МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ПрАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»

Кафедра інформаційних технологій

ДО ЗАХИСТУ ДОПУЩЕНА

Зав. кафедри _____

д.е.н., доц. С.І. Левицький

МАГІСТЕРСЬКА ДИПЛОМНА РОБОТА

ДОСЛІДЖЕННЯ ПРОЦЕСІВ ОПТИМІЗАЦІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗА ДОПОМОГОЮ КЕШУВАННЯ

Виконав

ст. гр. ІПЗ–111м _____ І.Д. Єременко

Науковий керівник

к.т.н., доц. _____ Ю.С. Резніченко

Запоріжжя

2023 р.

ПРАТ «ПВНЗ «ЗАПОРІЗЬКИЙ ІНСТИТУТ ЕКОНОМІКИ
ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ»
Кафедра інформаційних технологій

ЗАТВЕРДЖУЮ
Зав. кафедри
д.е.н., доцент Левицький С.І.
_____
03.10.2022 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ ДИПЛОМНУ РОБОТУ

ст. гр. ІПЗ–111м, спеціальності 121 «Інженерія програмного забезпечення»
ОП «Інженерія програмного забезпечення»
Єременка Івана Дмитровича

1.Тема: Дослідження процесів оптимізації програмного забезпечення за допомогою кешування
затверджена наказом по інституту № 02-16 від 03.10.2022 р.
2. Термін здачі студентом закінченої роботи: 12.01.2023 р.
3. Перелік питань, що підлягають розробці:

1. Виконати аналіз патернів об'єктно-орієнтованого та функціонального проектування та методів роботи з кешами
2. Створити концепцію модульної інтеграції кешів
3. Розробити бібліотеку Caching у ASP .NET Core (C#) та спроектувати «інтерфейс» для роботи з нею
4. Розробити Unit тести та Benchmark тести для бібліотеки Caching

4. Календарний графік підготовки магістерської дипломної роботи

№ етапу	Зміст	Терміни виконання	Готовність по графіку %, підпис керівника	Підпис керівника про повну готовність етапу, дата
1.	Формулювання теми магістерської дипломної роботи (збір практичного матеріалу за темою магістерської дипломної роботи)	20.10.22		
2.	І атестація І розділ магістерської дипломної роботи	27.10.22		
3.	ІІ атестація ІІ розділ магістерської дипломної роботи	17.11.22		
4.	ІІІ атестація ІІІ розділ магістерської дипломної роботи, висновки та рекомендації, додатки, реферат, перевірка програмою «Антиплагіат»	29.12.22		
5.	Доопрацювання магістерської дипломної роботи, підготовка презентації, отримання відгуку керівника та рецензії	10.01.23		
6.	Попередній захист магістерської дипломної роботи	12.01.23		
7.	Надання магістерської дипломної роботи на кафедру	за 3 дні до захисту		
8.	Захист магістерської дипломної роботи	19.01.23		

Дата видачі завдання: 03.10.2022 р.

Керівник магістерської дипломної роботи _____ Ю.С. Резніченко
(підпис) (прізвище та ініціали)

Завдання отримав до виконання _____ І.Д. Єременко
(підпис студента) (прізвище та ініціали)

РЕФЕРАТ

Магістерська дипломна робота містить 62 сторінки, 0 таблиць, 39 рисунків, 3 додатки, 3 лістинги, 26 бібліографічних посилань.

Метою магістерської дипломної роботи є створення концепції модульної інтеграції кешів та розробка на її основі ASP .NET Core (C#) бібліотеки.

Задачі магістерської дипломної роботи:

1. Виконати аналіз патернів об'єктно-орієнтованого та функціонального проектування.
2. Створити концепцію модульної інтеграції кешів.
3. Розробити бібліотеку Caching у ASP .NET Core (C#) та спроектувати «інтерфейс» для роботи з нею.
4. Розробити Unit тести та Benchmark тести для бібліотеки Caching.

Об'єктом дослідження є методи та програмні засоби кешування. Предметом дослідження є модульна інтеграція кешів та ASP .NET Core (C#) бібліотека. Використано методи системного аналізу, методи об’єктно-орієнтованого та функціонального проектування.

У першому розділі розглянуто патерни об'єктно-орієнтованого та функціонального проектування, методи роботи з кешами. Виконано аналіз програмних засобів кешування.

У другому розділі наведено огляд та переваги використання у розробці бібліотеки Caching наступного стеку технологій: ASP .NET Core, C#, JetBrains Rider, Redis, MemCached, NuGet, NUnit, Moq, FluentAssertions, Benchmark .NET.

У третьому розділі створено концепцію модульної інтеграції кешів та модель категоризованих ключів записів у кеші. Наведено опис архітектури та функціональних можливостей ASP .NET Core (C#) бібліотеки Caching. Наведено опис Unit та Benchmark тестування, а також розгортання бібліотеки Caching.

КЕШ, МОДУЛЬНА ІНТЕГРАЦІЯ, C#, ASP .NET CORE, REDIS, JETBRAINS RIDER, NUGET, MEMCACHED, MOQ, NUNIT, FLUENTASSERTIONS

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ....3
ВСТУП....4
РОЗДІЛ 1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ....6
1.1. Кеш та кеш-пам’ять....6
1.1.1 Розподілений кеш....8
1.1.2 Типи розподілених кешів....9
1.2. Патерни проектування....10
1.2.1 Патерни функціонального проектування....11
1.2.2 Патерни об’єктно-орієнтованого проектування....14
РОЗДІЛ 2 ІНСТРУМЕНТИ РОЗРОБКИ....19
2.1 Мова програмування C#....19
2.2 Платформа .NET....20
2.3 Бібліотека NUnit .NET....23
2.4 Бібліотека Moq .NET....26
2.5 Бібліотека BenchmarkDotNet....27
2.6 NuGet....32
2.7 JetBrains Rider....33
2.8 Redis....34
2.9 Memcached....36
РОЗДІЛ 3 РОЗРОБКА БІБЛІОТЕКИ CACHING У ASP .NET CORE (C#)....38
1.1 Архітектура та можливості бібліотеки Caching....38
1.2 CachingFlag....39
1.3 Категоризація ключів записів у кеші....41
1.4 Caching<T, TStoreFlag>....43
1.5 StoreOperationProvider....43
1.6 Cache<T, TStoreFlag>....43
1.7 ILoader<TArgs, TResult>....44
1.8 CachingLoader<TArgs, TEntity, TStoreFlag>....44

1.9 CacheStoreDecorator....46
1.10 «Безфлажні» кеші....49
1.10 Тестування бібліотеки Caching....50
1.11 Розгортання бібліотеки Caching у NuGet....50
ВИСНОВКИ....52
ПЕРЕЛІК ПОСИЛАНЬ....53
ДОДАТКИ....54
ДОДАТОК А....54
ДОДАТОК Б....56
ДОДАТОК В....57

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ ТА ТЕРМІНІВ

Слово/словосполучення	Скорочення	Умови використання
База даних	БД	У тексті
Центральний процесор	CPU	У тексті
Введення/виведення	I/O	У тексті
Оперативна пам’ять	RAM	У тексті
Мовний інтегрований запит	LINQ	У тексті
Common Type System	CTS	У тексті
Common Language Specification	CLS	У тексті
Dynamically Linked Library	DLL	У тексті
Microsoft Intermediate Language	MSIL	У тексті

ВСТУП

Процеси кешування відіграють важливу роль у сучасній розробці програмного забезпечення. Можливо цілком успішно розробляти без кешів, але найчастіше це призводить до проблем із швидкодією програмного забезпечення. В свою чергу це призводить до витрат та необхідності додаткової підтримки серверу.

Інтеграція з cash дозволяє розробнику не тільки зменшити навантаження на host, де розгорнуто програмне забезпечення, а й оптимізувати час виконання програми, зменшити витрати на підтримку host (особливо у випадку хмарного ресурсу), скоротити час розробки (особливо за рахунок етапу debug) тощо. Отже використання cash при розробці програмного забезпечення дозволяє підвищити ефективність процесу.

При роботі (або інтеграції) з кешами «людський фактор» має суттєвий вплив, як і при будь-якій іншій розробці. Наприклад, помилки при формуванні ключу запису у кеші приводять до bugs, що потребує довготривалого усунення (витрата часу розробки). Наприклад, неправильно налагоджена (часто не уніфікована) серіалізація/десеріалізація значень записів у кеші приводить до «прихованих» помилок, коли відсутнє поле/об’єкт у десеріалізованому графі, коли типи полів/об’єктів (особливо у випадках роботи з Node.js та JavaScript) є неправильними тощо. Саме тому робота з кешами має бути якомога більше контрольованою для розробника.

Для роботи з кешами доречно реалізувати сувору типізацію, виключення дублювання та структуризацію ключів записів у кеші, уніфіковану серіалізацію/десеріалізацію значень, що будуть закешовані. З цією метою доцільно розглянути патерни об’єктно-орієнтованого та функціонального проектування.

З метою перевірки стабільної роботи бібліотеки доцільно розробляти Unit тести. З метою перевірки швидкодії бібліотеки доцільно розробляти Benchmark тести. Бібліотека є додатковим «шаром» для роботи з кешами − швидкодія не буде такою ж, як при роботі з кешами «без шарів» між кодом-клієнтом кеша та кешем. З цього слідує факт (або навіть інваріант), що інтеграція з кешами за допомогою бібліотеки буде займати більше процесорного часу, ніж «пряма» робота з кешами через більшу кількість викликів методів, більшу кількість логічних конструкцій тощо.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Кеш та кеш-пам’ять

Кеш-пам’ять — це апаратне або програмне забезпечення, яке використовується для тимчасового зберігання даних у комп’ютерному середовищі [1-3].

Це невеликий об’єм швидшої та дорожчої пам’яті, який використовується для покращення продуктивності даних, до яких нещодавно або часто звертались. Кешовані дані тимчасово зберігаються на доступному носії, локальному для клієнта кешу та окремо від основного сховища. Кеш зазвичай використовується центральним процесором (CPU), програмами, веб-браузерами та операційними системами.

Кеш використовується тому, що масове або основне сховище не може відповідати вимогам клієнтів. Кеш скорочує час доступу до даних, зменшує затримку та покращує введення/виведення (I/O). Оскільки майже всі робочі навантаження програми залежать від операцій введення-виведення, процес кешування покращує продуктивність програми.

Коли клієнт кешу намагається отримати доступ до даних, він спочатку перевіряє кеш. Якщо там знайдено дані, це називається зверненням до кешу. Відсоток спроб, які призвели до попадання в кеш, називається коефіцієнтом або коефіцієнтом звернення до кешу.

Запитані дані, яких немає в кеші («промахи» кешу), витягуються з основної пам’яті та копіюються в кеш. Як це робиться та які дані вилучаються з кешу, щоб звільнити місце для нових даних, залежить від алгоритму кешування, протоколів кешу та системних політик, що використовуються.

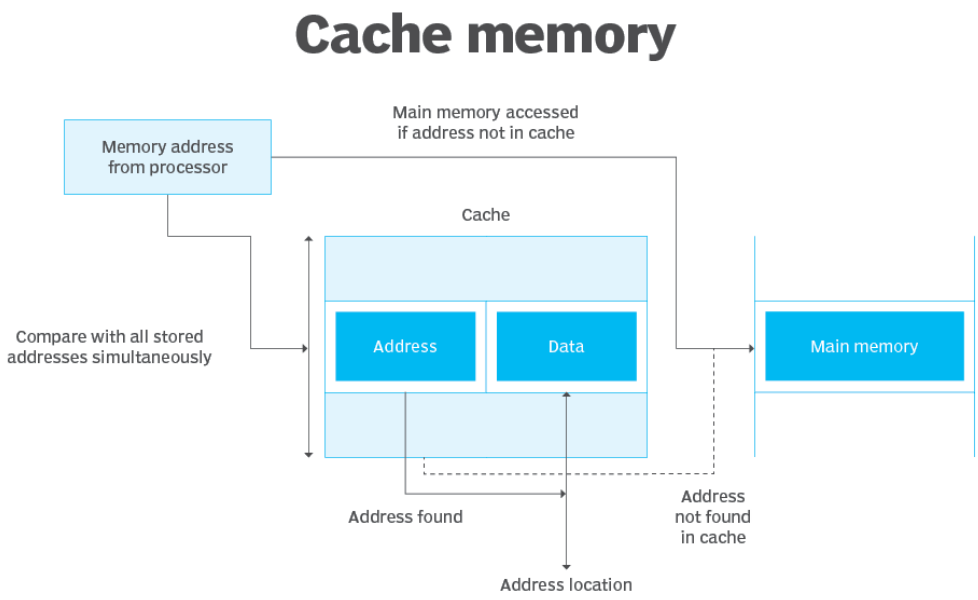


Рисунок 1.1 − Діаграма роботи процесору з кеш-пам’яттю

Такі веб-браузери, як Safari, Firefox та Chrome, використовують кешування веб-переглядача, щоб покращити продуктивність часто відвідуваних веб-сторінок. Коли користувач відвідує веб-сторінку, запитувані файли зберігаються в кеші для цього браузера в комп’ютерній пам’яті користувача.

Щоб отримати раніше відвідану сторінку, браузер отримує більшість необхідних файлів із кешу, а не надсилає їх повторно з веб-сервера. Цей підхід називається кеш-читання. Браузер може зчитувати дані з кешу браузера швидше, ніж перечитувати файли з веб-сторінки.

Розглянемо причини важливості кешу:

1. Використання кешу зменшує затримку активних даних. Це призводить до підвищення продуктивності системи або програми.
2. Кеш перенаправляє введення-виведення, скорочуючи операції введення-виведення до зовнішнього накопичувача та нижчих рівнів мережі зберігання даних
3. Дані можуть постійно зберігатися в традиційному сховищі або зовнішніх масивах зберігання. Це забезпечує узгодженість та цілісність даних за

допомогою таких функцій, як знімки та реплікація, надані сховищем або масивом.

Кеш-пам'ять або включена до CPU, або вбудована до мікросхеми на системній платі. У новіших машинах єдиний спосіб збільшити кеш-пам’ять – оновити системну плату та CPU до останнього покоління. Старі системні плати можуть мати порожні слоти, які можна використовувати для збільшення кеш-пам’яті.

1.1.1 Розподілений кеш

Розподілений кеш [4, 5] — це система, яка об’єднує оперативну пам’ять (RAM) кількох мережевих комп’ютерів у одне сховище даних у пам’яті, яке використовується як кеш даних для забезпечення швидкого доступу до даних. Хоча більшість кеш-пам’яті традиційно розміщується в одному фізичному сервері чи апаратному компоненті, розподілений кеш може вирости за межі пам’яті одного комп’ютера, об’єднавши разом кілька комп’ютерів, що називається розподіленою архітектурою або розподіленим кластером, для збільшення ємності та потужності обробки.

Розподілені кеші особливо корисні у середовищах з великим обсягом даних та навантаженням. Розподілена архітектура дозволяє поступове розширення/масштабування за рахунок додавання додаткових комп’ютерів до кластера, дозволяючи кеш-пам’яті зростати відповідно до зростання даних.

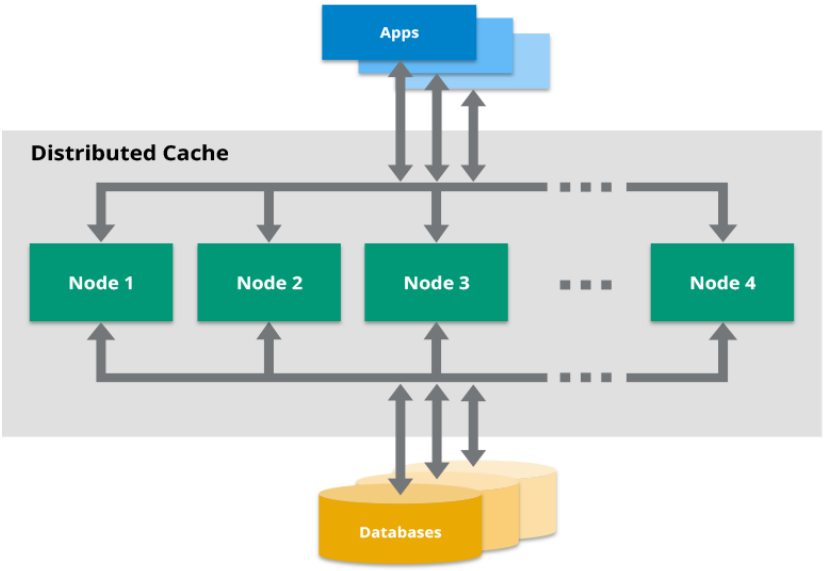


Рисунок 1.2 − Діаграма архітектури додатку с використаням розподілених кешів

1.1.2 Типи розподілених кешів

Популярними розподіленими кешами [6, 7], що використовуються у галузі розробки web-додатків, є Ehcache, Memcache, Redis, Riak, Hazelcast тощо.

Memcached [8] використовується Google Cloud у своїй платформі як службі. Це високопродуктивне розподілене сховище «ключ-значення», яке в основному використовується для зменшення навантаження на базу даних. Це як велика хеш-таблиця, розподілена між кількома машинами. Увімкнення доступу до даних за O(1) тобто постійний час.

Окрім пари «ключ-значення», Redis [9] — це розподілена система з відкритим вихідним кодом у пам’яті − також підтримує інші структури даних, такі як розподілені списки, черги, рядки, набори, відсортовані набори. Окрім кешування, Redis також часто розглядають як сховище даних NoSQL.

Ehcache [10] – це універсальна система розподіленого кешування загального призначення з відкритим вихідним кодом для Java-додатків та сервлетів. Ehcache створено для підвищення продуктивності, зниження навантаження на базу даних та спрощення масштабування. Ehcache є найбільш широко викори-

стовуваною системою кешування на основі мови Java, тому що вона надійна, перевірена, повнофункціональна та інтегрується з багатьма популярними бібліотеками та фреймворками. Ehcache надає як кешування в межах одного екземпляра програми, так і розподілене кешування з даними розміром Тб.

Riak KV [11] - це розподілена NoSQL база даних типу «ключ-значення», з розширеною локальною та мультикластерною реплікацією, яка гарантує читання та запис навіть у разі збоїв обладнання або мережевих розділів. Riak KV має швидку продуктивність, оскільки автоматизує розподіл даних за кластером, а також має досить надійну безперервність бізнесу з нехарактерною архітектурою, яка забезпечує високу доступність, і масштабується лінійно з використанням апаратного забезпечення.

Hazelcast [12] — це розподілена платформа обчислень та зберігання даних для запитів із незмінною затримкою, агрегації та обчислень із збереженням стану щодо потоків подій і традиційних джерел даних. Це дозволяє швидко створювати ресурсозберігаючі програми в режимі реального часу. Можливо розгорнути його в будь-якому масштабі від невеликих периферійних пристроїв до великого кластера хмарних примірників. Кластер вузлів Hazelcast спільно використовує як сховище даних, так і обчислювальне навантаження, яке може динамічно масштабуватися вгору та вниз. При додаванні до кластера, дані автоматично перебалансовуються в кластері, а поточні обчислювальні завдання знімають їх стан та масштаб із гарантією обробки.

1.2. Патерни проектування

Патерни проектування є типовими рішеннями типових проблем в дизайні програмного забезпечення. Кожен патерн схожий на план який розробник можете налаштувати для вирішення конкретної проблеми у коді [13].

З п’яти етапів розробки програмного забезпечення патерни проектування майже не використовуються на етапах аналізу, тестування чи документування. Патерни проектування мають найбільший вплив на етапі проектування:

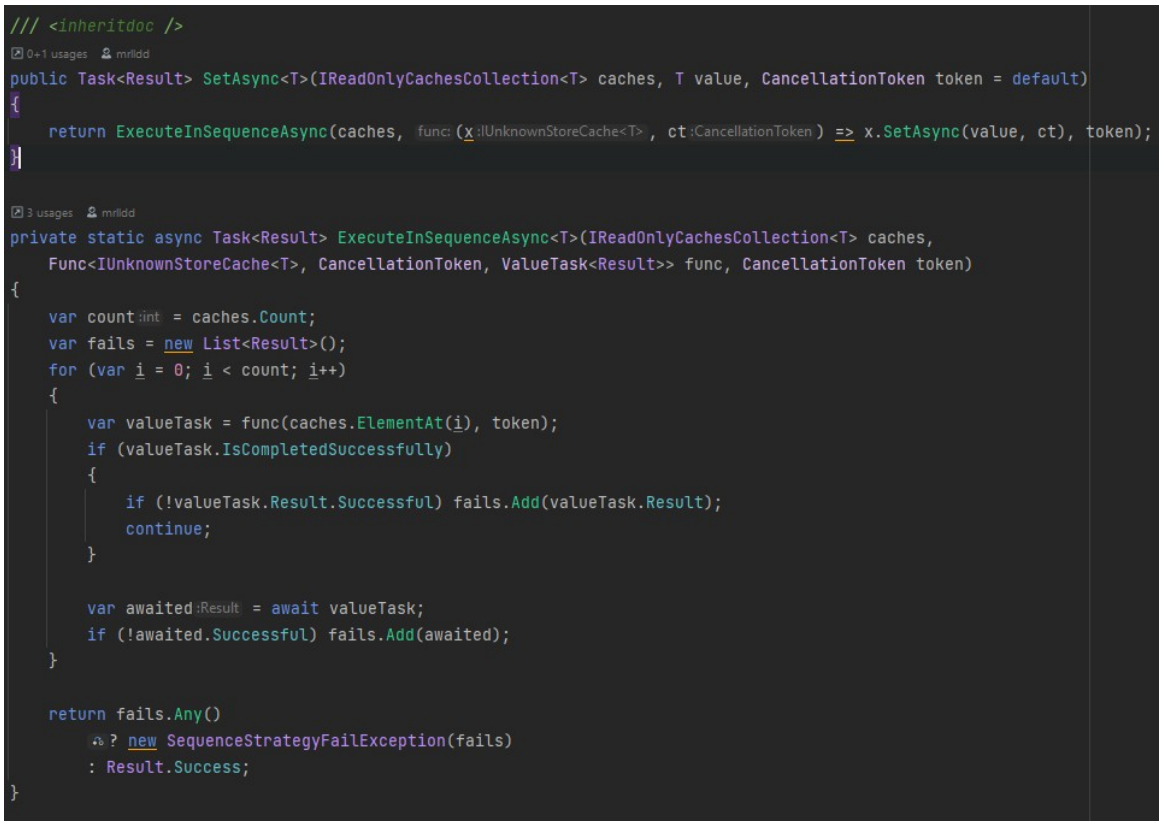
1. Патерни проектування допомагають аналізувати більш абстрактні області програми, надаючи конкретні, добре перевірені рішення.
2. Патерни проектування допомагають швидше писати код.
3. Патерни проектування підтримують повторне використання коду та ураховують зміни, надаючи добре перевірені механізми для делегування та композиції, а також інші методи повторного використання без успадкування.
4. Патерни проектування сприяють більш розбірливому та зручному для обслуговування коду.
5. Патерни проектування все частіше забезпечують «спільну мову».

1.2.1 Патерни функціонального проектування

Функція вищого порядку − це функція, яка приймає функцію як аргумент або повертає функцію. Функція вищого порядку відрізняється від функцій першого порядку, які не приймають функцію як аргумент і не повертають функцію як результат [14].

Монада − це шаблон проектування програмного забезпечення зі структурою, яка об’єднує фрагменти програми (функції) та загортає їх значення, що повертаються, у тип із додатковим обчисленням. На додаток до визначення монадного типу обгортки, монади визначають два оператори: один для обгортання значення в тип монади, а інший для складання разом функцій, які виводять значення типу монади (монадні функції). У мовах загального призначення монади використовуються, щоб скоротити шаблонний код, необхідний для звичайних операцій (наприклад, робота з невизначеними значеннями чи помилковими функціями, інкапсуляція шаблонного коду тощо). Функ-

ціональні мови використовують монади, щоб перетворити складні послідовності функцій у стислі конвеєри, які абстрагують потік керування та побічні ефекти [14].

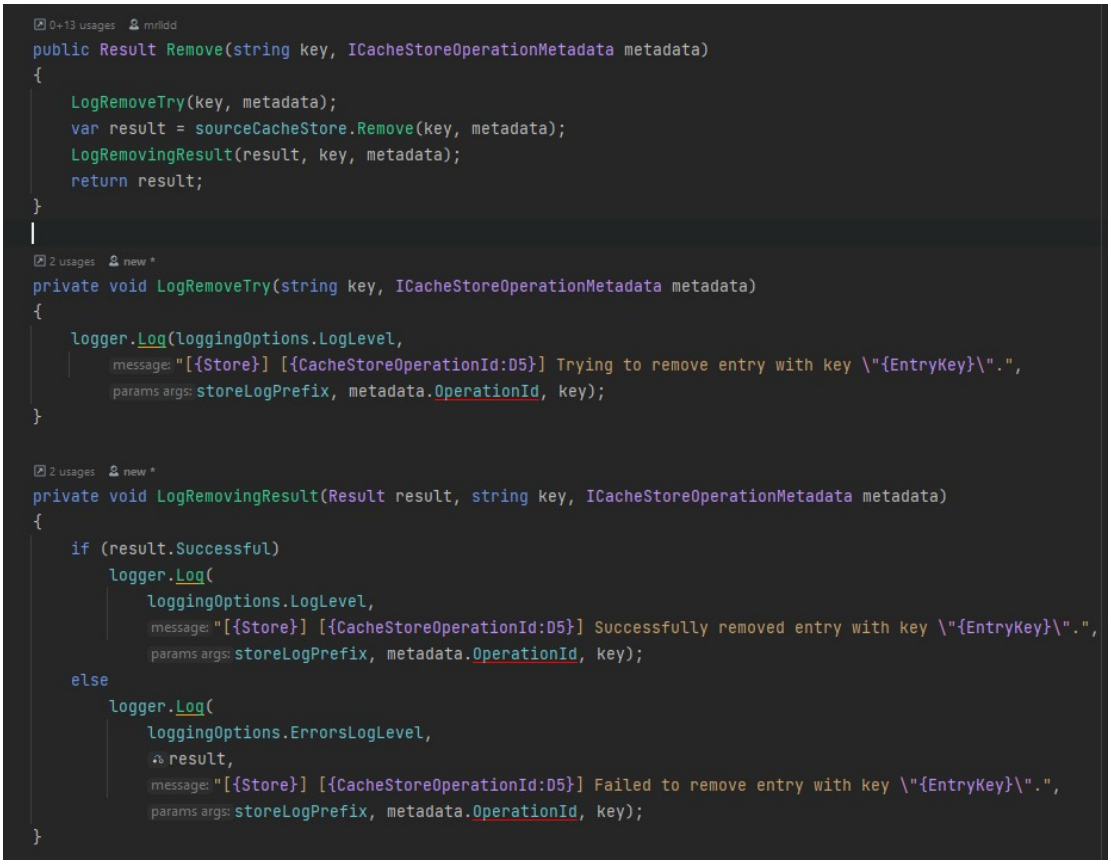


```
/// <inheritdoc />
0+1 usages  mrlldd
public Task<Result> SetAsync<T>(IReadOnlyCachesCollection<T> caches, T value, CancellationToken token = default)
{
    return ExecuteInSequenceAsync(caches, func: (x :IUnknownStoreCache<T> , ct :CancellationToken ) => x.SetAsync(value, ct), token);
}

3 usages  mrlldd
private static async Task<Result> ExecuteInSequenceAsync<T>(IReadOnlyCachesCollection<T> caches,
    Func<IUnknownStoreCache<T>, CancellationToken, ValueTask<Result>> func, CancellationToken token)
{
    var count :int  = caches.Count;
    var fails = new List<Result>();
    for (var i = 0; i < count; i++)
    {
        var valueTask = func(caches.ElementAt(i), token);
        if (valueTask.IsCompletedSuccessfully)
        {
            if (!valueTask.Result.Successful) fails.Add(valueTask.Result);
            continue;
        }

        var awaited :Result  = await valueTask;
        if (!awaited.Successful) fails.Add(awaited);
    }

    return fails.Any()
        ? new SequenceStrategyFailException(fails)
        : Result.Success;
}
```

Рисунок 1.3 − Використання патерну функції вищого порядку

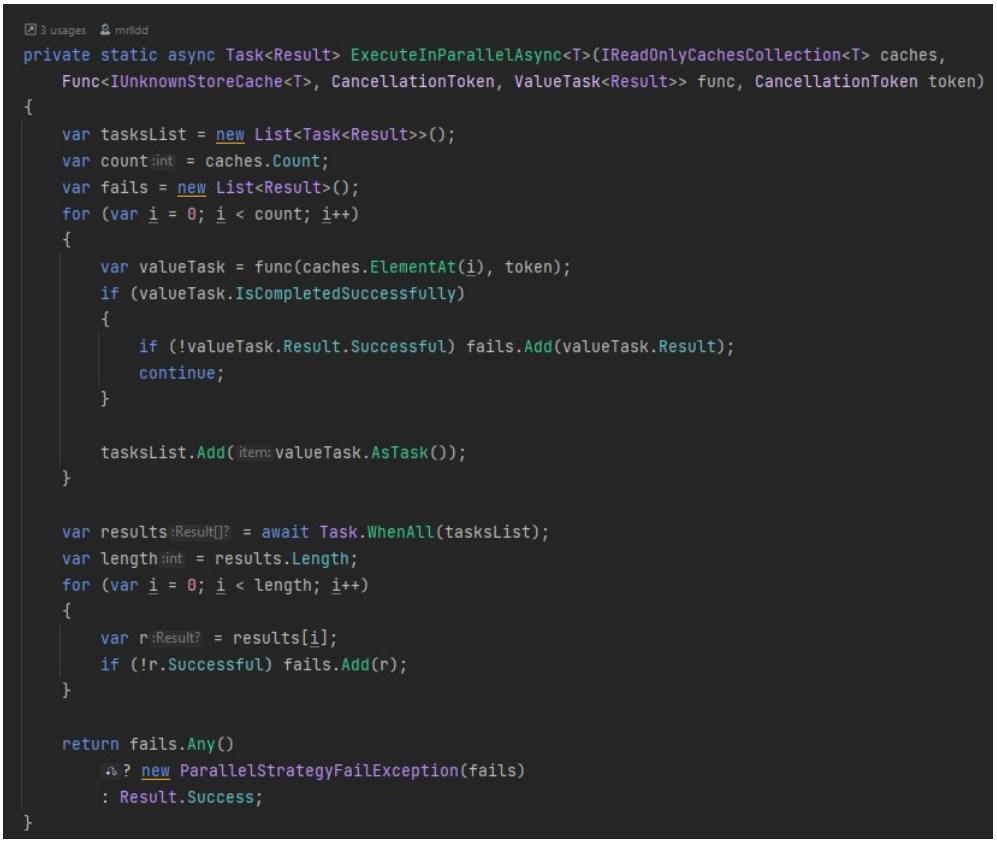
Монада Result - монада, яка представляє умови успіху та невдачі. У серії операцій зв’язування, якщо будь-яка функція повертає монаду помилки, наступні зв’язки пропускаються. Це дозволяє легко контролювати потік як для успішних, так і для помилкових випадків [14].

Перша «нода» монади створюється в найнижчому «шарі» бібліотеки – сервісів, що являють собою огортку на клієнт-кешу – у Store (що можуть бути декоровані, використання монади Result облегшує цей процес).



```
0+13 usages  mrlldd
public Result Remove(string key, ICacheStoreOperationMetadata metadata)
{
    LogRemoveTry(key, metadata);
    var result = sourceCacheStore.Remove(key, metadata);
    LogRemovingResult(result, key, metadata);
    return result;
}

2 usages  new *
private void LogRemoveTry(string key, ICacheStoreOperationMetadata metadata)
{
    logger.Log(loggingOptions.LogLevel,
        message: "[{Store}] [{CacheStoreOperationId:D5}] Trying to remove entry with key \"{EntryKey}\".",
        params args: storeLogPrefix, metadata.OperationId, key);
}

2 usages  new *
private void LogRemovingResult(Result result, string key, ICacheStoreOperationMetadata metadata)
{
    if (result.Successful)
        logger.Log(
            loggingOptions.LogLevel,
            message: "[{Store}] [{CacheStoreOperationId:D5}] Successfully removed entry with key \"{EntryKey}\".",
            params args: storeLogPrefix, metadata.OperationId, key);
    else
        logger.Log(
            loggingOptions.ErrorsLogLevel,
            result,
            message: "[{Store}] [{CacheStoreOperationId:D5}] Failed to remove entry with key \"{EntryKey}\".",
            params args: storeLogPrefix, metadata.OperationId, key);
}
```

Рисунок 1.4 − Використання монади Result у Decorator



```
3 usages  mrlldd
private static async Task<Result> ExecuteInParallelAsync<T>(IReadOnlyCachesCollection<T> caches,
    Func<IUnknownStoreCache<T>, CancellationToken, ValueTask<Result>> func, CancellationToken token)
{
    var tasksList = new List<Task<Result>>();
    var count :int = caches.Count;
    var fails = new List<Result>();
    for (var i = 0; i < count; i++)
    {
        var valueTask = func(caches.ElementAt(i), token);
        if (valueTask.IsCompletedSuccessfully)
        {
            if (!valueTask.Result.Successful) fails.Add(valueTask.Result);
            continue;
        }

        tasksList.Add( item: valueTask.AsTask());
    }

    var results :Result[]? = await Task.WhenAll(tasksList);
    var length :int = results.Length;
    for (var i = 0; i < length; i++)
    {
        var r :Result? = results[i];
        if (!r.Successful) fails.Add(r);
    }

    return fails.Any()
        ? new ParallelStrategyFailException(fails)
        : Result.Success;
}
```

Рисунок 1.5 − Використання монади Result у паралельних розрахунках

```
1+16 usages  mrlldd
public ValueTask<Result<T>> GetAsync<T>(string key, ICacheStoreOperationMetadata metadata,
    CancellationToken token = default)
{
    var task = Result.Of( asyncFactory: async () =>
    {
        var fromCache :string? = await distributedCache.GetStringAsync(key, token);
        return string.IsNullOrEmpty(fromCache)
            ? throw new CacheMissException(key)
            : Deserialize<T>(fromCache) ?? throw new DeserializationFailException(key, fromCache);
    });
    return new ValueTask<Result<T>>(task);
}
```

Рисунок 1.6 − Використання монади Result у запиті до кешу

1.2.2 Патерни об’єктно-орієнтованого проектування

Об’єктно-орієнтовані патерни проектування зазвичай показують зв’язки та взаємодію між класами чи об’єктами, не вказуючи кінцеві задіяні класи додатків чи об’єкти. Патерни, які передбачають змінний стан, можуть бути непридатними для функціональних мов програмування. Деякі шаблони можуть бути непотрібними у мовах, які мають вбудовану підтримку для вирішення про-

блеми, яку вони намагаються вирішити, а об’єктно-орієнтовані шаблони не обов’язково підходять для не об’єктно-орієнтованих мов [15].

Builder — креативний шаблон проектування, який дозволяє крок за кроком створювати складні об’єкти. Патерн дозволяє створювати різні типи та представлення об’єкта, використовуючи той самий код конструкції [15].

Рисунок 1.7 − Використання патерну Builder при реєстрації сервісів

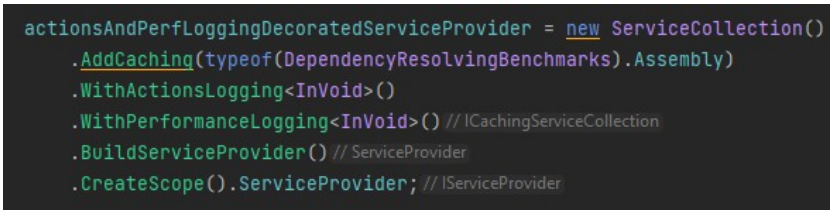
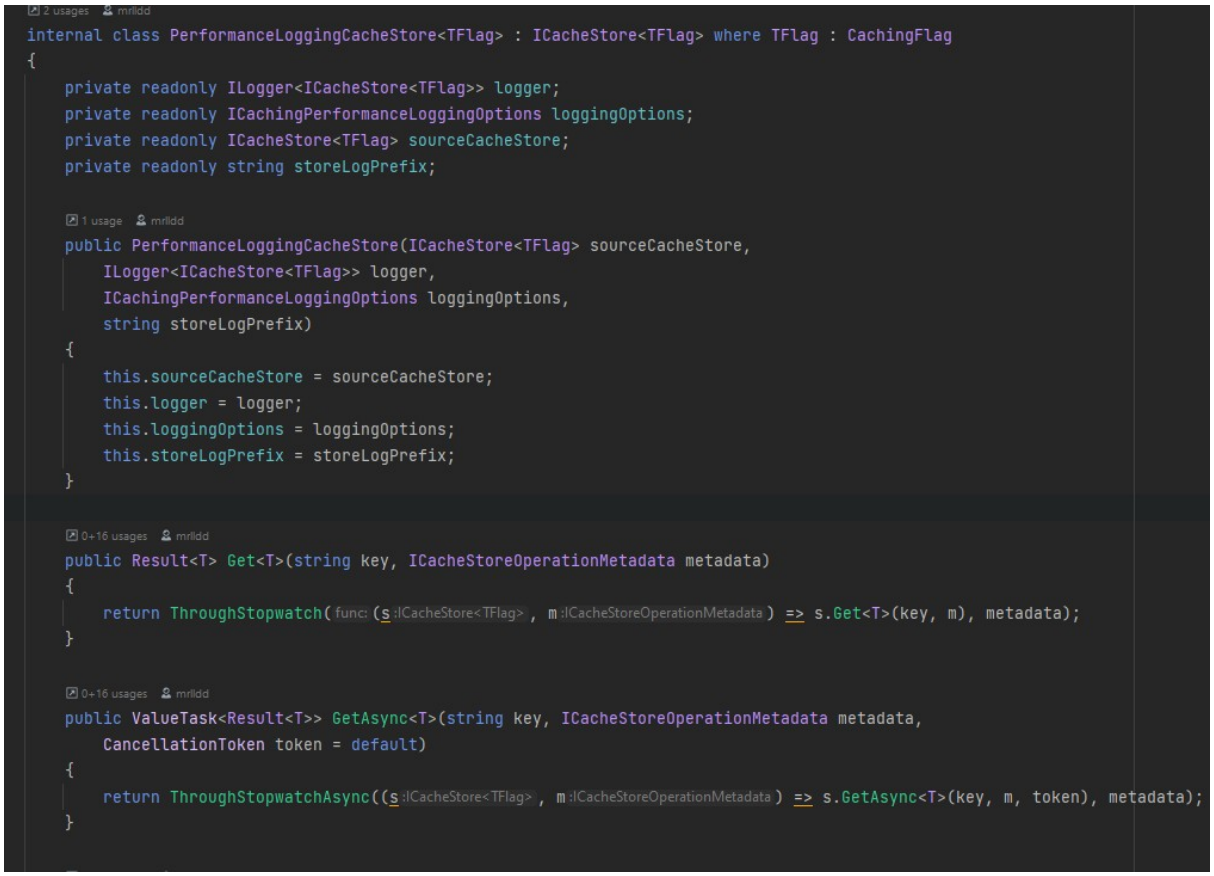


Рисунок 1.8 − Використання патерну Builder

Decorator — це структурний патерн проектування, який дає змогу додавати нову поведінку до об’єктів, розміщуючи ці об’єкти всередині спеціальних об’єктів-обгорток, які містять поведінку [15].



```
2 usages  mrlldd
internal class PerformanceLoggingCacheStore<TFlag> : ICacheStore<TFlag> where TFlag : CachingFlag
{
    private readonly ILogger<ICacheStore<TFlag>> logger;
    private readonly ICachingPerformanceLoggingOptions loggingOptions;
    private readonly ICacheStore<TFlag> sourceCacheStore;
    private readonly string storeLogPrefix;

    1 usage  mrlldd
    public PerformanceLoggingCacheStore(ICacheStore<TFlag> sourceCacheStore,
        ILogger<ICacheStore<TFlag>> logger,
        ICachingPerformanceLoggingOptions loggingOptions,
        string storeLogPrefix)
    {
        this.sourceCacheStore = sourceCacheStore;
        this.logger = logger;
        this.loggingOptions = loggingOptions;
        this.storeLogPrefix = storeLogPrefix;
    }

    0+16 usages  mrlldd
    public Result<T> Get<T>(string key, ICacheStoreOperationMetadata metadata)
    {
        return ThroughStopwatch( func: (s :ICacheStore<TFlag> , m :ICacheStoreOperationMetadata ) => s.Get<T>(key, m), metadata);
    }

    0+16 usages  mrlldd
    public ValueTask<Result<T>> GetAsync<T>(string key, ICacheStoreOperationMetadata metadata,
        CancellationToken token = default)
    {
        return ThroughStopwatchAsync((s :ICacheStore<TFlag> , m :ICacheStoreOperationMetadata ) => s.GetAsync<T>(key, m, token), metadata);
    }
```

Рисунок 1.9 − Використання патерну Decorator для огортання операцій до кешу

Factory Method — це шаблон створення, який надає інтерфейс для створення об’єктів, але дозволяє реалізаціям змінювати тип об’єктів, які будуть створюватися [15].

```
internal class StoreOperationProvider : IStoreOperationProvider
{
    private int currentId = 1;

    0+9 usages  mrlldd
    public ICacheStoreOperationMetadata Next(string cacheKeyDelimiter)
    {
        return new CacheStoreOperationMetadata(Interlocked.Increment(ref currentId), cacheKeyDelimiter);
    }
}
```

Рисунок 1.10 − Використання патерну Factory Method для створення метаданих

Strategy — це шаблон поведінкового проектування, який дозволяє визначити сімейство алгоритмів, помістити кожен із них у окремий клас та зробити їхні об’єкти взаємозамінними [15].

```
internal class Cache<T> : ICache<T>
{
    mrlldd
    public Cache(IReadOnlyCachesCollection<T> instances)
    {
        Instances = instances;
    }

    9+3 usages
    public IReadOnlyCachesCollection<T> Instances { get; }

    mrlldd
    public Task<Result<T>> GetAsync(CancellationToken token = default)
    {
        return GetAsync(GetFirstSuccessfulStrategy.Instance, token);
    }

    mrlldd
    public Result<T> Get()
    {
        return Get(GetFirstSuccessfulStrategy.Instance);
    }

    1 usage  mrlldd
    public Task<Result<T>> GetAsync(ICacheGetStrategy strategy, CancellationToken token = default)
    {
        return strategy.GetAsync(Instances, token);
    }

    1 usage  mrlldd
    public Result<T> Get(ICacheGetStrategy strategy)
    {
        return strategy.Get(Instances);
    }
```

Рисунок 1.11 − Використання патерну Strategy у «безфлажному» кеші

1.3 Аналіз програмних засобів кешування

Найчастіше комп'ютеризовану систему створюють у вигляді веб-додатку. Головною перевагою цього підходу є використання «тонкого клієнта». Одним із головних завдань є забезпечення швидкодії такої системи. Сьогодні розповсюджений шлях розв’язання цього завдання полягає у кешуванні даних.

У даній роботі розглянуто різні підходи щодо кешування даних та програмні засоби їхньої реалізації [16].

За механікою роботи відокремлено такі види кешування:

- «лінивий» кеш, що зберігає дані й віддає їх, поки кеш не застаріє;

- синхронізований кеш, коли клієнт разом із даними отримує мітку часу й під час наступного звернення запитує, чи не змінилися дані, щоб повторно їх не отримувати;
- кеш наскрізного запису, коли будь-яка зміна даних одночасно відбувається як у джерелі даних, так й у кеші.

За типом даних відокремлено такі види кешування:

- кешування даних, які відображаються користувачу;
- кешування даних, які використовують для формування html-розмітки.

За місцем розташування кешу відокремлено такі види кешування:

- кешування на стороні клієнта;
- локальний кеш веб-сервера;
- кешування сервером додатків.

У даній роботі виявлено, що, незважаючи на широке використання кешування, загальних правил вибору системи кешування не існує. Найчастіше вибір програмних засобів кешування здійснюється розробником на основі власного досвіду та вподобань. Тому актуальною є задача дослідження ефективності програмних засобів кешування для різних обсягів даних. Метою даного дослідження є отримання залежностей часу запису/зчитування від кількості даних у кеші та визначення ефективного програмного засобу кешування у результаті експериментів.

У результаті експериментального аналізу програмних засобів кешування у даній роботі зроблено висновки:

1. Програмні засоби кешування даних у оперативній пам’яті значно швидші, ніж файловий кеш.
2. Для великої кількості даних використання файлового кешу недоцільне, оскільки кожне значення зберігається у окремому файлі. Тестування файлового кешу для кількості ітерацій понад 5000 було перервано через тривалий час виконання.
3. Час запису даних перевищує час зчитування даних.

4. Для програмних засобів кешування даних у оперативній пам’яті час запису практично не залежить від кількості даних, що вже містяться в кеші.
5. Для програмних засобів кешування даних у оперативній пам’яті час зчитування незначно збільшується під час зростання кількості даних, що містяться в кеші.

У результаті аналізу програмних засобів кешування у даній роботі сформульовано:

1. У якості основного кешу вирішено обрати програмний засіб, що використовує для зберігання даних оперативну пам’ять.
2. Якщо ресурси оперативної пам’яті обмежені, то додатково вирішено використати файловий кеш для кешування до 5000 сутностей.

1.4 Висновки до першого розділу

У першому розділі розглянуто патерни об'єктно-орієнтованого та функціонального проектування, методи роботи з кешами. Виконано аналіз програмних засобів кешування.

РОЗДІЛ 2
ІНСТРУМЕНТИ РОЗРОБКИ

2.1 Мова программування C#

C# [17] − об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. C# надає мовні конструкції для безпосередньої підтримки цих концепцій, роблячи C# природною мовою для створення та використання програмних компонентів. До C# були додані функції для підтримки нових робочих навантажень і нових методів проектування

програмного забезпечення. За своєю суттю C# є об'єктно-орієнтованою мовою. Розробник визначає типи та їх поведінку.

Кілька функцій C# допомагають створювати надійні та довговічні програми. «Збір сміття» автоматично відновлює пам’ять, зайняту недоступними невикористаними об’єктами. Типи, які допускають значення NULL, захищають від змінних, які не посилаються на виділені об’єкти. Обробка винятків забезпечує структурований і розширений підхід до виявлення та відновлення помилок. Лямбда-вирази підтримують методи функціонального програмування. Синтаксис мовного інтегрованого запиту (LINQ) створює загальний шаблон для роботи з даними з будь-якого джерела. Підтримка мови для асинхронних операцій забезпечує синтаксис для побудови розподілених систем. C# має уніфіковану систему типів. Усі типи C# успадковуються від одного кореневого типу об’єкта. Усі типи мають набір загальних операцій. Цінності будь-якого типу можна зберігати, транспортувати та використовувати узгоджено. C# підтримує як визначені користувачем типи посилань, так і типи значень. C# дозволяє динамічно розподіляти об'єкти та вбудовано зберігати полегшені структури. C# підтримує загальні методи та типи, які забезпечують підвищену безпеку та продуктивність типів. C# надає ітератори, які дають змогу реалізаторам класів колекції визначати користувацьку поведінку для клієнтського коду.

2.2 Платформа .NET

Корпорацією Microsoft запропоновано новаторський компонентно-орієнтований підхід до програмування, який є розвитком об'єктно-орієнтованого спрямування. Відповідно до цього підходу, інтеграція об'єктів (можливо, гетерогенної природи) проводиться на основі інтерфейсів, що представляють ці об'єкти (або фрагменти програм) як незалежні компоненти. Такий підхід суттєво полегшує написання та взаємодія програмних компонент у гетерогенному середовищі проектування та реалізації. Стандартизується зберігання та повторне використання компонент програмного проекту в умовах розподіленої

мережевої середовища обчислень, де різні комп'ютери та користувачі обмінюються інформацією, наприклад, взаємодіючи у рамках дослідницького або бізнес-проекту [18].

Істотною перевагою слід вважати і можливість практичної реалізації принципу «будь-яка сутність є об'єктом» в гетерогенному програмному середовищі. Багато в чому це стало можливим завдяки удосконаленій, узагальненій системі типізації Common Type System (CTS).

Сувора ієрархічність організації просторів для типів, класів та імен сутностей програми дозволяє стандартизувати та уніфікувати реалізацію.

Новий підхід до інтеграції компонент додатків в середовищі обчислень Internet (веб-сервіси) дає можливість прискореного створення програм для широкого кола користувачів.

Універсальний інтерфейс. NET забезпечує інтегроване проектування та реалізацію компонентів додатків, розроблених відповідно до різних підходів до програмування.

Істотною перевагою конструктивного рішення. NET є компонентно-орієнтований підхід до проектування та реалізації програмного забезпечення. Суть підходу полягає у принциповій можливості створення незалежних складових програмного забезпечення з уніфікованою інтерфейсною частиною для багаторазового повторного та розподіленого використання. При цьому продуктивність рішення обумовлена багатомовністю інтегрованих програмних проектів.

У ході компіляції програма на. NET-сумісній мові програмування трансформується відповідно до заздалегідь заданої узагальненою специфікацією мови CTS. Система типів CTS повністю описує всі типи даних, які підтримує середовищем виконання, визначає їх взаємозв'язки та зберігає їх відображення в системі типів. NET.

Під Common Language Specification (CLS) розуміється набір правил, що визначають підмножина узагальнених типів даних, щодо яких гарантується, що

вони безпечні при використанні у всіх мовах. NET. Інтерфейси реалізуються за допомогою ASP.NET для веб-додатків.

У ході виконання процедури трансляції вихідний текст програми перетворюється компілятором у assembly та зберігається у динамічно приєднуваної бібліотеки Dynamically Linked Library (DLL) або виконуваного файлу.

Для кожного компілятора середовищем часу виконання проводиться необхідне відображення використовуваних типів у типи CTS, а програмного коду - у код «абстрактної машини» .NET Microsoft Intermediate Language (MSIL). У результаті програмний проект формується у вигляді збірки - самодостатнього компонента для розгортання, тиражування та повторного використання. Збірка ідентифікується цифровим підписом автора та унікальним номером версії.

2.3 Бібліотека NUnit .NET

NUnit [19] — це платформа модульного тестування для всіх мов .Net. Спочатку портована з JUnit поточна робоча версія була повністю переписана з багатьма новими функціями та підтримкою широкого спектру платформ .NET. Приклад використання xUnit:

```
using System.Collections.Generic;
using System.Linq;
using FluentAssertions;
using Functional.Object.Extensions;
using Microsoft.Extensions.DependencyInjection;
using Moq;
using mrlldd.Caching.Decoration.Internal.Logging.Performance;
using mrlldd.Caching.Extensions.DependencyInjection;
using mrlldd.Caching.Flags;
using mrlldd.Caching.Stores.Decoration;
using mrlldd.Caching.Stores.Internal;
using mrlldd.Caching.Tests.Stores.Base;
using NUnit.Framework;

namespace mrlldd.Caching.Tests.Stores
{
```

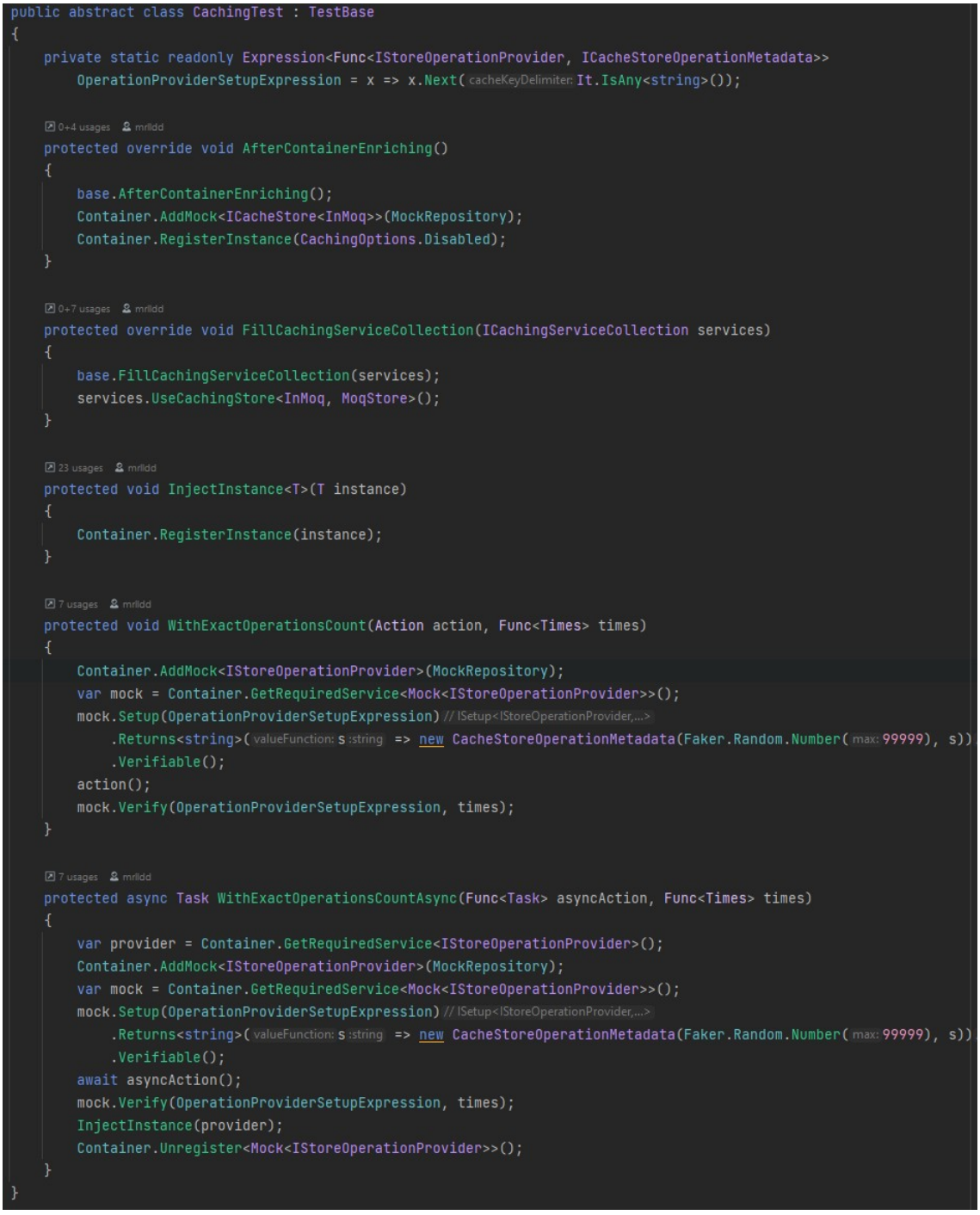
```
[TestFixture]
public class PerformanceLoggingVoidCacheStoreTests : LoggingStoreDecoratorTests
{
   protected override Times Times => Times.Once();

   protected override void FillCachingServiceCollection(ICachingServiceCollection services)
   {
      base.FillCachingServiceCollection(services);
      services.WithPerformanceLogging<InVoid>(DefaultLogLevel);
   }
}
```

```
[TestFixture]
public class PerformanceLoggingDecorationTests : TestBase
{
    protected override void FillCachingServiceCollection(ICachingServiceCollection services)
   {
      base.FillCachingServiceCollection(services);
      services.WithPerformanceLogging<InVoid>();
   }

   [Test]
   public void ProvidesAlreadyDecoratedStore()
   {
      Container
         .Effect(c => c.GetRequiredService<ICacheStoreProvider<InVoid>>().CacheStore
            .Should()
            .NotBeNull()
            .And.BeOfType<PerformanceLoggingCacheStore<InVoid>>());
   }

   [Test]
   public void HasRegisteredDecorator()
   {
      Container
         .Effect(c =>
         {
            var decorators =
c.GetRequiredService<IEnumerable<ICacheStoreDecorator<InVoid>>>().ToArray();
            decorators
               .Should()
               .NotBeNull();
            decorators.Length
               .Should()
               .Be(1);
            decorators[0]
               .Should()
               .NotBeNull()
               .And.BeOfType<PerformanceLoggingCacheStoreDecorator<InVoid>>();});}}}
```

2.4 Бібліотека Moq .NET

Крім фреймворків для створення та проведення Unit тестів при тестуванні часто бувають корисні такі фреймворки, які дозволяють імітувати або емулювати якусь функціональність, створювати мок-об'єкти. Подібних фреймворків існує безліч, і одним із найпопулярніших є Moq .NET [20].

Moq призначена для імітації об'єктів. Наразі імітується функціональність репозиторію. Для цього об'єкт Mock типується відповідним типом: var mock = new Mock<IRepository>(). Потім виконується налаштування mock-об'єкту за допомогою методу Setup(), а з допомогою методу Returns() визначається результат, що повертається (рис. 2.1).



```
public abstract class CachingTest : TestBase
{
    private static readonly Expression<Func<IStoreOperationProvider, ICacheStoreOperationMetadata>>
        OperationProviderSetupExpression = x => x.Next( cacheKeyDelimiter: It.IsAny<string>());

    0+4 usages  mrlldd
    protected override void AfterContainerEnriching()
    {
        base.AfterContainerEnriching();
        Container.AddMock<ICacheStore<InMoq>>(MockRepository);
        Container.RegisterInstance(CachingOptions.Disabled);
    }

    0+7 usages  mrlldd
    protected override void FillCachingServiceCollection(ICachingServiceCollection services)
    {
        base.FillCachingServiceCollection(services);
        services.UseCachingStore<InMoq, MoqStore>();
    }

    23 usages  mrlldd
    protected void InjectInstance<T>(T instance)
    {
        Container.RegisterInstance(instance);
    }

    7 usages  mrlldd
    protected void WithExactOperationsCount(Action action, Func<Times> times)
    {
        Container.AddMock<IStoreOperationProvider>(MockRepository);
        var mock = Container.GetRequiredService<Mock<IStoreOperationProvider>>();
        mock.Setup(OperationProviderSetupExpression) // ISetup<IStoreOperationProvider,...>
            .Returns<string>( valueFunction: s :string  => new CacheStoreOperationMetadata(Faker.Random.Number( max: 99999), s))
            .Verifiable();
        action();
        mock.Verify(OperationProviderSetupExpression, times);
    }

    7 usages  mrlldd
    protected async Task WithExactOperationsCountAsync(Func<Task> asyncAction, Func<Times> times)
    {
        var provider = Container.GetRequiredService<IStoreOperationProvider>();
        Container.AddMock<IStoreOperationProvider>(MockRepository);
        var mock = Container.GetRequiredService<Mock<IStoreOperationProvider>>();
        mock.Setup(OperationProviderSetupExpression) // ISetup<IStoreOperationProvider,...>
            .Returns<string>( valueFunction: s :string  => new CacheStoreOperationMetadata(Faker.Random.Number( max: 99999), s))
            .Verifiable();
        await asyncAction();
        mock.Verify(OperationProviderSetupExpression, times);
        InjectInstance(provider);
        Container.Unregister<Mock<IStoreOperationProvider>>();
    }
}
```

Рисунок 2.1 − Використання Moq у базовому класі для Unit тестів

2.5 Бібліотека BenchmarkDotNet

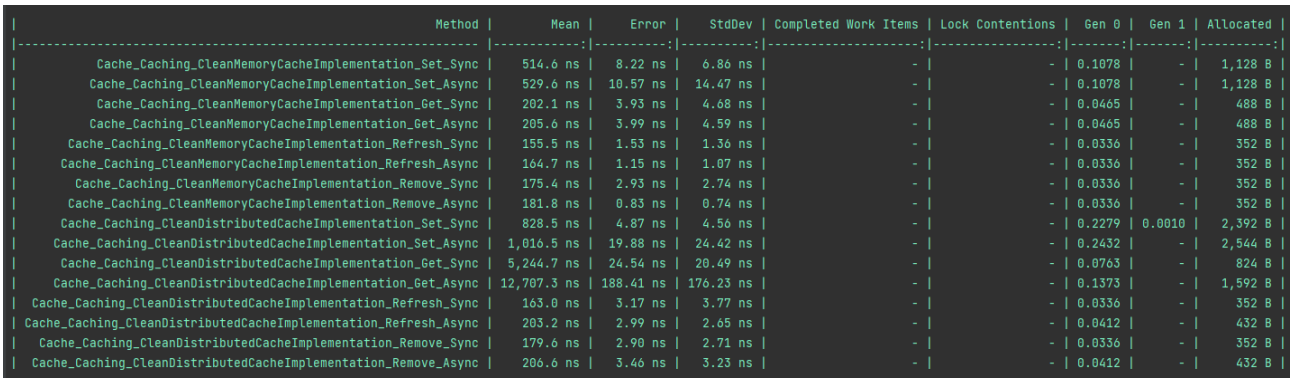
BenchmarkDotNet [21] допомагає трансформувати методи в еталони, відстежувати їхню ефективність та ділитися відтворюваними експериментами вимірювання. Надійні та точні результати гарантує статистична система

perfolizer. BenchmarkDotNet захищає від популярних помилок порівняльного аналізу та попереджає, якщо щось не так з дизайном тесту або отриманими вимірюваннями. Результати представлені у зручній формі, яка підкреслює всі важливі факти про експеримент. Бібліотека використовується понад 6800 проектами, включаючи .NET Runtime, підтримується .NET Foundation. Приклад використання BenchmarkDotNet:

```
using System.Threading.Tasks;
using BenchmarkDotNet.Attributes;
using Functional.Result;
using Microsoft.Extensions.DependencyInjection;
using mrlldd.Caching.Caches;
using mrlldd.Caching.Extensions.DependencyInjection;
using mrlldd.Caching.Flags;

namespace mrlldd.Caching.Benchmarks.Cache
{
   public class CleanCacheBenchmarks : Benchmark
   {
      private readonly ICache<byte, InDistributed> cleanDistributedCacheImplementation;
      private readonly ICache<int, InMemory> cleanMemoryCacheImplementation;

      public CleanCacheBenchmarks()
      {
         var cleanSp = new ServiceCollection()
            .AddDistributedMemoryCache()
            .AddCaching(typeof(CleanCacheBenchmarks).Assembly)
            .BuildServiceProvider()
            .CreateScope().ServiceProvider;
         cleanMemoryCacheImplementation = cleanSp.GetRequiredService<ICache<int,
InMemory>>();
         cleanDistributedCacheImplementation = cleanSp.GetRequiredService<ICache<byte,
InDistributed>>();
      }

      [Benchmark]
      public void Cache_Caching_CleanMemoryCacheImplementation_Set_Sync() =>
cleanMemoryCacheImplementation.Set(3);

      [Benchmark]
      public ValueTask<Result> Cache_Caching_CleanMemoryCacheImplementation_Set_Async()
=>
         cleanMemoryCacheImplementation.SetAsync(3);

      [Benchmark]
```

```
    public void Cache_Caching_CleanMemoryCacheImplementation_Get_Sync() =>
cleanMemoryCacheImplementation.Get();

    [Benchmark]
    public ValueTask<Result<int>>
Cache_Caching_CleanMemoryCacheImplementation_Get_Async() =>
        cleanMemoryCacheImplementation.GetAsync();

    [Benchmark]
    public void Cache_Caching_CleanMemoryCacheImplementation_Refresh_Sync() =>
        cleanMemoryCacheImplementation.Refresh();

    [Benchmark]
    public ValueTask<Result>
Cache_Caching_CleanMemoryCacheImplementation_Refresh_Async() =>
        cleanMemoryCacheImplementation.RefreshAsync();

    [Benchmark]
    public void Cache_Caching_CleanMemoryCacheImplementation_Remove_Sync() =>
        cleanMemoryCacheImplementation.Remove();

    [Benchmark]
    public ValueTask<Result>
Cache_Caching_CleanMemoryCacheImplementation_Remove_Async() =>
        cleanMemoryCacheImplementation.RemoveAsync();

    [Benchmark]
    public void Cache_Caching_CleanDistributedCacheImplementation_Set_Sync() =>
        cleanDistributedCacheImplementation.Set(3);

    [Benchmark]
    public ValueTask<Result>
Cache_Caching_CleanDistributedCacheImplementation_Set_Async() =>
        cleanDistributedCacheImplementation.SetAsync(3);

    [Benchmark]
    public void Cache_Caching_CleanDistributedCacheImplementation_Get_Sync() =>
        cleanDistributedCacheImplementation.Get();

    [Benchmark]
    public ValueTask<Result<byte>>
Cache_Caching_CleanDistributedCacheImplementation_Get_Async() =>
        cleanDistributedCacheImplementation.GetAsync();

    [Benchmark]
    public void Cache_Caching_CleanDistributedCacheImplementation_Refresh_Sync() =>
        cleanDistributedCacheImplementation.Refresh();

    [Benchmark]
    public ValueTask<Result>
Cache_Caching_CleanDistributedCacheImplementation_Refresh_Async() =>
```

```
        cleanDistributedCacheImplementation.RefreshAsync();

    [Benchmark]
    public void Cache_Caching_CleanDistributedCacheImplementation_Remove_Sync() =>
        cleanDistributedCacheImplementation.Remove();

    [Benchmark]
    public ValueTask<Result>
Cache_Caching_CleanDistributedCacheImplementation_Remove_Async() =>
        cleanDistributedCacheImplementation.RemoveAsync();}}
```



Method	Mean	Error	StdDev	Completed Work Items	Lock Contentions	Gen 0	Gen 1	Allocated
Cache_Caching_CleanMemoryCacheImplementation_Set_Sync	514.6 ns	8.22 ns	6.86 ns	-	-	0.1078	-	1,128 B
Cache_Caching_CleanMemoryCacheImplementation_Set_Async	529.6 ns	10.57 ns	14.47 ns	-	-	0.1078	-	1,128 B
Cache_Caching_CleanMemoryCacheImplementation_Get_Sync	202.1 ns	3.93 ns	4.68 ns	-	-	0.0465	-	488 B
Cache_Caching_CleanMemoryCacheImplementation_Get_Async	205.6 ns	3.99 ns	4.59 ns	-	-	0.0465	-	488 B
Cache_Caching_CleanMemoryCacheImplementation_Refresh_Sync	155.5 ns	1.53 ns	1.36 ns	-	-	0.0336	-	352 B
Cache_Caching_CleanMemoryCacheImplementation_Refresh_Async	164.7 ns	1.15 ns	1.07 ns	-	-	0.0336	-	352 B
Cache_Caching_CleanMemoryCacheImplementation_Remove_Sync	175.4 ns	2.93 ns	2.74 ns	-	-	0.0336	-	352 B
Cache_Caching_CleanMemoryCacheImplementation_Remove_Async	181.8 ns	0.83 ns	0.74 ns	-	-	0.0336	-	352 B
Cache_Caching_CleanDistributedCacheImplementation_Set_Sync	828.5 ns	4.87 ns	4.56 ns	-	-	0.2279	0.0010	2,392 B
Cache_Caching_CleanDistributedCacheImplementation_Set_Async	1,016.5 ns	19.88 ns	24.42 ns	-	-	0.2432	-	2,544 B
Cache_Caching_CleanDistributedCacheImplementation_Get_Sync	5,244.7 ns	24.54 ns	20.49 ns	-	-	0.0763	-	824 B
Cache_Caching_CleanDistributedCacheImplementation_Get_Async	12,707.3 ns	188.41 ns	176.23 ns	-	-	0.1373	-	1,592 B
Cache_Caching_CleanDistributedCacheImplementation_Refresh_Sync	163.0 ns	3.17 ns	3.77 ns	-	-	0.0336	-	352 B
Cache_Caching_CleanDistributedCacheImplementation_Refresh_Async	203.2 ns	2.99 ns	2.65 ns	-	-	0.0412	-	432 B
Cache_Caching_CleanDistributedCacheImplementation_Remove_Sync	179.6 ns	2.90 ns	2.71 ns	-	-	0.0336	-	352 B
Cache_Caching_CleanDistributedCacheImplementation_Remove_Async	206.6 ns	3.46 ns	3.23 ns	-	-	0.0412	-	432 B

Рисунок 2.2 – Результат використання BenchmarkDotNet

2.6 NuGet

Важливим інструментом для будь-якої сучасної платформи розробки є механізм, за допомогою якого розробники можуть створювати, ділитися та використовувати корисний код. Часто такий код об’єднується в «пакети», які містять скомпільований код DLL, необхідним у проектах.

Для .NET Core підтримується механізм обміну кодом NuGet [22], який визначає, як створюються, розміщуються та споживаються пакети для .NET.

Пакет NuGet — це один zip-файл із розширенням .nupkg, який містить скомпільований код DLL, інші файли, пов’язані з цим кодом, та описовий маніфест. Розробники з кодом для спільного використання створюють пакети та публікують їх на загальнодоступному або приватному host. Споживачі пакетів отримують ці zip з відповідних hosts, додають їх до своїх проектів, а потім викликають функціональні можливості пакету в коді свого проекту.

2.7 JetBrains Rider

JetBrains Rider [23] − кросплатформне інтегроване середовище розробки програмного забезпечення для платформи .NET, що розробляється компанією JetBrains. Підтримуються мови програмування C #, VB.NET та F#.

У основі проекту лежить інший продукт JetBrains − ReSharper. Середовище підтримує платформи .NET Framework, .NET Core та Mono.

JetBrains Rider надає більше 2200 інспекцій, які допомагають знаходити помилки та проблеми у структурі коду. Для усунення виявлених проблем доступно більше 1000 автоматичних виправлень, які можна застосовувати спільно або окремо. Для глобального відстеження проблем у проектах JetBrains Rider передбачено інструмент аналізу помилок по рішенню.

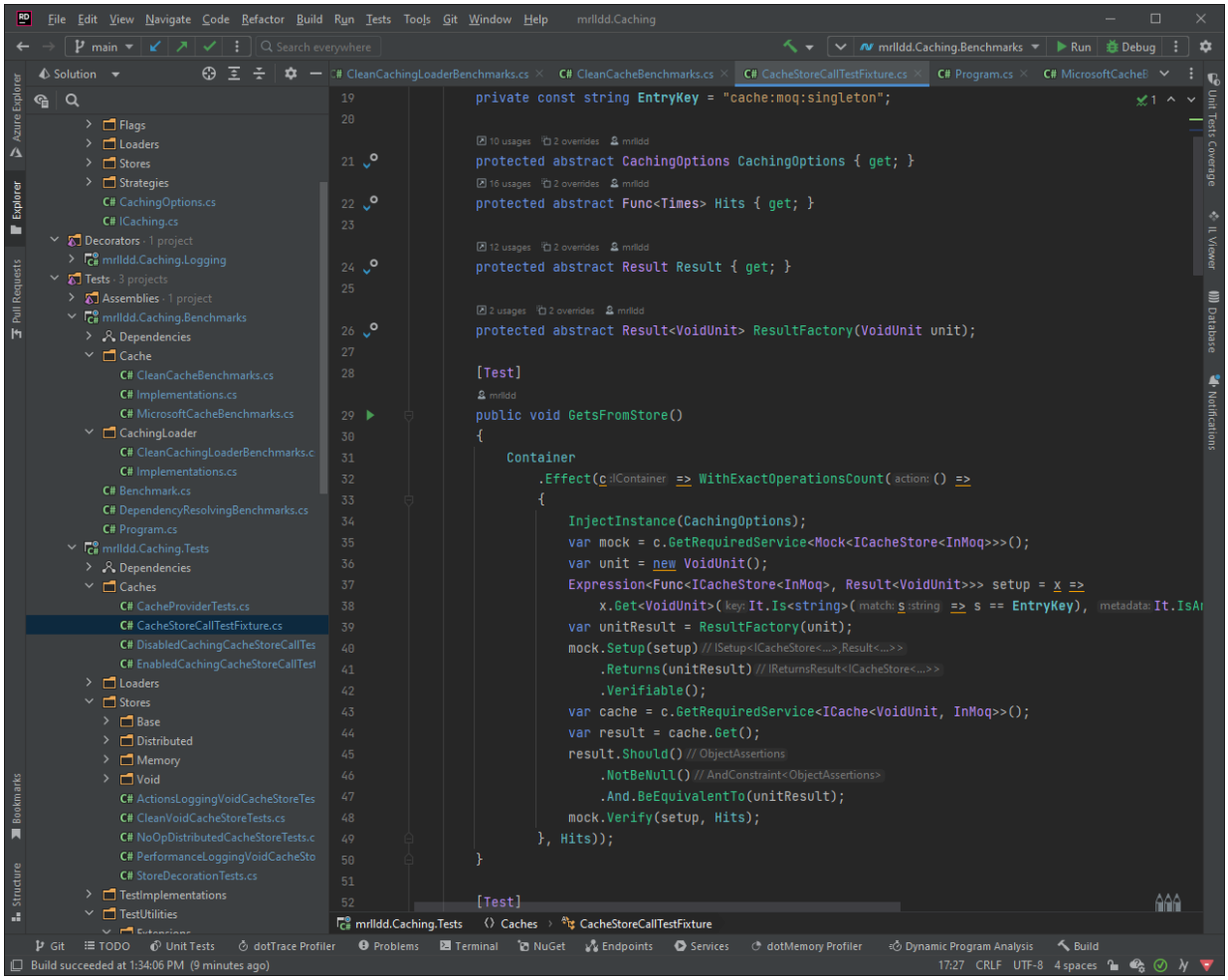


Рисунок 2.3 – Проект у JetBrains Rider

2.8 Redis

Redis [9] – це сховище даних у пам’яті з відкритим вихідним кодом, яке використовується як база даних, кеш-пам’ять, механізм потокової передачі та посередник повідомлень.

Зовнішні програми спілкуються з Redis за допомогою TCP-сокета та спеціального протоколу Redis. Цей протокол реалізовано в клієнтських бібліотеках Redis для різних мов програмування. Redis надає утиліту командного рядка, яку можна використовувати для надсилання команд до Redis.

У середі .NET для роботи з Redis є бібліотека StackExchange.Redis, що надає коннектор до Redis-серверу. Приклад використання StackExchange.Redis:

```
using StackExchange.Redis;
using System;
using System.Threading.Tasks;

namespace ReferenceConsoleRedisApp
{
   class Program
   {
      static readonly ConnectionMultiplexer redis = ConnectionMultiplexer.Connect(
         new ConfigurationOptions{
            EndPoints = {"localhost:6379"}});
      static async Task Main(string[] args) {
         var db = redis.GetDatabase();
         var pong = await db.PingAsync();
         Console.WriteLine(pong);}}}
```

2.9 Memcached

Memcached [8] − безкоштовна високопродуктивна система кешування об’єктів розподіленої пам’яті з відкритим кодом, призначена для прискорення динамічних веб-додатків шляхом зменшення навантаження на базу даних.

Memcached — це сховище ключів та значень у пам’яті для невеликих фрагментів довільних даних (рядків, об’єктів) із результатів викликів бази даних, викликів API або візуалізації сторінки.

Для роботи з Memcached у .NET є бібліотека EnyimMemcached, що надає коннектор до Memcached-серверу. Приклад використання EnyimMemcached:

```
using Enyim.Caching;
using Enyim.Caching.Configuration;
using Enyim.Caching.Memcached;
MemcachedClientConfiguration config = new MemcachedClientConfiguration();
config.Servers.Add(new IPEndPoint("hostname", port));
config.Protocol = MemcachedProtocol.Binary;
config.Authentication.Type = typeof(PlainTextAuthenticator);
config.Authentication.Parameters["userName"] = "username";
config.Authentication.Parameters["password"] = "password";
config.Authentication.Parameters["zone"] = "";…
```

2.10 Висновки до другого розділу

У другому розділі наведено огляд та переваги використання у розробці бібліотеки Caching наступного стеку технологій: ASP .NET Core, C#, JetBrains Rider, Redis, MemCached, NuGet, NUnit, Moq, FluentAssertions, Benchmark .NET.

РОЗДІЛ 3
РОЗРОБКА БІБЛІОТЕКИ CACHING У ASP .NET CORE (C#)

1.1 Архітектура та можливості бібліотеки Caching

Оскільки інтерфейс розробника має бути якомога більше простий (код для інтеграції з бібліотекою, код для користування кешами тощо), а також код, що виконується, має давати якомога більше контролю, у бібліотеці Caching використовується механізм рефлексії, наданий середою .NET.

Класи у просторі імен System.Reflection разом із System.Type дозволяють отримувати інформацію про завантажені збірки та типи, визначені в них, такі як класи, інтерфейси та типи значень (структури та перерахування). Reflection використано для створення екземплярів типу під час виконання, а також для виклику та доступу до них.

Збірки містять модулі, модулі містять типи, а типи містять члени. Reflection надає об’єкти, які інкапсулюють збірки (класс Assembly), модулі та типи. Reflection використано для динамічного створення екземпляра типу, зв’язування типу з існуючим об’єктом або отримання типу з існуючого об’єкта. Потім викликано методи типу або отримати доступ до його полів та властивостей.

Reflection надає класи, такі як Type та MethodInfo, для представлення типів, членів, параметрів та інших сутностей коду. При використанні Reflection, не виконується робота безпосередньо з цими класами, більшість з яких є абстрактними.

Оскільки бібліотека Caching використовується у середовищі ASP .NET Core, що надає механізм Dependency Injection (реалізація одного з принципів SOLID, а саме Dependency Inversion Principle), використано можливість додати сервіси, що надає бібліотека Caching за допомогою методу розширення

AddCaching(this IServiceCollection services, params Assembly[] assemblies) під час реєстрації сервісів.

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddCaching(typeof(Startup).Assembly);
}
```

Рис 3.1 – Використання методу AddCaching

Метод аналізує надані збірки та збирає усі типи, що успадковуються від класів Cache<TEntity, TFlag>, CachingLoader<TArg, TEntity, TFlag>, ILoader<TArg, TEntity>. Кожен з цих типів має у собі шаблонний параметр TEntity, що зберігається до кешу або завантажуватися з кешу чи будь-якого джерела.

Cache<TEntity, TFlag> та CachingLoader<TArg, TEntity, TFlag> мають у собі об’єкт-сервіс, що успадковується від ICacheStore<TFlag> − обгортки для клієнта-кеша.

1.2 CachingFlag

Шаблонний параметр TFlag у Cache<TEntity, TFlag> та CachingLoader<TArg, TEntity, TFlag> є параметром-маркером. На його основі бібліотека Caching будує сервіси та «розуміє», де та з якими параметрами сутність буде збережена. Цей параметр успадковується від базового класу CachingFlag.

Бібліотека Caching надає вбудовані Store та флаги:

- InMemory − Store з цим флагом використовує сервіс IMemoryCache з простору Microsoft.Extensions.Caching.Memory.
- InDistributed − Store з цим флагом використовує сервіс IDistributedCache з простору Microsoft.Extensions.Caching.Distributed (якщо такого сервісу

немає, бібліотека реєструє NoOpDistributedCache, що реалізує інтерфейс IDistributedCache та є «об’єктом-заглушкою»)

- InVoid − Store з цим флагом є «об’єктом-заглушкою», що не робить ніяких реальних дій та призначений лише для використання у тестових умовах.

Також розробник може розширити кількість доступних Store, зробивши свій флаг для Store та реалізацію Store. Наприклад, якщо розробник хоче зберігати об’єкти сутності у файлах:

```
public sealed class InFile : CachingFlag
{
        private InFile()
        { }
}
```

Рисунок 3.2 – Флаг для Store, що зберігає об’єкти сутності у файлах

```
public class FileStore : ICacheStore<InFile>
{
        /*...*/
}
```

Рисунок 3.3 – Приклад Store з флагом InFile, що зберігає об’єкти сутності у файлах

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddCaching(typeof(Startup).Assembly)
                .UseCachingStore<InFile, FileStore>();
}
```

Рисунок 3.4 – Реєстрація Store, що зберігає об'єкти сутностей до файлів

1.3 Категоризація ключів записів у кеші

Для рішення проблеми з дублюванням та структуруванням ключів для бібліотеки Caching створено модель категоризованих ключів записів у кеші.

Структурований ключ складається з трьох частин (рис. 3.5):

- префікс − частина, що є «автором» запису у кеші; це може бути або кеш (значення «cache») або кешуючий Loader (значення «loader»);
- ключ сутності − частина, що має (але не обов’язково) ідентифікувати сутність та буде збережена/завантажена з кешу; значення залежить від реалізації;
- суфікс − частина, що ідентифікує собою ключ об’єкту сутності; значення залежить (але не обов’язково) від зовнішніх даних (наприклад, id користувача, id сесії користувача тощо); якщо суфікс статичний, то об’єкт сутності у кеші може бути Singleton при умові, що ключ сутності також статичний.

За замовчуванням, «делімітер» частин у ключі − двокрапка (:), але це також може бути модифіковано розробником.

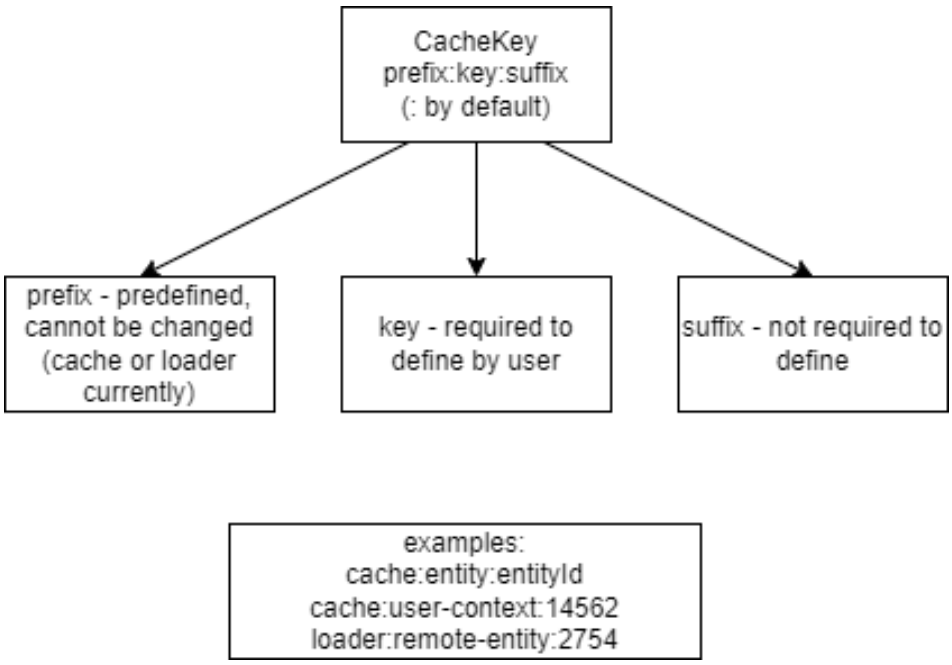


Рисунок 3.5 – Структура ключу запису у кеші

Категоризацію ключів записів у кеші зображено на рис. 3.6.

Розглянемо приклади, що демонструють використання моделі категоризованих ключів записів у кеші (рис. 3.7, 3.8, 3.9).

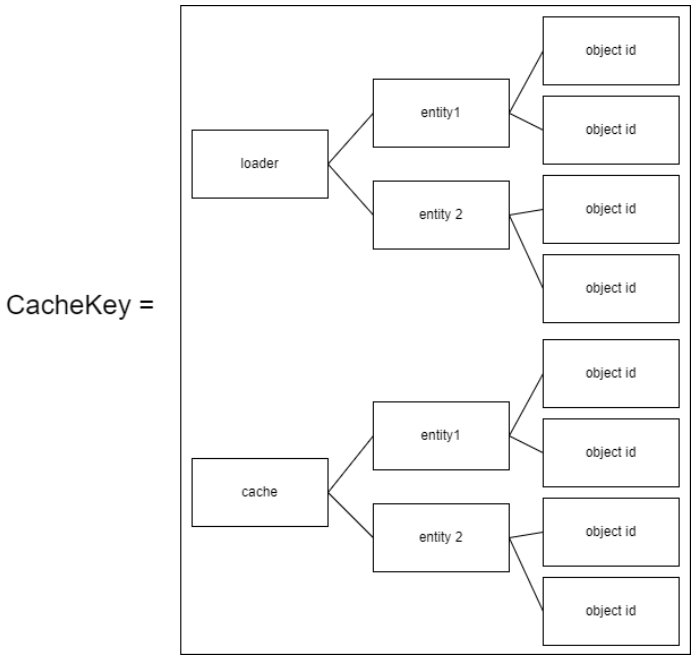


Рисунок 3.6 – Категоризація ключів записів у кеші

```
public class ContextfulCache : Cache<UserContext, InDistributed>
{
        private readonly IUserContextProvider provider;

        protected override CachingOptions Options
                => CachingOptions.Enabled(TimeSpan.FromMinutes(20));

        protected override string CacheKey => "user-context";

        protected override string CacheKeySuffix => provider.SessionId;

        public ContextfulCache(IUserContextProvider provider)
                => this.provider = provider;
}
```

Рисунок 3.7 – Приклад кешу с динамічним суфіксом ключів записів

```
public class IntCache : Cache<int, InMemory>
{
        protected override CachingOptions Options => CachingOptions.Enabled(TimeSpan.FromSeconds(15));
        protected override string CacheKey => "singleton-number";
}
```

Рисунок 3.8 – Приклад кешу з Singleton

```
public class RemoteEntityCachingLoader : CachingLoader<int, RemoteEntity, InMemory>
{
        protected override CachingOptions Options => CachingOptions.Enabled(TimeSpan.FromMinutes(5));
        protected override string CacheKey => nameof(RemoteEntity);

        protected override string CacheKeySuffixFactory(int args)
                => args.ToString();
}
```

Рисунок 3.9 – Приклад «кешуючого» Loader з статичним ключем та динамічним суфіксом ключа

1.4 Caching<T, TStoreFlag>

Caching<T, TStoreFlag> − базовий клас, від якого успадковані Cache<T, TStoreFlag> та CachingLoader<TArg, TEntity, TStoreFlag>. Цей клас надає методи для запитів (Set, Get, Refresh, Delete) до Store з відповідним флагом, в котрих також передає метадані для кожної операції зі Store (наприклад, id операції).

1.5 StoreOperationProvider

StoreOperationProvider − сервіс-фабрика, що надає метадані для кожної операції зі Store.

1.6 Cache<T, TStoreFlag>

Cache<T, TStoreFlag> − базовий клас для сервісів-кешів, що утилізує методи базового класу Caching.

На рис. 3.10, 3.11 наведено приклади реалізації та використання кешу.

```
public class IntCache : Cache<int, InMemory>
{
        protected override CachingOptions Options => CachingOptions.Enabled(TimeSpan.FromSeconds(15));
        protected override string CacheKey => "singleton-number";
}
```

Рисунок 3.10 – Приклад кешу

```
public class IntCacheUsingService
{
        private readonly ICache<int, InMemory> intCache;

        public IntCacheUsingService(ICache<int, InMemory> intCache)
                => this.intCache = intCache;


        public int CalculateWithMemoization(int a, int b)
        {
                var fromCache = intCache.Get();
                if (fromCache.Successful)
                {
                        return fromCache;
                }

                var result = PerformExpensiveCalculation(a, b);
                intCache.Set(result);
                return result;
        }
}
```

Рисунок 3.11 – Приклад використання кешу в іншому сервісі

1.7 ILoader<TArgs, TResult>

ILoader<TArgs, TResult> − інтерфейс для реалізації Loader сервісів, що використовуються у CachingLoader<TArgs, TEntity, TStoreFlag>.

Оскільки реалізації цього інтерфейсу вже додані до сервіс-провайдеру, ці Loaders також можуть використовуватись розробником (рис. 3.12).

```
public class RemoteEntityLoader : ILoader<int, RemoteEntity>
{
        public async Task<RemoteEntity> LoadAsync(int args, CancellationToken token = default)
        {
                await Task.Delay(5000, token);
                return new RemoteEntity(args);
        }
}
```

Рисунок 3.12 – Приклад Loader

1.8 CachingLoader<TArgs, TEntity, TStoreFlag>

CachingLoader<TArgs, TEntity, TStoreFlag> − базовий клас для «кешуючих» Loader, що утилізує методи базового класу Caching та надає методи GetOrLoad() та GetOrLoadAsync() з опцією «оминути» кеш при завантаженні даних (оновлення запису у кеші).

Методи завантажують дані з будь-якого джерела, якщо відповідного запису немає у кеші або якщо запит до кешу «оминається» при завантаженні. Після завантаження методи зберігають результат до кешу та повертають результат завантаження.

Приклад використання базового класу зображено на рис. 3.13, 3.14.

Діаграму послідовності використання CachingLoader зображено на рис. 3.15.

```
public class RemoteEntityCachingLoader : CachingLoader<int, RemoteEntity, InMemory>
{
        protected override CachingOptions Options => CachingOptions.Enabled(TimeSpan.FromMinutes(5));
        protected override string CacheKey => nameof(RemoteEntity);

        protected override string CacheKeySuffixFactory(int args)
                => args.ToString();
}
```

Рисунок 3.13 – Приклад «кешуючого» Loader

```
public class RemoteEntityStorage
{
    private readonly ICachingLoader<int, RemoteEntity, InMemory> cachingLoader;

    1 usage  new *
    public RemoteEntityStorage(ICachingLoader<int, RemoteEntity, InMemory> cachingLoader)
    {
        this.cachingLoader = cachingLoader;
    }

    1 usage  new *
    public async Task<RemoteEntity> LoadByIdAsync(int id, CancellationToken cancellationToken = default)
    {
        var loadingResult = await cachingLoader.GetOrLoadAsync(id, token: cancellationToken);
        if (!loadingResult.Successful)
        {
            throw loadingResult;
        }

        return loadingResult;
    }

}
```

Рис 3.14 – Приклад сервісу, що використовує «кешуючий» Loader

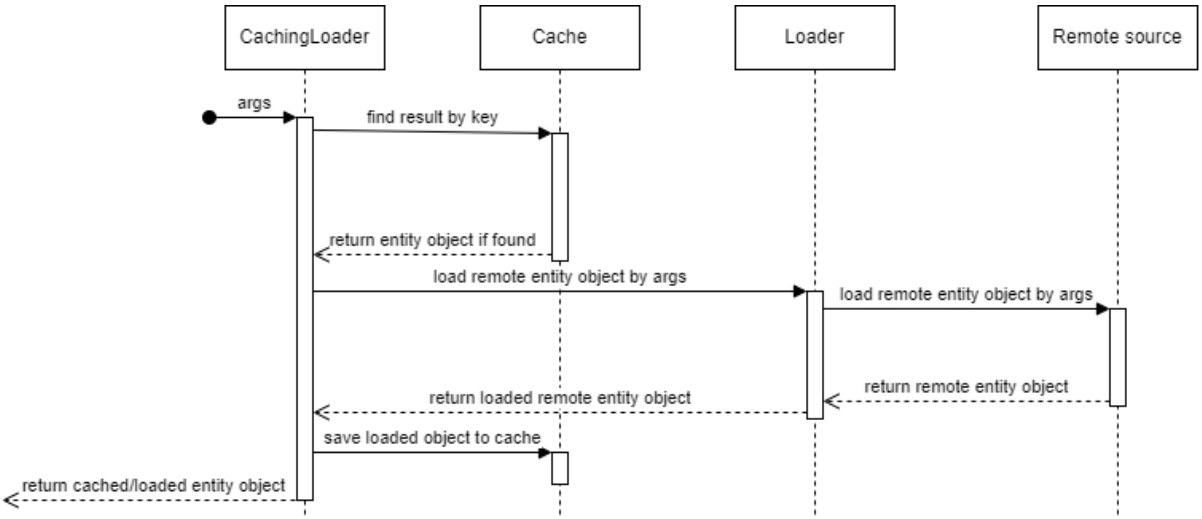


Рисунок 3.15 – Діаграма послідовності використання CachingLoader

1.9 CacheStoreDecorator

У бібліотеці Caching передбачена «декорація» будь-яких Store. Розробнику потрібно створити клас-сервіс, що реалізує

ICacheStoreDecorator<TStoreFlag>, де потрібно буде реалізувати метод декорації Store.

Приклад використання «декорації» зображено на рис. 3.16, 3.17, 3.18.

```
public class WrappingDecorator<TStoreFlag> : ICacheStoreDecorator<TStoreFlag> where TStoreFlag : CachingFlag
{
        public ICacheStore<TStoreFlag> Decorate(ICacheStore<TStoreFlag> cacheStore)
                => new WrappedCacheStore<TStoreFlag>(cacheStore);

        public int Order => 2754;
}
```

Рисунок 3.16 – Приклад реалізації сервіса-декоратора Store

```
public class WrappedCacheStore<T>: ICacheStore<T> where T : CachingFlag
{
        /*...*/
}
```

Рисунок 3.17 – Приклад реалізації Store-обгортки

```
public void ConfigureServices(IServiceCollection services)
{
        services.AddCaching(typeof(Startup).Assembly)
                .Decorators<InMemory>().Add<WrappingDecorator<InMemory>>()
                .Decorators<InDistributed>().Add<WrappingDecorator<InDistributed>>();
}
```

Рисунок 3.18 – Додавання «декорації» до Store з відповідними флагами

Також бібліотека Caching надає «логуючі декоратори» для будь-яких Store (рис. 3.19).

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCaching(typeof(Startup).Assembly)
        .WithActionsLogging<InVoid>()
        .WithPerformanceLogging<InVoid>();
}
```

Рисунок 3.19 – Приклад додавання «логуючої декорації» до Store з флагом InVoid

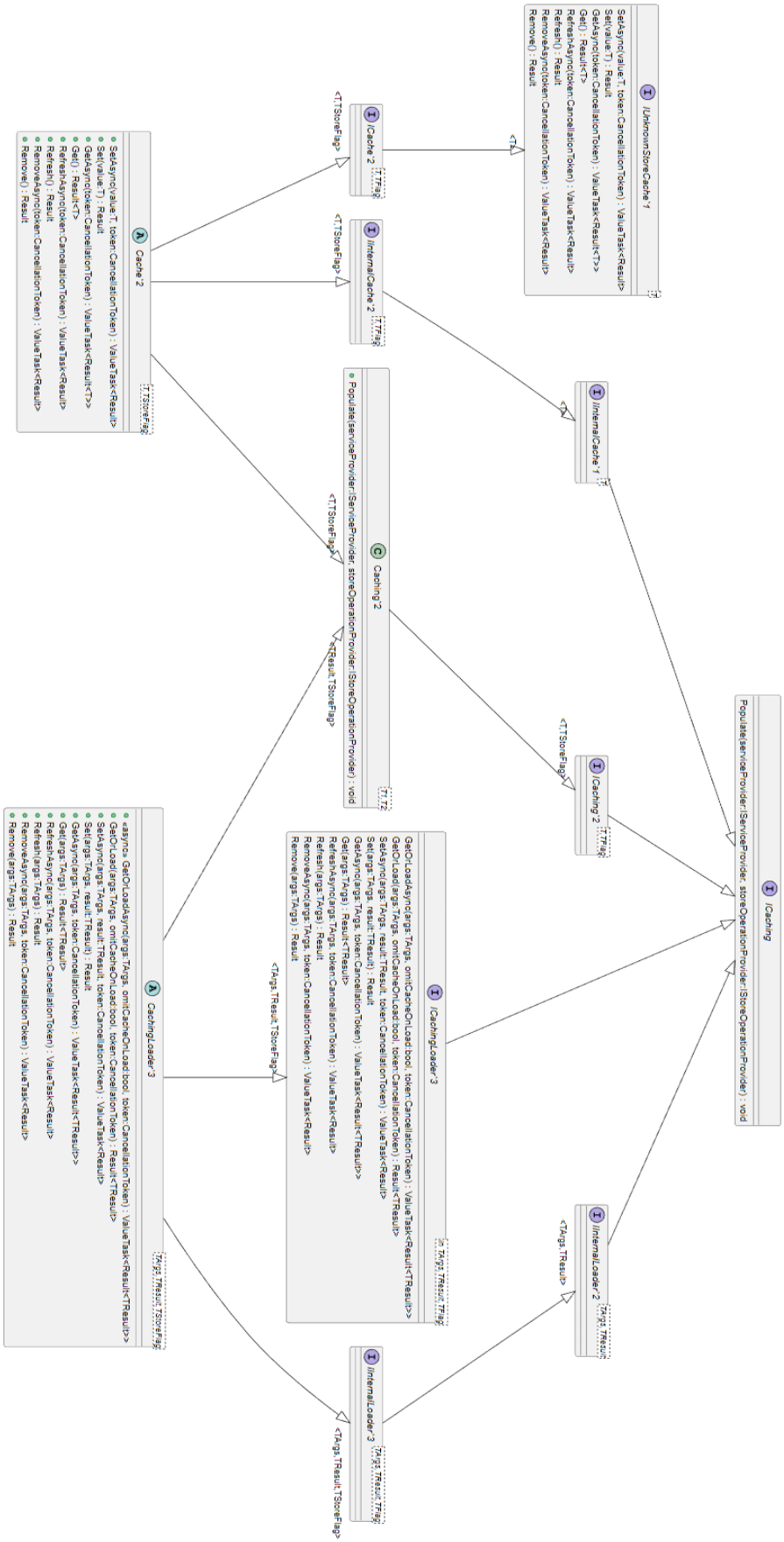


Рисунок 3.20 – Діаграма класів Caching, Cache та CachingLoader

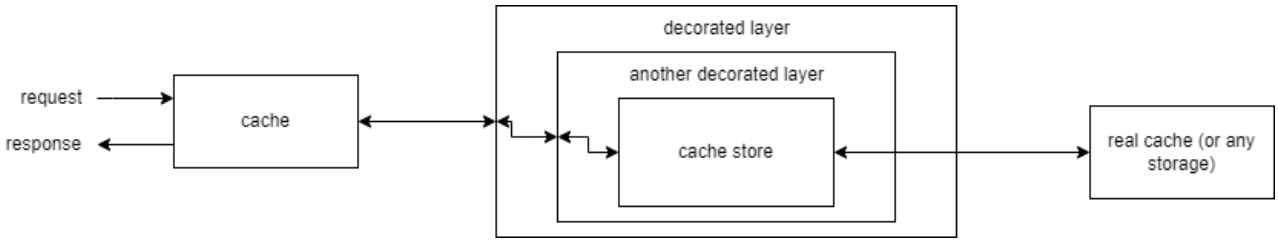


Рисунок 3.21 – Виконання запиту до кешу

1.10 «Безфлажні» кеші

Кеші без флагу Store розцінюються як будь-який кеш для сутності взагалі або як колекція усіх можливих кешів для цього типу сутності (рис. 3.22).

```
public class RemoteEntityStorage
{
    private readonly ICache<RemoteEntity> cache;

    1 usage  new *
    public RemoteEntityStorage(ICache<RemoteEntity> cache)
    {
        this.cache = cache;
    }

    1 usage  new *
    public Task SetAsync(RemoteEntity remoteEntity, CancellationToken cancellationToken = default)
        => cache.SetAsync(remoteEntity, SequenceStrategy.Instance, cancellationToken);
}
```

Рисунок 3.22 – Приклад використання «безфлажного» кешу

«Безфлажний» кеш є контейнером для колекції усіх кешів, що мають флаги та надає інтерфейс для розробника як для звичайного «флажного» кешу. Для зручної роботи з колекцією кешів у інтерфейсі ICache<TEntity> передбачено використання стратегій (ICachingRemoveStrategy, ICachingRefreshStrategy, ICachingSetStrategy, ICachingGetStrategy).

1.10 Тестування бібліотеки Caching

Method	Mean	Error	StdDev	Gen 0	Completed Work Items	Lock Contentions	Allocated
Cache_Microsoft_Memory_Set_Sync	300.28 ns	5.410 ns	4.796 ns	0.0501	-	-	528 B
Cache_Microsoft_Memory_Set_Async	301.06 ns	2.460 ns	2.301 ns	0.0501	-	-	528 B
Cache_Microsoft_Memory_Get_Sync	54.23 ns	0.349 ns	0.292 ns	0.0023	-	-	24 B
Cache_Microsoft_Memory_Get_Async	55.36 ns	0.385 ns	0.321 ns	0.0023	-	-	24 B
Cache_Microsoft_Memory_Remove_Sync	56.28 ns	0.241 ns	0.226 ns	-	-	-	-
Cache_Microsoft_Memory_Remove_Async	58.01 ns	0.388 ns	0.344 ns	-	-	-	-
Cache_Microsoft_DistributedMemory_Set_Sync	352.68 ns	2.016 ns	1.886 ns	0.0677	-	-	712 B
Cache_Microsoft_DistributedMemory_Set_Async	359.17 ns	2.174 ns	2.034 ns	0.0677	-	-	712 B
Cache_Microsoft_DistributedMemory_Get_Sync	52.52 ns	0.315 ns	0.246 ns	-	-	-	-
Cache_Microsoft_DistributedMemory_Get_Async	57.88 ns	0.229 ns	0.214 ns	-	-	-	-
Cache_Microsoft_DistributedMemory_Refresh_Sync	51.17 ns	0.177 ns	0.166 ns	-	-	-	-
Cache_Microsoft_DistributedMemory_Refresh_Async	54.05 ns	0.384 ns	0.321 ns	-	-	-	-
Cache_Microsoft_DistributedMemory_Remove_Sync	57.43 ns	0.271 ns	0.240 ns	-	-	-	-
Cache_Microsoft_DistributedMemory_Remove_Async	59.18 ns	0.605 ns	0.505 ns	-	-	-	-

Рисунок 3.23 – Результати Benchmark тестів «чистих» кешів

Method	Mean	Error	StdDev	Median	Gen 0	Completed Work Items	Lock Contentions	Gen 1	Allocated
Cache_Caching_CleanMemoryCacheImplementation_Set_Sync	536.5 ns	4.37 ns	3.87 ns	535.0 ns	0.1078	-	-	-	1,128 B
Cache_Caching_CleanMemoryCacheImplementation_Set_Async	549.7 ns	8.37 ns	6.99 ns	549.7 ns	0.1078	-	-	-	1,128 B
Cache_Caching_CleanMemoryCacheImplementation_Get_Sync	213.3 ns	3.80 ns	3.17 ns	211.8 ns	0.0465	-	-	-	488 B
Cache_Caching_CleanMemoryCacheImplementation_Get_Async	227.7 ns	1.80 ns	1.60 ns	227.2 ns	0.0465	-	-	-	488 B
Cache_Caching_CleanMemoryCacheImplementation_Refresh_Sync	162.4 ns	1.46 ns	1.14 ns	162.2 ns	0.0336	-	-	-	352 B
Cache_Caching_CleanMemoryCacheImplementation_Refresh_Async	174.9 ns	3.38 ns	3.00 ns	174.0 ns	0.0336	-	-	-	352 B
Cache_Caching_CleanMemoryCacheImplementation_Remove_Sync	179.1 ns	1.10 ns	1.03 ns	178.9 ns	0.0336	-	-	-	352 B
Cache_Caching_CleanMemoryCacheImplementation_Remove_Async	188.4 ns	1.04 ns	0.97 ns	188.1 ns	0.0336	-	-	-	352 B
Cache_Caching_CleanDistributedCacheImplementation_Set_Sync	889.7 ns	15.02 ns	12.54 ns	887.5 ns	0.2279	-	-	0.0010	2,392 B
Cache_Caching_CleanDistributedCacheImplementation_Set_Async	972.7 ns	8.43 ns	7.88 ns	973.7 ns	0.2432	-	-	0.0010	2,544 B
Cache_Caching_CleanDistributedCacheImplementation_Get_Sync	5,344.0 ns	33.93 ns	31.74 ns	5,341.3 ns	0.0763	-	-	-	824 B
Cache_Caching_CleanDistributedCacheImplementation_Get_Async	12,601.7 ns	178.58 ns	167.04 ns	12,560.6 ns	0.1373	-	-	-	1,592 B
Cache_Caching_CleanDistributedCacheImplementation_Refresh_Sync	177.3 ns	3.52 ns	5.16 ns	178.9 ns	0.0336	-	-	-	352 B
Cache_Caching_CleanDistributedCacheImplementation_Refresh_Async	221.1 ns	2.63 ns	2.33 ns	221.1 ns	0.0412	-	-	-	432 B
Cache_Caching_CleanDistributedCacheImplementation_Remove_Sync	186.0 ns	3.69 ns	6.37 ns	183.0 ns	0.0336	-	-	-	352 B
Cache_Caching_CleanDistributedCacheImplementation_Remove_Async	215.1 ns	0.78 ns	0.73 ns	215.1 ns	0.0412	-	-	-	432 B

Рисунок 3.24 – Результати Benchmark тестів кешів, що надає бібліотека Caching

1.11 Розгортання бібліотеки Caching у NuGet

Бібліотека Caching розгортається у NuGet за допомогою консольного інструменту NuGet, що надає середа .NET.

```
dotnet nuget push foo.nupkg
```

Рисунок 3.25 – Приклад пакету, що завантажується до NuGet

Бібліотека Caching отримує версію за допомогою Python-скрипту, що викликається при кожному push до гілки main у Git-репозиторії (за допомогою

Git-хуків). Це надає розробникам можливість використовувати конкретну версію бібліотеки Caching, що залежить від потреб розробки.

3.12 Висновки до третього розділу

У третьому розділі створено концепцію модульної інтеграції кешів та модель категоризованих ключів записів у кеші. Наведено опис архітектури та функціональних можливостей ASP .NET Core (C#) бібліотеки Caching. Наведено опис Unit та Benchmark тестування, а також розгортання бібліотеки Caching.

ВИСНОВКИ

Внаслідок виконання даної магістерської дипломної роботи отримано наступні результати:

1. У результаті поєднання патернів об'єктно-орієнтованого та функціонального проектування створено концепцію модульної інтеграції кешів.
2. Створено модель категоризованих ключів записів у кеші, що вирішує проблему дублювання та структурування ключів.
3. На основі концепції модульної інтеграції кешів розроблено бібліотеку Caching у ASP .NET Core (C#) та «інтерфейс» для роботи з нею.
4. Бібліотеку Caching повністю покрито Unit тестами та частково покрито Benchmark тестами, що доводить стабільність її використання у реальних web-додатках.

ПЕРЕЛІК ПОСИЛАНЬ

1. Microsoft Technical Documentation [Електронний ресурс] − Режим доступу: https://learn.microsoft.com/en-us/docs/ − Дата доступу: 02.09.22
2. Refactoring Guru [Електронний ресурс] − Режим доступу: https://refactoring.guru − Дата доступу: 03.09.22
3. Andrew S. Tanenbaum, David J. Wetherall. Computer Networks, 5th Edition, 2011, с. 571 – 582
4. Stackoverflow [Електронний ресурс] − Режим доступу: https://stackoverflow.com/ − Дата доступу: 01.09.22
5. Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship, Pearson Education, 2008, 464 pp.
6. Розподілений кеш [Електронний ресурс] − Режим доступу: https://uk.wikipedia.org/wiki/%D0%A0%D0%BE%D0%B7%D0%BF%D0%BE%D0%B4%D1%96%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9_%D0%BA%D0%B5%D1%88 − Дата доступу: 01.10.22
7. S. Kim, D. Chandra, Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture, in PACT, 2004.
8. Memcached [Електронний ресурс] − Режим доступу: https://memcached.org/ − Дата доступу: 11.10.22
9. Redis [Електронний ресурс] − Режим доступу: https://redis.io/ − Дата доступу: 11.11.22
10. Ehcache [Електронний ресурс] − Режим доступу: https://www.ehcache.org/ − Дата доступу: 11.11.22
11. Riak KV Електронний ресурс] − Режим доступу: https://riak.com/products/riak-kv/index.html − Дата доступу: 11.11.22
12. Hazelcast [Електронний ресурс] − Режим доступу: https://hazelcast.com/ − Дата доступу: 11.11.22

13. Design Patterns [Електронний ресурс] − Режим доступу: http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf − Дата доступу: 21.11.22
14. Шаблони проектування. Початок [Електронний ресурс] − Режим доступу: https://travelscode.com/shabloni-proektuvannya-pochatok/ − Дата доступу: 21.11.22
15. Е. Фрімен, Е. Робсон, Б. Бейтс, К. Сієрра Head First. Патерни проєктування, Фабула, 2020, 672 с.
16. Єременко І.Д. «Аналіз програмних засобів шаблонного кешування», Збірник тез XXIV науково-практичної студентської конференції ЗІЕІТ, Запоріжжя, ЗІЕІТ, 2022 - 89 с.
17. Andrew Troelsen, Philip Japikse. Pro C# 7: With .NET and .NET Core 8th Edition, 2019, с. 643-655
18. ASP .NET Documentation [Електронний ресурс] − Режим доступу: https://learn.microsoft.com/en-us/aspnet/core/?WT.mc_id=dotnet-35129-website&view=aspnetcore-7.0 − Дата доступу: 21.11.22
19. NUnit Documentation Site [Електронний ресурс] − Режим доступу: https://docs.nunit.org/ − Дата доступу: 21.11.22
20. Unit Testing: Moq Framework [Електронний ресурс] − Режим доступу: https://learn.microsoft.com/en-us/shows/visual-studio-toolbox/unit-testing-moq-framework − Дата доступу: 21.11.22
21. NuGet Documentation [Електронний ресурс] − Режим доступу: https://learn.microsoft.com/en-us/nuget/ − Дата доступу: 21.11.22
22. Rider [Електронний ресурс] − Режим доступу: https://www.jetbrains.com/ru-ru/rider/ − Дата доступу: 21.11.22
23. Jeffrey Richter. CLR via C#, 2013, с. 802 – 827
24. Caching [Електронний ресурс] − Режим доступу: https://github.com/mr-lldd/dotnet-caching − Дата доступу: 20.12.22

25. Стандарт підприємства. «Методичні вказівки до виконання кваліфікаційних робіт (випускних, дипломних, магістерських). Основні вимоги». СТП 29-2016.
26. ДСТУ 3008-2015. Документація. Звіти у сфері науки і техніки. Структура і правила оформлення / А. Стогній (керівн. розроб.). – Вид. офіц. – [Чинний від 2015-06-22]. – К. : ДП «УкрНДНЦ», 2016. – 26 с. – (Державний стандарт України)

ДОДАТКИ

ДОДАТОК А

ФРАГМЕНТ ЛІСТИНГУ. Клас Caching<T, TStoreFlag>

```
public abstract class Caching<T, TStoreFlag> : ICaching<T, TStoreFlag>
      where TStoreFlag : CachingFlag
   {
      private IStoreOperationProvider StoreOperationProvider { get; set; } = null!;

      private ICacheStore<TStoreFlag> Store { get; set; } = null!;

      protected abstract CachingOptions Options { get; }

      protected abstract string CacheKey { get; }

      protected virtual string CacheKeyDelimiter => ":";

      protected abstract string CacheKeyPrefix { get; }

      public void Populate(IServiceProvider serviceProvider,
         IStoreOperationProvider storeOperationProvider)
      {
         var storeProvider = serviceProvider.GetService<ICacheStoreProvider<TStoreFlag>>();
         if (storeProvider == null) throw new StoreNotFoundException<TStoreFlag>();

         Store = storeProvider.CacheStore;
         StoreOperationProvider = storeOperationProvider;
         EnrichWithDependencies(serviceProvider);
      }

      protected virtual void EnrichWithDependencies(IServiceProvider serviceProvider)
      {
      }

      private string CacheKeyFactory(string suffix)
      {
         return string.Join(CacheKeyDelimiter, CacheKeyPrefix, CacheKey, suffix);
      }

      protected internal Result PerformCaching(T? data, string keySuffix)
      {
         if (!Options.IsCaching) return new DisabledCachingException();

         var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
         var key = CacheKeyFactory(keySuffix);
         return Store.Set(key, data, Options, operation);
      }
```

```
    protected internal ValueTask<Result> PerformCachingAsync(T? data, string keySuffix,
        CancellationToken token = default)
    {
        if (!Options.IsCaching) return new ValueTask<Result>(new DisabledCachingException());
        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);
        return Store.SetAsync(key, data, Options, operation, token);
    }

    protected Result<T> TryGetFromCache(string keySuffix)
    {
        if (!Options.IsCaching) return new DisabledCachingException();

        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);
        return Store.Get<T>(key, operation);
    }

    protected ValueTask<Result<T>> TryGetFromCacheAsync(string keySuffix,
CancellationToken token = default)
    {
        if (!Options.IsCaching) return new ValueTask<Result<T>>(new
DisabledCachingException());

        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);
        return Store.GetAsync<T>(key, operation, token);
    }

    protected Result Refresh(string keySuffix)
    {
        if (!Options.IsCaching) return new DisabledCachingException();

        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);

        return Store.Refresh(key, operation);
    }

    protected ValueTask<Result> RefreshAsync(string keySuffix, CancellationToken token)
    {
        if (!Options.IsCaching) return new ValueTask<Result>(new DisabledCachingException());

        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);
        return Store.RefreshAsync(key, operation, token);
    }

    protected Result Remove(string keySuffix)
    {
        if (!Options.IsCaching) return new DisabledCachingException();
```

```
        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);

        return Store.Remove(key, operation);
    }

    protected ValueTask<Result> RemoveAsync(string keySuffix, CancellationToken token)
    {
        if (!Options.IsCaching) return new ValueTask<Result>(new DisabledCachingException());

        var operation = StoreOperationProvider.Next(CacheKeyDelimiter);
        var key = CacheKeyFactory(keySuffix);
        return Store.RemoveAsync(key, operation, token);
    }
}
```

ДОДАТОК Б
ФРАГМЕНТ ЛІСТИНГУ. Клас Cache<T, TStoreFlag>

```
public abstract class Cache<T, TStoreFlag> : Caching<T, TStoreFlag>,
      ICache<T, TStoreFlag>,
      IInternalCache<T, TStoreFlag>
      where TStoreFlag : CachingFlag
   {
      protected sealed override string CacheKeyPrefix => "cache";

      protected virtual string CacheKeySuffix
      {
         get
         {
            var type = typeof(T);
            return $"{type.Namespace}.{type.Name}";
         }
      }

      public ValueTask<Result> SetAsync(T value, CancellationToken token = default)
      {
         return PerformCachingAsync(value, CacheKeySuffix, token);
      }

      public Result Set(T value)
      {
         return PerformCaching(value, CacheKeySuffix);
      }

      public ValueTask<Result<T>> GetAsync(CancellationToken token = default)
      {
         return TryGetFromCacheAsync(CacheKeySuffix, token);
      }

      public Result<T> Get()
      {
         return TryGetFromCache(CacheKeySuffix);
      }

      public ValueTask<Result> RefreshAsync(CancellationToken token = default)
      {
         return RefreshAsync(CacheKeySuffix, token);
      }

      public Result Refresh()
      {
```

```
        return Refresh(CacheKeySuffix);
    }

    public ValueTask<Result> RemoveAsync(CancellationToken token = default)
    {
        return RemoveAsync(CacheKeySuffix, token);
    }

    public Result Remove()
    {
        return Remove(CacheKeySuffix);
    }

    protected sealed override void EnrichWithDependencies(IServiceProvider serviceProvider)
    {
        base.EnrichWithDependencies(serviceProvider);
    }
}
```

ДОДАТОК В
ФРАГМЕНТ ЛІСТИНГУ. Клас Cache<T, TStoreFlag>

```
public abstract class CachingLoader<TArgs, TResult, TStoreFlag> : Caching<TResult,
TStoreFlag>,
    ICachingLoader<TArgs, TResult, TStoreFlag>, IInternalLoader<TArgs, TResult, TStoreFlag>
    where TResult : class
    where TStoreFlag : CachingFlag
  {
    protected ILoader<TArgs, TResult> Loader { get; private set; } = null!;

    protected sealed override string CacheKeyPrefix
      => "loader";

    public async ValueTask<Result<TResult>> GetOrLoadAsync(TArgs args, bool
omitCacheOnLoad = false,
      CancellationToken token = default)
    {
      var keySuffix = CacheKeySuffixFactory(args);
      if (!omitCacheOnLoad)
      {
        var gettingTask = TryGetFromCacheAsync(keySuffix, token);
        var inCache = gettingTask.IsCompletedSuccessfully
          ? gettingTask.Result
          : await gettingTask;
        if (inCache.Successful) return inCache.UnwrapAsSuccess();
      }

      var loaded = await Loader.LoadAsync(args, token);
      var onLoadFinishTask = OnLoadFinishAsync(args, loaded, token);
      if (!onLoadFinishTask.IsCompletedSuccessfully)
      {
        await onLoadFinishTask;
      }
      var cachingTask = PerformCachingAsync(loaded, keySuffix, token);
      if (!cachingTask.IsCompletedSuccessfully)
      {
        await cachingTask;
      }

      return loaded;
    }

    public Result<TResult> GetOrLoad(TArgs args, bool omitCacheOnLoad = false,
CancellationToken token = default)
    {
```

```
            var keySuffix = CacheKeySuffixFactory(args);
            if (!omitCacheOnLoad)
            {
                var fromCache = TryGetFromCache(keySuffix);
                if (fromCache.Successful) return fromCache;
            }

            var loaded = Loader.LoadAsync(args, token).GetAwaiter().GetResult();
            var onLoadFinishTask = OnLoadFinishAsync(args, loaded, token);
            if (!onLoadFinishTask.IsCompletedSuccessfully)
            {
                onLoadFinishTask.GetAwaiter().GetResult();
            }
            PerformCaching(loaded, keySuffix);
            return loaded;
        }

        public ValueTask<Result> SetAsync(TArgs args, TResult result, CancellationToken token =
default)
        {
            return PerformCachingAsync(result, CacheKeySuffixFactory(args), token);
        }

        public Result Set(TArgs args, TResult result)
        {
            return PerformCaching(result, CacheKeySuffixFactory(args));
        }

        public ValueTask<Result<TResult>> GetAsync(TArgs args, CancellationToken token =
default)
        {
            return TryGetFromCacheAsync(CacheKeySuffixFactory(args), token);
        }

        public Result<TResult> Get(TArgs args)
        {
            return TryGetFromCache(CacheKeySuffixFactory(args));
        }

        public ValueTask<Result> RefreshAsync(TArgs args, CancellationToken token = default)
        {
            return RefreshAsync(CacheKeySuffixFactory(args), token);
        }

        public Result Refresh(TArgs args)
        {
            return Refresh(CacheKeySuffixFactory(args));
        }

        public ValueTask<Result> RemoveAsync(TArgs args, CancellationToken token = default)
        {
```

```
        return RemoveAsync(CacheKeySuffixFactory(args), token);
    }

    public Result Remove(TArgs args)
    {
        return Remove(CacheKeySuffixFactory(args));
    }

    protected sealed override void EnrichWithDependencies(IServiceProvider serviceProvider)
    {
        base.EnrichWithDependencies(serviceProvider);
        var foundLoader = serviceProvider.GetService<ILoader<TArgs, TResult>>();
        Loader = foundLoader ?? throw new LoaderNotFoundException<TArgs, TResult>();
    }

    protected abstract string CacheKeySuffixFactory(TArgs args);

    protected virtual ValueTask OnLoadFinishAsync(TArgs args, TResult result,
CancellationToken token) => new();
}
```