КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
ІМЕНІ ТАРАСА ШЕВЧЕНКА

Факультет Комп'ютерних наук та кібернетики
Кафедра Теорії та технології програмування

Кваліфікаційна робота
на здобуття ступеня бакалавра

за спеціальністю 122 Комп'ютерні науки
на тему:

ПОРІВНЯЛЬНИЙ АНАЛІЗ МОНОЛІТНОЇ ТА МІКРОСЕРВІСНОЇ АРХІТЕКТУР НА ПРИКЛАДІ КЛІЄНТ-СЕРВЕРНОЇ ВЗАЄМОДІЇ

Виконав студент 4-го курсу
Антон ЧЕРНОВ _____

Науковий керівник:
доцент, кандидат фіз.-мат. наук
Віктор ВОЛОХОВ _____

Засвідчую, що в цій роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____

Роботу розглянуто й допущено до захисту на засіданні кафедри теорії та технології програмування
«___» _____ 2023 р., протокол № __
Завідувач кафедри
Микола НІКІТЧЕНКО _____

Київ – 2023 рік

РЕФЕРАТ

Обсяг роботи 49 сторінок, 10 ілюстрацій, 7 джерел посилань, 1 таблиця, 3 додатки.

КЛІЄНТ, МІКРОСЕРВІС, МЕРЕЖА, МОНОЛІТ, ПРОТОКОЛ, СЕРВЕР, API, CONTAINER, DOCKER, IMAGE, KUBERNETES, REST, WEB SOCKET

Об’єкт дослідження – використання монолітної та мікросервісної архітектур для проєктування клієнт-серверного продукту.

Предмет дослідження – використання відмінностей проєктування продукту із використанням монолітної та мікросервісної архітектури.

Мета роботи полягає у виявлення переваг та недоліків клієнт-серверних архітектур, що використовуються для створення програмних продуктів, та їх порівняння.

Методи дослідження – аналіз літературних та електронних джерел, проведення тестування, комп’ютерне моделювання та проєктування.

Було проведено аналіз наявних технологій для реалізації монолітної та мікросервісної архітектур клієнт-серверного продукту. Розроблено власний продукт для аналізу продуктивності прийнятих архітектурних рішень. Проведено аналіз засобів навантажувального тестування. Проведено аналіз продуктивності монолітної та мікросервісної архітектур.

Результат роботи – проведено аналіз та зроблено висновки про використання мікросервісної та монолітної архітектури для проєктування клієнт-серверного продукту.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧЕННЯ

API – Application Programming Interface, інтерфейс прикладного програмування.

CAP – Consistency, Availability, Partition tolerance, узгодженість, доступність, толерантність до розділів.

DOM – Document Object Model, об'єктна модель документа.

HTML – HyperText Markup Language, мова розмітки гіпертексту.

HTTP – HyperText Transfer Protocol, протокол передачі гіпертексту.

HTTPS – HyperText Transfer Protocol Secure, захищений протокол передачі гіпертексту.

IP – Internet Protocol, Інтернет-протокол.

P2P – Peer-to-peer.

REST – Representational State Transfer, передача представницького стану.

SSR – Server-side rendering, рендеринг на стороні сервера.

TCP – Transmission Control Protocol, протокол управління передачею.

URL – Uniform Resource Locator, уніфікований покажчик ресурсів.

ПЗ – Програмне забезпечення.

СУБД – Система управління базами даних.

ВСТУП

Оцінка сучасного стану об’єкта розробки. Наразі доволі велика частина ПЗ використовує клієнт-серверну технологію в основі своєї роботи. Двома найпопулярнішими архітектурами для її реалізації є монолітна та мікросервісна. Обидві ці архітектури широко вживані, тому є актуальними та застосовними.

Актуальність роботи та підстави для її виконання. Робота по аналізу та порівняння цих двох архітектур є актуальною, адже при проєктуванні нових продуктів, або у процесі оптимізації уже створених, потрібно обирати між монолітом та мікросвервісом, а вони мають у собі як переваги, так і недоліки, тому для прийняття більш оптимальних рішень потрібно добре розумітися не лише в ідеї продукту, а й в архітектурі ПЗ.

Мета й завдання роботи. Метою роботи є аналіз та порівняння мікросервісної та монолітної архітектур. Для її досягнення було сформовано такі завдання :

1. Розглянути актуальний стан монолітної та мікросервісної архітектур та проаналізувати основні проблеми.
2. Розробити проєктування власного продукту та обрати інструментарій для розробки.
3. Розробити продукт, з використанням мікросервісної та монолітної архітектур.
4. Провести тестування продуктивності.
5. Отримання результатів тестування, та їх аналіз, формулювання висновків.

Методи дослідження. Методами дослідження є аналіз літературних та електронних джерел, проведення тестування, комп’ютерне моделювання та проєктування.

Можливі сфери застосування. Можливою сферою застосування результатів даної роботи є сфера розробки програмного забезпечення. З використанням даної роботи може процес проєктування програмного продукту може бути значно полегшений, що дає змогу бізнесу розробляти продукт швидше, та із використанням меншої кількості ресурсів.

РОЗДІЛ 1. ОСНОВИ ТЕХНОЛОГІЇ "КЛІЄНТ-СЕРВЕР"

1.1. Клієнт-серверна архітектура

Для того, щоб ми могли розглянути принципи роботи архітектури «клієнт-сервер» нам необхідно ввести визначення для кращого розуміння. Отож:

- Сервер – це програма, яка очікує на запити зі сторони програми-клієнта та відповідає на них.
- Клієнт – це програма, з якою взаємодіє користувач, внаслідок чого виникає необхідність в отриманні додаткових даних, які вона отримує за допомогою запитів до сервера.

З вищесказаного стає зрозуміло, що клієнт та сервер виконують кардинально різні задачі, а тому потрібно чітко визначити їх ролі. Роль сервера – централізований захист інформації, її попередня обробка та розповсюдження клієнтам. Для цього на сервері проводиться робота із базами даних, попередня обробка інформації та у деяких випадках генерація основи візуальної частини клієнтської частини продукту. Останній пункт також називають SSR, та використовують для зменшення навантаження на клієнтську частину у Web додатках. Усе це необхідно для того, щоб у кожного користувача була можливість отримувати актуальну інформацію та за допомогою використання щонайменшої кількості ресурсів відобразити її саме у клієнтській частині продукту. Таким чином клієнт – це частина системи, яка по суті своїй є користувацьким інтерфейсом для взаємодії з даними, що розміщені на серверній частині.

Проте не варто забувати, що оскільки клієнт та сервер незалежні, доступу до ресурсів один одного у них немає, саме тому потрібно додати 3 складову до цієї моделі — мережу. Мережа, в такому випадку, виступає посередником між програмами, що використовується для комунікації.

Реалізація цієї архітектури зараз поділяють на 2 основних види:

1. Дворівневий – реалізація передбачає наявність однієї програми-клієнта та однієї програми-сервера.
2. Багаторівневий – різновид архітектури «клієнт-сервер», де функція обробки інформації винесена на кілька окремих серверів.

Яку саме структуру обирати для конкретного продукту — дискусійне питання, адже на проєктування та створення багаторівневої архітектури потрібно витратити більше ресурсів, насамперед час розробників та архітекторів, проте вона дозволяє краще масштабувати продукт в майбутньому.

1.2. P2P архітектура

Розглянемо й іншу архітектуру, що наразі конкурує із клієнт-серверною.

P2P архітектура характеризується тим, що всі вузли в мережі можуть виконувати як клієнтські, так і серверні функції. Кожен пір може взаємодіяти безпосередньо з іншими пірами, без потреби в централізованому сервері. Це означає, що кожен вузол має можливість надавати ресурси та послуги іншим вузлам в мережі, а також запитувати та отримувати ресурси від інших вузлів.

З іншого боку, в архітектурі Client-Server існує чітка рольова розбивка між клієнтами та серверами. Клієнти запитують послуги або ресурси у серверів, які надають відповідні відповіді або результати. Сервери виконують функцію централізованого розподілу та обробки ресурсів, а також забезпечують безпеку та надійність мережевої взаємодії.

Однією з основних переваг P2P архітектури є висока резистентність до відмов, оскільки немає одного центрального сервера, від якого залежить вся мережа. Кожен пір може продовжувати свою роботу, навіть якщо інші вузли відмовляють. Також P2P архітектура зазвичай дозволяє ефективно використовувати ресурси мережі, оскільки вузли можуть спільно надавати ресурси.

У Client-Server архітектурі централізовані сервери забезпечують керування та розподіл ресурсів, що дозволяє забезпечити високу масштабованість та контроль над доступом до ресурсів. Ця архітектура також спрощує управління безпекою та надійністю, оскільки ці аспекти можуть бути централізовано контрольовані на рівні серверів.

Обираючи між P2P та Client-Server архітектурами, необхідно враховувати конкретні вимоги проєкту, такі як масштабованість, безпека, надійність та управління ресурсами. Кожна архітектура має свої переваги та обмеження, і правильний вибір залежить від конкретного сценарію використання та потреб бізнесу.

1.3. Протоколи та технології мережевого зв'язку

Для реалізації взаємодії клієнт-сервер необхідно визначитись за допомогою яких протоколів та технологій зв’язку буде відбуватись комунікація. Технології дозволяють встановити зв’язок між програмами та відповідають за обмін інформацією, протоколи у свою ж чергу — це певний набір правил, які забезпечують правильність передачі даних. Розглянемо основні протоколи:

- TCP/IP - основа протоколів для зв’язку у WEB додатках. Поділяється на 4 рівні: прикладний, транспортний, міжрівневий та рівень доступу до середовища передачі.
- HTTP/HTTPS - HTTP(HyperText Transfer Protocol) - протокол прикладного рівня, що спочатку був створений для передачі гіпертекстових документів(HTML), але пізніше набув можливості передавати довільні дані. HTTPS(HyperText Transfer Protocol Secure) - надбудова над протоколом HTTP, що дозволяє при передачі даних шифрувати їх, що забезпечує вищий рівень безпеки.
- WebSocket - протокол зв’язку, побудований поверх TCP з’єднання, що дозволяє обмінюватись даними з браузера, де запущено

клієнтський продукт, до вебсервера, використовуючи постійне з’єднання. Це дозволяє реалізувати концепцію реактивного програмування, де програма повинна виконувати певні дії, як реакцію на дії зовні. Наприклад: Сервер не буде виконувати операцію редагування сутності у базі даних до того, як програма-клієнт не відправляє відповідне повідомлення з потрібним набором даних.

Також в цьому пункті хотілося б розглянути поняття API та REST.

API - (Application Programming Interface) це контракт, який представляє програма. Це свого роду договір, де програма показує які вона може виконувати функції, та що для цього потрібно зробити (які дані передати, яким чином).

Rest - це інтерфейс для управління інформацією без додаткових внутрішніх шарів. Це забезпечується тим, що кожна дія чітко визначена за допомогою URL, куди повинен прийти запит. Відсутність внутрішніх шарів говорить про те, що дані не змінюються при відправленні, тобто сервер буде отримувати ті самі дані, що йому відправив клієнт.

1.4. Типи серверних програм

Серверні програми можна поділити на декілька видів, таким чином буде простіше пояснити різницю в архітектурі клієнт-сервер. Розглянемо кілька видів серверних програм

1. Вебсервер — відповідає за отримання даних із бази даних та їх обробку.
2. Сервер бази даних — відповідає за зберігання та редагування даних, що використовуються у додатку.
3. Проксі сервер — відповідає за переадресування запитів до потрібних серверів.

Ці типи можуть бути поєднанні, для прикладу: при використанні nginx - створюється сервер, що приймає дані з клієнтських програм та перенаправляє

запити до вебсерверів, що повинні виконувати необхідні дії, проте оскільки він і сам відправляє відповіді до сервера, його можна також віднести до вебсервера.

З перелічених типів можемо бачити, що більшість WEB застосунків використовують, по меншій мірі, 3 програми: вебсервер, клієнт та сервер бази даних. Це може викликати запитання чому тоді архітектура саме дворівнева, адже використовуються 3 рівні. Це обумовлено тим, що сервер бази даних не має бізнес-логіки, а отже архітектура розрізняє лише ті складові, що розв'язувати конкретні задачі проєкт, без зберігання даних. Таким чином багаторівнева архітектура використовує ще один рівень — проксі сервер, який перенаправляє запити до конкретних серверних програм — мікросервісів.

1.5. Основні проблеми технології 'клієнт-сервер'

Для подальшого порівняння видів архітектури клієнт-сервер, необхідно визначити її основні проблеми, і тоді ми зможемо які проблеми можуть бути вирішені, і як, а які залишаться. Отже, основними проблемами є:

- Масштабування — коли кількість клієнтів зростає, можуть виникати проблеми з масштабуванням серверної інфраструктури для задоволення наростального навантаження.
- Надійність — якщо сервер виходить з ладу або втрачає з'єднання, клієнти можуть бути вимушені чекати або отримувати помилки.
- Безпека клієнт-серверні додатки вимагають забезпечення безпеки передачі даних між клієнтом і сервером. Використання шифрування, аутентифікації та авторизації допомагає забезпечити захист інформації в процесі комунікації.

Останні 2 проблеми можуть вирішені за допомогою різних сторонніх рішень, до прикладу: для розв'язання проблеми із надійністю може використовуватись балансування навантаження, зокрема, згаданий вище nginx

дозволяє це зробити, а безпеку зокрема можна вирішити за допомогою використання захищених протоколів, хешування чутливих даних, дотримання ISO/IEC 27001.

Додаткові проблеми технології "клієнт-сервер", які також потрібно врахувати порівнюючи типи архітектур, включають:

- Продуктивність: Залежно від розміру та складності продукту, використання клієнт-серверної архітектури може створювати проблеми з продуктивністю. Недостатня швидкодія комунікації між клієнтом і сервером може призвести до затримок в обробці запитів і відповідей.
- Залежність від мережі: Клієнт-серверна архітектура передбачає постійний зв'язок між клієнтом та сервером через мережу. Це означає, що недоступність мережі або відсутність з'єднання може призвести до недоступності продукту для користувачів.
- Синхронність: У деяких випадках, особливо при великому обсязі даних або складних операціях, синхронний характер взаємодії між клієнтом і сервером може призвести до блокування ресурсів та зниження продуктивності.

Для розв'язання цих проблем можна використовувати різні підходи та технології, такі як:

- Кешування: Використання кешування даних може покращити продуктивність, зменшити навантаження на сервер та знизити залежність від мережі. Кешування може бути реалізовано на рівні сервера або клієнта.
- Асинхронність: Використання асинхронних запитів та відповідей може допомогти уникнути блокування ресурсів та покращити продуктивність програми. Асинхронність може бути досягнута за допомогою асинхронних запитів HTTP або використанням спеціальних протоколів, таких як WebSocket.

У порівнянні між монолітною та мікросервісною архітектурою, ці проблеми можуть мати різну вагомість та способи вирішення залежно від конкретних вимог і особливостей проєкту.

1.6. Монолітна та мікросервісна архітектура

Монолітна архітектура сервера передбачає, що серверна частина продукту — це одна програма, і всі її складові частини працюють одночасно, написані за допомогою однієї мови програмування та фреймворку. Своєю чергою мікросервісна архітектура дозволяє розбити програму на різні за логікою частини, які працюють незалежно одна від одної. Ці частини можуть бути написані з використанням різних технологій, що дозволяє бути більш гнучким в питанні інструментів розробки та розгортаються окремо, що змушує розробників продумати методи комунікації між ними. Для цього часто використовують інші програмі продукти, до прикладу Kafka [1].

Тепер розглянемо як кожна з них розв'язувати проблеми з попереднього пункту. Питання масштабування краще розв’язано у мікросервісній архітектурі, адже для зміни та додавання нових модулів не потрібно змінювати інші мікросервіси, що зменшує вірогідність виникнення проблем. І хоча монолітна архітектура не дозволяє повністю позбутись цієї проблеми, вона дозволяє швидше та простіше розгортати продукт, і він не повинен використовувати не вбудовані в мову програмування технології для комунікації між модулями.

Також доволі важливою темою у використанні мікросервісної архітектури є поняття стану системи, оскільки при проєктуванні системи потрібно розуміти де і яким чином ми зможемо отримувати актуальний стан усієї системи. З цим пов’язана CAP теорема, в якій говориться, що у системі з розподіленими обчисленнями може виконуватись не більше 2-х із пунктів [2]:

1. Consistency - всі клієнти бачать одні й ті ж дані одночасно, незалежно від того, до якого вузла вони підключаються. Для цього щоразу, коли дані

записуються на один вузол, вони повинні бути миттєво переслані або скопійовані на всі інші вузли системи, перш ніж запис буде вважатися "успішним".

2. Availability - будь-який клієнт, який робить запит на дані, отримує відповідь, навіть якщо один або декілька вузлів не працюють. Інший спосіб сказати це - всі робочі вузли в розподіленій системі повертають коректну відповідь на будь-який запит, без винятку.
3. Partition tolerance - втрачене або тимчасово затримане з'єднання між двома вузлами. Толерантність до відмов означає, що кластер повинен продовжувати працювати, попри будь-яку кількість розривів зв'язку між вузлами в системі.

Ця теорема означає те, що неможливо виконати усі 3 властивості системи, адже мікросервісна архітектура передбачає, що кожна складова частина застосунку матиме окремий доступ до інформації, а саме тому потрібно продумувати які саме властивості вашої системи є важливішими, враховуючи специфіку продукту. На основі цього рішення можна обрати різні інструменти, які будуть краще задовольняти вимоги проєкту.

1.7. Клієнтська частина продукту

Клієнт — з точки зору архітектури простіша частина продукту, оскільки вона завжди є умовно монолітною. Проте і вони поділяються на кілька видів, а саме:

- WEB-клієнти - це клієнтські додатки, які використовують веббраузер як платформу для відображення інтерфейсу користувача. Вони написані з використанням вебтехнологій, таких як HTML, CSS і JavaScript. Вебклієнт можуть бути статичними сторінками, динамічними односторінковими додатками (Single-Page Applications - SPA) або багатосторінковими додатками.

- Мобільні додатки — це додатки, які розробляються для мобільних пристроїв, таких як смартфони й планшети. Вони можуть бути написані з використанням нативних технологій платформи (наприклад, Swift/Objective-C для iOS або Java/Kotlin для Android) або використовувати гібридні технології, такі як React Native або Flutter.
- Десктопні додатки — це додатки, які призначені для встановлення і запуску на комп'ютерах або ноутбуках. Вони можуть бути написані з використанням платформо залежних технологій, таких як Java або C#, або використовувати кросплатформні фреймворки, такі як Electron або Qt.

РОЗДІЛ 2. ПРОГРАМНІ ЗАСОБИ РОЗРОБКИ КЛІЄНТ-СЕРВЕРНИХ ЗАСТОСУНКІВ

2.1 Розробка серверної частини

Проєктування та розробка серверної частини продукту є невіддільна та важливою частиною створення програмного продукту. Програмні засоби для її розробки включають:

1. Мови програмування та фреймворки — це ті складові, що визначають стек технологій, обмеження та переваги конкретної програми, адже різні мови програмування та відповідні їх фреймворки дозволяють як полегшити роботу інженера, так і оптимізувати виконання задач.

2. Інструменти розробки та налагодження — це частина, яка напряму на продукт не впливає, але без неї його створення та підтримка ускладнюється в рази. Насамперед потрібно піклуватися про систему контролю версій, яка могла б допомогти розробляти окремі складові продукту паралельно та допомагала усувати конфлікти. Також не варто забувати й про тестування продукту, адже при масштабуванні теперішньої системи та зміні функціонала можуть виникати небажані зміни, які можуть бути легко помічені за допомогою тестів. В залежності від мови та фреймворку можна вибрати різні засоби для створення тестів. Ще хочу помітити, що якщо команда обирає техніку розробки програмного забезпечення через тестування, що говорить про створення тестів з урахуванням усіх можливих сценаріїв використання коду до написання його написання, вибір технології тестування є невіддільною частиною проєктування програмного продукту.

3. Бази даних та системи управління базами даних (СУБД) - є важливими частинами, адже збереження інформації та робота по зберігання інформації на жорсткий диск відбувається саме тут. Без них

довелося б зберігати всі дані в оперативній пам’яті, що пришвидшило б роботу з ними, але не дозволило зберігати велику їх кількість. В залежності від вимог можна обирати різні за типом бази даних.

Нинішній розвиток технологій дозволяє нам бути доволі гнучкими у виборі інструментів розробки серверної частини, оскільки зараз більша частина робочого часу програми — це так звані I/O операції (читання, запис та видалення з пам’яті), тому більшість мов програмування та фреймворків вміють працювати асинхронно: за допомогою розділення на потоки, або інших інструментів розподілення ресурсів.

Поговоримо детальніше про мови та фреймворки, що використовуються для створення серверної програми. Для виявлення популярних мов скористаємось рейтингом популярності мов програмування та виділимо їх [3].

Java є найпоширенішою мовою для написання серверної частини продуктів, оскільки вона має увесь необхідний набір інструментів, довгу історію та підтримку спільноти. Також у поєднанні з нею використовуються такі популярні фреймворки як Spring, Java EE. Вона відома своєю надійністю, масштабованістю, можливістю запуску на великій кількості платформ та великою підтримкою бізнес-спільноти.

Другою за популярністю мовою є Python. Він простий у використанні і є мовою з високим рівнем абстракції, що значно полегшує вивчення. Також важливою частиною є фреймворки, що широко розповсюдженні у питанні використання цієї мови програмування, серед них: Django, Flask. Також ця мова програмування відома своїми можливостями для операцій з великою кількістю даних, довгими числами, тому вона часто використовується у сфері штучного інтелекту, що лише нарощує свою актуальність у наш час.

Доволі схожим за рівнем абстракції можна назвати Node.js – серверна мова програмування, яка використовує JavaScript для написання коду. З цього випливає що у розробників є можливість використовувати велику кількість

різних бібліотек для специфічних задач, адже саме javaScript наразі є найпопулярнішою мовою програмування, тому можемо припустити, що велика кількість специфічних питань уже вирішена, та саме ці рішення можуть бути використанні у розробці інших продуктів, за умови що існую необхідний пакет. Варто додати, що існує велика кількість фреймворків для Node.js, які дозволяють розробити повноцінну та швидку серверну частину продукту. Наразі одними з найбільших фреймворків є Nest.js та Express.

Кожен з цих варіантів має свої переваги та недоліки. Наприклад, Java забезпечує високу продуктивність, але може вимагати більше коду. Python має простоту та широку підтримку, але може бути менш продуктивним для великих навантажень. Node.js забезпечує швидкість та масштабованість, але може бути складнішим у використанні.

Вибір мови та фреймворку залежить від потреб проєкт. Наприклад, для великих корпоративних систем Java може бути кращим варіантом, тоді як для швидкого розгортання стартапу може бути більш слушним використання Python або Node.js.

Також варто зазначити, що під час проєктування серверної частини продукту варто замислитись над використанням шаблонів проєктування, адже вони дозволять пришвидшити розробку та зробити проєкт більш надійним. Використання навіть базових патернів може значно покращити роботу продукту. Самі патерн поділяються на 3 типи [4]:

1. Породжувальні патерни – забезпечують механізми створення об'єктів, приховуючи деталі процесу створення.
2. Структурні патерни – визначають способи організації класів та об'єктів в складніші структури. Вони допомагають забезпечити гнучкість та розширюваність системи.
3. Поведінкові патерни – визначають взаємодію між об'єктами та способи організації поведінки системи.

4. Архітектурні патерни – стосуються організації архітектури програмної системи. Вони визначають загальну структуру та організацію компонентів системи.

2.2 Бази даних та СУБД

Історія розвитку БД та СУБД починається з появи реляційної моделі в 1970-х роках, яку вперше висунув Едгар Кодд [5]. Вона стала основою для великої кількості комерційних та відкритих джерел СУБД, які існують сьогодні. Протягом останніх десятиліть з'явилися нові технології, такі як NoSQL, які дозволяють більш гнучко обробляти великі обсяги даних та забезпечувати високу продуктивність.

Сучасний вибір БД та СУБД базується на комбінації функціональних потреб проєкт, вимог до продуктивності, масштабованості, доступності та вартості впровадження та підтримки. Критерії, які мають бути враховані при виборі БД та СУБД, включають продуктивність, надійність, простоту у використанні, підтримку командної розробки та забезпечення безпеки даних.

Почнемо з реляційних баз даних, які використовують мову SQL запитів, для комунікації із серверною частиною продукту.

Основні компоненти реляційної бази даних включають таблиці, які представляють сутності та зв'язки між ними, атрибути, які визначають властивості сутностей, і ключі, які ідентифікують унікальні записи в таблицях. Однією з переваг реляційних баз даних є їх структурованість та можливість забезпечити цілісність даних. Вони дозволяють використовувати комплексні запити, включаючи об'єднання, сортування, групування та фільтрацію даних. Також варто зазначити, що існує припущення, що будь-який запит природною мовою можна сформулювати у мові SQL, якщо база даних правильно побудована, проте ні спростувати, ні довести цей факт формально ще не вдалося.

Проте, реляційні бази даних також мають свої недоліки. Наприклад, при роботі з великими обсягами даних або при високих навантаженнях вони можуть бути менш ефективними з точки зору продуктивності. Крім того, модифікація структури таблиць може бути складною та вимагати додаткових зусиль.

Кращий випадок використання реляційних баз даних включає ситуації, коли потрібно зберігати структуровані дані з чіткими зв'язками між ними, виконувати складні запити та забезпечувати цілісність даних. Реляційні бази даних широко використовуються у багатьох галузях, таких як фінанси, торгівля, логістика, управління відносинами з клієнтами (CRM) та інші.

Основними СУБД для реляційних баз даних є MySQL, PostgreSQL, Oracle, Microsoft SQL Server, які мають різні функціональні можливості та підтримують широкий спектр інструментів розробки.

Якщо коротко говорити про нереляційні бази даних, то хотілося б відзначити 2 види: документоорієнтовані бази даних, та Time-series dababases. Перша дозволяє не зберігати зв’язки між таблицями, що дозволяє простіше модифікувати та отримувати дані, проте вимагає більшого знання для формування складних запитів. Серед таких баз даних найбільш поширеною є Mongo.db, і її використання для розробників часто полегшено наявністю великої кількості різноманітних бібліотек для роботи з нею.

Time-series databases, своєю чергою призначенні для зберігання даних, які тісно пов’язані з часом. Насамперед це необхідно для продуктів, які надають послуги моніторингу, аналізу в реальному часі, прогнозуванні.

Особливість TSDB полягає у тому, що вона оптимізована для швидкого запису та запитів по часу. Вона забезпечує спеціалізовані механізми для зберігання, індексування та операцій з часовими рядами даних, такими як агрегація, інтерполяція, розрахунок статистики та інші аналітичні операції.

У світі існує кілька популярних Time-series databases, таких як InfluxDB, Prometheus, OpenTSDB, Graphite та інші. Кожна з них має свої особливості, можливості та рівень масштабованості. Вибір конкретної TSDB залежить від потреб проєкт, обсягу даних, потужностей обробки та інших факторів.

2.3 Розробка клієнтської частини

Клієнтська частина програмного продукту є графічним інтерфейсом, з яким буде взаємодіяти користувач тому для його розробки й проєктування потрібно опиратись на інші, порівняно з серверною частиною, критерії.

Насамперед необхідно визначитись із платформою, яку повинен підтримувати застосунок, наприклад, операційна система та її версії, для десктопних та мобільних додатків, або браузери для WEB додатків. Після того, як ми визначились із середовищем, в якому користувачі повинні взаємодіяти із клієнтом, можемо починати обирати мову програмування та фреймворки для наших потреб.

Для WEB додатків, зазвичай використовується мова програмування JavaScript, оскільки ця мова програмування була створена саме для використання у середовищі браузера у 1995 році. Це насамперед асинхронна мова програмування, що використовує лише один потік та працює в ньому. В цілому вона розроблялась для реалізації взаємодії та впровадження більшої функціональності у HTML елементи, проте пізніше стало можливим і виконання більш специфічних бізнес-задач., о, проте розробку можна вести й за допомогою Typesript - надбудови над JS, яка є строго типізованою, та є більш подібною до інших мов програмувань, таких як Java або C#. Вибір же фреймворку і досі викликає суперечки у спільноті, адже вони всі створені різними людьми, які бачать пріоритетність розв'язання проблем по-різному. У випадку із клієнтською частиною, ми не можемо не брати до уваги кількість ресурсів, які використовує продукт, якщо ми хочемо збільшити кількість

користувачів, адже у кожного різна конфігурація системи, що означає, що якщо клієнт чудово працює на одному комп’ютері — це далеко не завжди говорить про те, що він буде так само гарно працювати й на іншому.

Для створення ж продуктів для персональних комп’ютерів не існує єдиного стандарту, тому для різних задач та операційних систем модна обирати різні мови програмування та фреймворки. Для розробки на Windows зазвичай використовується C#, оскільки в ньому та його бібліотеках гарно реалізована взаємодія з даною операційною системою. Насамперед це викликано тим, що автором як Windows, так і C# є компанія Microsoft.

Java також є популярним інструментом для створення клієнтської частини програмного продукту, адже вона була розроблена таким чином, що для запуску на різних платформах вона використовує Java Vurtula Machine, яка дозволяє одному і тому самому коду працювати у багатьох середовищах. В тому числі й на мобільних пристроях. для цього до Java додають kotlin, що дозволяє створювати продуктивні користувацькі інтерфейси для операційної системи Android.

Для операційної ж системи IOS, та продуктів компанії Apple часто використовують Swift - мову програмування, розроблену компанією Apple. Вона була випущена у 2014 році та швидко набула популярності серед розробників. Основні переваги Swift називають безпека та надійність, адже ця сова програмування була створена з метою покращення безпеки програмного забезпечення. Вона має вбудовану систему безпеки типів, що допомагає уникнути багатьох типових помилок під час компіляції. Swift використовує прогресивний оптимізатор компілятора, що дозволяє створювати швидке та ефективне програмне забезпечення. Вона пропонує простий та експресивний синтаксис, що дозволяє розробникам писати чистий та зрозумілий код. Swift підтримує багатофункціональність, включаючи функціональне програмування, обробку помилок, генерики та інші сучасні функції. Вона має вбудовану

підтримку для роботи зі специфічними функціями та фреймворками, які використовуються в платформі Apple.Swift може співіснувати з кодом, написаним на мові Objective-C. Це означає, що розробники можуть поступово переходити до Swift, додаючи новий функціонал або модернізуючи чинний код без необхідності повного переписування програми.

Також варто зауважити, що тестування клієнтської частини є вкрай важливою частиною розробки, адже при проєктуванні ви не можете передбачити усі можливі варіанти дій користувача. Взаємодія ж з сервером відбувається на програмному рівні, тому тут не повинно виникати проблем. Для налагодження таких додатків можуть використовуватись різні підходи й види тестування, такі як:

- Unit тест — перевіряє роботу окремих функцій або модулів системи.
- Функціональний тест — перевіряє роботу продукту на рівні користувацького інтерфейсу, імітуючи дії реальної людини.
- Інтеграційний тест — перевіряє взаємодію різних компонентів та модулів системи між собою. Допомагає виявити помилки та проблеми, що виникають при інтеграції складових частин продукту.

2.4 Контейнеризація та оркестрація контейнерів

Після проєктування та написання коду виникає питання як змусити продукт працювати не локально на комп’ютері інженера, а у мережі. Для цього досить часто використовують технологію контейнеризації та оркестрації таких контейнерів. Це потрібно для того, щоб наш продукт міг запуститись у хмарних середовищах. Для контейнеризації часто використовують Docker. Для того, щоб розуміти як він працює, почнемо з основних понять:

- Image - це файл, який зберігає інструкції для створення Docker Container.

- Container - це стандартизована одиниця середовище, всередині якої запущена програма, яку ми зберігаємо в image

Для того, щоб ми могли працювати з докером, перш за все необхідно встановити середовище Docker, в якому налаштувати Unix подібну систему, в якій він власне і буде запускати наші додатки. У випадку із серверами проблем на цьому етапі майже не виникає, адже більшість з них працюють на операційній системі Linux, на обраному дистрибутиві, проте у випадку робочого простору інженера не завжди використовується Linux або MacOS. У випадку використання Windows необхідно налаштувати WSL(Windows subsystem for Linux), віртуально середу Linux всередині Windows.

Після запуску контейнерів, нам необхідно налаштувати комунікацію між ними, для цього і потрібен оркестратор контейнерів.

Оркестрація контейнерів є процесом автоматизації розгортання, керування та масштабування контейнерних додатків. Вона забезпечує ефективне управління контейнерами, що працюють в розподіленому середовищі, дозволяє керувати великою кількістю контейнерів, що працюють на різних фізичних або віртуальних машинах, забезпечує автоматичне розгортання, масштабування, балансування навантаження, відновлення після збоїв та інші функції для забезпечення безперебійної роботи додатків.

Одним з найпопулярніших інструментів оркестрації контейнерів є Kubernetes. Kubernetes (часто скорочено до K8s) надає повний набір функцій для автоматизації керування контейнерами. Він дозволяє декларативно описувати стан системи та автоматично керувати розгортанням, масштабуванням та керуванням життєвим циклом контейнерів. Він використовує так звані Nodes, для своїх цілей. Зазвичай на великих серверах розгортається кілька віртуальних машин, і Kubernetes використовує кожну з них, тобто він може працювати як із нодами на одному фізичному комп’ютері, так і на кількох. Наразі він став стандартом, та підтримується Google.

Крім Kubernetes, існують інші інструменти для оркестрації контейнерів, такі як Docker Swarm і Apache Mesos. Docker Swarm є вбудованим інструментом у Docker, який надає простий спосіб керування контейнерами у режимі кластера. Apache Mesos є розподіленим системним ядром, що дозволяє розгортати та керувати контейнерами разом з іншими обчислювальними ресурсами [6].

Оркестрація контейнерів забезпечує наступні переваги:

1. Масштабованість: За допомогою оркестрації контейнерів можна легко масштабувати додатки шляхом додавання або видалення контейнерів, забезпечуючи еластичність та високу доступність.
2. Балансування навантаження: Оркестрація контейнерів дозволяє автоматично розподіляти навантаження між контейнерами, забезпечуючи оптимальне використання ресурсів та підтримуючи стабільну продуктивність.
3. Збільшена доступність: Завдяки оркестрації контейнерів можна досягти високої доступності шляхом розподілу контейнерів на різні вузли, автоматичного відновлення після збоїв та резервного копіювання даних.
4. Простота управління: Оркестрація контейнерів надає зручні інтерфейси для керування контейнерами, такі як командні рядки, API та графічні інтерфейси, що спрощують керування додатками в розподіленому середовищі.
5. Швидке розгортання: Завдяки оркестрації контейнерів можна швидко розгортати нові версії додатків, забезпечуючи швидкий цикл розробки та постачання. Наприклад, якщо у вас є клієнт-серверний продукт, який складається з клієнтської програми, розробленої з використанням Angular, та серверного на основі Nest.js та MongoDB, ви можете використовувати Kubernetes для оркестрації цих компонентів.

Kubernetes дозволить автоматично масштабувати та керувати контейнерами з Angular та Nest.js, забезпечуючи їх безперебійну роботу та оптимальне використання ресурсів.

Переваги використання цієї технології проявляються ще більше при використанні мікросервісної архітектури, наведу кілька сценаріїв що демонструють їх:

1. Вебдодатки з великою кількістю користувачів: Якщо у вас є вебпродукт з великим навантаженням, оркестрація контейнерів дозволяє масштабувати його горизонтально, тобто додавати нові екземпляри контейнерів для обробки навантаження, що зростає.
2. Мікросервісна архітектура: У мікросервісній архітектурі продукт складається з набору незалежних сервісів, які можуть бути розгорнуті та масштабовані окремо.
3. Контейнеризація додатків з різними залежностями: Якщо ваш продукт має різні залежності, які можуть конфліктувати між собою, оркестрація контейнерів дозволяє розгортати ці залежності у відокремлених контейнерах. Це дозволяє забезпечити ізольованість та стабільність роботи продукту, оскільки конфлікти між залежностями не впливатимуть на всю систему.

РОЗДІЛ 3.Розробка клієнт серверного продукту.

3.1 Проєктування продукту та вибір інструментів розробки.

Для порівняння процесу побудови клієнт-серверного продукту за допомогою мікросервісної архітектури та монолітною побудуємо простий веб продукт. Серверна частина буде складатись з 2-х сутностей: продукт, та замовлення. Клієнтська частина даватиме можливість продивлятись список усіх продуктів, список усіх замовлень та деталі замовлення, при чому із деталізованою інформацією про продукт.

Для цієї задачі було вирішено використовувати мову програмування JavaScript для побудови клієнтської частини продукту, проте варто зазначити що для оптимізації процесу розробки та роботи самого продукту, необхідно обрати фреймворк. Серед найпопулярніших фреймворків та бібліотек хотілося б розглянути 3: React.js, Angular, Vue, адже кожен з них має свої особливості.

React.js - це бібліотека, яка у поєднанні з певними бібліотеками може бути використана як фреймворк для розробки клієнта. Отже, основним для цього інструменту є компонентний підхід, що дозволяє розбити користувацький інтерфейс на окремі незалежні частини, що робить керування та перевикористання попередньо написаного коду ефективнішим. Також на ефективність впливає і технологія віртуального DOM'у, який дозволяє швидко та без використання великої кількості ресурсів маніпулювати деревом елементів у браузері. Та не варто забувати про те, що React є лише бібліотекою, тому не є надто структурованим, що дає розробникам більшу гнучкість у питанні створення користувацького інтерфейсу.

Натомість Angular є повноцінним фреймворком, з усіма перевагами та недоліками. Серед переваг варто зазначити те, що він надає вбудовані рішення для поширених проблем без використання сторонніх залежностей. Також потрібно помітити, що він розроблений компанією Google, що забезпечує його високим рівнем документації та підтримки. Проте в цьому і криється основний

недолік цього фреймворку, адже він є досить суворо структурованим, що вимагає від розробника гарно розумітись, щоб написати ефективно робочу програму.

Vue ж, мабуть, найпростіший у вивченні фреймворк для розробки вебпродуктів, що дозволяє початківцям менше вивчати теорію, та навчатись на практиці. Також він є прогресивним фреймворком, та може бути легко інтегрованим у чинні користувацькі інтерфейси.

Також варто зазначити, що ці три фреймворки можуть і комбінуватись за допомогою технології iframe, яка дозволяє вбудовувати один вебдокумент (наприклад, HTML-сторінку або відео) в межах іншого вебдокумента. Він створює контейнер, в якому можна відображати зовнішні вміст або додаткову вебсторінку.

Для нашого продукту було обрано React.js, зважаючи на його переваги та недоліки, оскільки для розробки такої програми не потрібно залучати сторонні інструменти.

Серверну частину нашого продукту було вирішено написати, використовуючи мову Node.js, адже для її розгортання та контейнеризації необхідна мінімальна кількість ресурсів, та вона є досить ефективною.

Проте вибір фреймворку є досить важливим, адже реалізація навіть настільки нескладної взаємодії із клієнтом, без їх використання могла б призвести до втрати ефективності, що нівелювало б перевагу даної технології. Головними гравцями на цьому ринку на сьогодні є Express та Nest.js.

Nest.js доволі молодий фреймворк, який активно підтримується спільнотою розробників, та має синтаксис досить подібний до Java та C#. В ньому, по аналогії з Angular, реалізовано велику кількість розв'язання типових задач, серед яких:

- робота з нереляційними базами даних
- SSR

- Web Socket
- мікросервісна архітектура продукту

У поєднанні з гарно оформленою документацією та великою кількістю навчального матеріалу, Nest.js користується великою популярність серед Node.js розробників.

Express же своєю чергою старший за свого конкурента, що говорить нам про те, що він є гарно вивченим, та для нього існує велика база знань з усієї спільноти. Насамперед ця технологія приваблює своєю мінімалістичністю, по аналогії з React.js, Express реалізує лише базову частину функціонала, що дозволяє використовувати його з різними бібліотеками та компонентами, проте у нашому випадку і його буде достатньо.

Отже, для нашого продукту буде більш ефективним використання Express.js.

Власне для створення самих контейнерів альтернатив у нас немає, тому будемо використовувати Docker для цих задач.

У питанні ж оркестрації все доволі очевидно, виходячи з аналізу ринку, проте варто розуміти чому. Серед великої кількості таких інструментів найпоширенішими є Kubernetes, Docker Swarm, Apache Mesos.

Оскільки Docker допоміг популяризувати контейнери, напевно, не дивно, що його інструмент оркестрування, Docker Swarm, був одним з перших, хто отримав популярність після його випуску у 2013 році. Хоча деякі компанії все ще використовують Swarm, він краще підходить для менших робочих навантажень, і його майбутнє дещо невизначене. Mirantis придбала Swarm у Docker у 2019 році разом з Docker Enterprise.

Kubernetes - це платформа для оркестрування контейнерів, коріння якої сягає Borg, платформи, яку Google створив для управління своєю великою інфраструктурою. Kubernetes був відкритий Google у 2014 році, і з того часу він став проєкт Фонду нативних хмарних обчислень (Cloud Native Computing

Foundation, CNCF). Kubernetes затьмарив більшість інших ранніх платформ для оркестрування контейнерів, ставши стандартом де-факто.

Apache Mesos: Розробка Mesos почалася у 2009 році, ще до популяризації контейнерів, як інструмент для забезпечення управління кластерами. Він підтримує оркестровку як контейнерних, так і не контейнерних робочих навантажень. Mesos 1.0 був випущений у 2016 році. Компанія Mesosphere комерціалізувала Mesos за допомогою своєї пропозиції DC/OS, зробивши дуже складну технологію більш доступною для IT-команд. Однак у 2019 році Mesosphere змінила назву на D2iQ і перейшла на Kubernetes.

Як джерело збереження інформації для нашого застосунку було обрано MongoDB, оскільки вона є доволі простою у використанні. Для цього створимо 2 кластери, в яких будуть зберігатись дані. Згадуючи CAP теорему, можемо сказати, що із використанням наших кластерів, функціональності набору реплік та встановленням пріоритетності операції зчитування наша система може бути ідентифікована як AP, оскільки при кожному запиті ми отримуватимемо лише актуальну інформацію, проте при відсутності доступу до основного вузла, ми все ще будемо можливість працювати із другорядними, проте, не гарантуючи, що усі дані будуть актуальними, саме тому ми не можемо сказати, що усі вузли будуть надавати оновлену та актуальну інформацію.

Обраний інструментарій розробки монолітної версії продукту - Node.JS React.js, Docker, mongoDB. Для мікросервісного до списку технологій додаємо Kubernetes, nginx.

Структура серверної частини продукту з використанням мікросервісної архітектури виглядатиме так:

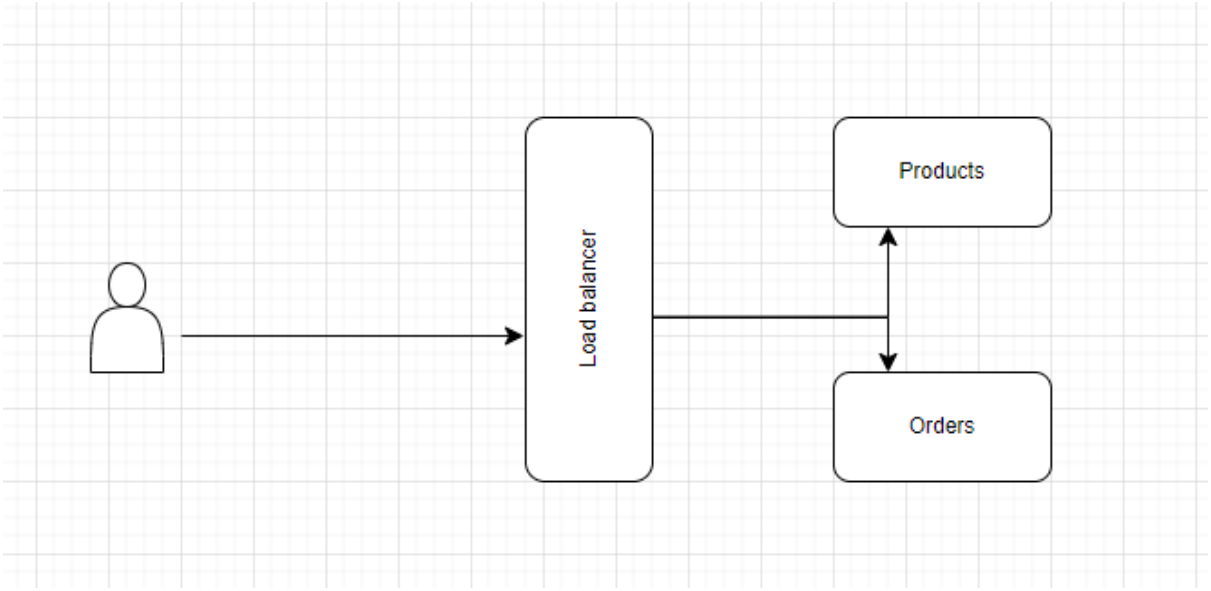

Рисунок 3.1 — Структура серверної частини з використанням мікросервісів.

3.2 Зовнішній вигляд клієнтської частини.

Клієнтський продукт однаково працює для обох архітектур, тому поглянемо як він виглядає:

Сторінка зі списком продуктів:

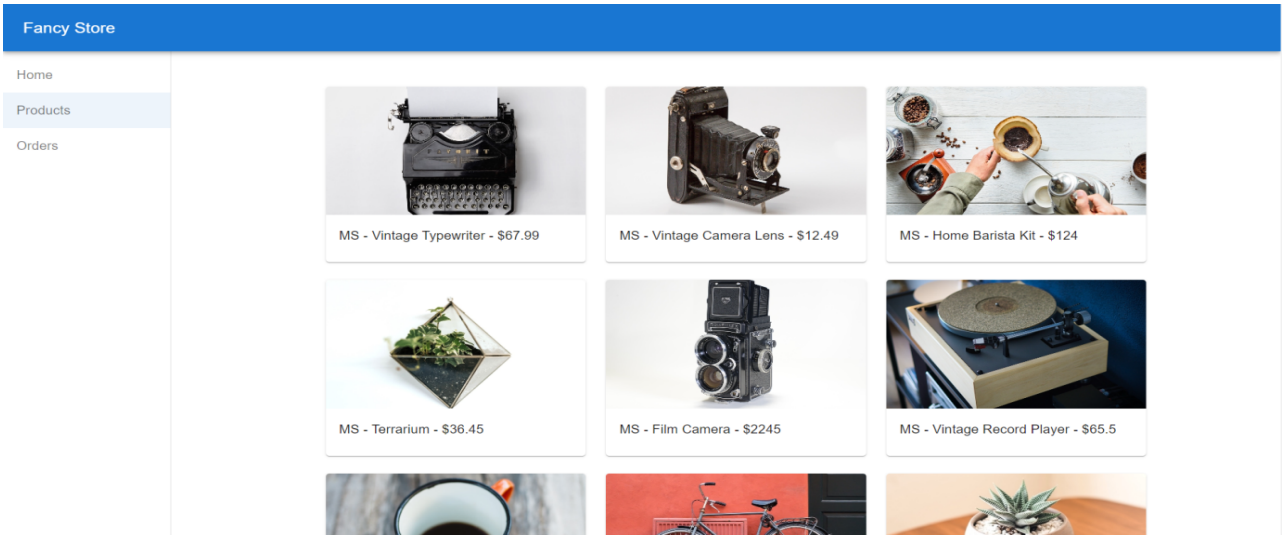

Рисунок 3.2 — сторінка продуктів

Сторінка зі списком замовлень:

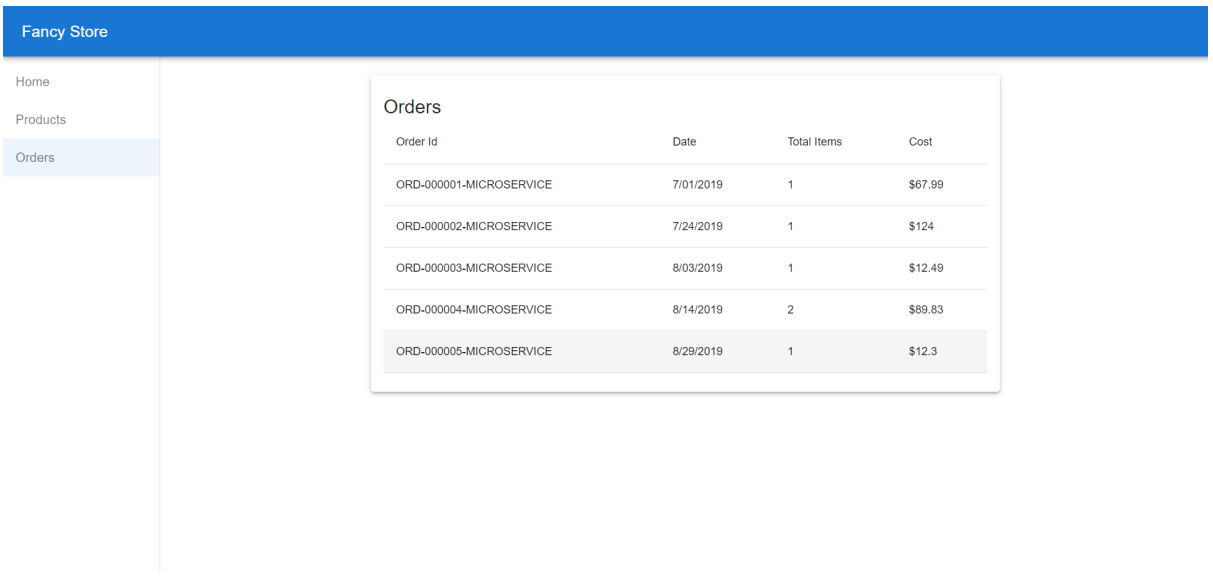

Рисунок 3.3 — сторінка замовлень

Детальна інформація про замовлення:

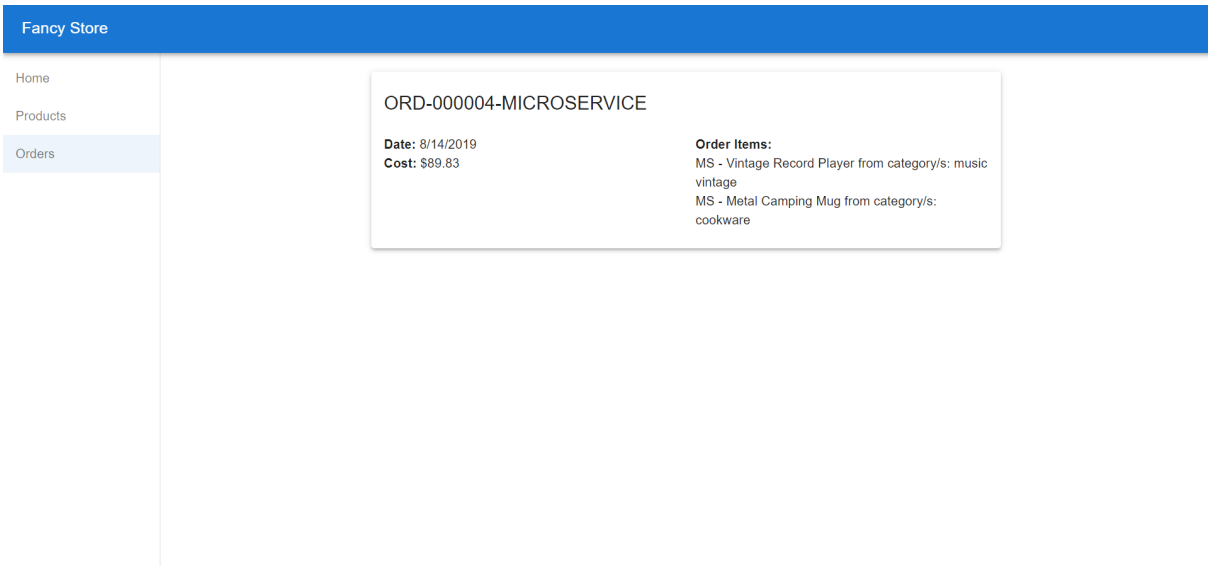

Рисунок 3.4 — сторінка детальної інформації про замовлення

Сторінка з помилкою, у разі якщо не вийшло отримати дані з сервера:

Рисунок 3.5 — сторінка зі станом помилки

Код із прикладом створення одного із компонентів наведено у додатку А.

3.3 Серверна частина продукту.

Серверна логіка у двох варіаціях нашого продукту дещо відрізняється одна від одної. Для монолітної архітектури маємо головну частину програми, що імпортує логіку сутностей, кожна з яких має можливість надати список усіх наявних об’єктів та детальну інформацію про кожен з них. Зауважимо, що при отриманні детальної інформації про замовлення програма також надає детальну інформацію і про продукт, який в нього входить, що є прямою взаємодією двох сутностей у рамках однієї програми.

Переробити ж таку версію у мікросервісну не є складною задачею, адже у нас уже є дві окремих сутності, питання полягає в тому, як вони повинні комунікувати один з одним. Для цього є 2 варіанти:

- клієнтська частина отримує деталі замовлення з ідентифікаторами продуктів, що містяться в ньому, і сама відправляє запити для отримання детальної інформації про кожен з них

- при запиті мікросервіс замовлень комунікує із мікросервісом продуктів, та самостійно формує деталізовану відповідь

Перший варіант може працювати, проте тоді нам потрібно буде змінювати клієнтську частину, та і в цілому наша система буде більш схожою на 2 окремих, нічим не пов’язаних із собою, сервіси. Тому обираємо другий варіант.

Для цього використовуватимемо nginx, у використанні якого отримаємо такі переваги:

- Проксі-сервер та балансувальник навантаження: Nginx може виконувати роль проксі-сервера для мікросервісної архітектури, приймаючи запити від клієнтів та передаючи їх до відповідних мікросервісів. Він також може функціонувати як балансувальник навантаження, розподіляючи навантаження між різними екземплярами мікросервісів.
- Управління маршрутизацією: Nginx дозволяє настроювати маршрутизацію запитів до різних мікросервісів на основі шаблонів URL або інших критеріїв. Це дозволяє ефективно керувати розподілом запитів до різних мікросервісів залежно від потреб продукту.
- Захист мікросервісів: Nginx може виконувати функції захисту мікросервісів, наприклад, використовуючи механізми автентифікації, авторизації, обмеження швидкості та захист від атак, таких як DDoS.

Для конфігурації кожного мікросервісу створено файл конфігурації, приклад якого наведено у додатку Б. Також було створено файл конфігурації Ingress, всередині якого ми конфігуруємо доменне ім’я кожного мікросервіса всередині Kubernetes мережі, тому комунікація між, до прикладу, мікросервісом замовлень та мікросервісом продуктів відбувається з використанням API, а саме, з огляду на конфігурацію: 'http://products-service:80/products'.

РОЗДІЛ 4. ТЕСТУВАННЯ МІКРОСЕРВІСНОЇ ТА МОНОЛІТНОЇ РЕАЛІЗАЦІЇ

4.1 Тестування продуктивності.

Власне, після запуску нашого продукту у локальному середовищі, необхідно провести їх тестування. Насамперед нас цікавить тестування продуктивності, завдяки чому ми зможемо провести аналіз та зрозуміти різницю між мікросеврвісною та монолітною реалізаціями нашого продукту.

Що ж таке тестування продуктивності, та що конкретно воно може нам дати. Згідно з Аяном Моліно: тестування продуктивності — це практика оцінки того, як система працює з точки зору швидкості відгуку і стабільності при певному робочому навантаженні. Тести продуктивності зазвичай проводяться для перевірки швидкості, стійкості, надійності та розміру програми [7]. Процес містити такі показники "продуктивності", як:

- час відгуку браузера, сторінки та мережі
- час обробки запитів на сервері
- допустима кількість одночасних користувачів
- споживання процесорної пам'яті; кількість і тип помилок, які можуть виникнути в додатку

Проведення такого роду тестування є необхідним перед впровадженням продукту до фінального користувача, оскільки воно дозволяє виявити проблемні місця системи та виправити їх до випуску самого продукту. Це може допомогти бізнесу заощадити чималі кошти, тому ця технологія користується великою популярністю. Розглянемо типи такого тестування:

- Тестування навантаження: цей вид тестування спрямований на визначення того, як система поводитися при певному навантаженні.

- Стрес-тестування: підвид тестування навантаження, що дозволяє перевірити роботу системи при надмірному навантаженні та дізнатись межу, після якої програма починає поводити себе некоректно.
- Тестування масштабованості: цей вид тестування дозволяє перевірити як система масштабується, в залежності від навантаження на неї, наприклад шляхом створення нових екземплярів окремих перевантажених сервісів.

Власне для нашого продукту будемо проводити поєднане тестування навантаження та стрес-тестування, поступово збільшуючи кількість запитів до системи.

Тестовий сценарій для цього доволі простий:

1. Перейти на сторінку зі списком замовлень.
2. Обрати замовлення, щоб переглянути деталізовану інформацію про нього.
3. Перейти на сторінку детальної інформації про замовлення, та впевнитись, що усі дані успішно отримані, в тому числі й детальна інформація про продукти, з яких сформовано замовлення

Оскільки тестування буде проводитись локально, для стрес-тестування буде достатнім використання 10 користувачів, яких будемо додавати поступово протягом хвилини, а потім ще хвилину вони будуть повторювати ці дії. Вимірюватимемо швидкість відповіді продукту, що дозволить нам зрозуміти наскільки швидко продукт передає дані, та скільки повинен буде чекати на отримання усіх необхідних даних користувач.

4.2 Засоби тестування.

Оскільки тестування продуктивності є вкрай важливим для бізнесу, на ринку присутня велика кількість інструментів, що дозволяють його проводити. Проведемо певний аналіз та оберемо ПЗ, яке буде оптимальним для розв'язання

нашої задачі та зробимо усю необхідну підготовчу роботу для проведення самого тестування.

Таблиця 4.1 Засоби проведення навантажувального тестування

ПЗ для навантажувального тестування	Засновники	Особливості
Apache JMeter	Apache Software Foundation	Вільне та відкрите програмне забезпечення, можливість створення складних тестових сценаріїв, підтримка різних протоколів зв'язку.
LoadRunner	Micro Focus	Комерційний продукт, широкий функціонал, підтримка різних протоколів та платформ, масштабованість.
Gatling	Gatling Corp	Вільне та відкрите програмне забезпечення, написане на Scala, зосереджений на продуктивності та простоті використання.
Tsung	ProcessOne	Вільне та відкрите програмне забезпечення, спеціалізоване на навантажувальному тестуванні для протоколу HTTP та WebDAV.
Locust	Locust.io	Вільне та відкрите програмне забезпечення, написане на Python, можливість розподіленого тестування та програмування тестових сценаріїв на Python.
Artillery	Artillery Software	Вільне та відкрите програмне забезпечення, спеціалізоване на тестуванні масштабованих систем та додатків з використанням протоколу HTTP.

Для нашого застосунку було обрано Apache Jmeter, адже в ньому представлена уся необхідна функціональність, та є можливість додати сторонні

розширення, а саме Selenium WebDriver, який дозволить нам проводити тестування з використанням реального браузера.

4.3 Тестування та аналіз результатів.

Для виконання нашого тестового сценарію створимо 2 окремих тести: для мікросервісної реалізації, та для монолітної. Необхідно додати 2 групи користувачів, та налаштувати тест, а саме кількість браузерів, період за який буде створено необхідну їх кількість, час виконання тестування, кількість циклів виконання, яких у нашому випадку нескінченність.

Тепер необхідно створити сам скрипт, який описує тестовий сценарій. Створимо його з використанням необхідного розширення та мови JavaScript.

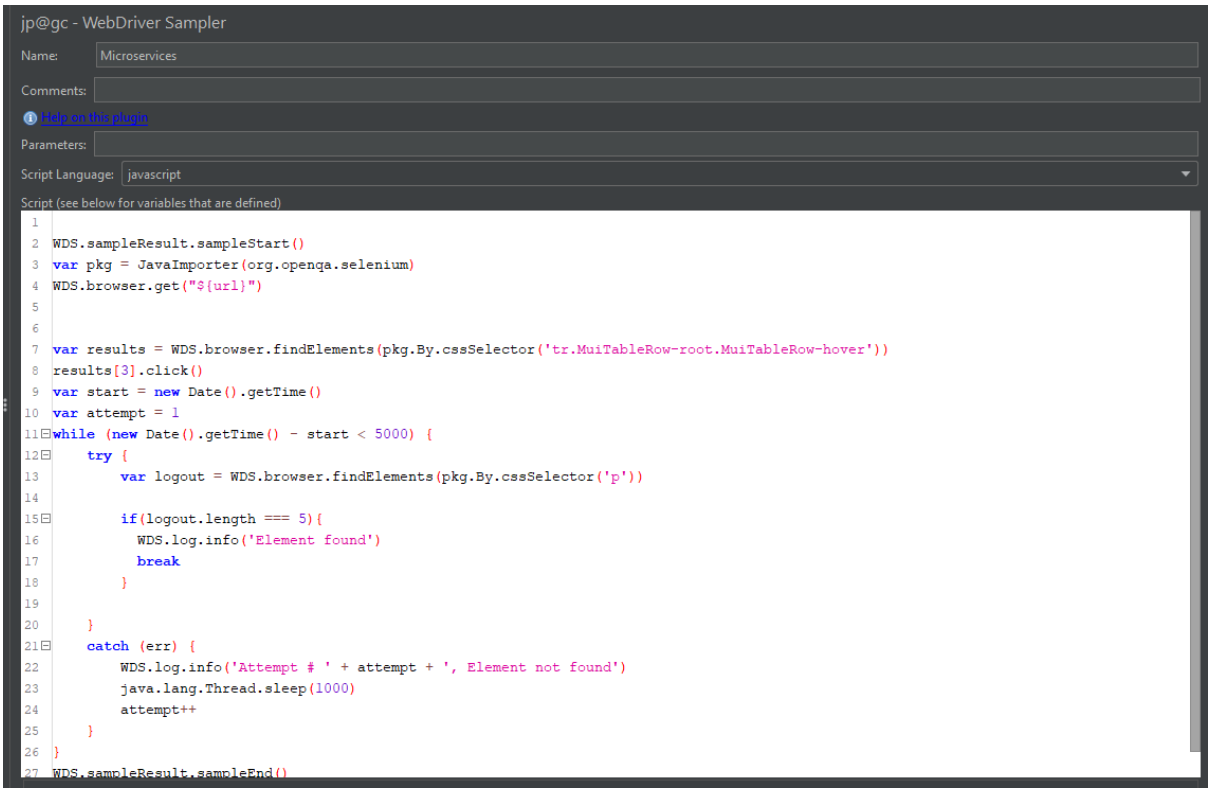

Рисунок 4.1 — скрипт тесту

Для монолітної ж тест відрізнятиметься лише 4 рядком, де будемо використовувати інше посилання. Конфігуруємо їх:

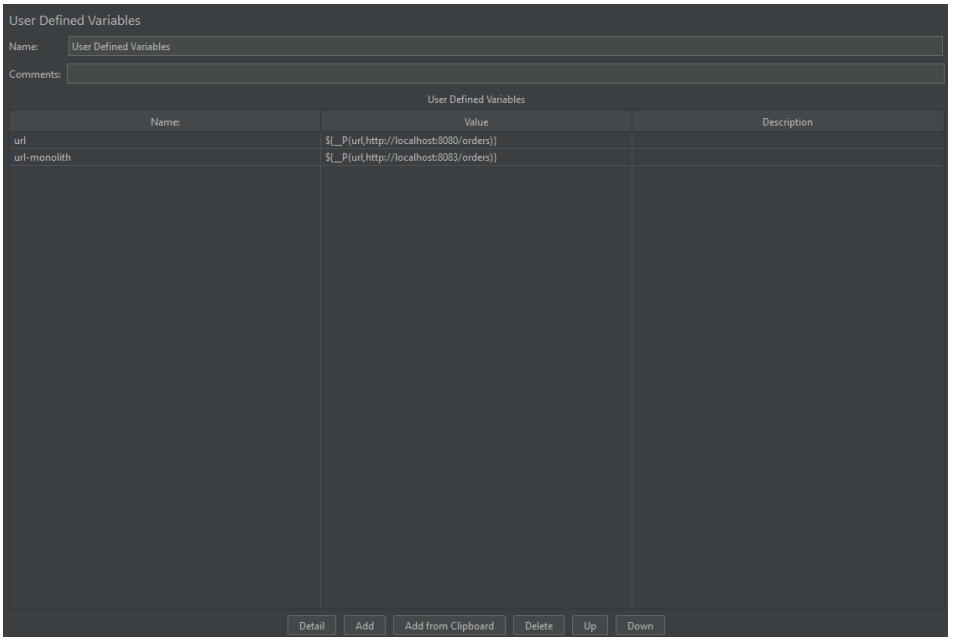

Рисунок 4.2 — посилання для тестування програм

Після завершення тестування отримуємо результати, приклад яких можна побачити у додатку В та за ними будуємо графіки, які допоможуть візуалізувати дані.

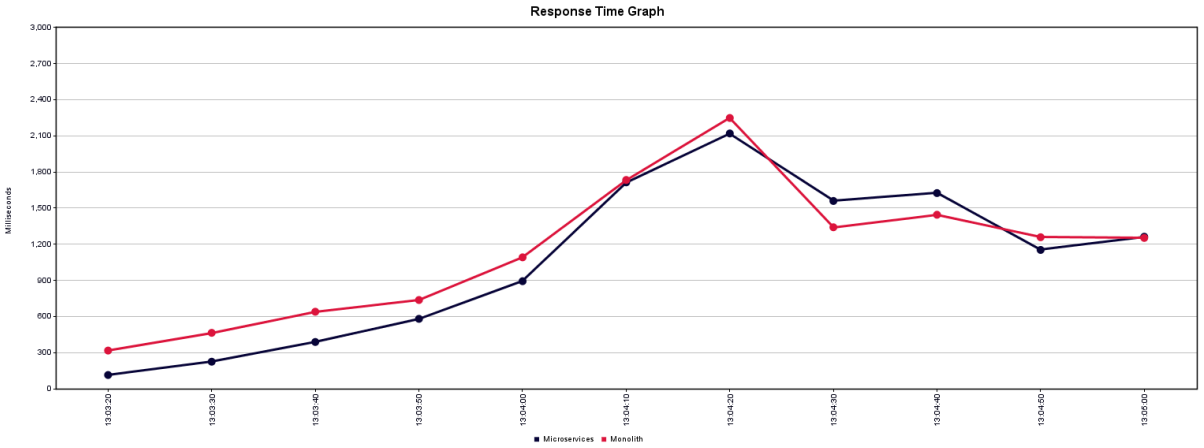

Рисунок 4.3 — графік середнього часу відповіді

Цей графік показує час, за який тест отримав відповідь від сервера (в нашому випадку за 1 секунду було запущено багато користувачів, тому це середнє значення часу виконання тесту). Тут бачимо, що мікросервісна архітектура працює швидше, оскільки у випадку з монолітом, при завантаженні однієї частини програми, інші також починають працювати повільніше.

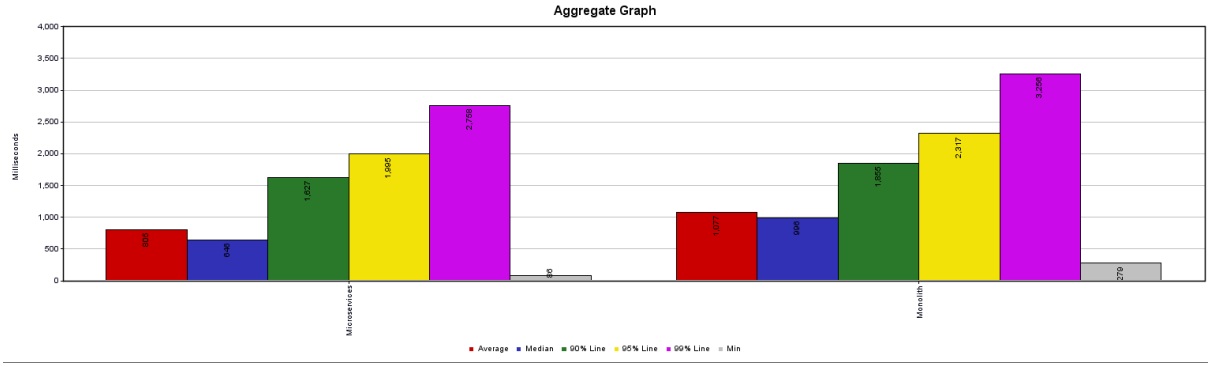

Рисунок 4.4 — діаграма часу відповіді

Цей графік показує середній час виконання тесту, медіанне значення, та 90, 95 та 99 процентилі. Тут ситуація також схожа, оскільки середні показники у моноліта вищі, в тому числі мінімальний час відповіді, це викликано тим, що для мікросервісної реалізації було використано розподілювач навантаження, тому відповідь була сформована дещо швидше.

Також не зайвим буде переглянути і цей графік:

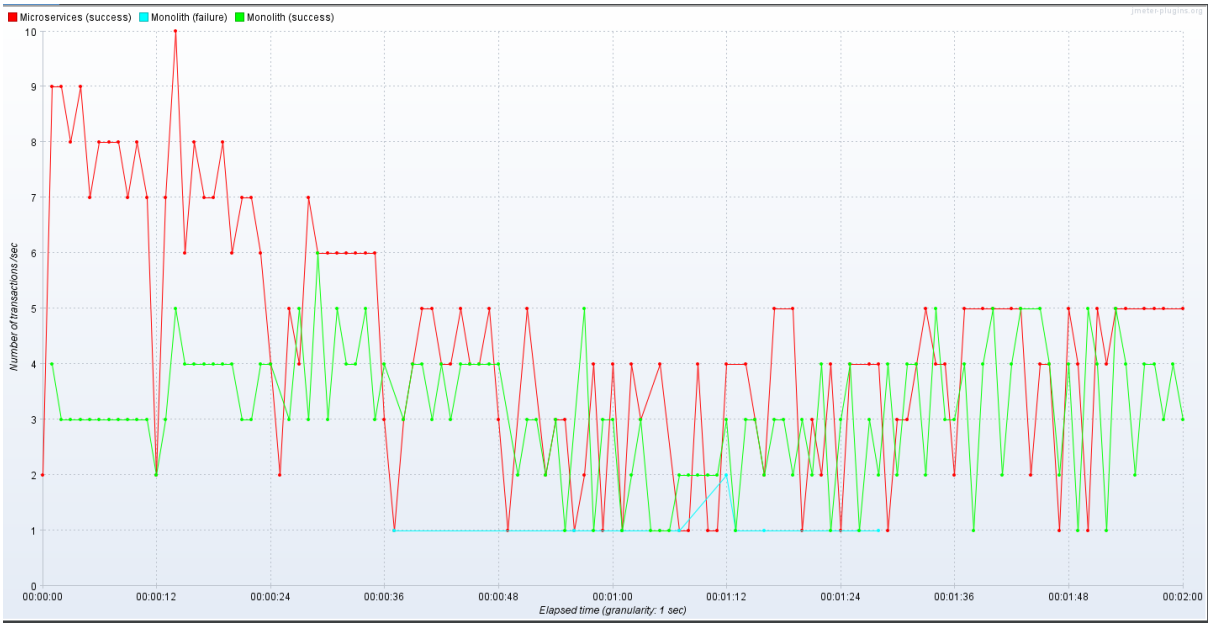

Рисунок 4.3 — графік кількості відповідей за секунду

Він показує кількість успішних запитів за певний проміжок часу. Тут можемо побачити, що для в цілому мікросервісна реалізація надавала більше успішних відповідей за секунду, та ні одної відповіді, що мала б помилку. Дещо інша картина для монолітної архітектури, адже вона, як можемо побачити, після

36 секунди тестування почала відповідати помилкою, з чого можемо зробити висновок, що монолітна реалізація виявилась менш відмовостійкою у порівнянні з мікросервісною.

Власне така поведінка пояснюється тим, що під час великого навантаження на один із модулів, у разі мікросервісної архітектури, інший працює без проблем, тому не відбувається затримки у відповідях. Натомість у монолітній архітектурі при навантаженні однієї частини програми, уся програма перестає працювати так, як було задумано.

ВИСНОВКИ

Відповідно поставленої мети, в ході виконання кваліфікаційної роботи, були реалізовано такі завдання :

1. Розглянуто актуальний стан монолітної та мікросервісної архітектур та проаналізувати основні проблеми.
2. Здійснено проєктування власного продукту, а також обрано інструментарій для розробки.
3. Розроблено продукт з використанням мікросервісної та монолітної архітектур.
4. Проведено тестування продуктивності.
5. Отримано результати тестування проведено їх аналіз, формулювання висновків.

У результаті проведеного дослідження було отримані певні цікаві результати, які не дозволяють точно висловити яка з клієнт-серверних архітектур є кращою, тому варто зауважити, що обидва варіанти можуть бути використані при розробці програмних продуктів, в залежності від вимог та потреб бізнесу.

Монолітна архітектура забезпечує простоту у розробці та випуску, оскільки всі компоненти продукту об'єднані в одному монолітному модулі. Вона може бути ефективною для невеликих проєктів зі стабільними вимогами, де швидкість розробки та випуску мають важливіше значення, ніж масштабованість та гнучкість.

З іншого боку, мікросервісна архітектура надає більшу гнучкість, масштабованість та можливість незалежної розробки та випуску окремих

сервісів. Цей підхід дозволяє зручно масштабувати окремі компоненти системи, використовувати різні технології та незалежно розвивати кожен сервіс. Однак, він також вимагає додаткової складності у керуванні та координації сервісів, а також збільшує навантаження на мережу та інфраструктуру.

Цей висновок точно відповідає стану сучасного рику, адже навіть великі технічні компанії використовують обидва види архітектур в залежності від проєкту.

Саме тому до проєктування архітектури застосунків потрібно підходити із конструктивним підходом та знанням предметної області, адже вибір рішення, яке не підходить проєкту, проте є дуже популярним у спільноті, не дасть бажаного результату, а може лише нашкодити як розвитку продукту, так і команді, яка розроблятиме його.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Kafka Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://kafka.apache.org/documentation/.
2. What is the CAP theorem? [Електронний ресурс] – Режим доступу до ресурсу: https://www.ibm.com/topics/cap-theorem.
3. Рейтинг мов програмування 2023. [Електронний ресурс]. – 2023. – Режим доступу до ресурсу: https://dou.ua/lenta/articles/language-rating-2023/.
4. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software / Gamma Erich, Helm Richard, Johnson Ralph, Vlissides John., 1994. – (AddisonWesley).
5. Codd E. A Relational Model of Data for Large Shared Data Banks [Електронний ресурс] / Edgar Codd – Режим доступу до ресурсу: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf.
6. Malmborg C. Container Orchestration Tools Comparison [Електронний ресурс] / Cliff Malmborg. – 2022. – Режим доступу до ресурсу: https://rafay.co/the-kubernetes-current/container-orchestration-tools-comparison/.
7. Molyneaux I. The Art of Application Performance Testing / Ian Molyneaux., 2009.

ДОДАТКИ

ДОДАТОК А

Приклад компонента клієнтської частини застосунку

OrderDetails.js

```
export default function OrderDetails() {
  const match = useRouteMatch();

  const [hasErrors, setErrors] = useState(false);
  const [order, setOrder] = useState({});

  const orderId = match.params.id;

  async function fetchOrder(orderId) {
    try {
      const response = await fetch(
        `${process.env.REACT_APP_ORDERS_URL}/${orderId}`
      );
      const order = await response.json();
      const orderItems = (await Promise.all(order.items.map(item =>
fetch(`${process.env.REACT_APP_PRODUCTS_URL}/${item}`))));
      order.items = [];
      for(let item of orderItems){
        const data = await item.json();
        order.items.push(data);
      }
      setOrder(order);
    } catch (err) {
      setErrors(true);
    }
  }

  useEffect(() => {
    fetchOrder(orderId);
  }, [orderId]);

  return (
    <Box sx={{ flexGrow: 1 }}>
      {hasErrors && (
```

```
  <Paper
    elevation={3}
    sx={{
      background: "#f99",
      padding: (theme) => theme.spacing(3, 2),
    }}
  >
    <Typography component="p">
      An error has occurred, please try reloading the page.
    </Typography>
  </Paper>
)}
{!hasErrors && (
  <Paper
    elevation={3}
    sx={{
      maxWidth: "800px",
      margin: "0 auto",
      padding: (theme) => theme.spacing(3, 2),
    }}
  >
    <Grid
      container
      spacing={3}
      justifyContent="flex-start"
      alignItems="stretch"
    >
      <Grid item xs={12}>
        <Typography variant="h5">{order.id}</Typography>
      </Grid>
      <Grid item md={6} xs={12}>
        <Typography component="p">
          <b>Date: </b>
          {order.date}
        </Typography>
        <Typography component="p">
          <b>Cost: </b>${order.cost}
        </Typography>
      </Grid>
      <Grid item md={6} xs={12}>
        <Typography component="p">
          <b>Order Items: </b>
```

```
            </Typography>
            {order.items &&
              order.items.map((item) => (
                <Typography key={item}>{item.name} from category/s: {item.categories.reduce((acc,
curr) => `${acc} ${curr}`)}</Typography>
              ))}
          </Grid>
        </Grid>
      </Paper>
    )}
  </Box>
 );
}
```

ДОДАТОК Б
Приклад конфігураційного файлу для Kubernetes

```
nginx:
apiVersion: networking.k8s.io/v1
kind: IngressClass
metadata:
  name: nginx
spec:
  controller: k8s.io/ingress-nginx

apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: microservices-ingress
spec:
  ingressClassName: nginx
  rules:
    - host: microservices.example.com
      http:
        paths:
          - path: /frontend
            pathType: Prefix
            backend:
              service:
                name: frontend-service
                port:
                  number: 80
          - path: /orders
            pathType: Prefix
            backend:
              service:
                name: orders-service
                port:
                  number: 80
          - path: /products
            pathType: Prefix
            backend:
              service:
                name: products-service
                port:
                  number: 80
```

Додаток В

Частина результатів тестування

	A	B	C	D	E	F
1	318	Microservices	200	OK	Users 1-1	TRUE
2	98	Microservices	200	OK	Users 1-1	TRUE
3	500	Monolith	200	OK	users monolith 2	TRUE
4	92	Microservices	200	OK	Users 1-1	TRUE
5	93	Microservices	200	OK	Users 1-1	TRUE
6	90	Microservices	200	OK	Users 1-1	TRUE
7	309	Monolith	200	OK	users monolith 2	TRUE
8	120	Microservices	200	OK	Users 1-1	TRUE
9	92	Microservices	200	OK	Users 1-1	TRUE
10	93	Microservices	200	OK	Users 1-1	TRUE
11	310	Monolith	200	OK	users monolith 2	TRUE
12	109	Microservices	200	OK	Users 1-1	TRUE
13	91	Microservices	200	OK	Users 1-1	TRUE
14	91	Microservices	200	OK	Users 1-1	TRUE
15	291	Monolith	200	OK	users monolith 2	TRUE
16	113	Microservices	200	OK	Users 1-1	TRUE
17	99	Microservices	200	OK	Users 1-1	TRUE
18	99	Microservices	200	OK	Users 1-1	TRUE
19	293	Monolith	200	OK	users monolith 2	TRUE
20	117	Microservices	200	OK	Users 1-1	TRUE
21	98	Microservices	200	OK	Users 1-1	TRUE
22	308	Monolith	200	OK	users monolith 2	TRUE
23	86	Microservices	200	OK	Users 1-1	TRUE
24	120	Microservices	200	OK	Users 1-1	TRUE
25	94	Microservices	200	OK	Users 1-1	TRUE
26	305	Monolith	200	OK	users monolith 2	TRUE