https://www.proconference.org/index.php/gec/article/view/gec32-00-018
DOI: 10.30890/2709-1783.2024-32-00-018

УДК 004.4:004.6

LOAD TESTING AND DATA RETRIEVAL OPTIMIZATION IN ENTITY FRAMEWORK CORE

НАВАНТАЖУВАЛЬНЕ ТЕСТУВАННЯ ТА ОПТИМІЗАЦІЯ ОТРИМАННЯ ДАНИХ У ENTITY FRAMEWORK CORE

Biloborodko O.I. / Білобородько О.І.
c.t.s. / к.т.н.
ORCID: 0000-0003-4025-183
Voronin R.V. / Воронін Р.В.
student / студент
Oles Honchar Dnipro National University,
Dnipro, Nauky Avenue, 72, 49010
Дніпровський національний університет імені Олеся Гончара,
Дніпро, пр. Науки, 72, 49010

Анотація. *У даній роботі розглядаються сучасні методики та інструменти для оптимізації доступу до даних і тестування програмного забезпечення, які сприяють підвищенню продуктивності та стабільності вебдодатків і систем з великими обсягами даних. Особлива увага приділяється використанню Entity Framework Core та аналізу обмежень, пов'язаних із продуктивністю цього інструменту. Також досліджується роль навантажувального тестування у виявленні та усуненні проблем в програмному забезпеченні, а інструмент AutoFixture використовується для автоматизації та підвищення ефективності процесу тестування. Ці інструменти та методики демонструють свою значимість для розробки надійних і ефективних програмних рішень.*

Ключові слова: *Entity Framework Core, оптимізація доступу до даних, тестування програмного забезпечення, вебдодатки, навантажувальне тестування, AutoFixture.*

Abstract. *This work examines contemporary methods and tools for optimizing data access and software testing, which enhance the productivity and stability of web applications and large data systems. Special attention is paid to the use of Entity Framework Core and the analysis of performance limitations associated with this tool. Additionally, the role of load testing in identifying and resolving software issues is explored, and the tool AutoFixture is used to automate and enhance the efficiency of the testing process. These tools and methodologies demonstrate their importance in developing reliable and effective software solutions.*

Key words: *Entity Framework Core, data access optimization, software testing, web applications, load testing, AutoFixture.*

Вступ.

У сучасному світі розробки програмного забезпечення, особливо в контексті вебдодатків та даних великих обсягів, ефективне управління даними та оптимізація доступу до них стають критично важливими завданнями. Entity Framework Core (EF Core) як один із лідируючих ORM (Object-Relational Mapping) інструментів, що пропонується Microsoft для .NET платформи, забезпечує розробників потужними засобами для моделювання та роботи з даними. Проте, незважаючи на зручність та гнучкість, які пропонує EF Core, проблеми з продуктивністю та масштабуванням можуть серйозно вплинути на загальну ефективність додатку. Тому під час розробки програмного забезпечення необхідно враховувати зазначені аспекти та приділяти увагу навантажувальному тестуванню та оптимізації доступу до даних.

Основний текст

Навантажувальне тестування є незамінним інструментом у визначенні та вирішенні проблем з продуктивністю та масштабуванням додатків. Цей процес дозволяє розробникам не тільки виявити слабкі місця у додатку, але й зрозуміти, як програма поводиться під час великих навантажень, що є ключовим для гарантування стабільності та відповідності до вимог реального часу. Особлива увага у цьому контексті приділяється оптимізації запитів до бази даних, зменшенню затримок та підвищенню продуктивності отримання даних.

Використання інструменту AutoFixture для генерації тестових даних може значно спростити процес тестування додатків, зокрема при розробці з використанням Entity Framework Core. Процес створення тестових об'єктів відіграє ключову роль у підготовці до тестування програмних систем. AutoFixture представляє собою бібліотеку для платформи .NET, розроблену для автоматизації процесів генерації тестових екземплярів об'єктів, спрямовану на оптимізацію аранжування тестів, що дозволяє ефективно моделювати та структурувати тестові сценарії. Її використання дозволяє автоматично ініціювати екземпляри будь-яких класів, автоматично заповнюючи їх атрибути псевдовипадково генерованими даними. Це значно скорочує час, необхідний для ручного введення значень, особливо при роботі з об'ємними або складними даними [1].

Крім того, AutoFixture надає можливість модифікації та налаштування процесу генерації, що дозволяє розробникам встановлювати специфічні алгоритми для створення даних, адаптованих до конкретних вимог проєкту. Таке налаштування забезпечує більший контроль над створюваними об'єктами та їх відповідністю до специфікацій.

Автогенерація тестових даних також забезпечує виявлення потенційних помилок чи дефектів, які можуть залишитися непоміченими при використанні традиційно сформованих тестових наборів. Завдяки широкій варіативності генерованих даних збільшується ймовірність виявлення непередбачуваних вад у програмному забезпеченні, підвищуючи, таким чином, загальну надійність і безпеку додатків.

На рисунку 1 представлений код, який демонструє використання бібліотеки AutoFixture для автоматизованого генерування тестових даних у системі управління рецептами. У цьому прикладі, за допомогою AutoFixture створюється 10000 унікальних об'єктів типу Recipe, кожен з яких ініціалізується випадково згенерованими значеннями для властивостей, які в сукупності й формують опис рецепту. Кожен рецепт містить наступні властивості:

- Guid AuthorId;
- string Name;
- string Description;
- string Instructions;
- int Portions.

```
public async Task GenerateTestData(CancellationToken token)
{
    var fixture = new Fixture();
    fixture.Behaviors.Add(new OmitOnRecursionBehavior());
    fixture.Customize<Recipe>(c => c
        .With(r => r.AuthorId, Guid.NewGuid())
        .With(r => r.Name, fixture.Create<string>().Substring(0, 10))
    );

    for (int i = 0; i < 10000; i++)
    {
        Recipe recipe = fixture.Create<Recipe>();
        await _recipeRepository.Add(recipe);
    }
}
```

Рисунок 1 – Приклад використання AutoFixture для створення тестових даних

Особливість цього методу полягає в налаштуванні об'єктів Recipe, де для кожного рецепту встановлюється унікальний AuthorId та випадкова назва, обмежена першими 10 символами. Генерування та завантаження такого набору даних займає близько 3 хвилин. Заповнення такої кількості даних тестувальником власноруч потребувало б значно більших часових витрат.

Entity Framework Core є відомим фреймворком, який дозволяє значно спрощувати роботу з базами даних у додатках, розроблених на платформі .NET. Водночас він може стикатися з проблемами швидкості виконання запитів, що впливає на загальну продуктивність додатків. Проблема N+1 запитів є однією з найбільш розповсюджених у використанні ORM, коли для отримання пов'язаних даних система спочатку виконує запит для отримання основних об'єктів, а потім окремі запити для кожного з цих об'єктів, щоб здобути додаткові дані. Такий підхід може значно збільшити загальну кількість запитів до бази даних.

Використання багатопотоковості може стати ефективним способом вирішення проблеми низької швидкості запитів у EF Core, оскільки дозволяє паралельне виконання операцій, які не залежать одна від одної, та зменшити час відповіді системи. Особливо це актуально в сценаріях, коли треба обробляти велику кількість даних або виконувати складні обчислення.

Entity Framework Core підтримує асинхронні операції з базами даних, які можуть бути використані для покращення продуктивності за допомогою асинхронних методів, таких як ToListAsync(), FirstOrDefaultAsync() та інші. Використання асинхронних методів дозволяє програмі продовжувати виконання інших завдань, поки чекає на завершення вводу/виводу операцій, таких як запити до бази даних. Це може значно покращити реактивність і загальну продуктивність вебдодатків і сервісів. Однак, при використанні багатопотоковості важливо ретельно планувати архітектуру додатку, щоб уникнути проблем, пов'язаних з конкуренцією та блокуванням ресурсів. Це передбачає правильне управління життєвим циклом контекстів бази даних, уникнення ділення стану між потоками, і забезпечення потокобезпечності операцій. Також, для більш ефективного використання багатопотоковості існує

можливість використання більш низькорівневих оптимізацій, таких як паралельне виконання не пов'язаних між собою запитів у різних потоках або розподіл обробки даних на декілька сервісів або вузлів. Такий підхід може зменшити загальний час обробки запитів і забезпечити краще масштабування додатку [2].

Розумне застосування багатопотоковості в EF Core може стати ключовим підходом до підвищення продуктивності та ефективності додатків, але вимагає глибокого розуміння потенційних викликів та уважного підходу до реалізації.

На рисунку 2 можна побачити приклад використання багатопотоковості у контексті завантаження даних за допомогою Entity Framework Core. У цьому фрагменті коду демонструється розподіл процесу запиту даних на декілька потоків, що виконуються асинхронно, з метою оптимізації витрат часу та збільшення продуктивності.

```
17 references
public async Task<List<TEntity>> Get(Expression<Func<TEntity, bool>> predicate, CancellationToken token = default)
{
    var entitiesCount = await _dbSet.CountAsync(token);
    var threadsNum = ThreadsNum;

    var tasks = new List<Task<List<TEntity>>>();
    for (int pageNum = 0; pageNum < threadsNum - 1; pageNum++)
    {
        int pageSize = entitiesCount / threadsNum;
        int localPageNum = pageNum;
        tasks.Add(Task.Run(() =>
        {
            using RecipeDbContext dbContext = new(_contextOptions);
            return IncludeNavigationProperties(dbContext)
                    .OrderBy(e => e.Id)
                    .Where(predicate)
                    .Skip(localPageNum * pageSize)
                    .Take(pageSize)
                    .ToList();
        }));
    }

    tasks.Add(Task.Run(() =>
    {
        using RecipeDbContext dbContext = new(_contextOptions);
        return IncludeNavigationProperties(dbContext)
                .OrderBy(e => e.Id)
                .Where(predicate)
                .Skip((threadsNum - 1) * (entitiesCount / threadsNum))
                .Take(entitiesCount - ((threadsNum - 1) * (entitiesCount / threadsNum)))
                .ToList();
    }));
    await Task.WhenAll(tasks);

    var results = tasks.SelectMany(task => task.Result);


    return results.ToList();
}
```

Рисунок 2 – Фрагмент коду з використанням багатопотоковості для оптимізації роботи отримання даних EF Core

Функція ініціює асинхронне завантаження об'єктів, розподіляючи їх за різними потоками на основі загальної кількості доступних об'єктів та визначеного числа потоків. Кожен потік обробляє свою частину даних, реалізуючи пагінацію через оператори Skip і Take. Таке розділення дозволяє зменшити навантаження на один потік, розподіляючи обробку даних між декількома, що призводить до зменшення часу відповіді від бази даних.

Завершення всіх потоків синхронізується за допомогою Task.WhenAll, яка забезпечує те, що виконання програми продовжиться лише після того, як всі асинхронні завдання будуть завершені. Результати з кожного потоку збираються у єдиний список, що дозволяє об'єднати дані для подальшої обробки або відображення.

Для визначення часу отримання записів з бази даних був написаний метод, код якого наведено на рисунку 3. Цей метод отримує усі записи з бази даних та повертає витрачений на це час в мілісекундах та кількість отриманих записів.

```
[AllowAnonymous]
[HttpGet("TestGetAll")]
0 references
public async Task<IActionResult> TestGetAll(CancellationToken token)
{
    Stopwatch stopwatch = new();

    stopwatch.Start();

    var result = await _recipeService.GetAllAsync(token);

    stopwatch.Stop();

    var results = new
    {
        Time = stopwatch.ElapsedMilliseconds,
        RecipesCount = result.Count,
    };

    return Ok(results);
}
```

Рисунок 3 – Метод для фіксування часу виконання запиту та кількості отриманих записів

На рисунку 4 можна побачити результати отримання записів в одному потоці, а на рисунку 5 – результати використання 100 потоків для виконання того ж запиту. Час виконання запиту у випадку використання 100 потоків у 8,2 рази менше порівняно з одним потоком.

Response body

```
{
  "time": 24798,
  "recipesCount": 10356
}
```

Рисунок 4 – Результат виконання запиту в одному потоці

Response body

```
{
  "time": 3007,
  "recipesCount": 10356
}
```

Рисунок 5 – Результат виконання запиту в 100 потоках

Цей підхід до виконання асинхронних операцій з використанням багатопотоковості є важливим для ефективного управління ресурсами та підвищення загальної продуктивності великих та складних застосунків, які вимагають інтенсивного взаємодії з базами даних.

Висновки.

Були розглянуті сучасні методики та інструменти оптимізації доступу до даних і тестування програмного забезпечення, які є ключовими для підвищення продуктивності та стабільності вебдодатків та систем з великими обсягами даними. Entity Framework Core виявився незамінним у моделюванні та обробці даних, хоча його ефективність може бути обмежена в певних випадках. Навантажувальне тестування відіграє вирішальну роль у виявленні та усуненні потенційних проблем, що дозволяє розробникам оптимізувати додатки для реальних умов експлуатації. Також було розглянуто інструмент AutoFixture, який допомагає оптимізувати процес тестування, забезпечуючи розробникам потужні засоби для створення різноманітних тестових сценаріїв, що сприяє виявленню непередбачуваних помилок. Використання розглянутих інструментів та підходів є важливим кроком у напрямку створення більш надійних і ефективних програмних рішень.

Література:

1 Бібліотека з відкритим кодом AutoFixture для .NET URL: https://autofixture.github.io/ (дата звернення: 18.04.2024)

2. Офіційна документація Entity Framework Core URL: https://learn.microsoft.com/en-us/ef/core/ (дата звернення: 18.04.2024)

Статья отправлена: 19.04.2024 р.