Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти другий (магістерський)

Дослідження архітектурних підходів до проєктування невеликих функціональних веб-додатків
(тема)

Виконав:
студент (ка) 2 курсу, групи ІПЗм-22-4

Юдін І.О.
(прізвище, ініціали)

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

Керівник проф. Руткас А.Г.
(посада, прізвище, ініціали)

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

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук

Кафедра програмної інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(шифр і назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Юдіну Іллі Олександровичу
(прізвище, ім'я, по батькові)

1. Тема роботи *«Дослідження архітектурних підходів до проєктування невеликих функціональних веб-додатків»*

Затверджена наказом по університету від 29.03. 2024р. № 250 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 21.06.2024

3. Вихідні дані до роботи: *дослідження архітектурних підходів до проєктування невеликих функціональних веб-додатків.*

4. Перелік питань, що потрібно опрацювати в роботі *вступ, аналіз предметної галузі, аналіз існуючих архітектурних підходів, порівняння архітектур та підходів, проведення експерименту, аналіз результатів, висновки.*

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Огляд літератури, аналіз проблеми та постановка задачі дослідження	22.01.2024 – 12.02.2024	виконано
2	Дослідження архітектурних підходів до проєктування невеликих функціональних веб-додатків	12.02.2024 – 25.03.2024	виконано
3	Програмна реалізація застосунку	25.02.2024 – 18.05.2024	виконано
4	Проведення експериментальних досліджень та аналіз результатів	01.04.2024 – 29.04.2024	виконано
5	Підготовка пояснювальної записки	22.04.2024 – 20.05.2024	виконано
6	Перевірка на плагіат та нормоконтроль	08.06.2024	виконано
7	Підготовка презентації та доповіді	12.06.2024	виконано
8	Рецензування	15.06.2024	виконано
9	Попередній захист	19.06.2024	виконано
10	Занесення диплома в електронний архів	19.06.2024	виконано
11	Допуск до захисту у зав. кафедри	20.06.2024	виконано

Дата видачі завдання 22.01.2024 р.

Студент (ка) _____ Юдін І.О.
(підпис)

Керівник роботи _____ проф. Руткас А.Г.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Пояснювальна записка: 63 стор., 13 рис., 1 табл., 13 джерел.

АРХІТЕКТУРНІ ПІДХОДИ, ВЕБ-ДОДАТКИ, ІНТЕГРАЦІЯ, ОПТИМІЗАЦІЯ, ПРОЄКТУВАННЯ.

Об'єктом дослідження є архітектурні підходи до проєктування невеликих функціональних веб-додатків. Задачею дослідження є аналіз, розробка та впровадження ефективних методів і технік проєктування, які сприяють покращенню якості та продуктивності веб-додатків.

Мета роботи – дослідження архітектурних підходів до проєктування невеликих функціональних веб-додатків, аналіз та порівняння існуючих методів і технік проектування, доскональне вивчення принципів роботи кожного з обраних підходів, розробка рекомендацій щодо впровадження найбільш ефективних архітектурних рішень, експериментальне створення власного функціонального веб-додатку та проведення тестування і оптимізації розробленого рішення.

У процесі наукового дослідження застосовуються передові підходи сучасних технологій розробки програмного забезпечення, зокрема ті, що пов'язані з оптимізацією архітектури веб-додатків. Великий акцент приділяється розгляду різноманітних архітектурних стилів та методів, їх адаптації до потреб невеликих проєктів, а також інтеграції з існуючими системами.

Основні напрямки дослідження включають аналіз існуючих архітектурних підходів, визначення ключових вимог до проєктування функціональних веб-додатків та розробку рекомендацій щодо впровадження оптимізованих архітектурних рішень.

В результаті дослідження розробляються та впроваджуються комплексні стратегії для проєктування та оптимізації невеликих функціональних веб-додатків. Це сприяє не лише покращенню якості та продуктивності розробки, але й забезпечує успішну інтеграцію змін у швидкозмінному середовищі веб-технологій.

ARCHITECTURAL APPROACHES, WEB APPLICATIONS, DESIGN, OPTIMIZATION, INTEGRATION.

The object of in-depth analysis and research is the architectural approaches to designing small functional web applications. The research aims to analyze, develop, and implement effective methods and techniques that enhance the quality and performance of web applications.

The purpose of the work is to study architectural approaches to designing small functional web applications, analyze and compare existing design methods and techniques, thoroughly study the principles of operation of each of the selected approaches, develop recommendations for implementing the most effective architectural solutions, experimentally create your own functional web application, and test and optimize the developed solution.

In the process of scientific research, advanced approaches of modern software development technologies are applied, particularly those related to optimizing the architecture of web applications. A significant emphasis is placed on examining various architectural styles and methods, their adaptation to the needs of small projects, and integration with existing systems.

The main research directions include the analysis of existing architectural approaches, identifying key requirements for designing functional web applications, and developing recommendations for implementing optimized architectural solutions.

As a result of the research, comprehensive strategies for designing and optimizing small functional web applications are developed and implemented. This not only improves the quality and performance of development but also ensures the successful integration of changes in the rapidly evolving web technology environment.

Я, Юдін Ілля Олександрович, студент гр. ІПЗм-22-4, здобувач вищої освіти на другому (магістерському) рівні кафедри «Програмна інженерія», заявляю: моя робота на тему «Дослідження архітектурних підходів до проєктування невеликих

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

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

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

API – Application Programming Interface;

C# – C Sharp;

FTS – Full Text Search;

IEEE - Institute of Electrical and Electronics Engineers;

S&T – Science and Technology;

URL – Uniform Resource Locator.

ВСТУП

Зміни в сучасному світі, обумовлені швидким розвитком інформаційних технологій та великим об'ємом даних, вплинули на багато галузей. Однією з ключових областей, яка не залишилася осторонь від цих трансформацій, є розробка веб-застосунків.

Веб-застосунки стали важливою складовою інтернету та вирішують широкий спектр завдань - від простих веб-сайтів до складних корпоративних систем. Однак розвиток веб-застосунків не обійшовся без своїх викликів та проблем.

Особливості предметної області веб-застосунків включають постійні зміни у технологічних стеках, високі вимоги до продуктивності, безпеки та доступності, а також потребу в швидкому впровадженні та масштабуванні. Важливим фактором є також специфіка взаємодії з користувачами та розвиток інтерфейсів для забезпечення комфортного та інтуїтивного користувацького досвіду.

У нашій роботі ми будемо звертати увагу на ці особливості та вирішувати актуальну проблему, пов'язану із покращенням якості та ефективності розробки веб-застосунків. Ми досліджуватимемо інноваційні підходи, методи та інструменти, які допоможуть нам оптимізувати розробку та забезпечити успішну інтеграцію змін у цій швидкозмінній галузі.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Загальна характеристика маппінгу

Архітектура веб-застосунків – це важлива галузь у сфері інформаційних технологій, яка дозволяє створювати та підтримувати веб-додатки, що працюють через мережу Інтернет. Ця область має багато історичних витоків та особливостей побудови, які варто розглянути.

Архітектура веб-застосунків сформувалася разом з розвитком Інтернету та поширенням веб-технологій. Початково, веб-сторінки були статичними і використовувались для відображення інформації. Проте з ростом популярності вебу та збільшенням обсягу інтерактивних веб-додатків, стала необхідною більш складна архітектура.

Однією з важливих подій у розвитку архітектури веб-застосунків було створення та популяризація технологій CGI (Common Gateway Interface) у 1990-х роках. Ця технологія дозволила взаємодіяти з сервером і генерувати веб-сторінки на основі даних, що були передані на сервер.

У подальшому, розвиток мов програмування для вебу (наприклад, PHP, Ruby, Python, JavaScript) та серверних середовищ (Apache, Nginx) сприяв появі багатошарових архітектур та фреймворків для створення веб-застосунків.

Зазвичай, веб-додатки використовують модель взаємодії "клієнт-сервер", де клієнт (браузер) відправляє запити до сервера, а сервер відповідає з веб-сторінками чи даними.

Для ефективності та забезпечення модульності веб-застосунки часто будуються у вигляді багатьох шарів (наприклад, шар бази даних, логіки застосунку, інтерфейсного шару).

Протокол HTTP і архітектурний стиль REST (Representational State Transfer) використовуються для взаємодії між клієнтом і сервером у веб-додатках.

Застосунки повинні бути розроблені та налаштовані для масштабованості та високої доступності, особливо якщо вони мають велику аудиторію.

Архітектура веб-застосунків є необхідною з багатьох причин:

- веб-застосунки повинні виконувати певні функції, і архітектурні рішення допомагають створити логіку для цього;
- веб-застосунки можуть вдосконалюватися та розширюватися з плином часу, і правильна архітектура допомагає забезпечити легкість розширення;
- архітектурні рішення допомагають забезпечити безпеку додатку, включаючи захист від атак та зламів;
- правильна архітектура робить розробку, тестування та підтримку веб-додатків більш ефективними.
- для багатьох застосунків важлива висока доступність та надійність, і це можливо завдяки правильній архітектурі.

Усі ці аспекти обумовлюють необхідність архітектурного підходу до розробки та підтримки веб-застосунків. Архітектура веб-застосунків допомагає зробити ці застосунки ефективними, масштабованими, безпечними та доступними для користувачів.

Компактні веб-застосунки (іноді їх також називають "мікро-веб-застосунками") - це тип веб-застосунків, які розроблені та побудовані з акцентом на мінімізацію обсягу коду, функціональності та завантаження. Основна ідея компактних веб-застосунків - надавати лише обмежений набір функцій, які необхідні для конкретної задачі чи послуги, і робити це максимально ефективно.

Компактні веб-застосунки створюються з дуже обмеженою кількістю коду, що робить їх невеликими та легкими для розгортання та підтримки.

Вони зазвичай спрямовані на вирішення конкретної задачі або надання певної послуги, що дозволяє їм бути високо спеціалізованими.

Мінімізація зайвого коду дозволяє компактним веб-застосункам працювати швидше і більш продуктивно.

Компактні веб-застосунки зазвичай легше масштабувати, оскільки вони мають обмежену функціональність і можуть бути розгорнуті як окремі екземпляри.

Зазвичай ці застосунки мають високу доступність, оскільки вони можуть бути розгорнуті на різних серверах або контейнерах для забезпечення надійності.

Мінімізація складності дозволяє зосередитися на розробці та підтримці ключових функцій без зайвих витрат часу і ресурсів. Вони можуть бути легко інтегровані в інші системи або мікросервіси завдяки своїй модульній природі.

Багато компактних веб-застосунків розгортаються в мікросервісному оточенні, де кожен компонент відповідає за кнкретну функцію.

Загалом, компактні веб-застосунки є важливим елементом сучасного програмування, оскільки вони дозволяють створювати швидкі, продуктивні та легко масштабовані рішення для конкретних завдань і послуг. Цей підхід допомагає оптимізувати використання ресурсів і зменшувати складність розробки та підтримки веб-застосунків.

1.2 Актуальність проблеми

Актуальність проблеми полягає в складності вибору оптимальної архітектури для маленьких застосунків. З одного боку, монолітна архітектура простіша у розробці та розгортанні, але обмежена в масштабуванні та ускладнює внесення змін.

З іншого боку, мікросервісна чи хмарна архітектура пропонують гнучкість та масштабованість, але вимагають більш складного управління та знань. Вибір поміж цими архітектурами важливий, адже він впливає на розвиток, підтримку, продуктивність та вартість проекту.

Тому знайти баланс, який відповідає потребам конкретного застосунку, є ключовою проблемою.

Вибір правильного архітектурного підходу від самого початку є критично важливим, оскільки це впливає на подальшу еволюцію та підтримку застосунку. Невірний вибір може призвести до збільшених витрат на розробку, складнощів у підтримці, проблем із масштабуванням і оновленнями, а також потенційних проблем з продуктивністю.

Раннє визначення архітектури дозволяє планувати ресурси, час та бюджет ефективніше, а також забезпечує більшу гнучкість у відповідності до майбутніх потреб та вимог.

Саме тому є необхідність встановити рекомендації та стандарти для вибору архітектурного підходу для компактних веб-застосунків.

1.3 Постановка задачі

Уході даної роботи нами буде досліджено предметну область підходів та архітектурних стандартів побудови веб застосунків, буде досліджено популярні підходи та виявлено їх ключові особливості на основі яких в кінці буде сформовано ідеальний підхід чи патерн для побудови компактних застосунків.

Ми проведемо порівняльний аналіз методів та фундаментальних підходів для визначення ідеальних елементів для нашої мети.

Ця робота спрямована на виявлення критеріїв для вибору архітектурних патернів.

Реалізація наукового дослідження складається з наступних етапів:

– аналіз предметної області;

– аналіз архітектурних особливостей сучасних архітектурних патернів;

– знаходження фундаментальних закономірностей;

– порівняльний аналіз та модифікація існуючих рішень.

2 АНАЛІЗ ІСНУЮЧИХ АРХІТЕКТУРНИХ ПІДХОДІВ

2.1 Cloud native architecture

Внутрішня хмарна архітектура розроблена спеціально для додатків, які планується розгортати в хмарі, і мікросервіси є критично важливою частиною.

Cloud native – це підхід до створення та запуску програм, який використовує переваги моделі доставки хмарних обчислень. Cloud native – це термін, який використовується для опису контейнерних середовищ, і він стосується того, як створюються та розгортаються програми, а не де.

Власні хмарні технології дають нам можливість запускати програми в загальнодоступних, приватних і гібридних хмарах. Внутрішня хмарна розробка є важливою для швидкого виведення програм на ринок; він допомагає людям, процесам і технологіям створювати, розгортати та керувати програмами, готовими до хмари.

Замість одного великого застосунку, функціональність розділяється на незалежні мікросервіси, кожен з яких відповідає за певну частину бізнес-логіки. Це дозволяє розробляти, оновлювати та масштабувати кожний сервіс окремо.

Додатки упаковуються разом з усім необхідним середовищем в контейнери, що забезпечує стандартизацію та портативність середовища на різних платформах.

Використовується еластичність хмарних сервісів для автоматичного масштабування та оптимального розподілу ресурсів в залежності від поточних потреб застосунку.

Застосовується підхід безперервної інтеграції та безперервного розгортання (CI/CD), що дозволяє швидко вносити зміни в застосунки з мінімальними перервами у їх роботі.

Далі розглянемо переваги та недоліки цієї архітектури.

Переваги Cloud-Native архітектури:

– гнучкість та масштабованість: легке масштабування сервісів в залежності від потреб;

- більш швидке впровадження: завдяки автоматизації ci/cd, застосунки можна швидше розгортати;
- ефективне використання ресурсів: оптимізація витрат завдяки платежам лише за використані ресурси;
- відмінна витривалість: висока доступність і стійкість до відмов.

Недоліки Cloud-Native архітектури:

- складність управління: управління численними мікросервісами може бути складним;
- проблеми з безпекою: високі вимоги до забезпечення безпеки і відповідності стандартам;
- високі вимоги до навичок: потребує кваліфікованих розробників із знанням хмарних технологій;
- можливий vendor lock-in: залежність від конкретного хмарного провайдера може ускладнити міграцію.

Далі на рисунку 2.1 наведено архітектурну структуру Cloud-Native Архітектури.

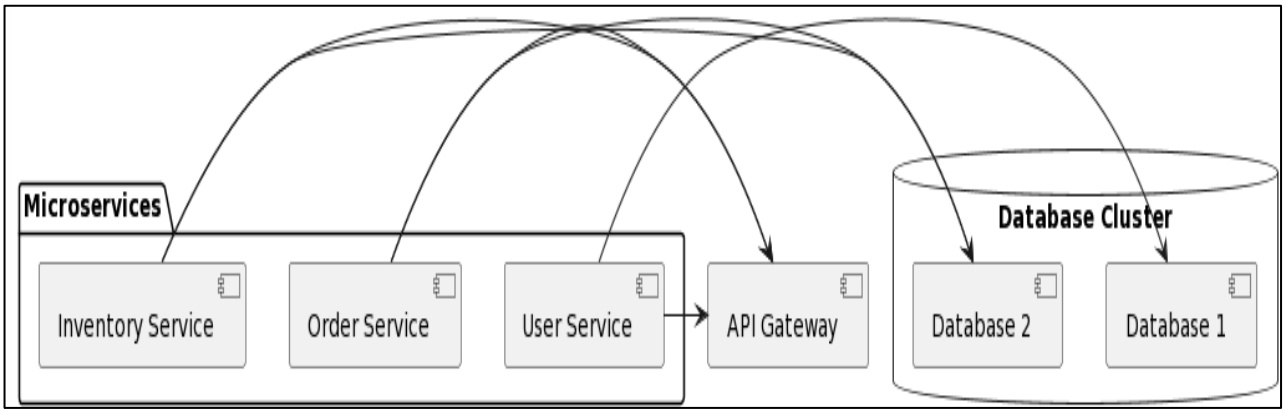

Рисунок 2.1 – Архітектурна структура Cloud-Native Архітектури

Ця архітектура представляє собою класичний приклад Cloud-Native підходу, де система розділена на мікросервіси.

API Gateway служить єдиною вхідною точкою для зовнішніх запитів і направляє їх до відповідних мікросервісів.

Мікросервіси (User Service, Order Service, Inventory Service) відповідають за окремі функції або бізнес-логіку. Вони працюють незалежно один від одного.

Database Cluster містить бази даних (Database 1 та Database 2), які використовуються різними мікросервісами.

Кожен мікросервіс може взаємодіяти з API Gateway та обмінюватися даними з базою даних, що забезпечує гнучкість та масштабованість системи.

2.2 Мікросервісна архітектура

У хмарній моделі складні додатки, створені як набір послуг і даних, повністю відокремлені від додатків. Мікросервіси – це архітектурний стиль, який структурує додаток як набір сервісів. Кожен сервіс можна написати на іншій мові програмування та протестувати окремо. Вони розгортаються незалежно та організовані навколо бізнес-можливостей.

Візьмемо приклад програми електронної комерції, розробленої з використанням архітектури мікросервісів. Кожен мікросервіс може зосереджуватися на одній бізнес-можливості (наприклад, кошик для покупок, пошук, огляд клієнтів). Кожна з них може бути окремою службою, написаною різними мовами програмування, розгорнутою в різній інфраструктурі та керованою різними командами.

Кожна служба спілкується з іншими за допомогою спрощеного протоколу.

Загалом, мікросервісна архітектура – це підхід у розробці програмного забезпечення, де застосунок розбивається на набір невеликих, незалежних служб. Кожен мікросервіс фокусується на виконанні однієї специфічної функції або бізнес-задачі і може бути розроблений, розгорнутий, працювати і оновлюватися незалежно від інших. Це забезпечує більшу гнучкість, легкість управління змінами та масштабуванням, а також сприяє неперервній інтеграції та доставці (CI/CD). Мікросервіси зазвичай взаємодіють через визначені API, що забезпечує їхню взаємодію в межах складної системи.

Мікросервіси розроблені таким чином, щоб кожен сервіс виконував одну конкретну функцію. Це розділення відповідальностей дозволяє легше управляти кодом, ізолювати помилки та полегшує розширення функціональності.

Кожен мікросервіс може бути розгорнутий незалежно, що дозволяє швидше впроваджувати оновлення та нові функції без ризику для всього застосунку.

Мікросервіси можуть масштабуватися незалежно один від одного. Це означає, що можна збільшити ресурси для конкретного сервісу, що відчуває підвищений попит, без впливу на інші сервіси.

Окремі команди можуть працювати над різними мікросервісами паралельно, що сприяє більш швидкій та ефективній розробці. Це також сприяє агільній методології та неперервному вдосконаленню продукту.

Переваги мікросервісної архітектури:

- незалежне масштабування: кожен мікросервіс може масштабуватися окремо в залежності від потреб;
- гнучкість та швидкість розгортання: незалежні мікросервіси можна розгортати та оновлювати без впливу на решту системи;
- відмінна витривалість та надійність: помилки в одному мікросервісі менше впливають на загальну систему.

Недоліки мікросервісної архітектури:

- складність управління: велика кількість мікросервісів може ускладнювати управління;
- загрози безпеки: кожен мікросервіс має свої точки доступу, що збільшує ризики безпеки;
- затрати на розробку та обслуговування: розробка та підтримка численних мікросервісів може бути ресурсомісткою.

Далі на рисунку 2.2 наведено архітектурну структуру мікросервісної архітектури.

Мікросервісна архітектура складається з таких ключових елементів:

- мікросервіси: незалежні сервіси, кожен з яких виконує певну функцію або бізнес-процес;

– арі шлюз: централізована точка доступу, яка управляє трафіком до різних мікросервісів;
– сервісний реєстр: використовується для відстеження і виявлення сервісів в мережі;
– бази даних: кожен мікросервіс може мати власну базу даних, ізоляцію даних;
– контейнеризація: мікросервіси часто упаковуються в контейнери для легшого розгортання та управління.

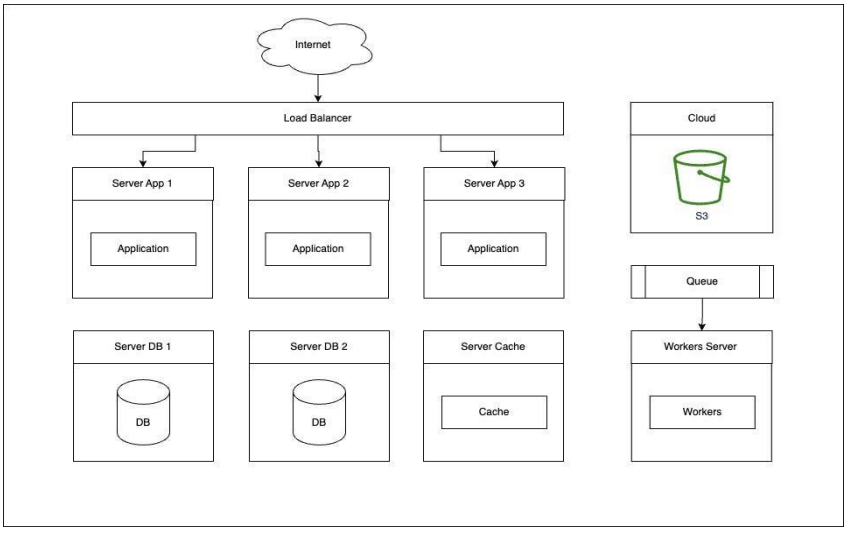

Рисунок 2.2 – Архітектурна структура мікросервісної архітектури

Взаємодія між цими компонентами здійснюється через мережеві запити, зазвичай HTTP/REST або асинхронні повідомлення, які дозволяють сервісам обмінюватися даними та функціональністю.

2.3 Монолітна архітектура

Монолітна архітектура – це традиційна модель програмного забезпечення, яка побудована як уніфікована одиниця, самодостатня та незалежна від інших програм. Слово «моноліт» часто приписують чомусь великому та льодовиковому, що не далеко від істини монолітної архітектури для розробки програмного забезпечення. Монолітна архітектура – це окрема велика обчислювальна мережа з єдиною кодовою базою, яка об’єднує всі бізнес-завдання.

Щоб внести зміни в програму такого типу, потрібно оновити весь стек, отримавши доступ до бази коду та створивши та розгорнувши оновлену версію інтерфейсу служби. Це робить оновлення обмежувальними та забирає багато часу.

Моноліти можуть бути зручними на ранніх стадіях життя проекту для полегшення керування кодом, когнітивних витрат і розгортання. Це дозволяє звільнити все в моноліті одночасно.

У монолітній архітектурі весь код застосунку, включаючи інтерфейс користувача, бізнес-логіку та базу даних, інтегрований у один єдиний кодовий файл. Це може спростити відстеження залежностей та зменшити кількість конфліктів у коді.

Оскільки застосунок існує як один суцільний блок, розгортання полягає у запуску одного виконуваного файлу або деплоїнгу на один сервер. Це робить процес розгортання відносно простим.

Масштабування монолітного застосунку може бути складним, оскільки це вимагає масштабування цілого застосунку, а не окремих його частин. Це може викликати проблеми з продуктивністю та ефективністю.

Оновлення або додавання функціоналу у монолітний застосунок може бути важким, оскільки зміни в одній частині програми можуть вплинути на інші. Це може призвести до додаткових помилок та складностей у тестуванні.

Далі на рисунку 2.3 наведено архітектурну структуру монолітної архітектури.

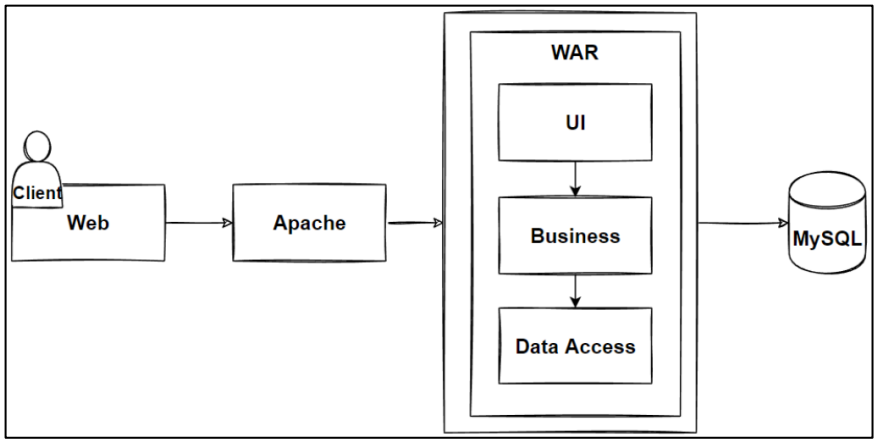

Рисунок 2.3 – Архітектурна структура монолітної архітектури

2.4 Event-driven serverless architecture та Cloud-based architecture

Архітектура, керована подіями (EDA), базується на роз’єднаних системах, які працюють у відповідь на події. Архітектура, керована подіями, використовує події для запуску та обміну даними між роз’єднаними службами. EDA існує вже давно, але тепер він має більшу актуальність у хмарі.

При правильному використанні вона може забезпечити значне збільшення маневреності, економії коштів і операційних переваг. Розподілений безсерверний EDA може виконувати код, відомий як функції, які автоматично масштабуються у відповідь на REST API або тригер події.

Для без серверної моделі керування сервером не потрібне. Безсерверна модель також швидко масштабується (тому можливі швидкі оновлення та розгортання) і не має стану.

Архітектура на основі хмарних технологій (Cloud-based architecture) - це архітектурний підхід до проектування і розробки програмних систем, який передбачає використання хмарних обчислень та послуг для створення, розгортання та обслуговування програмних застосунків. Вона ґрунтується на ідеї використання віддалених серверів та ресурсів, що надаються через Інтернет, для виконання обчислень, зберігання даних та надання доступу до програмних послуг.

Хмарні платформи надають можливість легко збільшувати або зменшувати ресурси в залежності від потреб. Це дозволяє підтримувати високий рівень доступності та продуктивності застосунків.

За допомогою хмарних обчислень можна забезпечити високий рівень доступності застосунків, завдяки розміщенню на багатьох серверах та автоматичному відновленню в разі відмови обладнання.

Хмарні ресурси можуть бути легко адаптовані до змінних навантажень. Це дозволяє оптимізувати витрати на обслуговування.

Користувачі можуть отримувати доступ до застосунків та даних з будь-якого місця та пристрою, якщо є Інтернет.

Хмарні платформи надають інструменти для автоматизації розгортання, моніторингу та керування застосунками та ресурсами.

Важливий аспект архітектури на основі хмарних технологій - це забезпечення безпеки. Хмарні провайдери зазвичай надають широкий спектр інструментів та послуг для захисту даних та інфраструктури.

Платність за використання: Більшість хмарних платформ пропонують модель оплати за використання, що дозволяє зменшити витрати на обладнання та інфраструктуру.

Застосування архітектури на основі хмарних технологій може бути корисним для різних видів застосунків, від веб-сервісів та мобільних додатків до обробки даних та штучного інтелекту. Вона дозволяє розробникам швидко розгортати та масштабувати свої застосунки без значних інвестицій у власну інфраструктуру.

Далі на рисунку 2.4 наведено архітектурну структуру Event-driven serverless архітектури.

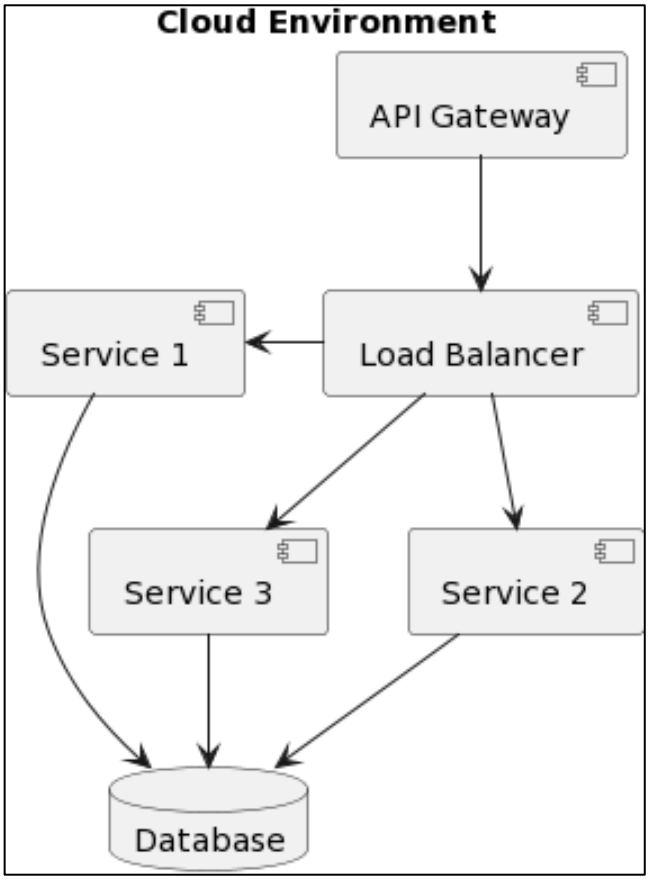

Рисунок 2.4 – Архітектурна структура Event-driven serverless архітектури

Плюси Cloud-based architecture:

– масштабованість: хмарні платформи надають можливість легко масштабувати ресурси в залежності від потреб. ви можете збільшувати або

зменшувати потужність обчислень, не потребуючи значних інвестицій у обладнання;

- доступність і надійність: хмарні постачальники зазвичай забезпечують високий рівень доступності, завдяки розміщенню даних та сервісів на багатьох серверах та резервному копіюванню. це дозволяє підтримувати високий рівень продуктивності та доступності;
- еластичність: ви можете легко змінювати розмір і обсяг ресурсів згідно з потребами. це особливо корисно для застосунків, які мають варіабельний обсяг навантаження;
- глобальний доступ: користувачі можуть отримувати доступ до застосунків та даних з будь-якого місця та пристрою, якщо є інтернет;
- автоматизація і оркестрація: хмарні платформи надають інструменти для автоматизації розгортання, моніторингу та керування застосунками та ресурсами. це спрощує адміністрування та забезпечує автоматичну масштабованість;
- безпека: більшість хмарних постачальників забезпечують рівень безпеки на високому рівні, включаючи шифрування даних, мережевий захист та ідентифікацію користувачів;
- оплата за використання: модель оплати за використання дозволяє ефективно використовувати ресурси та оплачувати лише те, що ви використовуєте, зменшуючи витрати на обслуговування.

Мінуси Cloud-based architecture:

- залежність від постачальника: використання хмарних послуг робить вас залежними від конкретного хмарного постачальника. ви обмінюєте контроль над інфраструктурою на більш високий рівень обслуговування;
- приватність і безпека даних: збереження даних у хмарних сервісах може породити питання щодо конфіденційності та безпеки даних, особливо для чутливої інформації;

– можливість відмови сервісу: якщо хмарний постачальник виникає проблеми або припиняє свою роботу, це може суттєво вплинути на доступність вашого застосунку;
– вартість: в хмарних послугах можуть бути певні витрати, особливо якщо ви використовуєте багато ресурсів або послуг;
– обмеження ресурсів: у хмарних рішеннях існують обмеження на кількість та типи ресурсів, які ви можете використовувати.

Далі на рисунку 2.5 наведено архітектурну структуру Cloud-based архітектури.

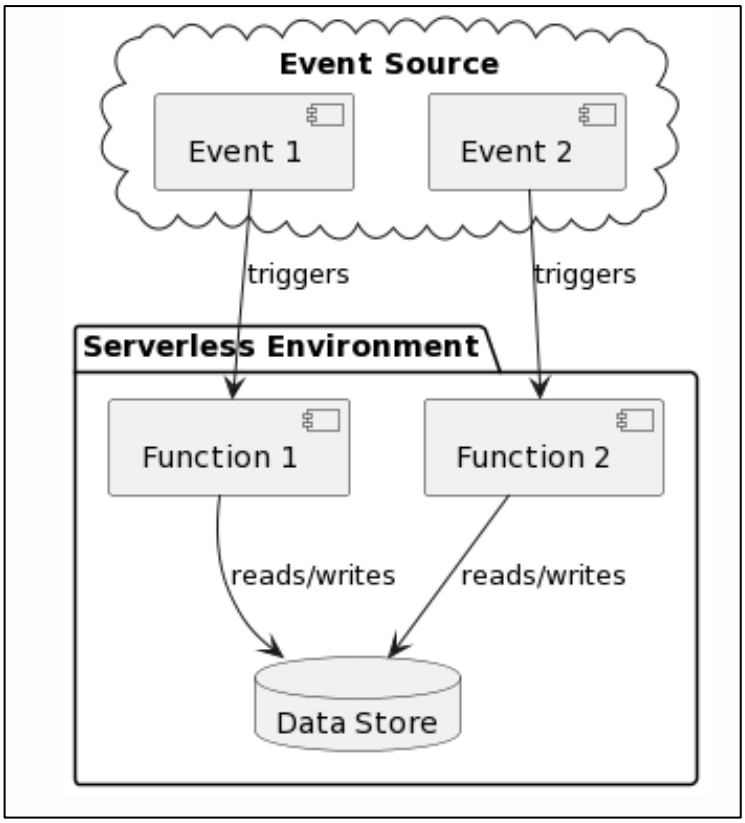

Рисунок 2.5 – Архітектурна структура Cloud-basedархітектури

У Event-driven serverless архітектурі основними складовими є:

– джерела подій (event sources): генерують події, що активують serverless функції;
– serverless функції: спеціалізовані функції, що викликаються відповідно до подій;

- системи зберігання даних (data stores): бази даних або сховища, з якими взаємодіють функції;
- api шлюзи: управляють вхідними запитами та розподіляють їх між функціями.

Функції без сервера реагують на події, обробляють запити та здійснюють взаємодію з базами даних або іншими службами, забезпечуючи динамічне та гнучке управління застосунком.

3 ПОРІВНЯННЯ АРХІТЕКТУР ТА ПІДХОДІВ

3.1 Виведення оптимального структурного архітектурного патерну

Монолітна та мікросервісна архітектури – це два різні підходи до проектування та розробки програмних застосунків. Кожен з них має свої переваги і обмеження, і вибір між ними залежить від конкретних потреб проекту.

Монолітна архітектура передбачає розробку і підтримку одного великого застосунку, де всі функції і компоненти розташовані разом в одному кодові.

Розробка, тестування і розгортання монолітного застосунку може бути більш простою, оскільки всі частини додатку знаходяться в одному місці.

Масштабувати монолітну архітектуру може бути складніше, оскільки всі компоненти пов'язані між собою і однаково масштабуються.

Зазвичай, оновлення однієї частини монолітного застосунку може призвести до важливих змін у всьому додатку, що може бути ризикованим і вимагати великих тестів.

Для менших проектів або там, де складність архітектури не є ключовою, монолітна архітектура може бути більш простою та зрозумілою.

Мікросервісна архітектура розбиває програму на невеликі, незалежні сервіси, які комунікують між собою через API. Кожен сервіс може бути розроблений, випробуваним і масштабованим окремо.

Мікросервіси можна масштабувати незалежно, що дозволяє оптимізувати використання ресурсів та забезпечити більшу гнучкість.

Підтримка та оновлення окремих сервісів є менш проблематичними, оскільки вони ізольовані один від одного.

Мікросервісна архітектура може бути складною для розробки та керування через велику кількість сервісів, які взаємодіють.

Для підтримки мікросервісної архітектури може бути необхідною додаткова інфраструктура, така як контейнери або оркестратори.

Для складних систем або великих проектів, де гнучкість і розширення є ключовими, мікросервісна архітектура може бути більш підходящою.

Монолітна архітектура може бути кращим вибором для невеликого додатку з кількома причинами:

Монолітний додаток включає всі його компоненти в одному кодові. Це може спростити розробку, тестування та супровід додатку, особливо для невеликих команд або початківців розробників.

У монолітній архітектурі немає необхідності в управлінні великою кількістю сервісів, як це буває у мікросервісній архітектурі. Це дозволяє уникнути зайвої складності та витрат на інфраструктуру.

Розробка та впровадження монолітного додатку може бути швидше завершена, оскільки всі компоненти розташовані разом. Це може бути важливим для швидкого виведення продукту на ринок.

Монолітний додаток може запускатися на одному сервері або в обмеженій обчислювальній інфраструктурі, що дозволяє заощадити гроші на обслуговуванні.

Оскільки всі компоненти розташовані в одному додатку, менше часу витрачається на оркестрацію та керування сервісами.

У команді розробників, яка має обмежені ресурси або обмежений досвід, працювати над монолітним додатком може бути більш продуктивно, оскільки комунікація та спільна робота менше ускладнені.

Звісно, важливо пам'ятати, що монолітна архітектура може мати свої обмеження, особливо при подальшому рості додатку. Якщо додаток зростає за розмірами і складністю, може виникнути необхідність переходити до більш розгалуженої архітектури, такої як мікросервіси. Але для невеликих додатків з обмеженими ресурсами та швидкими вимогами монолітна архітектура може бути ефективним та швидким способом впровадження програмного продукту.

3.2 Вибір між Cloud-based architecture та Event-driven serverless architecture

Cloud-based architecture (архітектура на основі хмарних технологій) може бути гарним підходом до розробки та розгортання маленьких застосунків з ключовими перевагами.

Маленьким застосункам зазвичай не потрібно інвестувати в власну інфраструктуру та обладнання. Замість цього, ви можете використовувати хмарні послуги за оплату за використання, що дозволяє зменшити початкові витрати.

Хмарні платформи надають можливість легко збільшувати або зменшувати ресурси в залежності від змінних потреб вашого застосунку. Це дозволяє підтримувати високий рівень доступності та продуктивності застосунку.

Застосунки, розгорнуті в хмарному середовищі, можуть бути доступні з будь-якого місця та на будь-якому пристрої, якщо є Інтернет. Це робить їх ідеальними для сучасного світу, де користувачі можуть бути розповсюджені по всьому світу.

Хмарні платформи надають інструменти для автоматизації розгортання, моніторингу та керування застосунками. Це спрощує адміністрування та забезпечує автоматичну масштабованість.

Багато хмарних постачальників забезпечують високий рівень безпеки, включаючи захист даних, мережеву безпеку та ідентифікацію користувачів. Це важливо для захисту інформації в маленьких застосунках.

Більшість хмарних платформ пропонують модель оплати за використання, що дозволяє зменшити витрати на обладнання та інфраструктуру і платити лише за реальне використання ресурсів.

Хмарні платформи дозволяють швидко розгортати та масштабувати застосунки, що важливо для невеликих компаній, які хочуть надавати свої послуги швидко та ефективно.

Постачальники хмарних послуг відповідають за оновлення та підтримку інфраструктури, що звільняє розробників від цього завдання і дозволяє їм зосередитися на розробці застосунку.

Загалом, Cloud-based architecture може забезпечити маленьким застосункам доступ до потужних інфраструктурних можливостей без необхідності великих витрат на інфраструктуру та обслуговування.

3.3 Поєднання Cloud-based architecture та монолітної архвтектури

Поєднання монолітної архітектури з Cloud-based architecture може бути корисним для певних проектів, особливо якщо ви хочете скористатися перевагами хмарних обчислень, але маєте існуючий монолітний додаток, який нам потрібно масштабувати або робити більш доступним.

Використання контейнерів, такі як Docker, для упаковки монолітного додатку та його залежностей. Це дозволяє створити портативну та ізольовану середу для вашого додатку, що полегшує розгортання та масштабування в хмарі.

Зберігати дані монолітного додатку у хмарних сховищах, таких як хмарні бази даних чи об'єктні сховища. Це може покращити доступність та забезпечити резервне копіювання даних.

Замість того, щоб самостійно виконувати всі функції, використовуйте хмарні послуги для певних компонентів вашого додатку. Наприклад, ви можете використовувати хмарні послуги для автентифікації, кешування, моніторингу тощо. Відзначемо додаткові елементи які покращать роботу подібної архітектури:

- CDN (Content Delivery Network) та кешування для покращення продуктивності та швидкості вашого монолітного додатку, особливо для статичного контенту;
- інфраструктурні сервіси хмари, такі як масштабовані сервери, балансувальники навантаження та автоматизація, для оптимізації роботи вашого монолітного додатку в хмарі;
- хмарні інструменти для моніторингу та логування, щоб відстежувати та аналізувати роботу вашого монолітного додатку в реальному часі;
- хмарні інструменти автоматизації для розгортання, масштабування та керування інфраструктурою вашого монолітного додатку.

Загалом – основна ідея полягає в тому, щоб використовувати хмарні можливості для покращення продуктивності та доступності вашого монолітного додатку, не обов'язково переходячи до повноцінної мікросервісної архітектури.

4 ПРОВЕДЕННЯ ЕКСПЕРЕМЕНТУ

Надалі нами буде обрано два типи архітектур які частіше використовуються у не великих додатках ніж інші та є доволі не закладними в реалізації особливо на початкових етапах розробки, це стане у нагоді під час тестування їх продуктивності, надалі нами буде проаналізовано ці архітектури, виявлено ключові компоненти, реалізовано програмно та протестовано із використанням однієї бази даних та одного набору даних, фактично тетування буде проведено у стандартизованій системі координат.

4.1 Вибір архітектур для порівняння

Нами було обрано Монолітні та багатошарові (n-шарові) архітектури з наступних причин: такі архітектури залишаються популярними варіантами для веб-додатків, особливо у випадках, коли пріоритетними є простота, згуртованість та швидкість розробки.

Монолітна архітектура характеризується однорівневими програмними застосунками, у яких код користувацького інтерфейсу та доступу до даних об'єднані в одній програмі з однієї платформи. Ця архітектура традиційно вважається привабливою через свою простоту та легкість розгортання, що робить її особливо підходящою для малих застосунків або проектів з обмеженими ресурсами. Головна перевага полягає в її простоті та в тому, що розробникам легше управляти та розгортати монолітний застосунок. Проте, зі зростанням складності застосунку, монолітний підхід може стати обмежувальним через труднощі у масштабуванні та оновленні застосунку без впливу на весь системний блок.

Багатошарова архітектура, часто називана n-шаровою архітектурою, передбачає розділення застосунків на окремі шари, такі як шар презентації, бізнес-логіки та доступу до даних. Ця архітектура покращує модульність та дозволяє розробникам управляти та оновлювати кожен шар незалежно, що робить застосунок легшим для обслуговування та масштабування. Цей підхід цінується за свою здатність організовувати структури програмування таким чином, що кожен

шар може зосередитись на своїй специфічній ролі, тим самим підвищуючи можливості обслуговування та масштабування застосунків.

Обидва стилі архітектури особливо важливі в сценаріях, коли команди розробників потребують балансу між складністю та вимогами до продуктивності. Монолітні архітектури часто хвалять за їх прямоту, роблячи їх підходящими для менш складних або менших проектів. З іншого боку, багатошарові архітектури та монолітні архітектури залишаються популярними варіантами для веб-застосунків, особливо коли простота розробки та розгортання є критично важливими.

Далі у таблиці 4.1 наведемо аналіз того чому інші архітектури не підійдуть для порівняння.

Таблиця 4.1 – Причини неефективності архітектур

Архітектура	Компоненти\Механізми	Причина відмови
Мікросервісна	Розподілені бази даних	Складність управління, потребує більше ресурсів для синхронізації даних.
	Міжсервісне зв'язування	Збільшує затримки та складність мережі, може перевантажити маленькі системи.
	Оркестрація контейнерів	Додаткова складність та ресурси, не виправдані для маленьких додатків.
Хмарна/Clean	Сервісні шлюзи	Для маленьких додатків може бути зайвою, оскільки вони можуть використовувати менш складні рішення без додаткових рівнів.
	Використання кешування на краю мережі (Edge Caching)	Збільшує складність та може не бути виправданим для додатків з малим трафіком.
	Широка автоматизація тестування і CI/CD	Висока вартість та складність впровадження можуть перевищувати потреби малого застосунку.

4.2 Багатошарова архітектура у невеликих веб-додатках

Багатошарова архітектура може бути ефективним рішенням для невеликих застосунків з кількох причин:

- модульність та гнучкість: багатошарова структура дозволяє розділити застосунок на окремі шари (презентація, бізнес-логіка, доступ до даних), кожен з яких може бути розроблений і тестований незалежно. це полегшує управління кодом і може спростити масштабування або оновлення частин застосунку з часом;
- легше утримання та оновлення: оскільки логіка чітко розділена між шарами, зміни в одному шарі часто не впливають на інші шари. наприклад, зміни в базі даних можуть не вимагати змін у користувацькому інтерфейсі. це може значно спростити утримання та оновлення застосунку;
- покращене тестування: можливість тестувати шари незалежно один від одного дозволяє більш ефективно використовувати автоматичне тестування. це також допомагає швидше виявляти помилки і забезпечувати вищу якість програмного продукту;
- відокремлення відповідальностей: кожен шар фокусується на своєму аспекті програми. наприклад, шар доступу до даних обробляє всю взаємодію з базою даних, тоді як бізнес-логіка шар відповідає за обробку даних. це дозволяє розробникам спеціалізуватися та бути більш продуктивними у своїй роботі;
- масштабованість: незважаючи на те, що масштабування монолітних застосунків може бути складнішим, багатошарова архітектура може пропонувати певний рівень масштабованості в межах кожного шару. наприклад, можна масштабувати лише базу даних або серверну частину, не впливаючи на інші компоненти системи;
- відповідність кращим практикам: використання багатошарової архітектури може допомогти застосувати кращі практики проектування та

розробки, такі як принципи solid і dry (не повторюй себе), що може підвищити загальну якість та надійність застосунку.

4.3 Проектування програмного застосунку

Нами з початку була розроблена та взятя як приклад до багатошарової архітектури – система ,файлова стркуктура якої наведена на на рисунку 4.1.

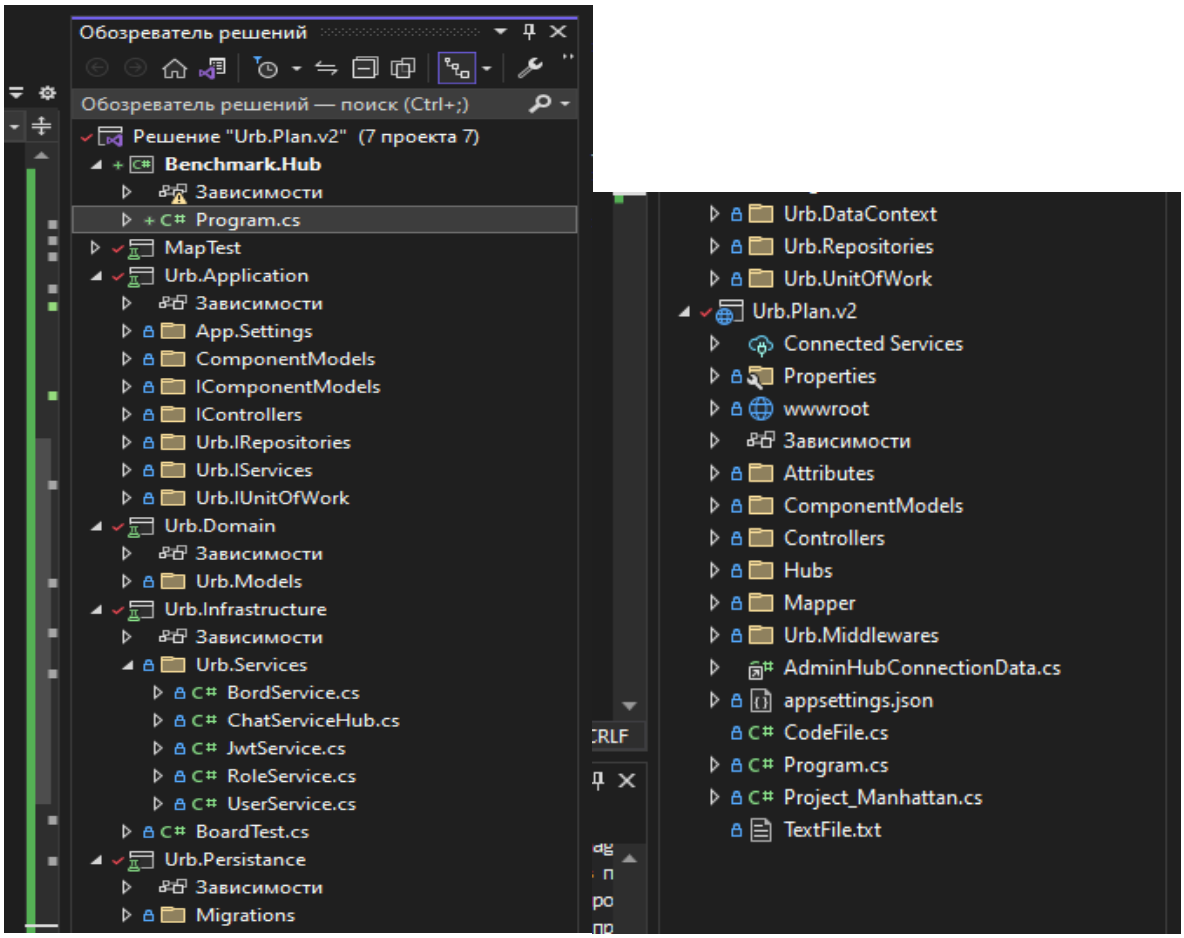

Рисунок 4.1 – Файлова система багатошарового застосунку

Детально розберемо структуру проекту для повного розуміння його складності та відповідності до багатошарової архітектури:

– benchmarkhub: проект не відноситься до структури адже за допомогою нього ми будемо тестувати два варіанти архітектури;

– program.cs: вхідна точка програми. відповідає за конфігурацію та запуск застосунку бенчмаркінгу;

- urb.application: цей проект, відповідає за презентаційний шар і містить логіку для взаємодії з користувачем;
- app.settings: конфігураційні налаштування застосунку;
- componentmodels / icomponentmodels: моделі даних, які використовуються для передачі даних між шарами;
- controllers: класи контролерів, які обробляють вхідні http запити;
- repositories: компоненти для доступу до даних;
- services / unitofwork: сервіси для виконання бізнес-логіки та управління транзакціями даних;
- urb.domain: цей проект містить основні класи домену, які представляють бізнес-модель;
- models: класи, що представляють сутності домену;
- urb.infrastructure: містить логіку, пов'язану з нижчим рівнем доступу до ресурсів, таких як бази даних, файлова система тощо;
- services: включає сервіси для взаємодії з базою даних, такі як boardservice, chatservice та інші;
- persistence: управління збереженням даних, зокрема через migrations та repositories;
- unitofwork: шаблон, який згруповує транзакції даних;
- urb.plan.v2: це основна вхідна точка застосунку;
- properties: налаштування проекту;
- wwwroot: каталог для статичних файлів, таких як html, css, javascript;
- attributes: класи атрибутів, які можуть бути використані для метаданих або контролю поведінки класів;
- componentmodels / controllers / hubs / mapper: компоненти для різних аспектів застосунку, включаючи мапінг даних та веб-сокети (hubs);
- urb.persistence: цей компонент, який знаходиться в проекті urb.infrastructure, є ключовим для роботи з даними. він відповідає за взаємодію з базою даних та управлінням даними;

– migrations: класи міграцій використовуються для версіонування бази даних та управління її змінами через код. це дозволяє безпечно вносити зміни в структуру бази даних в процесі розробки;
– repositories: репозиторії слугують шаром абстракції між бізнес-логікою застосунку та операціями доступу до даних. вони дозволяють ізолювати і оптимізувати запити до джерел даних;
– urb.datacontext: це клас контексту, що використовується entity framework для керування сесією з базою даних, включаючи запити, вставки, оновлення та видалення даних.

На малюнку 4.2 наведено загальний вигляд архітектури нашої системи.

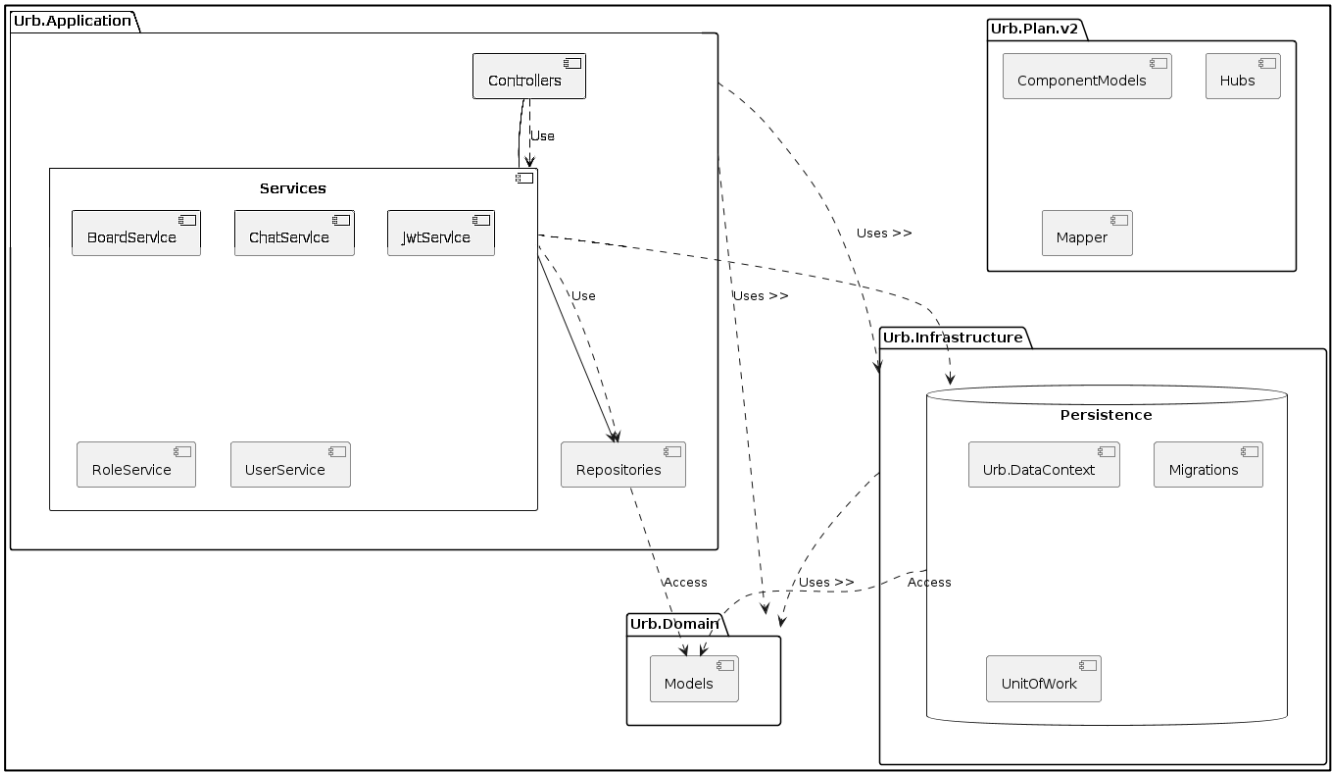

Рисунок 4.2 – Візуальна структура багатошарового зстосунку

4.4 Архітектура моноліту

Далі на основі наданої багатошарової архітектури – розробимо подібну за функціоналом але монолітну, тобто всі ключові елементи зберігутся.

Перехід від багатошарової архітектури до монолітної архітектури змусив до відмови від деяких елементів, які є зайвими або неефективними в контексті

монолітної структури. Наведемо декілька компонентів та механізмів, які оптимізуються або видаляються в монолітних системах:

Роздільні контексти баз даних (DataContexts): в монолітній архітектурі зазвичай достатньо одного контексту бази даних, оскільки всі модулі працюють в одному середовищі, і немає необхідності в роздільних контекстах для кожного мікросервісу.

Внутрішні API-шлюзи: в мікросервісних архітектурах часто використовуються API-шлюзи для маршрутизації запитів між сервісами, але в моноліті це не потрібно, оскільки вся функціональність зосереджена в одному застосунку.

Розширені механізми для внутрішнього зв'язування (Service Mesh): технології, як Istio або Linkerd, які використовуються для управління зв'язуванням і безпекою між мікросервісами, можуть бути надмірними для монолітної архітектури.

Комплексні патерни міжсервісного зв'язку: як очереді повідомлень або шини подій, що використовуються для асинхронного зв'язку між мікросервісами, часто стають непотрібними в моноліті, де зв'язок відбувається внутрішньо.

Декілька екземплярів для розгортання: у мікросервісах кожен сервіс може бути незалежно розгорнутий і масштабований, але в моноліті весь застосунок розгортається як єдиний блок, що зменшує складність управління версіями і розгортання

Декілька баз даних або схем баз даних: мікросервіси часто використовують окремі бази даних для забезпечення незалежності, але в моноліті загальна база даних може покращувати продуктивність і спрощувати управління даними.

Тож на рисунку 4.3 наведено змінену файлову структуру

Далі розглянемо детальніше усі складові компоненти:

- properties: зберігає конфігураційні файл налаштувань проекту;
- connected services: ймовірно містить інтеграції з зовнішніми сервісами, які застосунок використовує;

- wwwroot: каталог для статичних файлів, як html, css і javascript, які використовуються на клієнтській стороні;
- controllers: контролери, які обробляють вхідні запити http і керують потоком даних між користувацьким інтерфейсом і моделями даних;
- datacontext: використовується для конфігурації та взаємодії з базою даних через entity framework або інші orm;
- hubs: використовуються для веб-сокет з'єднань, зокрема використання signalr для реалітайм функціоналу;
- models: визначає структуру даних і бізнес-логіку. моделі часто використовуються для взаємодії з базою даних;
- pages: зберігає razor pages файли, якщо використовується цей підхід в asp.net для побудови сторінок;
- persistence: шар, який займається збереженням даних і може включати класи для управління базою даних, такі як репозиторії;
- repositories: класи репозиторіїв, які абстрагують доступ до даних, використовуючи патерн репозиторій;
- services: включає сервіси, які виконують бізнес-логіку, які можуть бути викликані контролерами;
- views: каталог, який містить razor views для відображення вмісту, якщо використовується mvc патерн;
- appsettings.json: конфігураційний файл, який містить налаштування, такі як рядки з'єднань, налаштування зовнішніх сервісів та інші параметри конфігурації;
- program.cs: вхідна точка застосунку, де налаштовується веб-хост та стартує застосунок.
- Ця структура в монолітному проекті дозволяє легко керувати всіма компонентами системи з одного місця, спрощуючи розробку та деплоймент.

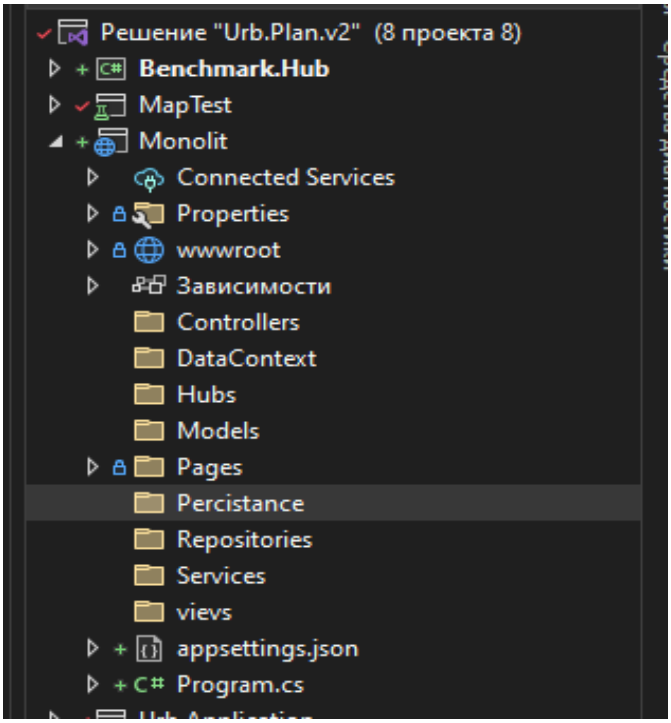

Рисунок 4.3 – Структура монолітного додатку

На рисунку 4.4 зображено загальний вигляд нашого моноліту(скорочений).

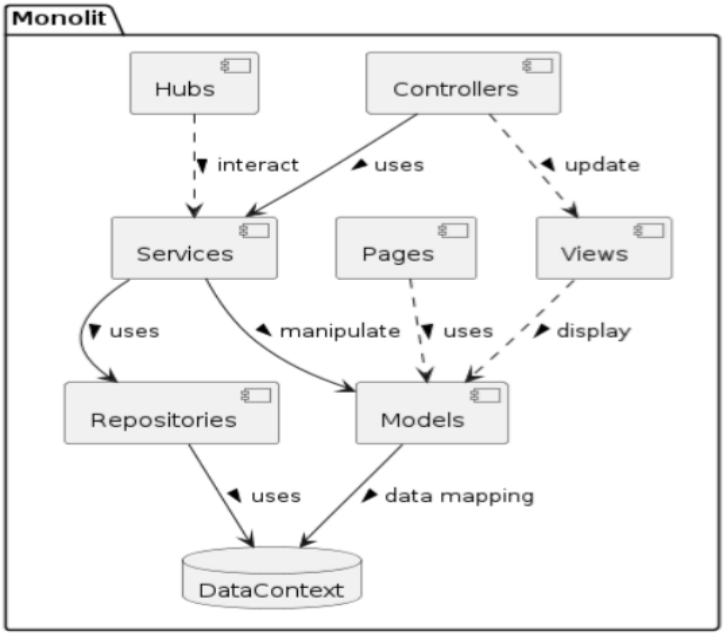

Рисунок 4.4 – Візуальна структура монолітного зстосунку

4.5 Проведення тестування

Вибір BenchmarkDotNet для тестування та порівняння наших додатків обумовлений кількома ключовими факторами, які важливі для ефективності та

точності бенчмаркінгу в середовищі .NET. BenchmarkDotNet є високоспеціалізованим інструментом, розробленим для забезпечення детального і точного аналізу продуктивності коду на платформах .NET, що робить його ідеальним для наших потреб.

Цей інструмент забезпечує вимірювання з мінімальним впливом на продуктивність, дозволяючи отримати реалістичні показники продуктивності, що є критично важливим при порівнянні різних підходів або архітектурних рішень в рамках наших проектів.

BenchmarkDotNet також пропонує гнучкість у налаштуванні тестів, що дозволяє детально контролювати сценарії тестування, вибирати специфічні параметри і налаштування виконання, що є незамінним для науково обґрунтованих експериментів. Інструмент здатен автоматично управляти процесом тестування, від керування процесами вимірювання та врахування зовнішніх впливів до збору та аналізу результатів. Окрім того, BenchmarkDotNet допомагає обробляти типові проблеми, пов'язані з точністю бенчмаркінгу, такі як варіації у виконанні через збір сміття в CLR або через різні стратегії JIT компіляції, що гарантує, що наші висновки базуються на надійних даних.

Крім технічної сторони, використання BenchmarkDotNet дозволяє нам підтримувати високий рівень стандартизації в тестуванні продуктивності наших проектів.

Це важливо для забезпечення консистентності та порівняльності результатів між різними командами та проектами, що є фундаментальним у нашому прагненні до постійного покращення та інновацій в розробці. Використання цього інструменту в кінцевому підсумку допомагає нам забезпечувати більш високу якість програмного забезпечення, оптимізуючи продуктивність і водночас враховуючи ресурси та обмеження наших систем.

Далі ми розмістили проект бенчмарку у нашому загальному рішенні(див. рис. 4.5).

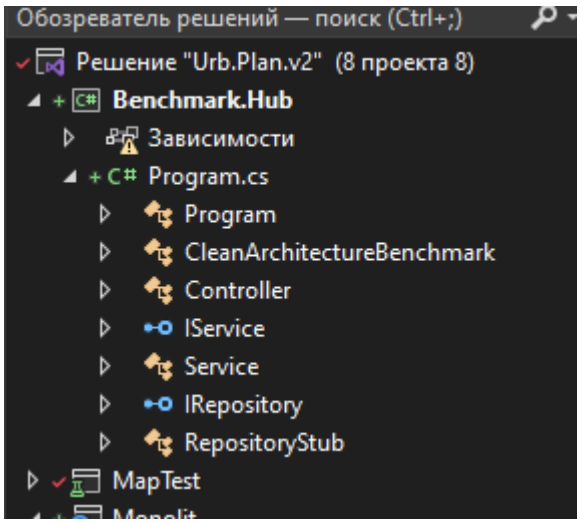

Рисунок 4.5 – Структура та залежності бенчмарку

Розташування проекту бенчмаркінгу у загальному рішенні має кілька важливих переваг, які сприяють ефективності розробки та підтримці застосунків у вашому портфоліо:

- спрощення доступу та спільного використання коду: коли проект бенчмаркінгу знаходиться у загальному рішенні разом з іншими проектами, всі розробники мають доступ до нього з одного місця. це полегшує спільне використання коду і ресурсів між різними проектами та командами, дозволяючи використовувати загальні бібліотеки чи утиліти для бенчмаркінгу без необхідності дублювання роботи або створення зайвих залежностей;
- консистентність тестування: розміщення проекту бенчмаркінгу в одному рішенні з основними застосунками забезпечує консистентність методологій тестування і метрик, використовуваних у різних проектах. це дозволяє з легкістю порівнювати продуктивність різних компонентів і модулів, забезпечуючи, що всі вони відповідають встановленим стандартам продуктивності;
- сприяння співпраці: централізація проекту бенчмаркінгу сприяє співпраці між командами, які працюють над різними частинами рішення. це зміцнює розуміння між командами щодо впливу різних архітектурних і кодових рішень на загальну продуктивність застосунків.

Нижче наведемо код для бенчмарку багатошарового застосунку:

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using Microsoft.Extensions.DependencyInjection;
using Urb.Infrastructure.Urb.Services;
using Urb.Persistance.Urb.DataConext;
using Microsoft.EntityFrameworkCore;
using Urb.Application.Urb.IServices;
using Microsoft.Extensions.Configuration;

public class Program
{
    public static void Main(string[] args)
    {
        BenchmarkRunner.Run<BordServiceBenchmarks>();}}
using BenchmarkDotNet.Attributes;
using Microsoft.EntityFrameworkCore;
using Urb.Persistance.Urb.DataConext;
using Urb.Infrastructure.Urb.Services;
using System.Threading.Tasks;
namespace Urb.Benchmark
{
    public class UserRegistrationBenchmark
    {
        private IUserService _userService;
        private UserTokenDataContext _context;

        [GlobalSetup]
        public void Setup()
        {
            var optionsBuilder = new
DbContextOptionsBuilder<UserTokenDataContext>();
            optionsBuilder.UseSqlServer("your-connection-string-here");

            _context = new UserTokenDataContext(optionsBuilder.Options);
            _userService = new UserService(_context, /* Add any other
dependencies */);}
        [Benchmark]
        public async Task RegisterUser()
        {
            var userRegistrationModel = new UserRegisterModel
            {
                Email = "test@example.com",
                Password = "Test123!"
            };
            await _userService.RegisterUser(userRegistrationModel);
        }
        [GlobalCleanup]
        public void Cleanup()
        {
            _context.Dispose();}}}
```

Метод RegisterUserBenchmark тестує продуктивність методу Register в UserService. Цей метод, ймовірно, займається реєстрацією користувачів, що включає валідацію даних, шифрування паролів і запис даних до бази даних. Бенчмарк вимірює, наскільки швидко цей процес виконується в ізольованому тестовому середовищі, що допомагає виявити можливі "вузькі місця" у продуктивності методу Register.

Далі наведемо код до бенчмарку нашої монолітної архітектури із абсолютно таким же запитом до бази даних:

```
using BenchmarkDotNet.Attributes;
using BenchmarkDotNet.Running;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using MyMonolithApp.Services;
using MyMonolithApp.Persistence;
using MyMonolithApp.Models;
public class Program
{
    public static void Main(string[] args)
    {
        BenchmarkRunner.Run<UserServiceBenchmark>();}}
public class UserServiceBenchmark
{
    private IUserService _userService;
    private ServiceProvider _serviceProvider;

    [GlobalSetup]
    public void Setup()
    {
        var services = new ServiceCollection();

        services.AddDbContext<AppDbContext>(options =>
            options.UseSqlServer("YourConnectionStringHere"));
        services.AddScoped<IUserService, UserService>(); UserService
виконує операції з користувачами
        services.AddScoped<IRepository<User>, UserRepository>();
UserRepository для взаємодії з базою даних користувачів

        _serviceProvider = services.BuildServiceProvider(true);
        _userService = _serviceProvider.GetService<IUserService>();}
    [Benchmark]
    public void RegisterUserBenchmark()
    {
        var result = _userService.Register(new User { Email =
"test@example.com", Password = "SecurePassword123" });
        // Припускаємо, що Register повертає об'єкт результату або
асинхронний Task}
    [GlobalCleanup]
    public void Cleanup() {
        _serviceProvider?.Dispose();}}
```

Розглянемо детальніше:

- program.main: точка входу, де ініціюється запуск бенчмарку;
- userservicebenchmark: клас, який визначає бенчмарк;
- setup: налаштування сервісів і контексту бази даних, необхідних для тестування;
- registeruserbenchmark: метод, який тестує реєстрацію користувачів у системі, вимірюючи продуктивність і ефективність сервісу;
- cleanup: завершення роботи, знищення провайдера послуг для звільнення ресурсів.

5 АНАЛІЗ РЕЗУЛЬТАТІВ

Після запуску бенчмарку у Release режимі ми отримели результати швидкодії обох варіантів проходження запиту у архітектурах, на рисунку 5.1 наведено результат відпрацювання багатошарової архітектури, на рисунку 5.2 та 5.3 – наведено результати відпрацювання монолітної архітектури та процесу виконання відповідно.

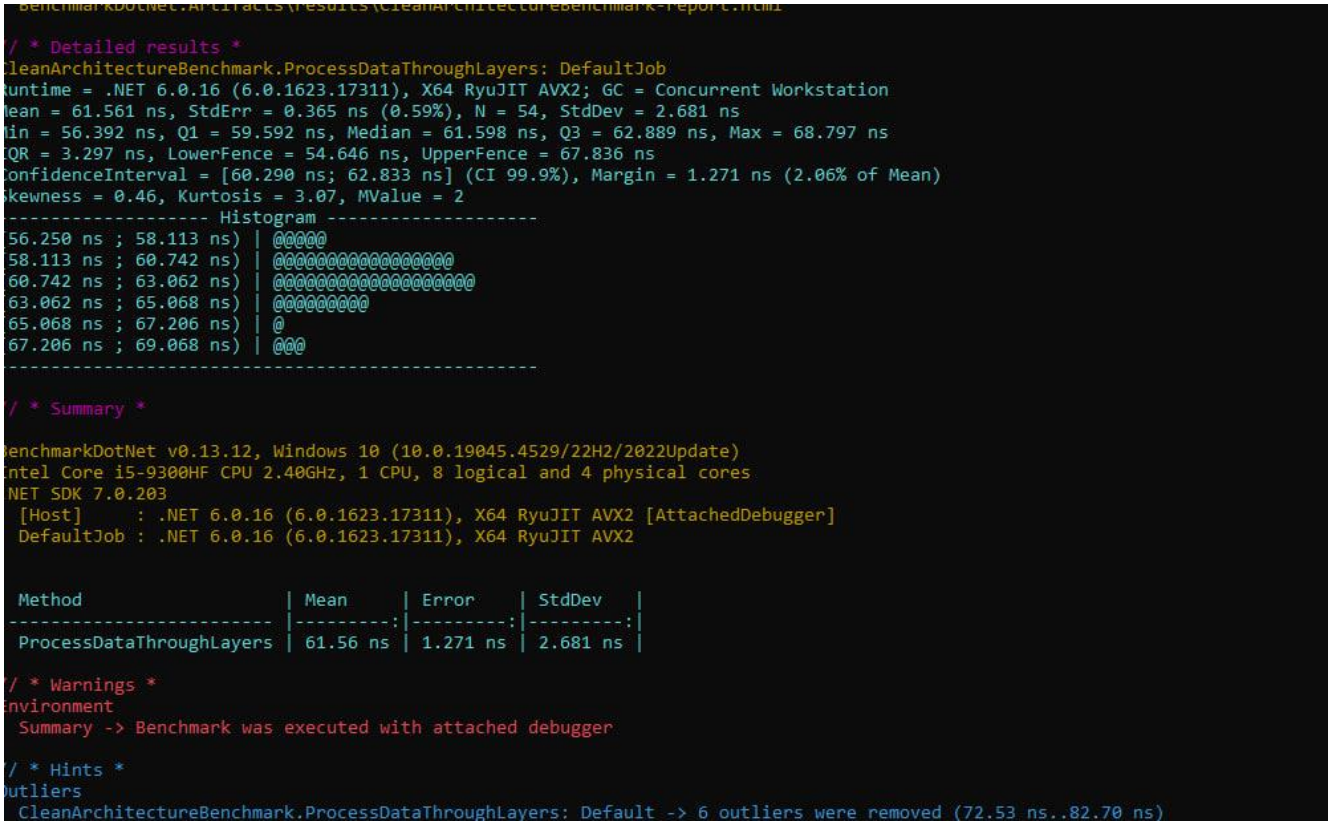

Рисунок 5.1 – Результат бенчмарку на багатошаровій архітектурі

Розглянемо приклад метрик:

- mean (середнє значення): середнє часове значення для виконання даної операції. наприклад, в результаті бенчмарку, який ви надали, середнє значення складає 61.56 наносекунд;
- error (похибка): ця метрика показує стандартну помилку вимірювань, що дає уявлення про стабільність ваших бенчмарк-тестів. в вашому випадку похибка складає 1.271 наносекунд;

– stddev (стандартне відхилення): показує, наскільки результати тестів відрізняються від середнього значення. менше стандартне відхилення вказує на більшу консистентність результатів. ваш результат показує стандартне відхилення 2.681 наносекунд.

```
[51.855 ns ; 53.801 ns) | @@@@@@@@@
[53.801 ns ; 55.631 ns) | @@@@
---------------------------------------------------

// * Summary *

BenchmarkDotNet v0.13.12, Windows 10 (10.0.19045.4529/22H2/2022Update)
Intel Core i5-9300HF CPU 2.40GHz, 1 CPU, 8 logical and 4 physical cores
.NET SDK 7.0.203
  [Host]     : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2 [AttachedDebugger]
  DefaultJob : .NET 6.0.16 (6.0.1623.17311), X64 RyuJIT AVX2


| Method                  | Mean     | Error    | StdDev   |
|------------------------ |---------:|---------:|---------:|
| ProcessDataThroughLayers | 53.51 ns | 1.001 ns | 0.836 ns |

// * Warnings *
```

Рисунок 5.2 – Результат бенчмарку на монолітній архітектурі

```
WorkloadResult  35: 8388608 op, 479776000.00 ns, 57.1938 ns/op
WorkloadResult  36: 8388608 op, 521242900.00 ns, 62.1370 ns/op
WorkloadResult  37: 8388608 op, 507885600.00 ns, 60.5447 ns/op
WorkloadResult  38: 8388608 op, 525566300.00 ns, 62.6524 ns/op
WorkloadResult  39: 8388608 op, 539502400.00 ns, 64.3137 ns/op
WorkloadResult  40: 8388608 op, 494560700.00 ns, 58.9562 ns/op
WorkloadResult  41: 8388608 op, 511152300.00 ns, 60.9341 ns/op
WorkloadResult  42: 8388608 op, 515945900.00 ns, 61.5055 ns/op
WorkloadResult  43: 8388608 op, 473047900.00 ns, 56.3917 ns/op
WorkloadResult  44: 8388608 op, 492050300.00 ns, 58.6570 ns/op
WorkloadResult  45: 8388608 op, 481698200.00 ns, 57.4229 ns/op
WorkloadResult  46: 8388608 op, 518000900.00 ns, 61.7505 ns/op
WorkloadResult  47: 8388608 op, 517139600.00 ns, 61.6478 ns/op
WorkloadResult  48: 8388608 op, 486299700.00 ns, 57.9714 ns/op
WorkloadResult  49: 8388608 op, 494811800.00 ns, 58.9862 ns/op
WorkloadResult  50: 8388608 op, 577112800.00 ns, 68.7972 ns/op
WorkloadResult  51: 8388608 op, 520271200.00 ns, 62.0212 ns/op
WorkloadResult  52: 8388608 op, 509069400.00 ns, 60.6858 ns/op
WorkloadResult  53: 8388608 op, 502973000.00 ns, 59.9591 ns/op
WorkloadResult  54: 8388608 op, 522413700.00 ns, 62.2766 ns/op
```

Рисунок 5.3 – Workflow викликів

Результати показують, що монолітна архітектура має менший середній час відповіді для заданого запиту процесування даних. Середній час виконання в монолітній архітектурі складає приблизно 53.51 наносекунди, що є значно швидше

порівняно з іншою архітектурою, де цей час становить 61.56 наносекунди. Такий результат може свідчити про вищу ефективність монолітної архітектури для конкретних типів завдань або сценаріїв, де важливі швидкість обробки та мінімізація затримок.

Це обумовлено кількома факторами, монолітна архітектура, яка інтегрує всі компоненти програми в єдиний процес, здатна забезпечувати швидший доступ до ресурсів та управління даними через зниження затримок, пов'язаних з мережевими запитами між мікросервісами. Всередині одного процесу немає необхідності в серіалізації даних або їх пересиланні між різними службами, що також знижує загальну затримку.

Водночас, монолітна архітектура може мати свої недоліки, особливо коли йдеться про масштабування та управління складними застосунками, але в контексті даного бенчмарку вона виявилася більш продуктивною. Такий результат може слугувати важливим чинником при виборі архітектури для нових проектів або при плануванні оптимізації існуючих систем.

ВИСНОВКИ

У нашому дослідженні ми проаналізували різні архітектурні підходи, зокрема Cloud-Native, мікросервісну, монолітну, та Event-driven serverless архітектури. Cloud-Native підхід забезпечує гнучкість, швидке розгортання, ефективне використання ресурсів та відмінну витривалість, але має складності в управлінні та високі вимоги до безпеки та навичок.

Мікросервісна архітектура пропонує незалежне масштабування та спрощене оновлення, але супроводжується складностями управління та можливими загрозами безпеки. Монолітна архітектура ідеальна для невеликих проектів з простим розгортанням, але обмежена в масштабуванні та оновленні. Event-driven serverless архітектура ефективна для відповіді на події, але може мати обмеження залежно від хмарного провайдера.

На основі аналізу ми визначили, що оптимальним рішенням для нашого проекту буде поєднання монолітної архітектури з Cloud-based architecture. Таке поєднання забезпечує простоту та централізованість монолітного підходу з гнучкістю та масштабованістю хмарних технологій.

Воно дозволяє ефективно використовувати ресурси хмари для масштабування та оптимізації продуктивності, одночасно зберігаючи переваги єдиного кодового базису та спрощеного процесу розробки, характерних для монолітної архітектури.

Також нами було порівняно підходи монолітнох та багатошарової архітектур, нами було написано бенчмарк який заміряв продуктивність, середню швидкість обох варіантів у одній системі координат, та ми з’ясували що монолітна архітектура швидше, має меншу похибку та веде себе більш стабільно під навантаженням.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Аналіз предметної області \ A Guide to Choosing the Right .NET Object Mapper? 2023р., URL: https://learn.microsoft.com/ru-ru/sql/relational-databases/search/full-text-search?view=sql-server-ver16 (дата звернення:05.04.2024).

2. Аналіз існуючих рішень, URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html (дата звернення: 05.04.2024).

3. What is API Governence and How Does it Work?, URL: https://www.mongodb.com/resources/basics/full-text-search#:~:text=Full-text%20search%20involves%20reviewing,PDFs%2C%20and%20more. дата звернення: 10.04.2024).

4. Enhancing Database Text Search., URL: https://medium.com/@havus.it/enhancing-database-search-full-text-search-fts-in-mysql-1bb548f4b9ba (дата звернення: 10.04.2024).

5. API Governence (FTS) in MySQL, URL: https://medium.com/@havus.it/enhancing-database-search-full-text-search-fts-in-mysql-1bb548f4b9ba (дата звернення: 10.04.2024).

6. Аналіз API Governence \ HigLabo.Mapper, Creating Fastest Object Mapper in the World with Expression Tree, 2024р., URL: https://www.codeproject.com/Articles/5275388/HigLabo-Mapper-Creating-Fastest-Object-Mapper-in-t (дата звернення: 10.04.2024).

7. Free Public Data Sets For Analysis, URL: https://www.tableau.com/learn/articles/free-public-data-sets (дата звернення: 17.04.2024).

8. Аналіз reflection ammit \ Comparison of Object Mapper Libraries, Jon Dodd, 2021р., URL: https://dev.to/jdinnovensa/comparison-of-object-mapper-libraries-gm2 (дата звернення: 17.04.2024).

9. FastCompileExpression бібліотека \ Fast Compiler for C# Expression Trees and the lightweight LightExpression alternative.? 2024р., URL: https://github.com/dadhi/FastExpressionCompiler (дата звернення: 17.04.2024).

10. Optimization Factors in Modeling and Testing Hardware and Semiconductor Defects by Dynamic Discrete Event Simulation \ Arabian, J. H., Видавництво: ХНУРЕ, 2009р., URL: https://openarchive.nure.ua/entities/publication/be918fba-8755-4d34-be6d-fc1464089d77 (дата звернення: 05.05.2024).

11. Falatiuk H., Shirokopetleva M., Dudar Z. Investigation of Architecture and Technology Stack for e-Archive System. 2019 IEEE International Scientific-Practical Conference Problems of Infocommunications, Science and Technology (PIC S&T), Kyiv,Ukraine,8-11October2019.2019.,URL: https://doi.org/10.1109/picst47496.2019.9061407 (дата звернення: 05.05.2024).

12. DATA EXCHANGE MODEL IN THE INTERNET OF THINGS CONCEPT / I. Afanasieva et al. Telecommunications and Radio Engineering. 2019. Vol. 78,no. 10.P. 869–878.
URL: https://doi.org/10.1615/telecomradeng.v78.i10.30 (date of access: 05.05.2024).

13. Аналіз результатів бенчмаркіунгу
URL: https://github.com/dotnet/BenchmarkDotNet (date of access: 05.05.2024)