ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: **«МІКРОСЕРВІСНА АРХІТЕКТУРА ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ ВИСОКОНАВАНТАЖЕНИХ СИСТЕМ»**

на здобуття освітнього ступеня магістр

зі спеціальності: 126 Інформаційні системи та технології
(*код, найменування спеціальності*)

освітньо-професійної програми: Інформаційні системи та технології
(*назва*)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Віктор ШУШУРА
(*підпис*) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. ІСДМ–63

Віктор ШУШУРА
Ім'я, ПРІЗВИЩЕ

Керівник: Каміла СТОРЧАК
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ
д.т.н, професор

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інформаційні системи та технології

Ступінь вищої освіти Магістр

Спеціальність 126 Інформаційні системи та технології

Освітньо-професійна програма Інформаційні системи та технологій

ЗАТВЕРДЖУЮ
Завідувач кафедрою
Інформаційних систем та технологій
_____ Каміла Сторчак
"____"_____ 2024 року

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Шушурі Віктору Олексійовичу
(*прізвище, ім'я, по батькові здобувача*)

1. Тема роботи: «Мікросервісна архітектура підвищення продуктивності високонавантажених систем»
Керівник кваліфікаційної роботи Сторчак Каміла Павлівна, доктор технічних наук, професор
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «15» жовтня 2024 р. № 320
2. Строк подання кваліфікаційної роботи «27» грудня 2024 р.
3. Вихідні дані до кваліфікаційної роботи:
методи розробки мікросервісної архітектури;
система хмарних обчислень;
система планування ресурсів підприємства.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).
1. Порівняння монолітної та мікросервісної архітектури.
2. Система планування ресурсів підприємства: переваги, недоліки, причини

впровадження.

3. Використання мікросервісів для налаштування мультитенантного SaaS.

5. Перелік ілюстративного матеріалу: *презентації*

1. Порівняння монолітної та мікросервісної архітектур.
2. Переваги мікросервісів.
3. ERP-система.
4. Функції ERP систем.
5. 3-рівнева архітектура.
6. Концептуальна модель налаштування продукту для багатокористувацького SaaS.
7. Адаптація SaaS до можливості налаштування.
8. Архітектура моделі програмного забезпечення додатку процесу адаптації послуги під конкретні потреби певної особи або групи осіб.
9. Еталонна архітектура використання мікросервісів для налаштування.
10. Висновки, апробації.

6. Дата видачі завдання: «15» жовтня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Опрацювання літературних джерел	15.10 – 25.10.24	Виконано
2	Порівняння монолітної та мікросервісної архітектур	26.10 – 10.11.24	Виконано
3	Впровадження ERP-систем, їхні недоліки та можливості використання мікросервісів для вирішення проблем впровадження	11.11 – 20.11.24	Виконано
4	Створення персоналізованого середовища багатокористувацького SaaS з використанням мікросервісів.	21.11 – 09.12.24	Виконано
5	Оформлення роботи вступ, висновки, реферат	10.12 – 14.12.24	Виконано
6	Розробка демонстраційних матеріалів	15.12 – 20.12.24	Виконано
7	Попередній захист роботи	25.12.24	Виконано

Здобувач вищої освіти _____ Віктор ШУШУРА
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Каміла СТОРЧАК
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 86 с., 2 табл., 21 рис., 30 джерел.

Мета роботи: розробити теоретичні основи та практичні рекомендації щодо ефективного використання мікросервісної архітектури для створення гнучких, масштабованих та персоналізованих ERP-систем в моделі SaaS.

Об`єкт дослідження – процес управління корпоративними знаннями, зокрема способи їх збору, зберігання, аналізу та використання для підтримки бізнес-процесів.

Предмет дослідження – мікросервісна архітектура, зокрема інтеграція штучного інтелекту в корпоративні бази знань, розробка чат-бота та його ефективність у вирішенні типових запитань співробітників.

Короткий зміст роботи: В роботі порівнюються переваги та недоліки монолітної та мікросервісної архітектури, розглядаються критерії вибору архітектури та методи міграції від монолітної до мікросервісної архітектури.

Аналізуються причини впровадження ERP-систем, їхні недоліки та можливості використання мікросервісів для вирішення проблем впровадження.

Досліджується застосування мікросервісів в хмарних обчисленнях, розглядаються концептуальні моделі налаштування SaaS-систем, а також механізми динамічного та децентралізованого налаштування.

Була створена узагальнена еталонна архітектура для забезпечення створення унікального середовища багатокористувацьких SaaS за допомогою мікросервісів.

КЛЮЧОВІ СЛОВА: МІКРОСЕРВІСНА АРХІТЕКТУРА, МОНОЛІТНА АРХІТЕКТУРА, ХМАРНІ ОБЧИСЛЕННЯ, ERP-СИСТЕМА, SAAS, КОРИСТУВАЧІ, КЛІЄНТИ, НАЛАШТУВАННЯ, ЕТАЛОННА АРХІТЕКТУРА

ABSTRACT

Text part of the master's qualification work:
86 pages, 21 pictures, 2 tables , 30 sources.

The purpose of the work: to develop theoretical foundations and practical recommendations for the effective use of microservice architecture to create flexible, scalable, and personalized ERP systems in the SaaS model.

The object of research is the process of corporate knowledge management, in particular, methods of its collection, storage, analysis and use to support business processes.

The subject of research is microservice architecture, in particular the integration of artificial intelligence into corporate knowledge bases, the development of a chatbot and its effectiveness in solving typical employee questions.

Summary of the work: The paper compares the advantages and disadvantages of monolithic and microservice architectures, considers the criteria for choosing an architecture and migration methods from monolithic to microservice architecture.

The reasons for implementing ERP systems, their shortcomings and the possibilities of using microservices to solve implementation problems are analyzed.

The use of microservices in cloud computing is studied, conceptual models of SaaS system configuration are considered, as well as mechanisms for dynamic and decentralized configuration.

A generalized reference architecture was created to ensure the creation of a unique multi-user SaaS environment using microservices.

KEYWORDS: MICROSERVICE ARCHITECTURE, MONOLITHIC ARCHITECTURE, CLOUD COMPUTING, ERP SYSTEM, SAAS, USERS, CUSTOMERS, CONFIGURATION, REFERENCE ARCHITECTURE

ЗМІСТ

ВСТУП

Актуальність теми. Сучасні бізнес-процеси вимагають гнучких, масштабованих та надійних IT-рішень. Традиційні монолітні системи все частіше не можуть задовольнити ці вимоги через обмежені можливості масштабування, складності впровадження змін та високу вартість підтримки. Мікросервісна архітектура пропонує альтернативний підхід, який дозволяє розробляти більш гнучкі та масштабовані системи.

У сучасну еру цифрової трансформації існує термінова потреба в гнучких, ефективних і масштабованих інформаційних системах у корпоративному світі, при цьому системи планування ресурсів підприємства (ERP) відіграють критичну роль в інтеграції та оптимізації бізнес-процесів. Однак зростаюча складність і вимога до гнучкості підкреслюють обмеження традиційних архітектур ERP.

Ці платформи часто покладаються на застарілу технологію, побудовані на монолітних архітектурах і дотримуються архаїчних моделей ціноутворення.

Багато таких систем стикаються з проблемами сумісності на сучасних операційних системах, страждають від застарілих графічних інтерфейсів і не є зручними для користувачів.

У контексті цифрової трансформації необхідність у гнучких, ефективних і масштабованих корпоративних інформаційних системах є надзвичайно важливою, при цьому системи планування ресурсів підприємства (Enterprise Resource Planning - ERP) знаходяться на передньому плані інтеграції та оптимізації бізнес-процесів. Зіткнувшись із зростаючою складністю та вимогами до гнучкості, традиційні архітектури ERP демонструють значні обмеження.

Мета роботи - розробити теоретичні основи та практичні рекомендації щодо ефективного використання мікросервісної архітектури для створення гнучких, масштабованих та персоналізованих ERP-систем в моделі SaaS.

Широкий аналіз та реальний випадок застосування формують ядро цього дослідження, зосереджуючи увагу на проектуванні та розробці програмної платформи ERP, адаптованої для малих і середніх підприємств (МСП).

Ця платформа не лише відповідає специфічним вимогам МСП, але й охоплює сучасні концепції розподіленої архітектури та нові парадигми програмування на основі хмари. Мета полягає в оптимізації економічної ефективності для кінцевих користувачів, навіть з моделлю ціноутворення, що відрізняється від існуючих платформ, дослідження завершується створенням прототипу системи ERP.

Об`єкт дослідження – процес управління корпоративними знаннями, зокрема способи їх збору, зберігання, аналізу та використання для підтримки бізнес-процесів.

Предмет дослідження – мікросервісна архітектура, зокрема інтеграція штучного інтелекту в корпоративні бази знань, розробка чат-бота та його ефективність у вирішенні типових запитань співробітників.

Наукова новизна роботи полягає в розробці нових підходів, методів та інструментів, які дозволяють ефективніше вирішувати проблеми, пов'язані з інтеграцією мікросервісів в традиційні ERP-системи, забезпеченням персоналізації, підвищенням продуктивності та безпеки.

Практичне значення: Використання мікросервісів для налаштування клієнтів SaaS в ERP-системах є перспективним напрямком розвитку корпоративних інформаційних систем. Цей підхід дозволяє підвищити гнучкість, масштабованість, надійність і ефективність бізнес-процесів, а також знизити витрати на розробку і підтримку системи.

Апробація результатів та публікації. Шушура В.О. Тестування продуктивності мікросервісів у хмарному середовищі. ІІ Міжнародна науково-практична конференція «Сучасні аспекти діджиталізації та інформатизації в програмній та комп'ютерній інженерії». Збірник тез. – К.: ДУІКТ, 2024.

1 МОНОЛІТНА VS МІКРОСЕРВІСНА АРХІТЕКТУРА

1.1 Монолітна архітектура

Традиційно додатки мають монолітну архітектуру, що означає, що всі функції та всі необхідні компоненти обробляються як єдиний прикладний блок. Більшість великих і успішних програм назвали монолітними, але монолітна архітектура має кілька недоліків. Основними проблемними моментами є складність і розмір таких програм.

Монолітна архітектура - це традиційний метод розробки програмного забезпечення, коли всі функції інкапсульовані в одному додатку. Монолітне програмне забезпечення розробляється для того, щоб бути самодостатнім. Цей тип архітектури є тісно пов'язаним, що означає, що якщо один з компонентів відсутній, то він не буде виконаний або скомпільований.

Монолітна архітектура має переваги та недоліки. До переваг монолітної архітектури можна віднести наступні:

- менше наскрізних проблем - простіше підключати компоненти до наскрізних проблем, коли все простіше підключати компоненти до наскрізних проблем, коли все працює через один додаток;

- менше операційних накладних витрат - потрібно налаштувати лише один додаток, і менш складне розгортання - потрібно розгорнути лише один додаток.

Недоліки монолітної архітектури наступні:

- пов'язана - в неї особливо важко вносити зміни, коли моноліт стає дуже складним;

- безперервне розгортання - весь додаток повинен розгортатися при кожному оновленні;

- масштабованість - важко масштабувати, коли різні модулі мають суперечливі вимоги до ресурсів вимоги до ресурсів;

і надійність - помилка в будь-якому компоненті може потенційно вивести з ладу весь додаток.

Розладнати монолітну програму складно, і внесення невеликих змін до окремого аспекту програми може спричинити проблеми регресії у всій програмі [25]. Регресійне тестування має охоплювати всі критичні функції. Ці проблеми призводять до повільних циклів розвитку [26].

Щоб подолати проблеми монолітної структури, ідеї полягають у тому, щоб розділити програму на менші, але взаємопов'язані служби, де кожна служба відповідає за свою власну функціональність, але спілкується з іншими службами через інтерфейси прикладного програмування (API) [25].

Цей підхід вирішує основну проблему складності великої програми, розбиваючи програму на набір керованих сервісів, які набагато швидше розробляти та набагато легше підтримувати [27].

Крім того, це дозволило розділити розробку на менші команди, де кожна команда відповідає за розробку та підтримку власного сервісу. Наявність невеликої різноманітної команди та невеликого додатка додає ще більше переваг, оскільки кожна служба може бути розроблена за власною технологією, а компоненти слабо зв'язані. Тому сервіс можна розгортати, масштабувати та тестувати незалежно [28].

Зазвичай застарілі додатки зростають у розмірі та складності, що призводить до того, що через кілька років розробки вони перетворюються на жахливе монолітне програмне забезпечення, а недоліки монолітної архітектури переважують її переваги.

Виправлення помилок і додавання нових функцій до таких додатків є складною і трудомісткою операцією. Масштабування зазвичай неможливе або вимагає багато роботи.

За таких обставин організації починають шукати нове архітектурне рішення. Мікросервісна архітектура стає стандартом за замовчуванням у більшості підприємств, оскільки за останні кілька років було реалізовано багато проектів з використанням цієї архітектури, і результати виявилися дуже позитивними. Такі провідні компанії, як Amazon, eBay, Netflix, PayPal, Twitter та інші успішно перейшли на мікросервісну архітектуру.

Мікросервісний архітектурний стиль - це підхід до розробки єдиного додатку як набору невеликих сервісів, кожен з яких працює у своєму процесі і взаємодіють за допомогою легких механізмів, найчастіше HTTP API ресурсів. Ці сервіси побудовані навколо бізнес-можливостей і можуть бути розгорнуті незалежно за допомогою повністю автоматизованих механізмів розгортання. Існує мінімальне централізоване управління цими сервісами, які можуть бути написані різними мовами програмування і використовувати різні технології зберігання даних.

Основними трьома принципами мікросервісної архітектури є:

- Мікросервіс має єдину відповідальність: подібно до принципу єдиної відповідальності з принципів SOLID, де кожен клас повинен мати лише одну відповідальність. Кілька мікросервісів не повинні мати однакову відповідальність, і жоден з окремих мікросервісів не повинен мати більше ніж однієї відповідальності. Кожен мікросервіс повинен надавати повну бізнес-можливість як єдине ціле. Іншими словами, мікросервіси повинні виконувати лише одну функцію.

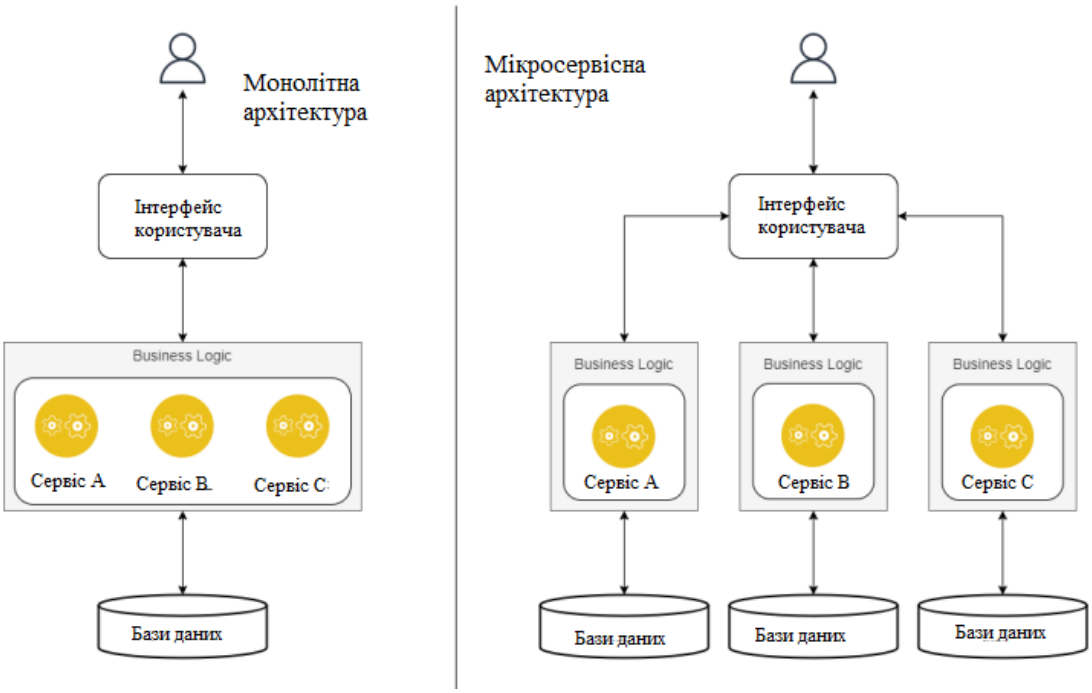

Рис. 1.1. Порівняння монолітної та мікросервісної архітектур

Таблиця 1.1

Порівняння монолітної та мікросервісної архітектур

Характеристика	Монолітна архітектура	Мікросервісна архітектура
Структура	Єдиний блок коду	Розподілені сервіси
База даних	Спільна база даних	Окремі бази даних
Розробка	Складна, довга	Швидка, гнучка
Масштабування	Складне	Легке, незалежне
Стійкість	Низька	Висока

- Мікросервіс є автономним: це самодостатній сервіс, який можна розгортати незалежно. Через свою автономність, він повинен містити всі залежності, такі як бібліотеки та середовища виконання - веб-сервери, контейнери, віртуальні машини тощо.

- Мікросервіс є поліглотом: він розкриває свої кінцеві точки у вигляді API та абстракцій всі деталі своєї реалізації, такі як логіка реалізації, архітектура, технології тощо.

Однією з основних причин, чому мікросервісна архітектура вважається кращим варіантом, ніж монолітна архітектура, є декомпозиція складних додатків на менші компоненти, які легше розробляти, керувати та підтримувати, ніж єдиний монолітний додаток. Поділ додатків на окремі, незалежні мікросервіси дозволяє окремим командам керувати ними в межах організації з розробки програмного забезпечення та працювати незалежно. Оскільки мікросервіси є автономними і взаємодіють через відкриті протоколи, вони можуть розроблятися незалежно з використанням різних технологій і мов програмування. Зазвичай команди, що розробляють мікросервіси, організовуються навколо бізнесу, а не технічних можливостей.

Кожна нова вимога повинна задовольнятися лише одним мікросервісом, щоб зберегти незалежний розвиток. Незалежність та автономія дозволяють масштабувати мікросервіси горизонтально, технічно і в межах організації, оскільки команди можуть бути меншими і більш гнучкішими. Отже, архітектура

мікросервісів покращує технічні аспекти і підвищує гнучкість бізнесу та можливість швидше надавати нові функції.

Інші варті згадуючі вигоди від мікросервісної архітектури були порівнянні з монолітною архітектурою - мікросервіс може розгортатися самостійно, і немає ніякої необхідності знову почати ціле застосування.

Інші переваги мікросервісної архітектури порівняно з монолітною архітектурою:

− Можливість розгортання: немає необхідності перезапускати всю програму. Можливість виявлення критичної функціональності бізнесу дозволяє розгортати відповідні мікросервіси в більш надлишковому середовищі.

− Надійність: несправність мікросервісу впливає на сам мікросервіс, не обов'язково на всю програму. Слабко пов'язана архітектура робить мікросервіси більш відмовостійкими.

- Хмарність: характеристики розгортання роблять мікросервіси чудовими для еластичності хмари. Мікросервіси — це хмарні програми. Оскільки мікросервіси є незалежними процесами, кожен із них можна розгорнути в окремому контейнері чи віртуальній машині в хмарі. Мікросервіси можна оновлювати та масштабувати окремо. Масштабованість може контролюватися вимогами до навантаження на вимогу. Такий підхід забезпечує більш детальну еластичність застосування. Такі рішення, як контейнери Docker або Rocket, разом із інструментами оркестровки Docker Swarm, Mesos або Kubernetes дозволяють використовувати архітектуру мікросервісів як архітектуру для програм, готових до роботи в хмарі.

− Модифікованість: кожен мікросервіс інкапсульований; отже, більша гнучкість у використанні нових фреймворків, бібліотек, джерел даних та інших ресурсів. Управління розробкою додатків на основі мікросервісів розподілено між невеликими командами, які працюють більш незалежно. Архітектура мікросервісу дозволяє досягти кращого узгодження розробників із бізнес-користувачами, оскільки архітектура мікросервісу організована навколо бізнес-

можливостей, і розробники можуть легко зрозуміти точку зору користувача та створювати мікросервіси, які краще узгоджуються з потребами бізнесу.

Архітектура мікросервісів не є панацеєю і має недоліки. Складність є її найбільшим недоліком порівняно з монолітною архітектурою. Архітектура мікросервісів ускладнює проєкт, будучи розподіленою системою. Розгортання, масштабування та моніторинг є більш складними завданнями в архітектурі мікросервісів, ніж у монолітній архітектурі.

І монолітна, і мікросервісна архітектури мають свої переваги та недоліки, і вибір між ними залежить від різних факторів, таких як розмір проєкту, досвід команди, складність системи, бажана масштабованість тощо. Це необхідно для розробників та архітекторам, щоб уважно оцінити, чи є розкладання існуючого моноліту правильним шляхом і чи є самі мікросервіси правильним призначенням.

Монолітна архітектура краще підходить для простого легкого застосування. Архітектурне рішення мікросервісу є кращим вибором для складних додатків, що розвиваються. Монолітні програми слід модернізувати до мікросервісної архітектури, коли:

− Монолітна програма стає занадто великою та складною для підтримки чи розширення. Виконувати щоденні операції з технічного обслуговування, додавати нові функції чи виправляти існуючі проблеми стає дуже дорого як з точки зору ресурсів, так і часу.

− Важливими аспектами є модульність і децентралізація. Архітектура мікросервісу дозволяє працювати над кожним мікросервісом окремо. Такі проблеми, як масштабованість, можна застосувати лише до конкретного мікросервісу, а не до всієї програми.

− Перевага щодо отримання довгострокових вигод порівняно з короткостроковими.

Середовище, яке підтримує мікросервіси, принципово потребує набору базових вимог, щоб забезпечити певний рівень розумності. Організація повинна бути готова взяти на себе накладні витрати на їх створення та підтримку. Накладні витрати не будуть незначними. Якісно виконані мікросервіси

потребують часу та грошей. Кожна організація повинна мати внутрішню групу, відповідальну за інфраструктуру, розробку та експлуатацію для використання мікросервісів. Ця група повинна складатися з найкращих розробників організації або навіть зовнішніх консультантів.

Немає єдиного правила для налаштування інфраструктури для мікросервісів. Кожна область нижче описує функціональні можливості, які мають бути реалізовані в інфраструктурі.

Безперервна інтеграція/безперервна доставка: організація повинна вирішити, як створювати, тестувати та розгортати мікросервіси. Ці операції мають бути автоматичними. Сьогодні багато систем побудови забезпечують конвеєрну функціональність. Група, відповідальна за мікросервісну інфраструктуру, повинна визначитися зі стратегією як це зробити і вибрати інструменти для цього:

− Контроль джерела: як слід зберігати та підтримувати вихідний код.

− Інструмент побудови: як слід побудувати мікросервіс.

− Інструмент тестів: як мають бути реалізовані тести.

− Інструмент розгортання: як слід розгортати мікросервіс.

Віртуальні машини/контейнери та хмара: ще одним важливим рішенням є те, яку технологію використовувати для середовища виконання. Багато підприємств із наявними програмами, що працюють на стабільній інфраструктурі віртуальної машини, вибирають підхід «пальцем у воду». Розгортаючи контейнери на віртуальних машинах, вони отримують переваги зрілого моніторингу та ізоляції з більш швидкими процесами DevOps. Порівняно з контейнерами, що працюють на голому металі, вони втрачають деяку продуктивність, масштабованість і вартість. Але це, безумовно, дійсний шлях до переходу. Архітектура мікросервісу є природним додатком для хмарних програм. А хмарна програма визначається як програма, створена з нуля для архітектури хмарних обчислень. Це означає, що додаток є хмарним, якщо він розроблений так, ніби його розгортають у розподіленій і масштабованій інфраструктурі.

Моніторинг є важливою частиною інфраструктури мікросервісів. Організації повинні дотримуватись п’яти принципів для встановлення

ефективнішого моніторингу, які наведено нижче. Ці принципи дозволять організаціям вирішувати як технологічні зміни, пов’язані з мікросервісами, так і організаційні зміни пов’язані з ними:

− Контролювати контейнери та їх вміст.

− Попередження про якість обслуговування, але не про продуктивність контейнера.

− Моніторинг послуг, які є еластичними та мають багато місцеположень.

− API моніторингу.

− Віднести моніторинг до організаційної структури.

Ведення журналу відіграє важливу роль у підтримці програми. Щоб зробити це ефективно для мікросервісів служба журналювання повинна бути централізованою та мати потужний візуалізатор. Кращі практики для мікросервісів журналювання наведено нижче.

− Співвідносьте запити з унікальним ідентифікатором.

− Додайте унікальний ідентифікатор у відповіді.

− Структуруйте свої дані журналу.

− Додайте контекст до кожного запиту.

− Запис журналів у локальне сховище.

− Реєструйте корисні та значущі дані.

1.2 Критерії дослідження методів міграції від застарілого монолітного програмного забезпечення в архітектуру мікросервісу

У цьому розділі розглянемо інформацію про критерії різних методів міграції від застарілого монолітного програмного забезпечення до архітектури мікросервісу. Оскільки кожна застаріла монолітна програма може відрізнятися багатьма аспектами, було введено список критеріїв для порівняння методів міграції з різних точок зору:

− Кількість мікросервісів-кандидатів: щоб оцінити потенційну кількість мікросервісів, виявлених під час переходу на архітектуру мікросервісів у застарілих монолітних програмах.

− Розмір мікросервісу: щоб оцінити потенційний розмір мікросервісу, вилученого із застарілої монолітної програми.

− База даних: щоб оцінити, чи підтримує метод міграції адаптацію монолітної бази даних до архітектури мікросервісу.

− Підключення мікросервісів: щоб оцінити, чи метод міграції підтримує встановлення зв’язку між мікросервісами, що розкладаються.

− Автоматизація: оцінити можливість повної автоматизації методу міграції.

− Технологічний стек: для оцінки технологічного стеку, який використовується під час переходу від монолітної архітектури до архітектури мікросервісу.

− Реалізація та інструменти: для оцінки деталей реалізації та інструментів, що використовуються під час переходу від монолітної архітектури до архітектури мікросервісу.

− Якість коду: оцінити вплив якості коду на перехід від монолітної архітектури до архітектури мікросервісу.

Оцінка вилучення на основі сховища

В технічній літературі описують методику ідентифікації мікросервісів у монолітних системах. Основна ідея методики полягає в тому, що декомпозиція повинна проводитися на основі унікальних таблиць бази даних і пар фасадів. Кожну унікальну пару можна вважати кандидатом на мікросервіс. Усі бізнес-функції, які використовуються парою фасаду та таблиці бази даних, повинні бути включені в мікросервіс. У процесі розкладання, фасади, бізнес функції та таблиці бази даних повинні бути ідентифіковані, а також мають бути знайдені унікальні пари. Пропонована методика складається з наступних чотирьох кроків.

Декомпозиція бази даних

Першим кроком є відображення таблиць бази даних у підсистеми. Кожна підсистема представляє бізнес-сферу організації. Таблиці, не пов'язані з бізнес-

процесом, називаються підсистемою керування. На рис. 1.2 представлена частина декомпозиції бази даних, виконана в додатку DataProvider.

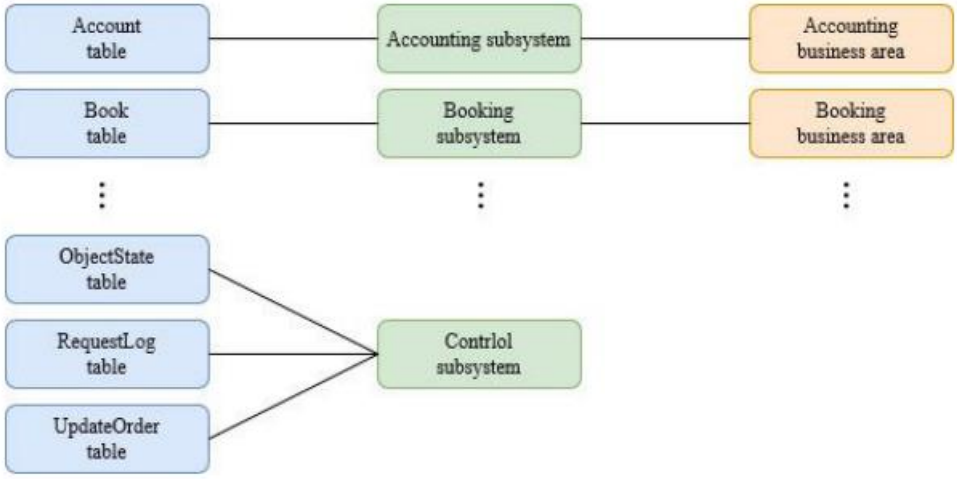

Рис. 1.2. Декомпозиція бази даних

Програма DataProvider містить 15 таблиць бази даних, дев'ять підсистем і вісім різних бізнес-сфер. Цей крок методології дозволяє ідентифікувати ряд таблиць і бізнес-сфер. Ідентифікація таблиць бази даних — завдання, яке потребує лише технічних навичок. З іншого боку, ідентифікація бізнесу підсистеми та призначення таблиці для них вимагають додаткових зусиль для розуміння бізнес-процесу.

Граф залежностей

На другому кроці було створено графік залежностей між фасадами, бізнес-функціями та таблицями бази даних. Він показує бізнес-функціональність і залежності бази даних. Рис. 1.3 ілюструє деякі графіки програми DataProvider.

П'ять графіків були досить простими: містили лише одну таблицю бази даних, один рівень бізнес-функціональності та 0 залежностей від інших таблиць бази даних і підсистем бізнес-функціональності. Інші 12 таблиць бази даних були об'єднані в один більш складний граф залежностей. Деякі бізнес-функції містять до чотирьох залежностей від інших таблиць бази даних. Здебільшого було визначено чотири рівні бізнес-функціональності для повноцінної роботи від фасаду до таблиці бази даних.

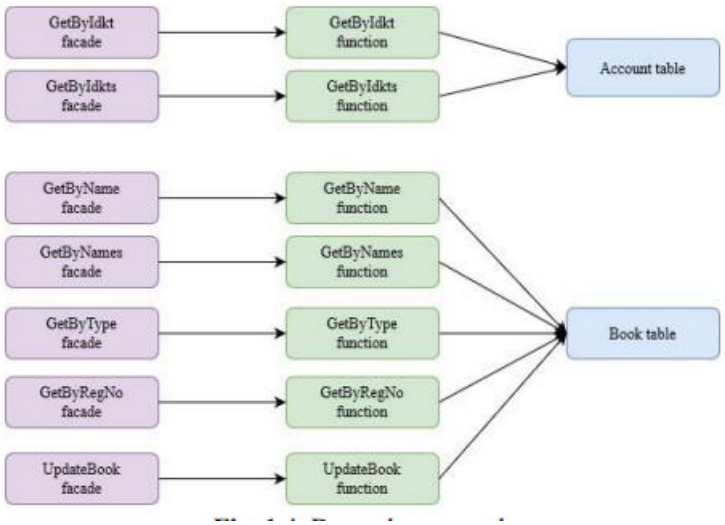

Рис. 1.3. Графік залежності

Таблиці бази даних і пари фасадів

На основі графа залежностей було ідентифіковано унікальні пари фасадів і таблиць бази даних і зіставлено з функціями бізнес-підсистеми. На рис. 1.4 представлено дві унікальні пари в додатку DataProvider.

Рис. 1.4. Дві унікальні пари в додатку DataProvider

Програма DataProvider має 68 унікальних пар таблиць бази даних і фасадів. П'ятнадцять фасадів були в парі лише з однією таблицею бази даних. Деякі з фасадів були в парі з різними таблицями бази даних до восьми разів. У більш складному графі залежностей існує більше унікальних пар з однаковим фасадом.

Кандидати в мікросервіс

На останньому кроці були визначені кандидати для перетворення на мікросервіси. Для кожної окремої пари, отриманої на попередньому кроці, перевірка була зосереджена на коді фасаду та бізнес-функцій, які знаходяться на

шляху від фасаду до таблиці бази даних у графі залежностей. На рис. 1.5 показано два кандидати для перетворення в мікросервіси програми DataProvider.

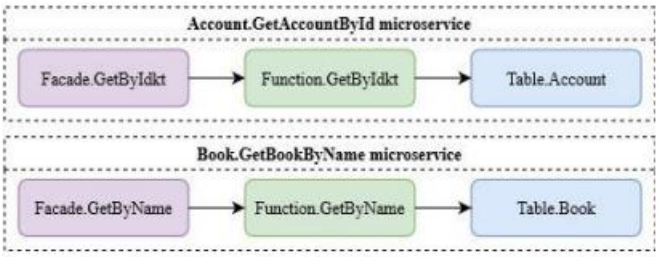

Рис. 1.5. Кандидати в мікросервіси

Декомпозиція за допомогою методу на основі сховища призвела до 37 мікросервісів-кандидатів, знайдених у програмі DataProvider.

Більше функцій і табличних підсистем мали більше мікросервісів-кандидатів. Можливо, розмір мікросервісу-кандидата може бути дуже малим, якщо він містить лише одну бізнес-функцію. Метод вимагає ідентифікації бізнес-підсистем. Для цього потрібні бізнес-знання, тому реалізація методу не може бути повністю автоматизована.

2 ПЛАНУВАННЯ РЕСУРСІВ ПІДПРИЄМСТВА

Бізнес-системи, відомі як системи планування ресурсів підприємства (Enterprise Resource Planning - ERP), консолідують і спрощують дані з кількох організаційних відділів в одне комплексне рішення, яке задовольняє вимоги всієї корпорації.

Системи ERP функціонують, безперешкодно інтегруючи та координуючи діяльність і завдання, які раніше були фрагментовані та підтримувалися старими, автономними та окремими спадковими системами.

Основою системи ERP є добре структурована база даних, яка підтримує операційні та прийняття рішень вимоги кінцевих користувачів по всій компанії та генерує інформацію для зовнішніх зацікавлених сторін, таких як регуляторні органи та інвестори.

Рис. 2.1. ERP - система

Системи ERP розглядаються як багатофункціональні за своєю природою, оскільки вони задовольняють інформаційні потреби всіх кінцевих користувачів, а також орієнтовані на процес, оскільки пропонують чітке, повне, логічне та точне

уявлення про бізнес-процеси фірми, які є групи взаємопов'язаних завдань, які приносять цінність підприємству. Бізнес-операції часто перетинають кордони відділів і, у багатьох випадках, організаційних кордонів, обмін даними та інформацією із зовнішніми діловими партнерами, такими як клієнти та постачальники, що робить ERP незамінним для компанії. Деякі ключові бізнес-процеси, включені в ERP-системи, показані на рисунку 2.1.

2.1 Причини впровадження ERP

Підприємства, які найчастіше використовують системи ERP, часто мають багато тих самих проблем і розчарувань. На рисунку 2.2 наведено основні аргументи для впровадження ERP бізнесом. Деякі з цих причин розглядаються нижче.

Рис. 2.2. Причини впровадження ERP

Поліпшення ефективності бізнесу

ERP покращує ефективність бізнесу завдяки численним передовим практикам, вбудованим у кілька бізнес-процесів. Найкраща практика – це бізнес-процедура, яка, як правило, вважається успішнішою та/або ефективнішою, ніж інші в певному секторі. Компанії, які впроваджують ERP, у кінцевому підсумку

перероблять свої раніше незв’язані, помилкові, повільні та неефективні процеси, щоб узгодити їх із найкращими практиками в програмному забезпеченні та можуть зменшити операційні витрати, такі як нижчі витрати на запаси, витрати на виробництво або придбання, витрати та збільшення процесів отримання прибутку, таких як час виходу на ринок, маркетинг і продажі та обслуговування клієнтів.

Кожне представлення найкращих практик відрізняє програмне забезпечення одного постачальника ERP від програмного забезпечення іншого, тому визначення найкращих практик системи ERP відповідає вимогам покупця є важливим при виборі продукту постачальника ERP, оскільки ця відповідність впливає на остаточний успіх впровадження. Щоб знайти передовий досвід у різних галузях промисловості та застосувати його у своїх рішеннях, постачальники фінансують значні науково-дослідні та дослідницькі ініціативи. Крім того, це дозволяє постачальнику ERP надавати спеціальні версії свого програмного забезпечення, відомі як вертикальні рішення, які є важливими через унікальні характеристики кожного промислового сектору.

Бажання зростання

Приклади стратегій зростання включають розширення ринку та проникнення, диверсифікацію продуктів, а також злиття та поглинання (mergers and acquisitions - M&A). Системи ERP допомагають з розширенням ринку через прогнозування попиту, яке генерує прогнози для оцінки майбутніх потреб у товарах. Розширене ціноутворення на основі правил є ще однією функцією програмного забезпечення ERP. Ця можливість дозволяє підприємствам зрозуміти сучасні тенденції та тренди сектора, споживачів і конкурентів перед внесенням будь-яких змін у ціни. Підприємства можуть розширити свої асортиментні лінії та запропонувати нові товари та функції своїм клієнтам, диверсифікуючи свої пропозиції. Дані про те, які товари продаються і кому, можуть бути знайдені в системах ERP, а також інформація про те, які продукти просто займають місце на полицях. Нарешті, ERP може допомогти стандартизувати процедури в організаціях під час діяльності з M&A, щоб інтегрувати їх у спільну платформу.

Полегшення роботи співробітників

Визнано, що системи ERP полегшують виконання обов’язків співробітників. ERP робить це частково, надаючи співробітникам доступ до інформації в реальному часі; ця функція значно покращує операційну діяльність, корпоративне управління та управління ризиками підприємства, в результаті чого горизонтально «пов’язана», зосереджена на процесі організація. Системи ERP також пропонують уніфікований інтерфейс користувача та набір інструментів, що підвищує точність, заохочує співпрацю та зменшує непорозуміння. Нарешті, системи ERP розширюють можливості користувачів, надаючи їм доступ до даних, які раніше було неможливо отримати через фрагментовані процедури, підтримувані багатьма старими системами.

Відсутність відповідності

Урядові та інституційні вимоги до відповідності продовжують зростати та розвиватися. Навігація численними законодавчими, нормативними документами та мандатами ланцюга поставок ніколи не була такою складною. Системи ERP можуть допомогти компаніям дотримуватися таких вимог, як GDPR, SOX або управління з контролю за якістю харчових продуктів і медикаментів

Інтеграція даних

З системами ERP дані краще інтегруються, оскільки вони збираються лише один раз і потім розподіляються по всій компанії, зменшуючи ризик неточностей і дублювань елімінуючи трудомістку перевірку даних і узгодження між системами. Оскільки всі користувачі мають доступ до актуальних, точних і всебічних даних, ця функція є вигідною для всіх них.

З ERP, оскільки дані тепер зберігаються в єдиному сховищі даних, процес виправлення помилок спрощується, оскільки їх потрібно виправити лише один раз. Процеси також краще інтегровані, оскільки вони керуються в межах однієї системи, а не розподілені між кількома системами, які були зібрані разом.

Коли системи корпорації з'єднані з кількох джерел, ситуація може викликати проблеми в операціях, призначених для підтримки ефективного функціонування організації.

Заміна старої ERP

Наявність кількох різнорідних систем або використання застарілої ERP-системи, яка працює на застарілій технології або не може підтримувати бізнес-процеси компанії, створює кошмар з обслуговування IT.

Ці системи можуть бути складними для налаштування, а встановлення виправлень і оновлень може займати цінний час і ресурси.

Крім того, оскільки постачальник може більше не працювати, оновлення цих систем може бути недоцільним.

2.2 Недоліки ERP-систем

Впровадження ERP-системи є значно більш складним, ніж просто встановлення комерційно доступного програмного забезпечення; це трудомісткий процес, який вимагає виконання різноманітних завдань і, якщо ним неправильно управляти, може призвести до провалу проекту.

Компанії не повинні легковажно ставитися до вибору впровадження ERP-системи через її важливість.

Усі працівники, від функціональних користувачів до IT-фахівців і вищого керівництва, повинні бути обізнані про цілі проекту ERP і співпрацювати для успішного впровадження.

Компанії, які розглядають можливість впровадження ERP-системи, повинні провести належну перевірку при виборі рішення, яке найкраще відповідає їхнім потребам, і співпрацювати з експертами, які можуть допомогти з різними завданнями, пов'язаними з впровадженням.

Проблеми з людьми

Вищий менеджмент може бути великою проблемою, якщо вони не встановлять переконливий "тон на верху", що система ERP є пріоритетом, або якщо вони не виділять достатньо ресурсів для її впровадження.

Відсутність підтримки з боку працівників також може бути проблемою.

Спадкові системи, які працівники використовували протягом багатьох років, можуть змусити їх почуватися досить комфортно. Вони можуть виступати проти додаткового навчання, організаційних змін і модифікацій бізнес-процесів, які є неминучими, або можуть стверджувати, що система є занадто складною, обмежувальною або негнучкою.

Працівники, які чинять опір системі ERP, можуть створювати непродуктивні обхідні шляхи або створювати власну "тіньову IT", такі як електронні таблиці або старі системи, внаслідок чого вони не використовують систему за призначенням.

Проблеми з програмним забезпеченням

Оскільки ERP-системи складні, їх інсталяція інколи вимагає оплати дорогих системних інтеграторів. Компанії часто намагаються взяти під контроль технології та використовувати їх для трансформації бізнес-процесів у кількісно виміряний та стійкий спосіб. Рівень складності, який раніше не зустрічався і який важко сприйняти, також може бути доданий різними можливостями, параметрами та вимогами до налаштування для підприємств з відносно простими бізнес-вимогами.

Ціна впровадження ERP-систем

Розгортання системи ERP може коштувати мільйони доларів і займати роки, особливо для великих міжнародних компаній. Крім того, після встановлення система ERP потребує постійного «догляду та підживлення», щоб підтримувати її актуальною, стабільною та сумісною з різними програмами, що постійно розвиваються, з якими вона може взаємодіяти. Компанії часто оновлюють і значно вдосконалюють систему ERP. Загальна вартість цього компонента ERP може бути дорожчою, ніж плата за початкове ліцензування програмного забезпечення та впровадження, разом узяті, оскільки плата за обслуговування вимагається щорічно.

Стандартизація

Зазначена вище перевага стандартизації бізнес-процесів також може бути недоліком, якщо жорсткість не відповідає культурі або очікуванням фірми. Крім

того, проблема, яку необхідно вирішити для ефективності встановлення ERP, полягає в тому, що поточна корпоративна культура може не сприяти обміну інформацією між бізнес-одиницями чи підрозділами.

Постачальники та системні інтегратори активно радять підприємствам використовувати найкращі практики для систем ERP, а не налаштовувати програмне забезпечення відповідно до їхніх унікальних робочих процесів. Виняток становлять випадки, коли компанії налаштовують програмне забезпечення відповідно до спеціальної бізнес-стратегії, яка відрізняє їх від конкурентів. Загальна вказівка полягає в тому, що налаштувати програмне забезпечення ERP для отримання цієї можливості необхідно, якщо певна процедура робить бізнес конкурентоспроможним або необхідна для дотримання законодавства.

2.3 Модулі

Системи ERP пропонуються як модулі, які є наборами підключених програмних додатків, які виконують ключові організаційні завдання, такі як бухгалтерський облік або виробництво. Кожен модуль призначений для підтримки певного бізнес-процесу. Основні модулі, які забезпечують базові функціональні можливості для управління бізнес-процесами, складають ядро ERP.

Це включає фінансовий менеджмент, ланцюг постачання, управління людськими ресурсами, управління відносинами з клієнтами та інші важливі бізнес-процеси. «Ядро» є центральною та фундаментальною частиною системи ERP і забезпечує інтегроване рішення для управління даними компанії. Ось деякі з найпоширеніших модулів, які можна знайти в системах ERP:

- Фінансовий менеджмент: Цей модуль відповідає за управління фінансовими транзакціями, такими як рахунки до сплати та рахунки до отримання, головна книга та фінансова звітність.

- Облік: Цей модуль включає підмодулі для обліку витрат, заробітної плати, управління основними засобами та інших функцій бухгалтерського обліку.
- Людські ресурси: Цей модуль управляє інформацією про працівників, пільгами, заробітною платою та іншими функціями HR.
- Закупівлі: Цей модуль охоплює всі аспекти закупівель, включаючи управління постачальниками, створення та управління замовленнями на покупку, а також управління запасами.
- Управління ланцюгом поставок: Цей модуль управляє потоком товарів і послуг, включаючи закупівлі, виробництво та логістику.
- Виробництво: Цей модуль допомагає управляти виробничим процесом, включаючи планування, розклад та відстеження.
- Продажі та маркетинг: Цей модуль підтримує діяльність з продажу та маркетингу, включаючи управління лідами, відстеження можливостей та управління взаємовідносинами з клієнтами.
- Управління взаємовідносинами з клієнтами (Customer Relationship Management - CRM): Цей модуль підтримує діяльність, пов'язану з клієнтами, таку як маркетинг, продажі та обслуговування клієнтів.

Конкретні модулі, включені в певну ERP-систему, можуть варіюватися в залежності від розміру організації та її унікальних бізнес-вимог. Більшість ERP-програм є достатньо гнучкими, щоб дозволити підприємствам купувати лише ті компоненти, які їм потрібні «à la carte», що дозволяє компаніям мати рішення, «спеціалізоване» відповідно до своїх потреб. Модульна архітектура має перевагу, оскільки дозволяє постачальникам ERP створювати продуктові рішення для конкретних галузей. Іноді модулі, які підтримують основну бізнес-діяльність, називають комплектом, який складається з кількох підмодулів або компонентів. Приклад показано на рисунку 2.3, де зображені модулі ERP для виробничої компанії.

Supply Chain		
Plant Maintenance	Purchasing	Quality Management
Sales and Distribution	Shop Floor Management	Inventory Management
Manufacturing	Warehouse Management	Advanced Planning
Financial Accounting		
General Ledger	Cash Management	Accounts Payable
Accounts Receivable	Fixed Assets	Financial Consolidation
Management Accounting		
Cost Center Accounting	Product Costing	Budgeting
Profit Center Accounting	Activity-Based Costing	Profitability Analysis
Human Resources		
Personnel Management	Payroll	Learning Management
Time and Attendance	Benefits	Recruitment Management

Рис. 2.3. Приклади модулів ERP для виробничої компанії

На завершення, різні модулі в ERP-системі працюють разом, щоб забезпечити комплексне рішення для управління бізнес-процесами та даними. Інтегруючи всі бізнес-функції в єдину систему, організації можуть спростити процеси, зменшити дублювання даних, покращити бізнес-аналітику та приймати більш обґрунтовані рішення.

2.4 Технологія

Система ERP має далекосяжний вплив, який впливає на користувачів у всій організації, а також на її клієнтів, постачальників та інших ділових партнерів. Оскільки необхідно підтримувати велику кількість користувачів, які мають різні вимоги до обробки та звітності, це так важливо мати вдосконалене та адаптоване програмне забезпечення, яке використовує найсучасніші технології. З огляду на те, що система ERP відіграє вирішальну роль у задоволенні операційних та інформаційних потреб організації, дуже важливо мати повне розуміння технології, яка підтримує інтегровану систему та забезпечує надійне, масштабоване та зручне рішення для керування широким спектром бізнес-процесів і даних.

Трирівнева архітектура клієнт-сервер

Архітектура клієнт-сервер[19] широко використовується в сучасних обчисленнях і є фундаментальним аспектом багатьох систем, включаючи системи ERP. Ця архітектура являє собою обчислювальну модель, у якій сервер (додаток Back-end) надає послуги клієнтам (додаток Frontend) через мережу. Клієнт запитує послугу або ресурс у сервера, а сервер відповідає, надаючи запитувану інформацію або виконуючи запитане завдання. Ця архітектура дозволяє ефективно та масштабовано розподіляти ресурси та завдання, оскільки сервер може обробляти запити від кількох клієнтів одночасно.

У цьому контексті ми можемо розділити програму на три логічні компоненти (3-рівнева архітектура), якими є рівень клієнта, рівень програми та рівень бази даних. Ця архітектура забезпечує масштабоване, гнучке та безпечне рішення для програмних додатків. На рисунку 2.4 показано детальне пояснення кожного рівня.

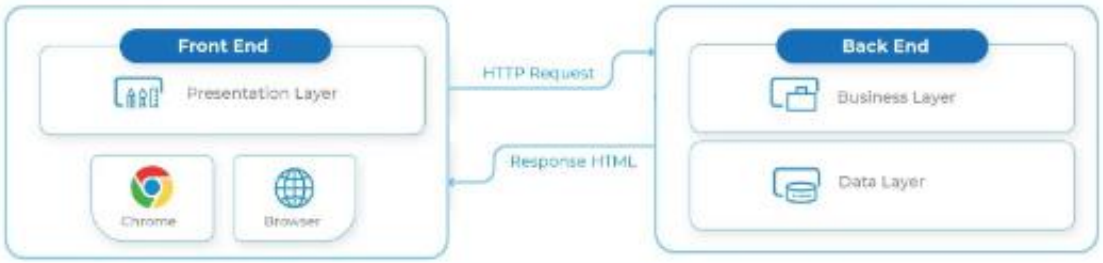

Рис. 2.4. Архітектура веб-додатка

- Рівень клієнта Презентаційний шар: Цей рівень відповідає за представлення інтерфейсу користувача кінцевому користувачу. Він надає інтерфейс, через який користувачі взаємодіють з додатком. Рівень клієнта може бути реалізований як автономний додаток або як веб-додаток, доступний через веб-браузер.

- Рівень додатка Бізнес-шар: Цей рівень відповідає за обробку запитів користувачів і повернення результатів до рівня клієнта. Він відповідає за обробку бізнес-логіки та обробку даних. Рівень сервера додатків зазвичай впроваджений як веб-сервер.

- Рівень бази даних Доступ до даних: Цей рівень відповідає за зберігання та управління величезною кількістю даних, що генеруються додатком. Це ключовий компонент веб-додатку, який зберігає та управляє інформацією для веб-додатку. Ви можете шукати, фільтрувати та сортувати інформацію на основі запиту користувача. Зазвичай він реалізується за допомогою надійної системи управління базами даних.

Триступенева архітектура дозволяє розділити обов'язки, при цьому кожен рівень має конкретну роль і відповідальність. Це розділення полегшує розробку, обслуговування та оновлення додатку, оскільки зміни можуть бути внесені в один рівень без впливу на інші. Крім того, розділяючи систему на окремі рівні, покращуються продуктивність і безпека.

Клієнт не має прямого доступу до даних, натомість усі дані проходять через сервер додатків, який контролює та регулює доступ до інформації. Це дозволяє більш ефективно та безпечно управляти даними.

Можливість розгортання серверів додатків на кількох машинах забезпечує вищу масштабованість, кращу продуктивність і краще повторне використання.

Трирівнева архітектура клієнт-сервер є широко використовуваною архітектурою для систем ERP, оскільки вона забезпечує масштабовану, гнучку та безпечну платформу для керування складними бізнес-процесами та даними.

Розгортання

Систему ERP можна розгорнути двома способами: локально або в хмарі, кожен з яких має свої переваги та недоліки. Найкращий метод працевлаштування залежить від конкретних потреб і вимог організації.

Локально

Традиційним підходом до розгортання ERP є локальна ERP. У цьому методі розгортання програмне забезпечення ERP встано влюється та працює на комп’ютерах у межах власних фізичних об’єктів організації. Це дозволяє повністю контролювати програмне забезпечення та дані, але також вимагає від організації надання необхідного апаратного забезпечення, зберігання та технічної

підтримки. Компанії, які обирають варіант «on-prem», зазвичай є більшими компаніями з більшими бюджетами, наявною IT-інфраструктурою та досвідченим IT-персоналом для підтримки програмного забезпечення та інфраструктури. Через великі попередні витрати, які часто включають вартість апаратного та програмного забезпечення, локальне ERP зазвичай розглядається як капіталовкладення.

Хмара

Хмарне розгортання ERP стає все більш популярним, коли ERP-система розміщується постачальником або третьою стороною на спільних обчислювальних ресурсах, до яких можна отримати доступ через Інтернет. Ці ресурси зберігаються в центрах обробки даних, призначених для розміщення різних програм на кількох платформах.

Цей метод розгортання пропонує більшу масштабованість і гнучкість, а також зменшені витрати на апаратне забезпечення та технічну підтримку, але вона вимагає, щоб організація довіряла третій стороні з доступом до своїх даних. Клієнти мають доступ до системи ERP за потребою і платять за програмне забезпечення щомісячно або щорічно. Цей метод оплати за програмне забезпечення ERP на основі підписки називається програмне забезпечення як послуга (SaaS), привабливий варіант для бізнесу, який прагне зменшити початкові витрати та планувати бюджет на ERP у довгостроковій перспективі.

Сьогодні майже всі постачальники ERP пропонують якусь форму хмарного розгортання, оскільки це має багато переваг у порівнянні з локальним розгортанням.

Переваги

- Одна з ключових переваг полягає в тому, що постачальники хмарних ERP підтримують, оновлюють і займаються обслуговуванням інфраструктури системи ERP.

- Хмарний ERP є більш масштабованим, ніж локальний, що ідеально підходить для стартапів і швидкозростаючих бізнесів.

- Компанії, які обирають хмарний ERP замість локального, тепер можуть насолоджуватися більшою впевненістю в тому, що постачальник хмари має актуальні контролі, такі як резервне копіювання даних, двофакторна аутентифікація, шифрування конфіденційних даних і план відновлення після катастрофи.

Недоліки

- Багато постачальників, які пропонують хмарні рішення, в основному зосереджені лише на одній конкретній області. Дуже мало постачальників хмарних послуг пропонують набір продуктів для задоволення потреб середніх і великих організацій.

- Багато рішень хмарного ERP обмежені в тому, що клієнт може зробити в плані налаштування.

- Хоча хмарний ERP зазвичай вважається менш дорогим, ніж локальний, дослідження показали, що протягом 10-річного періоду загальні витрати для кожного з них зрештою зрівнюються. Хоча витрати на хмарний ERP є меншими на початку, витрати з часом наздоганяють.

Таким чином, економічна ефективність хмарного ERP не є такою великою, як спочатку вважалося.

Налаштування

Незвично, коли система ERP повністю відповідає потребам компанії, особливо якщо компанія є великою глобальною організацією. Часто виникають проблеми, які виходять за межі того, що програмне забезпечення ERP може вирішити за допомогою конфігурації. Приклади цих питань включають:

- Створення додаткової функціональності, яка не надається системою ERP.

- Встановлення з'єднань між системою ERP та сторонніми системами.

- Додавання додаткових полів до бази даних ERP.

Ці типи проблем зазвичай потребують розробки та програмування для вдосконалення системи ERP. Це відоме як налаштування, яке передбачає додавання спеціального коду для розширення можливостей і функцій системи ERP. Зазвичай це налаштування виконується, коли всі спроби знайти рішення за

допомогою конфігурації зазнають невдачі. Оскільки персоналізація потребує часу та грошей, компаніям слід прагнути звести їх до мінімуму.

Ринок

Зростання ринку ERP можна пояснити зростаючим інтересом з боку малих і середніх підприємств та розвитком нових ERP-додатків для хмарних і мобільних платформ. Однак не всі постачальники ERP пропонують однакову якість програмного забезпечення, яке можна поділити на три категорії на основі специфічних критеріїв, як показано в таблиці 2.1.

- Рівень I (Enterprise Class) — це програмне забезпечення, розроблене для великих всесвітніх корпорацій зі значною ринковою капіталізацією та річним доходом, що перевищує 750 мільйонів доларів США. Ці рішення є дуже дорогими через їхні широкі можливості, включаючи здатність керувати складними організаційними структурами та вирішувати міжнародні проблеми, такі як кілька валют і різні правила бухгалтерського обліку. Лише деякі постачальники ERP мають необхідний розмір, ресурси та повну функціональність для підтримки великого обсягу щоденних транзакцій, з якими зазвичай стикаються компанії рівня I.
- Категорія ERP-систем Рівня II (середній ринковий клас - the Mid-Market Class) призначена для компаній середнього розміру та може бути поділена на вищу та нижчу підкатегорії. Системи верхнього рівня II призначені для компаній із річним доходом від 250 до 750 мільйонів доларів США. Системи нижчого рівня II зазвичай призначені для компаній із річним доходом від 10 до 250 мільйонів доларів США. Ці постачальники пропонують програмне забезпечення, яке призначене для однієї або кількох юридичних осіб і місць розташування, але з обмеженими функціями порівняно з рішеннями постачальників рівня I. Як наслідок, ці системи ERP дешевші, ніж системи рівня I. Як правило, їх легше впроваджувати та підтримувати, та розроблено спеціально лише для кількох галузей.
- Системи класу III (Small Business Class клас малого бізнесу) системи ERP призначені для менших підприємств з річним доходом нижче 10 мільйонів

доларів, які працюють в межах однієї країни. Вони є найбільш доступними серед різних рівнів систем ERP. Ринок насичений багатьма постачальниками програмного забезпечення в цій категорії, деякі з яких пропонують потужні точкові рішення, які можна використовувати для покращення системи ERP Рівня I або Рівня II.

Таблиця 2.1

Характеристики рівнів постачальників ERP

Рівень I	Рівень 2	Рівень 3
Висока складність Найвища вартість Багато галузевих рішень Великі компанії Підтримка глобальної функціональності	Середня складність Середня вартість Менше галузевих рішень Компанії середнього ринку Дійте в кількох країнах	Низька складність Найнижча вартість Найменше галузевих рішень Малі компанії Не підтримує глобальну функціональність

На ринку ERP спостерігається тенденція, коли постачальники Рівня II і Рівня III прагнуть обслуговувати більші компанії, покращуючи можливості свого програмного забезпечення та масштабованість, тоді як постачальники Рівня I охоплюють менші компанії, пропонуючи спрощені версії свого програмного забезпечення та купуючи хмарних постачальників. Це призводить до того, що межі між рівнями ERP стають менш чіткими, оскільки постачальники прагнуть збільшити свою частку ринку.

Вартість

Численні проекти ERP перевищують бюджет, часто через непередбачені організаційні чи технологічні проблеми, розширення масштабів або нереальний бюджет проекту. Складання бюджету для ERP може бути складним, оскільки деякі витрати важко передбачити спочатку. У цьому розділі детально описано всі витрати, пов’язані з обчисленням загальної вартості володіння системою (total cost of ownership - TCO). Деякі з цих витрат будуть одноразовими, а інші періодичними [16].

Вартість ліцензії на програмне забезпечення

Як правило, ціна системи ERP залежить від:

- Кількість співробітників, які будуть використовувати систему
- Рівень постачальника розгортається, програмне забезпечення рівня 1 дорожче за рівень 2, а рівень 3 буде найдешевшим
- Кількість придбаних модулів

Переважна більшість ліцензій на програмне забезпечення ERP надається з використанням безстрокової моделі ліцензування, яка вимагає сплати попередньої ціни ліцензії, перш ніж постачальник надасть доступ до програми протягом нескінченного періоду часу. Крім того, клієнти повинні оплачувати щорічні витрати на обслуговування, щоб отримати підтримку, оновлення та майбутні оновлення програмного забезпечення. Для локальних реалізацій постійне ліцензування є стандартним. У безстроковому ліцензуванні використовується кілька методів ліцензування:

- Ліцензування за іменованими користувачами. Компанія визначає, скільки унікальних користувачів буде використовувати ERP-систему, і сплачує ліцензійний збір за кожного з них. Безліч постачальників ERP пропонують різні категорії іменованих користувачів, такі як ліцензування для активних користувачів, для користувачів, які використовують більше можливостей системи і, отже, сплачують більший ліцензійний збір, або ліцензування для випадкових користувачів, для користувачів, які просто читають звіти або списки.

- Одночасне ліцензування користувачів. Тип безстрокової ліцензії дає змогу використовувати необмежену кількість призначених користувачів і облікових записів, але обмежує кількість осіб, які можуть активно використовувати програмне забезпечення одночасно. Ліцензування одночасних користувачів часто дешевше, ніж ліцензування іменованих користувачів, оскільки вимагає оплати лише за приблизну кількість одночасних користувачів. Однак важливо точно передбачити їх кількість одночасних користувачів, інакше працівники можуть зіткнутися з труднощами під час входу та використання програмного забезпечення.

В деяких випадках компанія має специфічні вимоги, які не можуть бути виконані ERP-системою, яку вона придбала, і модифікація ERP для задоволення цих вимог є занадто дорогою. Щоб вирішити цю проблему, можна використовувати стороннє програмне забезпечення, відоме як "додатки". Вони забезпечують додаткову функціональність або логіку для допомоги у вирішенні конкретних бізнес-потреб. Щоб забезпечити безперебійну інтеграцію з ERP-системою, рекомендується отримати рекомендації від постачальника ERP або системного інтегратора щодо того, яке рішення "додатка" буде найкращим.

Впровадження ERP-системи вимагає потужної та сучасної IT-інфраструктури. Якщо ERP працює на місці, компанії потрібно буде інвестувати в IT-апаратуру, таку як сервери, маршрутизатори, резервне копіювання, накопичувачі, настільні комп'ютери, ноутбуки, планшети та принтери. Також потрібно врахувати заходи для резервування, доступу до мережі, електропостачання та безпеки. Це може включати найм додаткового персоналу та збільшення площі дата-центру, з витратами, що варіюються від одноразових витрат, таких як придбання серверів, до постійних витрат, таких як комунальні рахунки та заробітні плати.

Вартість бази даних у проекті ERP може коливатися від кількох тисяч доларів до сотень тисяч доларів, залежно від кількох факторів, таких як тип бази даних, обсяг даних, що зберігаються, і кількість користувачів, які отримують доступ до бази даних. Постачальники ERP нададуть специфікації для типу бази даних, яка потрібна.

Витрати, пов'язані з впровадженням програмного забезпечення ERP, є одними з найдорожчих частин загальної вартості володіння (total cost of ownership - TCO). Вартість функціональних і технічних консультантів, які відіграють ключову роль у впровадженні, може бути значною частиною цих витрат залежно від того, наскільки компанія буде покладатися на системного інтегратора. Складність проекту та консультанти з різних географічних локацій також можуть вплинути на погодинну ставку.

Витрати на обслуговування та підтримку

Вартість ERP-системи не закінчується, коли програмне забезпечення запущено. Щоб забезпечити безперебійну роботу системи, компанії повинні мати план для постійного обслуговування та підтримки, також відомий як послуги управління додатками (application management services - AMS). Це включає функціональну та технічну підтримку, оновлення та виправлення, моніторинг програмного забезпечення, резервне копіювання та відновлення. Деякі компанії можуть виконувати ці завдання власноруч, а інші можуть найняти сторонню компанію для надання цих послуг. Типова вартість обслуговування коливається від 18% до 25% від початкової вартості ліцензії на програмне забезпечення та може бути сплачена безпосередньо постачальнику ERP або сторонній компанії, яка керує системою. Для підтримки програмного забезпечення доступні різні рівні, причому вищі рівні пропонують більше послуг за вищу вартість. Преміальна підтримка може включати наявність представника від постачальник ERP на місці під час впровадження проекту та протягом періоду часу після впровадження, а також пріоритетний доступ до довідкових квитків. Базова підтримка може надавати лише доступ до довідкового порталу, де клієнти можуть реєструвати квитки та отримувати відповіді на запитання.

Преміум-підтримка може включати присутність представника постачальника ERP на місці під час впровадження проекту та протягом певного часу після впровадження, а також пріоритетний доступ до запитів на допомогу. Базова підтримка може надавати лише доступ до порталу допомоги, де клієнти можуть реєструвати запити та отримувати відповіді на запитання.

Хмара

Загальна вартість володіння (total cost of ownership - TCO) у контексті хмари є такою ж, як ми обговорювали раніше, але часто всі витрати включені в єдину ліцензію на основі підписки. Клієнти, які підписуються на цю модель, отримують доступ до програми на визначений термін, наприклад, щомісяця або щорічно, що охоплює не лише використання програмного забезпечення, але й обслуговування, підтримку, заплановані оновлення та оновлення, що надаються постачальником хмари. Ця комплексна підписка часто включає базову пропозицію бази даних з

можливістю розширення сховища за додаткову плату. Переважно, ERP-додатки рекламуються як Програмне забезпечення як послуга (SaaS), модель доставки на основі хмари, в якій постачальник несе відповідальність за основну інфраструктуру, включаючи хостинг, обслуговування та підтримку. Це означає, що витрати на IT-інфраструктуру, коли система ERP розміщена постачальником або стороннім постачальником, об'єднуються в щомісячну плату за підписку. Компанії, які використовують публічні хмарні платформи, такі як Microsoft Azure, зазвичай сплачують регулярну плату за оренду інфраструктури на додаток до ліцензійних зборів за програмне забезпечення, що належать постачальнику ERP.

Конкуренти

Ринок ERP (планування ресурсів підприємства) зазнає значних змін з появою менших, більш гнучких гравців. Ці менші постачальники кидають виклик традиційній домінуючій позиції більших, усталених компаній, таких як SAP, Oracle та Microsoft. Характеризуючись своїми інноваційними, хмарними рішеннями, ці нові гравці зосереджуються на надання спеціалізованих і галузевих систем ERP. Цей підхід контрастує з універсальними монолітними системами, традиційно пропонованими великими постачальниками [45].

Ця тенденція до менших, більш гнучких постачальників ERP зумовлена зростаючою популярністю хмарних рішень. Ці рішення пропонують переваги, такі як нижчі витрати, більша масштабованість і гнучкість, що робить їх особливо привабливими для малих і середніх підприємств. Пандемія ще більше прискорила цей зсув, оскільки бізнес шукає рішення, які підтримують віддалену роботу та пропонують більшу оперативну гнучкість.

Ці функції є критично важливими для бізнесу при виборі ERP-системи, оскільки вони безпосередньо впливають на зручність використання, масштабованість і загальну ефективність бізнесу.

3 ВИКОРИСТАННЯ МІКРОСЕРВІСІВ ДЛЯ НАЛАШТУВАННЯ МУЛЬТИТЕНАНТНОГО SAAS

3.1 Хмарні обчислення

Хмарні обчислення є одним із найпоширеніших варіантів використання розподілених систем і за останнє десятиліття привернули величезну увагу науковців і промисловості. За останні роки вона стала однією з найбільш швидкозростаючих галузей промисловості та дослідницьких точок, зміна способу використання та управління розподіленими обчислювальними ресурсами, трансформація процесу розробки програмного забезпечення та створення багатьох революційних продуктів. Зростання популярності хмарних обчислень викликало гостру конкуренцію між технологічними гігантами, зокрема Amazon, Google, Microsoft і Alibaba.

Вони вже стали основними постачальниками хмарних послуг, запустивши такі хмарні платформи, як Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure та Alibaba Cloud, щоб пропонувати різноманітні хмарні сервіси та інфраструктури громадськості.

Поява хмарних обчислень революціонізувала наш підхід до обчислень і управління даними.

Хмарні обчислення — це модель для забезпечення універсального, зручного, на вимогу доступ до спільного пулу конфігурованих обчислювальних ресурсів (наприклад, мереж, серверів, зберігання, додатків і послуг), які можуть бути швидко надані та звільнені з мінімальними зусиллями з управління або взаємодії з постачальником послуг.

Ця модель хмари складається з п'яти основних характеристик, трьох моделей послуг і чотирьох моделей розгортання. Хмарні обчислення пропонують високо гнучку модель доставки послуг, що дозволяє отримувати доступ до різних ресурсів на вимогу, таких як зберігання, обчислювальна потужність і додатки, через Інтернет.

Це усуває необхідність у локальних серверах, переміщуючи обробку даних на онлайн віддалені сервери та пропонуючи економічну модель ціноутворення "плати за те, що використовуєш". Послуги, такі як Amazon Web Services (AWS), надають технологічну інфраструктуру, що дозволяє користувачам легко масштабувати операції.

Крім того, ця модель сприяє економічній ефективності, оскільки організації платять лише за ресурси, які вони споживають, підтримуючи масштабований і гнучкий підхід до управління ресурсами. Мережа, часто Інтернет, слугує каналом між користувачами та хмарними послугами, забезпечуючи управління даними з сильними заходами безпеки.

Основні характеристики

- Самообслуговування на вимогу: Користувачі можуть самостійно налаштовувати та управляти своїми обчислювальними потребами, такими як час сервера та мережеве зберігання, автоматично, усуваючи необхідність у прямій взаємодії з постачальниками послуг.
- Широкий доступ до мережі: Послуги доступні через інтернет, використовуючи стандартні методи, які підтримують різноманітні пристрої, включаючи смартфони, планшети, ноутбуки та настільні комп'ютери.
- Об’єднання ресурсів: обчислювальні ресурси постачальника об’єднуються для обслуговування різних клієнтів за моделлю з кількома клієнтами. Ресурси динамічно розподіляються та перерозподіляються на основі попиту користувача, причому користувач зазвичай не знає або не контролює точне фізичне розташування ресурсів, але може призначити розташування на більш широкому рівні (наприклад, країна, штат або центр обробки даних).
- Швидка еластичність: послуги можна швидко збільшити або зменшити відповідно до попиту, при цьому процес масштабування іноді відбувається автоматично. З точки зору користувача, пропозиція доступних ресурсів часто здається безмежною і може бути придбана в будь-якому обсязі в будь-який час.
- Вимірювана послуга: хмарні платформи автоматично відстежують, контролюють і звітують про використання ресурсів за допомогою функції

вимірювання, яка працює з відповідним рівнем деталізації, залежно від типу служби, наприклад місця для зберігання, потужності обробки, пропускної здатності або кількості активних користувачів. Цей облік забезпечує чітку видимість використання послуг як для постачальника, так і для споживача.

Моделі послуг

Хмарні обчислення революціонізували підхід бізнесу до технологій, пропонуючи спектр послуг, які відповідають різним вимогам щодо контролю, гнучкості та управління.

Оскільки організації переходять на рішення на основі хмари, розуміння різних моделей послуг стає критично важливим для використання всього потенціалу хмари.

Системи хмарних обчислень зазвичай є розподіленими системами з кількома клієнтами.

Завдяки високій масштабованості, високій доступності, відмовостійкості, системній моделі виставлення рахунків і низьким витратам на керування використання хмарних рішень стало для розробників і компаній природним способом створювати, тестувати та розгортати нові програми та послуги. Велика кількість компаній також планує перенести свої послуги в публічну або локальну приватну хмару.

Сучасні хмарні обчислення зазвичай мають чотири парадигми, а саме: інфраструктура як послуга (IaaS), платформа як послуга (PaaS), програмне забезпечення як послуга (SaaS) і функція як послуга (FaaS) (рис. 3.1).

Інфраструктура як послуга (IaaS)

Інфраструктура як послуга (IaaS) — це трансформаційний підхід до управління IT-ресурсами, який пропонує гнучкий і на вимогу доступ до основних інфраструктурних послуг через Інтернет. Це включає віртуальні машини, сховище та мережу, які можна налаштувати і виставляється рахунок на основі фактичного використання. IaaS надає організаціям безпрецедентний контроль над своїми IT-ресурсами, дуже нагадуючи традиційну локальну інфраструктуру. Це забезпечує легку масштабованість без необхідності дорогих початкових інвестицій в

апаратне забезпечення. IaaS надає споживачам можливість надавати обробку, зберігання та мережеві ресурси, розгортати та запускати різноманітне програмне забезпечення, включаючи операційні системи та програми. Цей рівень налаштування дозволяє організаціям адаптувати IT-середовище до своїх конкретних потреб, забезпечуючи безперебійну та ефективну роботу в хмарі.

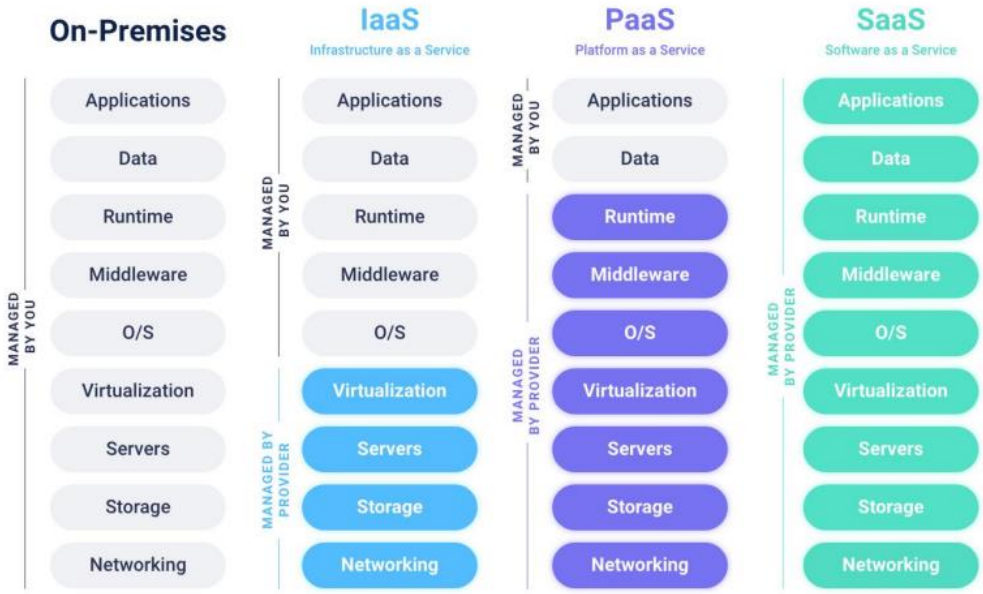

Рисунок 3.1 показує три основні моделі хмарних послуг, які формують те, що часто називають "стеком" хмарних обчислень, включаючи Інфраструктуру як послугу (IaaS), Платформу як послугу (PaaS) та Програмне забезпечення як послугу (SaaS)

IaaS — це найбільш проста та гнучка модель хмарних обчислень. Провайдери IaaS, такі як Amazon EC2 і Google Cloud Compute Engine, здають клієнтам в оренду обчислювальні ресурси разом зі сховищем і мережевими послугами. Хоча екземпляри операційної системи (ОС) на вимогу у формі віртуальних машин (VM) є основною одиницею розгортання в IaaS, виділені (голі) машини також можуть бути передані аутсорсингу [13].

Постачальники послуг використовують IaaS для автоматизованого розгортання своїх програм у середовищі з повним контролем над

інфраструктурою, ОС, проміжним програмним забезпеченням, залежностями, програмуванням, даними, середовищем виконання та програмами.

Платформа як послуга (PaaS)

Платформа як послуга (PaaS) є значною інновацією в хмарних обчисленнях, пропонуючи комплексні апаратні та програмні ресурси для розробки додатків на основі хмари. Ведучі постачальники PaaS спрощують розробку, ефективно керуючи основною інфраструктурою, що дозволяє зосередитися на створенні додатків. З PaaS ви звільнені від нагляду за інфраструктурою, що дозволяє приділяти увагу розгортанню та управлінню додатками, покращуючи операційну ефективність шляхом усунення забезпечення ресурсами, планування потужностей та обслуговування.

PaaS надає середовище для створення, тестування та управління програмними додатками без необхідності керувати основною хмарною інфраструктурою. Як користувач, ви контролюєте свої додатки та їх налаштування хостингу, спрощуючи процес розробки, дозволяючи зосередитися виключно на створенні та розгортанні ваших додатків.

PaaS функціонує на більш високому рівні, ніж IaaS. Постачальники PaaS беруть на себе відповідальність за ОС, проміжне програмне забезпечення та середовище виконання та здають в оренду готову до використання платформу, на якій клієнти можуть розробляти та розгортати свої послуги. У результаті клієнт PaaS не може керувати основною інфраструктурою (зазвичай віртуальною машиною та ОС), але має повний контроль над своїми розгорнутими програмами та іноді обмежений контроль над зберігання та мережа. AWS Elastic Beanstalk і Google App Engine є двома прикладами популярних рішень PaaS.

Програмне забезпечення як послуга (SaaS)

Програмне забезпечення як послуга (SaaS) — це модель хмарних обчислень, яка надає програмні додатки через інтернет на основі підписки. У цьому підході постачальники хмарних послуг керують і хостять додатки, забезпечуючи їх доступність, продуктивність і безпеку. Відомі приклади пропозицій SaaS включають Google Workspace, Microsoft Office 365 та Salesforce. SaaS надає повне

програмне рішення через Інтернет, включаючи додаток та його основну інфраструктуру, яка повністю обслуговується постачальником хмарних послуг. Цей підхід звільняє користувачів від управління інфраструктурою, оскільки постачальник займається оновленнями програмного забезпечення та заходами безпеки.

SaaS функціонує на навіть вищому рівні, ніж PaaS, надаючи клієнтам, як правило, кінцевим користувачам програмного забезпечення, готову до використання пакетну програму. Постачальники SaaS керують майже всім, що потрібно для роботи розміщеної програми, і обслуговують програмне забезпечення, наприклад оновлення програм, виправлення безпеки та виправлення помилок, усуваючи накладні витрати на керування, встановлення та використання програмного забезпечення в локальному середовищі. Кінцеві користувачі можуть орендувати та використовувати сервіс онлайн, скорочуючи витрати на розповсюдження та використання програмного забезпечення. Деякі з найпоширеніших типів додатків SaaS включають служби відеоконференцій і хмарні служби зберігання.

Користувачі можуть отримувати доступ до цих додатків через різні пристрої та веб-браузери, насолоджуючись доступним і спрощеним досвідом. SaaS дозволяє користувачам ефективно використовувати додатки, розміщені в хмарі, зосереджуючись виключно на використанні програмного забезпечення, а не на його обслуговуванні.

Моделі розгортання

Хмарні обчислення, ключовий фактор у сучасному управлінні IT-ресурсами, пропонують різні моделі розгортання, адаптовані до різних бізнес-потреб, вимог безпеки та вимог до масштабованості. Це дослідження зосереджується на моделях публічної, приватної, гібридної та спільної хмари, обговорюючи їх унікальні особливості, переваги та міркування.

Приватна хмара

Приватна хмара — це форма хмарних обчислень, що надає ексклюзивні ресурси та послуги через приватну мережу, присвячену виключно одній

організації. Вона пропонує підвищену безпеку та ізоляцію даних, з можливістю налаштування інфраструктури та програмного забезпечення під специфічні потреби та робочі процеси. Хоча приватні хмари пропонують надійну безпеку та контроль, вони можуть бути дорожчими через відповідальність організації за управління інфраструктурою та масштабування. Вони можуть бути розгорнуті на місці або хоститися сторонніми постачальниками, які спеціально обслуговують бізнеси, що потребують високої безпеки та стандартів відповідності.

Комбінація переваг хмарних обчислень з підвищеною безпекою даних та налаштуванням робить приватні хмари привабливим варіантом для бізнесів, які обробляють чутливі дані.

Публічна хмара

У публічних хмарних обчисленнях сторонній постачальник послуг управляє всіма рівнями інфраструктури, включаючи сервери, зберігання та обчислювальні ресурси.

Клієнти отримують доступ до цих ресурсів через Інтернет і отримують рахунки на основі фактичного використання, створюючи економічну та гнучку модель оплати за фактом використання. Ці хмарні середовища усувають необхідність значних інвестицій у дороге обладнання, демократизуючи доступ до хмарних обчислень.

Важливо зазначити, що публічні хмари працюють на спільній інфраструктурі, що пропонує економічні переваги, але викликає занепокоєння щодо ізоляції даних та конфіденційності, оскільки дані та програми кількох клієнтів співіснують на одній і тій же інфраструктурі.

Рисунок 3.2 показує, як публічний хмарний ринок складається з численних постачальників хмарних послуг Amazon, Microsoft та Google становлять 65% від загального ринку хмар у 2023 році. Решта публічного хмарного ринку розподілена між IBM, Alibaba, Oracle та кількома меншими гравцями.

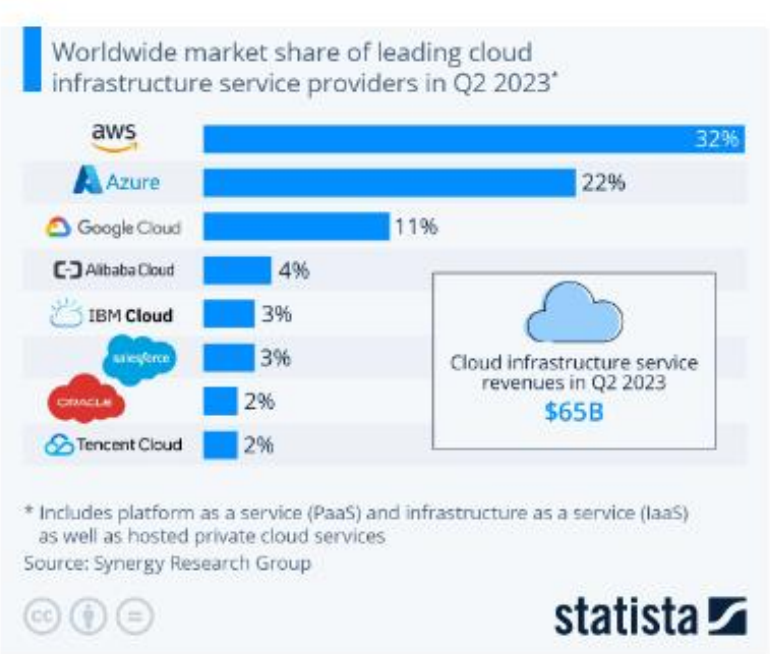

Рис. 3.2 Частка світового ринку провідних постачальників послуг хмарної інфраструктури

Гібридна хмара — це вдосконалена модель хмарних обчислень, яка поєднує публічні та приватні хмари, надаючи організаціям гнучкість у розподілі своїх додатків і навантажень за потреби. Ця конфігурація дозволяє мати більший контроль і масштабованість, ніж використання лише публічних хмар, дозволяючи бізнесу зберігати чутливі дані в безпеці, одночасно користуючись ефективністю публічної хмари. Це ідеально підходить для компаній, які потребують як сильної безпеки, так і можливості швидко адаптуватися та масштабуватися.

В таблиці 3.1 представлено порівняння основних постачальників хмарних послуг.

Гібридна хмара пропонує універсальну IT-інфраструктуру, яка підлаштовується під складні потреби сучасного бізнесу, покращуючи безпеку, відповідність і загальну ефективність. Це робить її цінним активом для компаній, які орієнтуються на швидко змінюваний цифровий світ:

- Вартість: Хмарні обчислення зменшують капітальні витрати на купівлю апаратного та програмного забезпечення, налаштування та управління локальними дата-центрами, що може швидко накопичуватися.

Таблиця 3.1

Порівняння основних постачальників хмарних послуг

Ім'я	Вартість/год	плюси	мінуси
Amazon AWS	$0.0255	Надійність, якість, професійна підтримка	Дорого, незважаючи на регулярне зниження ціни
Google GCP	$0.0475	Надійність, доступна ціна	Обмежені функції та послуги
Microsoft Azure	$0.043	Найкраща інфраструктура конфігурація	Незадовільний клієнтський досвід і технічна підтримка
IBM Cloud	$0.04	Гнучкість, швидкість, сумісність	Складне ціноутворення модель і платформа може бути повільним

- Швидкість: Хмарні послуги зазвичай надаються за запитом, дозволяючи швидко виділяти величезні обсяги обчислювальних ресурсів за кілька хвилин, що надає бізнесу гнучкість і полегшує планування потужностей.

- Глобальний масштаб: Це включає можливість еластично масштабувати IT-ресурси, надаючи необхідну кількість обчислювальної потужності, зберігання та пропускної здатності, коли і де це потрібно.

- Продуктивність: Хмарні обчислення усувають багато трудомістких завдань, пов'язаних з управлінням локальними дата-центрами, дозволяючи IT-командам зосередитися на більш важливих бізнес-цілях.

- Продуктивність: Хмарні послуги працюють на всесвітній мережі безпечних дата-центрів, які регулярно оновлюються до останнього покоління швидкого та ефективного обчислювального обладнання, пропонуючи переваги, такі як зменшена затримка в мережі та більші економії від масштабу.

- Надійність: Резервне копіювання даних, відновлення після катастроф і безперервність бізнесу є легшими та менш витратними, оскільки дані можуть

бути відображені на кількох резервних сайтах у мережі постачальника хмарних послуг.

- Безпека: Постачальники хмарних послуг зазвичай пропонують широкий набір політик, технологій і контролів для зміцнення безпеки, захищаючи дані, програми та інфраструктуру від загроз.

Огляд ринку

Зростання хмарних обчислень пояснюється здатністю хмари значно підвищити продуктивність бізнесу у великих підприємствах, зростаючим попитом на гібридні системи та системи Omnicloud, а також впровадженням моделей оплати за використання. Хмарні сервіси виграли популярності в країнах, що розвиваються, частково завдяки урядовим ініціативам, спрямованим на захист цілісності та безпеки даних. Пандемія COVID-19 прискорила впровадження хмарних обчислень через перехід до гібридних моделей роботи. Хоча проблеми з конфіденційністю та безпекою даних залишаються, великі підприємства все частіше звертаються до хмарних технологій для оптимізації витрат. Крім того, впровадження хмарних технологій зростає серед малих і середніх організацій, а уряди країн, що розвиваються, роблять значні інвестиції в моделі доставки хмарних технологій для підвищення продуктивності. На рисунку 3.3 показано зростання ринку хмарних обчислень.

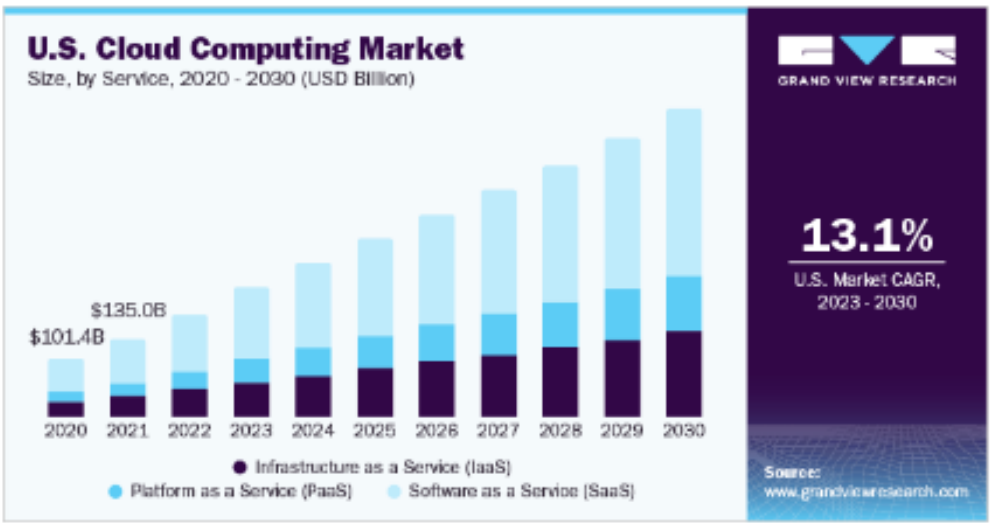

Рис. 3.3 Зростання ринку хмарних обчислень

3.2 Мікросервіси в контексті хмарних обчислень

Характеристики. Архітектура мікросервісів є модульним підходом до розробки програмного забезпечення, розбиваючи складні додатки на менші, незалежні компоненти — мікросервіси — адаптовані до конкретних бізнес-доменів, таких як управління запасами або обробка замовлень.

Внутрішньо мікросервіси інкапсулюють свою функціональність, працюючи через мережеві кінцеві точки та приховуючи деталі реалізації, такі як мови програмування або зберігання даних.

Це забезпечує ефективне управління складністю, при цьому кожен сервіс підтримує своє власне зберігання даних, уникаючи проблем з спільною базою даних. Зовні мікросервіси діють як "чорні ящики", пропонуючи функціональність без розкриття внутрішніх процесів. Цей підхід захищає від впливу внутрішніх змін, якщо інтерфейси залишаються сумісними. Це дозволяє безперебійні оновлення та обслуговування, підтримуючи незалежну розробку та безперервну інтеграцію.

Ключові характеристики мікросервісів включають слабку зв'язність для гнучкості та високу згуртованість для підтримуваності. Вони дозволяють цілеспрямовану масштабованість, паралельну командну роботу та надійну безпеку, при цьому кожен сервіс захищений окремо. Це робить мікросервіси ідеальними для створення адаптивного, масштабованого та стійкого програмного забезпечення в умовах швидко змінюваного бізнесу та технологічного середовища.

Інтеграція мікросервісів та хмарних обчислень є значним прогресом в архітектурі програмного забезпечення, сприяючи динамічним, масштабованим та стійким системам. Децентралізована природа мікросервісів ідеально узгоджується з хмарними середовищами, забезпечуючи гнучкість та масштабовану інфраструктуру для задоволення різних вимог до послуг. Хмарні платформи покращують оптимізацію ресурсів, забезпечуючи ефективні та економічні операції. Ця синергія дозволяє організаціям використовувати надійність хмарних

обчислень, підтримуючи складні взаємодії мікросервісів для підвищення масштабованості та стійкості.

Це дозволяє безперервну інтеграцію та розгортання, сприяючи швидким інноваціям. Крім того, стратегічний розподіл мікросервісів по різних регіонах у хмарі підвищує стійкість до збоїв та забезпечує послідовний досвід користувачів у всьому світі.

Приклад архітектури мікросервісів. Яскравим прикладом мікросервісів у дії є стрімінговий гігант Netflix, який став синонімом успішної реалізації цього архітектурного стилю, є свідченням інноваційного інженерного підходу компанії. Як ми можемо бачити на рис. 3.4, бекенд Netflix є конгломератом мікросервісів, які працюють на Amazon Web Services (AWS), що дозволяє їм надавати величезну кількість контенту по всьому світу з високою доступністю та стійкістю.

Кожен мікросервіс розроблений для виконання конкретної функції, такої як обробка запитів на вхід, обробка рекомендацій для користувачів або управління взаємодією з підтримкою клієнтів.

Цей поділ відповідальностей дозволяє незалежне масштабування та розробку сервісів, що є критично важливим, враховуючи різноманітність контенту Netflix та змінність попиту.

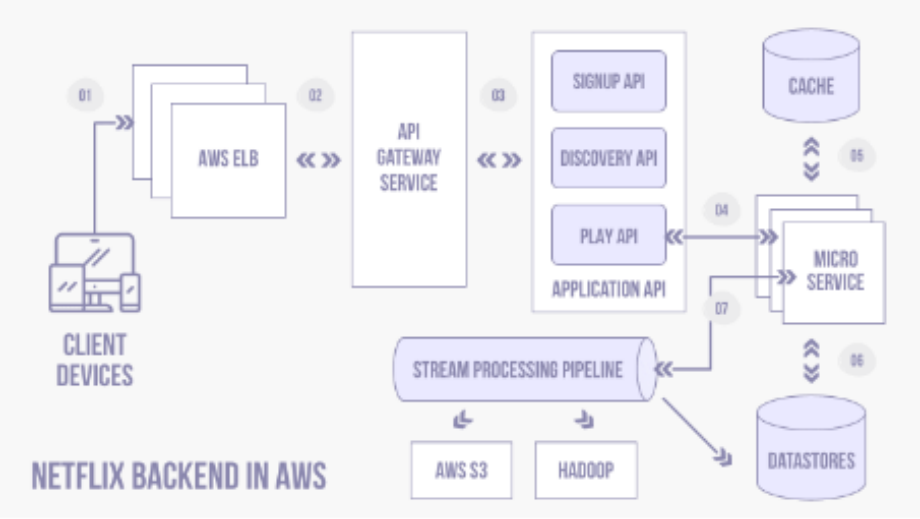

Рис. 3.4. Сервер Netflix в AWS

Архітектура мікросервісів є не лише основним компонентом їхньої бекенд-системи, але й підпирає їхню мережу доставки контенту Open Connect (CDN), забезпечуючи оптимальну продуктивність стрімінгу, розміщуючи сервери в мережах Інтернет-провайдерів (ISP) по всьому світу.

Ця архітектура сприяє швидкій та надійній доставці складних додатків у масштабах, ілюструючи здатність моделі мікросервісів підтримувати великомасштабні корпоративні системи ефективно та результативно.

Масштабованість

Мікросервіси відзначаються масштабованістю завдяки своїй здатності масштабуватися індивідуально. Ця детальна масштабованість дозволяє точно розподіляти ресурси між різними компонентами на основі коливань попиту, що призводить до підвищення ефективності використання ресурсів.

На відміну від монолітних архітектур, де масштабування часто вимагає масштабування всього застосунку, мікросервіси працюють незалежно.

Ця незалежність сприяє безперешкодному додаванню, видаленню, оновленню або масштабуванню кожної служби без переривання роботи решти системи. Організації отримують вигоду від цього, маючи можливість динамічно розподіляти ресурси мікросервісам, які зазнають сплесків попиту — наприклад, під час пікових сезонів покупок — і, аналогічно, зменшувати їх, коли попит знижується, тим самим оптимізуючи використання ресурсів і обчислювальної потужності в усій службі.

Надійність

Архітектура мікросервісів підвищує надійність програмних застосунків, використовуючи свою вроджену декомпозицію. Окремі служби можуть зазнати збою без того, щоб це призвело до зупинки всієї системи, запобігаючи виникненню єдиної точки відмови, яка може викликати каскадні збої. У порівнянні з цим, монолітні архітектури піддаються доміно-ефекту, коли збій одного компонента може паралізувати весь застосунок. Мікросервіси вбудовані для відмови, дозволяючи системі деградувати плавно і підтримувати функціональність навіть тоді, коли певні служби не працюють.

Однак збої в мережі та на машинах є неминучими, і повинні бути розроблені стратегії для обробки цих інцидентів без значного впливу на досвід користувачів.

Технологічна нейтральність

Архітектура мікросервісів вирізняється своєю технологічною гнучкістю, надаючи командам свободу вибору найбільш підходящого технологічного стеку для кожної окремої служби. Цей технологічно нейтральний підхід декомпонує служби від будь-яких єдиних, ранніх технологічних рішень, які часто обмежують цілі проекти. У межах цієї парадигми кожен мікросервіс може бути розроблений з використанням різних мов програмування та рішень для зберігання даних, відповідно до того, що найкраще відповідає його меті.

Розподілена розробка

Архітектура мікросервісів дозволяє командам розробників незалежно створювати, розгортати та управляти своїми сервісами, прискорюючи оновлення та додавання функцій з мінімальними перервами для загальної системи. Це сприяє швидкій адаптації до змінюваних бізнес-потреб. На відміну від монолітних додатків, які вимагають масштабних розгортань для незначних оновлень, мікросервіси підтримують цілеспрямовані, незалежні зміни до конкретних сервісів.

. *Оптимізація команди*

Архітектура мікросервісів підвищує продуктивність команди, дотримуючись "правила двох піц", де менші команди — ідеально достатньо великі, щоб їх нагодувати двома піцами — зазвичай виробляють результати вищої якості завдяки покращеній концентрації та керованості. Цей підхід, започаткований Amazon, забезпечує, що кожна команда працює над окремою кодовою базою, сприяючи ефективності та швидшому досягненню цілей. Гнучкість, властива мікросервісам, також дозволяє легко переназначати власність на сервіси, сприяючи безперешкодній адаптації архітектури до змінюваних організаційних структур, тим самим підтримуючи ефективність і результативність у довгостроковій перспективі.

Виклики

Хоча архітектура мікросервісів пропонує численні переваги, такі як підвищена масштабованість і покращена продуктивність команди, вона також вводить набір викликів, які можуть ускладнити проектування та обслуговування системи: складність оркестрації численних сервісів, підтримка узгодженості даних у розподілених системах і ефективне управління міжсервісною комунікацією. Вирішення цих викликів є необхідним для безперебійної архітектури мікросервісів.

Складність

Децентралізований підхід мікросервісів за своєю суттю призводить до систем з високим ступенем складності. Зі збільшенням кількості сервісів загальна система може стати більш складною для контролю та управління.

Виправлення помилок ілюструє цю складність; оскільки кожен мікросервіс генерує свої журнали, визначити джерело проблеми може стати суттєвим викликом. Ця складність вимагає надійних рішень для ведення журналів і моніторингу, які можуть агрегувати та корелювати журнали з різних сервісів, надаючи цілісне уявлення про стан системи та сприяючи швидшому вирішенню проблем.

Крім того, складність вимагає, щоб розробники та оператори мали чітке розуміння архітектури системи та комунікаційних патернів, що забезпечує їх здатність ефективно відстежувати та усувати проблеми, коли вони виникають.

Узгодженість даних

Забезпечення узгодженості даних між мікросервісами становить значні виклики через їх розподілену архітектуру. На відміну від монолітних систем, які покладаються на єдину базу даних, мікросервіси часто використовують окремі бази даних, що ускладнює підтримку традиційної узгодженості на основі транзакцій.

В результаті, розробникам потрібно перейти до таких патернів, як саги, і прийняти остаточну узгодженість, що може бути значною зміною парадигми, особливо при адаптації існуючих систем. Важливо поступово декомпонувати

програми, що дозволяє ретельно оцінити вплив кожної зміни на цілісність даних системи.

Комунікація між сервісами

В архітектурі мікросервісів, особливо в хмарних середовищах, комунікація між сервісами додає складності через використання розподілених мереж.

Унікальний API кожного мікросервісу вимагає ретельного управління для забезпечення сумісності, що є значним завданням, коли залучено сотні або тисячі API. Розподіл процесів на кілька мережевозалежних сервісів збільшує серіалізацію, передачу та десеріалізацію, що потенційно додає до затримки. Цей вплив на продуктивність, важко передбачити на етапі проектування або розробки, підкреслює необхідність поступового переходу до мікросервісів, що дозволяє оцінити зміни в затримці системи.

На рисунку 3.5 ілюструється відмінність між підходом мікросервісів і монолітною архітектурою. Останній характеризується тісно пов’язаними та взаємозалежними компонентами програмного забезпечення, будь-які зміни вимагають створення та розгортання всього стеку, що може бути повільним і схильним до помилок. Мікросервіси розроблені для подолання цих обмежень шляхом декомпозиції функціональності на окремі сервіси, кожен з яких має конкретну роль, таким чином забезпечуючи більш гнучку та масштабовану архітектуру.

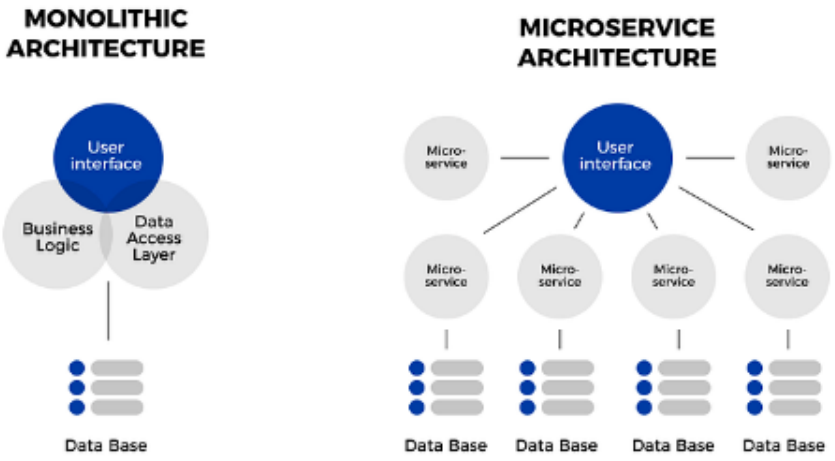

Рис. 3.5 Монолітні та мікросервісні архітектури

Таблиця 3.2

Огляд відмінностей між двома підходами

	Монолітна	Мікросервісна
Розгортання	Просте та швидке розгортання всієї системи	Вимагає окремих ресурсів, що ускладнює організацію розгортання
Масштабованість	Важко підтримувати та обробляти нові змінні; вся система повинна бути перерозгорнута	Кожен елемент можна масштабувати незалежно без простоїв
Спритність	Негнучкий і неможливий для впровадження нових технологій, мов або фреймворків	Інтеграція з новими технологіями для вирішення
Стійкість	Одна помилка чи проблема може вплинути на всю систему	Збій в одній мікрослужбі не впливає на інші служби
Безпека	Комунікація в одному пристрої забезпечує безпечну обробку даних	Для міжпроцесного зв’язку потрібні шлюзи API, що створює проблеми з безпекою
Розвиток	Неможливо розподілити зусилля команди через величезну неподільну базу даних	Над кожним компонентом може працювати команда розробників незалежно один від одного

3.3 Як моделювати мікросервіси

Наша мета – розробити мікросервіси, які можуть бути незалежно модифіковані, розгорнуті та мати свої функції, доступні користувачам без залежності від інших. Здатність оновлювати один мікросервіс незалежно від інших є вирішальною. У своїй основі, мікросервіси представляють тип модульної декомпозиції, але з мережевими взаємодіями між модулями. Це означає, що ми можемо покладатися на багато передових технологій у сфері модульного програмного забезпечення, щоб допомогти у визначенні наших меж. Маючи це на увазі, ми заглибимося в три основні концепції, важливі для визначення

ефективних меж мікросервісу: приховування інформації, згуртованість і зв’язок [25].

Приховування інформації

Зв'язки між модулями — це припущення, які модулі роблять один про одного. Зменшення припущень між модулями в мікросервісах спрощує їхні зв'язки, що полегшує модифікацію одного модуля без впливу на інші. Цей підхід також дозволяє розробникам вносити більш безпечні зміни, оскільки вони розуміють, як їхній модуль використовується іншими, запобігаючи необхідності змін у верхніх компонентах. Крім того, у мікросервісах такі модифікації можуть бути розгорнуті незалежно, підвищуючи переваги, викладені Парнасом: швидше розроблення, краща зрозумілість і підвищена гнучкість.

Зв'язність і когезія

Концепції зв’язку та згуртованості є невід’ємною частиною структури та стабільності архітектур мікросервісів. Розуміння їх взаємодії допомагає розробляти стабільні та ефективні системи. Досягнення правильного балансу між цими двома аспектами має вирішальне значення для ефективного функціонування мікросервісів.

- Згуртованість: згуртованість у мікросервісах — це стратегічне групування пов’язаних бізнес-функціональних можливостей, щоб зменшити потребу в змінах у багатьох сферах. Він наголошує на консолідації схожих моделей поведінки в одному місці, що спрощує процес модифікації та розгортання. Такий підхід призводить до сильної згуртованості, де тісно пов’язані функції містяться в одному мікросервісі, що забезпечує швидші та безпечніші оновлення та зміни.

- З’єднання: З’єднання в контексті мікросервісів передбачає проектування служб таким чином, щоб зміни в одній не потребували модифікацій в інших. Цей принцип розробки сприяє мінімальній обізнаності між службами, тим самим зменшуючи залежність між різними службами. Ідеальний стан слабкого з’єднання досягається, коли служби мають мінімальні взаємодії один з одним, зберігаючи свою незалежність. Такий підхід значно знижує ризики, пов’язані з тісно взаємопов’язаними системами, забезпечуючи більш надійну та гнучку

архітектуру сервісу. Цей баланс стосується не лише технічних аспектів, а й прийняття прагматичних рішень, які відповідають конкретному контексту та викликам проекту.

Типи зв'язку

Концепція зв'язку в системному дизайні є нюансованою і не такою простою, як може здаватися на перший погляд. Хоча це правда, що надмірний зв'язок може призвести до різних проблем у системній архітектурі, певний рівень зв'язку є неминучим і, в деяких випадках, необхідним. Ключовою метою в ефективному системному дизайні є не повне усунення зв'язку, а управління та мінімізація його обсягу.

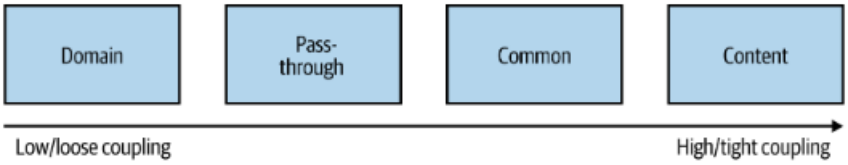

Рис. 3.6 Різні типи зв'язку, від слабкого (низького) до жорсткого (високого)

Різні типи зв’язків [23], як зображено на рисунку 3.6, забезпечують широкий спектр, починаючи від низького до високого. Низький зв’язок, як правило, бажаний, оскільки він вказує на систему, де компоненти працюють незалежно, підвищуючи гнучкість і легкість обслуговування. Високий зв’язок, з іншого боку, припускає тісно взаємопов’язану систему, де зміни в одному компоненті можуть мати значні пульсаційні ефекти, що робить його менш бажаним через підвищену складність і пов’язаний ризик. Розуміння цих варіацій та їх наслідків має вирішальне значення для проектування надійних, масштабованих і зручних систем.

- Взаємозв'язок доменів: Зв'язок між доменами відноситься до сценарію, коли один мікросервіс залежить від іншого за певними функціональними можливостями. Хоча такі взаємодії здебільшого неминучі в архітектурі мікросервісів, де співпраця між декількома сервісами необхідна для роботи системи, важливо мінімізувати ці взаємодії. Це є формою слабкого зв'язку в мікросервісах, але може призвести до проблем, якщо сервіс покладається занадто

сильно на багато нижчих сервісів, що свідчить про надмірну централізацію логіки. Проблеми також можуть виникати при обміні складними наборами даних між сервісами. Бажано обмінюватися лише найнеобхіднішою інформацією і мінімізувати обмін даними.

- Наскрізне з’єднання: наскрізне з’єднання в мікросервісах відбувається, коли один мікросервіс передає дані іншому виключно для використання наступним сервісом нижче. Ця форма зв’язку є особливо складною в рамках стратегій впровадження, оскільки передбачає, що ініціюючий сервіс усвідомлює не лише пряму мікросервіс одержувача, але може також знадобитися зрозуміти функціонування мікросервісу далі по ланцюжку. Це створює складну взаємозалежність, коли знання про численні служби та їх взаємодію стає необхідним, що ускладнює архітектуру

- Загальний зв'язок: Загальний зв'язок у мікросервісах відноситься до сценарію, коли кілька служб використовують один і той же набір даних, наприклад, спільну базу даних, пам'ять або файлову систему. Такий зв'язок стає проблематичним, коли зміни в даних впливають на декілька сервісів одночасно. Наприклад, якщо схема спільно використовуваної бази даних змінюється несумісно, всі сервіси, що покладаються на неї, потребують оновлення. Крім того, спільне використання може призвести до конфлікту ресурсів, оскільки декілька сервісів, які отримують доступ до однієї бази даних або файлової системи, можуть перевантажити або навіть вивести з ладу цей ресурс. Хоча іноді з цим можна впоратися, загальне з'єднання часто вказує на брак згуртованості в системі і може створювати операційні проблеми, що робить його однією з найменш бажаних форм зв'язку.

- Зв’язок вмісту: зв’язок вмісту відбувається, коли вищестояща служба нав’язливо змінює внутрішній стан нижчої служби, зазвичай шляхом прямого доступу та зміни бази даних останньої. Це дещо відрізняється від звичайного зчеплення, коли кілька служб взаємодіють із спільним набором даних, але визнають це як зовнішню, неконтрольовану залежність. Зв’язок вмісту стирає межі власності, ускладнюючи модифікацію системи для розробників. Чітке

розмежування в мікросервісах між змінними та незмінними елементами має вирішальне значення. Розробники повинні бути в курсі контракту на обслуговування, доступного зовнішнім сторонам, щоб уникнути зриву споживачів на першому етапі. Хоча загальне зчеплення має деякі проблеми з зчепленням вмісту, останнє створює додаткові ускладнення, які часто називають патологічним зчепленням. Прямий зовнішній доступ до бази даних ставить під сумнів визначення того, що можна безпечно змінювати, а що ні, підриваючи принцип приховування інформації. Тому краще уникати зчеплення вмісту через ці властиві ускладнення.

Проєктування, орієнтоване на домен

Визначаючи межі мікросервісу, ми в першу чергу зосереджуємося на самому домені, застосовуючи доменно-орієнтоване проектування (domain-driven design, DDD), щоб моделювати наш домен більш ефективно. DDD, представлений Еріком Евансом у «Дизайні, керованому доменом»[22], містить ключові поняття, які є вирішальними в цьому контексті. До них належать Ubiquitous Language, яка забезпечує однакове використання мови в домені; Агрегати, які групують пов’язані об’єкти домену в єдиний блок; і обмежений контекст, який встановлює сферу застосування для конкретної моделі. Ці принципи відіграють важливу роль у стратегії архітектури мікросервісів. Усюдисуща мова підкреслює важливість узгодження термінології в нашому коді з термінами, які використовують користувачі. Ця спільність мови між командою розробників і кінцевими користувачами спрощує моделювання реального домену та покращує спілкування. Інтеграція мови реального світу в код спрощує процес розробки. Це дозволяє розробникам під час виконання завдань або історій швидко зрозуміти вимоги та цілі, оскільки вони виражені термінами, знайомими як власнику продукту, так і команді розробників.

Агрегати

Агрегати в архітектурі мікросервісів розглядаються як самодостатні одиниці, кожна з яких має власний стан, ідентичність і життєвий цикл, які віддзеркалюють сутності реального світу. Ці агрегати підходять для реалізації у

вигляді машин станів, враховуючи притаманні їм життєві цикли. Дизайн фокусується на консолідації коду, який керує переходами між станами, з самим станом агрегату. Зазвичай один мікросервіс відповідає за один агрегат, але він може обробляти декілька. Наприклад, зведений рахунок-фактура включатиме різні рядкові платежі, кожен з яких важливий лише в контексті загального зведеного рахунку-фактури.

Роль мікросервісу поширюється на управління життєвим циклом і зберігання даних одного або декількох типів агрегатів. Якщо іншій службі потрібно змінити агрегат, вона повинна або безпосередньо запросити цю зміну або спонукати агрегат ініціювати власні переходи станів, можливо, у відповідь на події від інших мікросервісів. Агрегати розроблені з можливістю здатністю відхиляти невідповідні запити на зміну стану, що підкреслює важливість запобігання незаконним змінам станів при їх виконанні.

Обмежений контекст

Обмежений контекст зазвичай відображає більшу частину організації з чіткими обов’язками в її межах. Ця концепція зосереджена на приховуванні дрібних деталей реалізації, зберігаючи внутрішні аспекти, які не є необхідними для зовнішнього розуміння чи участі. З точки зору структури, обмежений контекст складається з одного або кількох агрегатів. Хоча деякі з цих агрегатів можуть бути видимими ззовні, інші залишаються внутрішніми, щоб зберегти цілісність контексту. Обмежені контексти також можуть формувати зв’язки з іншими контекстами, перетворюючись на залежності між службами в архітектурі мікросервісу.

Наприклад, складську послугу можна розглядати як обмежений контекст, який насичений такими діями, як обробка вихідних замовлень, отримання нових запасів та інші логістичні завдання. В іншому обмеженому контексті, наприклад у фінансовому відділі, увага зміщується на менш динамічні, але однаково важливі функції, такі як управління заробітною платою та обробка фінансових операцій. Кожен контекст працює в межах своєї власної сфери обов’язків, але може

взаємодіяти з іншими контекстами або залежати від них, відображаючи взаємопов’язаний характер служб у налаштуванні мікросервісів.

Доменно-керований дизайн у мікросервісах

Доменно-керований дизайн (DDD) ефективний в архітектурі мікросервісів завдяки його зосередженості на обмежених контекстах, які приховують внутрішні складності та представляють чіткі межі системи. Ці контексти допомагають підтримувати стабільні межі мікросервісу, гарантуючи, що внутрішні зміни не впливають на інші частини системи. Коли системи сегментовані вздовж обмежених контекстів, модифікації для бізнес-потреб обмежуються окремими мікросервісами, що спрощує розгортання та зменшує складність змін.

І агрегати, і обмежені контексти забезпечують згуртованість одиниць із чіткими інтерфейсами до більшої системи. Агрегати є зосередженими кінцевими автоматами для концепцій однієї області, тоді як обмежені контексти групують ці агрегати та представляють їх зовнішньому світу.

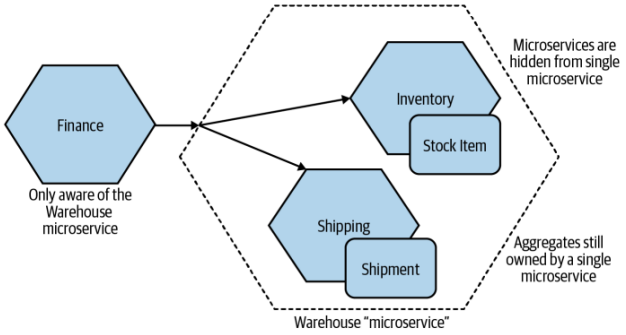

Рис. 3.7 Служба Warehouse внутрішньо було розділено на мікросервіси Inventory та Shipping

Ці конструкції ідеально підходять для визначення меж мікросервісів. Спочатку корисно працювати зі службами, які охоплюють повні обмежені контексти. У разі потреби послуги можна згодом розділити на менші без поділу на окремі агрегати, залишаючи такі внутрішні рішення непомітними для зовнішніх зацікавлених сторін. Наприклад, служба Warehouse внутрішньо може бути розділена на Inventory та Shipping, але зовні вона залишається окремою мікрослужбою Warehouse для користувачів, як показано на рисунку 3.7.

Керування робочим процесом. У середовищі мікросервісів складність взаємодії виходить за межі простого спілкування між двома сервісами. Критичним аспектом є оркестровка кількох мікросервісів, які працюють разом для виконання комплексних бізнес-процесів. Ця оркестровка вимагає тонкого підходу для підтримки цілісності та ефективності системи.

3.4 Хмарний обмін повідомленнями

Firebase Cloud Messaging (FCM)[30] це потужне безкоштовне хмарне рішення для повідомлень на iOS, Android і веб-додатках. Він забезпечує надійне та ефективне з'єднання між серверами та пристроями, що дозволяє доставляти сповіщення або повідомлення. FCM пропонує різноманітні параметри обміну повідомленнями, включаючи обмін повідомленнями за тематикою, який дозволяє надсилати повідомлення на кілька пристроїв, які вибрали певну тему, груповий обмін повідомленнями пристроїв, що дозволяє надсилати повідомлення пристроям, які належать до групи, і прямий обмін повідомленнями на окремі пристрої. Ця масштабованість робить його необхідним інструментом для розробників, які прагнуть ефективно залучати свою базу користувачів, з додатковими перевагами аналітики та відстеження продуктивності.

Як це працює. Firebase Cloud Messaging використовує архітектуру клієнт-сервер, де сервер FCM відповідає за обробку та маршрутизацію повідомлень. Клієнтська програма на пристрої користувача спілкується з FCM через SDK, який керує процесом реєстрації та генерацією маркерів. Цей маркер унікально ідентифікує екземпляр програми та забезпечує безпечну доставку повідомлень на пристрій. Повідомлення, які надсилаються із сервера розробника до серверної частини FCM, можуть залежати від корисного навантаження, керуючи FCM надсилати їх як сповіщення або повідомлення даних. Потім FCM оптимізує доставку повідомлень, ставлячи їх у чергу, керуючи пріоритетом і навіть об'єднуючи повідомлення для ефективності мережі. Ця архітектура підтримує

високий рівень масштабованості та надійності доставки повідомлень між платформами та пристроями по всьому світу.

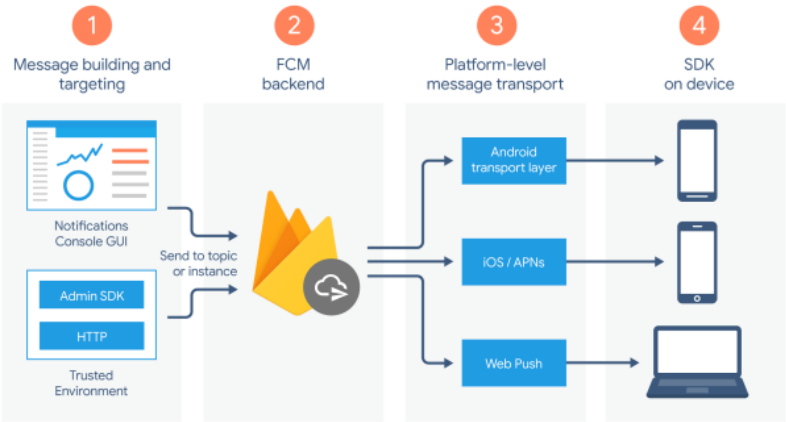

Рис. 3.8 Архітектура FCM

Рисунок 3.8 показує операційну архітектуру FCM:

1. Інструменти для створення запитів на повідомлення, включаючи графічний інтерфейс через редактор сповіщень для сповіщень і серверні середовища, як-от Cloud Functions для Firebase або App Engine, які підтримуються Firebase Admin SDK або серверним протоколом FCM, для комплексної обробки типів повідомлень.

2. Центральний сервер FCM, який обробляє вхідні запити на повідомлення, керує розповсюдженням повідомлень за темами та призначає метадані, як-от ідентифікатори повідомлень.

3. Транспортна система на рівні пристрою, яка забезпечує надходження повідомлень до місця призначення в залежності від платформи: пристрої Android, пристрої Apple і веб-протоколи push для браузерів.

4. FCM SDK, який знаходиться на пристрої кінцевого користувача та відповідає за відображення сповіщень або обробку повідомлень, залежно від стану програми та настроюваних логік.

3.5 Концептуальна модель налаштування для SaaS із кількома клієнтами

На рисунку 3.9 узагальнено основні поняття, пов’язані з налаштуванням SaaS із кількома клієнтами.

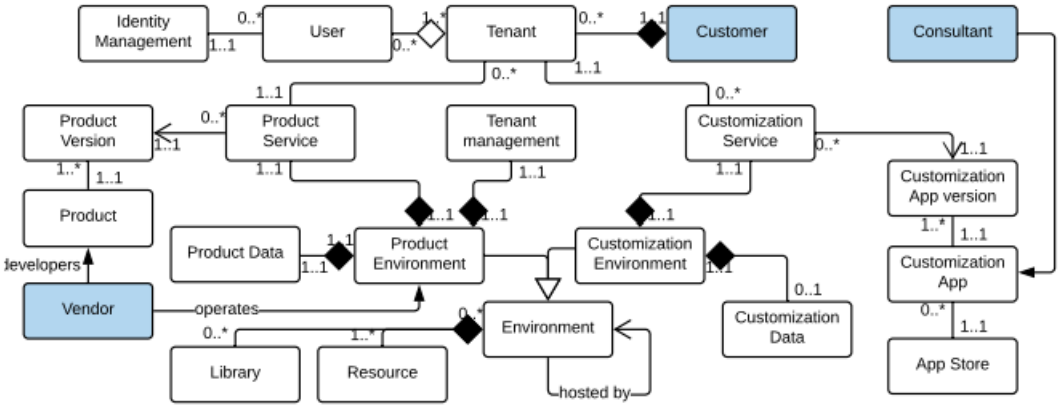

Рисунок 3.9 Концептуальна модель налаштування продукту для багатокористувацького SaaS

Є три різні ролі в екосистемі налаштування SaaS для багатьох клієнтів, тобто Постачальники, які надають основний SaaS, Клієнти, які підписуються на послугу, і Консультанти, яких наймають клієнти для налаштування SaaS. На практиці ці три ролі не обов’язково виконуються різними компаніями, наприклад, компанія-замовник може мати власну IT-команду та розробників, які компетентні виконувати налаштування. Компанія-продавець також може мати власну команду консультантів, яка продає власні послуги та виконує налаштування відповідно до запиту клієнта.

Продукт — це програмне забезпечення, вироблене постачальником. Він може мати кілька версій.

Сервіс продукту — це запущений екземпляр певної версії продукту, розміщений у середовищі продукту разом із даними продукту. Середовище продукту – це конкретне середовище, яке є самостійним обчислювальним блоком із програмним забезпеченням, бібліотеками та ресурсами, як-от контейнер Docker

або віртуальна машина. Служба продукту зазвичай працює в ексклюзивному середовищі, тобто одне середовище призначене для одного й лише одного екземпляра. Середовище може бути розміщено в іншому середовищі, наприклад, контейнер Docker може бути розміщено у віртуальній машині, а остання розміщена в хмарній інфраструктурі.

Одна служба продукту обслуговує кількох орендарів, і в той же час постачальник керує середовищем продукту. Постачальник зазвичай керує кількома службами продукту (і, отже, кількома середовищами продукту) одночасно: їм може знадобитися підтримувати екземпляри для кількох версій продукту для своїх клієнтів. Для однієї версії вони зазвичай запускають один основний екземпляр для використання у виробництві, один проміжний екземпляр для тестування користувачами та один екземпляр розробки для тестування та налагодження. Навіть у межах використання продукту може знадобитися кілька екземплярів продукту через обмеження навантаження або регіону. За таких налаштувань клієнт може мати підписку на кілька екземплярів продукту, кожен з яких підписується через іншого клієнта. Іншими словами, кожен орендар належить лише одному екземпляру продукту. Це спрощує виставлення рахунків і керування для постачальників. Кожен орендар охоплює певну кількість користувачів, які є особами, які мають право доступу до примірника продукту через цього орендаря. Зазвичай вони є працівниками клієнта, який стоїть за цим орендарем, але орендар також може включати користувачів із консалтингової компанії, які допомагають клієнту шляхом навчання, обслуговування або налаштування.

Програма налаштування – це програмний код, який реалізує налаштування продукту. Він може мати кілька версій. Служба налаштування – це запущений екземпляр однієї версії програми налаштування. Сервіс налаштування розміщується в середовищі налаштування. Подібно до середовища продукту, середовище налаштування також надає необхідні ресурси, бібліотеки та базу даних для запуску служби налаштування. База даних необов’язкова, оскільки деякі служби легкого налаштування можуть не мати бази. Служба налаштування

зареєстрована на орендаря, і вона змінює лише поведінку продукту для цього конкретного орендаря.

Розглянемо приклад. Магазини з кількома клієнтами, який відображає вимоги до налаштування. Він пропонує інтернет-магазини SaaS: клієнти можуть швидко створити власний веб-сайт для покупок. З точки зору постачальника програмного забезпечення магазину, кожен клієнт є орендарем з окремим веб-сайтом, на якому їхні кінцеві користувачі можуть переглядати та купувати товари.

Магазин має бути настроюваним. Наприклад, один із їхніх ключових клієнтів/орендарів вимагає, щоб їхній кошик для покупок містив опцію благодійної пожертви. Кожного разу, коли кінцевий користувач додає товар у свій кошик для покупок, він може пожертвувати певну суму грошей визначеній благодійній організації, що зрештою додає загальну вартість покупки.

Магазин наймає стороннього консультанта для реалізації цього налаштування, яке передбачає наступні зміни стандартного продукту Магазину. Їм потрібно змінити: (1) сховище бази даних, щоб мати можливість записувати суму пожертви для кожного товару в кошику для покупок, (2) бізнес-логіку для обліку цих пожертвувань, і (3) нарешті, інтерфейс користувача для завершення - користувачі можуть вибрати/побачити, скільки вони жертвують.

Як багатокористувацький SaaS, Магазин не може дозволити консультанту змінювати їх вихідний код продукту для реалізації такого налаштування, оскільки та сама схема бази даних і вихідний код обліку цін використовуються кількома орендарями. Змінення коду для одного орендаря заважало б надавати послуги іншим орендарям. Натомість послуга продукту Магазину має надавати лише стандартні функції, які є загальними для всіх орендарів. Налаштування, які вимагає саме конкретний магазин, мають виконуватися ізольовано, поза службою продукту. Після реєстрації в службі продукту це налаштування має змінювати поведінку служби продукту, як зазначено вище, тоді й лише тоді, коли запити користувачів прив’язані до цього клієнта.

Приклад ілюструє проблему, з якою стикаються багато постачальників SaaS: їхні послуги успішні для деяких клієнтів, лише якщо вони можуть

виконувати глибокі налаштування. Але постачальник не може дозволити такий самий спосіб глибокого налаштування, як для локальних продуктів, оскільки він повинен підтримувати службу з кількома клієнтами. Таким чином, їм потрібне рішення для налаштування, яке забезпечує як ізоляцію - налаштування для одного орендаря не повинні впливати на інших орендарів. Орендар ізоляція, особливо з точки зору безпеки, має першочергове значення. Так і асиміляцію - налаштування не повинно шкодити продуктивності та досвіду користувача SaaS. Іншими словами, «вигляд і відчуття» SaaS з налаштуваннями не повинні змінюватися порівняно з оригінальним SaaS.

3.6 Динамічне налаштування за допомогою мікросервісів

В літературі експериментували з підходом до використання (напів)інтрузивних мікросервісів для налаштування мультитенантного SaaS [22, 2]. Було дозволено орендарям замінювати детальні структури, тобто будь-які частини інтерфейсу користувача (UI), бізнес-логіки (BL) або бази даних (DB) у вихідному продукті зовнішніми мікросервісами. Основна логіка налаштування працює в цих мікросервісах як паралельні стеки за межами основного продукту. Однак, коли логіці налаштування потрібно отримати доступ до даних продукту або маніпулювати станом продукту, вона надсилає так званий «код зворотного виклику» до продукту, а останній динамічно інтерпретує код зворотного виклику під час виконання. Оскільки код зворотного виклику працює в тому ж контексті, що й замінений основний код продукту, теоретично він має еквівалентну силу, як і основний код продукту, що означає, що він може отримати доступ до будь-яких даних і змінити будь-які стани, доступні за допомогою коду товару. Таким чином, це забезпечує глибоке налаштування, оскільки те, що можна налаштувати, не обмежується API основного продукту. Цей спосіб не вимагає від основного продукту надання будь-яких спеціальних API для цілей налаштування.

Адаптація SaaS до можливості налаштування

Ми адаптували вихідний код Магазину з конкретним товаром, щоб увімкнути налаштування на основі мікросервісів, додавши загальну бібліотеку та виконавши переписування коду. Простий менеджер орендарів використовується для реєстрації відображення вихідних методів основного коду в код налаштування. На рис. 3.10 показано, як код налаштування взаємодіє з основним потоком коду основного продукту.

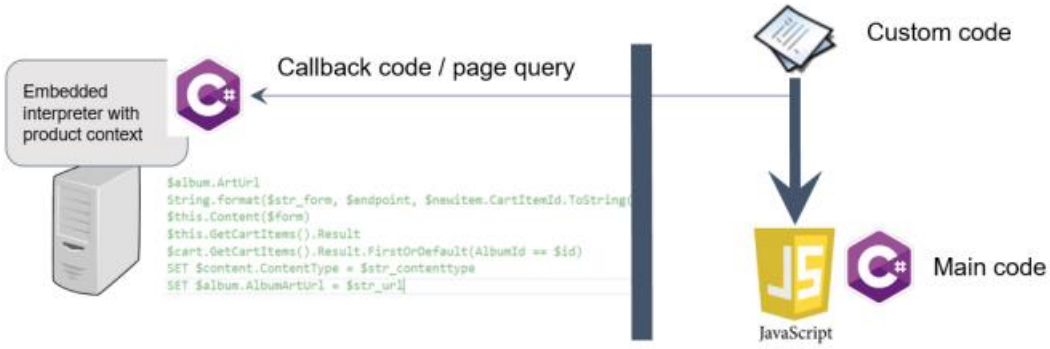

Рис. 3.10 Адаптація SaaS до можливості налаштування

Загальний перехоплювач забезпечує синхронне спрацьовування між основним продуктом і кодом унікального продукту. Перехоплювач надсилає поточний контекст виконання з основного продукту до відповідного мікросервісу унікального продукту для виконання логіки унікального продукту. Після виконання логіки налаштування мікросервіс налаштування надсилає код зворотного виклику до основного продукту, щоб застосувати налаштування та навіть запитувати інший контекст від основного продукту, якщо необхідна подальша логіка налаштування. Інтерпретатор коду зворотного виклику виконує нав’язливий код зворотного виклику в локальному контексті, щоб застосувати налаштування в основному коді.

Перед створенням і розгортанням програми, наприклад, MusicStore, як служби, ми виконали автоматичний перепис коду, щоб увімкнути механізми запуску та коду зворотного виклику. Зокрема, ми додаємо три частини коду на початку кожного методу. Лістинг 1 нижче показує приклад такого доданого коду для методу AddToCart у класі ShoppingCartController.

Лістинг 1 Запуск налаштування продукту

```
// first piece of code
var context = new Dictionary<string, object>()
{
    ["id"] = id,
    ["cart"] = cart,
    ["this"] = this
};
// second piece of code
ReturnValue rv = Interceptor.Intercept(
    Controllers.TenantController.currentUser,
    "MusicStore.Controllers.ShoppingCartController.AddToCart",
    false,
    context
);
// third piece of code
if(rv != null)
{
    return rv.Value;
}
```

Перший ініціалізує локальний контекст як об’єкт Dictionary і заповнює його методом параметри. Цей контекстний словник пізніше використовуватиметься інтерпретатором коду зворотного виклику. Другий викликає загальний перехоплювач з контекстом і назвою методу. Третій перевіряє, чи доступне значення, що повертається, щоб вирішити, чи слід пропустити оригінальне значення методу та повернути результати налаштування.

Для реалізації підтримки глибоких налаштувань у застарілому програмному забезпеченні потрібні дуже незначні зусилля. Зусилля зосереджено на загальних механізмах без конкретного розгляду фактичних вимог до налаштування або функцій.

Зразок налаштування

У настроюваному Магазині з конкретною послугою, ми виконали три сценарії використання налаштування.

Пожертвування: нам потрібно додати нову сторінку для кінцевих користувачів, щоб вибрати пожертву, новий стовпець у таблиці кошика для покупок, щоб відобразити пожертви, і нову бізнес-логіку обчислення загальної ціни для кошика для покупок.

Підрахунок відвідувань: ми хочемо записати, скільки разів відвідали кожен альбом. Така функція має запускатися кожного разу, коли завантажується перегляд деталей альбому.

Справжня обкладинка: ми хочемо використовувати назву альбому для пошуку зображення обкладинки в Bing Image2 і замінити оригінальне зображення-заповнювач.

Ми навмисно розробляємо ці варіанти використання, щоб досягти гарного покриття загальних вимог до налаштування веб-систем підприємства. На рівні інтерфейсу користувача вони охоплюють зміни всередині веб-сторінки, тобто додавання, видалення або зміну позиції елементів керування HTML (компонентів інтерфейсу користувача, таких як текст, кнопка, список, зображення тощо), а також додавання нової сторінки. Третій варіант використання також змінив логіку виконання браузером. На рівні бізнес-логіки вони охоплюють необхідність додавання або перевизначення логіки на стороні сервера, перевизначення дії для певних подій, зміни зв’язків між елементами керування інтерфейсу користувача та даними та виконання послуг, які надаються третьою стороною. На рівні бази даних їм потрібні нові таблиці, а також нові поля для існуючої таблиці. Ці вимоги підсумовуються на основі фактичних випадків налаштування локальних продуктів, наданих двома компаніями.

Ми реалізували три сценарії налаштування в TypeScript, використовуючи Node.js

HTTP-сервер для розміщення мікросервісу налаштування [22, 2]. Перші два сценарії потребують зберігання даних, і ми використовували MongoDB як базу даних клієнтів. Node.js і MongoDB працюють у двох контейнерах Docker, розміщених на тому ж вузлі, що й служба продукту. Весь код налаштування містить 384 рядки коду в п’яти файлах TypeScript (по одному файлу для кожного сценарію, а також два загальних файли для налаштування та запуску HTTP-сервера) і 175 рядків нового коду в чотирьох шаблонах HTML Razor (з яких два шаблони нові, а два інші скопійовані та вставлені з MusicStore, із 176 рядками коду, які не змінюються).

3.7 Децентралізоване налаштування за допомогою мікросервісів

Незважаючи на остаточну асиміляцію, яка означає, що орендарі можуть робити що завгодно для налаштування, як якщо б вони були розробниками основного продукту, наш інтрузивний підхід до налаштування мікросервісів остаточно не був прийнятий постачальниками програмного забезпечення, хто доручив це дослідження. Головна проблема – безпека. Оскільки партнери практично здатні робити все з основним продуктом під час виконання, це вимагає суворої перевірки постачальниками коду налаштування, що наразі не є прагматичним. У результаті ми звернулися до ненав'язливого способу налаштування на основі мікросервісів, який має дозволити постачальникам тримати під контролем код налаштування орендарів [14, 15].

Щоб зробити можливим ненав'язливе налаштування, існує головна передумова для веб-архітектури SaaS, тобто чітке відокремлення частини інтерфейсу користувача (UI) від частини серверної бізнес-логіки (BL) [15]. Це означає, що SaaS на основі Інтернету має бути розділений на частину WebUI та серверну службу(и) BL. Розділивши UI та BL основного продукту, ми можемо представити мікросервіси, які реалізують індивідуальні налаштування для основного продукту на рівні UI, BL та бази даних (DB). Зверніть увагу, що ми зосереджуємося на веб-платформі SaaS через її популярність. На рис. 3.11 показано огляд ненав'язливого підходу. Кожне налаштування для клієнта працює як окрема мікрослужба та динамічно реєструється в основній службі продукту для цього клієнта. API мікросервісу налаштування доступні для авторизованого доступу через шлюз API. Під час виконання, коли основна служба продукту досягає зареєстрованої точки налаштування для клієнта, основна служба продукту запускає відповідну настройку для цього клієнта, викликаючи REST API мікросервісів налаштування клієнта через шлюз API. Зверніть увагу, що мікросервіси налаштування були заздалегідь зареєстровані в менеджера орендарів.

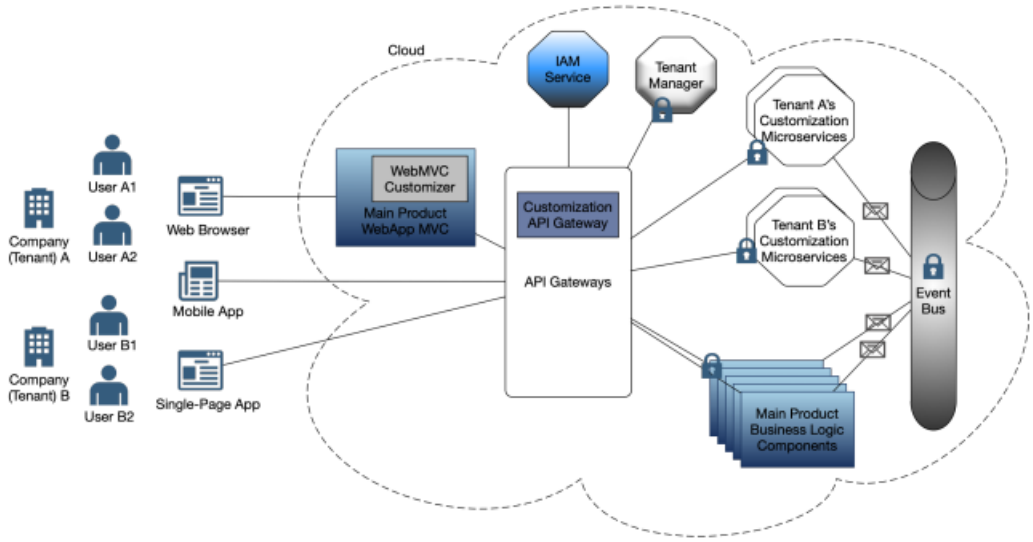

Рисунок 3.11 Архітектура моделі програмного забезпечення додатку процесу адаптації послуги під конкретні потреби певної особи або групи осіб [15]

Налаштовувач WebMVC — це локальний компонент, який вводиться в WebApp MVC основного продукту для перехоплення потоків основного продукту. Менеджер орендарів — це служба, яка керує налаштуваннями для кожного орендаря, що також схоже на інтрузивний підхід. Основна відмінність між інтрузивним і неінтрузивним підходами полягає в появі шлюзів API, служби керування ідентифікацією та доступом (IAM) і шини подій. У нашому ненав'язливому підході ми дотримуємося шаблону шлюзу API [19], щоб відокремити клієнтські програми (наприклад, програму WebMVC) від внутрішніх мікросервісів (для налаштування або основного потоку BL). Ключовим моментом децентралізованого підходу є те, що він дозволяє авторизований доступ орендарів налаштування мікросервісів до основного продукту BL через шлюзи API. Таким чином, мікросервіси налаштування орендарів можуть мати доступ до необхідного контексту виконання основного продукту BL, якщо це необхідно та дозволено. Децентралізований підхід може забезпечити глибоке налаштування, оскільки він дозволяє службі налаштування замінити компонент BL основного продукту для відповідного клієнта, якщо він авторизований. Авторизований доступ мікросервісів налаштування орендарів до основних компонентів BL продукту робить глибоке налаштування керованим. Це відрізняється від нав'язливого

способу надсилання коду «зворотного виклику» від мікросервісів налаштування до основного продукту для динамічної компіляції та виконання в контексті виконання основного сервісу. Служба IAM, створена на основі OpenID Connect4 або OAuth 2.05

Постачальник ідентифікаційної інформації може авторизувати індивідуальні налаштування для кожного орендаря. Використання стандартизованих і потужних механізмів автентифікації та авторизації OAuth 2.0 дуже важливий для ізоляції клієнта на рівні програми, особливо щодо налаштування. І останнє, але не менш важливе: шина подій дозволяє мікросервісам налаштування мати асинхронний зв'язок на основі подій із основними компонентами BL продукту для цілей налаштування. Ми запровадили перевірку концепції нашого ненав'язливого підходу для забезпечення глибокого налаштування еталонної програми для архітектури мікросервісів, eShopOnContainers [11]. MusicStore можна переробити, щоб увімкнути ненав'язливе налаштування, але ми вибрали eShopOnContainers, оскільки архітектура eShopOnContainers уже задовольняє наші передумови для ненав'язливого налаштування.

3.8 Еталонна архітектура для налаштування мікросервісами

У цьому розділі узагальнено два підходи до налаштування за допомогою мікросервісів. Ми представляємо основні принципи стилю налаштувань на основі мікросервісів і еталонну архітектуру, яка відповідає цим принципам.

Принципи

Ми використовуємо стиль налаштувань на основі мікросервісів, який керується набором принципів високого рівня або дизайнерських рішень, щоб відповідати вимогам ізоляції, асиміляції та економії масштабу. Перший набір принципів відповідає наступним вимогам ізоляції. Кожне налаштування – це послуга. Налаштування має бути автономно запущеним об'єктом із власним життєвим циклом, незалежним від продукту. Таке рішення дозволяє уникнути

збою в налаштуваннях (наприклад, мертвого циклу) від впливу на нормальну роботу основного продукту. Взаємодія служб налаштування з основним продуктом контролюється та адмініструється.

Послуга налаштування обслуговує одного і тільки одного орендаря. Цей принцип теж вказує, що не більше ніж одна служба продукту підключається до однієї служби налаштування, оскільки кожен клієнт належить лише до однієї служби продукту. Кожна служба налаштування працює у власному середовищі. Це запобігає впливу служб налаштування одна на одну під час виконання. Це також спрощує керування налаштуваннями, такими як моніторинг і виставлення рахунків. Служба налаштування має власну базу даних. Не можна допускати налаштування для зміни схеми бази даних продуктів. Він використовує свою ексклюзивну базу даних для зберігання даних, специфічних для налаштування. Служба налаштування не має прямого доступу ні до бази даних продукту, ні до іншої бази даних налаштування. Служба налаштування взаємодіє зі службою продукту та іншими службами налаштування лише через REST API. Інші способи зв’язку, такі як спільна база даних, спільна пам’ять або файли, зроблять сервіси більш тісними.

Ці принципи, які сприяють ізоляції, негативно впливають або на асиміляцію, або на економію масштабу. Чим більша ізоляція, тим менша ймовірність того, що налаштування можна буде асимілювати з основним продуктом. Так само більша ізоляція призводить до більшого використання ресурсів, що означає меншу користь для економії масштабу. Ми приймаємо наступні конструктивні рішення, щоб зменшити такі негативні наслідки та досягти кращого балансу щодо ізоляції, асиміляції та економії масштабу.

Середовище налаштування «наближене» до середовища продукту. Якщо середовище продукту розміщується хмарним провайдером, таким як Amazon AWS, середовища налаштування, пов’язані з ним, слід розгортати в інфраструктурі того самого хмарного провайдера у тому ж регіоні. Це забезпечує низьку затримку їх спілкування.

Зовнішня URL-адреса для доступу до служби налаштування відповідає URL-адресам продукту. Більшість служб налаштування невидимі для кінцевих користувачів, а використовуються лише опосередковано, коли користувачі викликають продукт. Коли послуга налаштування надається безпосередньо кінцевому користувачеві, URL-адреса для доступу до цієї служби має бути такою самою як URL-адреса продукту, щоб користувачі не відчували поділу.

Постачальник повинен забезпечити уніфіковане керування ідентифікацією для обох продуктів і налаштування, щоб користувачам не потрібен окремий вхід для використання налаштувань послуги. Середовища налаштування повинні займати мінімальну площу, щоб постачальники могли приймати велику кількість клієнтів для кожної послуги продукту. Іншими словами, сервіси налаштування мають бути легкими мікросервісами з мінімальним споживанням ресурсів. Постачальник повинен керувати всіма середовищами еластичним способом. Це зменшить загальну вартість ресурсів, особливо коли існує велика кількість легких, рідко використовуваних служб налаштування.

Постачальник повинен сприяти повторному використанню коду для додатків налаштування. Коли рішення для персоналізації підходить кільком клієнтам (це зазвичай відбувається, коли клієнти наймають того самого консультанта), клієнтам має бути легко повторно використовувати налаштування програми за допомогою коду, тобто створити новий екземпляр цієї програми.

Еталонна архітектура

Я створив еталонну архітектуру для підтримки налаштування SaaS із кількома клієнтами за допомогою мікросервісів клієнтів. Рисунок 3.12 ілюструє цю еталонну архітектуру з одним екземпляром продукту та двома службами налаштування. Продукт є типовим веб-додатком на основі браузера-сервера. Один екземпляр продукту обслуговує кілька клієнтів одночасно. Компонент керування клієнтом, як частина екземпляра продукту, контролює дійсний клієнт, який обслуговується цим екземпляром, уніфікована служба керування ідентифікацією контролює, які користувачі мають доступ через кожного клієнта. Екземпляр продукту та база даних розгортаються на одній віртуальній машині від

постачальника загальнодоступної хмари. Усі запити користувачів із браузера проходять через веб-проксі, який перетворює зручну URL-адресу на внутрішню адресу, яку використовує хмарний провайдер.

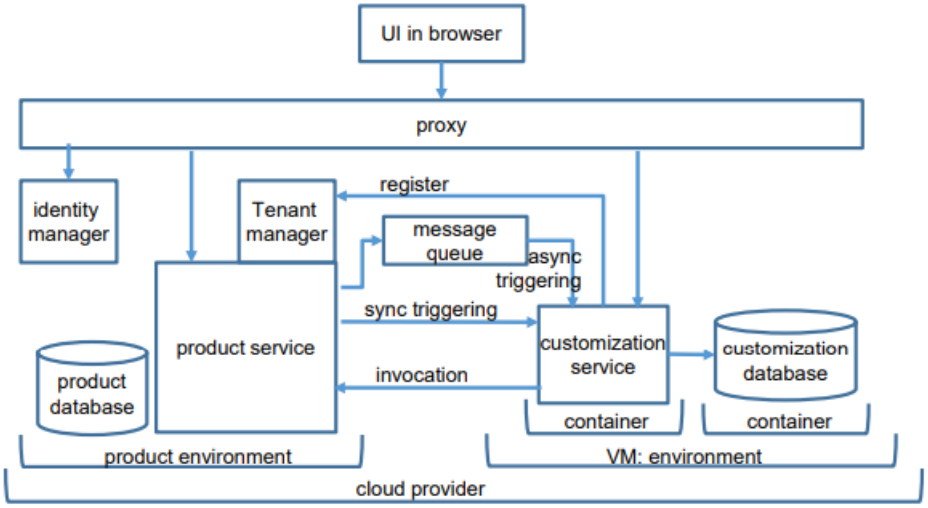

Рис. 3.12 Еталонна архітектура використання мікросервісів для налаштування

Служба налаштування налаштовує поведінку екземпляра продукту для одного з його орендарів, вводячи нові функції та замінюючи існуючі функції. Налаштування послуга реєструється для менеджера орендарів у процесі реєстрації налаштування, перш ніж її можна буде запустити. Нові функції можуть бути доступні через певну URL-адресу безпосередньо від користувачів або активовані екземпляром продукту за заздалегідь визначених обставин. Функції заміни запускаються екземпляром продукту, коли вихідну функцію в продукті збираються активувати. Менеджер орендарів визначає, коли запускати служби налаштування на основі реєстрації налаштування.Службі налаштування можуть знадобитися стандартні дані екземпляра продукту або змінити стан і дані екземпляра продукту. Це досягається шляхом виклику екземпляра продукту через виклики API (у ненав’язливому підході) або код зворотного виклику (в інтрузивному підході). Запуск і виклик — це два напрямки зв’язку між примірником продукту та службою налаштування.

Служба налаштування використовує власну базу даних для зберігання даних, які вона не може зберегти до стандартної бази даних. Служба

налаштування та база даних розгортаються в двох окремих середовищах, які, у свою чергу, розміщуються на віртуальній машині від того ж постачальника, що й середовище продукту.

Налаштування бази даних

Налаштування часто потребує розширення стандартного типу даних. Необхідно підтримувати два типи розширення схеми даних, тобто додавання нової сутності даних і додавання поля до існуючої сутності. Видалення сутності або поля не є обов'язковим для налаштування, оскільки служба налаштування може просто проігнорувати їх. Змінити тип поля можна, додавши нове поле та проігнорувавши початкове. Оскільки службі налаштування не дозволяється змінювати схему даних бази даних продукту, усі дані в розширеній сутності поля мають зберігатися в базі даних налаштування. Нову сутність даних можна реалізувати як таблицю в базі даних налаштування. Нове поле також можна реалізувати як таблицю в базі даних налаштування, як відображення з первинного ключа оригінального таблиці до розширеного поля.

Служба налаштування реєструє менеджера орендарів, як вона розширює стандартну схему даних. Таким чином служба продукту знає, як кожен орендар розширює свою базу даних, щоб він міг використовувати розширені дані.

Бази даних налаштування зазвичай мають просту схему та відносно невелику кількість даних. Тому доцільно використовувати полегшені технології, такі як PostgreSQL і MySQL.

Середовища налаштування

Середовище налаштування містить інфраструктуру, технічний стек і бібліотеки, необхідні службі налаштування під час виконання. Враховуючи, що кожна служба налаштування повинна мати унікальне ізольоване середовище, а послуга продукту може обслуговувати багато клієнтів, хмара постачальника може одночасно розміщувати велику кількість середовищ налаштування. Тому важливо зберегти мінімальний слід для кожного середовища налаштування та спростити керування цими середовищами.

Усі середовища налаштування мають використовувати однаковий тип інфраструктури, яка є одночасно легкою та легкою в управлінні. Контейнерна технологія, зокрема Docker, виявляється дуже придатною для цих цілей. Кожне середовище налаштування ізольовано в контейнері Docker, як середовище, у якому розміщено базу даних налаштування. Консультант надає постачальнику файл Docker, у якому вказується, як створити контейнер середовища налаштування, тобто вибір операційної системи, крок за кроком встановлення технічного стеку, завантаження вихідного коду додатка налаштування та, нарешті, визначення, як ініціалізувати технічний стек за допомогою програми налаштування. Постачальник створює образ Docker відповідно до файлу Docker у хмарі постачальника та створює екземпляр контейнера з образу, коли для клієнта потрібна служба налаштування. Постачальник повинен підтримувати кластер Docker, що складається з гнучкої кількості віртуальних машин від того самого хмарного постачальника. Залежно від кількості служб налаштування та навантаження на них, хмара постачальника повинна масштабуватися в кластері Docker або за його межами. Постачальник також може застосувати стандартний інструмент керування для моніторингу стану та споживання ресурсів кожним контейнером і за потреби вимкнути служби налаштування. Такі функції масштабування та керування можна реалізувати за допомогою інструментів Docker.

ВИСНОВКИ

Високонавантажені системи – це системи, які потрібно весь час масштабувати, правильно підбирати інфраструктуру і в цілому загальні архітектурні концепції. У цьому і полягає складність реалізації таких рішень, але з перспективи бізнесу – це вартує прикладених зусиль. ERP-система є високонавантаженою системою, вона є складним програмним комплексом, яка обробляє великі обсяги даних і виконує різноманітні завдання. Це пов'язано з її призначенням об'єднувати та автоматизувати всі бізнес-процеси підприємства.

Налаштування є широко поширеною практикою корпоративних програмних програм, таких як планування ресурсів підприємства (ERP). Постачальники програмного забезпечення розгортають свій корпоративний програмний продукт на території замовника, який потім часто налаштовують для різних конкретних потреб замовника. Коли корпоративні програми переміщуються в хмару як програмне забезпечення як послуга з кількома клієнтами (SaaS), традиційний спосіб локальної настройки стикається з новими проблемами, оскільки клієнт більше не має ексклюзивного контролю над програмою. Щоб надати підприємствам певні вимоги на додаток до спільного стандарту SaaS, постачальникам потрібен новий підхід для підтримки налаштувань багатокористувацького SaaS.

У кваліфікаційній роботі було представлено рішення для створення персоналізованого середовища багатокористувацького SaaS з використанням мікросервісів.

Від процесу конфігурації системи, який вимагає від користувача глибокого розуміння її внутрішньої структури розвинули рішення, щоб впровадити стратегію, яка передбачає мінімальне втручання в систему або в процес, який може бути більш практичним для промисловості.

На основі цих двох підходів, надали узагальнену еталонну архітектуру для забезпечення процесу адаптації та налаштування послуги під потреби замовників багатокористувацьких SaaS за допомогою мікросервісів.

Обговорення технічних викликів та потенційні рішення для реалізації еталонної архітектури дають нам більше інформації для прийняти рішення для процесу адаптації та налаштування послуги під потреби замовників.

Використання мікросервісів для налаштування кількох клієнтів SaaS в контексті високонавантажених ERP-систем є перспективним підходом, який дозволяє створювати гнучкі, масштабовані та адаптивні системи.

ПЕРЕЛІК ПОСИЛАНЬ

1. S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd edition [Text] / S. Newman – Sebastopol: O'Reilly Media, Inc., 2021.

2. S. Newman, Monolith to Microservices: Evolutionary Patterns to Transform edition [Text] / S. Newman – Sebastopol: O'Reilly Media, Inc., 2020.

3. S. Newman, Production-Ready Microservices: Building Standardized Systems Across an Engineering Organization [Text] / S. Newman – Sebastopol: O'Reilly Media, Inc., 2021.

4. C. Richardson, Microservices Patterns: With Examples in Java [Text] / C. Richardson – Shelter Island: Manning Publications, 2019.

5. K. Boniface, Designing Microservices with API Gateway and Kubernetes [Text] / K. Boniface – Sebastopol: O'Reilly Media, Inc., 2020.

6. C. Walls, Spring in Action, Sixth Edition [Text] / C. Walls - Shelter Island: Manning Publications, 2022.

7. H. Schildt, Java: The Complete Reference, Twelfth Edition [Text] / H. Schildt – New York: McGraw Hill, 2021.

8. What Are Microservices? [Електронний ресурс] // Nginx – 2023 – Режим доступу до ресурсу: https://www.nginx.com/learn/microservices/

9. Why use a microservices approach to building applications [Електронний ресурс] // Microsoft – 2022 – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/azure/servicefabric/service-fabric-overview-microservices

10 Microservices Architecture: All You Need To Know [Електронний ресурс] / Sam Newman // martinfowler.com – 2014 – Режим доступу до ресурсу: https://martinfowler.com/articles/microservices.html

11. Building Microservices: Using an API Gateway [Електронний ресурс] / Chris Richardson // microservices.io – 2016 – Режим доступу до ресурсу: https://microservices.io/patterns/apigateway.html

12. Pattern: Circuit Breaker [Електронний ресурс] / Chris Richardson //

microservices.io – 2016 – Режим доступу до ресурсу: https://microservices.io/patterns/reliability/circuit-breaker.html

13. Jerry Wallis. 6 serverless computing benefits and drawbacks to help decide if it's right for you, 2023. URL https://intuji.com/ serverless-computing-benefits-and-drawbacks/.

14. Avenga. Microservices vs. monoliths: which architecture will be best for your product?, 2022. URL https://www.avenga.com/magazine/ microservices-vs-monoliths/.

15. Adam Bellemare. Building Event-Driven Microservices. Oreilly & Associates Inc., 2020.

16. David Boyne. Inside event-driven architectures, 2023. URL https://serverlessland.com/event-driven-architecture/visuals/ inside-event-driven-architectures.

17. ERP Research. Independently compare erp systems and erp consulting, 2023. URL https://www.erpresearch.com/.

18. Expert Market Research. Global serverless computing market outlook, 2023. URL https://www.expertmarketresearch.com/reports/ serverless-computing-market.

19. Grand View Research. Cloud computing market size, share and trends analysis report by service, by deployment, by enterprise size, by end-use, by region, and segment forecasts, 2023. URL https://www.grandviewresearch.com/industry-analysis/ cloud-computing-industry.

20.https://www.thesunflowerlab.com/microservice-architecture-benefits-business-value/

21. Toomey J. What Is ERP? [Електронний ресурс] / Jennifer Toomey. – 2020. – Режим доступу до ресурсу: https://www.oracle.com/erp/what-is-erp/

22. Perkins B. What is ERP? Key features of top enterprise resource planning systems [Електронний ресурс] / Bart Perkins. – 2020. – Режим доступу до ресурсу: https://www.cio.com/article/2439502/what-is-erp-key-features-of-top-enterprise-resourceplanning-systems.html

23. Best ERP Software 2020: Top Rated ERP Systems Comparison [Електронний ресурс]. – 2020. – Режим доступу до ресурсу:

https://www.softwaretestinghelp.com/besterp-software-systems/

24. Toomey J. Oracle Enterprise Resource Planning (ERP) [Електронний ресурс] / Jennifer Toomey. – 2020. – Режим доступу до ресурсу: https://www.oracle.com/erp/

25. Newman S. Building Microservices – Sebastopol, California: O`Reilly Media, 2021. – 612 p.

26. O. A. Waraga, M. Bettayeb, Q. Nasir, and M. A. Talib, “Design and implementation of automated iot security testbed,” Computers & Security, vol. 88, p. 101 648, 2020.

27. Docker: Empowering app development for developers, https://www.docker.com/, [Online; accessed May-11-2021], 2021.

28. C. Lin, S. Nadi, and H. Khazaei, “A large-scale data set and an empirical study of docker images hosted on docker hub,” in Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2020.

29. Aws step functions, https://aws.amazon.com/step -functions/, [Online; accessed February-24-2021], 2020.

30. A. Goli, O. Hajihassani, H. Khazaei, O. Ardakanian, M. Rashidi, and T. Dauphinee, “Migrating from monolithic to serverless: A fintech case study,” in Companion of the ACM/SPEC International Conference on Performance Engineering, 2020, pp. 20–25.