.00
FastCompiledExpression	.NET 5.0	8.002 ns	0.3092 ns	0.8969 ns	7.934 ns	1.05	0.16	0.0089	56 B	1.00
SlowReflection	.NET 5.0	1,080.053 ns	21.4425 ns	47.9593 ns	1,078.085 ns	146.86	21.96	0.0610	384 B	6.86
FastReflection	.NET 5.0	735.505 ns	14.2707 ns	21.3598 ns	729.531 ns	97.61	16.96	0.0477	304 B	5.43
Baseline	.NET 6.0	7.159 ns	0.2613 ns	0.7622 ns	7.019 ns	0.95	0.18	0.0089	56 B	1.00
DynamicMethod	.NET 6.0	7.975 ns	0.2190 ns	0.2998 ns	7.886 ns	1.04	0.20	0.0089	56 B	1.00
Delegate	.NET 6.0	9.212 ns	0.3180 ns	0.8970 ns	9.139 ns	1.22	0.24	0.0089	56 B	1.00
CompiledExpression	.NET 6.0	8.006 ns	0.2431 ns	0.7129 ns	7.881 ns	1.06	0.19	0.0089	56 B	1.00
FastCompiledExpression	.NET 6.0	7.467 ns	0.2026 ns	0.4138 ns	7.398 ns	1.04	0.16	0.0089	56 B	1.00
SlowReflection	.NET 6.0	790.373 ns	15.8412 ns	22.7190 ns	786.504 ns	104.22	18.81	0.0305	192 B	3.43
FastReflection	.NET 6.0	840.921 ns	125.6284 ns	370.4181 ns	594.881 ns	103.01	44.64	0.0172	112 B	2.00
Baseline	.NET 7.0	9.319 ns	0.3673 ns	1.0655 ns	9.246 ns	1.23	0.25	0.0089	56 B	1.00
DynamicMethod	.NET 7.0	10.247 ns	0.3229 ns	0.9367 ns	10.045 ns	1.36	0.23	0.0089	56 B	1.00
Delegate	.NET 7.0	10.854 ns	0.5694 ns	1.6611 ns	10.593 ns	1.43	0.31	0.0089	56 B	1.00
CompiledExpression	.NET 7.0	8.408 ns	0.2392 ns	0.6902 ns	8.330 ns	1.10	0.18	0.0089	56 B	1.00
FastCompiledExpression	.NET 7.0	9.014 ns	0.2325 ns	0.5526 ns	8.988 ns	1.20	0.21	0.0089	56 B	1.00
SlowReflection	.NET 7.0	439.142 ns	8.7813 ns	15.1474 ns	436.838 ns	59.86	9.67	0.0305	192 B	3.43
FastReflection	.NET 7.0	250.846 ns	4.8429 ns	11.6961 ns	250.167 ns	33.11	6.15	0.0176	112 B	2.00

Рисунок 4.3 – Впорядковані результати

Також, було впорядковано результати за методами. Впорядковані результати за методами наведено на рисунку 4.4.

Method	Job	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen0	Allocated	Alloc Ratio
Baseline	Framework 4.8	7.794 ns	0.4898 ns	1.3571 ns	7.496 ns	1.00	0.00	0.0089	56 B	1.00
Baseline	.NET Core 3.1	8.708 ns	0.2633 ns	0.7682 ns	8.588 ns	1.16	0.22	0.0089	56 B	1.00
Baseline	.NET 5.0	6.064 ns	0.1778 ns	0.4775 ns	6.003 ns	0.80	0.14	0.0089	56 B	1.00
Baseline	.NET 6.0	7.159 ns	0.2613 ns	0.7622 ns	7.019 ns	0.95	0.18	0.0089	56 B	1.00
Baseline	.NET 7.0	9.319 ns	0.3673 ns	1.0655 ns	9.246 ns	1.23	0.25	0.0089	56 B	1.00
Delegate	Framework 4.8	7.974 ns	0.2216 ns	0.4910 ns	7.940 ns	1.09	0.19	0.0089	56 B	1.00
Delegate	.NET Core 3.1	8.108 ns	0.2269 ns	0.4736 ns	8.101 ns	1.12	0.18	0.0089	56 B	1.00
Delegate	.NET 5.0	8.224 ns	0.2436 ns	0.7028 ns	8.164 ns	1.08	0.19	0.0089	56 B	1.00
Delegate	.NET 6.0	9.212 ns	0.3180 ns	0.8970 ns	9.139 ns	1.22	0.24	0.0089	56 B	1.00
Delegate	.NET 7.0	10.854 ns	0.5694 ns	1.6611 ns	10.593 ns	1.43	0.31	0.0089	56 B	1.00
SlowReflection	Framework 4.8	1,610.503 ns	32.0085 ns	63.9245 ns	1,611.465 ns	224.01	32.66	0.0610	385 B	6.88
SlowReflection	.NET Core 3.1	1,396.654 ns	94.5429 ns	278.7618 ns	1,250.371 ns	189.04	52.35	0.0610	384 B	6.86
SlowReflection	.NET 5.0	1,080.053 ns	21.4425 ns	47.9593 ns	1,078.085 ns	146.86	21.96	0.0610	384 B	6.86
SlowReflection	.NET 6.0	790.373 ns	15.8412 ns	22.7190 ns	786.504 ns	104.22	18.81	0.0305	192 B	3.43
SlowReflection	.NET 7.0	439.142 ns	8.7813 ns	15.1474 ns	436.838 ns	59.86	9.67	0.0305	192 B	3.43
FastReflection	Framework 4.8	1,051.206 ns	20.8693 ns	44.0204 ns	1,042.916 ns	144.90	21.24	0.0477	305 B	5.45
FastReflection	.NET Core 3.1	846.239 ns	16.9672 ns	38.9849 ns	846.052 ns	114.68	20.01	0.0477	304 B	5.43
FastReflection	.NET 5.0	735.505 ns	14.2707 ns	21.3598 ns	729.531 ns	97.61	16.96	0.0477	304 B	5.43
FastReflection	.NET 6.0	840.921 ns	125.6284 ns	370.4181 ns	594.881 ns	103.01	44.64	0.0172	112 B	2.00
FastReflection	.NET 7.0	250.846 ns	4.8429 ns	11.6961 ns	250.167 ns	33.11	6.15	0.0176	112 B	2.00
CompiledExpression	Framework 4.8	39.070 ns	0.8423 ns	1.7016 ns	39.063 ns	5.45	0.88	0.0089	56 B	1.00
CompiledExpression	.NET Core 3.1	9.232 ns	0.5360 ns	1.5805 ns	9.259 ns	1.21	0.25	0.0089	56 B	1.00
CompiledExpression	.NET 5.0	7.738 ns	0.2609 ns	0.7610 ns	7.791 ns	1.02	0.20	0.0089	56 B	1.00
CompiledExpression	.NET 6.0	8.006 ns	0.2431 ns	0.7129 ns	7.881 ns	1.06	0.19	0.0089	56 B	1.00
CompiledExpression	.NET 7.0	8.408 ns	0.2392 ns	0.6902 ns	8.330 ns	1.10	0.18	0.0089	56 B	1.00
FastCompiledExpression	Framework 4.8	7.705 ns	0.2189 ns	0.4520 ns	7.730 ns	1.07	0.18	0.0089	56 B	1.00
FastCompiledExpression	.NET Core 3.1	9.285 ns	0.3705 ns	1.0924 ns	9.443 ns	1.21	0.20	0.0089	56 B	1.00
FastCompiledExpression	.NET 5.0	8.002 ns	0.3092 ns	0.8969 ns	7.934 ns	1.05	0.16	0.0089	56 B	1.00
FastCompiledExpression	.NET 6.0	7.467 ns	0.2026 ns	0.4138 ns	7.398 ns	1.04	0.16	0.0089	56 B	1.00
FastCompiledExpression	.NET 7.0	9.014 ns	0.2325 ns	0.5526 ns	8.988 ns	1.20	0.21	0.0089	56 B	1.00
DynamicMethod	Framework 4.8	8.103 ns	0.2260 ns	0.4768 ns	8.054 ns	1.12	0.17	0.0089	56 B	1.00
DynamicMethod	.NET Core 3.1	8.193 ns	0.2266 ns	0.5600 ns	8.254 ns	1.08	0.21	0.0089	56 B	1.00
DynamicMethod	.NET 5.0	6.892 ns	0.1918 ns	0.3098 ns	6.831 ns	0.93	0.15	0.0089	56 B	1.00
DynamicMethod	.NET 6.0	7.975 ns	0.2190 ns	0.2998 ns	7.886 ns	1.04	0.20	0.0089	56 B	1.00
DynamicMethod	.NET 7.0	10.247 ns	0.3229 ns	0.9367 ns	10.045 ns	1.36	0.23	0.0089	56 B	1.00

Рисунок 4.4 – Впорядковані результати за методами

Перед аналізом – важливо зазначити що ми не розраховували у нашому бенчмарку час «холодного старту».

Для початку структуруємо результати у таблицю 4.2.

Таблиця 4.2 – Результати бенчмаркінгу(таблиця виконана самостійно)

Метод	Патерн швидкості	Патерн відносно платформи
Baseline Method	Загалом має найнижчі значення затримки на всіх версіях .NET.	Стабільно швидкий на різних платформах, вказуючи на ефективність простого прямого.
DynamicMethod	Загалом швидший за інші складніші методи, але повільніший за Baseline.	Варіації швидкості не великі між різними версіями .NET, демонструючи стабільність цього підходу.
CompiledExpression та FastCompiledExpression	Поліпшення швидкості на новіших версіях .NET, що може вказувати на оптимізації в JIT-компіляторі та кращу підтримку виразів.	На платформах .NET Core та .NET 5+ швидкість зростає, але на .NET Framework є значні затримки.

За результатами ми бачимо деякі важливі та неочевидні патерни, такі як:

- наш динамічний метод – повільніший за expression, fust expression;
- fust expression api – яка повинна бути швидшою за звичайний expressionпрацює повільніше. хоча вони[6] заявляють що швидше(див. рис. 4.5).

Invoking the compiled delegate (comparing to the direct constructor call):

Method	Mean	Error	StdDev	Median	Ratio	RatioSD	Gen 0	Gen 1	Gen 2
DirectConstructorCall	7.736 ns	0.2472 ns	0.6336 ns	7.510 ns	0.57	0.05	0.0102	-	-
CompiledLambda	13.917 ns	0.2723 ns	0.3818 ns	13.872 ns	1.03	0.04	0.0102	-	-
FastCompiledLambda	13.412 ns	0.2355 ns	0.4124 ns	13.328 ns	1.00	0.00	0.0102	-	-

Рисунок 4.5 – Заява FustCompile бібліотеки про перевагу над звичайною компіляцією

Отже далі потрібно з’ясувати чому ми отримали такі дивні та неочевидні результати.

Аналізуючи результати важливо відмітити що наш метод Baseline працював на минулих версіях платформи швидше ніж на .Net7, а на Net6 повільныше за Net5.

Також чітко видно роботу із рефлексією відносно зростання версій, її швидкість зростала, та використовувала менше пам’яті.

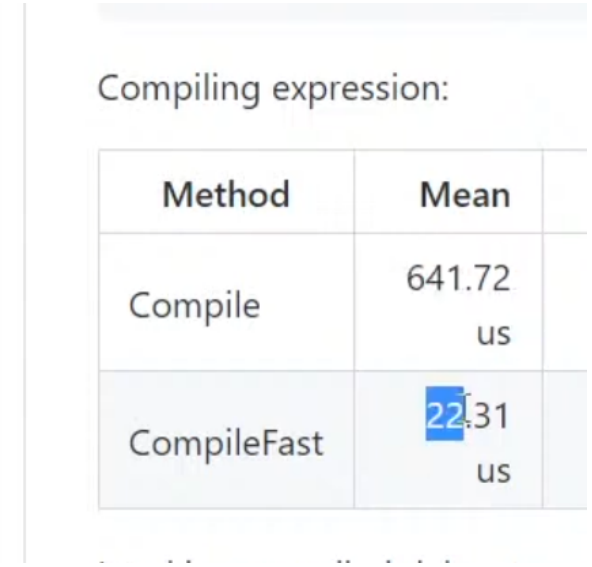
Аналізуючи наші патерни та проблеми із ними – ми з’ясували причину чому FustCompile повільніше звичайної компіляції(див.рис.4.6).

```
CompiledExpression |Framework 4.8 |    39.070 ns |
CompiledExpression |.NET Core 3.1 |     9.232 ns |
CompiledExpression |      .NET 5.0 |     7.738 ns |
CompiledExpression |      .NET 6.0 |     8.006 ns |
CompiledExpression |      .NET 7.0 |     8.408 ns |
```

Рисунок 4.6 – аналіз FustCompile

Отже тут ми бачимо причину появи FustCompile – з’явилась вона тому що на Net4.8 звичайна компіляція працювала дуже повільно, тож якщо використовувати цю варсію – варто використовувати відповідно FustCompile.

Також варто зазначити що хоча FustCompile бібліотека і не працює швидше компіляції – але вона працює ефективніше на початковому етапі компіляції, у «холодному старті»(див.рис.4.7).


Compiling expression:

Method	Mean
Compile	641.72 us
CompileFast	22.31 us

Рисунок 4.7 – Швидкість компіляції холодного старту

Отже очевидно відмітити що наразі, на актуальній платформі Net7 – швидкість та ефективність мапінгу оптимальніша на звичайній CompileExpression, це навіть ефективніше ніж робота напряму із IL генерації.

ВИСНОВКИ

У ході дослідження було розглянуто дві провідні бібліотеки для мапінгу даних в .NET - Mapster та AutoMapper. Mapster виявився більш продуктивним у більшості сценаріїв, забезпечуючи швидше мапінг і, в деяких випадках, менше споживання пам'яті. Ця бібліотека відома своєю гнучкістю та простотою в налаштуванні, що робить її популярним вибором для багатьох розробників.

Також були розглянуті різні архітектурні підходи до створення механізмів мапінгу, зокрема використання Source Generators та Expression Trees. Source Generators пропонують автоматизацію генерації коду мапінгу під час компіляції, що підвищує продуктивність і зменшує навантаження в рантаймі. Expression Trees, у свою чергу, надають гнучкість у створенні складних мапінгів і можуть бути інтегровані з ORM-фреймворками для ефективної взаємодії з базами даних.

Було визначено найефективнішу бібліотеку для мапінгу, проранжовано відносно версії платформи. Це було зроблено за допомогою бенчмарку реалізованого нами із використанням BenchmarkDotNet.

Нами було проведено порівняння різних методів мапінгу за допомогою бенчмарк модулю в який ми імпліментували різні методи та різні елементи збору метрик, для наших методів: BaseLine, DynamicMethod, Delegate, CompiledExpression, FastCompiledExpression, SlowReflection, та FastReflection. Це допомогло встановити, який з методів найбільш підходить для конкретних сценаріїв використання.

Це дослідження допоможе розробникам прийняти обґрунтоване рішення про використання тієї чи іншої бібліотеки або підходу до мапінгу, виходячи з потреб їхнього проекту або версії платформи.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз предметної області \ A Guide to Choosing the Right .NET Object Mapper? 2023р., URL: https://medium.com/@justindevx/a-guide-to-choosing-the-right-net-object-mapper-ab13c5623a70 (дата звернення: 23.01.2024).

2. Аналіз існуючих рішень \ Object Mappers in .Net, Shalin De Silva 2023р., URL: https://medium.com/@shalinds/object-mappers-in-net-975f64cc643d (дата звернення: 12.02.2024).

3. Mapping Objects with AutoMapper in .NET, URL: https://medium.com/c-sharp-progarmming/mapping-objects-with-automapper-in-net-b60a9444e470 (дата звернення: 13.02.2024).

4. What are the different approaches to Object-Object., URL: https://medium.com/@shalinds/object-mappers-in-net-975f64cc643d (дата звернення: 14.02.2024).

5. How to Map Object-to-Object using Automapper in .NET Core, URL: https://www.c-sharpcorner.com/article/how-to-map-object-to-object-using-automapper-in-net-core/ (дата звернення: 21.03.2024).

6. Аналіз мапінг бібліотек \ HigLabo.Mapper, Creating Fastest Object Mapper in the World with Expression Tree, 2024р., URL: https://www.codeproject.com/Articles/5275388/HigLabo-Mapper-Creating-Fastest-Object-Mapper-in-t (дата звернення: 12.03.2024).

7. Аналіз source generators \ Comparison of Object Mapper Libraries, Jon Dodd, 2021р., URL: https://dev.to/jdinnovensa/comparison-of-object-mapper-libraries-gm2 (дата звернення: 10.03.2024).

8. Аналіз reflection ammit \ Comparison of Object Mapper Libraries, Jon Dodd, 2021р., URL: https://dev.to/jdinnovensa/comparison-of-object-mapper-libraries-gm2 (дата звернення: 15.03.2024).

9. FastCompileExpression бібліотека \ Fast Compiler for C# Expression Trees and the lightweight LightExpression alternative.? 2024р., URL: https://github.com/dadhi/FastExpressionCompiler (дата звернення: 01.04.2024).

10. Optimization Factors in Modeling and Testing Hardware and Semiconductor Defects by Dynamic Discrete Event Simulation \ Arabian, J. H., Видавництво: ХНУРЕ, 2009р., URL: https://openarchive.nure.ua/entities/publication/be918fba-8755-4d34-be6d-fc1464089d77 (дата звернення: 01.04.2024).

11. 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: 05.04.2024).

12. DATA EXCHANGE MODEL IN THE INTERNET OF THINGS CONCEPT / I. Afanasieva et al. Telecommunications and Radio Engineering. 2019. Vol. 78,no. 10.P. 869–878.
URL: https://doi.org/10.1615/telecomradeng.v78.i10.30 (дата звернення: 11.06.2024).

13. Аналіз результатів бенчмаркіунгу URL: https://doi.org/10.1615/telecomradeng.v78.i10.30 (дата звернення: 11.06.2024)