

**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»**

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

«На правах рукопису»
УДК _____

До захисту допущено:

Завідувач кафедри

_____ Сергій СТИРЕНКО

«__» _____ 2023 р.

Дисертація

на здобуття ступеня магістра

**за освітньо-професійною програмою «Інженерія програмного забезпечення
комп'ютерних систем»**

зі спеціальності 121 «Інженерія програмного забезпечення»

**на тему: «Інструменти автоматизації розробки додатків з мікросервісною
архітектурою»**

Виконав:

студент VI курсу, групи ІМ-21мп

Хмара Владислав Віталійович _____

Керівник:

Доцент каф. ОТ, к.т.н.

Павлов Валерій Георгійович _____

Консультант:

Проф. каф. ОТ д.т.н.

Кулаков Юрій Олексійович _____

Рецензент:

Засвідчую, що у цій магістерській
дисертації немає запозичень з праць
інших авторів без відповідних посилань.

Студент _____
(підпис)

Київ – 2023 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) Інформатики та обчислювальної техніки
(повна назва)

Кафедра Обчислювальної техніки
(повна назва)

Рівень вищої освіти – другий (магістерський) за освітньо-професійною програмою

Спеціальність 121. Інженерія програмного забезпечення
(код і назва)

Освітня програма Інженерія програмного забезпечення комп'ютерних систем
(код і назва)

ЗАТВЕРДЖУЮ
 Завідувач кафедри
Сергій СТИРЕНКО
(підпис) (ініціали, прізвище)

« » _____ 2023 р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Хмарі Владиславу Віталійовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації Інструменти автоматизації розробки додатків з мікросервісною архітектурою

Науковий керівник дисертації Доцент каф. ОТ, к.т.н. Павлов ВГ.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від « 7 » листопада 2023 р. № 5168-с

2. Строк подання студентом дисертації _____

3. Об'єкт дослідження процес розробки додатків з мікросервісною архітектурою

4. Предмет дослідження автоматизовані інструменти для розробки додатків з мікросервісною архітектурою

5. Перелік завдань, які потрібно розробити: дослідити процес розробки мікросервісних застосунків; розробити і виконати оцінювання програмного забезпечення для конструювання макетів застосунків з мікросервісною

архітектурою

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормконтроль	д.т.н., проф. Кулаков Ю. О.		
1	к.т.н. доцент Павлов В.Г.		
2	к.т.н. доцент Павлов В.Г.		
3	к.т.н. доцент Павлов В.Г.		
4	к.т.н. доцент Павлов В.Г.		

7. Дата видачі завдання _____

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1	<i>Затвердження теми роботи</i>	<i>01.09.2023</i>	
2	<i>Вивчення та аналіз завдання</i>	<i>05.09.2023</i>	
3	<i>Розробка архітектури та загальної структури програми</i>	<i>23.09.2023</i>	
4	<i>Розробка структур окремих інтерфейсів програми</i>	<i>01.10.2023</i>	
5	<i>Програмна реалізація</i>	<i>25.10.2023</i>	
6	<i>Оформлення пояснювальної записки</i>	<i>25.11.2023</i>	
7	<i>Захист</i>	<i>19.12.2023</i>	

Студент

(підпис)

Хмара В. В.

(ініціали, прізвище)

Науковий керівник дисертації

(підпис)

Павлов В. Г.

(ініціали, прізвище)

РЕФЕРАТ

на магістерську дисертацію

виконану на тему: Інструменти автоматизації розробки додатків з мікросервісною архітектурою

студентом: Хмарою Владиславом Віталійовичем

Робота складається із вступу та чотирьох розділів. Загальний обсяг роботи: 106 аркушів основного тексту, 28 ілюстрацій, 29 таблиць. При підготовці використовувалася література з 39 різних джерел.

Актуальність. Мікросервісна архітектура є популярною у enterprise розробці і у стартапах. Вона допомагає вирішити деякі з проблем традиційної монолітної архітектури такі як: висока зв'язність, складність безперервної доставки, розгортання і масштабування.

Активно впроваджуватись мікросервісна архітектура стала після підвищення доступності контейнеризації і систем оркестрації контейнерів. Наприклад, Kubernetes з'явився у 2014, а Docker – у 2013.

При цьому мікросервісна архітектура потребує значно більш складної інфраструктури. Так, згідно з опитуваннями розробників за 2019-2020 роки, досі однією з важливих проблем є надто велике різноманіття інструментів, які повинні бути освоєні для створення додатка з мікросервісною архітектурою.

Мета і завдання дослідження. Метою магістерської роботи є зменшення часу, необхідного для створення додатків з мікросервісною архітектурою за рахунок застосування розробленого інструменту у вигляді програмного забезпечення, яке автоматизує певні етапи розробки цих додатків.

Для досягнення мети дослідження поставлено і вирішено такі завдання:

- дослідження мікросервісної архітектури і типових паттернів, які з нею використовуються;
- дослідження існуючих інструментів які використовують генерацію коду;

- формування функціональних і нефункціональних вимог до програмного забезпечення, яке розроблюється;
- розробка програмного забезпечення, яке прискорює розробку мікросервісних застосунків за рахунок:
 - генерації конфігураційних файлів для системи оркестрації контейнерів kubernetes для розгортання мікросервісів, баз даних, брокерів подій тощо;
 - генерації конфігураційних файлів для систем безперервної інтеграції і доставки;
 - генерації частини коду мікросервісів.
- проведення експериментів, які доводять переваги використання створеного інструменту.

Об’єкт дослідження – процес розробки додатків з мікросервісною архітектурою.

Предмет дослідження – інструменти для автоматизованої розробки додатків з мікросервісною архітектурою.

Методи досліджень. Для досягнення поставлених в магістерській роботі задач використовуються первісний теоретичний аналіз з наступною практичною реалізацією, результати застосування якої підтверджуються експериментально та обробляються з використанням методів статистичного аналізу.

Наукова новизна одержаних результатів роботи полягає у наступному:

- на основі проведеного аналізу елементів та патернів мікросервісної архітектури, а також сучасних умов її використання, були запропоновані рішення для прискорення розробки мікросервісних застосунків за допомогою автоматизації, ефективність використання яких у порівнянні з аналогами була експериментально доведена;

Особистий внесок здобувача. Магістерське дослідження є самостійно виконаною роботою, в якій відображено отримані особисто теоретичні та прикладні результати, які були перевірені експериментально. Формулювання мети та завдань проводилось спільно з науковим керівником.

Практична цінність. Створене в рамках дослідження програмне забезпечення може використовуватись для зменшення витрат у компаніях, які планують розробляти застосунки з мікросервісною архітектурою.

Ключові слова

Мікросервісна архітектура, інструменти для розробки програмного забезпечення, кодогенерація, Kubernetes, Docker.

Пояснювальна записка до магістерської дисертації

на тему: «Інструменти автоматизації розробки додатків з
мікросервісною архітектурою»

Зміст	
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І	
ТЕРМІНІВ	11
ВСТУП	12
РОЗДІЛ 1 ОГЛЯД ОСНОВНИХ ПРИНЦИПІВ І ПАТТЕРНІВ	
МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ	14
1.1. Загальні положення	14
1.2. Мікросервісна архітектура	16
1.3. Розгортання додатків з мікросервісною архітектурою	19
1.3.1. Підходи до розгортання мікросервісного застосунків	19
1.3.2. Контейнеризація за допомогою Docker	22
1.3.3. Система оркестрації контейнерів Kubernetes	24
1.4. Комунікація між сервісами	26
1.4.1. Комунікація між сервісами через HTTP	27
1.4.2. Обмін повідомленнями між сервісами	30
1.5. Підходи до сховища даних у мікросервісній архітектурі	31
Висновки до розділу 1	34
РОЗДІЛ 2 КОНСТРУЮВАННЯ ІНСТРУМЕНТУ ДЛЯ	
АВТОМАТИЗАЦІЇ РОЗРОБКИ ДОДАТКІВ З МІКРОСЕРВІСНОЮ	
АРХІТЕКТУРОЮ	35
2.1. Розробка вимог до програмного забезпечення	35
2.2. Моделювання бізнес-процесу	37
2.3. Вибір технологій для розробки програмного забезпечення	39
2.4. Архітектура застосунку	42
2.4.1 Архітектура консольного застосунку	42
2.4.2 Архітектура клієнтської частини програми	46
2.4.3 Архітектура серверної частини програми	49

2.5. Розгортання і випуск програмного забезпечення	51
Висновки до розділу 2.....	54

РОЗДІЛ 3 ОЦІНЮВАННЯ ІНСТРУМЕНТУ ДЛЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ.. 55

3.1. Методи оцінки інструментів розробки програмного забезпечення.	55
3.2. Порівняння з існуючими інструментами	58
3.2.1 Helm.....	59
3.2.2 Візуальні генератори конфігураційних файлів	61
3.2.3 JHipster	62
3.2.4 Загальні результати порівняння.....	65
3.3. Проведення експерименту для оцінки ефективності розроблених інструментів.....	67
3.3.1 Вибір статистичного тесту	67
3.3.2 Визначення розміру вибірки за допомогою аналізу статистичної потужності	70
3.3.3 Завдання учасників експерименту.....	73
3.3.4 Набір і навчання учасників експерименту	74
3.3.5 Обробка результатів експерименту	76
Висновки до розділу 3.....	80

РОЗДІЛ 4 РОЗРОБКА СТАРТАП-ПРОЕКТУ 81

4.1. Розробка ідеї стартап-проекту	81
4.2. Технічний аудит проекту	84
4.3. Аналіз ринкових можливостей запуску стартап-проекту	85
4.4. Розроблення ринкової стратегії проекту.....	96
4.4. Розроблення маркетингової програми стартап-проекту	97
Висновки до розділу 4.....	101

ВИСНОВКИ	102
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	103

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

CI/CD – (Continuous Integration/Continuous Delivery) це практика розробки програмного забезпечення, яка включає в себе автоматизований процес інтеграції коду, його тестування та автоматичне розгортання виробленого продукту на виробничий сервер.

API – (Application Programming Interface) це набір правил та протоколів, які дозволяють різним програмам чи компонентам взаємодіяти один з одним.

CLI – (Command Line Interface) це спосіб взаємодії з комп'ютерною системою або програмою шляхом введення текстових команд у командному рядку.

URL – (Uniform Resource Locator) це текстовий рядок, який вказує на місцезнаходження ресурсу в Інтернеті. Він містить інформацію про протокол доступу (наприклад, "http" або "https"), доменне ім'я, порт (за потреби), шлях до ресурсу та інші параметри, які визначають, який ресурс необхідно відобразити чи завантажити в браузері чи іншій програмі.

ВСТУП

Мікросервісна архітектура є зараз однією з найбільш популярних архітектур для enterprise компаній та стартапів. Цю архітектуру використовують такі компанії, як Ebay, Amazon, Netflix, Zalando, SoundCloud, Uber та багато інших.

Вона з'явилася з метою вирішення проблем, які часто виникають у монолітних архітектурах, особливо в сфері enterprise розробки. Набір цих проблем включає високу зв'язність, складність безперервної доставки, складність масштабування та розгортання застосунку.

Широкому впровадженню мікросервісної архітектури передували ряд змін, які зробили її використання більш практичним. Одним із цих змін було зростання доступності технологій контейнеризації та систем оркестрації контейнерів на початку 2010-х. Наприклад, Docker був випущений у 2013 році, Kubernetes - у 2014 році. До цього моменту найбільш популярним був підхід, в якому кожен мікросервіс розгортався на окремій віртуальній машині, що мало низький рівень ефективності.

Іншими змінами, які посприяли збільшенню популярності мікросервісної архітектури були зростання набору інструментів для розробки мікросервісів, підвищення доступності cloud-інфраструктури, розвиток паттернів та накопичення досвіду роботи з архітектурою.

Мікросервісна архітектура надає багато переваг, але їх ціною є значне ускладнення певних аспектів розробки. Сильно збільшується складність DevOps, адже мікросервісний додаток потребує значно більш комплексної інфраструктури. Так, наприклад, в типовому застосунку необхідно налаштовувати і підтримувати систему контейнеризації, систему оркестрації контейнерів, систему CI/CD, базу даних для кожного сервісу, брокер повідомлень тощо. Досі велике розмаїття технологій є однією з важливих проблем розробки застосунків з мікросервісною архітектурою [1]. Окрім того мікросервісні додатки мають сильніші вимоги до якості документації та тестування контрактів. Інструменти автоматизації розробки додатків з

мікросервісною архітектурою можуть певною мірою допомогти справлятися із додатковою складністю, яку вносить мікросервісна архітектура.

Метою магістерської роботи є зменшення часу, необхідного для створення додатків з мікросервісною архітектурою за рахунок застосування розробленого інструменту у вигляді програмного забезпечення, яке автоматизує певні етапи розробки цих додатків.

Для досягнення мети дослідження поставлено і вирішено такі завдання:

- дослідження мікросервісної архітектури і типових паттернів, які з нею використовуються;
- дослідження існуючих інструментів які використовують генерацію коду;
- формування функціональних і нефункціональних вимог до програмного забезпечення, яке розроблюється;
- розробка програмного забезпечення, яке прискорює створення мікросервісних застосунків за рахунок:
 - генерації конфігураційних файлів для системи оркестрації контейнерів kubernetes для розгортання мікросервісів, баз даних, брокерів подій тощо;
 - генерації конфігураційних файлів для систем безперервної інтеграції і доставки;
 - генерації частин коду самих сервісів.
- проведення експериментів, які доводять переваги використання створеного інструменту

РОЗДІЛ 1

ОГЛЯД ОСНОВНИХ ПРИНЦИПІВ І ПАТТЕРНІВ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ

1.1. Загальні положення

Архітектура програмного забезпечення – це опис підсистем і компонентів програмної системи та зв'язків між ними. В архітектурі програмного забезпечення можна виділити такі аспекти[2]:

- структура системи – визначається використовуваним архітектурним стилем: багат шарова, мікросервісна, мікроядерна архітектура тощо;
- архітектурні характеристики – критерії успіху системи (як правило не пов'язані з безпосередньою функціональністю системи): доступність, надійність, масштабованість тощо;
- архітектурні рішення – вони визначають чіткі правила за якими повинна бути сконструйована система;
- принципи проектування – відрізняються від архітектурних рішень тим, що є орієнтирами ніж чіткими правилами

Архітектурні стилі (їх також називають архітектурними шаблонами) під певним ім'ям описують зв'язки компонентів, що охоплюють різноманітні характеристики архітектури. Використання певного конкретно визначеного архітектурного стилю надає такі переваги[3]:

- надає команді чітке бачення проекту та спрощує комунікацію;
- надає набір загальних принципів, який може бути основою для архітектурних рішень;
- гарантує відповідність стандартів і методик, що може спростити підтримку коду і покращити його структурованість.

Існують такі приклади архітектурних стилей:

- Багатошарова архітектура – архітектура, в якій компоненти організовані в логічні горизонтальні шари, і кожен шар виконує конкретну функцію. Є найбільш популярною архітектурою.
- Конвеєрна архітектура – архітектура, в якій головними компонентами є конвеєри і фільтри. Добре працює для задач які включають просту односторонню обробку інформації.
- Мікроядерна архітектура – архітектура, що складається з ядра і плагінів. Плагіни - компоненти, які підключаються до ядра системи і таким чином дозволяють розширювати функціональність системи і адаптувати її під різні варіанти використання.
- Подійно-орієнтована архітектура – розподілена архітектура, що використовує події для запуску та обміну даними між слабо зв'язаними службами. Може виступати як окремою архітектурою, так і компонентом інших архітектур.

Архітектурні стилі можна розділити на монолітні і розподілені. В розподілених системах кілька елементів, що розгортаються, і які пов'язані протоколами віддаленого доступу. В монолітних розгортається лише один елемент.

До монолітних архітектур належать:

- Багатошарова архітектура
- Конвеєрна архітектура
- Мікроядерна архітектура

До розподілених архітектур належать:

- Сервісно-орієнтована архітектура
- Подійно-орієнтована архітектура
- Просторово-орієнтовна архітектура
- Мікросервісна архітектура

1.2. Мікросервісна архітектура

Мікросервісна архітектура – це архітектурний стиль, в якому застосунок є набором сервісів, які:

- Можуть бути розгорнуті незалежно один від одного
- Слабко зв'язані
- Розділені за бізнес-функціональністю
- Можуть розроблятися невеликою командою

Типовий застосунок з мікросервісною архітектурою являє собою багато сервісів, кожен з яких має свою базу даних, може бути відкритим до клієнта через API шлюз та може комунікувати з іншими сервісами синхронно або асинхронно[4] (див рис 1.1.).

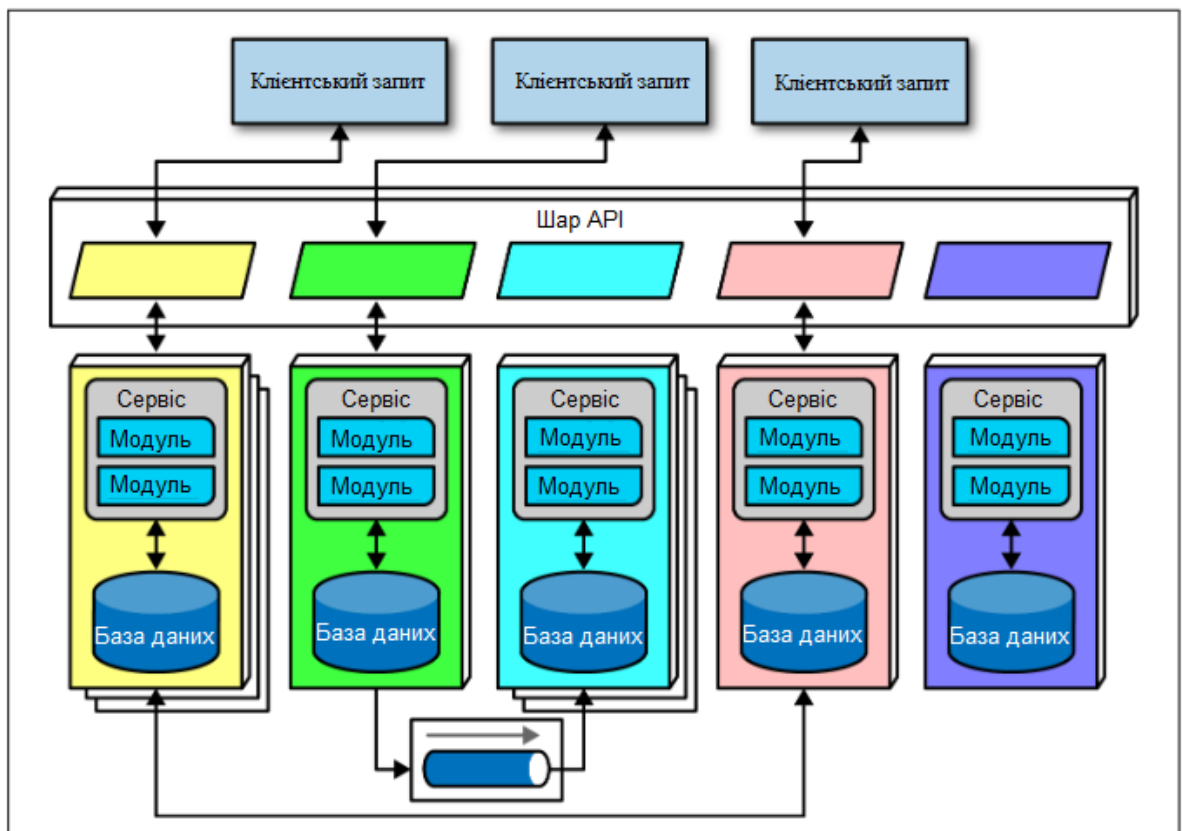


Рис. 1.1. Типова топологія мікросервісного застосунку

На мікросервісну архітектуру сильно вплинув доменно-орієнтований дизайн[1]. Особливо важливим є поняття обмеженого контексту. Під час визначення домену, в нього включаються багато сутностей і поведінок, яким відповідають код і схеми бази даних. В традиційній монолітній архітектурі, ці

сутності і поведінки часто роблять такими, щоб їх можна було використовувати у багатьох місцях, за допомогою спільних схем баз даних та класів, що можуть перевикористовуватись. У обмеженому контексті ж внутрішні частини, такі як код і схеми баз даних зв'язані для функціонування, але вони ніколи не зв'язуються з нічим поза контекстом. Це дозволяє кожному контексту визначати лише те, що йому потрібно, а не враховувати інші складові.

Мікросервісна архітектура надає ряд переваг у порівнянні з традиційними монолітними архітектурами[5]:

- Дозволяє безперервну доставку для великих та комплексних застосунків. Головним фактором у цьому є можливість розгортати сервіси незалежно один від одного. Так, якщо розробникам необхідно розгорнути зміни, які змінюють лише один сервіс, то їм не треба координуватись з іншими розробниками.
- Збільшує тестоздатність.. Той факт, що сервіси є невеликими і незалежними, спрощує написання автоматизованих тестів і пришвидшує їх виконання. Це також спрощує безперервну доставку, адже автоматизоване тестування є ключовою практикою безперервної доставки/розгортання
- Спрощує розробку і підтримку за рахунок невеликого розміру сервісів. Так, розробникам простіше розуміти код, маленька кодова база не сповільнює IDE, кожен сервіс запускається швидше ніж моноліт тощо.
- Покращує масштабованість за рахунок того, що мікросервіси можуть масштабуватись незалежно один від одного.
- Дозволяє розгортати сервіси на спеціалізованому обладнанні в залежності від специфіки сервісу. Так наприклад сервіси, які потребують інтенсивного використання процесора або пам'яті, можуть бути розгорнуті на обладнанні, яке оптимізоване для такого роду задач.
- Покращує ізоляцію несправностей. Більшість помилок виведуть з ладу лише один сервіс, а інші сервіси будуть продовжувати роботу.

- Дозволяє використання різних технологій для різних сервісів. Це дозволяє використовувати спеціалізовані мови програмування або фреймворки для конкретних сервісів. Спрощує тестування та перехід на нові технології.

З іншого боку мікросервісна архітектура має ряд недоліків, обмежень та складностей:

- Розбиття застосунку на мікросервіси є складним і якщо виконано неправильно, призводить до серйозних проблем. Так, при невдалому розбитті застосунку, мікросервіси будуть зв'язними і їх буде необхідно розгортати разом. Така система буде мати недоліки як традиційної монолітної архітектури, так і мікросервісної архітектури. Вибір розміру мікросервісів також призводить до складнощів. Занадто великі мікросервіси сильно збільшують складність з якою доводиться справлятися відповідальним за них командам. З іншого боку дуже малі мікросервіси збільшують витрати на комунікацію між сервісами і зменшують продуктивність застосунку, також це вони посилюють вимоги до інфраструктури. [6]
- Розподіленість системи значно збільшує складність багатьох аспектів розробки. Так, наприклад:
 - сервісам доводиться використовувати системи міжпроцесової комунікації замість простого виклику метода або функції;
 - при розробці потрібно враховувати можливість часткового виходу з ладу або недоступність деяких сервісів;
 - наявність бази даних у кожного сервісу ускладнює підтримку консистентності даних між сервісами, через неможливість використовувати звичайні транзакції;
 - написання автоматизованих тестів для декількох сервісів є складним.
- Висока складність пов'язана з розгортанням і управлінням мікросервісами. Через це використання мікросервісної архітектури потребує використання

спеціалізованих інструментів: системи CI/CD, системи управління контейнерами як Docker Swarm або Kubernetes тощо.

- Потребує високих стартових витрат через необхідність налаштування складної інфраструктури, інструментів CI/CD, засобів оркестрації контейнерів перед початком роботи застосунку.
- Підвищує вимоги до якості і кількості документації, адже комунікація між сервісами вимагає чіткого визначення і підтримки контрактів.

Частина з цих викликів, а саме складність розгортання і управління мікросервісами, високі стартові витрати, підвищені вимоги до якості і кількості документації, потребують високого ступеню автоматизації для вирішення.

1.3. Розгортання додатків з мікросервісною архітектурою

Мікросервісна архітектура дозволяє розгортати сервіси незалежно один від одного, що дозволяє знижує вимоги до комунікації між розробниками під час цього процесу. Також це дозволяє незалежно масштабувати застосунок. З іншого боку такий підхід вносить додаткову складність і часто потребує додаткових інструментів і більш складної інфраструктури. Існують декілька можливих підходів до розгортання мікросервісного застосунку:

- кожен сервіс розгортається як відповідний мові програмування пакет на одній машині
- кожен сервіс розгортається на окремій віртуальній машині в хмарі
- кожен сервіс розгортається як контейнер в хмарі

1.3.1. Підходи до розгортання мікросервісного застосунків

Підхід, в якому декілька сервісів розгортаються як відповідний мові програмування пакет – наприклад, JAR файл для Java, або папка з кодом і модулями для NodeJs, використовується рідко. Цей підхід є досить простим, адже не вимагає використання додаткових інструментів – для розгортання необхідно просто скопіювати сервіс на головну машину і запустити його. Крім того такий підхід є досить ефективним з точки зору використання ресурсів, адже

декілька сервісів використовують одну машину і операційну систему. Проте ряд важливих недоліків не дали цьому підходу набуту поширення:

- Набір технологій не є інкапсульованим. Так, DevOps команда повинна детально розуміти як правильно розгортати кожен сервіс – якою повинна бути версія середовища виконання, які версії пакетів повинні бути встановлені тощо. При цьому різні сервіси можуть використовувати різні мови програмування і фреймворки, а також різні версії одних і тих самих мов програмування і фреймворків. Це значно підвищує ймовірність помилки при розгортанні, а також робить необхідним значно більший рівень комунікації між розробниками і DevOps.
- Неможливість обмежити використання ресурсів сервісом. Так, наприклад, у випадку помилки в одному сервісі, яка веде до сильно збільшеного використання процесора або пам'яті, інші сервіси будуть працювати гірше через те, що вони залежать від тих самих ресурсів.
- Відсутність ізоляції між сервісами. Несправний сервіс може вплинути на інші сервіси, що сильно знижує надійність застосування.

Підхід, в якому кожен сервіс розгортається як образ віртуальної машини, вирішує ці недоліки. Для створення таких образів можуть використовуватись такі інструменти як Aminer або Packer. Aminer підтримує лише AWS, в той час як Packer крім нього підтримує ще Digital Ocean, EC2, Virtual Box, VMware. Так, для кожного сервісу створюється конфігураційний файл, який визначає, який базовий образ віртуальної машини повинен використовуватись, яке програмне забезпечення необхідно встановити, та як повинна бути налаштована система. Таким чином, образ віртуальної машини інкапсулює набір технологій необхідний для сервісу. Образ віртуальної машини може бути розгорнутий без модифікацій. Також при використанні такого підходу сервіси є ізольованими – несправний сервіс не може впливати на роботу інших сервісів або вкрасти у них ресурси процесора або пам'ять. Іншою важливою перевагою використання віртуальних машин є можливість використовувати високорозвинену хмарну

інфраструктуру. Так, наприклад, більшість хмарних провайдерів надають можливість використовувати балансувальники навантаження для трафіку між сервісами та автоматичне масштабування. З іншого боку, такий підхід теж має свої недоліки:

- Високі додаткові витрати ресурсів через необхідність підтримувати окрему віртуальну машину, включаючи її операційну систему, для кожного сервісу. Також є ризик того, що ресурси кожної віртуальної машини будуть використані не до кінця через те, що як правило хмарні платформи пропонують обмежений набір розмірів віртуальних машин.
- Повільне розгортання через те що ініціалізація і запуск віртуальної машини займають відносно багато часу. Також при цьому підході більшу кількість даних необхідно передати через мережу, що також сповільнює розгортання.

Більш сучасним підходом є використання контейнерів для розгортання сервісів. На відміну від розгортання за допомогою образів віртуальних машин, де кожен мікросервіс потребує власної операційної системи (див. рис. 1.2), контейнери дозволяють використовувати спільну операційну систему (див. рис. 1.3) при цьому зберігши ізоляцію процесів і ресурсів. Крім того у контейнері також знаходяться необхідні залежності, що дозволяє надійно розгорнути додаток будь-де без ризиків пов'язаних із сумісністю [7].

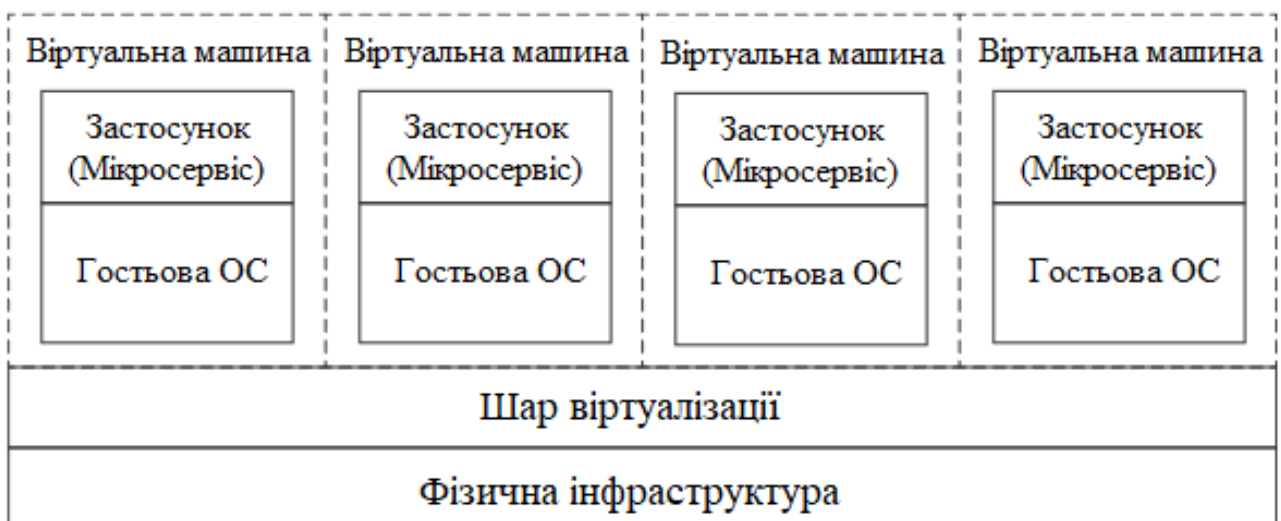


Рис. 1.2. Схема віртуалізації



Рис. 1.3. Схема контейнеризації

Як правило в контейнері запущений один процес, ізольований від інших контейнерів. З точки зору цього процесу – він працює на окремій машині. Так, наприклад, у більшості випадків контейнер має власну кореневу файлову систему та власну IP-адресу, що дозволяє уникнути конфліктів портів. При створенні контейнера можна визначити ресурси процесора і пам'ять, які будуть доступні контейнеру.

1.3.2. Контейнеризація за допомогою Docker

Найбільш популярним інструментом для контейнеризації є Docker [8]. Для розгортання мікросервісу у контейнері необхідно створити образ контейнера. Образ контейнера — це образ файлової системи, що складається з програми та будь-якого програмного забезпечення, необхідного для запуску служби. Часто це повна коренева файлова система Linux, хоча також використовуються більш легкі зображення.

Для створення образу контейнера використовується конфігураційний Docker-файл. В ньому визначається базовий образ, інструкції для встановлення програмного забезпечення і налаштування контейнера та команда яка запускається при створенні контейнера. На рисунку 1.4 зображений Dockerfile

для створення контейнера з простим NodeJs сервером. Ці файли можуть генеруватись за допомогою розробленого інструменту.

```

1  # Вибір базового образу контейнера
2  FROM node:18
3
4  # Вибір робочої директорії для застосунку
5  WORKDIR /home/nodejs-server
6
7  # Встановлення залежностей
8  COPY package*.json ./
9  RUN npm install
10
11 # Копіювання коду програми в директорію застосунку у контейнері
12 COPY . .
13
14 # Відкриття порту 8080
15 EXPOSE 8080
16
17 # Запуск Сервера
18 CMD [ "node", "server.js" ]

```

Рис. 1.4. Dockerfile для створення контейнера з простим NodeJs сервером

Створений образ контейнера повинен бути розміщений у реєстрі образів Docker. Docker hub є публічним реєстром, проте як правило використовуються приватні реєстри такі як AWS EC2 Container Registry або Docker Cloud registry. Після того як сервіс запакований як розміщений у реєстрі образ контейнера, можна створювати контейнери.

Найпростішим способом створення контейнеру є використання команди “docker run”. Цей спосіб має ряд проблем:

- Рушій Docker не придатний для обробки ситуацій де збій відбувся на рівні віртуальної машини, необхідно використовувати інші інструменти.
- Розгортається лише один контейнер, але як правило сервіси залежать від інших сервісів – таких як брокери повідомлень і бази даних, які бажано розгортати разом.

Друга проблема може бути вирішена за допомогою Docker Compose. Це інструмент, який дозволяє декларативно визначити набір контейнерів за допомогою конфігураційного файлу і потім запускати або зупиняти контейнери у групі. Головною проблемою цього інструменту є те, що він працює лише в рамках однієї машини. Для того щоб розгортати контейнери на багатьох машинах використовують інструменти оркестрації контейнерів, такі як Kubernetes і Docker Swarm.

1.3.3. Система оркестрації контейнерів Kubernetes

Kubernetes – це фреймворк оркестрації контейнерів Docker. У ньому машини, на яких запускаються контейнери розглядаються як пул ресурсів процесора, оперативної пам'яті та пам'яті тривалого зберігання. В залежності від доступності ресурсів і ресурсних потреб мікросервісу обирається віртуальна машина для розгортання кожного сервісу. Також існують додаткові налаштування, які дозволяють, наприклад, пріоритезувати розміщення мікросервісів на одній і тій самій машині, або навпаки розміщувати сервіси на різних машинах. Kubernetes забезпечує роботу бажаної кількості здорових екземплярів сервісу. При оновленні робота старих сервісів не завершується, доки нові сервіси не будуть готовими до обробки запитів. Також цей фреймворк має вбудовані можливості по балансуванню навантаження [9].

Основними елементами в Kubernetes є[10]:

- капсули (pods);
- розгортання (deployments);
- сервіси (services);
- ConfigMap.

Капсула є основною одиницею розгортання в Kubernetes. Вона складається з одного або кількох контейнерів, які використовують спільну IP-адресу та сховище. Капсула для екземпляра мікросервісу часто складається з одного контейнера. Але в деяких сценаріях пакет містить один або кілька додаткових контейнерів, які реалізують допоміжні функції. Наприклад, сервер NGINX може

мати допоміжний контейнер, в якому запущена утиліта, яка періодично виконує `git pull` для завантаження останньої версії мікросервісу.

Розгортання — декларативна специфікація капсули. Воно забезпечує постійну роботу бажаної кількості екземплярів капсули (екземплярів мікросервісу). Також воно підтримує керування версіями з постійними оновленнями та відкатами.

Сервіси надають клієнтам статичне/стабільне розташування в мережі. Це форма виявлення сервісів, що надаються інфраструктурою. Служба має IP-адресу та ім'я DNS, яке розв'язує цю IP-адресу та розподіляє навантаження між трафіком TCP і UDP між одним або кількома модулями. IP-адреса та ім'я DNS доступні лише в Kubernetes.

ConfigMap в Kubernetes є іменованими колекціями пар ім'я-значення, які визначають зовнішню конфігурацію для одного або кількох сервісів. Визначення контейнера капсули може посылатися на ConfigMap для визначення змінних середовища контейнера. Також можна використовувати ConfigMap для створення файлів конфігурації всередині контейнера. Крім того можливо зберігати конфіденційну інформацію, наприклад ключі, у формі ConfigMap під назвою Secret.

Ці елементи описуються у `yaml` файлах, які можуть генеруватись за допомогою розробленого інструменту.

1.4. Комунікація між сервісами

Комунікація між сервісами може бути синхронною або асинхронною. При синхронній комунікації сервіс надсилає запит і чекає на відповідь, перш ніж продовжити виконання (див. рис. 1.5). Найбільш типовий спосіб синхронної комунікації – це прямий виклик API через HTTP.

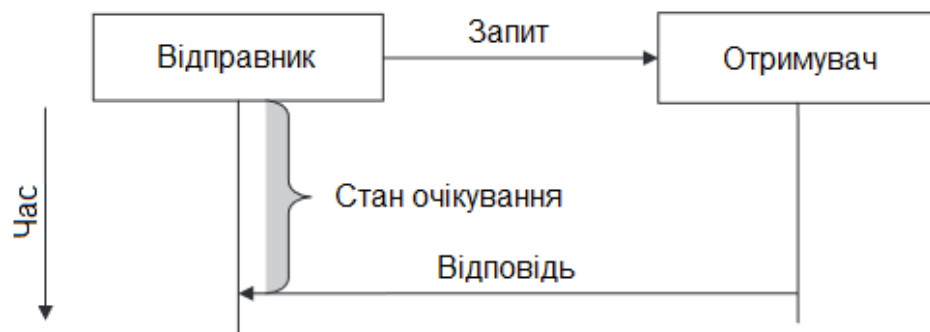


Рис. 1.5. Схема синхронної комунікації між сервісами

У випадку асинхронної комунікації сервіс надсилає запит і не очікує відповіді одразу (див. рис. 1.6). Замість цього, він отримує підтвердження, що запит був отриманий і оброблений, і може продовжити виконання інших завдань. Для асинхронної комунікації можуть використовуватись брокери повідомлень, такі як RabbitMQ або Apache Kafka.



Рис. 1.6. Схема асинхронної комунікації між сервісами

1.4.1. Комунікація між сервісами через HTTP

Використання прямих API-викликів для комунікації між сервісами є більш простим методом: він знайомий більшості розробників, для тестування існує багато простих інструментів (браузер, Postman, curl тощо), він спрощує архітектуру, адже для нього не потрібен проміжний брокер. Проте цей метод також має ряд недоліків.

По-перше, через відсутність проміжного брокера, при синхронній комунікації обидва сервіси які комунікують повинні працювати впродовж всього часу коли готуються і пересилаються запит і відповідь. Це знижує доступність і робить сервіси більш залежними один від одного.

По-друге, необхідність знати URL-адресу для того, щоб зробити HTTP запит, викликає певні труднощі. У типових застосунках, адреси як правило задаються наперед, наприклад, у спеціальних конфігураційних файлах. Але у мікросервісній архітектурі це неможливо зробити, адже кількість сервісів постійно змінюється в результаті автоматичного масштабування, падінь сервісів та оновлень. При цьому звичайно адреси призначаються сервісам автоматично.

Через це використовується динамічний механізм виявлення сервісів. У ньому головним компонентом є реєстр сервісів, який зберігає адреси сервісів та оновлюється кожен раз коли сервіс запускається або зупиняється. Існують два головних підходи до виявлення сервісів[4]:

- сервіси напряму взаємодіють з реєстром сервісів;
- інфраструктура розгортання відповідає за виявлення сервісів.

У першому підході, кожен сервіс сам реєструє свою адресу при запуску, викликаючи API реєстру (див. рис. 1.7). Також реєстр виконує перевірку стану сервісів, для того щоб видалити відповідні записи з реєстру у випадку, якщо сервіс не може обробляти запити через падіння або зупинку через масштабування або оновлення. Є два основних способи виконання перевірки стану сервісів:

- API реєстру сервісів має спеціальний URL на який періодично повинен робити запити кожен зареєстрований сервіс для того щоб вважатись здоровим.
- API кожного сервісу включає спеціальний URL, успішний запит на котрий означає що сервіс здоровий. Реєстр сервісів періодично виконує запити на цей URL кожного зареєстрованого сервісу.

Також у цьому підході виявлення сервісів відбувається на стороні клієнта. Так, кожен сервіс який бажає викликати API іншого сервісу, робить запис до реєстру для отримання списку сервісів і їх адрес. Після цього клієнт обирає з екземплярів бажаного сервісу один за допомогою наперед визначеного алгоритму балансування навантаження і робить запит.

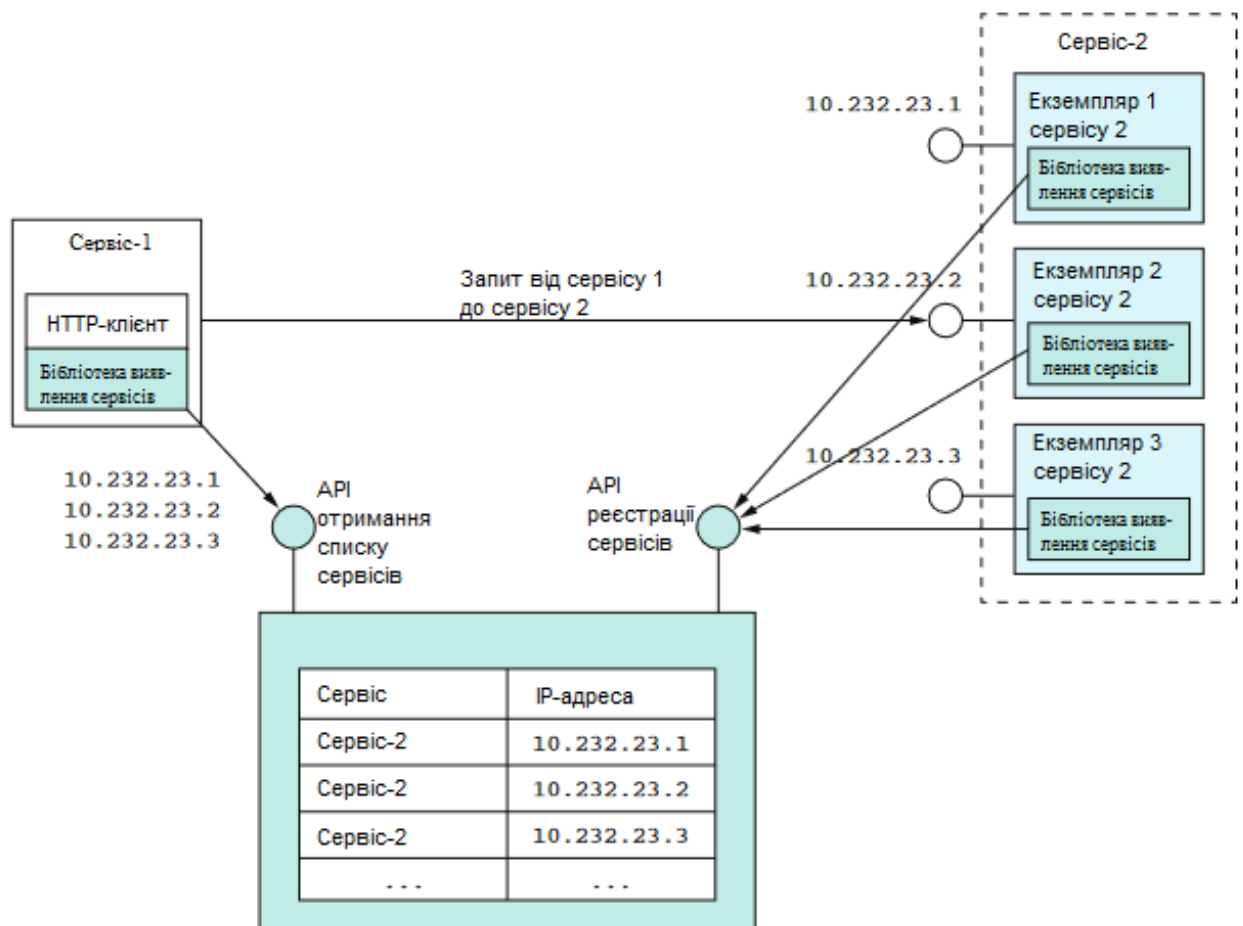


Рис. 1.7. Виявлення сервісів з прямою взаємодією реєстру і сервісів

Як правило, більшість функціональності необхідної для клієнтського виявлення і самореєстрації сервісів вже є реалізованою у бібліотеках, які

доступні для різних мов і фреймворків. Так, наприклад, для реєстру Eureka існують бібліотеки Eureka Java client і Ribbon. Головним недоліком такого підходу є необхідність додаткових витрат на встановлення і керування реєстром сервісів.

Через це часто використовуються засоби виявлення сервісів вбудовані в платформи розгортання, такі як Docker або Kubernetes (див. рис. 1.8). Адреси сервісів при такому підході реєструються спеціальним реєстратором, який включений у платформу розгортання. Кожному сервісу надається DNS-ім'я та віртуальна API-адреса. Коли сервіс викликає API-іншого сервісу він використовує це DNS ім'я. Після цього роутер, включений у платформу розгортання, автоматично направляє запит до одного з екземплярів необхідного сервісу, отримуючи віртуальну IP-адресу з DNS-ім'я.

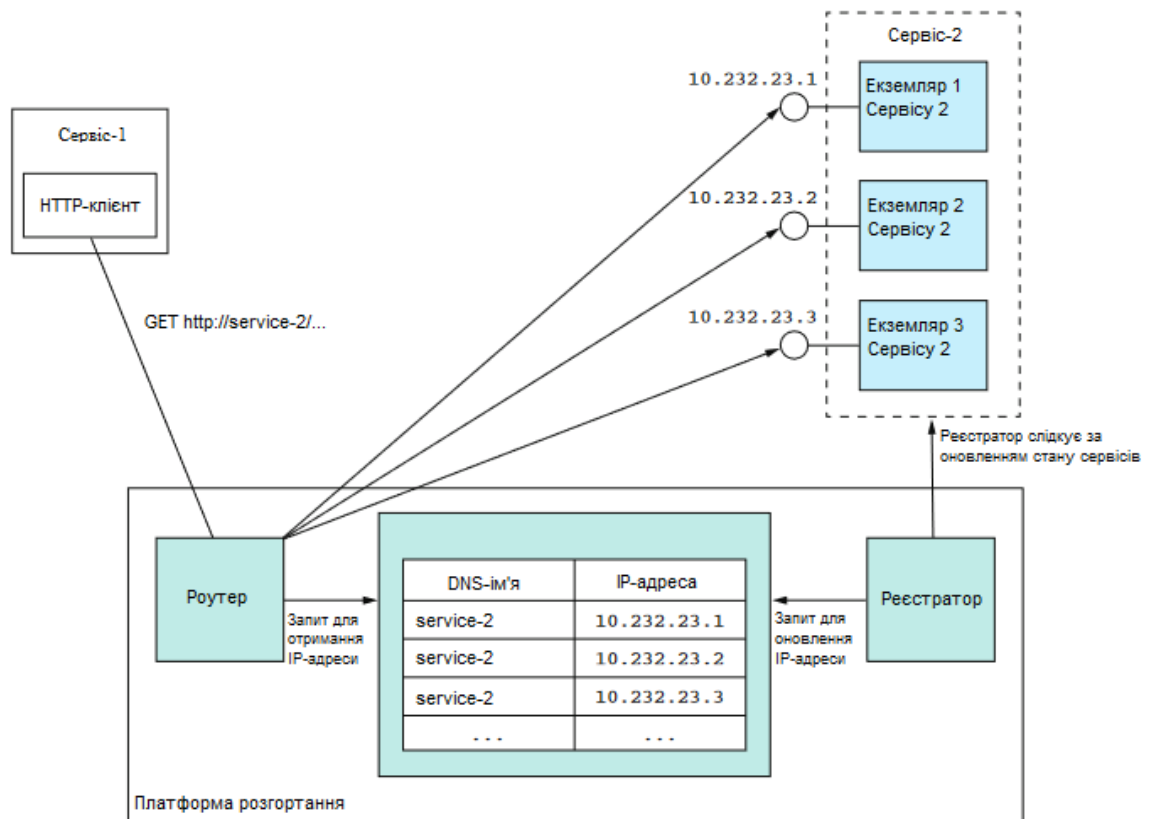


Рис. 1.8. Виявлення сервісів з використанням вбудованих засобів платформи розгортання

Таким чином виявлення сервісів повністю виконується платформою розгортання і в сервісах не буде коду, який буде за це відповідати. Це дозволяє

уникнути проблем пов'язаних з пошуком сумісних бібліотек для різних мов і фреймворків. Проте такий підхід не підходить для ситуацій, де частина сервісів обслуговується іншою платформою розгортання.

1.4.2. Обмін повідомленнями між сервісами

Під час використання обміну повідомленнями сервіси комунікують асинхронно надсилаючи один одному повідомлення. Як правило при цьому використовується брокер повідомлень, який є посередником між сервісами.

Якщо одному сервісу необхідно щось від іншого, то він відправляє йому повідомлення через брокер повідомлень. В якості відповіді інший сервіс теж може відправити повідомлення. При цьому комунікація є асинхронною, сервіс не блокується в очікуванні відповіді. Сервіси повинні бути сконструйовані з припущенням, що відповідь не буде отримана одразу.

Існують такі популярні брокери повідомлень:

- RabbitMQ
- ActiveMQ
- Apache Kafka
- AWS SQS

Як правило брокери повідомлень мають налаштування для оптимізації одних характеристик за рахунок інших. Наприклад, брокер із дуже низькою затримкою може не зберігати порядок, не гарантувати доставку повідомлень і зберігати повідомлення лише в пам'яті. Посередник обміну повідомленнями, який гарантує доставку та надійно зберігає повідомлення на диску, ймовірно, матиме вищу затримку.

Використання брокерів повідомлень має ряд переваг:

- Сервісу, який надсилає повідомлення не потрібно знати про деталі реалізації та існування інших сервісів, що забезпечує низьку зв'язність.
- Немає необхідності використовувати механізм виявлення сервісів, адже відправнику не потрібно знати адресу інших сервісів.

- Брокер повідомлень може зберігати повідомлення у черзі поки вони не зможуть бути оброблені. Це підвищує доступність і відмовостійкість застосунку у порівнянні з синхронною комунікацією, де обидва сервіси повинні успішно працювати впродовж підготовки і відправки запиту і відповіді.
- Мікросервіси можуть використовувати обмін повідомленнями для передачі зміни даних (наприклад, оновлення бази даних) іншим сервісам, зменшуючи пряму залежність від сховищ даних. Це може покращити узгодженість даних і зменшити потребу в синхронному доступі до бази даних.

З іншого боку використання обміну повідомленнями має також і деякі недоліки:

- Брокер повідомлень потенційно може стати обмежувальним фактором у продуктивності застосунку.
- У випадку відмови брокера повідомлень весь застосунок втратить функціональність. Брокер повідомлень повинен бути високодоступним і відмовостійким.
- Брокер повідомлень підвищує складність DevOps, адже він є ще одним компонентом, який потребує встановлення, конфігурації та підтримки.

Перші два фактора рідко є важливими, адже сучасні брокери повідомлень як правило є достатньо масштабованими, відмовостійкими та високодоступними.

Брокери повідомлень можуть бути розгорнуті у Kubernetes. Для цього необхідно створити відповідний yaml файл, який може генеруватись за допомогою розробленого інструменту.

1.5. Підходи до сховища даних у мікросервісній архітектурі

Однією з ключових характеристик мікросервісної архітектури є низька зв'язність. Одним з аспектів забезпечення слабкої зв'язності є підтримка

власного сховища даних для кожного мікросервісу. Існує декілька способів забезпечення приватного сховища даних:

- створення приватної бази даних для кожного сервісу;
- створення приватної схеми у спільній базі даних для кожного сервісу;
- створення приватних таблиць у спільній базі даних для кожного сервісу.

Підходи зі створенням приватної схеми та таблиць потребують менших додаткових витрат. Спосіб же зі створенням приватної бази даних підвищує пропускну здатність для сервісів та робить приналежність сховища даних до сервісу більш явним. Приватні бази даних можуть бути розгорнуті у Kubernetes. Для цього необхідно створити відповідний yaml файл, який може бути згенерований розробленим інструментом.

Забезпечення кожного сервісу приватним сховищем даних надає ряд важливих переваг:

- дозволяє підтримувати низьку зв'язність за рахунок того, що зміни у схемі бази даних одного сервісу не повипливають на інший сервіс;
- дозволяє розробникам уникнути необхідності узгоджувати зміни схеми бази даних з розробниками з команд, що відповідають за інші сервіси;
- дозволяє ізолювати сервіси один від одного на етапі виконання, наприклад, блокування певних записів у базі даних одним сервісом не призведе до помилок та затримок в інших сервісах;
- дозволяє використовувати для кожного сервісу оптимальну технологію для зберігання даних.

З іншого боку такий підхід має і свої недоліки:

- призводить до складнощів, якщо існує потреба в транзакціях, який повинні діяти в рамках декількох сервісів;

- призводить до складнощів, якщо для певних запитів необхідні дані з кількох баз даних;
- підвищує операційну складність через необхідність встановлювати, налаштовувати і підтримувати декілька баз даних, які до того ж можуть використовувати різні технології.

Висновки до розділу 1

Мікросервісна архітектура – це архітектурний стиль, в якому застосунок є набором сервісів, які:

- Можуть бути розгорнуті незалежно один від одного
- Слабко зв'язані
- Розділені за бізнес-функціональністю
- Можуть розроблятися невеликою командою

Розгортання мікросервісів як правило відбувається за допомогою контейнерів Docker. При цьому розгортанням керує система оркестрації контейнерів Docker.

Комунікація між сервісами частіше за все відбувається за допомогою обміну повідомленнями через брокер повідомлень. Головною перевагою такого підходу є збереження низької зв'язності, що є одним з головних принципів мікросервісної архітектури.

Дані у мікросервісній архітектурі прийнято зберігати у приватних базах даних. Головною причиною цього також є збереження низької зв'язності між сервісами.

Розробленим інструментом автоматизації розробки додатків з мікросервісною архітектурою можуть генеруватись:

- Dockerfile для сервісів;
- файли конфігурації для CI/CD.
- файли конфігурації kubernetes для розгортання мікросервісів;
- файли конфігурації kubernetes для встановлення і налаштування API-шлюзів.
- файли конфігурації kubernetes для встановлення і налаштування баз даних;
- файли конфігурації kubernetes для встановлення і налаштування брокерів повідомлень.

РОЗДІЛ 2

КОНСТРУЮВАННЯ ІНСТРУМЕНТУ ДЛЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

2.1. Розробка вимог до програмного забезпечення

Було розроблено такі функціональні вимоги до застосунку:

- Інструмент повинен надавати користувачу можливість автентифікуватись у системі за допомогою електронної пошти і пароля.
- Інструмент повинен надавати користувачу можливість зареєструватись у системі за допомогою електронної пошти і пароля.
- Інструмент повинен надавати користувачу можливість створити або видалити проєкт та змінити його ім'я.
- Інструмент повинен надавати користувачу можливість завантажувати і зберігати проєкти.
- Інструмент повинен надавати користувачу можливість відредагувати загальні налаштування проєкту.
- Інструмент повинен надавати користувачу можливість додавати, налаштовувати та видаляти мікросервіси у проєкті.
- Інструмент повинен надавати користувачу можливість додавати, налаштовувати та видаляти шлюзи API у проєкті.
- Інструмент повинен надавати користувачу можливість додавати, налаштовувати та видаляти бази даних у проєкті.
- Інструмент повинен надавати користувачу можливість додавати, налаштовувати та видаляти брокери повідомлень у проєкті.
- Інструмент повинен надавати користувачу можливість переміщати створені сутності (бази даних, мікросервіси тощо) по робочій області
- Інструмент повинен надавати користувачу можливість пов'язувати між собою створені сутності.

- Інструмент повинен надавати можливість користувачу встановлювати модулі з додатковими компонентами і налаштуваннями
- Інструмент повинен користувачу надавати можливість завантажувати конфігураційний файл для генерації проекту.
- Інструмент повинен надавати можливість користувачу згенерувати конфігураційні файли для розгортання та налаштування інфраструктури і частину коду мікросервісів для мікросервісного застосунку.

Були сформульовані такі нефункціональні вимоги до застосунку:

- Інструмент повинен бути сумісним з популярними браузерами;
- Інструмент повинен бути забезпечений інформаційним захистом.
- Інструмент повинен мати зручний та легкий у використанні інтерфейс;
- Інструмент повинен мати високу продуктивність роботи;

Також було розроблено діаграму варіантів використання (рис. 2.1).

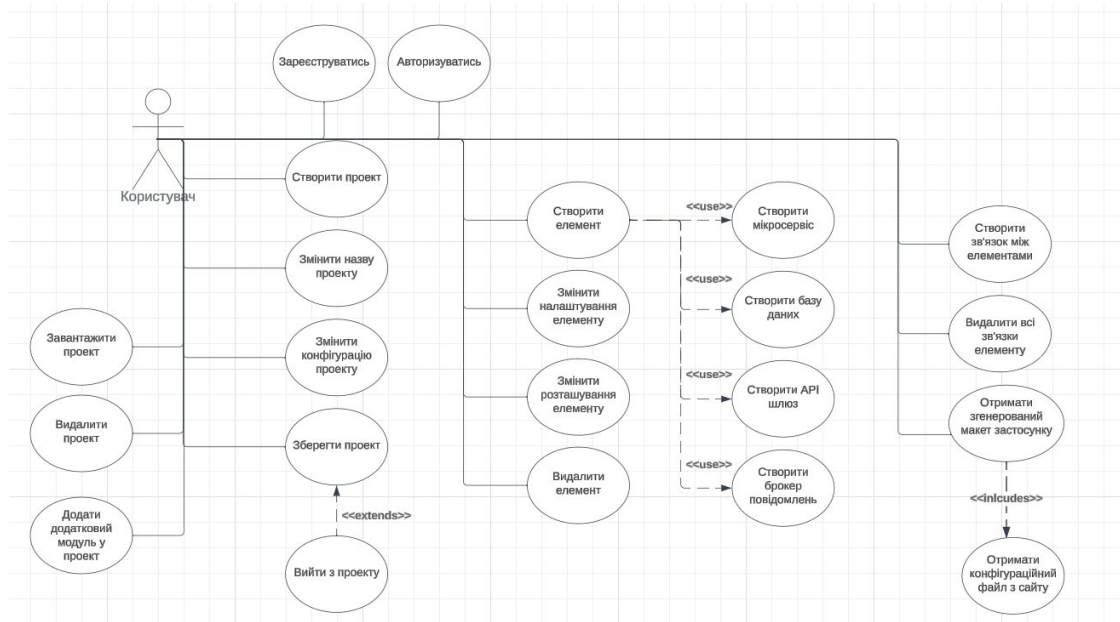


Рис. 2.1. Діаграма варіантів використання

2.2. Моделювання бізнес-процесу

Бізнес-процеси генерації користувачем частин мікросервісного застосунку і автентифікації та реєстрації користувача описані за допомогою BPMN діаграм (див. рис. 2.2 і 2.3). Найбільш типова послідовність дій користувача виглядає так:

- користувач реєструється, якщо він ще не зареєстрований;
- користувач виконує автентифікацію у веб-додатку;
- користувач створює проєкт у додатку;
- користувач додає бажані сутності мікросервісної архітектури;
- користувач налаштовує створені сутності і створює зв'язки між ними;
- користувач завантажує у веб-застосунку конфігураційний файл з даними проєкту;
- за допомогою консольного застосунку, вказавши шлях до завантаженого файлу і шлях за яким буде збережений проєкт, користувач генерує інфраструктуру і частину коду мікросервісного застосунку.

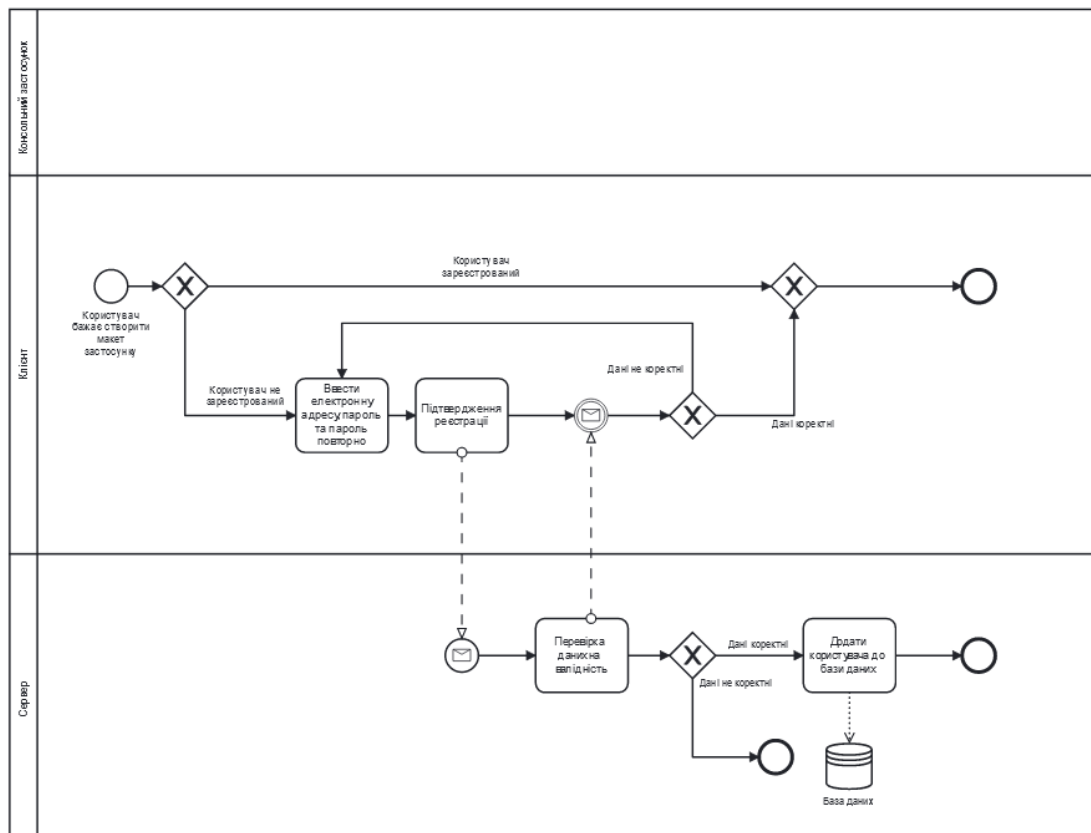


Рис. 2.2. BPMN діаграма процесів автентифікації та реєстрації користувача

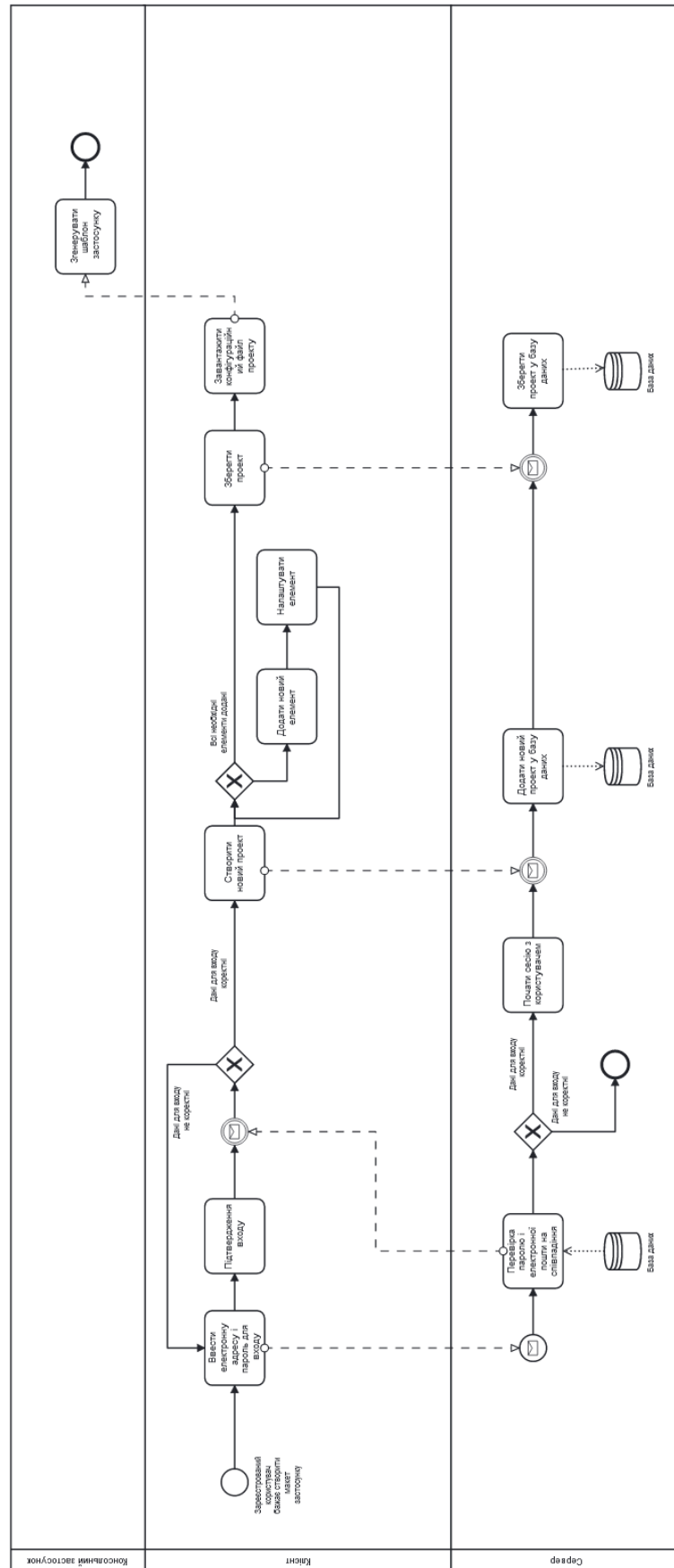


Рис. 2.3. BPMN діаграма процесу генерації користувачем частин мікросервісного застосунку

2.3. Вибір технологій для розробки програмного забезпечення

В якості клієнту планується створити односторінковий web-застосунок. При такому підході спочатку користувачу надсилається початковий HTML документ, а після цього нові дані передаються користувачу малими частинами при взаємодії з ним. При цьому сторінка не завантажується наново. Для розробки односторінкових web-застосунків найпопулярнішими інструментами є фреймворки Vue і Angular та бібліотека React. Всі ці інструменти мають схожий функціонал, але також мають певні відмінності(див. табл. 1.3).

Таблиця 2.1

Порівняння Angular[11], Vue[12] і React[13]

Характеристика	Angular	Vue	React
Стандартна архітектура	В якості стандартного способу структурування коду використовується шаблон Model-View-Controller(MVC). Застосунок складається з модулів, компонентів та сервісів, які пов'язуються за допомогою ін'єкції залежностей.	В якості стандартного способу структурування коду використовується шаблон Model-View-Controller(MVC). Крім цього не накладає особливих обмежень на структуру застосунку.	Не має стандартної архітектури

Продовження таблиці 2.1

Доступність документації і доступність підтримки спільноти	Має детальну і зрозумілу документацію, значну кількість якісних навчальних матеріалів. Має високу популярність та чисельну спільноту, що спрощує доступ до підтримки. Значна кількість бібліотек є сумісною з Angular.	Має детальну і зрозумілу документацію, значну кількість якісних навчальних матеріалів. Спільнота значною мірою проживає в Китаї і не володіє англійською, що дещо зменшує доступність підтримки.	Має детальну і зрозумілу документацію та значну кількість якісних навчальних матеріалів. Має високу популярність та чисельну спільноту, що спрощує доступ до підтримки. Значна кількість б і
Підтримувані мови програмування	Потребує використання мови отримання всіх переваг від використання фреймворку, хоча звичайний Javascript також підтримується	Підтримують і Javascript, і Typescript	
Складність освоєння	Висока	Низька	Низька

Головною різницею між наведеними інструментами є стандартна структура коду. Якісну документацію і користувацьку підтримку мають усі три інструменти. Також усі інструменти підтримують Typescript, який планується використовувати. Різниця ж у складності освоєння інструменту не є важливою у даному контексті.

Таким чином був обраний фреймворк Angular, так як він надає надійну стандартну структуру застосунку, що значно спростить його розробку і підтримку.

Серверних фреймворків існує багато для різних мов програмування:

- ASP .NET для C#;
- Spring Framework для Java;
- Django для Python;
- Ruby on Rails для Ruby;
- Express, Koa, Nest тощо для NodeJs.

Серверна частина застосунку є досить простою, тому було вирішено використовувати платформу NodeJs, що дозволяє розробляти і серверну, і клієнтську частину за допомогою мови Typescript.

Express, Koa і Nest є найбільш популярними серверними фреймворками для платформи NodeJs. При цьому Koa є еволюцією фреймворку Express і розроблений тими ж самими розробниками, тому вони є дуже схожими. Було обрано фреймворк Nest, який сильно від них відрізняється і має ряд переваг [14]:

- стандартна структура додатку у Nest є чітко визначеною, використовує ін'єкцію залежностей для пов'язування елементів і є дуже схожою на структуру додатку у Angular;
- фреймворк надає багато переваг при використанні Typescript;
- інструментарій фреймворку включає CLI, який надає можливість генерувати фрагменти коду додатку, що пришвидшує розробку.

Для розробки інтерфейсу командного рядка також було обрано NodeJs, адже будь-яка мова загального використання підходить для цього, а вибір NodeJs дозволить використовувати одну мову для всього проекту.

2.4. Архітектура застосунку

Застосунок складається з таких частин:

- клієнтська частина, яка надає користувачу графічний інтерфейс для налаштування бажаного застосунку;
- серверна частина, яка обслуговує авторизацію, збереження і завантаження проектів на клієнті;
- консольний застосунок, який безпосередньо відповідає за генерацію інфраструктури і частини коду застосунку.

2.4.1 Архітектура консольного застосунку

Так як консольний застосунок повинен підтримувати використання додаткових модулів для генерації інших типів елементів програм, то було вирішено використовувати мікроядерну архітектуру.

Сам консольний застосунок лише завантажує необхідні модулі з серверу та викликає їх як дочірні процеси, а також координує роботу між ними. Генерація усіх базових елементів (мікросервіси, бази даних, API-шлюзи, брокери повідомлень) відбувається за допомогою модуля **core**.

Для полегшення написання додаткових модулів було створено бібліотеку **microservice-generator-module**. Ця бібліотека бере на себе відповідальність за комунікацію та підтримку сумісності з основним застосунком, генерацію опису модуля, створення виконуваного файлу тощо. Також вона надає набір утиліт для спрощення створення модулів. Загальну схему взаємодії елементів застосунку можна побачити на рисунку 2.4.

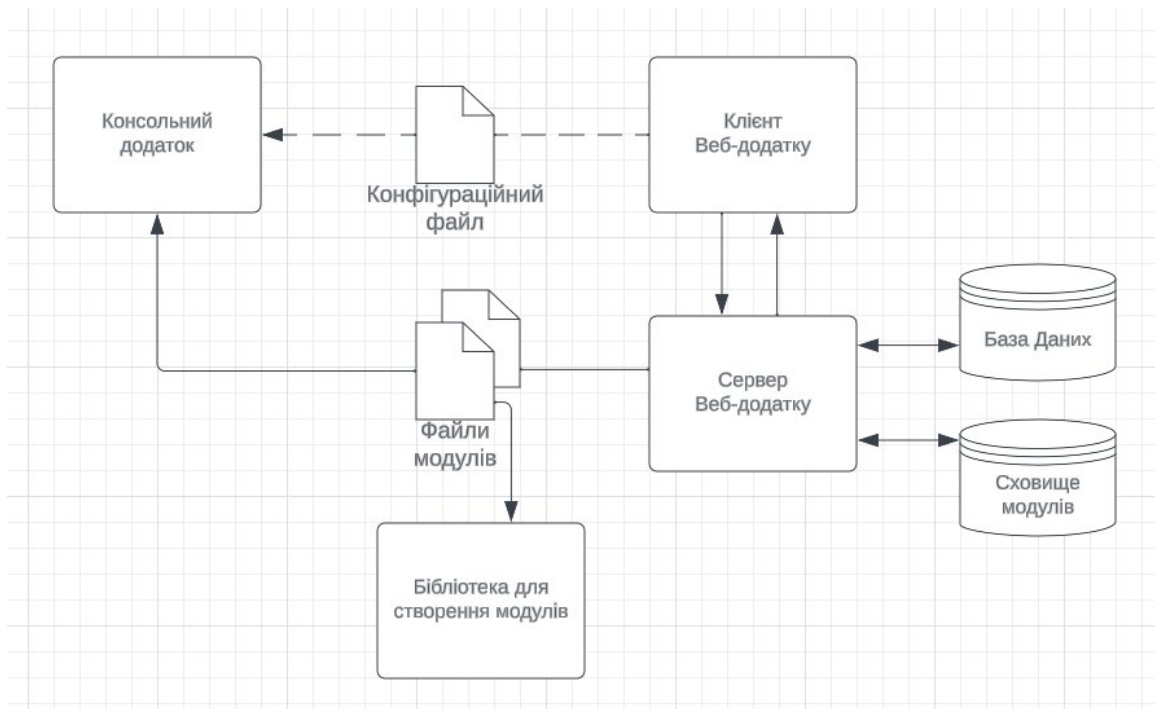


Рис. 2.4. Взаємодія елементів застосунку

Для генерації конфігураційних файлів та коду проекту, консольний застосунок використовує конфігураційний файл, що створюється за допомогою веб-застосунку. Структура цього конфігураційного файлу залежить від встановлених модулів. Структура файлу при використанні лише модуля **core** зображена на рисунку 2.5.

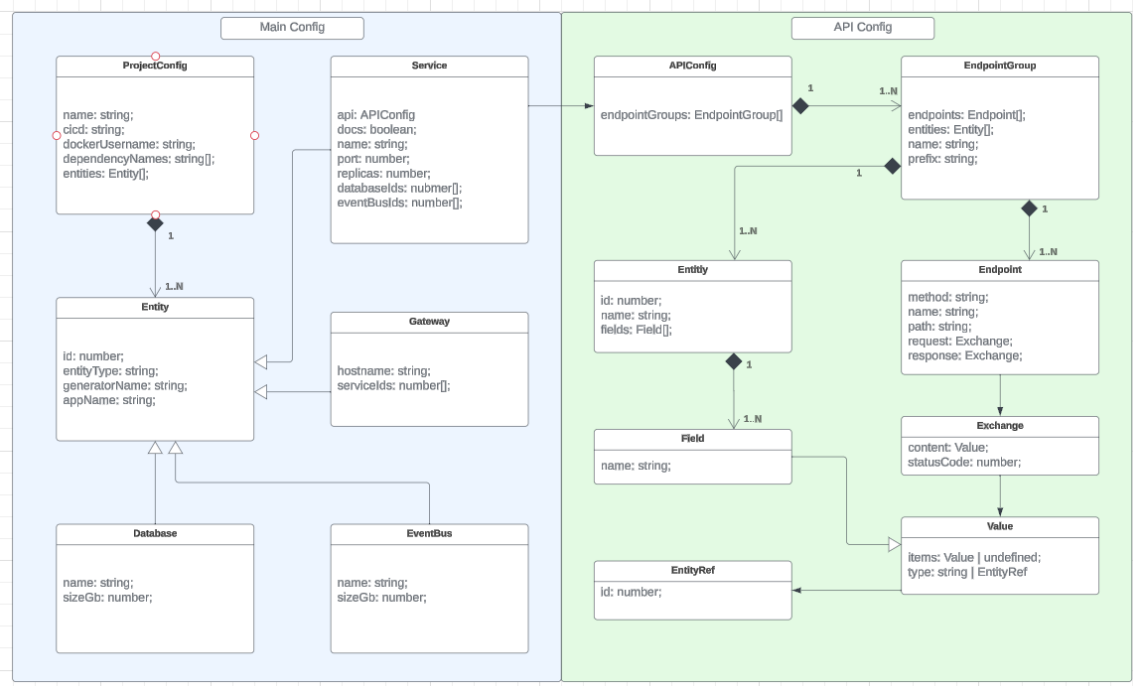


Рис. 2.5. Структура конфігурації

Обробка конфігураційного файлу починається з того, що консольний застосунок перевіряє наявність усіх необхідних залежностей проекту. Відсутні додаткові модулі завантажуються з сервера. Після цього консольний застосунок, надсилає запити на додаткові дані, які будуть використовуватись для генерації, для усіх додаткових модулів. Потім модулі генерують відповідні файли.

Діаграму класів для модуля можна побачити на рисунку 2.6.

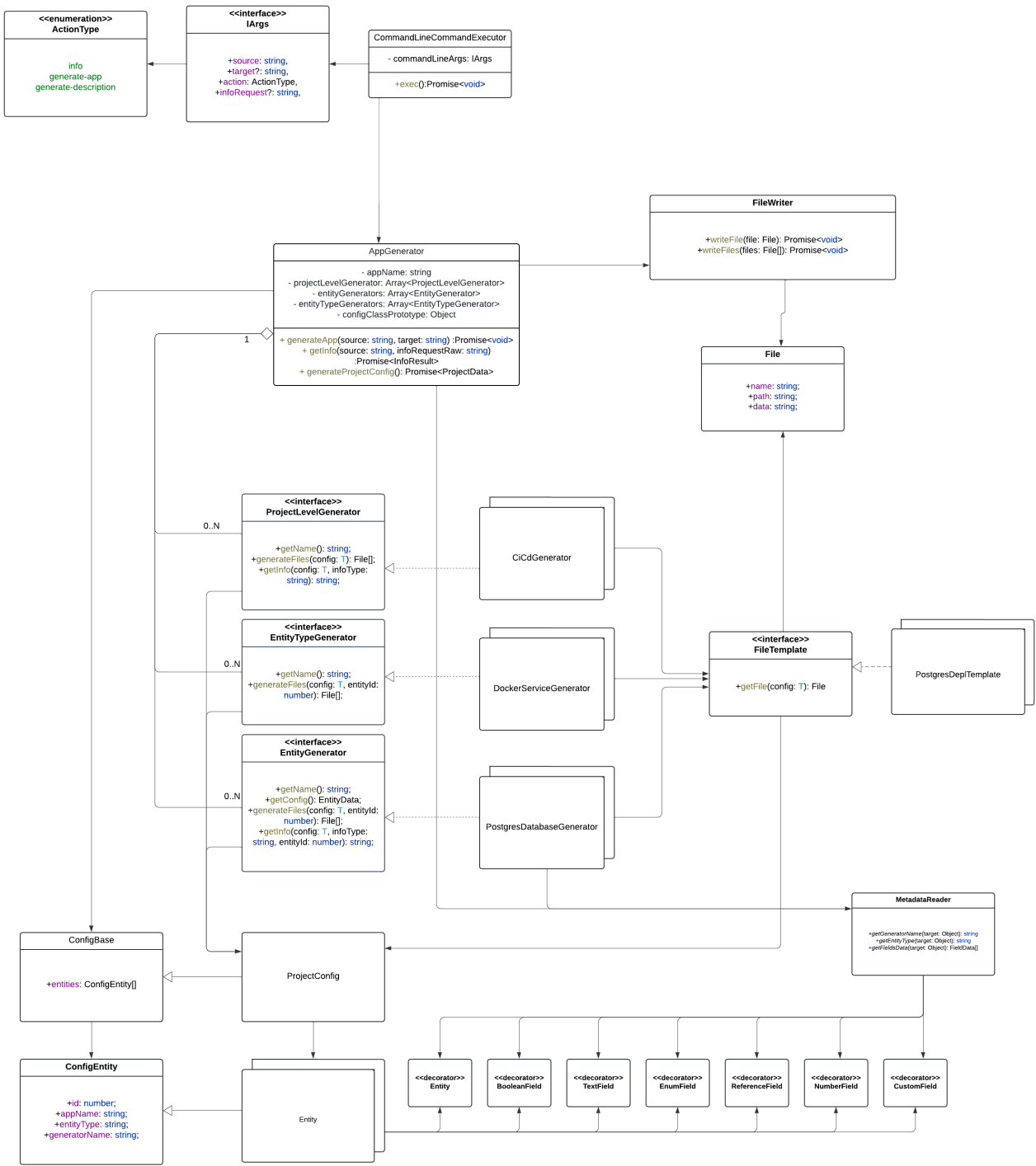


Рис. 2.6. Діаграма класів для модуля генерації

Головним класом у модулі є **AppGenerator**, він надається створеною бібліотекою. В конструкторі визначаються генератори різного рівня та назва модуля.

На етапі створення модуля, **AppGenerator** використовується для генерації JSON-файлу з метаданими модуля. Там вказані залежності модуля, його назва, додаткові поля для глобальних налаштувань проекту та додаткові сутності, які додає модуль з описом їх полів. Ці дані використовуються для встановлення необхідних модулів та для того, щоб визначити можливі опції при редагуванні проекту у веб-застосунку.

Дані про поля отримуються з метаданих класів, які репрезентують генеровані сутності. Ці метадані записуються за допомогою відповідних декораторів. Так, наприклад, параметр **hostname** API-шлюза визначається як текстове поле, за допомогою декоратора **@StringField**. За отримання метаданих сутностей відповідальний клас **MetadataProvider**.

Існує три рівня генераторів:

- **ProjectLevelGenerator** – застосовується для генерації файлів на основі обробки параметрів рівня проекту, таким наприклад є генератор, який генерує конфігураційні файли CI/CD.
- **EntityDecorator** – застосовується для генерації сутностей – баз даних, мікросервісів, API-шлюзів, брокерів повідомлень. Обмежений певним типом, так наприклад для баз даних Postgres буде окремий генератор.
- **EntityTypeGenerator** – також генерує сутності, але використовується якщо існує спільна логіка на рівні типу сутності. Так, наприклад, принцип формування файлів Kubernetes для мікросервісів у Docker є однаковим незалежно від мови, якою написані сервіси.

Генератори повертають набір об'єктів класу **File**, які потім екземпляр класу **FileWriter** записує у файлову систему.

2.4.2 Архітектура клієнтської частини програми

Так як клієнтська частина створена за допомогою фреймворку Angular, де модулі, сервіси та компоненти є головними складовими програми, наведемо схему з їхньою структурою (див. рис. 2.7), а також опис їх роботи (див. табл. 2.2).



Рис. 2.7. Структура клієнту

Таблиця 2.2

Компоненти клієнтської частини програми

Компонент	Опис
AppComponent	відповідає за клієнтський роутинг, повертає потрібні сторінки в залежності від адреси
HeaderComponent (ProjectsModule)	показує зазву веб-застосунку та кнопку для виходу
LoginComponent	форма авторизації
RegisterComponent	форма реєстрації
WorkspaceComponent	сторінка редагування проекту
ToolboxComponent	список сутностей, які можна використати у застосунку: мікросервіси, бази даних тощо
MenuEntityComponent	сутність, яку можна використати у застосунку, у меню, за допомогою компоненту можна додати сутність на робочу область
HeaderComponent	верхня панель на сторінці редагування проекту, містить назву проекту та кнопки для взаємодії з ним (збереження, вихід, експорт, редагування глобальної конфігурації)
EntityComponent	сутність, яку можна використати у застосунку, на робочій області, такі сутності можна редагувати, пов'язувати між собою та переміщати по робочій області

Продовження таблиці 2.2

EntityInputComponent	поле налаштувань сутності, компонент відповідає за вибір правильного типу поля в залежності від метаданих сутності
ProjectInputComponent	поле налаштувань проекту
EntityEditorComponent	меню редагування налаштувань сутності
ProjectEditorComponent	меню глобальних налаштувань проекту
ApiEditorComponent	форма редагування налаштувань API для мікросервісів
CanvasComponent	робоча область проекту
SidePanelComponent	бокова панель, для показу додаткових налаштувань у разі необхідності
ProjectComponent	сегмент для деталей поточного проекту, відображає назву проекту, дату його створення та останнього оновлення, містить кнопки для його видалення або завантаження
ProjectsComponent	сторінка управління проектами, яка складається з заголовку, списку проектів, сегменту для деталей поточного проекту і панелі інструментів
ProjectListComponent	список створених проектів

2.4.3 Архітектура серверної частини програми

Сервер написаний за допомогою фреймворку NestJs. У ньому програми складаються з модулів, провайдерів та контролерів. На рисунку 2.8. показана структура серверу. Також наведемо опис контролерів (таблиця 2.3).

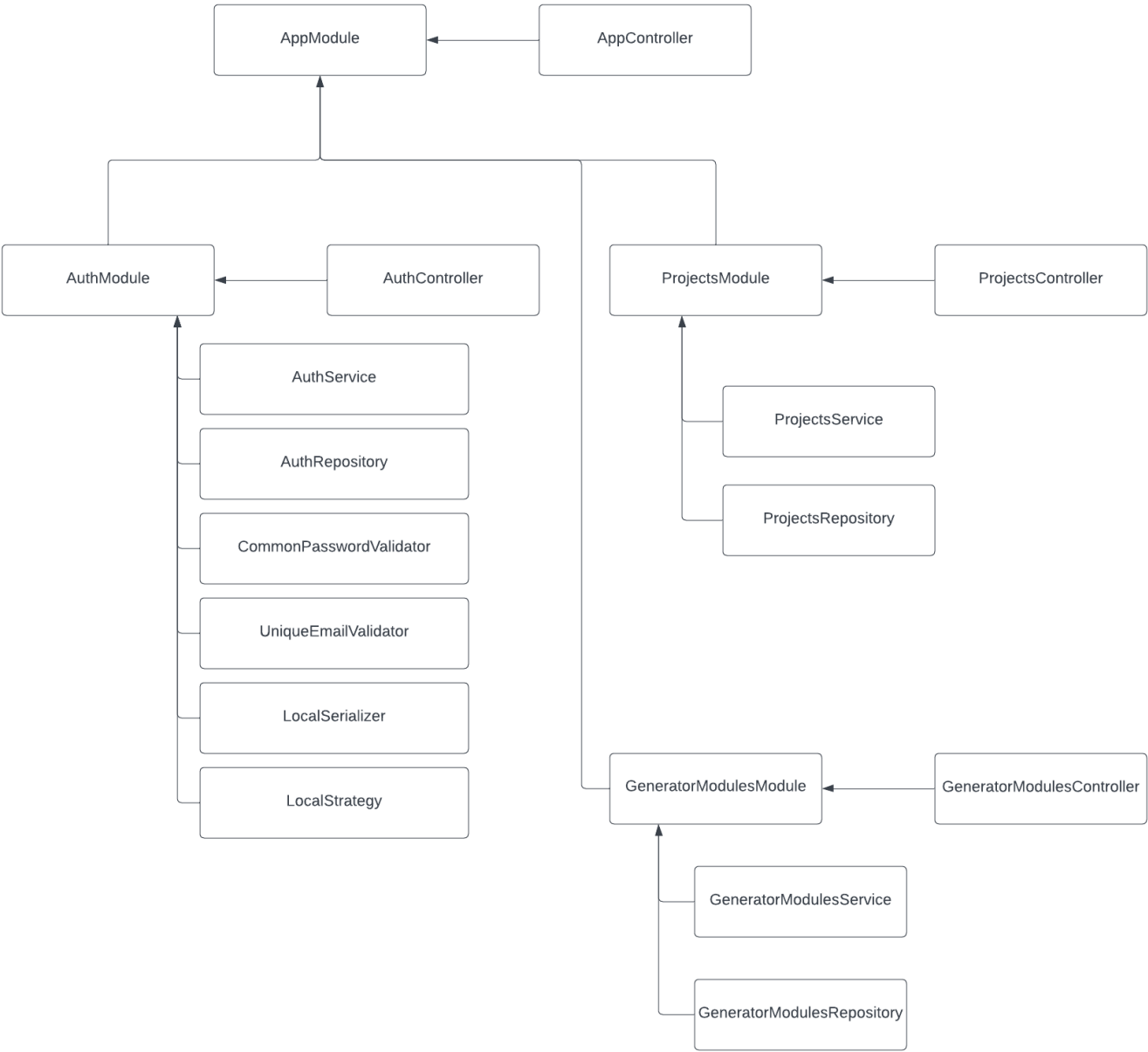


Рис. 2.8. Структура серверу

Таблиця 2.3

Контролери серверу

AppController	
HTTP GET health	перевірка функціонування серверу
ProjectController	
HTTP GET projects	надає список проектів певного користувача
HTTP GET projects/:id	надає всю інформацію про конкретний проект
HTTP POST projects	створює проект
HTTP POST projects/:id/save	зберігає зміну у проекті
HTTP DELETE projects/:id	видаляє проект
AuthController	
HTTP POST auth/signup	використовується для реєстрації користувачів
HTTP POST auth/signin	використовується для автентифікації користувачів
HTTP GET auth/user	надає інформацію про поточного користувача
HTTP POST auth/logout	закінчує сесію з користувачем
GeneratorModulesController	
HTTP GET modules	надає список доступних модулів і інформацію про них
HTTP GET modules/:id	надає метадані модулю
HTTP GET modules/:id/file	завантажує файл модулю

Сервіс має базу даних Postgres, яка використовується для зберігання даних про клієнтів, їхні проекти, а також про існуючі модулі для генерації.. Діаграма бази даних наведена на рисунку 2.9.

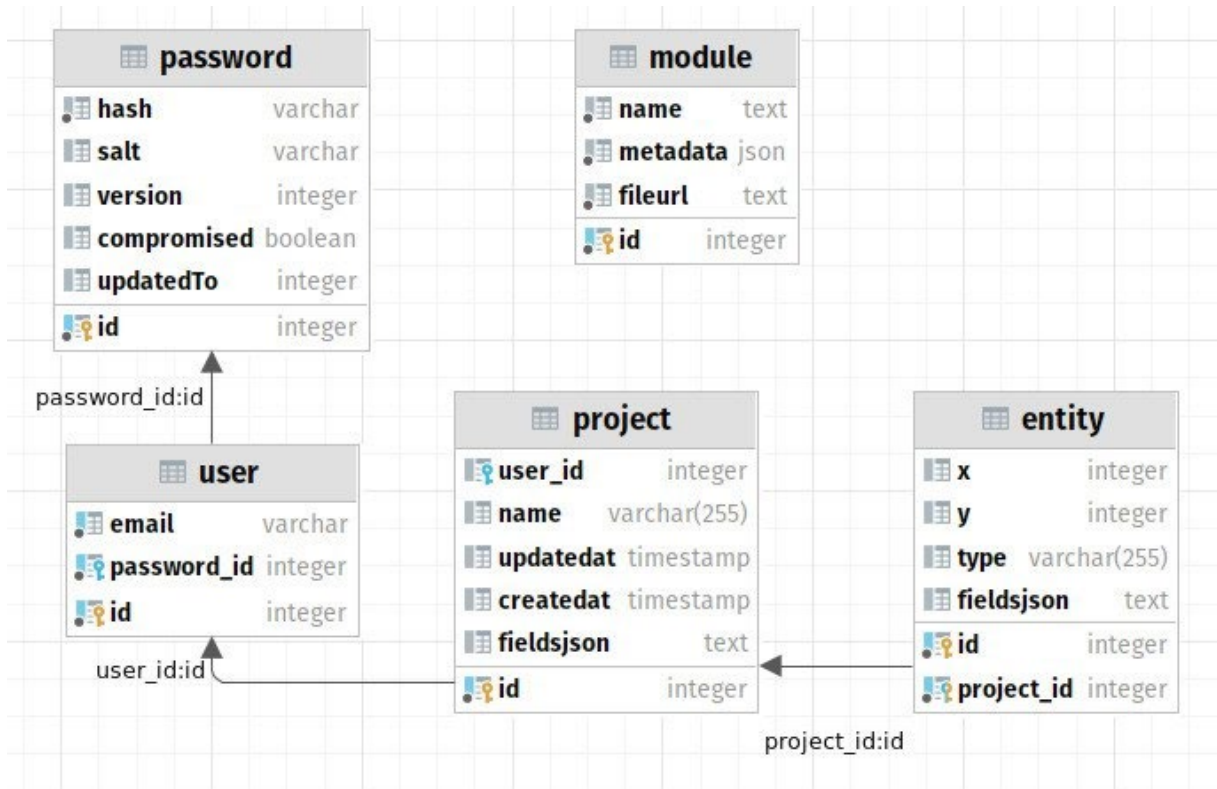


Рис. 2.9. Схема бази даних

2.5. Розгортання і випуск програмного забезпечення

В якості платформи для розгортання веб-застосунку було обрано Heroku, через простоту застосування і фінансову доступність. В якості платформи для CI/CD було обрано GitHub Actions[15], який надає усі необхідну функціональність для розгортання і при цьому є безкоштовним та вбудованим у GitHub.

GitHub Actions конфігурується за допомогою yaml файлу, в якому описується процес розгортання. Цей процес відбувається при доданні нового коду у головну гілку репозиторію. В цей час створюються необхідні образи Docker і за допомогою heroku-deploy розгортаються на Heroku. Приклади повідомлень про успішне розгортання клієнту і серверу зображено на рисунках 2.10 і 2.11.

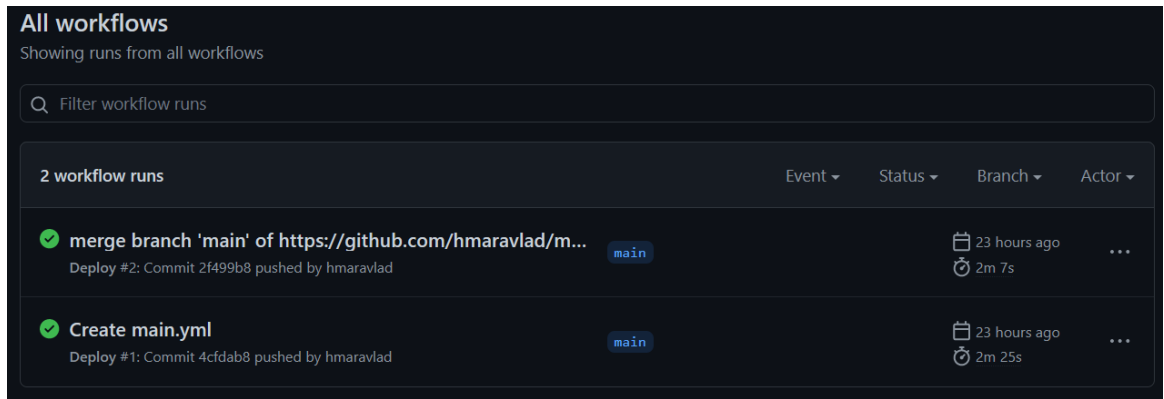


Рис. 2.10. Повідомлення про успішне розгортання клієнтської частини проекту

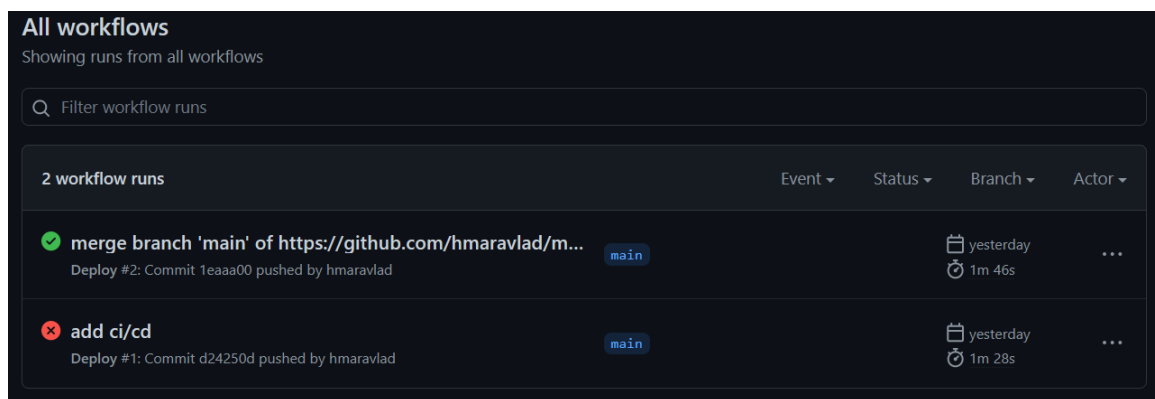


Рис. 2.11. Повідомлення про успішне розгортання серверної частини проекту

Консольний застосунок розповсюджується як виконуваний (exe) файл для Windows і як debian пакет для Linux. Для їх формування також використовується GitHub Actions. Процес пакування починається при зміні версії консольного застосунку. Виконувані файли формуються за допомогою бібліотеки rkg, після цього створюється deb пакет, файли архівуються і прикріплюються до нового релізу у GitHub (рис 2.12).

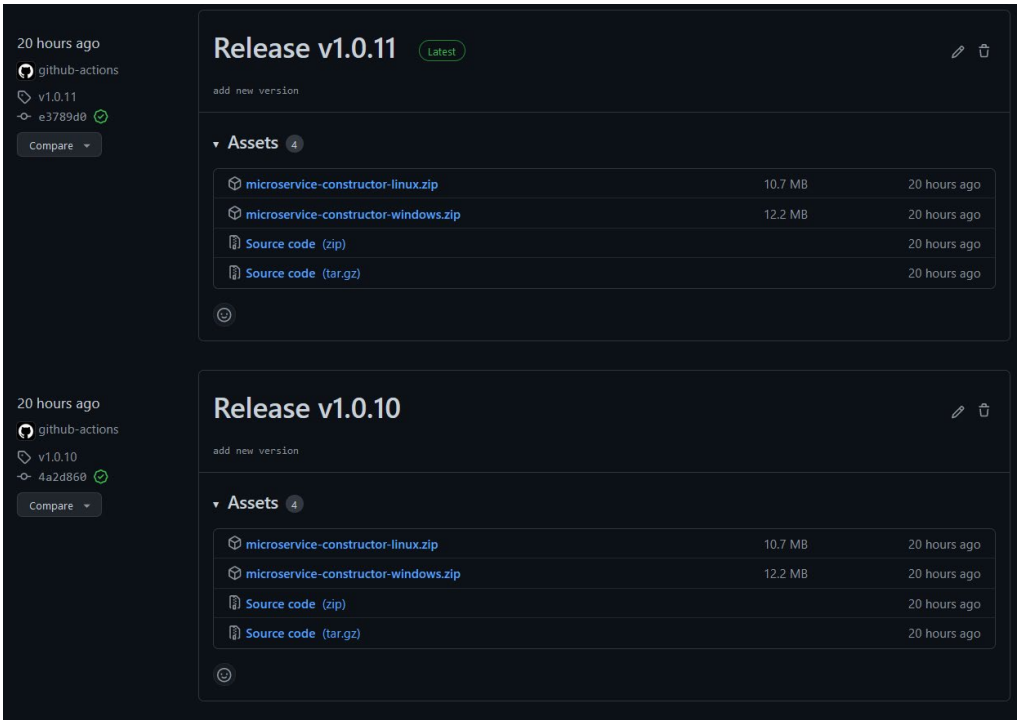


Рис. 2.12. Реліз у GitHub, де можна скачати консольний застосунок

Висновки до розділу 2

Було розроблено функціональні вимоги до застосунку. Ключовою вимогою є наявність можливості описати бажану інфраструктуру мікросервісного застосунку користуючись графічним інтерфейсом та згенерувати конфігураційні файли та частину коду цього застосунку на основі опису.

Застосунок складається з трьох головних частин:

- клієнтська частина, що надає користувачу графічний інтерфейс для налаштування бажаного застосунку;
- серверна частина, яка виконує авторизацію, збереження і завантаження проектів для клієнтської частини;
- консольний застосунок, який відповідає саме за генерацію макету застосунку.

Для розробки всіх трьох частин було використано мову Typescript. Для розробки клієнтської частини було використано фреймворк Angular. Для розробки серверної частини було використано фреймворк NestJs.

Клієнтську і серверну частину було розгорнуто на Heroku. Консольний застосунок був випущений як deb-пакет для Linux і виконуваний файл для Windows.

РОЗДІЛ 3

ОЦІНЮВАННЯ ІНСТРУМЕНТУ ДЛЯ АВТОМАТИЗАЦІЇ РОЗРОБКИ ДОДАТКІВ З МІКРОСЕРВІСНОЮ АРХІТЕКТУРОЮ

3.1. Методи оцінки інструментів розробки програмного забезпечення

Методи оцінки інструментів можна розділити на:

- Кількісні
- Якісні

Кількісні методи направлені на визначення вимірюваних ефектів від використання оцінюваного інструменту. Як правило цими ефектами є зниження часу або вартості розробки або підтримки програмного забезпечення.

Якісні методи ж направлені на визначення відповідності інструменту поставленим задачам. Конкретні функції та характеристики, включені в оцінку, базуються на вимогах користувачів і організаційних стандартах. Як правило оцінювачі оцінюють, чи надає інструмент необхідну функціональність зручним та ефективним способом на основі особистої думки.

Також методи оцінки інструментів можна класифікувати в залежності від організації процесу оцінювання. Існують такі види методів[16]:

- формальний експеримент;
- кейс-дослідження
- опитування

У формальному експерименті учасників експерименту просять виконати завдання або набір завдань за допомогою різних інструментів які оцінюються. Учасники розподіляються між інструментами так, щоб уникнути викривлень у результатах. Результати можуть бути проаналізовані за допомогою звичайних статистичних технік. Оцінювання інструментів за допомогою проведення формального експерименту є найбільш корисним у таких умовах:

- переваги, які надає інструмент, можна чітко виміряти;

- існує можливість знайти людей, які готові взяти участь в експерименті;
- інструмент сконцентрований на виконанні одного конкретного завдання;
- можна навчитись роботі з інструментом за відносно низький час;
- незалежність оцінки від контексту є бажаним результатом.

Кейс-дослідження передбачають оцінку методу/інструменту після його використання в «реальному» проекті. Проведення кейс-досліджень має ряд переваг:

- Проведення кейс-дослідження може бути включене до звичайної діяльності компанії з розробки програмного забезпечення.
- У кейс-дослідження немає проблем з масштабуванням результатів, адже воно вже відбувається на реальному проекті.
- Кейс-дослідження дозволяє визначити чи будуть виявлені очікувані ефекти в організаційних і культурних умовах конкретної компанії.

З іншого боку кейс-дослідження також мають набір важливих недоліків:

- При незначній реплікації або без неї вони можуть давати неточні результати.
- Немає гарантії, що подібні результати будуть виявлені в інших проектах.
- Не завжди існує можливість інтерпретувати результати дослідження. Наприклад, недостатньо знати значення продуктивності для проекту з новим інструментом. Також необхідно мати базове значення продуктивності з яким можна його порівняти.
- Стандарти і процедури проведення кейс-досліджень рідко є узгодженими. Різні дисципліни як правило мають різні підходи до їх проведення.

Опитування використовуються, коли в організації (або групі організацій) використовуються різні інструменти. У цій ситуації користувачів інструментів

можна попросити надати інформацію про будь-яку оцінювану властивість (наприклад, продуктивність розробки, дефекти після випуску тощо). Інформація від кількох користувачів інструменту передбачає реплікацію. Це означає, що ми можемо використовувати статистичні методи для оцінки відмінностей між інструментом щодо оцінюваних властивостей. Головна відмінність між формальним експериментом і опитуванням полягає в тому, що опитування зазвичай проводиться після завершення експерименту та гірше контролюється, ніж експеримент. Через це опитування мають ряд недоліків:

- Опитування показують лише асоціацію, а не причинність.
- Існує ризик статистичного викривлення через специфіку компаній, які погодились на опитування. Так, наприклад, компанії, які погодились на опитування і зберігають необхідну для дослідження інформацію скоріше за все будуть більш якісно організованими, що може стати причиною викривлення.
- Для проведення опитування необхідно, щоб різні організації зберігали дані, які можна порівнювати.

З іншого боку опитування мають і ряд переваг:

- Вони використовують вже наявні дані.
- Вони можуть підтвердити, що ефект поширюється на багато організацій.
- Для їх проведення і аналізу використовуються стандартні статистичні методи.

При типовому аналізі функціональності, одна особа визначає вимоги, зіставляє їх з різними функціям, а потім оцінює, наскільки альтернативні інструменти їх підтримують, випробувавши інструмент або переглянувши відповідну літературу/документацію. Але окрім цього три способи організації процесу оцінювання також можна застосувати до якісного аналізу:

- Якісний експеримент: оцінка функцій, виконана групою потенційних користувачів. Це передбачає розробку критеріїв оцінки

та методу аналізу результатів за допомогою стандартних підходів до аналізу функціональності, але при цьому організовуючи процес оцінювання як експеримент (наприклад, з використанням випадкового набору потенційних користувачів для проведення оцінки).

- Якісне прикладне дослідження: оцінка функцій, яка проводиться після того, як інструмент було використано на практиці в реальному проекті.
- Якісне опитування: оцінка функцій, виконана людьми, які мають досвід використання або вивчали оцінювані інструменти. Це передбачає розробку критеріїв оцінки та методу аналізу результатів за допомогою стандартних підходів до аналізу функціональності, але при цьому організовуючи процес оцінювання як опитування

Так як переваги, які надає програмне забезпечення для генерування шаблонів мікросервісних застосунків, можна чітко виміряти, програмне забезпечення сконцентроване на виконанні одного завдання, має низький поріг входу і ми бажаємо оцінити його ефективність незалежно від контексту в якості головного способу оцінки було обрано саме формальний експеримент.

3.2. Порівняння з існуючими інструментами

Було проведено пошук ПЗ, яке дозволяло б певною мірою автоматизувати розробку мікросервісних застосунків. Були знайдені такі інструменти:

- Менеджер пакетів Helm
- Генератори yaml файлів, як k8syaml, gimlet або Kubernetes YAML Generator.
- Генератор застосунків JHipster

Розглянемо кожен з них окремо.

3.2.1 Helm

Helm – це менеджер пакетів для Kubernetes [17]. У Helm ці пакети називаються чартами (charts). Кожен пакет описує набір пов’язаних ресурсів Kubernetes. Так, один пакет може описувати цілий мікросервісний веб-застосунок з серверами, базами даних, API шлюзами та брокерами повідомлень, або ж якийсь один з цих елементів.

Кожен пакет описується за допомогою набору файлів, де визначається назва пакету, версія пакету, набір шаблонів для конфігураційних файлів kubernetes тощо. (див. табл. 3.1)

Таблиця 3.1

Структура файлів Helm пакету

Chart.yaml	Містить головну інформацію про пакет
LICENSE	Текстовий файл, що містить ліцензійну інформацію про пакет
README.md	Зрозумілий людині файл з описом пакету
values.yaml	Файл, який визначає набір значень, які будуть підставлені у шаблони за замовчуванням
values.schema.json	JSON-схема для перевірки структури файлу values.yaml
charts/	Директорія, що містить пакети від яких залежить даний пакет
crds/	Директорія для визначення спеціальних ресурсів
templates/	Директорія з шаблонами для конфігураційних файлів Kubernetes

У Helm пакети підтримують версіонування за допомогою семантичного версіонування. Це схема нумерації версій в якій номер версії складається з трьох

чисел – мажорної, міnorної і патч версій [18], і яка широко застосовується [19]. Також Helm дозволяє виконувати депрекацію пакетів, тобто позначати їх як ті, що не є рекомендованими для використання.

Головним елементом пакетів Helm є шаблони ресурсів Kubernetes. Для написання шаблонів використовується мова шаблонів Go доповнена рядом спеціальних функцій.

Наведемо приклад шаблону найпростішого конфігураційного файлу для розгортання Kubernetes. Він являє собою yaml файл у якому ряд значень вигляду `{{ .Values.<назва поля> }}` підставляються із зовнішнього файлу. Наприклад, виставивши у файлі `values.yaml` значення “serviceName” у “nginx”, “imageVersion” у “1.14.2”, “replicaCount” у “3” та “port” у “80”, зможемо згенерувати конфігураційний файл для розгортання трьох реплік http-серверу nginx версії 1.14.2

```

! example.tpl
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: {{ .Values.serviceName }}-deployment
5    labels:
6      app: {{ .Values.serviceName }}
7  spec:
8    replicas: {{ .Values.replicaCount }}
9    selector:
10     matchLabels:
11       app: {{ .Values.serviceName }}
12   template:
13     metadata:
14       labels:
15         app: {{ .Values.serviceName }}
16     spec:
17       containers:
18       - name: {{ .Values.serviceName }}
19         image: {{ .Values.serviceName }}:{{ .Values.imageVersion }}
20         ports:
21         - containerPort: 80

```

Рис. 3.1. Приклад шаблону для розгортання у Kubernetes

```
! values.yaml
1  serviceName: "nginx"
2  imageVersion: "1.14.2"
3  replicaCount: "3"
4  port: "80"
```

Рис. 3.2. Приклад заповнення значень для шаблону

На відміну від розробленого програмного забезпечення Helm не надає графічного інтерфейсу – взаємодія з Helm відбувається за допомогою CLI. Також Helm використовується саме для генерації конфігураційних файлів Kubernetes – за допомогою нього не генерується код самих мікросервісів.

3.2.2 Візуальні генератори конфігураційних файлів

Існує ряд інструментів, які мають схожу функціональність – вони надають графічний інтерфейс для спрощеного створення конфігураційних файлів для kubernetes. Цей інтерфейс являє собою набір зрозумілих людині полів для вводу конфігурації ресурсів Kubernetes (на рис. 3.3 продемонстровано створення конфігураційного файлу для розгортання трьох реплік http-серверу nginx версії 1.14.2).

Kubernetes YAML Generator

Kubernetes Object Type

Deployment

Name

nginx

Image

nginx:1.14.2

Replicas

3

Container Port

80

GENERATE YAML ➤

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 80
```

DOWNLOAD ⬇️ 📄

Рис. 3.3. Приклад візуального генератора конфігураційних файлів

Прикладами таких інструментів є:

- k8syaml[20];
- gimlet[21];

- Kubernetes YAML Generator[22].

В залежності від специфіки інструментів це дозволяє:

- Приховати складність – у деяких інструментах такого роду користувачу надаються лише основні налаштування, що дозволяє швидко створити ресурс (Kubernetes YAML Generator).
- Спростити вивчення Kubernetes – у інших інструментах (k8syaml, gimlet) такого роду набагато більше опцій, які розділені на тематичні сегменти, при цьому пояснення опцій і можливість їх змінити знаходяться в одному місці.
- Підтримати консистентність даних – так, часто при зміні одного значення, треба його змінити у багатьох місцях в конфігураційному файлі. Наприклад, назва сервісу часто використовується не тільки в параметрі “name”, але і в параметрах “labels ” і “matchLabels”. Таким чином використання шаблонів Helm або подібних інструментів з графічним інтерфейсом виключає можливість помилки, в якій користувач випадково не змінює один з параметрів конфігурації.
- Забезпечити правильність конфігураційного файлу – такі інструменти забезпечують синтаксичну правильність конфігураційного файлу за рахунок того, що користувач може змінювати лише значення певних полів.

На відміну від розробленого ПЗ (та розглянутого перед цим Helm) такі інструменти дозволяють створювати лише окремі ресурси, а не інтегровані набори ресурсів, що сильно обмежує їхню сферу застосування. Крім того вони застосовуються лише для генерації конфігураційних файлів – вони не генерують код самих мікросервісів.

3.2.3 JHipster

JHipster - це інструмент для автоматизації розробки мікросервісних застосунків.

Для генерації застосунків використовуються конфігураційні jdl (DSL для опису конфігурації) файли. Ці файли можна редагувати як у спеціалізованому редакторі JDL Studio, який також візуалізує схему застосунку (див. рис. 3.4), так і у текстових редакторах загального призначення таких як VSCode або Eclipse.

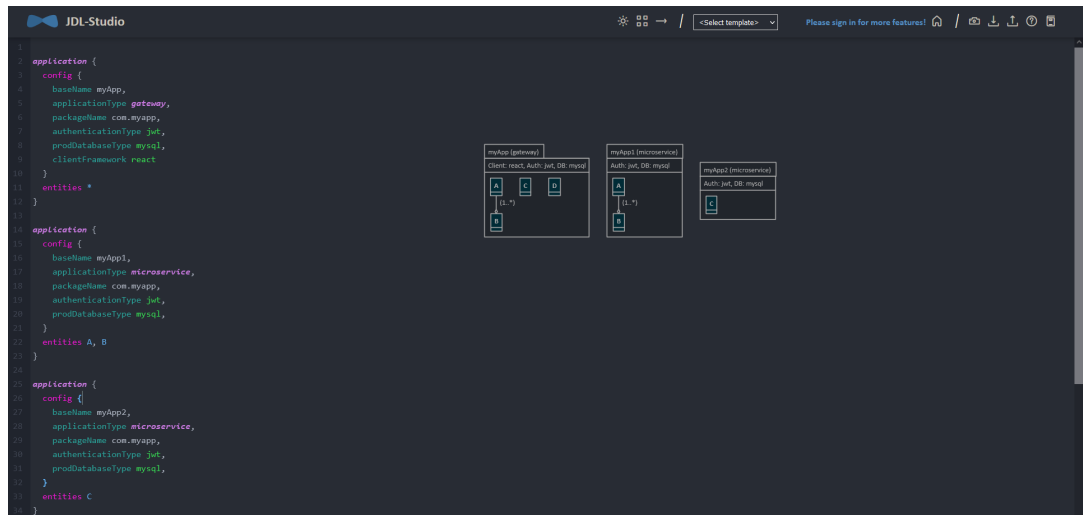


Рис. 3.4. JDL Studio

ЇHipster підтримує такі типи ресурсів:

- API шлюзи, які обробляють web-трафік та обслуговують фронт-енд;
- реєстри мікросервісів, які використовуються для виявлення сервісів і надання їм конфігурації, для цього використовуються або Consul, або ЇHipster Registry;
- власне мікросервіси.

Параметри усіх цих елементів описуються у конфігураційному jdl файлі. Так для кожного сервісу визначається назва, тип автентифікації, тип бази даних тощо (для прикладу визначення простого сервісу див рис. 3.5).

```

26 application {
27   config {
28     baseName service1
29     applicationType microservice
30     packageName com.service1
31     authenticationType jwt
32     prodDatabaseType postgresql
33   }
34   entities C
35 }

```

Рис. 3.5. Визначення сервісу у JDL

Також JHipster також підтримує генерацію коду самих сервісів. Для цього у конфігураційному JDL файлі описується набір сутностей, кожна з яких має набір полів певного типу а також зв'язки з іншими сутностями. Для кожної сутності генерується код сервісу, який підтримує CRUD-операції, а також код для створення відповідної таблиці у базі даних. (на рисунках 3.6, 3.7 зображений опис сутності Employee, яка пов'язана з сутностями Job і Department, на рисунку 3.8 зображена схема мікросервісу)

```

32 entity Employee {
33   firstName String
34   lastName String
35   email String
36   phoneNumber String
37   hireDate Instant
38   salary Long
39   commissionPct Long
40 }
70 relationship ManyToMany {
71   Job{task(title)} to Task{job}
72 }
73
74 relationship OneToMany {
75   Employee to Job{employee}
76   Department to Employee{department}
77 }

```

Рис 3.6, 3.7. Визначення сутностей і зв'язків у JDL

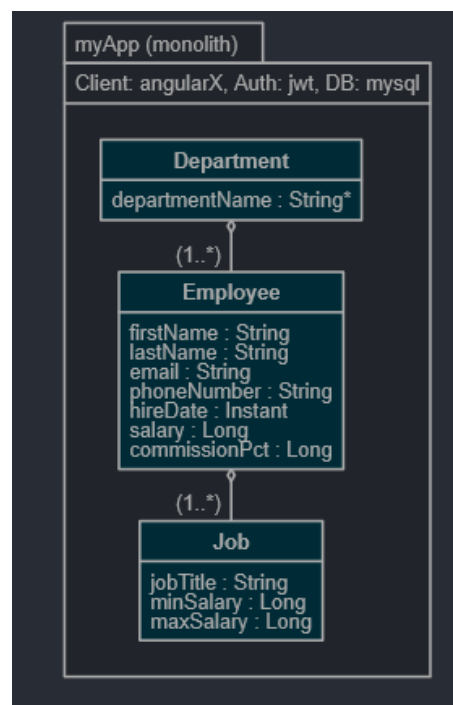


Рис 3.8. Схема мікросервісу у JDL Studio

Після цього з конфігураційного jdl файлу за допомогою CLI JHipster генерується код описаного застосунку. Для генерації конфігураційних файлів Kubernetes необхідно запустити додатковий генератор. При цьому на відміну від

розробленого ПЗ конфігураційні файли Kubernetes для розгортання баз даних та брокерів повідомлень не генеруються.

3.2.4 Загальні результати порівняння

В таблиці 3.2 розглянуто ступінь підтримки розглянутими альтернативами та розробленим ПЗ функціональності для автоматизації розробки мікросервісних застосунків.

Таблиця 3.2

Порівняння функціональності розробленого ПЗ з аналогами

Функція	Helm	k8syaml	JHipster	Розроблене ПЗ
Наявність графічного інтерфейсу для редагування конфігурації	-	+	-	+
Наявність візуальної репрезентації архітектури застосунку у вигляді схеми/графіку	-	-	+	+
Можливість генерувати конфігураційні файли Kubernetes для розгортання мікросервісів	+	+	+	+
Можливість генерувати конфігураційні файли Kubernetes для розгортання баз даних	+	+	-	+
Можливість генерувати конфігураційні файли Kubernetes для розгортання брокерів повідомлень	+	+	-	+
Можливість генерувати конфігураційні файли Kubernetes для розгортання шлюзів API	+	+	+	+
Можливість генерувати конфігураційні файли для наборів взаємопов'язаних ресурсів	+	-	+	+

Продовження таблиці 3.2

Можливість генерувати код власне мікросервісів	-	-	+	+
Можливість генерувати документацію для мікросервісів	-	-	+	+
Можливість генерувати автоматичні тести для мікросервісів	-	-	+	+
Можливість генерувати конфігураційні файли для CI/CD	-	-	+	+
Можливість генерувати код фронт-енду	-	-	+	-
Управління секретними значеннями	+	-	-	+

З даної таблиці можемо побачити, що найближчим аналогом є JHipster. І Helm, і k8syaml обмежені генерацією лише конфігураційних файлів kubernetes. При цьому k8syaml має досить схожий підхід до конфігурування Kuberene^{ts} як і розроблене ПЗ, проте у ньому існує лише можливість генерації окремих ресурсів, а не їх взаємопов'язаних наборів. Хоча у JHipster генерація конфігураційних файлів відбувається за допомогою написання коду, проте він надає візуальну схему застосунку. Також він підтримує генерацію коду, документації та тестів мікросервісів а також конфігураційних файлів kubernetes для їх розгортання. Тому порівнювати створене ПЗ будемо саме з JHipster.

3.3. Проведення експерименту для оцінки ефективності розроблених інструментів

3.3.1 Вибір статистичного тесту

Метою роботи є зменшення часу необхідного для розробки додатків з мікросервісною архітектурою за рахунок розробки програмного забезпечення, яке автоматизує певні етапи розробки. На основі цієї мети сформулюємо нуль-гіпотезу H_0 і альтернативну гіпотезу H_1 щодо взаємозв'язку використовуваного інструменту і часу витраченого на розробку. При цьому порівнювати будемо з інструментом JHipster який є найближчим аналогом.

H_0 – створення мікросервісного застосунку і налаштування відповідної інфраструктури за допомогою JHipster займає стільки ж часу, скільки і з використанням розробленого програмного забезпечення

H_1 – створення мікросервісного застосунку і налаштування відповідної інфраструктури за допомогою розробленого програмного забезпечення займає менше часу, ніж при використанні інструменту JHipster.

Для перевірки статистичних гіпотез існують різні види тестів. Тести вибираються в залежності від типів даних, а також певних ключових властивостей даних[23]. Виділяють такі типи даних:

- Неперервні – дані можуть набувати двох конкретних дійсних значень, так що вони також можуть набувати всіх дійсних значень між ними (навіть значень, які як завгодно близькі).
- Дискретні – дані можуть набувати таких значень, де на кожній стороні існує не нескінченно малий проміжок, який не містить значень. При цьому значення мають порядок ранжування, шкала, проте, не обов'язково є лінійною. Прикладом такої величини є шкала болю від 1 до 10; оцінка болю 6 не обов'язково вдвічі гірша за 3.
- Категоричні – дані не мають порядку ранжування. Прикладом є колір або форма.

У нашому випадку незалежною змінною є використовуваний інструмент, який має категоричний тип даних. Залежною змінною є час витрачений на

створення мікросервісного застосунку і налаштування його інфраструктури. Час в даному контексті має неперервні значення.

Статистичні тести поділяються на:

- параметричні;
- непараметричні.

Як правило параметричні тести потребують менше даних ніж непараметричні тести для того, щоб отримати статистично значимий результат. При цьому параметричні тести вимагають від даних виконання певних умов, в той час як непараметричні тести можна використовувати навіть, якщо ці умови не виконуються. Цими умовами є:

- дані повинні бути нормально розподілені;
- дані повинні мати однакову дисперсію і середнє квадратичне відхилення;
- дані повинні бути неперервними.

Існують такі параметричні тести:

- Коефіцієнт кореляції Пірсона – це значення r , яке є пропорцією між коваріацією двох змінних і добутком їх середніх квадратичних відхилень, і яке показує ступінь за яким 2 неперервні змінні лінійно корелюють одна з одною. Значення r , що дорівнює 1,0, означає, що дані повністю корельовані, і одна змінна може бути використана для обчислення іншої. Значення r , що дорівнює нулю, означає, що 2 змінні абсолютно випадкові. Значення r , яке дорівнює -1,0 означає повну негативну кореляцію.
- t-критерій Стюдента – тип аналізу, який використовується для порівняння середніх значень двох груп і визначення того, з якою ймовірністю відмінності між ними є результатом випадковості.
- Дисперсійний аналіз (ANOVA) — це тест, який використовує середні значення та дисперсії для визначення F-коефіцієнту. F-коефіцієнт потім використовується для визначення того, однаковими чи різними є групи даних. Коли перевірка гіпотези виконується за

допомогою дисперсійного аналізу, нульова гіпотеза полягає у тому, що всі групи однакові. Цей тест використовується коли необхідно порівняти три чи більше груп.

Так як незалежна змінна, що використовується нами не є неперервною, ми не можемо використати коефіцієнт кореляції. Через те, що ми перевіряємо два інструменти – JHipster і розроблене ПЗ, перевірити необхідно дві групи. Таким чином ми можемо використати t-критерій Стюдента.

Альтернативою t-критерію Стюдента є U-критерій Манна-Уїтні [24]. Це непараметричний тест нульової гіпотези про те, що для випадково вибраних значень X і Y з двох сукупностей ймовірність того, що X буде більшим за Y , дорівнює ймовірності того, що Y буде більшим за X . На відміну від t-критерію Стюдента його можна використовувати у випадку, якщо розподіл значень у групах не відповідає нормальному.

Для етапу оцінки необхідного розміру вибірки припустимо відповідність значень у групах нормальному розподілу та однаковість середніх квадратичних відхилень у групах. Таким чином може використовуватись t-критерій Стюдента. Дійсну відповідність даним цим критеріям перевіримо після збору експериментальних даних і у випадку їх невиконання змінимо тест на U-критерій Манна-Уїтні.

Існують декілька типів t-критерію Стюдента:

- Одновибірковий t-критерій – перевіряє чи має середнє значення генеральної сукупності значення, визначене нульовою гіпотезою.
- Парований двовибірковий t-критерій – перевіряє таку нульову гіпотезу, що середні значення двох генеральні сукупностей рівні. Складається з вибірки відповідних пар подібних одиниць або однієї групи одиниць, яка була перевірена двічі.
- Непарований двовибірковий t-критерій – також перевіряє таку нульову гіпотезу, що середні значення двох генеральні сукупностей рівні. Використовується, коли порівнюються дві окремі, незалежні і

однаково розподілені вибірки і порівнюють одну змінну з кожної з двох генеральних сукупностей.

Так як ми будемо порівнювати дві групи, які будуть розподілені в залежності від використовуваного інструменту, і так як у нас немає вже відомого середнього значення генеральної сукупності – необхідно використовувати двовибірковий t-критерій. Парований t-критерій можна було б використати, обравши такий порядок проведення експерименту, де кожен учасник створює мікросервісний застосунок двічі – по черзі кожним з інструментів. При цьому порядок в якому використовуються інструменти може бути рандомізований для того, щоб уникнути викривлення пов'язаного з тим, що учасник експерименту вчиться в процесі його проведення [25]. Такий дизайн експерименту не був обраний через те, що він суттєво ускладнює набір учасників через збільшення часу необхідного на проведення експерименту. Таким чином було обрано непарований двовибірковий t-критерій.

3.3.2 Визначення розміру вибірки за допомогою аналізу статистичної потужності

Виділяють два роди помилок, які виникають при перевірці статистичних гіпотез[26]:

- Помилка першого роду – це помилка, допущена, коли нуль-гіпотеза H_0 є відхиленою, хоча при цьому є істинною.
- Помилка другого роду - це помилка, допущена, коли нуль-гіпотеза H_0 не є відхиленою, але при цьому є хибною.

Статистична значущість α – це ймовірність вчинення помилки першого роду.

Статистична потужність $1 - \beta$ - це ймовірність того, що тест правильно відкидає нульову гіпотезу H_0 коли альтернативна гіпотеза H_1 істина, де β - це ймовірність помилки другого роду.

Для α і β оберемо традиційне значення – 0.05 [27].

Для визначення розміру вибірки використаємо взаємовідношення[28]:

$$d\sqrt{\frac{n}{2}} = t_{1-\alpha} + t_{1-\beta} \quad (3.1)$$

де:

- d – розмір ефекту
- n – розмір вибірки
- $t_{1-\alpha}$ – це квантиль нормального розподілу для статистичної значущості
- $t_{1-\beta}$ – це квантиль нормального розподілу для статистичної потужності

Для того, щоб отримати розмір вибірки перепишемо це рівняння так:

$$n = 2 \frac{(t_{1-\alpha} + t_{1-\beta})^2}{d^2} \quad (3.2)$$

Розмір ефекту d визначається за допомогою формули:

$$d = \frac{m_A - m_B}{\sigma} \quad (3.2)$$

де:

- m_A – середнє значення групи А
- m_B – середнє значення групи В
- σ – середньоквадратичне відхилення будь-якої групи (адже ми припускаємо, що вони є однаковими)
- Для розміру ефекту існують традиційні значення[28]:
- малий ефект $d = 0.20$;
- середній ефект $d = 0.50$;
- великий ефект $d = 0.80$.

Проте ми очікуємо ще більш значний розмір ефекту, ніж стандартний розмір для великого ефекту. Для того, щоб обчислити розмір ефекту було проведено пілотний експеримент з трьома учасниками. Це досить поширена практика, яка дозволяє знайти недоліки у дизайні експерименту до його повномасштабного проведення, а також отримати попередній розмір ефекту для обчислення розміру вибірки [25]. Результати пілотного експерименту наведено у таблиці 3.3.

Таблиця 3.3

Результати пілотного експерименту

№ Учасника	Час на створення мікросервісного застосунку і його розгортання з використанням JHipster (хв)	Час на створення мікросервісного застосунку і його розгортання з використанням розробленого ПЗ (хв)
1	37	26
2	44	36
3	35	29

Середнє значення для групи, яка використовувала JHipster $m_A = 38.67$.
 Середнє значення для групи, яка використовувала розроблене ПЗ $m_B = 30.33$.
 Середньоквадратичне відхилення другої групи $\sigma = 4.19$.

Таким чином можемо обчислити розмір ефекту за допомогою формули 3.2:

$$d = \frac{m_A - m_B}{\sigma} = \frac{38.67 - 30.33}{4.19} = 1.99$$

Значення для квантилів для $\alpha = \beta = 0.05$ мають табличні значення $t_{1-\alpha} = t_{1-\beta} = 1.64$ [29].

Тепер можемо обчислити необхідний розмір вибірки за допомогою формули 3.1:

$$n = 2 \frac{(1.64 + 1.64)^2}{1.99^2} = 5.43$$

$$n = 6$$

Таким чином мінімально можливим розміром вибірки для однієї групи є $n = 6$. Але краще набрати дещо більшу кількість учасників, на випадок, якщо реальний розмір ефекту буде меншим, що є цілком можливим, адже в пілотному експерименті брав участь розробник експерименту, який краще знає розроблене ПЗ, що могло вплинути на результат.

3.3.3 Завдання учасників експерименту

Так як метою є перевірити економію часу при створенні мікросервісного застосунку і налаштуванні інфраструктури для його підтримки, то завдання, розроблене для виконання учасниками експерименту, полягає у створенні найпростішого мікросервісного застосунку. Спрощення в першу чергу стосується бізнес логіки, адже вона є специфічною для кожного застосунку і тому її розробку важко оптимізувати. Спрощення необхідне для того, щоб мати можливість провести експеримент за такий час, який дозволить знайти достатню кількість учасників. При цьому аспекти розробки пов'язані з налаштуванням інфраструктури суттєво спрощені для експерименту не були. Технології для написання сервісів було обрано так, щоб була можливість набрати учасників, які мають досвід роботи з ними. Технології і паттерни інфраструктури були обрано у відповідність з інформацією про найбільш відповідні мікросервісній архітектурі технології і паттерни наведеною у розділі 1.

Завдання полягає у створенні та розгортанні у Kubernetes кластері на Digital Ocean двох простих мікросервісів написаних з використаннями фреймворку NestJs. Перший мікросервіс повинен надавати через API шлюз доступ до операцій читання, створення, оновлення та видалення для простого ресурсу замовлень Order (таблиця 3.4). Другий мікросервіс – для ресурсу предметів меню Item (таблиця 3.5). Мікросервіси повинні мати можливість комунікувати між собою за допомогою HTTP з використанням механізмів виявлення сервісів. Для підтвердження цього сервіс для ресурсу Order повинен мати точку доступу API для отримання повної кількості екземплярів ресурсу Item. Дані повинні зберігатись у базах даних PostgreSQL. Кожен сервіс повинен мати свою базу даних. Правильність роботи сервісу перевіряється за допомогою автоматизованих тестів API створених у Postman.

Таблиця 3.4

Опис ресурсу Order

id	номер
number	номер
name	рядок
description	рядок
status	рядок
price	номер

Таблиця 3.5

Опис ресурсу Item

id	номер
name	рядок
description	рядок
price	номер

3.3.4 Набір і навчання учасників експерименту

Учасників експерименту було набрано з команди працівників компанії Labs Neural (див.табл.3.6). Кількість набраних учасників $n = 8$ на групу.

При наборі учасників були зібрані такі дані:

- комерційний досвід роботи (у роках);
- досвід роботи з мовою Typescript (самозвітований за шкалою від 1 до 3);
- досвід роботи з фреймворком NestJs (самозвітований за шкалою від 1 до 3);
- досвід роботи з PostgreSQL (самозвітований за шкалою від 1 до 3);
- досвід використання Cloud-сервісів (самозвітований за шкалою від 1 до 3);
- досвід використання Docker (самозвітований за шкалою від 1 до 3);
- досвід використання Kubernetes (самозвітований за шкалою від 1 до 3);

Більшість набраних учасників мають як мінімум 2 роки комерційного досвіду в якості розробників програмного забезпечення на платформі NodeJs та мають середній досвід роботи з cloud-платформами і контейнерами Docker, проте мають низький рівень досвіду з Kubernetes.

Учасників було призначено випадково між розробленим ПЗ і інструментом JHipster, щоб розподілити варіації поведінки учасників між умовами рівномірно і уникнути селективного упередження [30].

Навчальні матеріали містили інструкції з використання розробленого ПЗ, інструкції з використання JHipster та інформацію про роботу з Kubernetes у Digital Ocean. Також було показано запис з короткою роботою розробленого ПЗ та JHipster. Учасникам було надано аккаунти у Digital Ocean і DockerHub.

Таблиця 3.6

Демографічні дані учасників експерименту

№	Використаний інструмент	Комерційний досвід роботи (років)	Досвід роботи з мовою TypeScript	Досвід роботи з NestJs	Досвід роботи з PostgreSQL	Досвід використання Cloud-сервісів	Досвід використання Docker	Досвід використання Kubernetes
1	JH	2.6	2	2	3	1	2	1
2	JH	2.62	2	2	2	2	2	1
3	JH	3.55	3	3	3	2	1	1
4	JH	1.97	2	2	2	2	2	1
5	JH	3.34	3	2	2	2	2	1
6	JH	4.62	3	3	3	3	3	1

Продовження Таблиці 3.6

7	ЖН	0.95	1	1	2	1	1	1
8	ЖН	3.13	2	2	3	2	2	1
9	Роз. ПЗ	2.74	2	2	2	2	2	1
10	Роз. ПЗ	2.23	2	1	2	1	1	1
11	Роз. ПЗ	3.31	2	2	2	2	2	1
12	Роз. ПЗ	3.72	2	2	3	2	2	1
13	Роз. ПЗ	4.27	3	2	3	3	3	1
14	Роз. ПЗ	2.45	2	2	3	2	1	1
15	Роз. ПЗ	3.76	3	2	2	1	1	1
16	Роз. ПЗ	1.2	2	1	2	1	1	1

3.3.5 Обробка результатів експерименту

Результати експерименту можна побачити на таблиці 3.7.

Таблиця 3.7

Результати експерименту

ЖHipster		Розроблене ПЗ	
№ учасника	Час виконання завдання (хв)	№ учасника	Час виконання завдання (хв)
1	38	1	34
2	43	2	32
3	48	3	36
4	37	4	41
5	44	5	35
6	44	6	33
7	51	7	28
8	48	8	34

Тепер після збору результатів ми можемо перевірити виконання припущень про дані, необхідних для того, щоб використання параметричних тестів було можливим. Спочатку перевіримо рівність середніх квадратичних відхилень.

Для першої групи середнє квадратичне відхилення $s_A = 4.57$

Для другої групи середнє квадратичне відхилення $s_B = 3.44$

На практиці як правило цю умову вважають виконаною, якщо значення відхилення відрізняються менше, ніж в 2 рази [31], що у нашому випадку виконується.

Після цього перевіримо чи є розподіл наших величин нормальним. Для цього використаємо тест Шапіро-Уїлка, адже він має високу статистичну потужність і є одним з найпопулярніших [32]. Статистика цього теста рораховується за формулою:

$$W = \frac{(\sum a_i y_{(i)})^2}{\sum (y - \bar{y})^2} \quad (3.3)$$

де $y_{(i)}$ – це статистика i -го порядку, а a_i – це i -те очікуване значення нормалізованого порядку статистик. Ці значення a_i можна отримати з таблиці підготованої Шапіро і Уїлком. Після цього значення статистики W порівнюється з 1. Якщо W значно менше 1, гіпотезу нормальності буде відхилено. Для перевірки нормальності за допомогою тесту Шапіро-Уїлка використаємо статистичний калькулятор[33]. Для цього введемо туди отримані експериментальні дані та традиційний рівень статистичної значущості $\alpha = 0.05$.

І перша, і друга група мають нормальний розподіл зі значенням W відповідно 0.9376 і 0.95 (див.рис. 3.9, 3.10).

Shapiro-Wilk Test Calculator

*The maximum sample size is 5000

Name:

Significance level (α):

Outliers:

Rounding:

Data:

```

38
43
48
37
44
44
51
48
  
```

Reporting Shapiro-Wilk test in APA Format

The Shapiro-Wilk test did not show a significant departure from normality, $W(8) = .94, p = .658$

Рис. 3.9. Результати перевірки нормальності першої групи

Shapiro-Wilk Test Calculator

*The maximum sample size is 5000

Name:

Significance level (α):

Outliers:

Rounding:

Data:

```

34
32
36
41
35
33
28
34
  
```

Reporting Shapiro-Wilk test in APA Format

The Shapiro-Wilk test did not show a significant departure from normality, $W(8) = .95, p = .770$

Рис. 3.10. Результати перевірки нормальності другої групи

Тепер після того, як ми підтвердили виконання необхідних умов, ми можемо обчислити t статистику.

Для цього використаємо формулу [34]:

$$t = \frac{X_A - X_B}{s_p \sqrt{\frac{2}{n}}} \quad (3.4)$$

де:

- X_A і X_B – це середні значення груп
- n – розмір вибірки

- s_p – зведене стандартне відхилення

Зведене стандартне відхилення можна обчислити за формулою:

$$s_p = \sqrt{\frac{s_A^2 + s_B^2}{2}} \quad (3.5)$$

де s_A і s_B – це середнє квадратичне відхилення відповідно для першої і другої групи.

Обчислимо t за допомогою формули 3.4 зі значеннями $X_A = 44.125$, $X_B = 34.125$, $s_A = 4.57$, $s_B = 3.44$, $n = 8$.

$$s_p = \sqrt{\frac{4.57^2 + 3.44^2}{2}} = 4.05$$

$$t = \frac{44.125 - 34.125}{4.05 \sqrt{\frac{2}{8}}} = 4.94$$

Тепер для перевірки статистичної значущості порівняємо отримане значення t статистику з теоретичним значенням t -розподілу. Це значення є табличним і отримується зі значення статистичної значущості $\alpha = 0.05$ та значення ступенів вільності $df = 2n - 2 = 12$. У нашому випадку це значення дорівнює 1.78 [29]. Так як воно менше за t -значення, яке ми отримали у процесі розрахунків – ми можемо відкинути нуль-гіпотезу H_0 , і прийняти гіпотезу H_1 , яка полягає у тому, що створення мікросервісного застосунку і налаштування відповідної інфраструктури за допомогою розробленого програмного забезпечення займає менше часу, ніж при використанні інструменту JHipster.

Висновки до розділу 3

Методи оцінки інструментів можна класифікувати в залежності від організації процесу оцінювання. Існують такі види методів:

- формальний експеримент;
- кейс-дослідження
- опитування

Також методи оцінки інструментів можна розділити на:

- Кількісні
- Якісні

Для проведення оцінки створеного програмного забезпечення було обрано кількісний формальний експеримент.

Було сформульовано дві гіпотези:

H₀ – створення мікросервісного застосунку і налаштування відповідної інфраструктури за допомогою JHipster займає стільки ж часу, скільки і з використанням розробленого програмного забезпечення

H₁ – створення мікросервісного застосунку і налаштування відповідної інфраструктури за допомогою розробленого програмного забезпечення займає менше часу, ніж при використанні інструменту JHipster.

Для перевірки гіпотез було обрано непаровний двовибірковий t-тест. За допомогою аналізу статистичної потужності було оцінено необхідний розмір вибірки для однієї групи $n = 6$. Було проведено експеримент зі співробітниками компанії Labs Neural з розміром вибірки для однієї групи $n = 8$. Розрахунок t значення дозволив відкинути нуль-гіпотезу **H₀**, і прийняти гіпотезу **H₁**, яка полягає у тому, що створення мікросервісного застосунку і налаштування відповідної інфраструктури за допомогою розробленого програмного забезпечення займає менше часу, ніж при використанні інструменту JHipster.

РОЗДІЛ 4
РОЗРОБКА СТАРТАП-ПРОЕКТУ

4.1. Розробка ідеї стартап-проекту

Суть ідеї стартап-проекту полягає у наданні користувачам платформи для автоматизованої розробки мікросервісних застосунків. Основні напрямки застосування ідеї та вигоди для користувача наведені у таблиці 4.1.

Таблиця 4.1

Ідея стартап-проекту		
Зміст ідеї	Напрямки застосування	Вигоди для користувача
Надання користувачу візуального інтерфейсу для формування опису бажаного застосунку, з якого генеруються частини готового застосунку і його інфраструктури	Генерація частини коду мікросервісів	Пришвидшення розробки мікросервісів
	Генерація документації API для мікросервісів	Пришвидшення розробки документації для мікросервісів
	Генерація автоматичних тестів для API мікросервісів	Пришвидшення написання тестів для мікросервісів
	Генерація конфігураційних файлів Kubernetes для розгортання мікросервісів, баз даних, брокерів повідомлень і API шлюзів	Пришвидшення налаштування інфраструктури проекту, зменшення порогу навчання для успішного налаштування інфраструктури проекту
	Генерація конфігураційних файлів для CI/CD	Пришвидшення налаштування CI/CD для проекту

Тепер порівняємо характеристики ідеї проекту з конкурентами:

- Helm
- k8yaml
- JHipster

Результати порівняння можемо бачити на таблиці 4.2.

Таблиця 4.2

Порівняння характеристик ідеї проекту з конкурентами

Характеристики ідеї проекту	Товари/концепції конкурентів				W	N	S
	Власне ПЗ	Helm	k8yaml	JHipster	(слабка сторона)	(нейтральна сторона)	(сильна сторона)
Візуальне редагування окремих елементів мікросервісної архітектури	+	-	+	-			+
Візуальне редагування всього застосунку	+	-	-	-			+
Наявність графічної схеми архітектури проекту	+	-	-	+			+
Можливість генерувати конфігураційні файли Kubernetes для розгортання мікросервісів	+	+	+	+		+	

Продовження таблиці 4.2

Можливість генерувати конфігураційні файли Kubernetes для розгортання баз даних	+	+	+	-		+	
Можливість генерувати конфігураційні файли Kubernetes для розгортання брокерів повідомлень	+	+	+	-		+	
Можливість генерувати конфігураційні файли Kubernetes для розгортання шлюзів API	+	+	+	+		+	
Можливість генерувати конфігураційні файли для наборів взаємопов'язаних ресурсів	+	+	-	+		+	
Можливість генерувати код власне мікросервісів	+	-	-	+			+
Можливість генерувати документацію для мікросервісів	+	-	-	+			+
Можливість генерувати автоматичні тести для мікросервісів	+	-	-	+			+
Можливість генерувати конфігураційні файли для CI/CD	+	-	-	+			+

Можливість генерувати код фронт-енду	-	-	-	+	+		
Управління секретними значеннями	+	+	-	-			+

В результаті порівняння можемо бачити, що хоч і більшість характеристик наявна у конкурентних продуктах, кожен конкурентний продукт має ряд важливих слабких характеристик. Так, Helm не підтримує візуальне редагування конфігурації, а також генерацію коду, документації та тестів для мікросервісів. Інструмент k8yaml також підтримує генерацію лише конфігураційних файлів Kubernetes і крім того неможливість створювати конфігурації для пов'язаних наборів ресурсів є серйозним недоліком. JHipster у свою чергу також не підтримує візуальне редагування конфігурації, а до того ж має обмеження у генерації певних ресурсів Kubernetes, хоча і підтримує генерацію фронт-енду на відміну від розробленого ПЗ.

4.2. Технічний аудит проекту

В рамках технічного аудиту проекту розглянемо технологічну здійсненність ідеї. На таблиці 4.3 подано інформацію про наявність і доступність необхідних технологій.

Таблиця 4.3.

Технологічна здійсненність ідеї проекту

Аспекти реалізації ідеї проекту	Технології їх реалізації	Наявність технологій	Доступність технологій
Консольний застосунок для генерації коду мікросервісів і конфігураційних файлів	Мови програмування загального використання	Наявні – Javascript, Typescript, C#, Java тощо	Доступні – наявні технології є безкоштовними

Продовження таблиці 4.3

Веб-застосунок для візуального редагування конфігурації	Мови для веб-програмування, фреймворки і бібліотеки для розробки фронт-енду	Наявні – фреймворки Angular, Vue, React для Javascript/Typescript	Доступні – наявні технології є безкоштовними
Серверний застосунок для керування і зберігання даних користувачів	Мови загального використання, фреймворки і бібліотеки для розробки бек-енду, системи управління базами даних	Наявні – фреймворки Koa, NestJs, Express для Javascript/Typescript, Spring для Java, ASP.NET для C# тощо. СУБД – Postgres, MySQL	Доступні – наявні технології є безкоштовними

В якості технологій було обрано мову Typescript для всіх частин застосунку – для спрощення розробки за рахунок використання лише однієї мови програмування. В якості фреймворку для фронт-енду було обрано Angular, в якості фреймворку для бек-енду було обрано NestJs. Обидва фреймворки було обрано завдяки наявності строгої структури застосунку з використання ін'єкції залежностей.

4.3. Аналіз ринкових можливостей запуску стартап-проекту

Спочатку проведемо попередній аналіз попиту. У таблиці 4.4. наведено дані по розміру ринку – загальні продажі компаній, що використовують мікросервісну архітектуру, а також дані про наявність перешкод для входу на ринок.

Таблиця 4.4

Розмір ринку мікросервісної розробки

Показники стану ринку	Характеристика
Загальний обсяг продаж, \$	2 760 000 000 [35]
Динаміка ринку	Зростає [35]
Специфічні вимоги до стандартизації та сертифікації	Відсутні
Сукупний середньорічний темп зростання	16.33% [35]

Таким чином ми можемо бачити, що мікросервісну архітектуру використовує значна кількість компаній, які мають достатню кількість ресурсів для того, щоб вкластись в автоматизацію.

Проаналізуємо потенційні цільові групи клієнтів стартап-проекту. У таблиці 4.5 наведені групи клієнтів, їх характеристики та перелік вимог до товару для кожної групи.

Таблиця 4.5

Потенційні цільові групи клієнтів стартап-проекту

Цільова аудиторія	Відмінності у поведінці потенційних груп клієнтів	Вимоги споживачів до товару
Розробники компаній, які вже використовують мікросервісну архітектуру	Вже мають налаштовану інфраструктуру, тому потребують саме спрощення розширення цієї інфраструктури на нові сервіси	Продукт повинен спрощувати/пришвидшувати підтримку та розширення мікросервісних застосунків, продукт повинен бути розширюваним, для того щоб включити вже створені рішення при розробці

Продовження таблиці 4.5

Розробники компаній, які планують перейти на мікросервісну архітектуру	Не мають налаштованої інфраструктури, налаштування початкової інфраструктури є одним з найбільших викликів при переході[36]	Продукт повинен спрощувати/пришвидшувати розробку мікросервісних застосунків, особливо на початкових етапах розробки
Розробники стартапів	Не мають налаштованої інфраструктури, експериментують з великою кількістю технологій	Продукт повинен спрощувати/пришвидшувати розробку мікросервісних застосунків, особливо на початкових етапах розробки Продукт повинен спрощувати прототипування

Розглянемо потенційні фактори загроз (табл. 4.6.) та можливостей (табл. 4.7.) на ринку.

Таблиця 4.6
Фактори загроз

Фактор	Зміст загрози	Можлива реакція компанії
Поява аналогічного товару	Поява аналогічного застосунку може призвести до зменшення прибутків через те, що клієнти можуть обрати альтернативний застосунок	Розширення набору підтримуваних технологій для автоматизації, покращення користувацької підтримки, розбудова лояльної клієнтської бази за рахунок маркетингу та підтримки

Продовження таблиці 4.6

Фінансування проекту	Недостатнє фінансування проекту	Залучення додаткових інвесторів, концентрація на основному наборі функцій і технологій та поступове додавання функцій і підтримуваних технологій вже після випуску застосунку
Охоплення клієнтів	Для розробки мікросервісних застосунків використовують широкий набір технологій. Певні технології можуть не підтримуватись застосунком, що лімітує охоплення клієнтів.	Побудова основи застосунку, що забезпечує розширюваність; Поступове розширення набору підтримуваних технологій; проведення досліджень для визначення пріоритетності додавання підтримки кожної технологій (в залежності від її популярності, розміру компаній які її використовують тощо)

Таблиця 4.7

Фактори можливостей

Фактор	Зміст можливості	Можлива реакція компанії
Функціональність продукту	Товар надає нову функціональність, яку потребують користувачі	Використання переваги для входу на ринок, розширення набору технологій для розширення набору потенційних клієнтів

Продовження таблиці 4.7

Розвинений ринок	Клієнти мають достатньо ресурсів, які вони можуть витратити на продукт	Використання переваги для входу на ринок
Зростання ринку	Кількість потенційних клієнтів зростає з часом	Використання переваги для входу на ринок

Проаналізуємо пропозицію на ринку. Характеристики конкуренції подані у таблиці 4.8.

Таблиця 4.8.

Характеристики конкуренції

Особливості конкурентного середовища	В чому проявляється дана характеристика	Вплив на діяльність підприємства
Тип конкуренції: монополістична	Продукти гравців ринку не є прямими аналогами один одного	Відсутність прямих аналогів дозволяє отримати значну ринкову перевагу
За рівнем конкурентної боротьби: світовий	Конкуренція не обмежується локальним або національним рівнем, клієнти та потенційні конкуренти можуть знаходитись у будь-якій частині світу	Повинні бути обрані пріоритетні регіони (США, Європа, Південно-Східна Азія тощо), на них повинні бути спрямовані рекламні кампанії, програмне забезпечення потребує англійської локалізації, а також локальними мовами за можливості

Продовження таблиці 4.8

За галузевою ознакою - внутрішньогалузева	Програмне забезпечення може використовуватись лише для автоматизації розробки мікросервісних застосунків	Використання відсутності конкуренції з інших галузей для отримання ринкової переваги
Конкуренція за видами товарів - товарно- родова	Конкуренція на рівні технології задоволення потреб клієнтів	Концентрація на розробці функціональності ПЗ, яке покращує рівень задоволення потреби користувачів – спрощення/пришвидшення розробки мікросервісних застосунків
За характером конкурентних переваг - нецінова	ПЗ конкурентів надають різну функціональність	Включення у розробку унікальної функціональності, включення у розробку елементи функціональності конкурентів
За інтенсивністю: немарочна	Торгова марка не є ключовим елементом конкуренції	Концентрації на розробці унікальної функціональності

Проведемо аналіз конкуренції за Портером[37] (таблиця 4.9). Він полягає у визначенні впливу на ринок 5 факторів: рівня конкурентної боротьби, ризику

появи нових конкурентів, впливу постачальників на ринок, впливу клієнтів на ринок та продуктів-замінників.

Таблиця 4.9

Аналіз конкуренції за Портером

Складові аналізу		Висновки
Прямі конкуренти в галузі	Helm, JHipster, k8yaml	Інтенсивність конкурентної боротьби з боку прямих конкурентів є низькою, адже вони не є прямими аналогами
Потенційні конкуренти	Немає вхідних бар'єрів, вартість бренду є низькою, стартові витрати є порівняно низькими	Є високий ризик появи нових конкурентів у майбутньому
Постачальники	Розробка ПЗ не потребує традиційних постачальників, потребуються Cloud-провайдери, ПЗ для розробки	Постачальники не диктують умови роботи на ринку, cloud-послуги, ПЗ для розробки є досить доступними
Клієнти	Велика кількість потенційних клієнтів, відсутність залежності від існуючих каналів дистрибуції, відсутність прямих продуктів-замінників	Здатність окремих споживачів до впливу на проект є досить низькою
Товари-замінники	Товари прямих конкурентів не мають важливих функцій	Товари замінники не вносять значних перешкод для входу на ринок

За результатами цього аналізу можемо бачити, що в цілому вихід на ринок є можливим, при цьому найбільшим фактором ризику є висока можливість появи нових конкурентів.

На основі аналізу конкуренції, характеристик ідеї проекту, факторів загроз та можливостей ринку та вимог споживачів визначимо перелік факторів конкурентоспроможності (Таблиця 4.10).

Таблиця 4.10

Обґрунтування факторів конкурентоспроможності

Фактор конкурентоспроможності	Обґрунтування
Функціональність - Наявність можливості візуального редагування конфігурації	Наявність можливості візуального редагування конфігурації спрощує освоєння програми, що потрібно користувачам
Функціональність - Повнота типів підтримуваних елементів інфраструктури мікросервісних додатків	Від кількості типів підтримуваних технологій залежить розмір набору потенційних клієнтів
Функціональність - Наявність можливості генерації частини коду мікросервісів	Наявність можливості генерації частин коду мікросервісів прискорює розробку, що потрібно користувачам
Функціональність - Можливість генерувати конфігурації/код для наборів взаємопов'язаних ресурсів	Наявність можливості генерації конфігурацій/коду для наборів взаємопов'язаних ресурсів прискорює розробку, що потрібно користувачам
Ціна	Зниження ціни товару збільшує ймовірність того, що користувач його придбає

Продовження таблиці 4.10

Можливість розширення функціональності застосунку за допомогою додаткових модулів	Така можливість може збільшити коло потенційних клієнтів за рахунок того що додаткова функціональність може покрити вимоги нових користувачів
Простота використання	Є ризик, що ПЗ яке складно використовувати отримає погану репутацію та, що існуючі клієнти з часом відмовляться від його використання

Проведемо порівняння розробленого ПЗ за факторами конкурентоспроможності з конкурентами. Для визначення слабких і сильних сторін проекту визначимо для кожного проекту бал від 0 до 3 за кожним фактором (таблиця 4.11). Можемо побачити, що сильними сторонами розробленого ПЗ є всі елементи функціональності, простота використання та можливість розширення функціональності застосунку за допомогою додаткових модулів. Слабкою ж стороною є ціна, адже при будь якій ціні розроблене ПЗ буде еквівалентним або дорожчим за продукти конкурентів, адже вони є безкоштовними.

Таблиця 4.11

Порівняння продуктів за факторами конкурентоспроможності

Фактор конкурентоспроможності	Розроблене ПЗ	JHipster	k8yaml	Helm
Функціональність - Наявність можливості візуального редагування конфігурації	3	1	2	0

Продовження таблиці 4.11

Функціональність - Повнота типів підтримуваних елементів інфраструктури мікросервісних додатків	3	2	1	3
Функціональність - Наявність можливості генерації частини коду мікросервісів	3	3	0	0
Функціональність - Можливість генерувати конфігурації/код для наборів взаємопов'язаних ресурсів	3	3	0	3
Ціна	0	3	3	3
Можливість розширення функціональності застосунку за допомогою додаткових модулів	3	3	0	3
Простота використання	3	2	3	1

Проведемо SWOT-аналіз[38]. Виділимо сильні (Strength) та слабкі (Weak) сторони проекту, а також можливості (Opportunities) та загрози (Threats) (Таблиця 4.12). На її основі визначимо альтернативи ринкової поведінки (Таблиця 4.13).

Таблиця 4.12

SWOT-матриця

Сильні сторони: функціональність застосунку; простота використання; можливість розширення функціональності застосунку за допомогою додаткових модулів.	Слабкі сторони: ціна.
---	--------------------------

Продовження таблиці 4.12

Можливості: відсутність конкурентів, що повторюють функціональність продукту; розвинений ринок для збуту; зростання ринку для збуту;	Загрози: ризик появи нових конкурентів; потенційні проблеми з фінансуванням проекту;
--	---

Таблиця 4.13

Альтернативи ринкового впровадження стартап-проекту

Орієнтовний комплекс заходів	Ймовірність отримання ресурсів	Орієнтовні строки реалізації
Визначення пріоритетного набору технологій для підтримки, вихід і маркетинг лише у пріоритетних регіонах, надання частини функціональності безкоштовно, оплата за преміум-функції/технічну підтримку	Низький строк виходу на ринок та менший обсяг робіт збільшує ймовірність отримання ресурсів	6 місяців
Підготовка підтримки для розширеного набору технологій, вихід і маркетинг лише у більшій кількості регіонів, надання частини безкоштовно, оплата за преміум-функції/технічну підтримку	Високий строк виходу на ринок та більший обсяг робіт зменшує ймовірність отримання ресурсів	1 рік

З огляду на ймовірність отримання ресурсів та орієнтовні строки реалізації було обрано такий комплекс заходів:

- визначення пріоритетного набору технологій для підтримки;

- вихід і маркетинг лише у пріоритетних регіонах;
- надання частини функціональності безкоштовно, оплата за преміум-функції/технічну підтримку.

4.4. Розроблення ринкової стратегії проекту

Опишемо групи потенційних користувачів (таблиця 4.14). Так як усі наведені групи мають достатньо високий рівень готовності користувачів сприйняти продукт та орієнтовний попит, а також низьку інтенсивність конкуренції та складність входу в сегменті – було вирішено обрати усі групи як цільові. Таким чином, так як компанія працюватиме з усім ринком, пропонуючи стандартизовану програму, раціонально використовувати масовий маркетинг.

Таблиця 4.14

Групи потенційних користувачів

Цільова група потенційних клієнтів	Готовність споживачів сприйняти продукт	Орієнтовний попит в межах цільової групи	Інтенсивність конкуренції в сегменті	Складність входу у сегмент
Розробники компаній, які вже використовують мікросервісну архітектуру	середній	середній	низька	низька
Розробники компаній, які планують перейти на мікросервісну архітектуру	високий	високий	низька	низька
Розробники стартапів	високий	високий	низька	низька

Так як ми знаходимося у ситуації, де жоден гравець на ринку не надає всього комплекту необхідних функцій у розробленому програмному забезпеченні, базовою стратегією розвитку оберемо стратегію диференціації. Ця стратегія полягає у фокусі на надання товару особливих властивостей, які виділяють його на фоні конкурентів.

Ця стратегія має такі переваги:

- знижує ступінь замінності товару, що підвищує лояльність клієнта до компанії, зменшує чутливість клієнта до ціни і таким чином підвищує рентабельність;
- лояльність клієнтської бази перешкоджає новим конкурентам при спробі виходу на ринок;
- підвищення рентабельності зменшує ризики для компанії у випадку потенційного зростання витрат.

В якості стратегії конкурентної поведінки було обрано стратегію розширення первинного рівня попиту. Вона полягає у навчанні споживачів користуванням товаром, формуванні регулярного попиту тощо. Вибір цієї стратегії обґрунтований тим, що взаємний тиск конкурентів не є високим і тим, що розроблене програмне забезпечення має функціональність, якою не володіють конкуренти, що дозволить простіше залучити нових користувачів.

4.4. Розроблення маркетингової програми стартап-проекту

Сформуємо маркетингову концепцію товару. Визначимо ключові переваги розробленого ПЗ перед конкурентами (Таблиця 4.15).

Таблиця 4.15

Переваги стартап-проекту над конкурентами

Потреба	Вигода, яку пропонує товар	Ключові переваги перед конкурентами
Пришвидшення написання коду мікросервісів	Генерація частини коду мікросервісів	Наявність функції, наявність можливості візуального редагування конфігурації

Продовження таблиці 4.15

Пришвидшення написання тестів та документації для мікросервісів	Генерація тестів для перевірки API та документації API мікросервісів	Наявність функції, наявність можливості візуального редагування конфігурації
Спрощення написання конфігураційних файлів для інфраструктури мікросервісного застосунку	Генерація конфігураційних файлів для інфраструктури мікросервісного застосунку	Наявність функції, підтримка більшої кількості базових типів елементів інфраструктури, наявність можливості візуального редагування конфігурації

Розробимо трирівневу маркетингову модель товару[39] (Таблиця 4.16).

Таблиця 4.16

Трирівнева маркетингова модель товару

Рівні товару	Сутність та складові		
Товар за задумом	Програмне забезпечення для автоматизації розробки мікросервісних застосунків		
Товар у реальному виконанні	Властивості/ характеристики	Монотонні(М)/ Немонотонні(Нм)	вартісні(Вр)/ технічні(Тх)/ технологічні(Тл)/ ергономічні(Е)/ органолептичні(Ор)
	надає можливість візуального редагування конфігурації	Нм	Тх

Продовження таблиці 4.16

Товар у реальному виконанні	надає можливість генерації конфігураційних файлів kubernetes	Нм	Тх
	надає можливість генерації частини коду мікросервісів	Нм	Тх
	надає можливість генерації документації та тестів для мікросервісів	Нм	Тх
	має достатньо високу швидкодію при використанні	М	Тл
	є зручним у використанні	Нм	Е
	сумісний з Windows і Linux	Нм	Тл
	відсутність критичних проблем, які не дозволяють надійно працювати у програмі	М	Тл
	Якість: тестування на основі розроблених функціональних і нефункціональних вимог для програмного забезпечення		
	Пакування: веб-застосунок, exe файл для Windows, deb пакет для Linux		
	Марка: Microservice App Generator		
Товар із підкріпленням	До продажу: доступ до безкоштовних функцій товару, допоміжні та навчальні матеріали		
	Після продажу: технічна підтримка		

Програмне забезпечення має значно нижчі вимоги до системи збуту у порівнянні з фізичними товарами. Інтернет дозволяє легко надавати користувачам програмне забезпечення напряму без посередників. Таким чином основним каналом збуту є створений веб-додаток, через який можна розповсюджувати exe файли для Windows і deb пакети.

Розробимо концепцію маркетингових комунікацій (Таблиця 4.17).

Таблиця 4.17

Концепція маркетингових комунікацій

Специфіка поведінки цільових клієнтів	Клієнти часто готові самі шукати програмне забезпечення, яке задовольняє їх вимоги
Канали комунікацій, якими користуються цільові клієнти	Інтернет, месенджери (Whatsapp, Viber, Telegram), соціальні мережі (Facebook, LinkedIn, X)
Ключові позиції, обрані для позиціонування	функціональність ПЗ: можливість генерації конфігураційних файлів Kubernetes та коду мікросервісів з опису застосунку, який створюється за допомогою візуального редагування
Завдання рекламного повідомлення	донести до користувача інформацію про розроблене програмне забезпечення, його функції та потреби, які воно може задовольнити
Концепція рекламного звернення	акцентування уваги на функціональності застосунку

Висновки до розділу 4

На основі створеного програмного забезпечення було розроблено стартап-проект. В рамках розробки стартап проекту було виконано такі етапи:

- розробка ідеї;
- технологічний аудит проекту;
- аналіз ринкових можливостей проекту;
- розробка ринкової стратегії проекту;
- розробка маркетингової стратегії проекту.

В рамках аналізу ринкових можливостей проекту було виділено три головних конкуренти: JHipster, k8yaml і Helm, проаналізовано їх сильні і слабкі сторони та загальні властивості конкуренції на ринку. За результатами аналізу ринок є привабливим для входу.

Було виділено потенційні групи цільових користувачів:

- розробники компаній, які планують перейти на мікросервісну архітектуру;
- розробники стартапів;
- розробники компаній, які вже використовують мікросервісну архітектуру.

В рамках аналізу було визначено, що кожна з цих груп має достатньо високий рівень готовності користувачів сприйняти продукт та орієнтовний попит, а також низьку інтенсивність конкуренції та складність входу в сегменті. Тому було вирішено використовувати масовий маркетинг.

В якості базової стратегія розвитку було обрано стратегію диференціації, яка полягає фокусі на надання товару особливих властивостей, які виділяють його на фоні конкурентів.

В якості стратегії конкурентної поведінки було обрано стратегію розширення первинного рівня попиту за допомогою навчання споживачів користуванням товаром, формуванні регулярного попиту тощо.

В якості основного каналу збуту було обрано створений веб-додаток, через який можна розповсюджувати exe файли для Windows і deb пакети. Це дозволяє уникнути посередників і також не потребує додаткових витрат.

ВИСНОВКИ

Були розглянуті і проаналізовані патерни та типові інфраструктурні елементи, які використовуються у мікросервісній архітектурі. Були досліджені основні існуючі інструменти які використовують генерацію коду. На основі огляду і аналізу предметної області сформульовані функціональні і нефункціональні вимоги до розробленого програмного забезпечення.

Практично реалізовано програмне забезпечення для автоматизованої генерації конфігураційних файлів для розгортання і налаштування інфраструктури мікросервісних застосунків, а саме сервісів, баз даних, API-шлюзів і брокерів повідомлень, а також частини коду, документації та автоматичних тестів власне мікросервісів.

Розроблене програмне забезпечення включає веб-додаток, що складається з клієнтської та серверної частин, а також консольний додаток. Усі компоненти реалізовані мовою Typescript, а для розробки використовувалась IDE Webstorm. Веб-додаток надає графічний інтерфейс для опису архітектури застосунку, а консольний додаток дозволяє згенерувати частини застосунку на основі цього опису.

Для перевірки ефективності розробленого програмного інструменту у порівнянні з аналогами було проведено контрольований експеримент. В результаті експерименту було виявлено статистично значущу перевагу у швидкості розробки.

На основі спроектованого програмного забезпечення розроблено стартап проект. Сформульовано ідею проекту, проведено технічний аудит, аналіз ринкових можливостей запуску проекту, розроблення ринкової стратегії та маркетингової програми проекту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

- 1) Zhang H., Li S., Jia Z., Zhong C., & Zhang C. "Microservice architecture in reality: An industrial inquiry." 2019 IEEE international conference on software architecture (ICSA). IEEE, 2019.
- 2) Richards, M. Fundamentals of Software Architecture. An Engineering Approach. Sebastopol, 2020
- 3) Brown S. Software Architecture for Developers. Vol 1. Victoria, 2018
- 4) Richardson C. Microservices Patterns. New York, 2019
- 5) Surianarayanan C., Ganapathy G., Raj P. Essentials of Microservices. Architecture Paradigms, Applications, and Techniques. Boca Raton, 2020
- 6) Wolff E. Microservices. Flexible Software Architecture. Crawfordsville, 2016
- 7) Nadareishvili I., Mitra R., McLarty M., Amundsen M. Microservice Architecture. Aligning Principles, Practices and Culture. Sebastopol, 2016
- 8) Christudas B. Practical Microservices Architectural Patterns. Trivandrum, Kerala, 2019
- 9) Vohra D. Kubernetes Microservices with Docker. White Rock, 2016
- 10) Офіційна документація Kubernetes [Електронний ресурс]. Режим доступу до ресурсу: <https://kubernetes.io/docs/home/>
- 11) Офіційна документація Angular [Електронний ресурс]. Режим доступу до ресурсу: <https://angular.io/docs>
- 12) Офіційна документація Vue [Електронний ресурс]. Режим доступу до ресурсу: <https://devdocs.io/vue~1-guide/>
- 13) Офіційна документація React [Електронний ресурс]. Режим доступу до ресурсу: <https://reactjs.org/docs/getting-started.html>
- 14) Офіційна документація NestJs [Електронний ресурс]. Режим доступу до ресурсу: <https://docs.nestjs.com/>
- 15) Офіційна документація GitHub Actions [Електронний ресурс]. Режим доступу до ресурсу: <https://docs.github.com/en/actions>

- 16) Kitchenham, B. A. "Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods." *ACM SIGSOFT Software Engineering Notes* 21.1, 1996, 11-14.
- 17) Офіційна документація Helm [Електронний ресурс]. Режим доступу до ресурсу: <https://helm.sh/docs/>
- 18) Офіційна документація семантичного версіонування [Електронний ресурс]. Режим доступу до ресурсу: <https://semver.org/lang/uk/>
- 19) Lam, P., Dietrich, J., & Pearce, D. J. (2020). *Putting the semantics into semantic versioning. Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. doi:10.1145/3426428.3426922
- 20) Інструмент k8yaml [Електронний ресурс]. Режим доступу до ресурсу: <https://k8syaml.com/>
- 21) Інструмент gimlet [Електронний ресурс]. Режим доступу до ресурсу: <https://gimlet.io/k8s-yaml-generator>
- 22) Інструмент Kubernetes Yaml Generator [Електронний ресурс]. Режим доступу до ресурсу: <https://syshunt.com/kubernetes-yaml-generator/>
- 23) Neideen, T., & Brasel, K. (2007). Understanding Statistical Tests. *Journal of Surgical Education*, 64(2), 93–96.
- 24) Nachar, N. (2008). The Mann-Whitney U: A Test for Assessing Whether Two Independent Samples Come from the Same Distribution. *Tutorials in Quantitative Methods for Psychology*, 4(1), 13-20.
- 25) Ko, A. J., LaToza, T. D., & Burnett, M. M. A practical guide to controlled experiments of software engineering tools with human participants. *Empirical Software Engineering*, 20, 2015, 110-141.
- 26) Dybå, T., Kampenes, V. B., & Sjøberg, D. I. K. (2006). A systematic review of statistical power in software engineering experiments. *Information and Software Technology*, 48(8), 745–755.
- 27) Lieber, R. L. (1990). Statistical significance and statistical power in hypothesis testing. *Journal of Orthopaedic Research*

- 28) Cohen, J., 2013. Statistical power analysis for the behavioral sciences. Academic press.
- 29) Owen, D. B. (1962). Handbook of statistical tables. (No Title).
- 30) Dybå, T., & Dingsøyr, T. (2008). Strength of evidence in systematic reviews in software engineering. Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement - ESEM '08. doi:10.1145/1414004.1414034
- 31) McDonald, J.H., University of Delaware. 2009. Handbook of Biological Statistics.
- 32) Das, K.R. and Imon, A.H.M.R., 2016. A brief review of tests for normality. American Journal of Theoretical and Applied Statistics, 5(1), pp.5-12.
- 33) Статистичний калькулятор. URL: <https://www.statskingdom.com/shapiro-wilk-test-calculator.html>
- 34) Cressie, N. A. C., & Whitford, H. J. (1986). How to Use the Two Samplet-Test. Biometrical Journal, 28(2), 131–148.
- 35) Insights on the Microservices Architecture Global Market to 2027 [Електронний ресурс]. Режим доступу до ресурсу: <https://www.businesswire.com/news/home/20221101006158/en/Insights-on-the-Microservices-Architecture-Global-Market-to-2027---Increasing-Need-for-Digital-Transformation-to-Meet-Business-Requirements-is-Driving-Growth---ResearchAndMarkets.com>
- 36) Di Francesco, P., Lago, P., & Malavolta, I. (2018). *Migrating Towards Microservice Architectures: An Industrial Survey. 2018 IEEE International Conference on Software Architecture (ICSA)*. doi:10.1109/icsa.2018.00012
- 37) Porter, M. (1985). On competition. Updated and Expanded Edition. Boston, MA: Harvard Business School Publishing
- 38) Sammut-Bonnici T., and Galea D. "SWOT analysis." Wiley Encyclopedia of management (2015): 1-8.
- 39) Розроблення стартап-проекту [Електронний ресурс]: Методичні рекомендації до виконання розділу магістерських дисертацій для

студентів інженерних спеціальностей / За заг. ред. О.А. Гавриша. –
Київ : НТУУ «КПІ», 2016. – 28 с.