МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

КАФЕДРА ЕЛЕКТРОНІКИ, ІНФОРМАЦІЙНИХ СИСТЕМ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Кваліфікаційна робота

другий (магістерський)
(рівень вищої освіти)

на тему **Порівняння монолітної та мікросервісної архітектур на прикладі корпоративного застосунку**

Виконав: студент 2 курсу, групи 8.1211-2іпз
спеціальності 121 Інженерія програмного забезпечення
(код і назва спеціальності)
освітньої програми Інженерія програмного забезпечення
(код і назва освітньої програми)
В.В Новак
(підпис, ініціали та прізвище)
Керівник доцент, В.І. Попівщий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)
Рецензент директор ТОВ Дісітел П.О. Лютий
(посада, вчене звання, науковий ступінь, підпис, ініціали та прізвище)

Запоріжжя
2022

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ІНЖЕНЕРНИЙ НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ім. Ю.М. Потебні
ЗАПОРІЗЬКОГО НАЦІОНАЛЬНОГО УНІВЕРСИТЕТУ

Кафедра Електроніки, інформаційних систем та програмного забезпечення

Рівень вищої освіти _____другий (магістерський)_____

Спеціальність ___121_Інженерія програмного забезпечення_____
(код та назва)

Освітня програма ___Інженерія програмного забезпечення
(код та назва)

ЗАТВЕРДЖУЮ

Завідувач кафедри_____Т.В. Критська_____

“_12_” __вересня_____2022_року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Новаку Віктору Вікторовичу
(прізвище, ім’я, по батькові)

1.Тема роботи Порівняння монолітної та мікросервісної архітектур на прикладі корпоративного застосунку

керівник роботи Попівщий Василь Іванович, доцент
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від 02.06.2022 р. №597-с

2. Строк подання студентом кваліфікаційної роботи 1 грудня 2022 р.

3. Вихідні дані магістерської роботи

- комплект нормативних документів ;
- технічне завдання до роботи.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

- огляд та збір літератури стосовно теми кваліфікаційної роботи;
- огляд та аналіз існуючих рішень і аналогів;
- дослідження проблеми вибору між монолітною і мікросервісною архітектурою для вирішення різноманітних бізнес-задач;
- створення програмного продукту та його опис;
- перелік вимог для роботи програми;
- надати висновки та рекомендації стотосовно дослідженої проблеми

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)

слайдів презентації

6. Консультанти розділів магістерської роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата Завдання прийняв

7. Дата видачі завдання 01.09.2022

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської роботи	Строк виконання етапів магістерської роботи	Примітка
1	Аналіз предметної області	17.09 – 21.09.2021	виконано
2	Формулювання основної задачі дипломної роботи та узгодження її з науковим керівником	22.09 – 13.10.2021	виконано
3	Аналіз існуючих методів рішення	15.10 – 29.10.2021	виконано
4	Дослідження існуючих засобів для реалізації монолітної архітектури	30.10 – 26.11.2021	виконано
5	Дослідження існуючих засобів для реалізації мікросервісної архітектури	28.11 – 14.12.2021	виконано
6	Узгодження подальших дій з науковим керівником	15.12 – 25.02.2021	виконано
7	Реалізація корпоративного застосунку на основі монолітної архітектури	26.02 – 23.04.2022	виконано
8	Реалізація корпоративного застосунку на основі мікросервісної архітектури	24.04 – 08.07.2022	виконано
9	Представлення отриманих результатів науковому керівнику та узгодження плану подальшого дослідження	08.07 – 14.08.2022	виконано
10	Усунення додаткових проблем та «багів»	15.08 – 16.10.2022	виконано
11	Тестування та порівняння показників монолітної та мікросервісної архітектури	16.10 – 30.10.2022	виконано
12	Розгортання корпоративних застосунків	01.11 –15.11.2022	виконано
13	Оформлення звіту	16.11 – 1.12.2022	виконано

Студент _____ В.В. Новак
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.І. Попівщий
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ І.А. Скрипник
(підпис) (ініціали та прізвище)

АНОТАЦІЯ

Сторінок: 104

Рисунків: 56

Таблиць: 5

Джерел: 31

Новак В. В. Порівняння монолітної та мікросервісної архітектур на прикладі корпоративного застосунку: кваліфікаційна робота магістра спеціальності 121 «Інженерія програмного забезпечення» / наук. керівник В. І. Попівщий. Запоріжжя : ЗНУ, 2022. 104 с.

Мета і завдання дослідження полягають в визначенні переваг та недоліків монолітної і мікросервісної архітектур при розробці корпоративного застосунку та надання рекомендацій, яку з цих архітектур треба використовувати для конкретної предметної області в залежності від бізнес-задач, які виникнуть при майбутній розробці.

У процесі дослідження була розглянута проблема вибору архітектури корпоративних застосунків. Проведено теоритичні дослідження у галузі монолітної та мікросервісної архітектур. Розроблено корпоративні застосунки на основі монолітної і мікросервісної архітектур для доставки різних продуктів клієнту від різних партнерів. Виконано тестування для кожного застосунку. Виконано порівняння характеристик написаних застосунків за якісними та кількісними показниками. Визначено переваги та недоліки розглянутих архітектур та надано рекомендації щодо їх використання.

Ключові слова: *веб-сайт, веб-сервер, система управління базами даних, моноліт, клієнт, база даних, мікросервісна архітектура, Flyway, Java Spring, Spring Cloud, Spring Security, API-Gateway, Docker, Docker-compose, JWT, централізоване конфігурування, виявлення сервісів, Git, Kubernates, Minikube, Kubectl*.

SUMMARY

Pages: 104

Figures: 56

Tables: 5

Sources: 31

Novak V.V. Comparison of Monolithic and Microservice Architectures (Case Study: Corporate App): master's qualification thesis of specialty 121 "Software engineering" / science. manager V.I. Popivshchy. Zaporizhzhia: ZNU, 2022. 104 p.

The aim and objectives of the study are to determine the advantages and disadvantages of monolithic and microservice architectures when developing a corporate application and to provide recommendations on which of these architectures should be used for a specific subject area depending on the business tasks that will arise during future development.

In the process of research, the problem of choosing the architecture of corporate applications was considered. Theoretical studies were conducted in the field of monolithic and microservice architectures. Developed corporate applications based on monolithic and microservice architectures for delivery of various products to the client from various partners. Tested for each application. A comparison of the characteristics of the written applications was performed by qualitative and quantitative indicators. The advantages and disadvantages of the considered architectures are determined and recommendations for their use are provided.

Keywords: *website, web-server, database management system, monolith, client, database, mi-croservice architecture, Flyway, Java Spring, Spring Cloud, Spring Security, API-Gateway, Docker, Docker-compose, JWT, Service discovery, Centralized configu-ration, Git, Kubernates, Minikube, Kubectl*.

ЗМІСТ

ВСТУП

Актуальність теми

При розробці застосунків, програмісти та системні архітектори зіштовхуються з проблемою вибору архітектури. Адже правильно вибрана архітектура дає певний відсоток шансу на успішний спроєктований та зпрограмований продукт. І, тому програмісти перед розробкою програмної системи обговорюють на нарадах, за допомогою мозкового штурму: тенденції, переваги та недоліки кожної з існуючих архітектур. Адже, на жаль немає готового, загального шаблону, який буде в собі містити найкращі характеристики, щоб побудувати успішний програмний продукт.

При розробці майбутньої програмованої системи, програміст повинен розуміти, що додаток побудований на тій, чи іншій архітектурі, повинен мати такі характеристики, як відмовостійкість, розширюваність, надійність, масштабованість, супроводжуваність, безпеку, зручність використанні, тощо.

Монолітна архітектура є однією з популярних шаблонів, вона будується на основі клієнт-сервер, в якій функції презентації, обробки програм та управління даними фізично розділені. Вона передбачає наявність і використання наступних компонентів програми: клієнтський застосунок, підключений до сервера, який в свою чергу підключений до бази даних. Всі ці компоненти підтримуються та розробляються, як деякі незалежні модулі. Можна виділити такі переваги даної архітектури: високий рівень безпеки, масштабованість, надійність, конфігурованість, тощо. Але, ця архітектура має свої недоліки, вони пов’язані з тим, що розроблювані застосунки мають деяку складність створення, розгортання та адміністрування. Також, висока вартість серверів, бо для великої кількості запитів користувачів, повинна бути висока продуктивність та швидка відповідь між сервером застосунків та баз даних.

Однією з популярних архітектур, ще є мікросервісна архітектура — це архітектурний стиль, який призначений для побудови невеликих додатків-сервісів, кожен з яких працює у своєму власному процесі та спілкується з

рештою, використовуючи прості та швидкі протоколи передачі даних HTTP.

Основні властивості та функції мікросервісів це те, що вони можуть незалежно масштабуватися, тобто зміна в одному мікросервісі не вплине на інші; якщо залучається новий розробник, то зможе відразу приступити до написання іншого мікросервісу; сервіси надають простоту заміни реалізації на якусь іншу; незалежне додавання нового функціоналу в систему; також є така функція як еластичність, тобто вихід з ладу одного сервісу зазвичай не призводить до виходу з ладу всієї системи. Але потрібно розуміти, що немає найкращої технології, і в кожної є свої проблеми та недоліки.

Мікросервісна архітектура має проблеми з розподіленими транзакціями, проблеми CAP-теореми, значні викладні витрати на інфраструктуру, ускладнене налагодження, зневадження помилок в робочому сервісі. При написанні мікросервісів, їх незалежність призводить до дублювання коду, ускладненого тестування, розгортання, безпеки, що не є ефективним методом.

Також, при розробці застосунку, а особливо клієнтської частини, дизайнери та програмісти зіштовхуються з проблемою розробки інтерфейсу користувача, адже він повинен бути простим у використанні, легким для розуміння та навігації. Мати налаштовану безпеку застосунку, адже клієнт повинен розуміти, що всі його дані знаходяться в безпеці, і не будуть викрадені та використані в зловмисних цілях. Додаток повинен бути стійким до збоїв та мати неабияку швидкість, для того, щоб опрацювати запити клієнта, витративши на це мінімум часу.

Таким чином, проблема вибору архітектури є актуальною, адже від цього залежить успішний, побудований, легкий у використанні, швидкодійний та стійкий до збоїв проєкт.

Мета і завдання дослідження

Мета і завдання дослідження полягають в визначенні характеристик монолітної і мікросервісної архітектур при розробці корпоративного застосунку та надання рекомендацій, яку з цих архітектур треба

використовувати для конкретної предметної області в залежності від бізнес-задач, які виникнуть при майбутній розробці. Виконати тестування для кожного застосунку. Провести порівняння характеристик написаних застосунків за якісними та кількісними показниками.

Об’єкт дослідження

Об’єктом дослідження є мікросервісна та монолітна архітектури.

Предмет дослідження

Предметом дослідження є визначення характеристик мікросервісної та монолітної архітектур.

Методи дослідження

Для вирішення поставленої задачі використовуються наступні методи дослідження:

1. Аналіз особливостей та існуючих рішень мікросервісної та монолітної архітектур.
2. Аналіз переваг та недоліків кожної з архітектур.
3. Аналіз різноманітних бібліотек, призначених для реалізації.
4. Аналіз систем автоматичного управління та розгортання застосунків у контейнерах.
5. Аналіз різних інструментаріїв, які призначені для управління ізольованими контейнерами.
6. Експериментування з різними інструментами, які призначені для нагляду за статусом роботи мікросервісів.
7. Аналіз бібліотек тестування монолітної архітектури та мікросервісів.

Наукова новизна одержаних результатів

Наукова новизна одержаних результатів дослідження полягає у тому, що для розробки корпоративного застосунку були використані монолітна та

мікросервісна архітектури, що дало змогу в процесі тестування зі зміною навантаження визначити переваги та недоліки цих двох реалізацій.

Практичне значення одержаних результатів

Практичне значення одержаних результатів дослідження полягає у тому, що було розроблено корпоративний застосунок на основі мікросервісної та монолітної архітектур. Дані рішення дозволяють зрозуміти всі переваги і недоліки вибраної архітектури. З цього можна зробити висновок, що представлений у роботі підхід, надає розуміння, при яких бізнес-задачах слід використовувати один з цих архітектур, щоб в подальшому проект не став провальним, а був успішним, готовим до розширення, стійким до збоїв, масштабованим, тощо.

Апробація одержаних результатів

Результати дослідження були представлені на І всеукраїнській науково- практичній конференції здобувачів вищої освіти, аспірантів та молодих вчених «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України» [1], на науково-технічній конференції студентів, аспірантів, магістрантів і викладачів Інженерного навчально-наукового інституту Запорізького національного університету «Молода наука-2022» [2], на ІІ Всеукраїнській науково-практичній конференції «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України» [3] і на міжнародній науково-практичній конференції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету «Перспективи сталого розвитку в умовах глобалізації в економічному, управлінському та інженерному аспектах» [4].

Глосарій

Архітектура (англ. Architecture) — це структура компонентів, їхні взаємозв’язки, а також принципи та вказівки, що керують їх проектуванням та еволюцією з часом.

База даних (англ. Database) — сукупність даних, організованих відповідно до концепції, яка описує характеристику цих даних і взаємозв'язки між їх елементами; ця сукупність підтримує щонайменше одну з областей застосування, містить схеми, таблиці, подання, збережені процедури та інші об'єкти.

Веб-сайт (англ. Website) — сукупність вебсторінок та залежного вмісту, доступних у мережі Інтернет, які об'єднані як за змістом, так і за навігацією під єдиним доменним ім'ям, може розміщуватися як на одному, так і на кількох серверах.

Веб-сервер (англ. Webserver) — це сервер, що приймає HTTP-запити від клієнтів, зазвичай веб-браузерів, видає їм HTTP-відповіді, зазвичай разом з HTML-сторінкою, зображенням, файлом, медіа-потоком або іншими даними.

Виявлення сервісів (*англ. Service discovery*) — це патерн мікросервісної архітектури, що дозволяє спростити взаємодію між додатками, в умовах можливої зміни числа їхніх «інстансів» та мережевого розташування.

Система управління базами даних (англ. Database management system) — набір взаємопов'язаних даних (база даних) і програм для доступу до цих даних, надає можливості створення, збереження, оновлення та пошуку інформації в базах даних з контролем доступу до даних.

Клієнт (англ. Client) — це апаратний чи програмний компонент обчислювальної системи, що посилає запити серверу, взаємодіє із сервером, використовуючи певний протокол.

Мікросервіс (англ. Microservice)— це індивідуальний сервіс, який реалізовано з єдиною метою, що він буде самодостатнім і не буде залежати від інших мікросервісів.

Мікросервісна архітектура (англ. Microservice architecture) — архітектурний стиль, за яким єдиний застосунок будується як сукупність невеличких сервісів, кожен з яких працює у своєму власному процесі та спілкується з рештою, використовуючи прості та швидкі протоколи передачі даних, зазвичай HTTP.

Моноліт (англ. Monolith) — є додатком, що доставляється через єдине розгортання, таким є додаток, доставлений у вигляді однієї WAR (Java) або додаток Node (NodeJs) з однією точкою входу.

Централізоване конфігурування (*англ. Centralized configuration*) — це система, яка дозволяє читати дані конфігурації з файлу, який розташований в різноманітних джерелах, наприклад git-репозиторій.

API-Gateway — це шлюз, який організовує єдину точку доступу, для комунікації з мікросервісами, надає послуги, такі як моніторинг, захист, тощо.

Distributed tracing — розподілене трасування або трасування розподіленого запиту, призначений для того, щоб більш ефективно відстежувати та усувати збої, в ході спілкування між сервісами.

Docker — інструментарій для управління ізольованими Linux— контейнерами.

Docker-compose — це інструмент для визначення, розгортання та запуску багатоконтейнерних програм Docker.

Git — розподілена система керування версіями файлів та спільної роботи.

HikariCP — найшвидший пул з'єднань, який призначений для того, щоб проводити великовагові операції з базою даних, з великою кількістю запитів.

Flyway — інструментарій для супроводу баз даних і синхронізації їхньої структури з пов'язаним із нею програмним забезпеченням.

Java Spring — рівневий фреймворк для розробки Java/J2EE додатків, оснований на коді, опублікованому в “Expert One-on-One J2EE Design and Development” Родом Джонсоном, включає найповніший легкий контейнер,

який забезпечує централізовану, автоматизовану конфігурацію та з'єднання об'єктів.

JWT — це стандарт токена доступу на основі JSON, стандартизованого в RFC 7519, використовується для верифікації тверджень.

Kubernates — це інтрумент, який призначений для керування кластером контейнерів Linux як єдиної системи.

Kubectl — інтерфейс командного рядка, який призначений для управління командами Kubernates.

Load balancing — інструмент, який дозволяє розподілити вхідний трафік порівну між серверами, з метою підвищення ефективності та обробки.

Maven — популярний інструментарій створення корпоративних Java- проектів.

Minikube — спеціалізована конфігурація Kubernetes, призначена для розгортання на локальній машині, наприклад комп'ютера розробника, застосовується для вивчення та локальних експериментів над Kubernetes.

PostgreSQL — об'єктно-реляційна система керування базами даних.

RabbitMQ — платформа, що реалізує систему обміну повідомленнями між компонентами програмної системи на основі стандарту AMQP.

Spring Cloud — це проект, який дозволяє створювати розподілені програми з мікросервісною архітектурою.

Spring Cloud Sleuth — бібліотека, яка призначена для трасування логів, надає легко провести діагностику та знайти необхідну інформацію (логи).

Spring Security — це фреймворк Java/Java EE, що надає механізми побудови системи аутентифікації та авторизації, а також інші можливості забезпечення безпеки для промислових програм, створених за допомогою Spring Framework.

РОЗДІЛ 1 МОНОЛІТНА І МІКРОСЕРВІСНА АРХІТЕКТУРА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Огляд проблеми визначення архітектури

Проблема вибору архітектури для розроблюваного додатку є часто обговорюваною в середовищі розробників та системних архітекторів. Різні джерела по різному оцінюють ситуацію, приводячи як позитивні, так і негативні сторони використання монолітної та мікросервісної архітектури.

Чому проблема вибору архітектури так важлива для розробки? — В ході розробки та зростання проекту, взаємозалежностей між компонентами програми стає все більше, будь-який програмний код має залежні частини від інших, а класи вимагають наявності інших класів, з якими відбувається взаємодія, код складніше зрозуміти. Тому внаслідок цього, швидкість розробки зменшується, проект стає менш гнучким, масштабованим, зміна в одному компоненті призводить до змін по всьому проєкту.

Майбутня архітектура програмного забезпечення, повинна бути такою, щоб процес розробки і супроводження був швидким, простим та саме головне ефективним. Потрібно розуміти, що добре побудована архітектура на сам перед, це одна із запорук успіху. Адже, якщо потрібно буде розширювати, додавати новий функціонал, тестувати, налагоджувати або щось змінювати, то в подальшій розробці, це не складе жодних труднощів.

Розроблюване програмне забезпечення повинно бути побудовано таким чином, щоб архітектура, яка використовується в розробці, мала такі характеристики як:

1. Ефективність — тобто архітектура повинна вирішувати ті задачі, які були поставлені та виконувати свої функції.
2. Розширюваність — повинна бути можливість додавати нові функції та новий функціонал, при цьому, щоб не відбувалося ніякої зміни в

самій структурі програми. Рекомендується використовувати патерни проєктування.

3. Відмовостійкість — архітектура повинна, продовжувати свою роботу, якщо виникнуть помилки. Дана особливість, повинна забезпечувати деякі точки відновлення; логування даних, а саме, де і коли виникла проблема; якщо виникла серйозна проблема, то забезпечите її не розповсюдження на окремі компоненти.
4. Гнучкість системи — реалізувати майбутню систему таким чином, щоб з легкістю можна було вносити нові зміни до побудованого функціоналу без великих проблем і глобальних замін компонентів. Чим менше виникне проблем и помилок, при внесенні змін, тим гнучкіше система побудована і бути мати конкуренцію серед інших систем.
5. Масштабованість в процесі розроблення програмного забезпечення, повинна забезпечувати такі фактори, як термін розробки та розподілення процесу розробки при додаванні нових співробітників.
6. Безпека програмного забезпечення — повинна забезпечувати запобіганню будь-яких вразливостей, які можуть нашкодити та привести до повної поломки програми. Потрібно побудувати та реалізувати такі заходи, які можуть підвищити безпеку та зменшити вразливість та дефекти розроблюваних компонентів програмного забезпечення.
7. Збереження зворотньої сумісності, щоб в разі зміни версії програмного забезпечення, або комп’ютерного обладнання, програма продовжувала працювати та добре взаємодіяти з новими версіями. Нові зміни повинні бути більш пристосованими до попередніх версій програми.
8. Обслуговуваність — після випуску програмного продукту, навіть після ретельного тестування та розробки, можуть виникати деякі проблеми, несправності, дефекти. Тому потрібно забезпечувати

обслуговування програми, надавати покращення для неї та вирішувати отримані проблеми в ході використання.

9. Зручність використання, розроблений застосунок, який би він не був розмірів, повинен бути легким у використанні. Користувачі повинні без великих труднощів розуміти функціонал програми, як вона працює та займати на це все мінімальну кількість часу.
10. Тестованість — це один з важливих факторів, які потрібно проводити при певних етапах розробки та перед випуском програми в світ. Якщо архітектура була побудована правильно, то при тестуванні компонентів, вона буде мати менше помилок та працювати надійніше. Тестування є одним з важливих принципів та критеріїв, адже це надає інформацію про те, наскільки добре або погано, було побудовано поточне програмне забезпечення та його дизайн.
11. Можливість повторного використання. В даному випадку систему потрібно будувати таким чином, щоб в майбутньому можна було з легкістю повторно використовувати компоненти програми.
12. Гарно побудований та зрозуміло написаний код є одним з важливих факторів розробки архітектури. Будь-яка людина, чи присутня вона в розробці на даний момент, чи буде присутня в майбутньому, повинна з легкістю читати та розуміти, що відбувається в будь-якому написаному фрагменті коду. Розроблюваний проєкт повинен: добре структурованим; мати особливість швидкого розуміння тих чи інших компонентів; не містити дублювання коду; мати підписаний коментар до змінних, функцій, тощо; по можливості, мати стандартні загальноописані рішення, тощо.

1.2 Аналіз монолітної та мікросервісної архітектур

В 20 столітті Едсгером Дейкстри і Девіда Парнаса, в науковій роботі було покладено початок архітектури програмного забезпечення як основа для

побудови різних програм, починаючи від маленьких програм зі словами «Hello World» до великих підприємницьких програм, які виконують важливу роботу. Дані вчені говорять, що правильно побудована структура програмного забезпечення відіграє велику роль в майбутній розробці.

Основними функціями правильної побудованої програмної архітектури є те, що потрібно, як можна менше мінімізувати складність системи, зрозуміти, як краще потрібно зробити розбиття на частини великих компонентів, як правильно побудувати взаємодію між ними, як буде відбуватися обмін інформацією між цими компонентами, чи можна буде додатково розширити їх без великої зміни в інших, чи можемо ми це зробити за допомогою абстрагування та розділення задач, тощо. Архітектура повинна мати задовольняти поставлені вимоги, виконувати чітко свої бізнес-задачі, використовувати певні архітектурні стилі, різні шаблони проєктування (наприклад, шаблон «Модель-Вигляд- Контролер»), які полегшують розробку та роблять систему більш гнучкою, та масштабованою. Адже, якщо до кінця не буде правильно продумано задовільного та обгрунтованого рішення, то в наслідок цього, розробка може «затягнутися» та призвести до великих змін в вже побудованій системі.

Потрібно визначити, який архітектурний шаблон треба використовувати, щоб побудувати програмне забезпечення, яке буде стійким до збоїв, готовим до розширення, мати високий рівень безпеки, масштабованим, надійним та конфігурованим. Вибрати базу даних, яка буде в змозі обробляти достатню кількість запитів від клієнта. Сайт повинен мати стильний дизайн, бути зрозумілим для користувача і мав змогу зацікавлювати більше нових клієнтів.

Одними із популярних шаблонів проєктування в програмуванні є монолітна та мікросервісна архітектури. Окрім аналізу і формулювання вимог, дані архітектури надають міцний фундамент для розробки програмного забезпечення.

В процесі розробки нового додатку, потрібно розуміти, який з даних архітектурних шаблонів підходить для конкретної предметної області, бізнес- задач тощо. В кожній з цих архітектур є свої переваги та недоліки, адже поки не існує одного гарного рішення для всіх задач.

Монолітна архітектура (рис.1) — це традиційна уніфікована модель розробки програмного забезпечення. Монолітний у цьому контексті означає складений з одного шматка. Згідно з Кембриджським словником, прикметник «монолітний» також означає «занадто великий і не піддається зміні» [5]. Він являється деяким додаток, розгортання якого, здійснюється наприклад в вигляді JAR-файлу або Node з єдиною точкою входу. При побудові додатка на основі моноліта, всі процеси будуть управлятися всередині одного модуля.

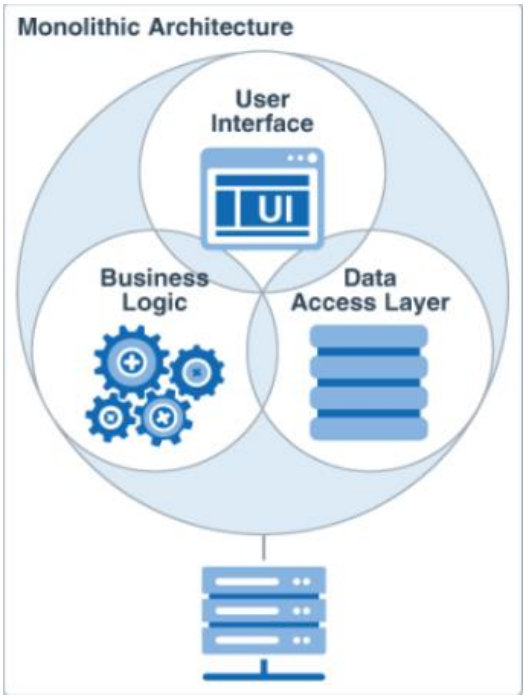


Рис.*1 Монолітна архітектура*

Монолітна архітектура будується на основі клієнт-сервер, в якій функції презентації, обробки програм та управління даними фізично розділені. Рівень презентації відображає різну інформацію, на прикладі застосунку доставки, це може бути різні партнери, послуги, продукти, які доступні на цьому сайті. Рівень обробки програм займається бізнес-логікою, яка в свою чергу

призначена для контролю функціональності програми, щоб виконувати детальні обробки. Рівень управління даними призначений для розміщення серверів баз даних, де в свою чергу зберігається інформація. Монолітний підхід є стандартною моделлю побудови та створення програмного забезпечення. Але, його тенденція йде на спад, оскільки створення «моноліту» надає певну кількість проблем, пов’язаних із обробкою величезної кодової бази, впровадженням нових технологій, розгортанням, додаванням нових змін тощо.

Моноліт спроєктований таким чином, щоб бути автономним, компоненти програми взаємопов’язані та взаємозалежні. У такій архітектурі кожен компонент та всі сервіси, реалізовані за допомогою одного набору технологій (і мови програмування), і використовують загальні бібліотеки коду. Всі сервіси працюють з одним сервером баз даних, що дозволяє кожному сервісу звертатися до бази даних безпосередньо.

Переваги:

1. Легко реалізувати. Даний підхід є стандартним способом створення застосунків, тому, будь-яка команда може мати необхідні знання та можливості для розробки застосунків на основі монолітної архітектури.

2. Простий в розгортанні. В кінці розробки монолітної системи не потрібно працювати з багатьма розгортаннями — лише один файл або каталог.

3. Високий рівень безпеки.

4. Розробникам не потрібно вивчати різні програми, адже можуть зосередитися лише на одній програмі.

5. Проблеми затримки мережі та безпеки відносно менші.

6. Простіше тестування та налагодження. На відміну від мікросервісної архітектури, монолітну — набагато легше налагоджувати та тестувати. Оскільки вона є єдиною та неподільною і може запускати наскрізне тестування набагато швидше.

7. Конфігурованість.

8. Меньша кількість наскрізних проблем. Наскрізні проблеми — це проблеми, які впливають на все програмне забезпечення, наприклад введення журналу, кешування, обробка, моніторинг продуктивності, тощо. Дана область функціональності стосується лише поточного одного програмного застосунку, тому і легше працювати з цією системою.

Недоліки:

1. Все ядро працює в одному і тому ж самому адресному просторі, тобто збій в одному з компонентів програми, порушить працездатність всієї системи.

2. Створення нового додаткового функціоналу може вплинути на деяку зміну в інших компонентах.

3. Зі збільшенням розмірів застосунків, збільшується час запуску та розгортання.

4. Масштабування програми може бути складним. Кожна копія екземпляра програми матиме доступ до всіх даних, що робить збільшення споживання пам’яті, а кешування менш ефективним. Зі збільшенням обсягу даних, архітектура можливо не зможе масштабуватися.

5. Ознайомлення з великою кількістю коду, може займати великий час як серед розробників, так і серед «новачків».

6. Систему, можливо буде важко розуміти та робити зміни, що в наслідок цього, розвиток програмного забезпечення сповільнюється, якість коду знижується, модульність з часом руйнується.

Мікросервісна архітектура — це архітектурний шаблон, в якому застосунок будується за рахунок невеликих сервісів, кожен з яких працює у своєму власному процесі та спілкується з іншими, використовуючи прості та швидкі протоколи передачі даних HTTP (рис.2). Ці послуги будуються з урахуванням бізнес-можливостей і можуть бути незалежно розгорнуті за допомогою повністю автоматизованого механізму розгортання. Існує мінімум централізованого управління цими службами, які можуть бути написані різними мовами програмування та використовувати різні технології зберігання даних [6]. Ключові концепції, які потрібно використовувати при

розробці мікросервісної архітектури — це декомпозиція та розгрупування. Функціональність додатків має бути повністю незалежною одна від одної.

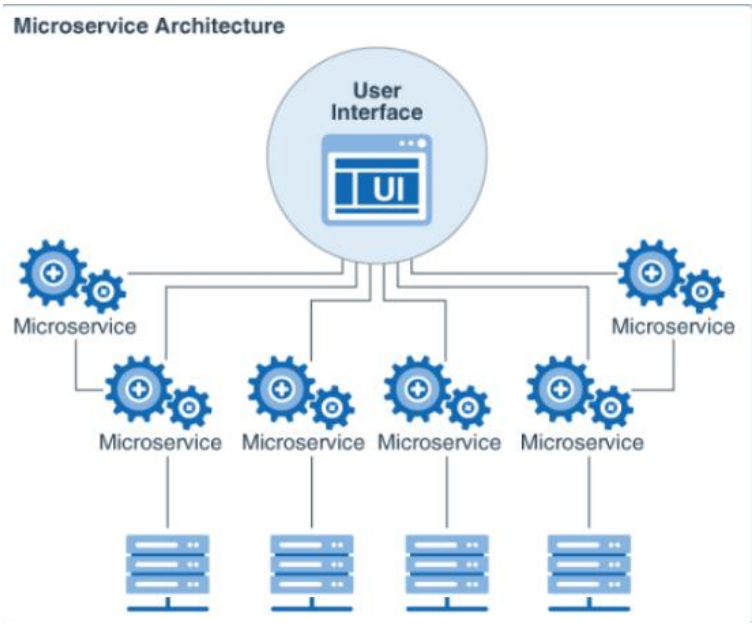


Рис.*2 Мікросервісна архітектура*

Мікросервісна архітектура має суттєвий вплив на зв’язок між застосунком і базою даних. Замість використання спільної бази даних з іншими мікросервісами, кожен з них має свою власну. Часто це може призвести до дублювання даних. Але будування бази даних на кожний мікросервіс є важливим аспектом, якщо потрібно мати вигоду від даної архітектури, та, щоб вона забезпечувала слабкий зв’язок. Ще одна перевага наявності окремої бази даних, полягає в тому, що кожен сервіс може використовувати тип бази даних, який найкраще підходить для його потреб. Кожен сервіс пропонує безпечну границю якогось модуля, щоб інші мікросервіси, могли бути написані на різних мовах програмування. Архітектура мікросервісу містить багато шаблонів, наприклад виявлення та реєстрація служб, кешування, API-Gateway, безпека тощо.

Існують важливі принципи, які потрібно обов'язково дотримуватися в архітектурі мікросервісів:

1. Принцип єдиної відповідальності, один із принципів шаблону проектування SOLID. В ньому йдеться мова про те, що кожен компонент, підрозділ, клас, метод або мікросервіс повинен відповідати за одну і тільки одну задачу. Кожен мікросервіс повинен мати єдину відповідальність і функціональність. Можна сказати, що: певна кількість мікросервісів, які повинні розробитися, дорівнює кількості необхідних функцій. База даних — децентралізована, і, як правило, кожен сервіс має власну базу.

2. Створення сервісів за їх бізнес-можливостями. Зараз в світі існує велика кількість технологій, і кожна з них, може як найкраще підходити для реалізації тієї або іншої задачі. В монолітних застосунках така характеристика має серйозний недолік, адже ми не можемо використовувати різні технології. Мікросервіс ніколи не обмежується, якимось відповідним технологічним стеком або сховищем бази даних. Тут використовується всі ті технології, які найбільше підходять для вирішення бізнес-задач.

3. Розробка на випадок збою. Мікросервісна архітектура повинна бути розроблена та побудована з урахуванням випадків збою. Вихід з ладу одного сервісу, не повинен впливати на інший сервіс або систему в цілому. Функції повинні залишатися доступними.

Сервіс зазвичай реалізує набір окремих функцій, наприклад управління замовленнями, управління клієнтами, тощо. Кожний мікросервіс є міні-додатком зі своєю власною шестикутною архітектурою, що складається з бізнес-логіки та різних адаптерів. Деякі мікрослужби можуть надавати API, який використовується іншими мікросервісами або клієнтами програми. Інші мікросервіси можуть надавати веб-інтерфейс. Під час виконання кожен екземпляр часто є хмарною віртуальною машиною або контейнером Docker [7].

Переваги:

1. Кожен компонент розгортається незалежно від інших.

2. Мікросервіси використовують прості протоколи зв’язку, такі як HTTP і JSON для обміну даними між сервісами.

3. Логіка програми розбивається на невиликі компоненти з чітко визначеними межами відповідальності.

4. Масштабованість. Система здатна обробляти великий обсяг роботи та бути легко розширеною. Якщо певний мікросервіс стикається з великим навантаженням через те, що клієнти використовують його в надлишку, тоді потрібно масштабувати лише цей мікросервіс. Це означає те, що мікросервісна система підтримує горизонтальне масштабування.

5. Висока стійкість до відмов: збій в одному сервісі не вплине на інші. Таким чином, проблеми в інших сервісах, не завадять всій робочій системі.

6. Простота — через невеликий розмір сервісу, коду в ньому менше, і тому не потрібно розбиратися у великій кількості деталей реалізацій, що в результаті дає розробникам меньше часу для того, щоб розібратися, як система працює.

7. В кожному сервісі можна вибрати або замінити стек технологій.

8. Краще організовані. Через їх незалежний і розподілений характер, вони дозволяють організаціям мати менші групи розробників із чітко визначеними сферами відповідальності.

Недоліки:

1. Складність створення розподіленої системи. Дана складність зростає зі збільшенням кількості мікросервісів.

2. Обмін повідомленнями між сервісами має складний характер. Тому що, кожен функціональний елемент ізольований. За рахунок цього потрібна особлива ретельність при побудові комунікації між ними. Зі збільшенням кількості сервісів, складність побудови комунікації зростає.

3. Зростання кількості сервісів, несе за собою характер зростання кількості баз даних. Тому, що мікросервісна архітектура повинна дотримуватися патерну «Одна база даних на сервіс». Якщо для кожного створеного сервісу використовувати одну й ту ж саму базу, це призведе до використання анти- шаблону, а отже, призведе до поганої побудови архітектури.

4. Складність розгортання. Самостійне розгортання мікросервісів є складним.

5. Складність тестування — спочатку потрібно розібратися з роботою кожного сервісу, а згодом, проводити тестування взаємодії його з іншими мікросервісами.

6. Для використання всередині окремих організацій, вони не дуже підходять, так як, можуть виявитися складними у використанні.

7. Мають меншу безпеку порівняно з монолітними застосунками. Це по'вязано з тим, що зв’язок між сервісами відбувається через мережу.

Кожна з цих архітектур є потужною складовою в розробці програмного забезпечення. Вони мають ряд, як позитивних, так і негативних харакетристик, які можуть вплинути на майбутню розробку. Після ознайомлення та вивчення, можна порівняти дані архітектури за такими характеристиками:

1. Простота:
    a. В випадку моноліту — простий в реалізації, управлінні, розгортанні, тестуванні та вимагає меншої підготовки команди розробників.
    b. В випадку мікросервісів — вони потребують ретельного управління, оскільки розгортання додатків відбувається на різних серверах і в свою чергу, використовують API.
2. Доступність:
    a. В випадку моноліту — при виході з ладу якогось компоненту, може постраждати вся система.
    b. В випадку мікросервісів — компоненти пов’язані не дуже жорстко між собою, бо являють собою самостійні та незалежні одне від одного елементи.
3. Кросплатформеність:
    a. В випадку моноліту, то він повністю являється залежним від конкретної бази даних, мови програмування, платформи.

b. В випадку мікросервісів — в них є можливість повністю використовувати різні бази даних, технології, середовища відповідні їхнім бізнес-процесам.

4. Заміна комплектуючих:
    a. В випадку моноліту, зміна одного компоненту, може понести за собою зміну і в інших компонентах програми, що може призвести до затримки та швидкодії.
    b. В випадку мікросервісів, зміна в одному мікросервісі компонентів, не веде за собою зміну в інших.

1.3 Аналіз принципів та патернів, які використовуються в монолітній та мікросервісній архітектурах

З початку першого розроблюваного програмного забезпечення прошло багато часу. За цей період, програмісти та архітектори все більше зіштовхувалися з проблемами, які призводять до того, що проєкт не міг бути масшатабованим, а додавання нових функцій та переробка компонентів програми, займало багато часу. Тому, програмісти і не тільки, задумалися та винайшли деякі принципи та патерни, які можуть покращити розроблюване програмне забезпечення.

Одними з принципів, які рекомендують використовувати в розробці програмного забезпечення — це SOLID-принципи [8]. На даний момент найбільш популярні це: принцип єдиного обов'язку, тобто кожен об'єкт має виконувати лише один обов'язок; принцип відкритості/закритості — програмні сутності повинні бути відкритими для розширення, але закритими для змін; принцип підстановки Лісков — об'єкти в програмі можуть бути заміненими їх нащадками без зміни коду програми; принцип розділення інтерфейсу — багато спеціалізованих інтерфейсів краще за один універсальний; принцип інверсії залежностей — залежності всередині системи

будуються на основі абстракцій, що не повинні залежати від деталей; навпаки, деталі мають залежати від абстракцій.

В моноліті та мікросервісах вище згадані принципи рекомендують використовувати. Але, окрім цього, додатково в мікросервісах існує певна кількість патернів, які широко застосовуються в розробці. Одними з них являються:

1. Decompose by subdomain — даний патерн використовується для розкладання монолітів. Цей підхід розбиває один великий домен на окремі субдомени (сервіси), кожний із яких має своє завдання та обов'язок [9]. Він підходить для готових монолітних корпоративних застосунків, які мають чітко визначені межі між модулями, які в свою чергу, пов'язані з піддоменами. Тобто даний патерн надає можливість розкладання моноліту без значного переписування існуючого коду. Субдомени класифікують таким чином:
    a. Основні — ключова відмінність, являється цінною частиною програмного застосунку.
    b. Допоміжні — дана відмінність пов'язана з тим, чим займається бізнес.
    c. Загальні — ідеально реалізуються за рахунок готового застосунку.

Даний шаблон передбачає, що побудована мікросервісна архітектура, буде мати слабкі зв'язки між компонентами, буде забезпечувати розширюваність, стійкість, масштабованість, ремонтопридатність, тощо. Але, він має деякий недолік — це велика кількість мікросервісів, що в наслідок, може ускладнювати їх пошуки та інтеграцію.

2. Decompose by business capability — даний патерн надає нам можливість розкладання мікросервісів за їх бізнес-задачами. Він використовується тоді, коли, кожен мікросервіс повинен відповідати за свою функціональність. Наприклад, мікросервіс «Category» — повинен відповідати тільки за дані, які стосуються категорій, тощо.

В даному патерні є свої переваги, він створює та надає стабільну архітектуру, мікросервіси слабко пов'язані, складність впровадження нових

розробників до розробки — мала, так як, мікросервіс відповідає тільки за єдиний обов’язок і побудована структура сервісу є простою в розумінні та розробці. Недоліки даного шаблону, є такі, що сам дизайн програмного забезпечення тісно пов’язаний з побудованою та готовою бізнес-моделлю і вимагає повне розуміння бізнес-можливостей застосунку.

3. Database per service — хорошою практикою є використання баз даних для кожного окремого сервісу. Наприклад, сервіс замовлень повинен зберігати інформацію про замовлення в свою базу даних, а інший мікросервіс в іншу.

Використовуючи даний шаблон, не рекомендується, щоб мікросервіси мали прямий доступ до баз даних інших сервісів. Дані повинні бути конфіденційними та доступними лише через API конкретного сервісу. Транзакції служби включають лише її базу даних [9].

Існує декілька способів зберігання даних мікросервісів. За рахунок них, нам не потрібно надавати, кожний раз сервер бази даних для кожної служби. Якщо буде використовуватися реляційна база даних, наприклад MySQL, Oracle DB, PostgreSQL, то можливі такі варіанти:

- Приватні таблиці на сервіс, що говорить про те, що даний сервіс володіє певним набором таблиць, доступ до яких має лише служба, якій було призначене дане налаштування.
- Схема, для кожного сервісу — кожна служба може мати схему бази даних, яка буде приватною для цієї служби.
- Сервер бази даних на сервіс — кожна служба буде мати свій налаштований сервер бази даних.

Приватні таблиці та схема на сервіс мають найменші накладні витрати. Деяким високопродуктивним мікросервісам може знадобитися власний сервер бази даних.

Даний патерн надає такі переваги: кожен мікросервіс використовує тільки ту базу даних, яка найкраще відповідає її потребам та бізнес-задачам; мікросервіси слабо пов’язані; зміни компонентів одного сервісу не впливають на інші. Але патерн має і такі недоліки: бізнес-транзакції, які охоплюють

декілька послуг, тяжко підтримувати та реалізовувати, бо в наслідок цього, можна зіткнутися з проблемами CAP-теореми. Крім того, багато сучасних NoSQL-баз даних можуть не підтримувати їх. Реалізація запитів, які об’єднують велику кількість даних, що в свою чергу знаходяться в декількох базах даних, є складним і потребують ретельних перевірок. Також, може виникати така проблема, що при використанні декількох різних баз даних, керування буде мати складний характер.

Існують деякі рішення, які призначені для реалізації запитів та бізнес- транзакцій, які охоплюють деяку кількість мікросервісів:

- Патерн Saga.
- Композиція API — виконує об'єднання запитів, а не баз даних. Наприклад, клієнт хоче отримати свої персональні дані та замовлення. Спочатку буде подано запит на отримання даних поточного клієнта з мікросервісу «Users», згодом відбудеться запит на отримання замовлень клієнта до мікросервісу «Order», і клієнт отримає всі ці дані.
- Розподіл відповідальності за командний запит (CQRS).

4. Saga — шаблон, який призначений для отримання бізнес-транзакції, які охоплюють декілька сервісів. Saga — це послідовність локальних транзакцій. Кожна транзакція оновлює базу даних і публікує повідомлення або подію, щоб ініціювати наступну локальну транзакцію в сагі. Якщо вона не порушує бізнес-правила, тоді сага виконує серію компенсуючих транзакцій, які скасовують зміни, внесені попередніми локальними транзакціями [9].

Існують два способи узгодження послідовності локальних операцій:

1. Хореографія, яка говорить про те, що кожна транзакція буде публікувати події домену, а ті в свою чергу, будуть запускати локальні транзакції в інших мікросервісах.

2. Оркестрування, являє собою такий механізм, який включає в себе управління подіями між окремими послугами.

При розробці монолітної архітектури, використання шаблону Saga буде відігравати важливу роль в реалізації, адже він буде надавати потужний механізм підтримки узгодженості даних у кількох мікросервісах. Але при його використанні, потрібно знати, що кожен сервіс, який використовує і налаштований під даний патерн, повинен оновлювати свою базу даних та повідомляти інші сервіси, публікуванням подій або повідомлень.

1.4 Аналіз перебудови монолітної до мікросервіної архітектури

Кожне успішне розроблене програмне забезпечення, яке являється популярним та широко використованим в світі, може потребувати набагато більше функціоналу. Компанія, яка розроблювала даний проєкт, може вирішити розділити моноліт на мікросервіси. Для успішного переходу від монолітної до мікросервісної архітектури, рекомендують декілька етапів: моноліт, мікросервісний моноліт, мікросервіси, оркестрація або хореографія бізнес-сервісів [10].

Етап 1. Програмне забезпечення, яке побудовано на основі монолітної архітектури. Поточний етап можна описати таким чином — це початкова точка розробки та розгортання. Застосунок вже побудований та знаходиться в готовому вигляді, він має свою базу даних та виконує всі ті бізнес-задачі, які було обговорено в ході розробки. Але даний етап має проблеми. Вся система побудована в єдиному екземплярі, її складно масштабувати, виконує велику кількість бізнес-задач і потребує ще більше. Для додавання нового функціоналу потрібно «підстраюватись» під стек, тих технологій, на яких побудована дана архітектура. При розгортанні всього застосунку, система повинна бути повністю протестована, що в свою чергу призводить до уповільнення швидкості релізів, бо велика кількість часу йде на вирішення проблем та «багів», які були виявлені в ході тестування. Для того, щоб перейти на наступний етап, потрібно виділити бізнес-задачі в окремі сервіси, при цьому потрібно використовувати принцип, який говорить про те, що кожен

сервіс повинен виконувати свою задачу, і в ході змін не впливати на роботу інших компонентів.

Етап 2. Мікросервісний моноліт, всі частини моноліту стали незалежними мікросервісами, і вони повинні підтримувати комунікацію між собою. В цьому випадку, всі частини моноліту розпалися на мікросервіси, і їх об’єдинили павутиной синхронних та асинхроних інтеграцій. Але зарахунок цього виникають проблеми. Утворені зв’язки між мікросервісами ускладнюють аналіз виниклих проблем. Наприклад, ми подаємо запит на отримання персональних даних, замовлень та яким чином були оплачені ці замовлення. Даний запит повинен пройти через декілька мікросервісів і в ході отримання замовлення він зупинився. І виникає питання, що ж все таки сталося при отриманні даних. Архітектура, яка склалася на даному етапі не є ефективною, а навпаки, її тепер все більш складно розуміти.

Для того, щоб перейти на наступний етап, потрібно використовувати готові технології, які допоможуть полегшити та оптимізувати роботу майбутнього мікросервісного застосунку. Одними з популярних технологій, які слід використовувати це:

1. API-Gateway, який буде використовуватися для локалізації синхронних взаємодій, буде надавати моніторинг, логування, трасування даних будь-якого трафіку, який буде відбуватися між мікросервісами.
2. Service Discovery, який призначений для відстеження працездатності мікросервісів та, який буде перенаправляти трафік на інші робочі екземпляри.
3. Інші додаткові технології, наприклад, Circuit breaker, Distributed Tracing, тощо.

Етап 3. Мікросервіси. Вони нічого не знають одне про одного, працюють зі своєю базою даних. API та повідомленнями обмінюються за допомогою Kafka, RabbitMQ, тощо. Кожен мікросервіс вирішує тільки одну бізнес-задачу і, вирішує її чітко, за рахунок вибору різних технологій та стратегій

масштабування. На цьому етапі більшість компаній не йдуть далі, тому що, бізнес- задачі вирішуються достатньо швидко і ефективно.

Етап 4. Оркестрація або/та хореографія. В бізнес-сервісах зазвичай є візуальною платформою, де з'єднуються сервіси, виставляються тригери та умови розгалуження, контролюються всі потоки даних: реалізовано трасування запитів, логування подій, автомасштабування за умовами. Самі вони нічого не знають про специфіку бізнес-процесів.

1.5 Аналіз існуючих рішень, які побудовані на основі монолітної та мікросервіної архітектур

1.5.1 Рішення від компанії Glovo

Glovo — іспанська компанія [11], яка пропонує сервіс доставки через мобільний додаток (рис.3). Станом на липень 2019 компанія оперує у більш, ніж 20 країнах. Клієнти роблять замовлення будь-якого товару, невеликого розміру, вагою до 9 кг. Переважно це може бути їжа, ліки, документи, якась додаткова допомога, тощо. У Європі, наприклад, на 85% складають замовленння покупок та доставки їжі. Після оформлення замовлення, клієнт має можливість відстежувати переміщення кур'єра на маршруті в режимі реального часу. Даний додаток побудований на основі монолітної архітектури. Веб-застосунок побудований на основі Vue.js-фреймворка. Також даний сервіс доставки побудований на мовах програмування Swift, Ruby та Kotlin, що в свою чергу надає можливість використання його на пристроях IOS та Android.

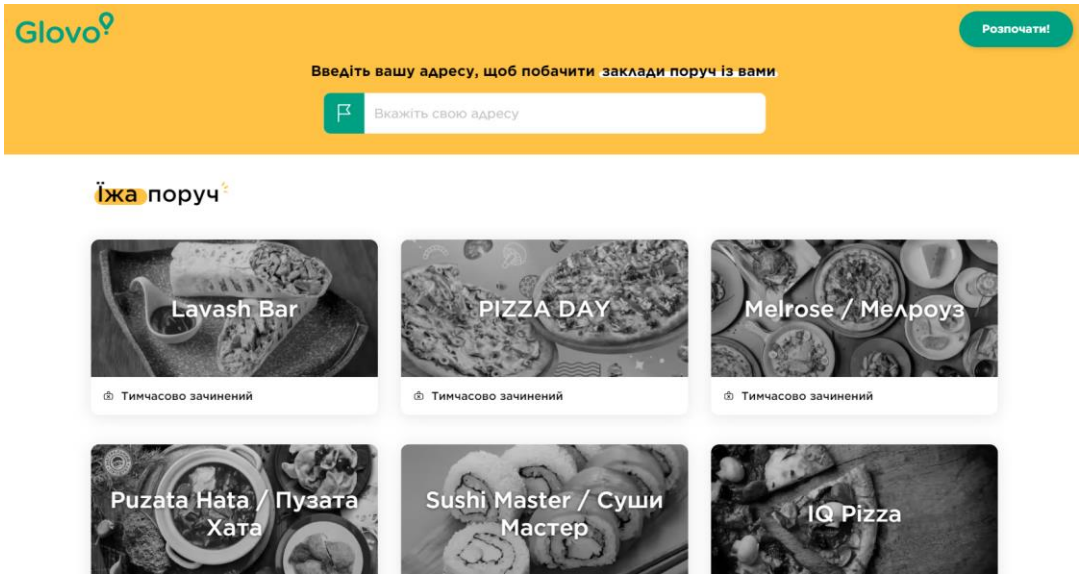


Рис.*3 Рішення від компанії Glovo*

1.5.2 Рішення від компанії Netflix

Netflix — це сервіс трансляцій на основі підписки [12], який дає користувачам змогу переглядати серіали й фільми без реклами на пристроях, підключених до Інтернету. Netflix побудований на основі мікросервісної архітектури. Він працює в двох хмарах: Amazon Web Services и Open Connect (Netflix content delivery network). Backend включає сервіси, бази даних та сховища, що повністю працюють у хмарі AWS. Backend переважно обробляє все, крім потокового відео. Компоненти «бека» з відповідними сервісами AWS наведені нижче:

- Масштабовані обчислювальні екземпляри: AWS EC2.
- Масштабована пам'ять: AWS S3.
- Микросервіси для бізнес-логіки: фреймворк створений командою Netflix.
- Масштабовані та розподілені бази даних: MySQL, Cassandra.
- Робота з обробкою великих даних та аналітикою: AWS EMR, Hadoop, Spark, Flink, Kafka.

Рис.*4 Рішення від компанії Netflix*

1.6 Висновки з розділу 1

В даному розділі було проведено огляд проблеми визначення архітектури; аналіз переваг та недоліків, а також принципів і патернів монолітної та мікросервісної архітетури; аналіз існуючих рішень. На основі цього, можна зробити та сформулювати наступні висновки:

1. Визначення відповідної архітектури відіграє важливу роль в розробці програмного забезпечення. Даний аспект надає неабиякий успіх в розроблюваній системі та проекті в цілому.

2. Одними з популярних архітектурних шаблонів вважаються монолітна та мікросервісна архітектури. Вони мають ряд позитивних та негативних характеристик, які відіграють важливу роль в розробці застосунку.

3. Мікросервіси вважаються більш трудоємкими, ніж моноліт, тому більшість фахівців рекомендують спочатку використовувати монолітну архітектуру.

4. Проведено аналіз принципів та патернів, призначених для монолітної та мікросервісної архітектур. Вони вважаються популярними у використані серед розробників програмного забезпечення.

5. Проведені та перевірені етапи переходу від монолітної архітектури до мікросервісної і визначено в чому їх призначення.

6. Проаналізовано готові рішення, які побудовані на основі монолітної та мікросервісної архітектури.

РОЗДІЛ 2 МІКРОСЕРВІСИ З SPRING BOOT І SPRING CLOUD

2.1 «Spring Framework» та «Spring Cloud» — інструменти для побудови мікросервісної архітектури

Spring Framework — це величезний набір бібліотек і інструментів, які спрощують розробку програмного забезпечення за рахунок впровадження залежностей. Вони надають доступ до даних, перевірку, інтернаціоналізацію, аспектно-орієнтоване програмування, тощо [13]. Це популярний вибір для Java-проєктів. Вона працює з іншими мовами на основі JVM, такими як Kotlin і Groovy.

Основні особливості Spring Boot полягають в тому, що він дозволяє створювати повноцінні додатки, має вбудований сервер Tomcat, автоматично конфігурує систему, тощо. Spring Framework дуже популярний у використанні серед розробників, бо він економить багато часу, надаючи вбудовані реалізації для багатьох аспектів розробки програмного забезпечення, таких як:

1. Spring Data, яка спрощує доступ до даних для реляційних і NoSQL баз даних.

2. Spring Batch, який забезпечує потужну обробку великих обсягів записів.

3. Spring Security , структура безпеки, яка абстрагує функції безпеки до додатків.

4. Spring Cloud, надає розробникам інструменти для швидкого створення деяких загальних шаблонів в розподілених системах.

5. Spring Integration, це реалізація корпоративної інтеграції. Вона полегшує інтеграцію з іншими корпоративними програмами, за допомогою використання повідомлень і декларативних адаптерів.

Причина вибору даного фреймворку полягає в тому, що він полегшує та надає готові методи та рішення, наприклад, як використання інтерфейсів замість класів для відокремлення рівнів додатків за допомогою ін’єкції

залежностей. Наявність великої кількості доступних модулів та інших пов’язаних сторонніх бібліотек, які можна поєднати з фреймворком, показує нам те, що він являється потужним механізмом для розробки програмного забезпечення. Він усуває більшу частину процесу самостійного налаштування, надаючи конфігурації за замовчуванням та інструменти, які автоматично налаштовуються.

Spring Boot надає деякі попередньо визначені стартові пакети, сторонні бібліотеки і інструменти. Наприклад, «spring-boot-starter-web» допоможе створити веб-застосунок, або бібліотека Jackson, яка призначена для обробки JSON-даних.

Spring Cloud — це інструмент, який дозволяє створювати загальні шаблони у розподілених системах (configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state) [14]. Дані системи надають бібліотеки, за допомогою яких, розробник без проблем може швидко та якісно створити сервіси та надати їм налаштування.

Особливості Spring Cloud заключаються в тому, що він надає такі механізми, які дозволяють легко і швидко налаштувати простий мікросервіс. Одними з механізмів є розподілена конфігурація, маршрутизація, реєстрація сервісу, спілкування між мікросервісами, балансування навантаження, автоматичні перемикачі, захист даних, тощо.

2.2 Мікросервісна архітектура на сучасному стеку Java-технологій

Для побудови застосунків на основі мікросервісної архітектури, існує стандартний стек технологій, який є популярним серед багатьох розробників. В ході виконання дипломної роботи, було побудовано мікросервісну архітектуру на основі даних нижче технологій:

1. API-Gateway — це інструмент керування, який розташований між клієнтом і набором сервісів. Простими словами — це єдина точка доступу.

Шлюз API діє як проксі-сервер, який приймає всі запити та маршрутизує їх на різні сервіси і повертає відповідний результат [15].

API-Gateway, являється деяким посередником між клієнтом та запрошуваним сервісом. Клієнт спілкується лише з однією URL-адресою, якою керує шлюз. API-gateway розділяє запити, що надходить від клієнта та визначає на який саме сервіс він повинен надійти.

Мікросервіс побудований на основі цього патерну, надає маршрутизацію запитів до інших мікросервісів. Також він надає додаткові налаштування, наприклад такі, як:

1) безпека (автентифікація) — перевіряє запити клієнтів, які пройшли автентифікацію, згодом розподіляє та надсилає їх до конкретних мікросервісів;

2) моніторинг — використовується для збору показників і інформації під час проходження запиту через сервіс; також, використовується, для того, щоб можна було підтвердити наявність критично важливих фрагментів інформації про запити користувачів, таким чином забезпечуючи ведення журналу; дозволяє централізувати збір основних показників, наприклад, кількість викликів та час відповіді сервісів;

3) балансування навантаження;

4) управління запитами;

5) кешування;

6) маршрутизація.

Побудувати маршрутизацію можна за допомогою yaml-файлу (файл налаштувань) або за допомогою java-конфігурацій.

В ході виконання дипломної роботи використувалася бібліотека Spring Cloud Gateway. Даний інструмент є дуже популярним серед розробників, так як він, є неблокуючим, тобто це означає, що ніколи не блокуються основні потоки. Вони є завжди доступними для запитів на сервіси і обробляють їх асинхронно у фоновому режимі, щоб згодом, повернути відповідь після

завершення обробки. Spring Cloud Gateway пропонує кілька можливостей, зокрема:

1) зіставлення маршрутів для мікросервісів. Spring Cloud Gateway не обмежується однією URL-адресою. За допомогою нього можна визначити кілька точок входу маршруту, тобто кожна кінцева точка сервіса, отримує власне відображення маршруту. Але найпоширенішим варіантом використання, є створення єдиної точки входу, через яку проходитимуть усі запити до сервісів;

2) використання фільтрів, які можуть перевіряти та реагувати на запити та відповіді, що надходять через шлюз. Ці фільтри дають змогу управляти та виконувати велику кількість дій над усіма викликами мікросервісів. Іншими словами, ці фільтри дозволяють нам змінювати вхідні та вихідні запити та відповіді HTTP;

3) будування предикатів, вони являються деякими об’єктами, які призначені для перевірки запитів на відповідність певному набору заданих умов перед виконанням або обробкою. Spring Cloud Gateway містить набір вбудованих фабрик предикатів маршрутів;

4) інтегрує автоматичний вимикач, в якому можна налаштувати конфігурацію, яка у випадку помилки або довгого часу очікування запиту, відправить fallback-відповідь клієнту з повідомленням помилки;

5) має інструменти обмеження швидкості запитів — це потрібно для того, щоб позбавитися великої кількості запитів від клієнта. Відповідь запиту в мікросервісній архітектурі може трішки запізнитися, тому потрібно на певний час обмежити кількість запитів.

На рисунку 5 зображено налаштований паттерн API-Gateway за домогою Spring Cloud Gateway. Для побудови перенеправлення запитів на інші мікросервіси, в дипломній роботі, використовувався за рахунок java- конфігурації. Наприклад, клієнт подає запит GET на отримання даних оплати на адресу «http://localhost:9191/api/payment», api-gateway приймає його, та починає зіставляти запит з налаштованими маршутами. Якщо запит

відповідає маршруту, він надсилається далі до веб-обробника шлюзу. Цей, в свою чергу, пропускає запит через налаштовані фільтри. Коли запит, пройшов всі перевірки, шлюз перенаправляє на конкретний мікросервіс з шляхом «/api/payment», де в кінцевому варіанті, він повертає клієнту всі дані по оплаті замовлень.

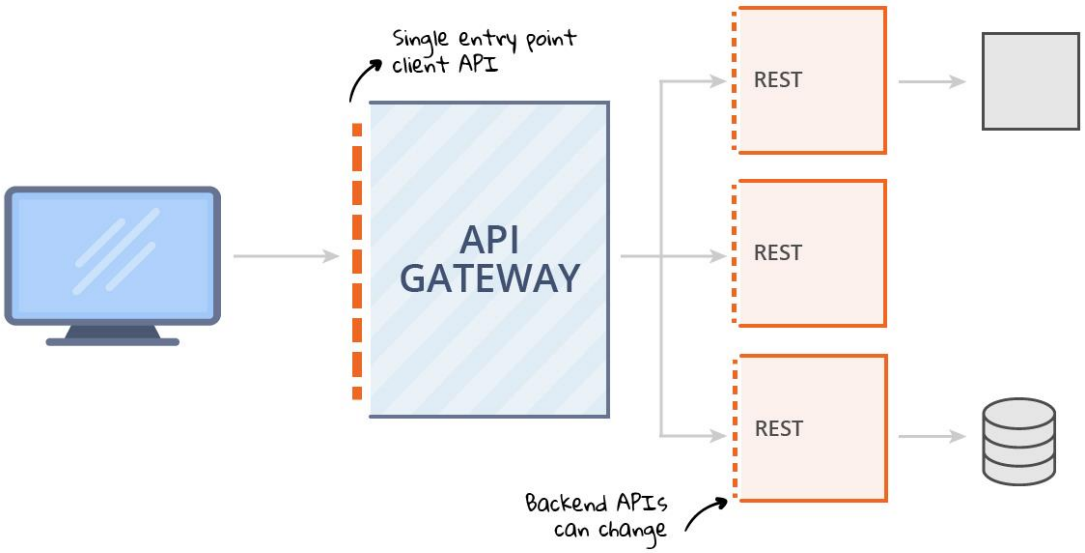


Рис.5 *Паттерн API-Gateway*

2. Service Discovery — це інструмент, який призначений для автоматичного пошуку мікросервісів в мережах. У мікросервісній архітектурі нам потрібно ім’я хоста або IP-адресу, за домогою яких, ми будемо мати змогу знаходження мікросервісів. Дана концепція існує з початку розроблення розподілених систем та відома як «виявлення служб». Він може бути налаштованим різними способом, наприклад, як файл властивостей зі всіма адресами усіх віддалених сервісів. Налаштування даного інструменту відіграє дуже важливе значення для мікросервісів та хмарних обчислень, взагалом з двох ключових причин:

1) горизонтальне масштабування, щоб мати змогу розділення системи на малі структурні компоненти та рознесення їх на окремі фізичні машини;

2) відмовостійкість — шаблон, який призначений для вирішення виниклих проблем та запобіганню каскадному розповсюдженню їх в мікросервісній архітектурі.

Існує два типи виявлення сервісів — на стороні сервера та клієнта. Виявлення сервісів на стороні сервера дозволяє клієнтським програмам знаходити служби через маршрутизатор або балансувальник навантаження. Виявлення служб на стороні клієнта дозволяє клієнтським програмам знаходити служби, переглядаючи або запитуючи реєстр послуг, у якому екземпляри та кінцеві точки служб містяться в реєстрі послуг [16]. Існують декілька компонентів Service Discovery:

1) постачальник реєструється в реєстрі послуг під час входу та скасовує реєстрацію, коли відбувається вихід з системи;

2) споживач — отримує поточне знаходження постачальника в реєстрі послуг, а згодом підключається до нього;

3) реєстр сервісів — це деяка база даних, яка зберігає в собі розташування екземплярів сервісів. Реєстр послуг повинен бути кожен раз доступним, щоб була можливість переглядати розташування в мережі мікросервісів.

В ході виконання дипломної роботи використовувалася бібліотека Spring Cloud Netflix: Eureka.

Eureka Server — це бібліотека, яка містить в собі інформацію про сервісні застосунки. Кожен з них реєструється в сервері зі своїм портом та IP- адресою, і за допомогою exo-запитів повідомляє про свою активність. Існують такі аналоги цієї бібліотеки, як Zookeeper, Cloud Foundry, Consul. Щоб налаштувати сервер потрібно використовувати анотацію «@EnableEurekaServer» та деякі додаткові налаштування. Щоб налаштувати інші мікросервісні застосунки і сервер міг їх знаходити, вони повинні мати анотацію «@EnableEurekaClient» (вказує про те, що сервіс є деяким екземпляром якогось мікросервісу і подає запит на раєстрацію в сервері) та налаштування, в яких задається адреса сервера і деякі додаткові параметри,

щоб вони були помітні для нього. Наприклад, можна задати, такі налаштування:

1) spring.application.name — унікальне ім'я програми;
2) server.port — порт, на якому буде запущено програму, за замовчуванням використовується порт 8761;
3) eureka.client.register-with-eureka — визначає, чи реєструється сервіс, як клієнт на Eureka Server;
4) eureka.client.fetch-registry — визначає те, чи потрібно отримувати інформацію про зареєстрованих клієнтів.

Eureka-client, за замовчуванням, запускається в стані «STARTING». Це дає екземпляру можливість виконати ініціалізацію для конкретного застосунку, перш ніж буде відбуватися обслуговування його трафіку. Eureka- client спочатку намагається зв'язатися з сервером у спільній зоні. Але якщо пошук не успішний, то він перемикається на інші зони. Якщо не відбувалося ні якого з’єднання в період 30 секунд, то він їх скидає.

Клієнт взаємодіє з сервером, таким чином:

1) eureka client реєструє інформацію про запущений екземпляр на сервері;

2) кожні 30 секунд клієнт надсилає запит на сервер та інформує про те, що екземпляр «живий». Якщо сервер не помічає оновлення протягом 90 секунд, він видаляє екземпляр зі свого реєстру;

3) eureka client отримує інформацію про реєстр від сервера і кешує її. Дана інформація періодично оновлюється. Він перевіряє час між останнім оновленням інформації та поточним, де згодом клієнт автоматично обробляє її;

4) коли клієнт отримав оновлення, він перевіряє надану інформацію з сервером, порівнюючи при цьому, кількість екземплярів, що повертаються. Якщо інформація не збігається, то вона знову отримується з реєстру. Клієнт отримує інформацію у форматі JSON;

5) після завершення роботи клієнт надсилає запит на скасування серверу. Таким чином, екземпляр видаляється з реєстру екземплярів сервера.

На рисунку 6 зображено приклад, як створені мікросервіси (Instance A, B, C), направлені до мікросервісу Service Registry. Коли клієнт реєструється в ньому, він надає метадані про себе, такі як хост, порт, URL-адресу, індикатор працездатності та інші відомості. Це потрібно для того, щоб зареєструвати сервіси, які знаходяться в певній мережі. Налаштування відбувається так, що кожен мікросервіс налаштовується в певній зоні, яка в свою чергу має адресу сервера.

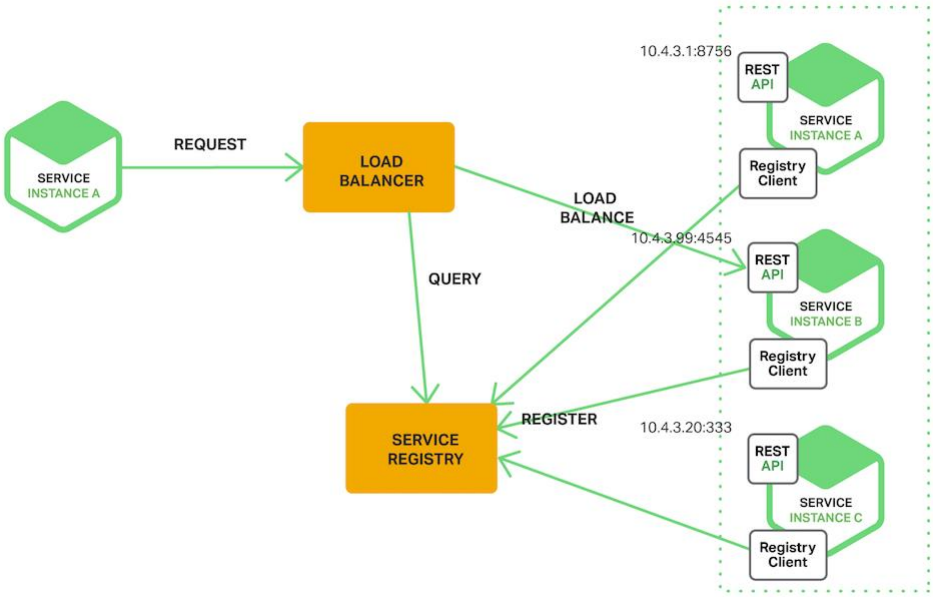


Рис.*6 Service Discovery — інструмент пошуку мікросервісів в мережі*

3. Spring Cloud Config — це інструмент, який надає клієнтську та серверну підтримку зовнішньої конфігурації. За допомогою Config Server ми можемо оперувати центральним місцем для керування зовнішніми властивостями програм у всіх середовищах. Налаштування відбувається таким чином, що створюється мікросервіс, який звертається до файлу конфігурацій, який в свою чергу знаходиться у віддаленому сховищі (Рис. 7).

Spring Cloud Config має власне сховище для керування властивостями, але також, він може взаємодіяти з такими проєктами:

1) consul — сервіс виявлення, який дозволяє екземплярам, зареєструватися в ньому. Налаштовані сервіси, зможуть подати запит до Consul, щоб мати змогу пошуку розташування екземплярів сервіса. Він містить базу даних сховища — пару «ключ-значення», яку Spring Cloud Config використовує для зберігання даних конфігурації програми;

2) eureka — проект Netflix, який, як і Consul, пропонує подібні можливості пошуку служб. Також має свою базу даних — пару «ключ-значення», яку можна використовувати з Spring Cloud Config;

3) git (https://git-scm.com/) — система керування версіями, яка дозволяє проводити управління та відстеження зміни в будь-якому текстовому файлі. Spring Cloud Config інтегрується з внутрішнім репозиторієм Git і зчитує конфігураційні дані.

За замовчуванням реалізація серверної системи зберігання даних використовує GIT, бо він легко підтримує помічені версії середовищ конфігурації, а також має доступ до великої кількості інструментів для керування вмістом. До нього можна додавати нові реалізації та без проблем підключати його до конфігурації Spring. Даний файл зберігає в собі ті налаштування, які можуть знадобитися іншим мікросервісам. Тому, щоб не повторювати одні й ті ж конфігурації для всіх сервісів, достатньо налаштувати мікросервіс, який відповідає за віддалене центральне сховище, і додати кілька конфігурацій для того, щоб була змога звертатися до цього сховища.

Функції Spring Cloud Config Server:

1. API на основі ресурсів для зовнішньої конфігурації (пари «ім’я- значення»).

2. Шифрування та дешифрування значень властивостей (симетричних або асиметричних).

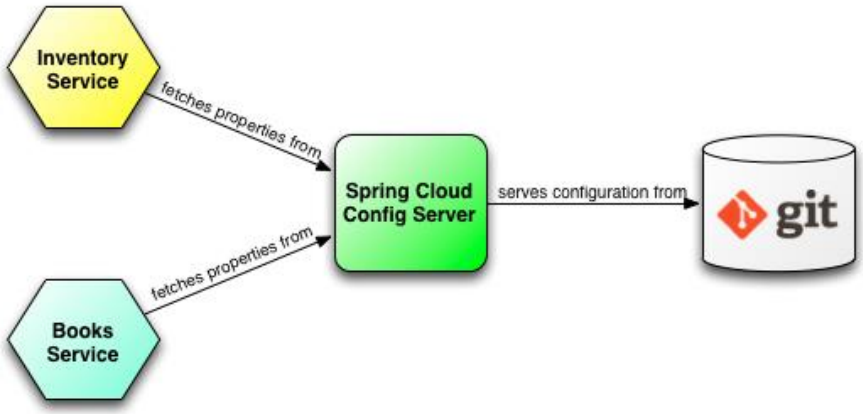


Рис.*7 Spring Cloud Config Server — централізоване сховище конфігурацій*

4. Load balancing (балансувальник навантаження) — це інструмент, який дозволяє розподілити вхідний трафік порівну між серверами, з метою підвищення ефективності та обробки. Також він покращує доступ до мікросервісів, швидко реагує та запобігає перевантаженню.

Балансувальник навантаження має такі характеристики:

1) він може бути в ролі фізичного пристрою, де знаходиться певна кількість віртуальних екземплярів сервісів, що працюють на деякому спеціально налаштованому обладнанні;

2) розроблений для високого підвищення продуктивності та безпеки веб-застосунків та мікросервісів, незалежно від їхнього розташування;

3) має здатність використовувати всі можливі алгоритми балансування навантаження та метод найменшого з’єднання для розподілу трафіку.

Балансувальник навантаження має можливість виявлення працездатних ресурсів, і, якщо в ході роботи виникає проблема з сервером, який не може виконати запит по тій чи іншій причині, він відміняє та не надсилає трафік. Незалежно від того, чи це програмне забезпечення, апаратне забезпечення, або якийсь алгоритм, який він використовує, балансувальник буде розподіляти трафік на різні сервери в своєму пулі ресурсів. Його призначення полягає в тому, щоб гарантувати, що сервер не буде перевантажений та ненадійний. В свою чергу, даний аспект надає ефективно мінімізувати час відповіді та максимізувати пропускну здатність. Налаштовані балансувальники

навантаження повинні забезпечувати продуктивність і безпеку, необхідні для підтримки застосунків, а також складних робочих процесів, що в них відбуваються.

Існують два види балансування навантаження: балансування на стороні сервера та клієнта.

Балансування на стороні серверу. Наприклад, клієнт подає запити на отримання даних. Балансувальник навантаження приймає їх та пересилає до якогось визначеного одного серверу (наприклад, за допомогою алгоритму випадкового вибору). За рахунок цього, він запобігає прямому зв’язку клієнта з внутрішніми налаштованими серверами, надає безпечну взаємодію в мережі та запобігає атакам на мікросервіси. Якщо виникла проблема і сервери не доступні для взаємодії, то він може пересилати запити на резервні налаштовані балансувальники, або відображати повідомлення про збій.

Балансування на стороні клієнта відбувається по іншому, система сама вирішує до якого серверу надсилати запит (Рис. 8). Балансувальник надає клієнту налаштований список IP-адрес серверів, а потім випадково вибирає з списку для кожного запрошуваного з’єднання.

Існують деякі популярні алгоритми балансування, наприклад випадковий вибір, вибір екземплярів в тому самому порядку, найменша кількість підключень, зважена метрика, хеш IP-адреси [17]. Кожен з цих видів має свої переваги та недоліки, використання кожного з них залежить від поставленої бізнес задачі.

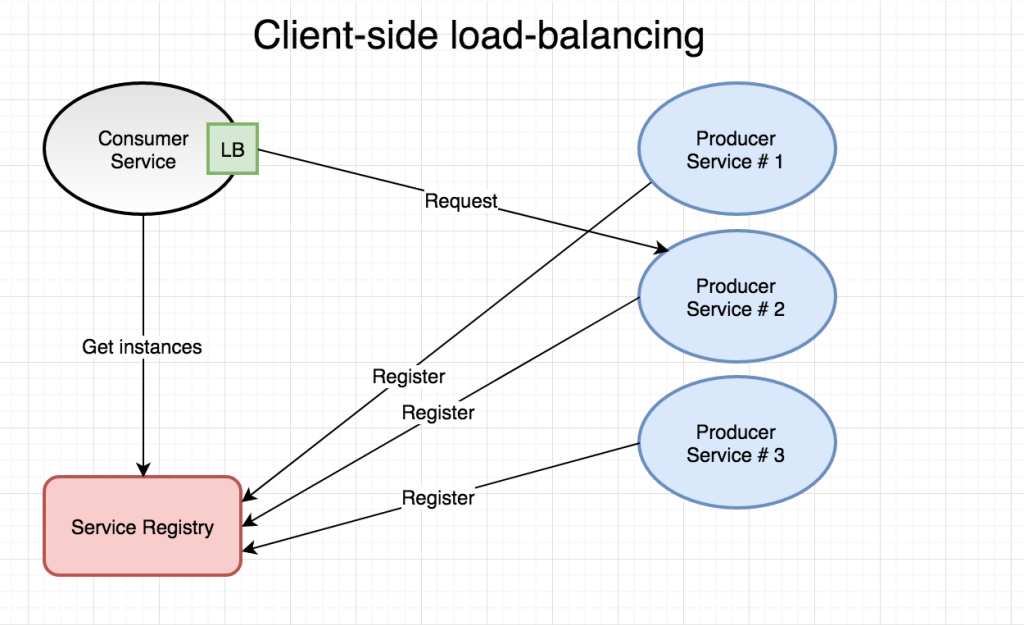


Рис.*8 Client-side load balancing — розподіляє вхідний трафік порівну між сервісами*

В ході реалізації програмного забезпечення, використовувалася бібліотека Netflix Ribbon. Вона основується на балансуванні навантаження, на стороні клієнта. Даний інструмент надає, такі функції, як:

1) виявлення сервісів;
2) конфігуровані правила балансування навантаження;
3) шляхом постійної перевірки визначає доступність серверів;
4) фільтрація серверів, за допомогою заданих критеріїв перевірки;
5) підтримка декількох протоколів, наприклад HTTP;
6) кешування.

5. Circuit breaker — це бібліотека, яка допомагає контролювати взаємодію між службами та надає такі функції як відмовостійкість та стійкість до затримок, що в свою чергу допомагає всій системі швидко оброблювати запити користувача. Модель автоматичного вимикача реалізована за прикладом роботи електричного вимикача. В електричній системі автоматичний вимикач виявляє, якщо через нього протікає занадто великий струм, то він розриває зв'язок та утримує систему від пошкодження наступних компонентів. В програмному автоматичному вимикачі, коли викликається

віддалений сервіс, він слідкує за цим запитом. Якщо вони тривають надто довго, автоматичний вимикач втручається та відкликає їх. Патерн автоматичного вимикача відстежує всі виклики до віддаленого ресурсу, і якщо достатня кількість викликів не вдається, то він швидко зробить запобігання майбутніх викликів до ресурсу, що вийшов з ладу.

При великій кількості запитів, навіть найнадійніші сервіси можуть привести до збоїв. В мікросервісній архітектурі нажаль це неминуче і, тому потрібно прийняти запобіжний захід. Виникає проблема, наприклад, клієнт подає запит на отримання своїх попередніх замовлень, цей запит направляється до одного мікросервісу, а той в свою чергу залежить від інших мікросервісів. На шляху отримання замовлень, один мікросервіс вибуває з ладу, і не зможе повернути результат клієнту. Тому на допомогу нам приходить автоматичний вимикач, який в свою чергу спрямований на запобігання виходу з ладу окремого компонента за його межі, і тим самим не дає вивести з ладу всю систему [18]. Ідея полягає в тому, що, коли робота мікросервісу стає надто повільною або часто надає відповідь з помилками, автоматичний перемикач спрацює і майбутні запити негайно повернуть повідомлення про помилку.

На рисунку 9 зображено як працює автоматичний вимикач. Він має три стани. Стан «Замкнутий» — стан говорить про те, що все в нормі, і всі запити можуть отримати доступ до мікросервісів. Але, якщо через велику кількість запитів спричинено якісь несправності (наприклад, винятки) і вони перевищують заданий лічильник відмов і тайм-аутів, вимикач спрацьовує і переходить в відкритий стан. Стан «Відкритий» — запити не передаються до мікросервісу, але у відповідь надсилається клієнту повідомлення про помилку. Також, автоматичний вимикач, може зробити перехід від відкритого стану до напіввідкритого стану, щоб мати можливість повторної перевірки несправностей в сервісах. Стан «Напіввідкритий» — у цьому стані до служби дозволено надсилати обмежену кількість запитів. За умови, якщо сервіс

надсилає успішні відповіді, автоматичний вимикач повертається в замкнутий стан, і його лічильники несправностей і часу очікування скидаються.

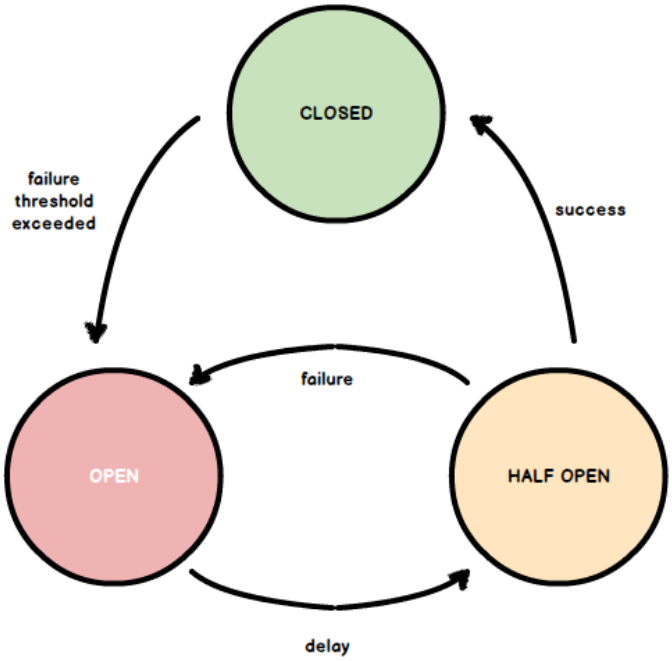


Рис.9 *Circuit breaker — бібліотека відмовостійкості*

Шаблон автоматичного вимкнення забезпечує стабільність, доки система відновлюється після збою та знижує вплив на продуктивність. Завдяки цьому можна підтримувати певний показник часу відгуку системи, швидко відхиляючи запит на операцію, яка, швидше за все, завершиться зі збоєм, замість того, щоб чекати, поки не закінчиться час очікування операції, або чекати протягом невизначеного часу. Якщо автоматичне вимкнення породжує подію щоразу, коли вона змінює стан, ця інформація може використовуватися для моніторингу працездатності.

6. Distributed tracing (Трасування запитів) (рис.10) — розподілене трасування або трасування розподіленого запиту є корисним інструментом для команд IT та DevOps, він призначений для того, щоб допомогти більш ефективно відстежувати та усувати збої, які виникають в ході спілкування між мікросервісами [19]. Наприклад, ми хочемо отримати послідовність інформації, коли подається запит між декількома мікросервісами та базою

даних. Даний інструмент надасть нам інформацію, про те, який запит був відправлений, через які мікросервіси він проходить свій шлях, скільки часу обробляється запити в кожному мікросервісі, скільки наступному сервісу потрібно було почекати, щоб попередній виконав свою задачу, якщо виникли якісь проблеми, то інструмент надасть інформацію про те, де саме вона виникла і яким чином. Також він може надати нам інформацію про продуктивність, а саме яка кількість запитів завершилась помилкою, або скільки запитів за секунду може обробити будь-який сервіс.

В інструменті трасуванні запитів є деяка термінологія, яка допомагає розуміти, що означає кожен компонент, і для чого він призначений:

1) trace id — являє собою, який унікальний ідентифікатор кожного запиту;

2) span — кожен запит, який буде надходити від клієнта, може проходити через велику кількість мікросервісів і тому, даний компонент призначений для того, щоб показати шлях, тобто через, яку кількість сервісів він проходить та, яку кількість відповідей він повернув;

3) журнал, показує детальну інформацію про шлях запитів між мікросервісами, а також показує додаткові події;

4) тег, призначений для того, щоб робити анатоції span, щоб з легкістю можна було фільтрувати та знаходити необхідну інформацію.

В ході виконання дипломної роботи використовувалася бібліотека Zipkin. Вона є однією із розподілених інструментів трасування. Бібліотека допомагає збирати метадані, необхідні для відстеження та усунення проблем затримки в мікросервісній архітектурі. Інтерфейс користувача Zipkin також представляє діаграму залежностей, яка показує, скільки відстежених запитів пройшло через кожну програму. Це може бути корисним для визначення сукупної поведінки, включаючи шляхи помилок або виклики застарілих служб [20].

Додатково з бібліотекою Zipkin, в розподіленому трасуванні використовувалася бібліотека Spring Cloud Sleuth. Вона призначена для

трасування логів в розроблюваному застосунку. Робота її заключається в тому, що бібліотека добавляє унікальний id, за допомогою якого, можна легко зробити діагностику та знайти інформацію (логи), які відносяться до того чи іншого запиту, який було проведено на мікросервіс.

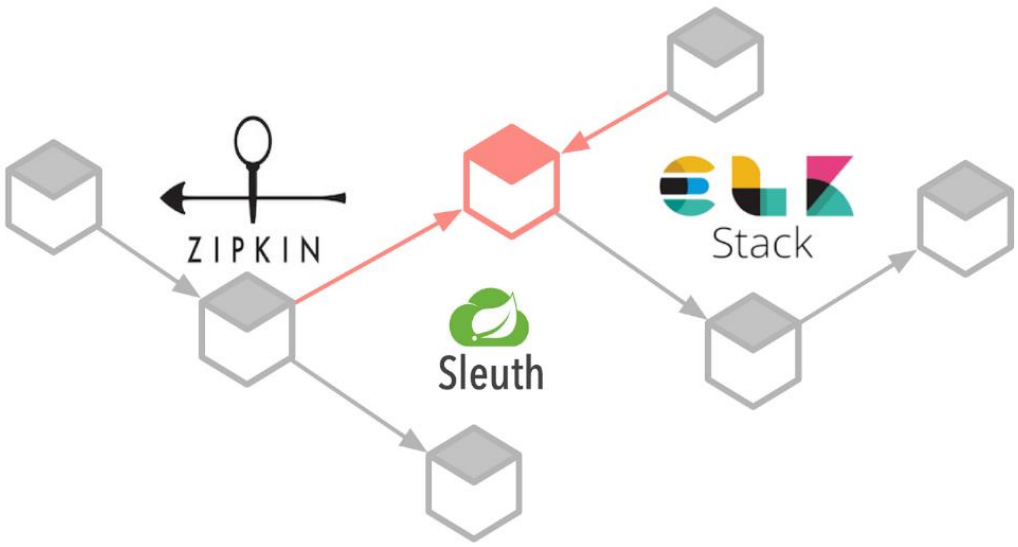


Рис.*10 Distributed tracing — бібліотека розподіленого трасування*

7. Rabbit MQ — брокер повідомлень, який приймає, зберігає та пересилає двійкові блоки даних — повідомлення [21]. Вони можуть містити, що завгодно, наприклад інформацію про процес або завдання, яке має бути виконано в іншій програмі, або це може бути, будь-яке інше текстове повідомлення. Інструмент для керування чергою зберігає в собі повідомлення, до того часу, поки одержувач не підключиться та не вилучить дане повідомлення з черги. Згодом вона обробляє повідомлення.

Брокер повідомлень діє як посередник для різних сервісів, який використовується для того, щоб можна було зменшити навантаження та час доставки повідомлень до інших серверів, які зазвичай займають багато ресурсів. Коли відбувається додавання в чергу повідомлень, ми робимо більш пришвидшене виконання запитів. За рахунок цього, ми позбавляємось від затримки часу відповіді на запит та не проводимо ресурсомісткі операції.

Постановка в чергу повідомлень також корисна, коли потрібно розповсюдити повідомлення багатьом споживачам або збалансувати навантаження.

Даний інструмент відповідає всім принципам протоколу AMQP (Advanced Message Queuing Protocol) — це відкритий стандарт корпоративного обміну повідомленнями [22]. Даний протокол надає такі концепції та поняття, які використовується в налаштуваннях:

1) виробник, його призначення полягає в тому, щоб відправляти потрібні повідомлення;
2) споживач — отримує повідомлення;
3) повідомлення, деяка інформація (наприклад, в json-форматі), яка надсилається від виробника до споживача;
4) з’єднання. TCP—з’єднання між застосунками та брокером RabbitMQ;
5) канал — віртуальне з'єднання. Його призначення полягає в тому, щоб під час публікації або споживання повідомлень із черги, проводити всі дії через канал;
6) точка обміну, куди відправляються повідомлення. Він розподіляє повідомлення в одну або декілька черг;
7) черга — вона зберігає посилання на повідомлення та віддає копії споживачам;
8) прив’язка — правило, яке повідомляє точці обміну в яку з створених черг, повідомлення повинні відправлятися.

Брокер повідомлень працює таким чином, що видавець надсилає повідомлення в якусь налаштовану точку обміну. Потім він маршрутизує повідомлення в певну чергу, відповідно до правил прив’язки між ними. Черга зберігає посилання цього повідомлення. Як тільки споживач готовий отримати повідомлення з черги, сервер створює копію повідомлення за посиланням та відправляє його. Споживач отримує повідомлення та надсилає брокеру підтвердження. Він може знаходитися на зовсім іншому сервері, або може бути розташований на одному сервері разом з видавцями. Запит може бути створений однією мовою програмування та оброблений іншою. Програми

будуть спілкуватися через повідомлення, які надсилають один одному, що означає, що відправник і одержувач мають низький зв’язок між собою. Брокер, отримавши підтвердження, видаляє копію повідомлення з черги. Потім видаляє з оперативної пам'яті та з диска.

2.3 Бібліотеки та інструменти використані в мікросервісному застосунку

Для реалізації мікросервісного застосунку, використовувалися багато бібліотек та інструментів, які можуть покращити та полегшити розробку програмного застосунку:

1. Spring Security — платформа, яка призначена в забезпеченні застосунків аутентифікацією, авторизацією та іншими функціями безпеки. Наприклад, функція безпеки запитів, яка будується на основі URL-адрес веб- запитів або регулярних виразів. Це потрібно для того, щоб користувач або зловмисник не мали доступу до захищених ресурсів, якщо вони не аутентифіковані та не мали змоги дістатися до корпоративних конфіденційних даних.

Окрім автентифікації, за допомогою даного інструменту можна шифрувати дані користувача та безпечно зберігати паролі. Ми можемо використовувати різні реалізації алгоритмів, наприклад алгоритм BCrypt.

Spring Security надає такі переваги, при його використанні: захист від різних можливих зловмисних атак, фіксація та обробка сесії, легке інтегрування із застосунками; конфігурація; портативність, тощо. Також він має такі функції, як авторизація; принцип єдиного входу; локалізація програмного забезпечення; cookie — дані; принцип запам’ятовування входу користувача в систему; надає легкий протокол автентифікації облікових записів, як окремих користувачів, так і цілої організації; HTTP-авторизація; автентифікація WEB-форм; базова автентифікація доступу; стандарт перевірки прав користувача за допомогою сервісу авторизації OAuth 2.0, тощо.

Даний інструмент є дуже популярним та вживаним серед програмістів. Він потребує ретельного підходу та налаштування. Адже погано налаштована безпека застосунку, може дуже сильно вплинути на бізнес, зловмисник без проблем, може витягнути дані користувача, сфальсифікувати та нашкодити.

2. HikariCP — найшвидший пул з'єднань, який призначений для того, щоб проводити великовагові операції з базою даних, з великою кількістю запитів. Даний інструмент надає, такі переваги як:

1) оптимізація байт-коду: оптимізує код доти, доки скомпільований байт-код не стане найменшим, щоб кеш центрального процесору міг завантажувати більше програмних кодів;

2) оптимізація проксі та перехоплювача: скорочення коду, наприклад, проксі-сервер HikariCP Statement має лише 100 рядків коду;

3) настроюваний тип колекції, що надає підвищену ефективність одночасного читання та запису;

4) пул посилань на базу даних встановлюється заздалегідь;

5) виявляє аномальні посилання та звільняє ті, що не використовуються.

Усі пули посилань на бази даних відповідають основним правилам проектування та реалізують інтерфейс javax.sql.DataSource. Найбільш важливим методом є getConnection(), який використовується для отримання з'єднання (це посилання на базу даних).

3. Бібліотека spring-boot-starter-validation, яка призначена для перевірки даних. Використовуються, як анатоції над полями класу, даний підхід надає нашому коду не бути «брудним». При передачі розміченого таким чином об'єкта класу у «валідатор» відбувається перевірка на обмеження, наприклад тіло запиту, змінні шляхи (наприклад, id в /food/{id}), параметри запиту, тощо.

4. Бібліотека spring-boot-starter-web — призначена для того, щоб майбутній застосунок був web-застосунком.

5. Бібліотека java-jwt. JWT — один із найпоширеніших методів, автентифікації запитів. Цей вид захисту та автентифікації має ряд переваг:

1) зручність (не потрібно при кожному запиті передавати логін та пароль);

2) менше запитів до бази даних (токен може містити в собі базову інформацію про користувача);

3) простота реалізації (досить використовувати готову бібліотеку для генерації та розшифрування токена).

Токен — це просто рядок, який генерується на запит користувача. Він надає додаткові функції, які дозволяють викликати захищені ресурси. Користувач реєструється в системі, потім робить запит на генерацію токена. Далі він може робити авторизовані запити на сервер, використовуючи токен. Як правило, токен поміщають в header запиту для зручності передачі та зчитування.

При генерації токена на сервері, в нього поміщають, при необхідності, дані про сесію користувача або іншу необхідну інформацію, шифрують за допомогою алгоритмів шифрування та секретного ключа. Секретний ключ має бути добре захищений. Якщо токен є ще дійсним, і зловмисник не зміг його підробити, користувач може вільно виконувати запити. В іншому випадку, якщо токен не є дійсним, і не підробленим, то користувачу йде заміна токена, а тобто гененерація нових access та refresh токенів. В гіршому випадку, сесія користувача закривається, а токени видаляються, і користувачу потрібно знову буде заходити під своїми даними.

6. Log4j — надійна, швидка та гнучка бібліотека, яка призначена для ведення журналу даних. Її можна налаштувати для введення логування даних в методах застосунків. Вона може надавати зберігання інформації в файлах, з заданою обмеженою кількістю мегабайтів для самих файлів. Ведення журналу відігрує велику роль та є важливою складовою циклу розробки в багатопоточних програмах та розподілені системах.

7. Бібліотека Spring Data JPA. Її призначення полягає в тому, що вона дозволяє легко впроваджувати репозиторії на основі JPA (Java Persistence API), також надає розширену підтримку рівнів доступу до даних та полегшує

створення додатків, які використовують технології доступу до даних. Вона має такі особливості: динамічне виконання запитів; створення запитів за допомогою анотації «@Query()»; конфігурація; безпека запитів, тощо.

8. Бібліотека flyway-core, яка використовується для міграції даних до бази даних. В разі її зміни, в програмі з легкістю можна змінити структуру, додати до файлу міграції нові зміни, і працювати з відновленою базою.

2.4 Висновки з розділу 2

В даному розділі було проведено аналіз інструментів, технологій, бібліотек, які використовуються для побудови міксросервісної архітектури. На основі цього можна сформулювати такі висновки:

1. Spring Cloud — це потужна платформа, яка надає безліч готових реалізацій, які допомагають, без труднощів налаштувати мікросервісну архітектуру.

2. Для побудови застосунків доречно використовувати Java Spring Framework, так як він надає необхідні засоби, залежності та реалізації, за допомогою яких можна побудувати монолітну та мікросервісну архітектури.

3. Щоб полегшити розробку програмного забезпечення, краще використовувати готові біблотеки та інструменти.

4. Ознайомлено та вивчено основні інструменти, які надає Java Spring Cloud, для побудови базової мікросервіної архітектури.

РОЗДІЛ 3 РОЗРОБКА ТЕСТОВИХ ЗАСТОСУНКІВ З МОНОЛІТНОЮ І МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1 Призначення розробки

Побудовані корпоративні застосунки на стороні сервера призначені для ознайомлення схожості та відмінностей між монолітною та мікросервісною архітектурами. Визначити їх переваги та недоліки, та зрозуміти, яку з них вигідніше використовувати для поставлених майбутніх бізнес-задач. Виконати тестування для кожного застосунку та порівняти їх за отриманими показниками.

Корпоративні застосунки, які побудовані на стороні клієнту, призначені для того, щоб доставляти багато різних продуктів від партнерів, шкидко і безпечно. Зацікавати нових партнерів для подальшого розширення, просування та співпраці з ними. Зацікавати нових співробітників та кур’єрів на роботу. Збільшувати кількість клієнтів та зарахунок цього, збільшувати обсяги продажів.

3.2 Функціональні та нефункціональні вимоги

3.2.1 Функціональні вимоги

Сайт, який був розроблений в ході виконання переддипломної роботи повинен забезпечувати наступні функціональні вимоги:

1. Клієнт повинен мати змогу замовляти, оплачувати продукти.
2. Клієнт повинен мати особистий кабінет, в якому він може редагувати свої дані та створювати способи оплати.
3. Клієнт повинен мати змогу переглядати статус доставки та інформацію про кур’єра.
4. Клієнт повинен мати змогу переглядати історію замовлень.

5. Партнер повинен мати свій особистий кабінет, переглядати замовлення, які відносяться до його ресторану, та редагувати дані продуктів.
6. Кур’єр повинен мати особистий кабінет, для того, щоб переглядати замовлення, приймати чи відмовлятися від них, і в разі кінцевої доставки, замовлення видаляти з його персонального кабінету.
7. На сторінках сайту повинні бути блоки призначені для реєстрації.
8. В разі реєстрації клієнта, кур’єра, співробітника або партнера, повинно бути підтвердження на пошті.
9. Клієнт повинен мати змогу перегляду інформації перед оплатою, вибирати спосіб оплати, залишати деякий коментар для кур’єра та певні чайові.

3.2.2 Діаграми використання

Дійові особи:

1. Клієнт — людина, яка замовляє продукти в партнера, проводить оплату та має можливість перегляду статусу доставки.

Для клієнта виділяються наступні варіанти використання (рис.11):

- Клієнт завітав на сайт.
- Перегляд інформації на сайті.
- Перегляд партнерів.
- Реєстрація та вхід до особистого кабінету.
- Редагування персональних даних в кабінеті.
- Створення та видалення способів оплати.
- Перегляд історії замовлень.
- Перегляд додаткової інформації щодо доставки.
- Перегляд категорій та продуктів, які відносяться до вибраного партнера.
- Можливість додавання певної кількості продуктів.
- Можливість оплати, заповнення даних адреси доставки.

- Можливість вибору кількості чайових та залишати повідомлення призначених для кур’єра.

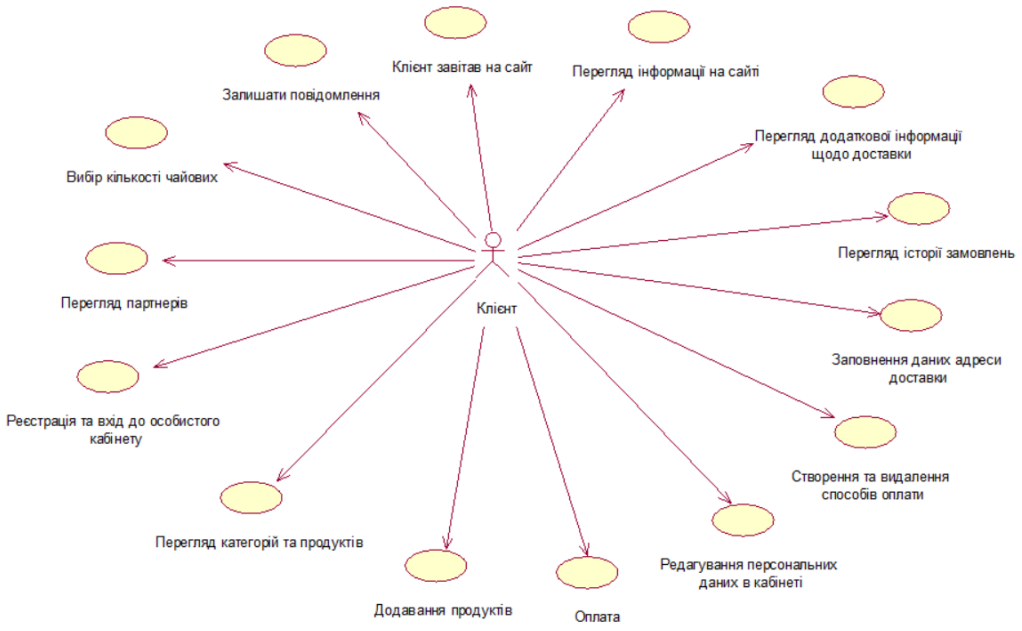


Рис.*11 Варіанти використання для клієнта*

2. Кур’єр — людина, яка займається швидкою доставкою продуктів для клієнтів. Для нього виділяються наступні варіанти використання (рис.12):

- Кур’єр завітав на сайт.
- Перегляд інформації на сайті.
- Реєстрація та вхід в особистий кабінет.
- Приймати або відмовитись від замовлення.
- Перегляд замовлення.
- Перегляд детальної інформації про замовлення.

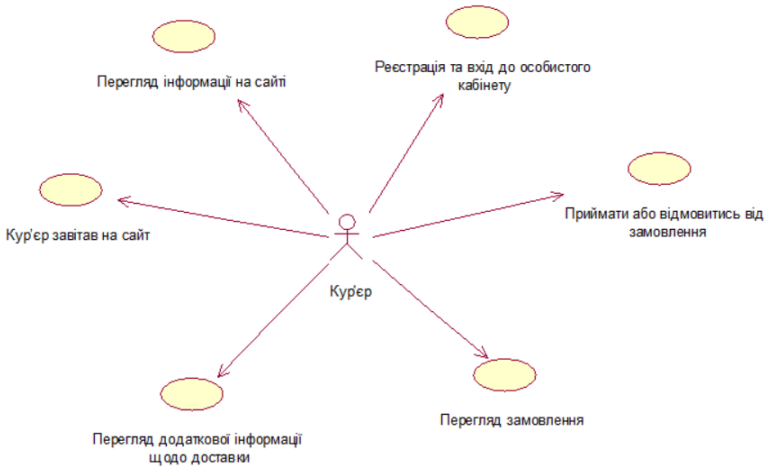


Рис.*12 Варіанти використання для кур'єра*

3. Партнер — людина, яка представляє свої продукти для продажу. Для нього виділяють наступні варіанти використання (рис.13):

- Партнер завітав на сайт.
- Перегляд інформації на сайті.
- Реєстрація та вхід в особистий кабінет.
- Перегляд замовлень.
- Створення та редагування продуктів.

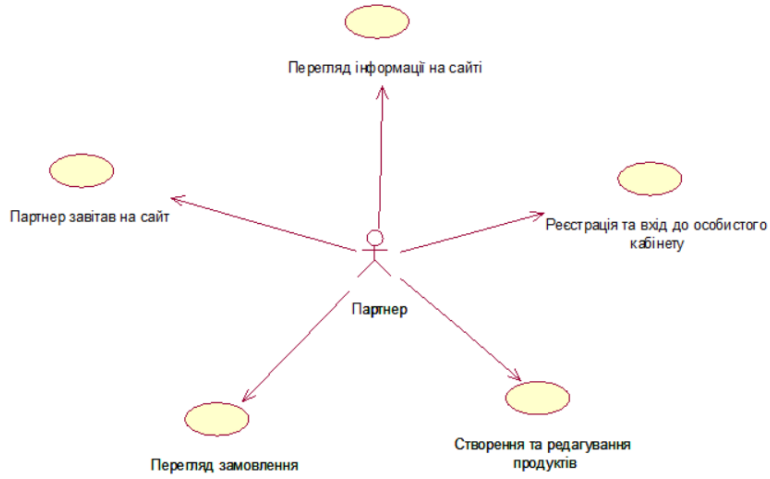


Рис.*13 Варіанти використання для партнера*

4. Адміністратор (співробітник) — людина, яка займається повним адмініструванням даних. Для нього виділяються наступні варіанти використання (рис.14):

- Адміністратор завітав на сайт.
- Перегляд інформації на сайті.
- Реєстрація та вхід в особистий кабінет.
- Перегляд інформації щодо доставки, клієнтів, співробітників, тощо.
- Створення та редагування даних.

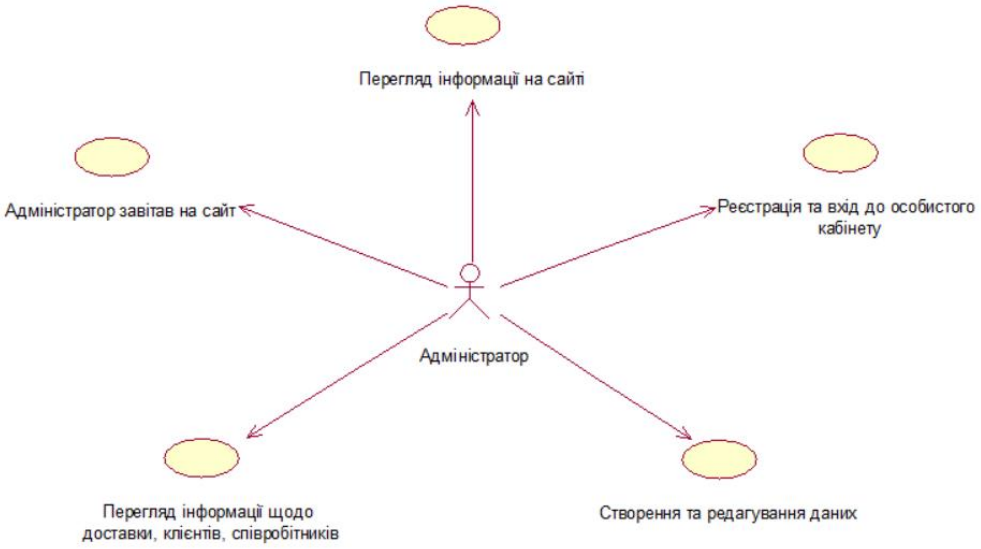


Рис.*14 Варіанти використання для адміністратора*

3.2.3 Нефункціональні вимоги

1. Зручний та стильний дизайн сайту.
2. Простота в експлуатації.
3. Усі взаємодії здійснюються через «браузер» користувача.
4. Користувачі, кур’єри та партнери не повинні мати доступ до даних співробітників доставки.
5. Можливість відновлення після певних збоїв серверної частини.
6. Швидка відповідь на запити.

7. Забезпечити безпеку входу до особистого кабінету, використовуючимаркери доступу (jwt-токени).

3.3 Вимоги до інтерфейсу

На основі аналізу загальних вимог до проектування інтерфейсної частини програми, було виділено ряд рекомендацій для створення стильного для користувача інтерфейсу. Можна виділити наступні з них:

1) головна сторінка повинна мати партнерів, які співпрацюють з доставкою, мати певну інформацію, яка призначена для реєстрації нових кур’єрів та партнерів;
2) сторінка реєстрації кур’єрів повинна мати швидку та легку форму заповнення даних, деяку додаткову інформацію, для того, щоб зацікавити все більше нових кур’єрів;
3) сторінка реєстрації партнерів повинна мати більше полів заповнення даних форми реєстрації, для того, щоб можна було дізнатися інформацію про нового партнера. Також додати деяку додаткову інформацію, щоб партнер розумів, чому нам важлива співпраця з ним;
4) сторінка менеджменту для партнерів, яка призначена для того, щоб він міг переглядати замовлення, які стосуються його підприємства та мати можливість створювати та редагувати дані, пов’язані з продуктами;
5) сторінка менеджменту для кур’єрів повинна бути проста у використанні, та мати можливість перегляду поточних замовлень. А також, мати вибір прийняття та відмови від замовлення;
6) сторінка менеджменту для адміністраторів, повинна мати всю інформацію про клієнтів, партнерів, співробітників і, мати можливість створювати та редагувати дані;

7) сторінка контенту партнера, яка має різні категорії, які в свою чергу мають продукти, що відносяться до цих категорій. На ній повинна бути корзина, яка показує, які продукти і, яка їх кількість на даний момент;
8) сторінка оплати, на якій повинна бути можливість перегляду замовлених продуктів, вибору оплати, заповнення даних адреси доставки, вибору чайових для кур'єра та, мати можливість залишати додаткове повідомлення для нього. Також повинен бути остаточний чек пов'язаний з оплатою;
9) вікно перегляду детальної інформації про поточне замовлення;
10) вікно перегляду історії замовлень;
11) вікно редагування особистих даних клієнта;
12) вікно створення та видалення кредитних карток.

3.4 Діаграми послідовності

Діаграма послідовності для варіанта використання: «Клієнт реєструється в додатку»

Клієнт натискає на кнопку «Розпочати», перед ним з'являється вікно з реєстрацією, він вводить та заповнює дані в полях, та отримує повідомлення про те, щоб підтвердити свою особу. Йому на електронну пошту надсилається посилання. Користувач переходить за ним і цим самим підтверджує реєстрацію та повертається на головну сторінку (рис.15).

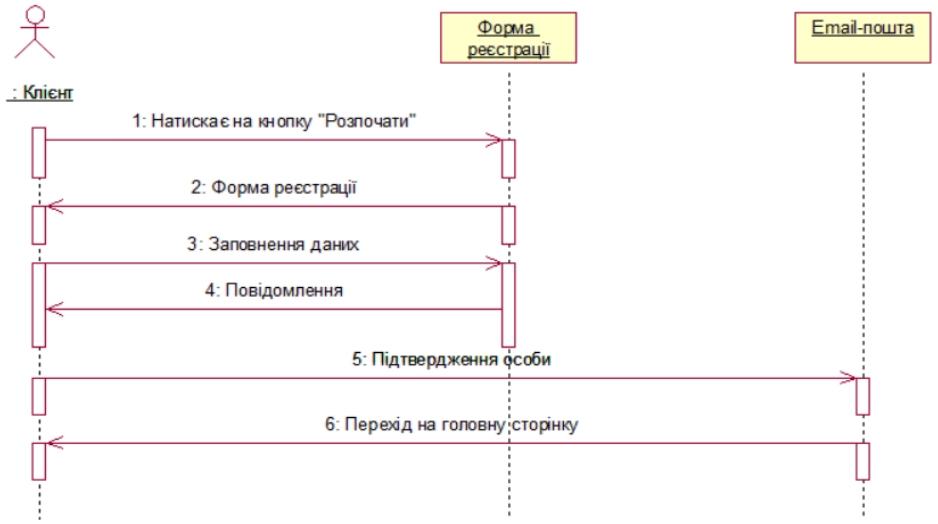


Рис.*15 Клієнт реєструється в додатку «Vityunchik-Delivery»*

Діаграма послідовності для варіанта використання: «Оформлення замовлення»

Клієнт вибирає ресторан, в якому хоче замовити продукти. Після чого, переглядає та вибирає категорії, які містять в собі різні продукти. Далі відбувається перегляд вибраних продуктів та їх загальна кількість і ціна. Після, відбувається перехід на сторінку оплати, заповнення даних, вибір додаткових побажань для кур'єра. Після оплати, клієнту відкривається вікно детальної інформації, яка стосується поточного замовлення (Рис.16).

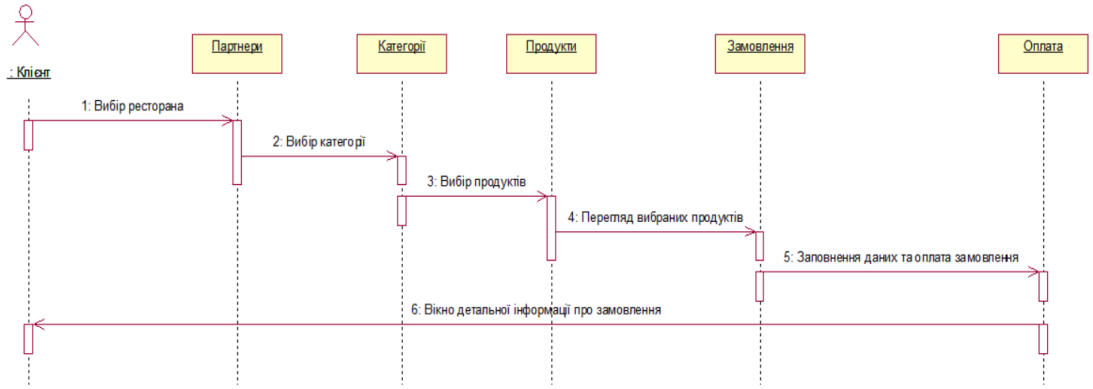


Рис.*16 Оформлення замовлення*

Діаграма послідовності для варіанта використання: «Створення та видалення кредитної карти»

Клієнт в своєму особистому кабінеті натискає на кнопку створення кредитної карти, заповнює дані та отримує успішне або неуспішне повідомлення про створення. Якщо клієнт хоче видалити кредитну карту, він натискає на кнопку видалення над потрібною кредитною картою, і отримує повідомлення про успішне або неуспішне видалення (Рис.17).

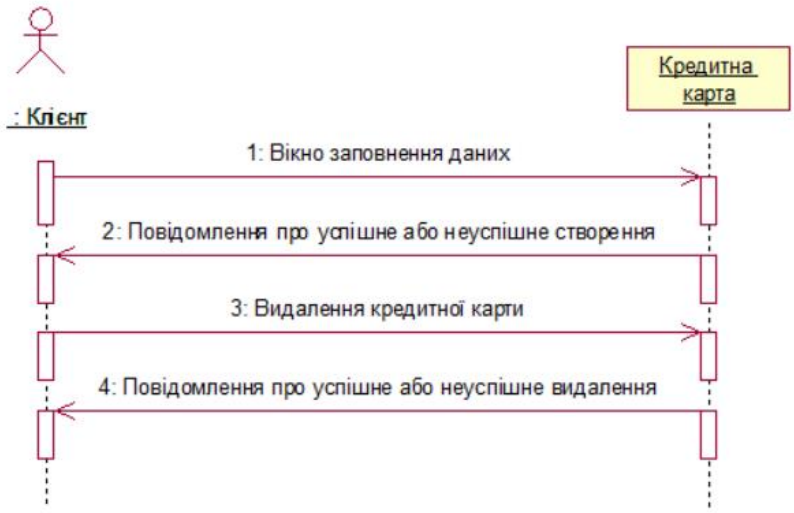


Рис.*17 Створення та видалення кредитної карти*

Діаграма послідовності для варіанта використання: «Співробітники заходять на сторінку адміністрування»

Адміністратор, партнер або кур’єр заходить на сторінку автентифікації, вводить свої дані та в результаті отримує детальний перегляд і деяке управління сайтом. В разі неуспішного входу отримує повідомлення (рис. 18).

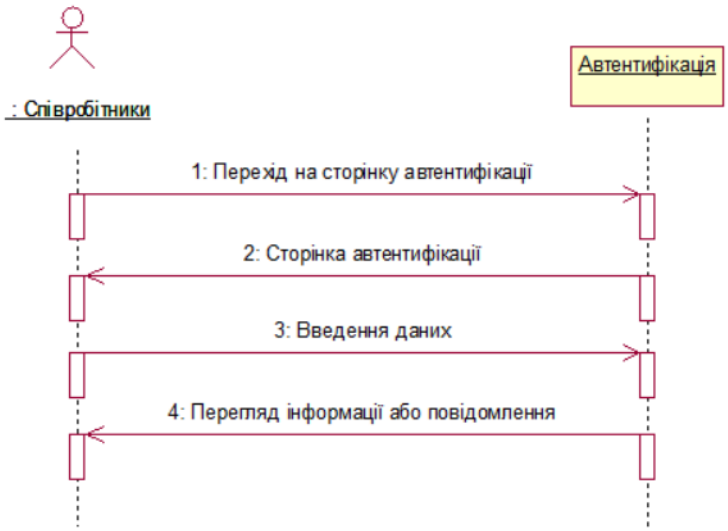


Рис.*18 Співробітники заходять на сторінку адміністрування*

3.5 Діаграма класів

На рисунку 19 зображена діаграма класів, яка побудована на основі монолітної архітектури. Першим чином йде інтерфейс ICrud — з CRUD- операціями, де згодом AbstractService «імплементує» ці методи і надає певну реалізацію за замовчуванням. Після, окремі класи-сервіси, можуть перевантажувати та надавати реалізацію тільки тим методам, які потрібні для роботи. В сервісах лежать репозиторії, які успадковуються від JpaRepository — це додатковий зручний механізм для взаємодії з сутностями бази даних, організації їх у репозиторії, вилучення даних, у випадках для цього буде достатньо оголосити інтерфейс і метод у ньому, без імплементації [23]. Сервіси використовують дані репозиторії за допомогою dependency injection (ін'єкція залежності) в конструкторі, де за «лаштунками» Java Spring Framework за допомогою контейнера «впроваджує» об’єкти в інші об’єкти або «залежності». В контролерах, за допомогою ін'єкції залежностей, отримується змінна сервісу, і за допомогою неї, ми можемо взаїмодіяти зі всіма операціями, які, в свою чергу, працюють з базою даних.

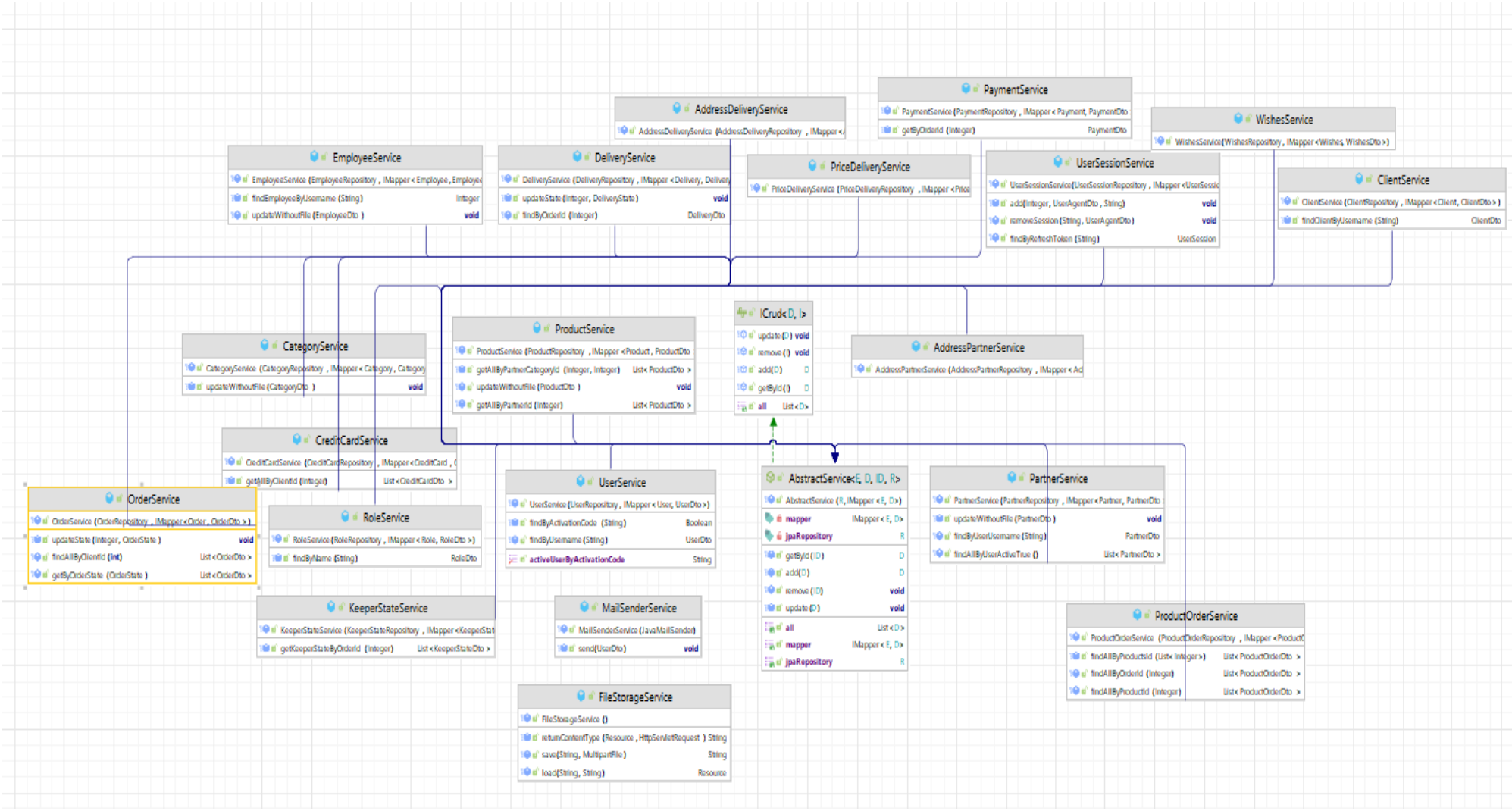


Рис.*19 Діаграма класів монолітної архітектури*

В таблиці 1 відображено мікросервіси, які працюють кожен на своєму конкретному порту. Деякі з них мають можливість взаємодії з базою даних. Тому для них, теж вказані порти на яких вони працюють.

Як видно з таблиці, основні мікросервіси, працюють зі своєю базою даних, а це означає, що побудована мікросервісна архітектура відповідає одному з декількох патернів, а саме це «Одна база даних на один сервіс».

Таблиця *1*

Порти, на яких працюють мікросервіси, а також бази даних з якими вони взаємодіють.

Мікросервіси		**Бази даних**	
Назва сервісу	**Порт**	**Назва баз**	**Порт**
users-service	8081	userdb	5441
products-service	8082	productdb	5442
orders-service	8083	orderdb	5443
payment-service	8084	paymentdb	5444
delivery-service	8085	deliverydb	5445
security-service	8787	userdb	5441
cloud-config	9296		
discrovery	8761		
cloud-gateway	9191		
hystrix-dashboard	9295		

На рисунках 20-22 зображено мікросервісну архітектуру. Вона працює з тим же самим механізмом, як і монолітна архітектура. Але в даному випадку, кожен мікросервіс побудований окремо і працює зі своєю вказаною базою даних.

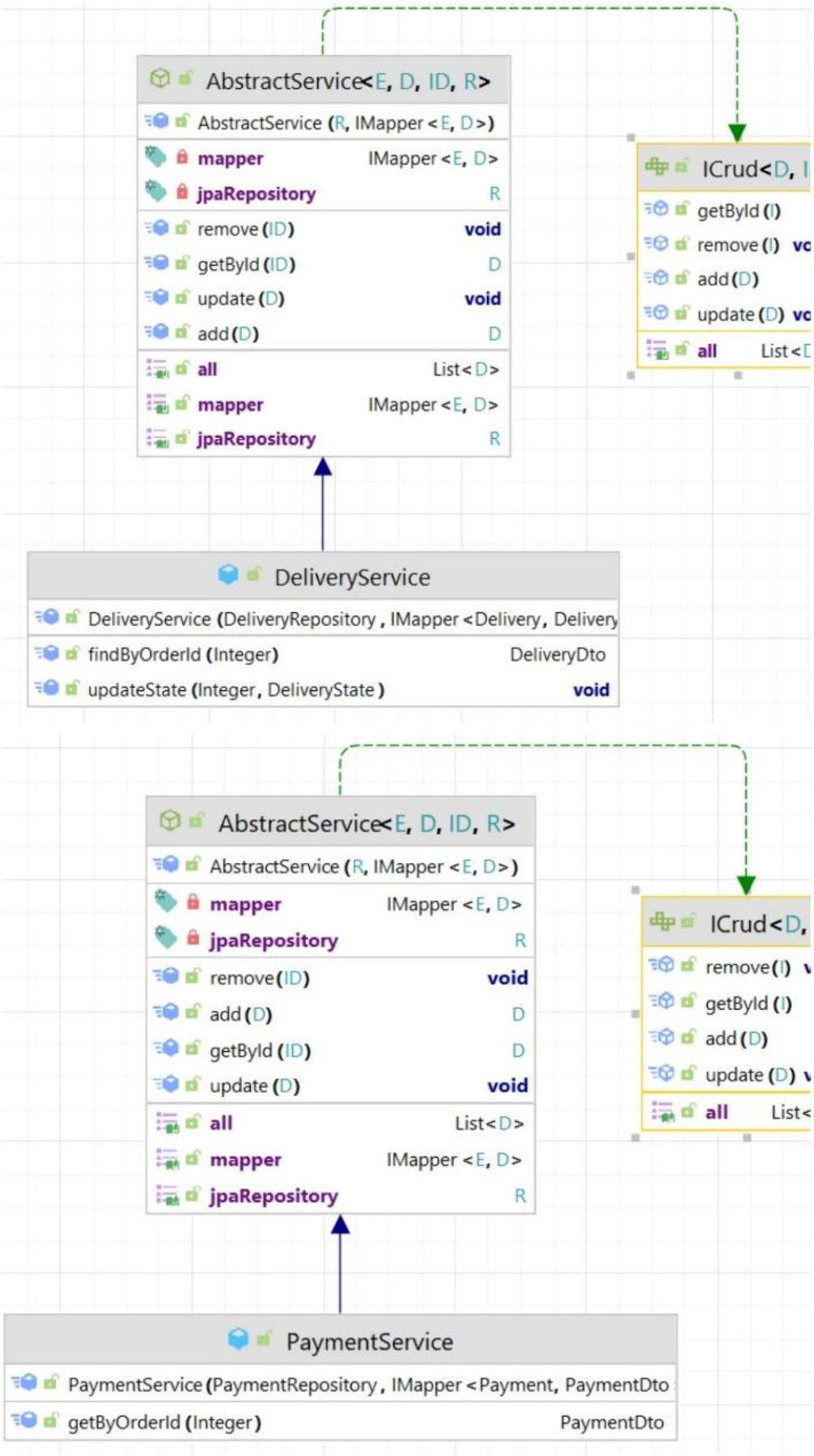


Рис.*20 Діаграма класів мікросервісів «Delivery» та «Payment»*

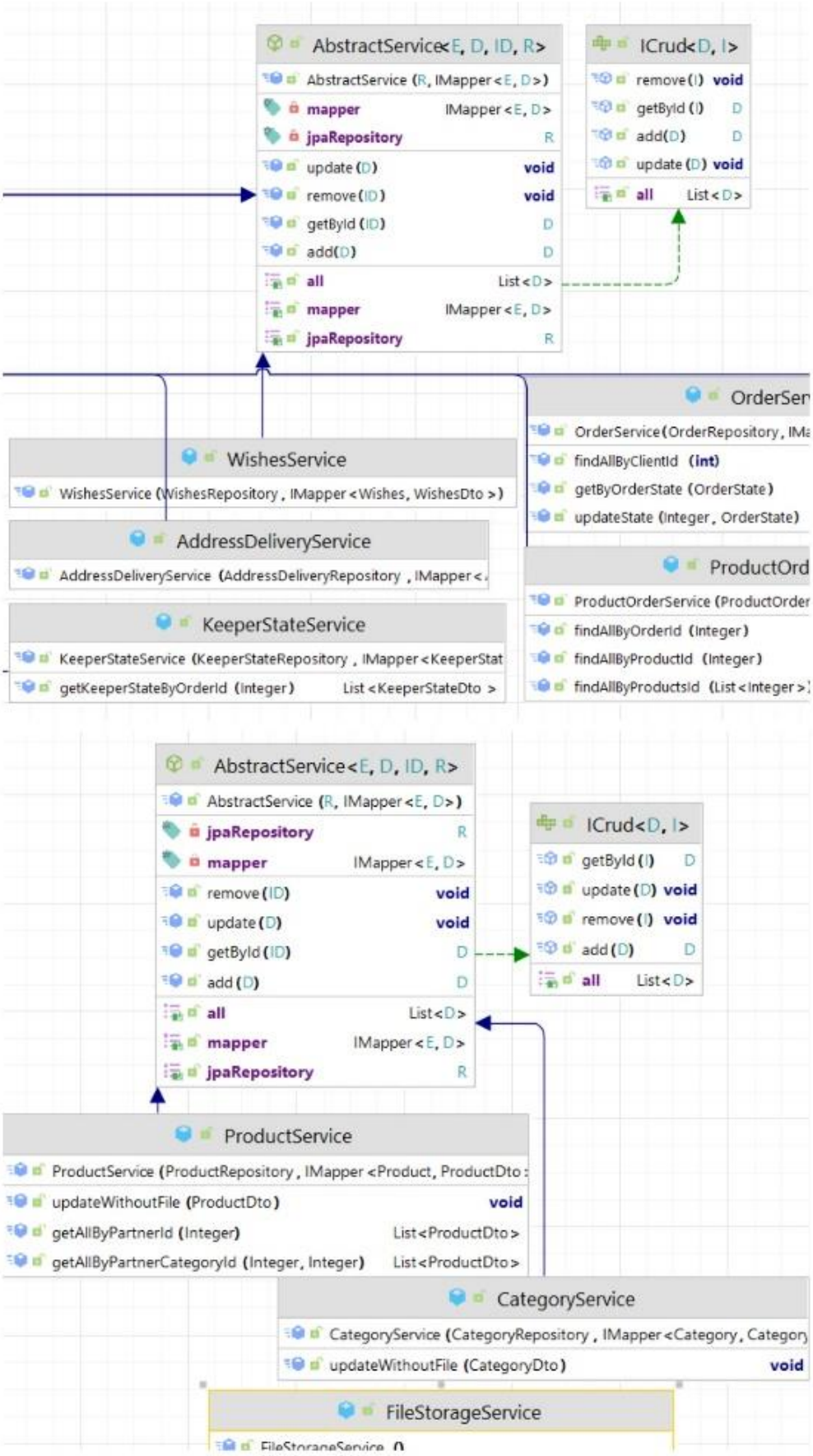


Рис.*21 Діаграма класів мікросервісів «Order» та «Products»*

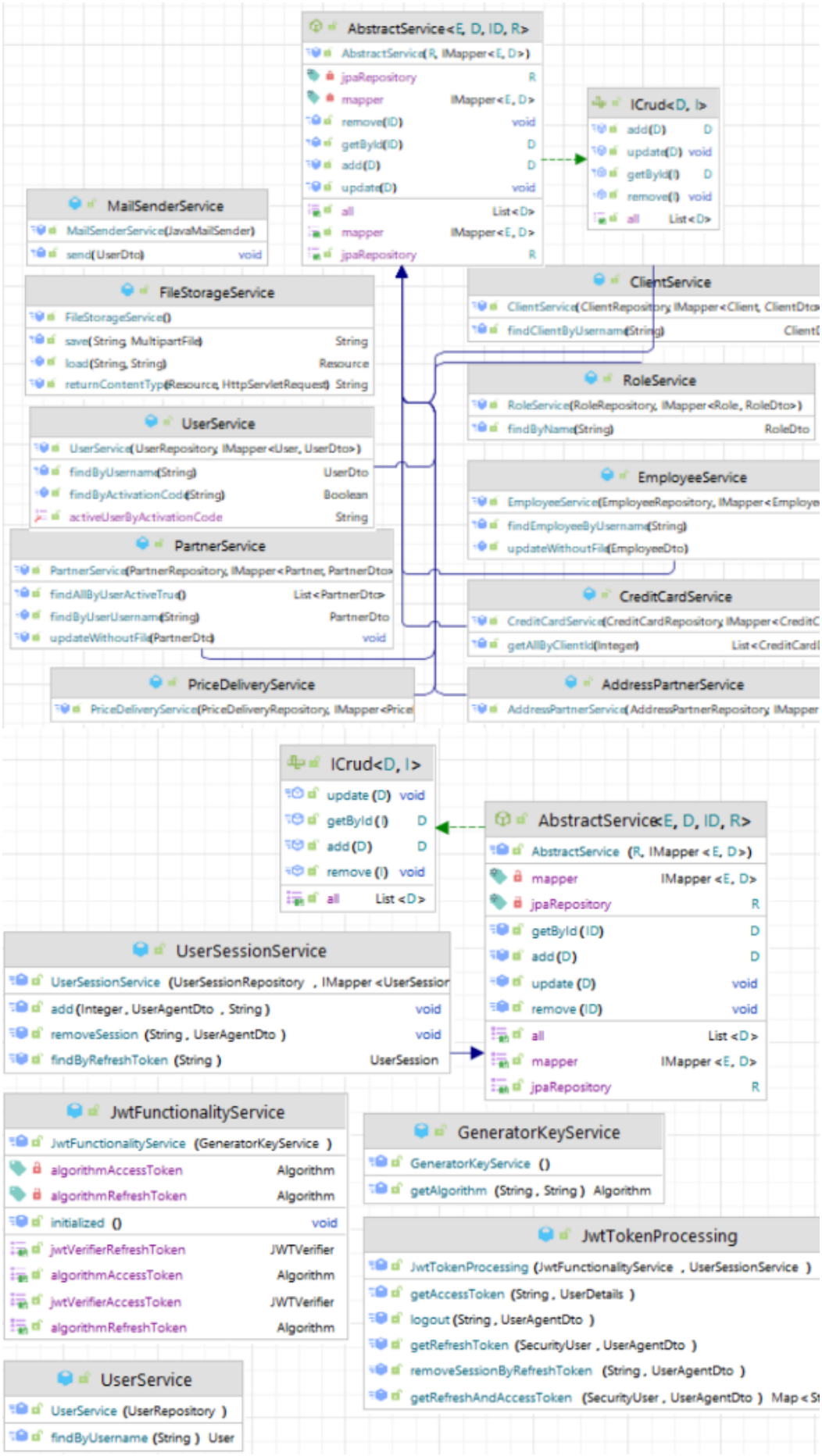


Рис.22 *Діаграма класів мікросервісів «Users» та «Security»*

Нижче наведено більш детальний опис методів для деяких класів, що присутні на діаграмі класів.

На рисунку 23 відображено клас «User», він призначений для взаємодії з таблицею «Users», яка знаходиться в базі даних. Коли отримуються дані з бази, вони конвертуються в поля з такими назвами, і з ними можна працювати як з простим програмним кодом.

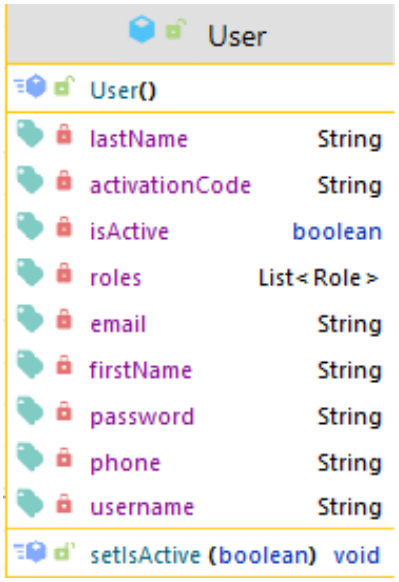


Рис.*23 Клас «User»*

На рисунку 24 відображено клас «Order» він призначений для взаємодії з таблицею бази даних. На рисунку 25 відображено перелічуваний тип даних «OrderState» — призначений для зміни та відображення статусу замовлення.

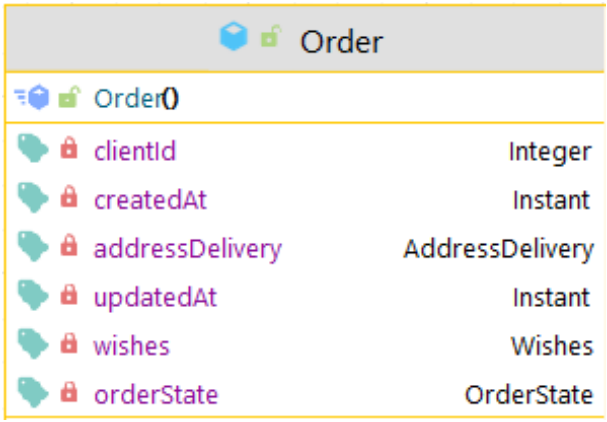


Рис.*24 Клас «Order»*

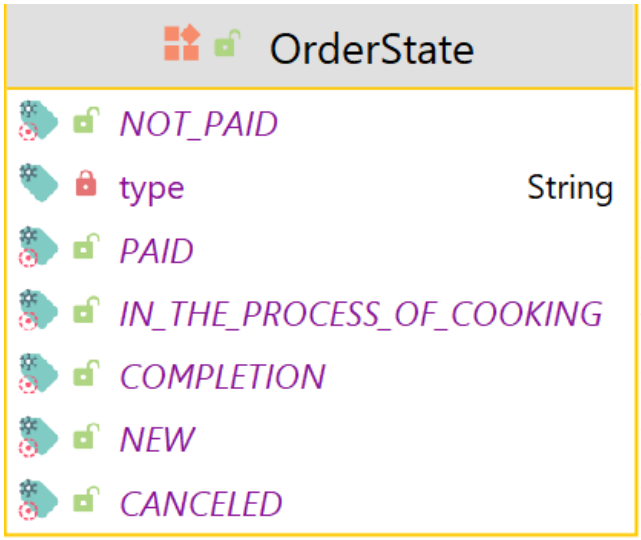


Рис.*25 Перелічувальний тип даних «OrderState»*

3.6 Засоби реалізації

Побудовані корпоративні застосунки були виконані у середовищі розробки IntelliJ Idea, об’єктно-орієнтованою мовою Java, за допомогою Java Spring Framework та Java Spring Cloud — для серверної частини, а саме для побудови монолітної та мікросервісної архітектур. В ролі бази даних використовувалася PostgreSQL — об'єктно-реляційна система керування базами даних. Для клієнтської частини використовувався Angular Framework. Для оновлення міграцій даних використовувався Flyway Migration. Для побудови системи захисту використовувалася бібліотека JWT та вбудована система безпеки Java Spring Security.

3.7. Модулі і алгоритми

В лістингі 1 відображено перевантажений метод «configure()». Він містить в собі налаштування безпеки. Метод захищає запити від не автинтифікованих користувачів, де це потрібно. І, дозволяє проводити запити там, де користувач, може без автентифікації проводити операції.

«http.csrf().disable()» — метод, який виключає csrf-безпеку. CSRF — атака міжсайтового підроблення запитів, яка змушує кінцевого користувача здійснювати небажані виклики на сервери веб-додатків, де кінцевий користувач уже пройшов автентифікацію [24]. Причина вимкнення полягає в тому, що програма відкрита для всіх або, коли перебуває на стадії розробки або тестування.

«mvcMatchers("/assets/**").permitAll()» — метод в якому поміщається певний шаблон, а інший викликаний метод говорить про те, що коли буде приходити запит від клієнта і він буде дорівнювати заданому шаблону, то запит буде дозволено.

«anyRequest().authenticated()», призначений для того, щоб перевіряти будь-які інші запити та обробляти їх тоді, якщо клієнт автентифікований.

Лістинг *1. Налаштування конфігурації безпеки.*

```
    @Override
        public void configure(HttpSecurity http) throws
Exception {
            http
                    .csrf().disable()
.sessionManagement().sessionCreationPolicy(SessionCreationP
olicy.STATELESS).and()

.exceptionHandling().authenticationEntryPoint(jwtAuthentica
tionEntryPoint).and()
                    .authorizeRequests()
                    .mvcMatchers("/assets/**").permitAll()
                    .mvcMatchers(this.api                  +
"/partner").hasAnyAuthority(Roles.ADMIN.getType(),
Roles.PARTNER.getType())
                    .mvcMatchers(this.api                  +
"/role").hasAnyAuthority(Roles.ADMIN.getType(),
Roles.PARTNER.getType())
                    .mvcMatchers(this.api    +    "/address—
partner").hasAnyAuthority(Roles.ADMIN.getType(),
Roles.PARTNER.getType())
                    .mvcMatchers(this.api                  +
"/category").hasAnyAuthority(Roles.ADMIN.getType())
                    .mvcMatchers(this.api                  +
"/product").hasAnyAuthority(Roles.ADMIN.getType(),
Roles.PARTNER.getType())
```

```
                    .mvcMatchers(this.api    +    "/credit—
card").hasAnyAuthority(Roles.CLIENT.getType())
                    .mvcMatchers(this.api                  +
"/product/**").hasAnyAuthority(Roles.ADMIN.getType(),
Roles.PARTNER.getType())
                    .anyRequest().authenticated().and()
                    .logout().logoutUrl(this.api           +
"/auth/logout").and()
                    .addFilterBefore(new
RequestAuthorizationFilter(jwtFunctionalityService,
tokenHeader), UsernamePasswordAuthenticationFilter.class);

        }
```

В лістингі 2 відображно метод «getAll()». «ResponseEntity» відправляє, за допомогою сервісу, список клієнтів та HTTP-код. Якщо виникла проблема отримання даних, глобальний метод слухача помилок, відправить неуспішну відповідь.

Лістинг *2. Метод отримання списку клієнтів*

```
      @GetMapping
        public ResponseEntity<List<ClientDto>> getAll() {
            LOGGER.debug("Received clients!");
            return                                      new
ResponseEntity<>(clientService.getAll(), HttpStatus.OK);
        }
```

В лістингі 3 відображено метод «save()». Він отримує шлях та файл, а далі проводить його обробку. Задається шлях, де потрібно провести збереження. Файл, який може мати однакову назву і може замінити інший, надається унікальна назва. Після, цього він зберігається на сервері, а клієнту повертається успішна або неуспішна відповідь.

Лістинг *3. Збереження файлу на сервері в певній папці.*

```
    public  String  save(String  path,  MultipartFile  file)
throws IOException {
            try {
                Path root = Paths.get(path);
                String              fileName              =
UUID.randomUUID().toString()        +         "—"         +
file.getOriginalFilename();
                Path filePath = root.resolve(fileName);
```

```
                if (Files.exists(filePath)) {
                    fileName = UUID.randomUUID().toString()
+ "—" + file.getOriginalFilename();
                    filePath = root.resolve(fileName);
                }
                Files.copy(file.getInputStream(),
filePath);
                return fileName;
            } catch (IOException e) {
                logger.error(e.getMessage(), e);
                throw new IOException(e.getMessage());
            }
        }
```

В лістингі 4 відображено метод «loadFileWithResources()», його призначення полягає в тому, щоб обробити майбутній файл. Ми отримуємо шлях файлу та його назву. Далі сервер переглядає його за заданим шляхом. Якщо файл, який потрібно завантажити з сервера — існує, він його повертає користувачу. В іншому разі, якщо файлу не існує, відправиться повідомлення про помилку. А до журналу логування, буде занесено відповідний запис.

Лістинг *4. Завантаження файлу з сервера, з певної папки.*

```
    public Resource loadFileWithResources(String path,
String filename) throws IOException, URISyntaxException {
            String root = path + "/" + filename;
            try (InputStream url =
this.getClass().getClassLoader().getResourceAsStream(root);
) {
                if (url != null) {
                    Resource resource = new
ByteArrayResource(url.readAllBytes());
                    if (resource.exists() ||
resource.isReadable()) {
                        return resource;
                    } else {
                        throw new
FileNotFoundException("Could not read the file!");
                    }
                } else {
                    throw new FileNotFoundException("File
does not exists!");
                }
            } catch (FileNotFoundException e) {
                logger.error(e.getMessage(), e);
```

```
                throw new FileNotFoundException("Error: " +
e.getMessage());
            }
        }
```

В лістингі 5 відображено метод «changeOrderPaymentState()» — призначений для зміни статусу оплати та подальшого збереження в базу даних. Коли клієнт зробив замовлення і вибрав статус оплати, для кур'єра надається в подальшому інформація про вид оплати.

Лістинг *5. Зміна статусу оплати та збереження його в базу даних.*

```
    @GetMapping("order—payment/{order—id}")
        public                          ResponseEntity<String>
changeOrderPaymentState(@PathVariable("order—id")    Integer
orderId) {
            PaymentDto              payment               =
this.paymentService.getByOrderId(orderId);
            OrderState state =

payment.getPaymentType().equals(PaymentType.MONEY.getType()
)
                                ? OrderState.NOT_PAID
                                : OrderState.PAID;
            this.orderService.updateState(orderId, state);
            LOGGER.debug("Received   order—state   {}   !",
state);
            return   new   ResponseEntity<>(state.getType(),
HttpStatus.OK);}
```

3.8. Проєкт інтерфейсу

Головним критерієм в розробці інтерфейсу є простота та легкість в розумінні і використанні, тільки тоді даний програмний продукт стане популярним, масовим у своєму вжитку для різних категорій людей.

Ілюстрації представленні нижче (Рис.26-32), демонструють сайт, який був виконаний і побудований за допомогою Angular Framework.

Рис.26 *Головна сторінка при вході*

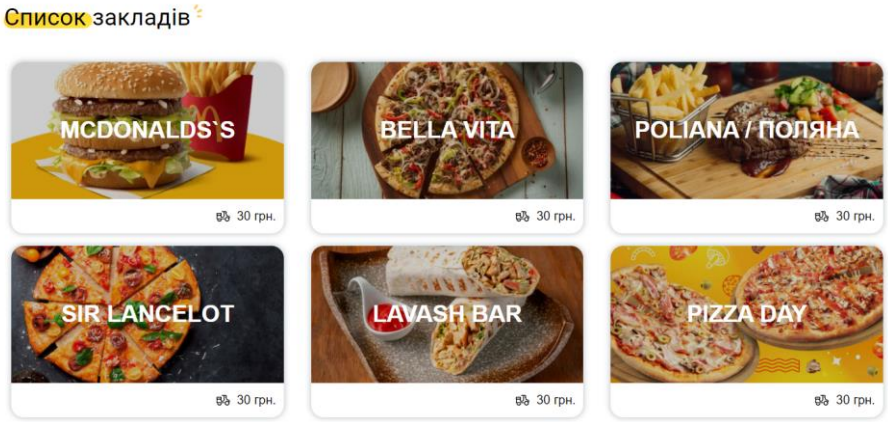


Рис.27 *Список партнерів, де можна зробити замовлення*

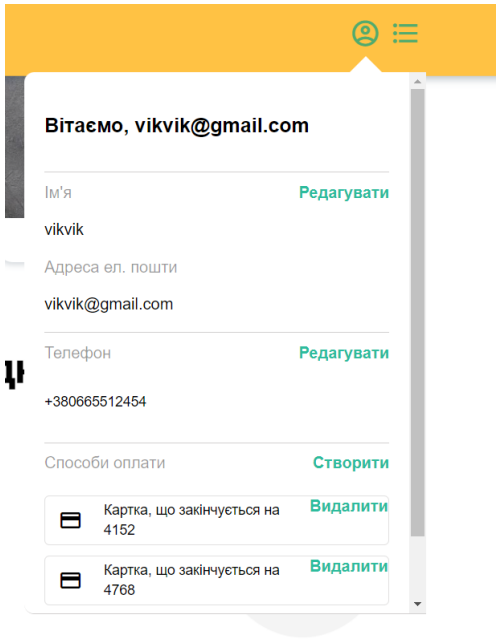


Рис.28 Персональний кабінет клієнта

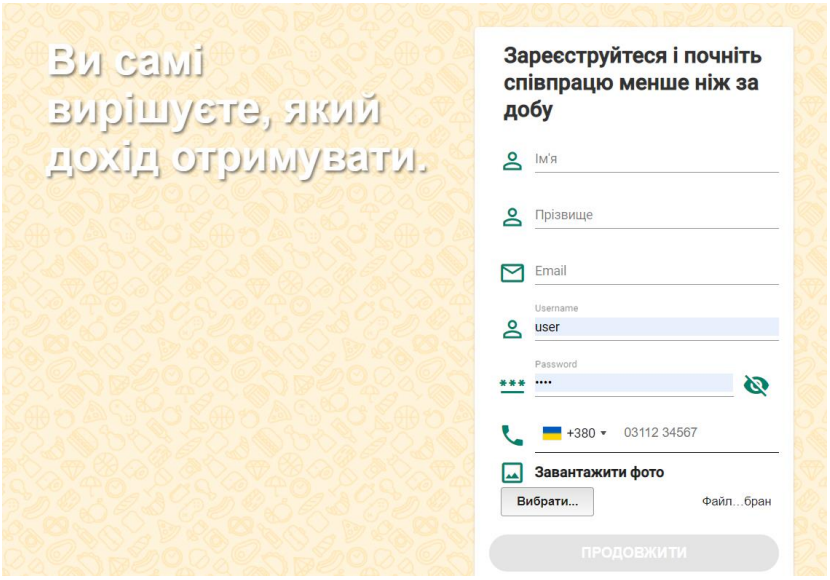


Рис.*29 Сторінка реєстрації нового співробітника*

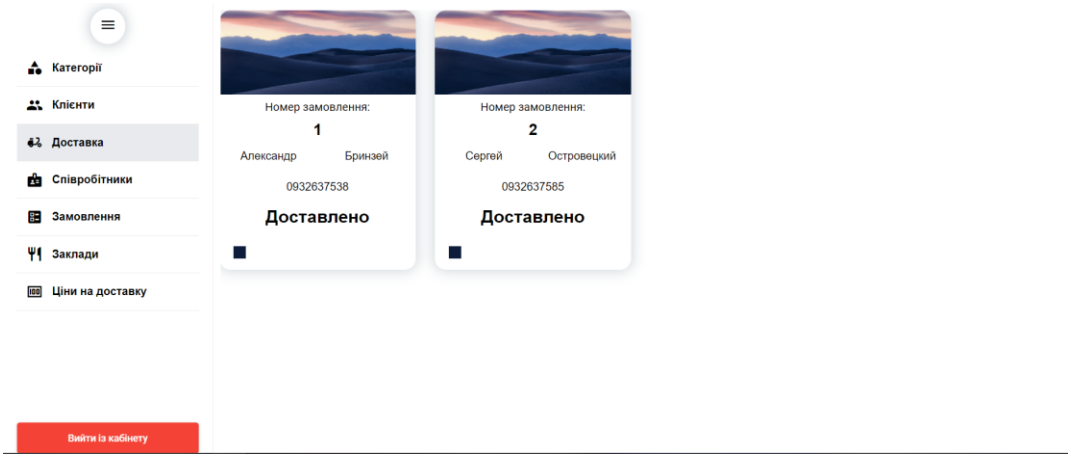


Рис.*30 Сторінка адміністраторів*

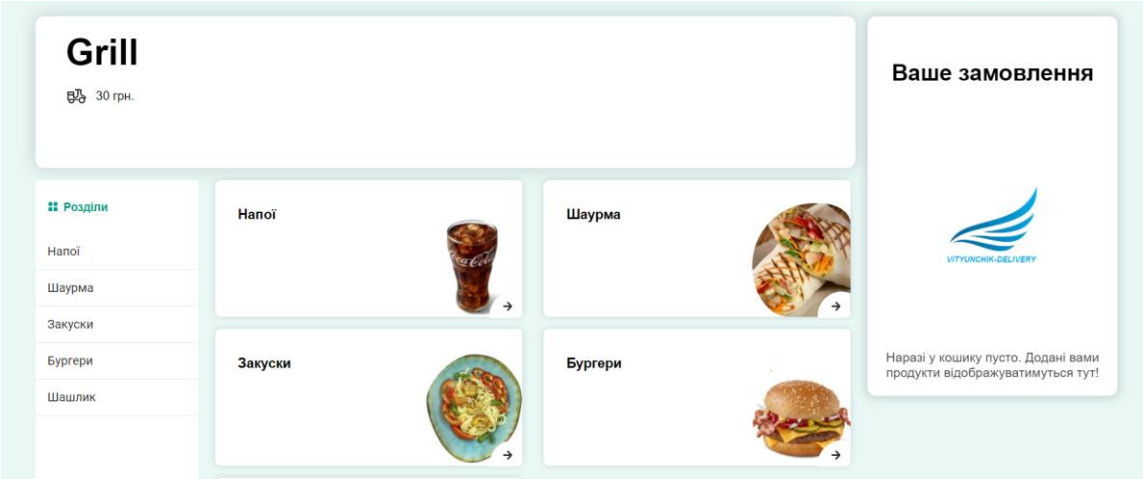


Рис.*31 Сторінка категорій партнера*

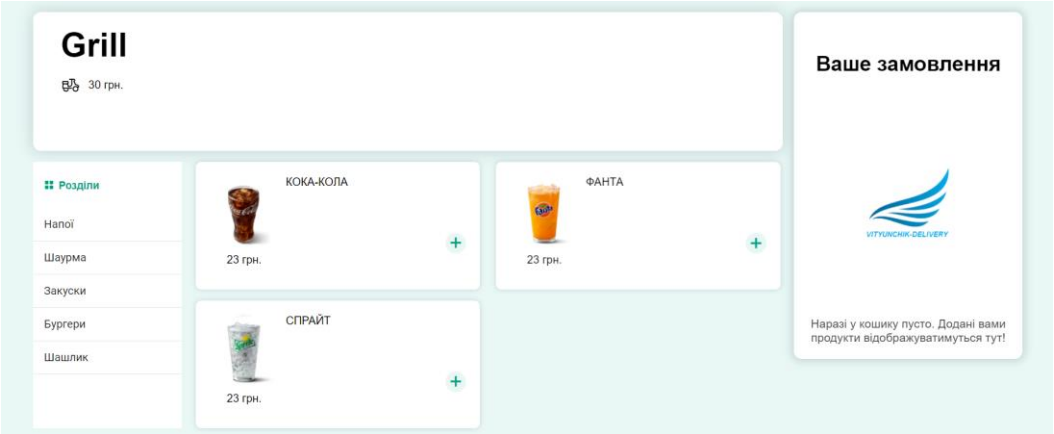


Рис.*32 Сторінка продуктів партнера*

3.9 Висновки з розділу 3

В даному розділі було реалізовано застосунки на основі монолітної та мікросервісної архітектур; проведено аналіз функціональних та нефункціональних вимог застосунків; побудовано діаграми класів, діаграми використання, діаграми послідовності, тощо. На основі цього, можна зробити такі висновки:

1. Було реалізовано корпоративні застосунки на основі монолітної та мікросервісної архітектури.

2. Побудовано та надано опис діаграми класів монолітної та мікросервісної архітектур.

3. Використовувалися бібліотеки Flyway, Log4j, HikariCP.

4. Для клієнтської частини використовувався Angular Framework.

5. Для серверної частини використовувалися Java Spring Framework та Java Spring Cloud.

6. Додаток має можливість замовлення та оплати продуктів.

7. Клієнт, адміністратор, кур’єр та партнери мають свої персональні кабінети, з яких вони можуть управляти операціями.

8. Застосунок має стильний дизайн та легкий у керуванні функціонал.

9. Монолітна та мікросервісна архітектура взаємодіють з базою даних PostgreSql.

РОЗДІЛ 4 ДОСЛІДЖЕННЯ РЕЗУЛЬТАТІВ РОБОТИ МОНОЛІТНОГО І МІКРОСЕРВІСНОГО ЗАСТОСУНКІВ

4.1 Розгортання монолітного та мікросервісного застосунків

4.1.1 Розгортання монолітного застосунку за допомогою інструменту Maven

Maven — це один із популярних інструментів створення корпоративних Java-проектів з відкритим вихідним кодом, призначений для виконання значної частини важкої роботи поза процесом побудови [25]. Головним аспектом проекту Maven 2 є об’єктна модель POM. Вона містить опис проєкту, включаючи інформацію про керування версіями та налаштуваннями, залежності, програми та ресурси тестування, та багато іншого. POM приймає форму файлу XML (pom.xml), який розміщується в домашньому каталозі проєкту, наприкалад як відображено в лістингі 6.

Лістинг 6. *XML-файл монолітної архітектури.*

```
<?xml version="1.0" encoding="UTF—8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema—instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.
0.0 https://maven.apache.org/xsd/maven—4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring—boot—starter—
parent</artifactId>
        <version>2.5.6</version>
        <relativePath/>
    </parent>
    <groupId>com.server</groupId>
```

```
        <artifactId>itdelivery</artifactId>
        <version>0.0.1—SNAPSHOT</version>
        <name>itdelivery</name>
        <description>itdelivery server</description>
        <properties>
            <java.version>14</java.version>
        </properties>
        <packaging>jar</packaging>
        <dependencies>
            <dependency>
                <groupId>com.zaxxer</groupId>
                <artifactId>HikariCP</artifactId>
            </dependency>
        <build>
            <plugins>
                <plugin>
        <groupId>org.springframework.boot</groupId>
            <artifactId>spring—boot—maven—
plugin</artifactId>
                </plugin>
            </plugins>
            <finalName>itdelivery</finalName>
        </build>
    </project>
```

Розгортання корпоративного застосунку на основі монолітної архітектури, відбувалося в декілька етапів [25]:

1. Спочатку виконувалася команда clean — для очищення проекту.
2. Команда validate — виконує перевірку, на те, чи є структура проекту повною та правильною.
3. Команда compile — компілює вихідні тексти програми.
4. Команда test — тестує зібраний код заздалегідь підготовленим набором тестів.

5. Команда package — упаковка відкомпільованих класів та інших ресурсів у JAR-файл.
6. install — встановлення програмного забезпечення на локальний Maven-репозиторій, щоб зробити його доступним для інших проектів.
7. deploy — стабільна версія програмного забезпечення, поширюється на віддалений Maven-репозиторій, щоб зробити його доступним для інших користувачів.

4.1.2 Розгортання мікросервісного застосунку за допомогою інструментів Docker та Kubernates

Kubernates — це інтрумент, який призначений для керування кластером контейнерів Linux як єдиної системи [26]. Kubernetes управляє та запускає контейнери Docker на великій кількості хостів. Він забезпечує розміщення та реплікацію великої кількості контейнерів. Проект був заснований компанією Google, а також підтримується такими компаніями, як IBM, Microsoft, RedHat і Docker. Kubernetes дозволяє розробникам розгортати свої програми самостійно. Він, також допомагає, автоматично відстежувати та перепланувати застосунки в разі збою обладнання.

Kubernetes абстрагує апаратну інфраструктуру та представляє весь центр обробки даних, як деякий єдиний великий обчислювальний ресурс. Він дозволяє розгортати та запускати компоненти програмних систем. Kubernetes вибирає сервер для кожного компонента, розгортає його, та надає змогу легкого знаходження та спілкування зі всіма іншими компонентами програми. Він використовується в найбільших центрах обробки даних, наприклад в тих, які призначені, для хмарних обчислень. Kubernetes дозволяє пропонувати розробникам просту платформу для розгортання та запуску, при цьому не вимагаючи якихось знань про десятки тисяч запущених програм, які запущені на апаратному забезпеченні. Велика кількість компаній визнають Kubernetes, як найкращий спосіб запуску застосунків. Він стає стандартним способом запуску розподілених систем як у хмарі, так і на локальній машині.

Проєкт Kubernates надає такі механізми, які при використанні технології Docker, дозволяють масштабувати та запускати контейнери на великій кількості хостів, а також виконувати їх балансування. Інструменти Kubernates має важливі концепції, які призначені для розгортання корпоративного застосунку [27]:

1) вузол (node) — він являє собою деяку фізичну або віртуальну машину, на якій в свою чергу розгорнуті контейнери Docker. Кожен вузол має свій сервіс запуску контейнерів та має додаткові компоненти, які призначені для центрального управління вузлом;
2) поди (pods) — це компонент запуску та управління одного або декількох контейнерів, які будуть працювати на якомусь одному вузлі, де в свою чергу буде забезпечуватися розділення ресурсів та взаємодія між процесами;
3) томи (volume) — певний сервіс збереження файлів або інших компонентів, якими можуть користуватися контейнери, які розгорнуті в конкретному поді.

Кожен з вказаних вище об’єктів управління помічаються мітками та селекторами. Вони є важливим механізмом Kubernetes — дозволяють вибрати, який з об’єктів треба використовувати для запрошуваної операції.

Також в інструменті Kubernates присутні такі концепції:

4) контролер (controller) — це процес, який призначений для управління станом кластера, який виконується в свою чергу за допомогою налаштованих міток;
5) сервіси (services) — сукупність пов'язаних подів, де доступ та їх виконання відбувається за допомогою налаштованих міток та селекторів;
6) kubectl — інтерфейс командного рядка, який призначений для управління командами Kubernates;
7) minikube — спеціалізована конфігурація Kubernetes, призначена для розгортання на локальній машині (наприклад, комп'ютер

розробника), застосовується для вивчення та локальних експериментів над Kubernetes [27].

Docker — це інструмент, який дозволяє автоматизувати розгортання та управління застосунками в середовищах з підтримкою контейнеризації [28]. Він дозволяє упаковувати застосунки із всіма їх залежностями в контейнер, який в свою чергу в подальшому може бути розгорнутим в будь-якій системі.

Розгортання мікросервісів відбувалося в декілька етапів. Спочатку для кожного мікросервісу був налаштований Dockerfile (Лістинг 7). Він містить в собі налаштування, які дозволяють запускати jar-файл на вказаному порту.

Лістинг *7. Docker-файл мікросервісу «Delivery»*.

```
# syntax=docker/dockerfile:1
FROM openjdk:14.0.1
ADD target/delivery—service—server.jar delivery—service—server.jar
ENTRYPOINT ["java", "—jar", "/delivery—service—server.jar"]
EXPOSE 8084
```

Далі в поточній директорії, де знаходиться Dockerfile, виконуються команда «docker build -t vitnov29/deliveryservice .», яка призначена для створення образу з вказаною назвою. Потів виконується команда «docker push vitnov29/deliveryservice:latest», яка в свою чергу, надсилає дане зображення на віддалене особисте сховище з вказаною версією.

Далі йде розгортання данного зображення за допомогою інструменту Kubernates. В лістингі 8 зображено всі налаштування розгортання мікросервісу:

1) apiVersion — версія API, що використовується для створення об'єкта Kubernetes;
2) kind — тип створюваного об'єкту;
3) metadata — дані, що дозволяють ідентифікувати об'єкт;
4) spec — значення, що потребує стан об'єкту;

5) replicas — вказує, скільки модулів створити під час розгортання.

Для розгортання даного мікросервісу вказується версія, далі вказуються метадані та кількість модулів, які потрібно створити. Після цього, вказано стан об'єкту для завантаження зображення з віддаленого особистого сховища Docker та на якому порті його запускати.

Лістинг *8. Yaml-файл, який призначений для розгортання мікросервісу «Payment».*

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: delivery—service—app
  labels:
    app: delivery—service—app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: delivery—service—app
  template:
    metadata:
      labels:
        app: delivery—service—app
    spec:
      containers:
        — name: delivery—service—app
          image: vitnov29/deliveryservice:latest
          imagePullPolicy: Always
          ports:
            — containerPort: 8085
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: delivery—service—svc
spec:
  ports:
    — targetPort: 8085
      port: 80
  selector:
    app: delivery—service—app
```

Далі за допомогою команд «kubectl», всі мікросервіси з налаштованими для розгортання файлами додати за запустити.

4.1.3 Розгортання мікросервісного застосунку за допомогою інструментів Docker та Docker Compose

Docker Compose — це інструмент для визначення, розгортання та запуску багатоконтейнерних програм Docker [29]. Використання даного інструменту складається з декількох етапів:

1. Спочатку потрібно визначити Dockerfile в папці проєкту.
2. Далі створити файл «docker-compose.yaml» та визначити в ньому служби, які призначені, наприклад, для розгортання клієнтського застосунку, сервера, баз даних, тощо.
3. Виконати запуск даного файлу за допомогою команди «docker- compose up».

В лістингі 9 відображено налаштування, яке призначено для розгортання мікросервісу «Users». В ньому присутній сервіс з змінними середовища, також присутні томи, для того, щоб мати можливість збереження даних. Вказано порт, на якому буде запускати та слухати контейнер. Вказано додаткові налаштування, для того, щоб можна було переглядати додаткову інформацію в командному рядку. Також, існують залежності від бази даних та сервіса пошуку мікросервісів та вказано зображення, яке буде завантажуватися з

віддаленого особистого сховища Docker. Далі, даний файл запускається, та надає доступ до всіх сервісів, які були налаштовані в ньому.

Лістинг *9. «docker-compose.yaml», який призначений для розгортання мікросервісу «Users».*

```
    usersservice:
        environment:
          EUREKA_SERVER:                    http://service—
registry:8761/eureka
          EMPLOYEE_PICTURE: public/employee—picture
          PARTNER_PICTURE: public/partner—picture
        volumes:
          — "C:/Users/admin/Desktop/public/:/public"
        ports:
          — 8081:8081
        stdin_open: true
        tty: true
        container_name: users—service—server
        depends_on:
          — postgresuserdb
          — service—registry
        image: vitnov29/users—service
```

4.2 Порівняння показників монолітної та мікросервісної архітектур

Коли, застосунки вже побудовані, їм потрібно провести тестування. Для того, щоб визначити, яка ж з архітектур — монолітна чи мікросервісна, мають свої переваги та недоліки. Їх потрібно порівняти за показаниками, такими, як швидкість відповіді на велику кількість запитів, кількість провалених запитів, середнє значення відповіді, коефіцієнт помилок, пропускна здатність, тощо. Щоб перевірити дані показники, існують деякі бібліотеки навантаження, одними з них вважаються бібліотеки: JMeter та Gatling.

JMeter — це інструмент, який призначений для проведення навантажувального тестування [30]. Він надає можливість імітувати велику кількість запитів, та переглядати інформацію про їх стан в вигляді таблиць, графів, дерев, діграм, тощо. Інструмент має механізм авторизації та зміни заголовків запитів.

Gatling — це інструмент, який призначений для проведення навантажувального тестування та визначення продуктивності застосунків [31]. Він надає можливість імітувати велику кількість запитів за секунду та надає детальний перегляд інформації в html- сторінці про розподіл часу відповіді, кількість користувачів, кількість успішних та провалених запитів тощо.

4.2.1 Показники тестування в інструменті JMeter

На рисунку 33-34 зображено показники, які були отримані при тестуванні запитів отримання даних співробітників. Кількість запитів дорівнювала 1000. Переглядаючи дані показники, можна зробити такий висновок, що монолітна архітектура переважає за мікросервісну по таким показникам:

1. Середнє значення відповіді монолітної архітектури — 1308 мс, мікросервісної — 2080 мс.
2. Мінімальний час відповіді монолітної архітектури — 135 мс, мікросервісної — 156 мс.
3. Максимальний час відповіді монолітної архітектури — 3902 мс, мікросервісної — 4436 мс.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request E...	1000	1308	135	3902	860.01	0.00%	241.0/sec	1431.77	463.92	6083.0
TOTAL	1000	1308	135	3902	860.01	0.00%	241.0/sec	1431.77	463.92	6083.0

Рис.33 *Отримання 1000 даних співробітників на основі монолітної архітектури*

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request Em...	1000	2080	156	4436	744.64	0.00%	221.2/sec	1573.86	425.75	7286.2
TOTAL	1000	2080	156	4436	744.64	0.00%	221.2/sec	1573.86	425.75	7286.2

Рис.*34 Отримання 1000 даних співробітників на основі мікросервісної архітектури*

На рисунку 35-36 зображено показники, які були отримані при тестуванні запитів створення продуктів. Кількість запитів дорівнювала 700. Переглядаючи дані показники, можна зробити такий висновок, що мікросервісна архітектура переважає за монолітну по таким показникам:

1. Середнє значення відповіді мікросервісної архітектури — 929 мс, монолітної — 1827 мс.
2. Мінімальний час відповіді мікросервісної архітектури — 70 мс, монолітної — 769 мс.
3. Пропускна здатність мікросервісної архітектури — 198.2/сек, монолітної — 217.9/сек.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	700	1827	769	2869	464.57	0.00%	217.9/sec	137.02	19029.23	644.0
TOTAL	700	1827	769	2869	464.57	0.00%	217.9/sec	137.02	19029.23	644.0

Рис.*35 Створення 700 продуктів на основі монолітної архітектури*

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	700	929	70	2880	456.41	0.00%	198.2/sec	119.71	17312.52	618.3
TOTAL	700	929	70	2880	456.41	0.00%	198.2/sec	119.71	17312.52	618.3

Рис.*36 Створення 700 продуктів на основі мікросервісної архітектури*

На рисунку 37-40 зображено показники, які були отримані при тестуванні запитів видалення продуктів. Кількість запитів дорівнювала 1000. Переглядаючи дані показники, можна зробити такий висновок, що монолітна архітектура переважає за мікросервісну по таким показникам:

1. Середнє значення відповіді монолітної архітектури — 916 мс, мікросервісної — 3125 мс.
2. Мінімальний час відповіді монолітної архітектури — 124 мс, мікросервісної — 1937 мс.
3. Максимальний час відповіді монолітної архітектури — 2499 мс, мікросервісної — 4611 мс.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	916	124	2499	564.74	1.10%	391.1/sec	216.68	577.08	567.3
TOTAL	1000	916	124	2499	564.74	1.10%	391.1/sec	216.68	577.08	567.3

Рис.*37 Видалення 1000 продуктів на основі монолітної архітектури*

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
976	20:52:35.046	DELETE Thread 1-...	HTTP Request	2298	✓	392	1511	2298	1
977	20:52:35.055	DELETE Thread 1-...	HTTP Request	2289	✓	392	1511	2289	2
978	20:52:35.018	DELETE Thread 1-...	HTTP Request	2330	✓	392	1511	2330	1
979	20:52:35.126	DELETE Thread 1-...	HTTP Request	2224	✓	392	1511	2224	1
980	20:52:34.982	DELETE Thread 1-...	HTTP Request	2368	✓	392	1511	2368	2
981	20:52:35.048	DELETE Thread 1-...	HTTP Request	2302	✓	392	1511	2302	1
982	20:52:35.075	DELETE Thread 1-...	HTTP Request	2275	✓	392	1511	2275	2
983	20:52:34.985	DELETE Thread 1-...	HTTP Request	2367	✓	392	1511	2367	1
984	20:52:35.749	DELETE Thread 1-...	HTTP Request	1604	✗	16588	1511	1604	2
985	20:52:35.021	DELETE Thread 1-...	HTTP Request	2333	✗	16588	1511	2332	1
986	20:52:35.082	DELETE Thread 1-...	HTTP Request	2274	✓	392	1511	2274	1
987	20:52:35.062	DELETE Thread 1-...	HTTP Request	2294	✗	16588	1511	2294	1
988	20:52:35.024	DELETE Thread 1-...	HTTP Request	2332	✓	392	1511	2332	1
989	20:52:36.140	DELETE Thread 1-...	HTTP Request	1220	✗	16588	1511	1219	2
990	20:52:35.478	DELETE Thread 1-...	HTTP Request	1881	✗	16588	1511	1881	1
991	20:52:35.043	DELETE Thread 1-...	HTTP Request	2317	✓	392	1511	2317	1
992	20:52:35.077	DELETE Thread 1-...	HTTP Request	2283	✓	392	1511	2283	1
993	20:52:35.016	DELETE Thread 1-...	HTTP Request	2359	✓	392	1511	2359	1
994	20:52:34.977	DELETE Thread 1-...	HTTP Request	2398	✓	392	1511	2398	0
995	20:52:34.976	DELETE Thread 1-...	HTTP Request	2399	✓	392	1511	2399	1
996	20:52:34.960	DELETE Thread 1-...	HTTP Request	2415	✓	392	1511	2415	1
997	20:52:34.978	DELETE Thread 1-...	HTTP Request	2397	✓	392	1511	2397	2
998	20:52:35.343	DELETE Thread 1-...	HTTP Request	2035	✗	16588	1511	2034	1
999	20:52:34.879	DELETE Thread 1-...	HTTP Request	2499	✗	16588	1511	2498	1
1000	20:52:34.982	DELETE Thread 1-...	HTTP Request	2401	✓	392	1511	2401	2

Рис.*38 Детальна інформація видалення 1000 продуктів на основі монолітної архітектури*

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	1000	3125	1937	4611	643.75	0.20%	211.1/sec	78.45	311.69	380.5
TOTAL	1000	3125	1937	4611	643.75	0.20%	211.1/sec	78.45	311.69	380.5

Рис.*39 Видалення 1000 продуктів на основі мікросервісної архітектури*

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
976	20:23:54.947	DELETE Thread 1-21...	HTTP Request	4396		346	1512	4396	2
977	20:23:55.570	DELETE Thread 1-63...	HTTP Request	3774		346	1512	3774	1
978	20:23:54.766	DELETE Thread 1-47	HTTP Request	4578		346	1511	4578	1
979	20:23:55.884	DELETE Thread 1-88...	HTTP Request	3460		346	1512	3460	1
980	20:23:54.834	DELETE Thread 1-11...	HTTP Request	4512		346	1512	4512	1
981	20:23:55.502	DELETE Thread 1-57...	HTTP Request	3846		346	1512	3846	1
982	20:23:55.071	DELETE Thread 1-29...	HTTP Request	4277		346	1512	4277	1
983	20:23:55.583	DELETE Thread 1-65...	HTTP Request	3765		346	1512	3765	1
984	20:23:54.997	DELETE Thread 1-26...	HTTP Request	4352		346	1511	4352	1
985	20:23:54.862	DELETE Thread 1-13...	HTTP Request	4490		346	1511	4490	1
986	20:23:55.381	DELETE Thread 1-44...	HTTP Request	3971		346	1512	3971	2
987	20:23:54.770	DELETE Thread 1-51	HTTP Request	4583		346	1511	4583	1
988	20:23:54.837	DELETE Thread 1-92	HTTP Request	4516		346	1512	4516	2
989	20:23:54.835	DELETE Thread 1-11...	HTTP Request	4518		346	1512	4518	1
990	20:23:55.169	DELETE Thread 1-29...	HTTP Request	4185		346	1512	4185	1
991	20:23:54.777	DELETE Thread 1-57	HTTP Request	4577		346	1512	4577	0
992	20:23:55.290	DELETE Thread 1-48...	HTTP Request	4066		346	1512	4066	1
993	20:23:54.747	DELETE Thread 1-29	HTTP Request	4611		346	1512	4611	2
994	20:23:54.973	DELETE Thread 1-24...	HTTP Request	4385		346	1512	4385	1
995	20:23:54.874	DELETE Thread 1-14...	HTTP Request	4484		346	1512	4484	1
996	20:23:55.310	DELETE Thread 1-50...	HTTP Request	4049		346	1512	4049	1
997	20:23:54.949	DELETE Thread 1-23...	HTTP Request	4413		346	1512	4413	1
998	20:23:55.678	DELETE Thread 1-94...	HTTP Request	3707		346	1511	3707	1
999	20:23:55.139	DELETE Thread 1-33...	HTTP Request	4314		15478	1512	4314	2
1000	20:23:55.367	DELETE Thread 1-54...	HTTP Request	4087		19678	1511	4086	1

Рис.*40 Детальна інформація видалення 1000 продуктів на основі мікросервісної архітектури*

На рисунку 41-44 зображено показники, які були отримані при тестуванні запитів створення клієнтів. Кількість запитів дорівнювала 500. Переглядаючи дані показники, можна зробити такий висновок, що монолітна архітектура переважає за мікросервісну по таким показникам:

1. Середнє значення відповіді монолітної архітектури — 8 мс, мікросервісної — 85 мс.
2. Мінімальний час відповіді монолітної архітектури — 4 мс, мікросервісної — 8 мс.
3. Максимальний час відповіді монолітної архітектури — 23 мс, мікросервісної — 423 мс.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	500	8	4	23	2.43	1.80%	473.5/sec	249.83	287.14	540.3
TOTAL	500	8	4	23	2.43	1.80%	473.5/sec	249.83	287.14	540.3

Рис.*41 Створення 500 клієнтів на основі монолітної архітектури*

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
476	20:47:59.898	POST Client Threa...	HTTP Request	8		545	621	8	1
477	20:47:59.903	POST Client Threa...	HTTP Request	8		545	621	8	1
478	20:47:59.905	POST Client Threa...	HTTP Request	7		545	621	7	1
479	20:47:59.901	POST Client Threa...	HTTP Request	12		284	621	11	0
480	20:47:59.907	POST Client Threa...	HTTP Request	9		545	621	9	1
481	20:47:59.908	POST Client Threa...	HTTP Request	10		545	621	9	2
482	20:47:59.911	POST Client Threa...	HTTP Request	8		545	621	7	1
483	20:47:59.911	POST Client Threa...	HTTP Request	8		545	621	8	1
484	20:47:59.915	POST Client Threa...	HTTP Request	8		545	621	8	1
485	20:47:59.917	POST Client Threa...	HTTP Request	7		545	621	7	1
486	20:47:59.919	POST Client Threa...	HTTP Request	7		545	621	7	1
487	20:47:59.922	POST Client Threa...	HTTP Request	5		545	621	5	0
488	20:47:59.922	POST Client Threa...	HTTP Request	5		545	621	5	0
489	20:47:59.924	POST Client Threa...	HTTP Request	6		545	621	6	1
490	20:47:59.926	POST Client Threa...	HTTP Request	7		545	621	7	1
491	20:47:59.931	POST Client Threa...	HTTP Request	7		545	621	7	1
492	20:47:59.931	POST Client Threa...	HTTP Request	7		545	621	7	1
493	20:47:59.931	POST Client Threa...	HTTP Request	7		545	621	7	1
494	20:47:59.933	POST Client Threa...	HTTP Request	8		545	621	8	1
495	20:47:59.936	POST Client Threa...	HTTP Request	7		545	621	7	2
496	20:47:59.939	POST Client Threa...	HTTP Request	7		545	621	7	1
497	20:47:59.941	POST Client Threa...	HTTP Request	7		545	621	7	1
498	20:47:59.945	POST Client Threa...	HTTP Request	7		545	621	7	2
499	20:47:59.947	POST Client Threa...	HTTP Request	7		545	621	7	1
500	20:47:59.949	POST Client Threa...	HTTP Request	6		545	621	6	1

Рис.*42 Детальна інформація створення 500 клієнтів на основі монолітної архітектури*

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	500	85	8	423	91.78	20.00%	272.6/sec	123.32	168.00	463.2
TOTAL	500	85	8	423	91.78	20.00%	272.6/sec	123.32	168.00	463.2

Рис.*43 Створення 500 клієнтів на основі мікросервісної архітектури*

Sample #	Start Time	Thread Name	Label	Sample Time(ms)	Status	Bytes	Sent Bytes	Latency	Connect Time(ms)
476	20:15:00.742	POST Client Thread ...	HTTP Request	21		276	631	21	2
477	20:15:00.726	POST Client Thread ...	HTTP Request	47		276	631	47	1
478	20:15:00.748	POST Client Thread ...	HTTP Request	27		276	631	27	1
479	20:15:00.756	POST Client Thread ...	HTTP Request	24		276	631	24	1
480	20:15:00.722	POST Client Thread ...	HTTP Request	59		510	631	59	2
481	20:15:00.748	POST Client Thread ...	HTTP Request	34		510	631	33	1
482	20:15:00.746	POST Client Thread ...	HTTP Request	38		510	631	37	2
483	20:15:00.743	POST Client Thread ...	HTTP Request	47		510	631	47	1
484	20:15:00.755	POST Client Thread ...	HTTP Request	36		510	631	35	2
485	20:15:00.726	POST Client Thread ...	HTTP Request	66		510	631	66	1
486	20:15:00.765	POST Client Thread ...	HTTP Request	30		510	631	30	2
487	20:15:00.753	POST Client Thread ...	HTTP Request	42		510	631	42	2
488	20:15:00.771	POST Client Thread ...	HTTP Request	28		510	631	28	2
489	20:15:00.778	POST Client Thread ...	HTTP Request	22		276	631	22	1
490	20:15:00.770	POST Client Thread ...	HTTP Request	30		510	631	30	2
491	20:15:00.758	POST Client Thread ...	HTTP Request	42		510	631	42	1
492	20:15:00.729	POST Client Thread ...	HTTP Request	72		510	631	72	1
493	20:15:00.762	POST Client Thread ...	HTTP Request	42		510	631	42	1
494	20:15:00.760	POST Client Thread ...	HTTP Request	44		510	631	44	1
495	20:15:00.782	POST Client Thread ...	HTTP Request	22		510	631	22	1
496	20:15:00.778	POST Client Thread ...	HTTP Request	29		276	631	29	1
497	20:15:00.781	POST Client Thread ...	HTTP Request	26		276	631	26	2
498	20:15:00.787	POST Client Thread ...	HTTP Request	22		510	631	22	1
499	20:15:00.786	POST Client Thread ...	HTTP Request	23		276	631	23	2
500	20:15:00.782	POST Client Thread ...	HTTP Request	29		510	631	29	1

Рис.*44 Детальна інформація створення 500 клієнтів на основі мікросервісної архітектури*

На рисунку 45-46 зображено показники, які були отримані при тестуванні запитів отримання даних партнерів. Кількість запитів дорівнювала 1000. Переглядаючи дані показники, можна зробити такий висновок, що монолітна архітектура переважає за мікросервісну по таким показникам:

1. Середнє значення відповіді монолітної архітектури — 1855 мс, мікросервісної — 2724 мс.
2. Мінімальний час відповіді монолітної архітектури — 33 мс, мікросервісної — 218 мс.
3. Максимальний час відповіді монолітної архітектури — 4323 мс, мікросервісної — 5094 мс.

Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request Pa...	1000	1855	33	4323	1119.31	0.00%	220.7/sec	2544.97	424.59	11808.0
TOTAL	1000	1855	33	4323	1119.31	0.00%	220.7/sec	2544.97	424.59	11808.0

Рис.*45 Отримання 1000 даних партнерів на основі монолітної архітектури*

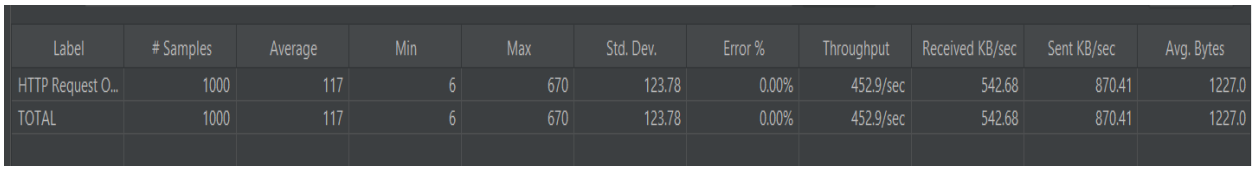
Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request Part...	1000	2724	218	5094	1114.17	0.00%	172.3/sec	2405.76	331.52	14295.7
TOTAL	1000	2724	218	5094	1114.17	0.00%	172.3/sec	2405.76	331.52	14295.7

Рис.*46 Отримання 1000 даних партнерів на основі мікросервісної архітектури*

На рисунку 47-48 зображено показники, які були отримані при тестуванні запитів отримання замовлень. Кількість запитів дорівнювала 1000. Переглядаючи дані показники, можна зробити такий висновок, що монолітна архітектура переважає за мікросервісну по таким показникам:

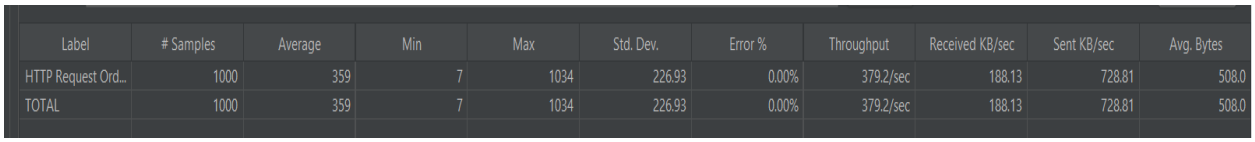
1. Середнє значення відповіді монолітної архітектури — 117 мс, мікросервісної — 359 мс.
2. Мінімальний час відповіді монолітної архітектури — 6 мс, мікросервісної — 7 мс.

3. Максимальний час відповіді монолітної архітектури — 670 мс, мікросервісної — 1034 мс.



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request O...	1000	117	6	670	123.78	0.00%	452.9/sec	542.68	870.41	1227.0
TOTAL	1000	117	6	670	123.78	0.00%	452.9/sec	542.68	870.41	1227.0

Рис.*47 Отримання 1000 замовлень на основі монолітної архітектури*



Label	# Samples	Average	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request Ord...	1000	359	7	1034	226.93	0.00%	379.2/sec	188.13	728.81	508.0
TOTAL	1000	359	7	1034	226.93	0.00%	379.2/sec	188.13	728.81	508.0

Рис.*48 Отримання 1000 замовлень на основі мікросервісної архітектури*

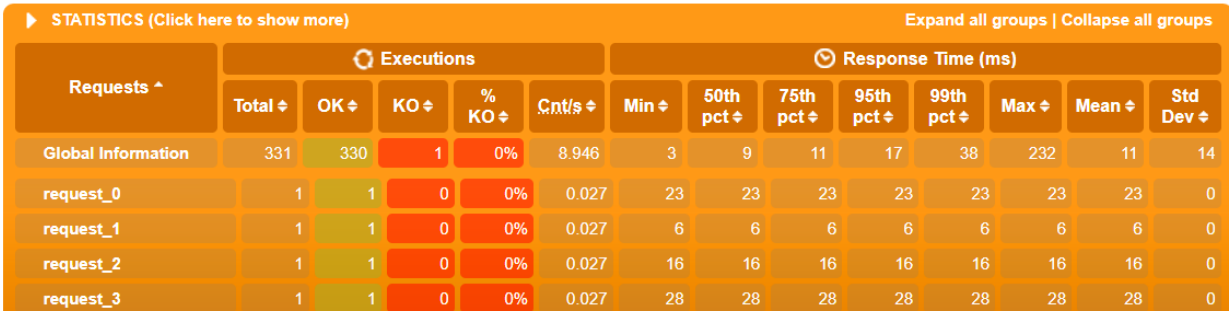
4.2.2 Показники тестування в інструменті Gatling

На рисунку 49-50 зображено показники, які були отримані при тестуванні запитів прийняття замовленнь кур'єрами. На таблиці 2 зображено порівняння отриманих показників.

Таблиця *2*

Показники тесту «Прийняття замовлень кур'єрами»

Показники	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	330	247
Неуспішні запити	1	1
Мінімальний час відповіді	3 мс	5 мс
Максимальний час відповіді	232 мс	136 мс



STATISTICS (Click here to show more) — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	331	330	1	0%	8.946	3	9	11	17	38	232	11	14
request_0	1	1	0	0%	0.027	23	23	23	23	23	23	23	0
request_1	1	1	0	0%	0.027	6	6	6	6	6	6	6	0
request_2	1	1	0	0%	0.027	16	16	16	16	16	16	16	0
request_3	1	1	0	0%	0.027	28	28	28	28	28	28	28	0

Рис.*49 Прийняття замовлень кур'єрами на основі монолітної архітектури*

STATISTICS (Click here to show more) — Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	248	247	1	0%	11.273	5	10	12	16	49	136	12	10
request_0	1	1	0	0%	0.045	136	136	136	136	136	136	136	0
request_1	1	1	0	0%	0.045	10	10	10	10	10	10	10	0
request_2	1	1	0	0%	0.045	50	50	50	50	50	50	50	0

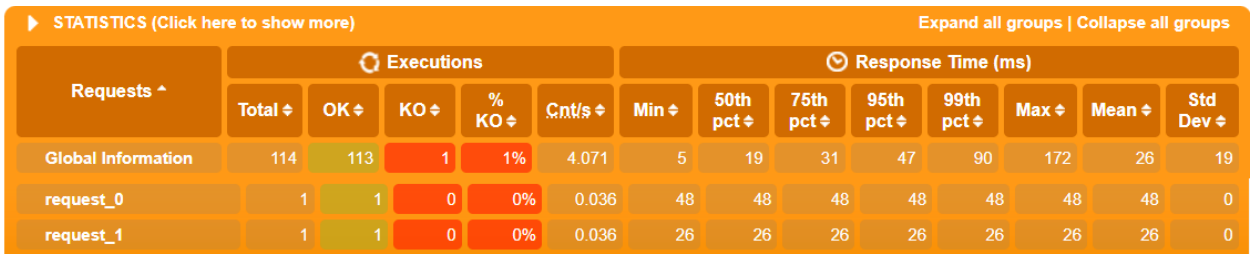
Рис.*50 Прийняття замовлень кур'єрами на основі мікросервісної архітектури*

На рисунку 51-52 зображено показники, які були отримані при тестуванні запитів, які призначені для оформлення замовлення та оплати. На таблиці 3 зображено порівняння отриманих показників.

Таблиця *3*

Показники тесту «Оформлення замовлення та оплата»

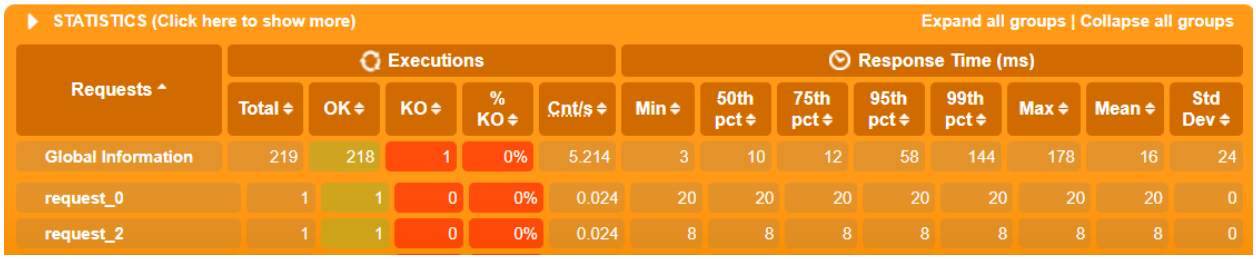
Показники	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	113	218
Неуспішні запити	1	1
Мінімальний час відповіді	5 мс	3 мс
Максимальний час відповіді	172 мс	178 мс



STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	114	113	1	1%	4.071	5	19	31	47	90	172	26	19
request_0	1	1	0	0%	0.036	48	48	48	48	48	48	48	0
request_1	1	1	0	0%	0.036	26	26	26	26	26	26	26	0

Рис.*51 Оформлення замовлення та оплата на основі монолітної архітектури*



STATISTICS (Click here to show more) Expand all groups | Collapse all groups

Requests	Executions					Response Time (ms)							
	Total	OK	KO	% KO	Cnt/s	Min	50th pct	75th pct	95th pct	99th pct	Max	Mean	Std Dev
Global Information	219	218	1	0%	5.214	3	10	12	58	144	178	16	24
request_0	1	1	0	0%	0.024	20	20	20	20	20	20	20	0
request_2	1	1	0	0%	0.024	8	8	8	8	8	8	8	0

Рис.*52 Оформлення замовлення та оплата на основі мікросервісної архітектури*

На рисунку 53-54 зображено показники, які були отримані при тестуванні запитів реєстрації співробітників. На таблиці 4 зображено порівняння отриманих показників.

Таблиця *4*

Показники тесту «Реєстрація співробітників»

Показники	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	22	30
Неуспішні запити	1	1
Мінімальний час відповіді	6 мс	9 мс
Максимальний час відповіді	38 мс	116 мс

Рис.*53 Реєстрація співробітників на основі монолітної архітектури*

Рис.*54 Реєстрація співробітників на основі мікросервісної архітектури*

На рисунку 55-56 зображено показники, які були отримані при тестуванні запитів реєстрації партнерів. На таблиці 5 зображено порівняння отриманих показників.

Таблиця *5*

Показники тесту «Реєстрація партнерів»

Показники	Монолітна архітектура	Мікросервісна архітектура
Успішні запити	31	22
Неуспішні запити	1	1
Мінімальний час відповіді	8 мс	8 мс
Максимальний час відповіді	35 мс	48 мс

Рис.*55 Реєстрація партнерів на основі монолітної архітектури*

Рис.*56 Реєстрація партнерів на основі мікросервісної архітектури*

4.3 Рекомендації, щодо використання мікросервісної та монолітної архітектур

Можна зробити такі висновки після розробки двох застосунків на основі монолітної та мікросервісної архітектури:

1. Для розробки однієї і тієї ж бізнес задачі краще використовувати монолітну архітектуру, так як потребується менше людей для розробки, має просте розгортання, полегшене тестування, тощо.

2. Щоб побудувати застосунок на основі мікросервісної архітектури, треба мати достатній досвід, розуміти патерни, які потрібно використовувати в ній, мати досвід в DevOps методології.

3. Монолітна архітектура працює тільки з однією базою даних та мовою програмування, а мікросервісна — може працювати з декількома базами даних та може бути написана на різних мовах програмування. Ці особливості треба брати до уваги, тоді, якщо проєкт являється «великим» і потрібно залучати все більше нових співробітників, які працюють на різних мовах програмування та базах даних.

4. Якщо буде відбуватися перехід від монолітної до мікросервісної архітектури, то потрібно виділити більше часу на початкову фазу проєкта, а саме виділити час на правильне розташування границь, за якими буде відбуватися розділення на мікросервіси.

5. Якщо відбувається розгортання, то монолітна архітектура буде мати менше проблем, ніж при розгортанні мікросервісів. Тому що вони вимагають більш ретельного аналізу, для того, щоб забезпечити зворотню сумісність та мінімізацію ризиків.

На мою думку, для розробки простого застосунку краще використовувати монолітну архітектуру, так як потребується невеликий досвід в розробці. Якщо застосунок буде великим в обсязі і потрібно використовувати різні технології, то найкращим вибором буде мікросервісна архітектура. В будь-якому випадку існує декілька етапів переходу від монолітної до мікросервісної архітектури.

4.4 Висновки з розділу 4

В даному розділі було проведено аналіз методів, які призначені для розгортання монолітної та мікросервісної архітеткур; проведено дослідження інструментів, які полегшують розгортання; проведено тестування застосунків, тощо. На основі цього, можна сформулювати такі висновки:

1. Проведено дослідження розгортання додатку на основі монолітної архітектури за допомогою інструменту Maven.
2. Проведено дослідження розгортання додатку на основі мікросервісної архітектури за допомогою інструментів Kubernates та Minikube.
3. Проведено дослідження розгортання додатку на основі мікросервісної архітектури за допомогою інструментів Docker та docker-compose.
4. Проведено тестування корпоративних застосунків за допомогою інструменту JMeter.
5. Проведено тестування корпоративних застосунків за допомогою інструменту Gatling.
6. Визначено найкращі показники монолітної та мікросервісної архітектури.
7. Надано рекомендації щодо використання монолітної та мікросервісної архітектури.

ВИСНОВКИ

В ході виконання дипломної роботи було спроєктовано та реалізовано клієнтські застосунки на основі монолітної та мікросервісної архітектури. Тому, можна зробити такі висновки:

1. Досліджена проблема вибору між монолітною та мікросервісною архітектурою.

2. Проведено теоритичні дослідження у галузі монолітної та мікросервісної архітектури.

3. Досліджені засоби, бібліотеки та інструменти для вирішення поставленої проблеми.

4. Реалізовано та виконано тестування монолітної та мікросервісної архітектури на прикладі корпоративних застосунків.

5. Проведено тестування корпоративних застосунків за допомогою інструментів JMeter та Gatling.

6. Проведено дослідження розгортання додатків на основі монолітної та мікросервісної архітектур, за допомогою інструментів Maven, Kubernates, Minikube, Docker.

7. Виконано порівняння характеристик написаних застосунків за якісними та кількісними показниками.

8. Визначено переваги та недоліки розглянутих архітектур.

9. Надано рекомендації щодо використання монолітної та мікросервісної архітектури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Новак В. В., магістрант, Попівщий В. І., доцент, науковий керівник. ГОЛОВНІ РИСИ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ. І всеукраїнська науково-практична конференція здобувачів вищої освіти, аспірантів та молодих вчених «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України» : матеріали І всеукраїнської науко-во-практичної конференції здобувачів вищої освіти, аспірантів та молодих вчених «Актуальні питання сталого науково-технічного та соціально-економічного розвитку регіонів України» 19-21 жовтня 2021 року. Запоріжжя: ЗНУ, 2021. С. 334–336.
2. Новак В. В., магістрант, Попівщий В. І., доцент, науковий керівник. Розробка корпоративного застосунку на основі монолітної та мікросервісної архітектури. Молода наука-2022 : Зб. наук. праць студентів, аспірантів і молодих вчених. Запоріжжя : ЗНУ, 2022. Т. 5. С.180–182.
3. Новак В. В., магістрант, Попівщий В. І., доцент, науковий керівник. Головні риси мікросервісної архітектури. 18-20 жовтня 2022 року ІІ Всеукраїнська науково-практична конференція за участю молодих науковців «АКТУАЛЬНІ ПИТАННЯ СТАЛОГО НАУКОВО-ТЕХНІЧНОГО ТА СОЦІАЛЬНО-ЕКОНОМІЧНОГО РОЗВИТКУ РЕГІОНІВ УКРАЇНИ»: зб. наук. праць студентів, аспірантів і молодих вчених. Запоріжжя : ЗНУ, 2022. С.307-309.
4. Новак В. В., магістрант, Попівщий В. І., доцент, науковий керівник. ПОРІВНЯННЯ МОНОЛІТНОЇ ТА МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ НА ПРИКЛАДІ КОРПОРАТИВНИХ ЗАСТОСУНКІВ. Міжнародної науково-практичної конференції Інженерного навчально-наукового інституту ім. Ю.М. Потебні Запорізького національного університету «Перспективи сталого розвитку в умовах глобалізації в економічному, управлінському та інженерному аспектах» 3 – 4 листопада 2022 р. С.250-252.

5. Монолітна архітектура. URL: https://www.techtarget.com/whatis/definition/monolithic-architecture (дата звернення: 10.04.2022).
6. Martin Fowler — Microservices. URL: http://martinfowler.com/articles/microservices.html (дата звернення: 26.04.2022).
7. CARNELL, John; SÁNCHEZ, Illary Huaylupo. Spring microservices in action. Simon and Schuster, 2021 (дата звернення: 27.04.2022).
8. MEYER, Bertrand. Object-oriented software construction. Englewood Cliffs: Prentice hall, 1997 (дата звернення: 27.04.2022).
9. Pattern: Microservice Architecture. URL: https://microservices.io/patterns/microservices.html (дата звернення: 27.04.2022).
10. Від мікросервісного моноліту до оркестратору. URL: https://blog.byndyu.ru/2020/04/blog-post_14.html?m=1 (дата звернення: 27.04.2022).
11. Glovo. URL: https://glovoapp.com/ua/uk/zaporizhzhya/ (дата звернення: 18.05.2022).
12. Netflix. URL: https://www.netflix.com/ua/ (дата звернення: 25.05.2022).
13. Learn Microservices with Spring Boot. ISBN-13 (pbk): 978-1-4842-6130-9 ISBN-13 (electronic): 978-1-4842-6131-6 https://doi.org/10.1007/978-1-4842-6131-6 Copyright © 2020 by Moisés Macero García (дата звернення: 20.05.2022).
14. CHRISTUDAS, Binildas. Spring cloud. In: Practical Microservices Architectural Patterns. Apress, Berkeley, CA, 2019 (дата звернення: 04.07.2022).
15. API Gateway. URL: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do (дата звернення: 04.07.2022).
16. BAILEY, Misty. Discovery Spring 2011. (дата звернення: 05.07.2022).

17. BOURKE, Tony. Server load balancing. " O'Reilly Media, Inc.", 2001 (дата звернення: 14.07.2022).
18. Circuit Breakers, Discovery, and API Gateways in Microservices. Fabrizio Montesi, Janine Weber. [Submitted on 19 Sep 2016 (v1), last revised 21 Sep 2016 (this version, v2)] (дата звернення: 19.07.2022).
19. Distributed Request Tracing using Zipkin and Spring Boot Sleuth. MALLANNA, S.; DEVIKA, M. Distributed request tracing using zipkin and spring boot sleuth. *Int. J. Comput. Appl*, 2020, 975: 8887. (дата звернення: 20.07.2022).
20. Zipkin. URL: https://zipkin.io/ (дата звернення: 20.07.2022).
21. Rabbit MQ. Режим доступ: https://www.rabbitmq.com/tutorials/tutorial-one-spring-amqp.html (дата звернення: 25.07.2022).
22. KRAMER, Joshua. Advanced message queuing protocol (AMQP). Linux Journal, 2009, 2009.187: 3. (дата звернення: 25.07.2022).
23. GIERKE, Oliver, et al. Spring Data JPA-Reference Documentation. URL: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/ [utoljára megtekintve: 2017. 04. 21.], 2012. (дата звернення: 27.07.2022).
24. BLATZ, Jeremiah. Csrf: Attack and defense. McAfee® Foundstone® Professional Services, White Paper, 2007. (дата звернення: 26.07.2022).
25. SMART, John Ferguson, et al. An introduction to Maven 2. JavaWorld Magazine. Available at: http://www. javaworld. com/javaworld/jw-12-2005/jw-1205-maven. html, 2005, 27. (дата звернення: 01.08.2022).
26. LUKSA, Marko. Kubernetes in action. Simon and Schuster, 2017. (дата звернення: 02.08.2022).
27. Kubernates Concepts. URL: https://kubernetes.io/docs/concepts/overview/components/ (дата звернення: 02.08.2022).
28. Docker. URL: https://docs.docker.com/get-started/overview/ (дата звернення: 03.08.2022).

29. Docker Compose. URL: https://docs.docker.com/compose/. (дата звернення: 08.08.2022).
30. HALILI, Emily H. Apache JMeter. Birmingham: Packt Publishing, 2008. (дата звернення: 04.08.2022).
31. «Soirée de présentation Gatling FrontLine». Gatling Paris User Group (in French). Meetup. Retrieved September 1, 2017. (дата звернення: 04.08.2022).

Декларація

академічної доброчесності

здобувача ступеня вищої освіти ЗНУ

Я, Новак Віктор Вікторович, студент 2 курсу, форми навчання денної, Інженерного навчально-наукового інституту ім. Ю.М. Потебні, спеціальність 121 Інженерія програмного забезпечення, адреса електронної пошти ipz17bd-20@stu.zsea.edu.ua, — підтверджую, що написана мною кваліфікаційна робота на тему «**Порівняння монолітної та мікросервісної архітектур на прикладі корпоративного застосунку**» відповідає вимогам академічної доброчесності та не містить порушень, що визначені у ст.42 Закону України «Про освіту», зі змістом яких ознайомлений.

- заявляю, що надана мною для перевірки електронна версія роботи є ідентичною її друкованій версії;

- згоден на перевірку моєї роботи на відповідність критеріям академічної доброчесності у будь-який спосіб, у тому числі за допомогою інтернет-системи, а також на архівування моєї роботи в базі даних цієї системи.

Дата 30.11.2022 _____ Новак Віктор Вікторович
(студент)

Дата 30.11.2022 _____ Попівщий Василь Іванович
(науковий керівник)