Міністерство освіти і науки України
Державний ВНЗ «Національний гірничий університет»

Факультет інформаційних технологій
(факультет)

Кафедра програмного забезпечення комп'ютерних систем
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА
дипломної роботи

магістра
(назва освітньо-кваліфікаційного рівня)

галузь знань *12 Інформаційні технології*
(шифр і назва галузі знань)

спеціальність *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

спеціалізація *Програмне забезпечення систем*
(код і назва спеціалізації)

освітній рівень *магістр*
(назва освітнього рівня)

кваліфікація *Інженер-програміст*
(назва кваліфікації)

на тему: *Дослідження ефективності методів контролю якості додатків з мікросервісною архітектурою*

Виконавець:
студент *6* **курсу , групи** *121М-16-1*

(підпис) *Поштак Р.В.* (прізвище та ініціали)

Керівники	Посада, прізвище, ініціали	Оцінка	Підпис
проекту	*доц. Алексєєв М.О.*		
розділів:			
Економічний	*доц. Касьяненко Л.В.*		

Рецензент			

Нормоконтроль	*доц. Коротенко Л.М.*		

Дніпро
2018

Міністерство освіти і науки України
Державний вищий навчальний заклад
«Національний гірничий університет»

ЗАТВЕРДЖЕНО:
завідувач кафедри

програмного забезпечення комп'ютерних систем
(повна назва)

_____ І.М. Удовик
(підпис) (прізвище, ініціали)

« » _____ 20 ____ року

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

спеціальності *121 Інженерія програмного забезпечення*
(код і назва спеціальності)

студенту *121М-16-1* *Поштак Р.В.*
(група) (прізвище та ініціали)

Тема дипломної роботи *Дослідження ефективності методів контролю якості додатків з мікросервісною архітектурою*

1 ПІДСТАВИ ДЛЯ ПРОВЕДЕННЯ РОБОТИ

Наказ ректора Державного ВНЗ «НГУ» від *26.12.2017 р.* № *2127-л*

2 МЕТА ТА ВИХІДНІ ДАНІ ДЛЯ ПРОВЕДЕННЯ РОБІТ

Об'єкт досліджень – методологія проектування та тестування мікросервісних додатків

Предмет досліджень – процес створення та тестування мікросервісних додатків

Мета НДР – створення вдосконаленої методики проектування мікросервісних систем, дослідження існуючих методів контролю якості мікросервісних додатків

Вихідні дані для проведення роботи:

наукові дослідження методів проектування мікросервісних систем;

бібліотеки мов програмування високого рівня для створення мікросервісних систем.

3 ОЧІКУВАНІ НАУКОВІ РЕЗУЛЬТАТИ

Універсальна методика створення мікросервісних додатків, яка може бути використана для проектування мікросервісних додатків за допомогою різних програмних платформ. Тестова система на основі мікросервісного підходу

Наукова новизна результатів, що очікуються, полягає у: проведені аналізу та виявленні недоліків традиційного підходу до розробки мікросервісних систем та у створенні методики проектування мікросервісних систем на різних мовах програмування.

Практична цінність результатів полягає у: розробленні алгоритмів для сторення, розгортання та масштабування мікросервісних додатків за допомогою різних програмних засобів та створення тестової системи на основі мікросервісного підходу

4 ВИМОГИ ДО РЕЗУЛЬТАТІВ ВИКОНАННЯ РОБОТИ

Результати досліджень мають бути подано у вигляді, що дозволяє безпосереднє використання методики. Згідно виробничих функцій та професійних задач магістра, які виносяться на кваліфікаційну роботу, повинні бути розроблені відповідні програми.

5 ЕТАПИ ВИКОНАННЯ РОБІТ

Найменування етапів робіт	**Строки виконання робіт (початок – кінець**
Аналіз архітектурних стилів створення веб-додатків	15.10.2017 – 15.11.2017
Аналіз інфраструктури та оптимізація процесів створення і розгортання мікросервісів	16.11.2017 – 15.12.2017
Дослідження платформ для побудову Msa та приклад створення тестової системи	16.12.2017 – 15.01.2018

6 РЕАЛІЗАЦІЯ РЕЗУЛЬТАТІВ ТА ЕФЕКТИВНІСТЬ

Економічний ефект від реалізації результатів роботи очікується позитивним завдяки скороченню часу на проектування, розробку та моніторинг мікросервісної системи та економії за рахунок скорочення витрат на покупку апаратному забезпечення системи.

Соціальний ефект від реалізації результатів роботи очікується позитивним завдяки від реалізації методики створення мікросервісних систем завдяки скороченню витрат на їх реалізацію.

7 ДОДАТКОВІ ВИМОГИ

Завдання видав _____ *Алексєєв М.О.*
(підпис) (прізвище, ініціали)

Завдання прийняв до виконання _____ *Поштак Р.В.*
(підпис) (прізвище, ініціали)

Дата видачі завдання: *15.10.2017 р.*

Термін подання дипломного проекту до ДЕК _____

Реферат

Пояснительная записка: 106 с., 27 рис., 3 прил., 78 источника.

Объект исследования: методология проектирования и тестирования микросервисних приложений.

Цель магистерской работы: создание усовершенствованной методики проектирования микросервисних систем, исследование существующих методов контроля качества микросервисних приложений.

Методы исследования. При решении поставленной задачи использовались научные достижения в областях разработки микросервисних систем и программного обеспечения.

Научная новизна полученных результатов заключается в проведении анализа и выявлении недостатков традиционного подхода к разработке микросервисних систем и в создании методики проектирования микросервисних систем на различных языках программирования.

Практическое значение работы заключается в разработке алгоритмов для создания, развертывания и масштабирования микросервисних приложений с помощью различных программных средств и создания тестовой системы на основе микросервисного подхода.

Область применения. Разработанная методика может применяться для проектирования и создания микросервисних приложений.

Значение работы и выводы. Усовершенствованная методика позволяет проектировать микросервисные приложения со значительным сокращением как материальных затрат, так и временных, что подтверждается разработанным программным продуктом в данной магистерской работе.

Прогнозы по развитию исследований. Разработать универсальную методику создания микросервисних приложений, которая может быть использована для проектирования микросервисних приложений с помощью различных программных платформ. Разработать тестовую систему на основе микросервисного подхода.

В разделе «Экономика» проведены расчеты трудоемкости разработки программного обеспечения, расходов на создание ПО и длительности его разработки, а также проведены маркетинговые исследования рынка сбыта созданного программного продукта.

Список ключевых слов: MSA, API, CPU, DEVOPS, DOCKER, HTTP, HYPER-V, SSH, МИКРОСЕРВИСЫ, КОНТЕЙНЕРИЗАЦИЯ, ОПЕРАЦИОННАЯ СИСТЕМА, БАЗА ДАННЫХ.

Реферат

Пояснювальна записка: 106 с., 27 рис., 3 додатків., 78 джерела.

Об'єкт дослідження: методологія проектування та тестування мікросервісних додатків.

Мета магістерської роботи: створення вдосконаленої методики проектування мікросервісних систем, дослідження існуючих методів контролю якості мікросервісних додатків.

Методи дослідження. При рішенні поставленої задачі використовувалися наукові досягнення в областях розробки мікросервісних систем та програмного забезпечення.

Наукова новизна результатів, що очікуються, полягає у проведені аналізу та виявленні недоліків традиційного підходу до розробки мікросервісних систем та у створенні методики проектування мікросервісних систем на різних мовах програмування.

Практична цінність результатів полягає у розробленні алгоритмів для сторення, розгортання та масштабування мікросервісних додатків за допомогою різних програмних засобів та створення тестової системи на основі мікросервісного підходу.

Область застосування. Розроблена методика може застосовуватися для проектування та створення мікросервісних додатків.

Значення роботи та висновки. Удосконалена методика дозволяє проектувати мікросервісні додатки зі значним скороченням як матеріальних витрат, так і тимчасових, що підтверджується розробленим програмним продуктом в даній магістерській роботі.

Прогнози щодо розвитку досліджень. Розробити універсальну методику створення мікросервісних додатків, яка може бути використана для проектування мікросервісних додатків за допомогою різних програмних платформ. Розробити тестової систему на основі мікросервісного підходу.

У розділі «Економіка» проведені розрахунки трудомісткості розробки програмного забезпечення, витрат на створення ПЗ й тривалості його розробки, а також провести маркетингові дослідження ринку збуту створеного програмного продукту.

Список ключових слів: MSA, API, CPU, DEVOPS, DOCKER, HTTP, HYPER-V, SSH, МІКРОСЕРВІСИ, КОНТЕЙНЕРИЗАЦІЯ, ОПЕРАЦІЙНА СИСТЕМА, БАЗА ДАНИХ.

The abstract

Explanatory note:106 p., 27 fig., 3 applications, 78 sources.

Object of research: the methodology of designing and testing micro-service applications.

The purpose of the degree project: creation of advanced methodology for designing microservice systems, research of existing methods of quality control the microservice applications.

Methods of research. In solving the problem, scientific achievements in the areas of development of microsystems systems and software were used.

The scientific novelty of the received results are to analyze and identify the shortcomings of the traditional approach to the development of microsystems and to create a methodology for designing microsystems in different programming languages.

The practical value of work is is to develop methods for retrieving, deploying and scaling micro-service applications using various software tools and creating a test system based on the microservice approach.

The scope. Develop technique can be used for designing and creating microservice applications.

The value of the work and conclusions. Advanced technique allows the design of micro-service applications with significant reductions in both material costs and time, as evidenced by the developed software in the master's work.

Projections on development research. Develop a universal methodology for creating microservice applications, which can be used to design microservice applications through various software platforms. Develop a test system based on a microservice approach.

In section "Economics" calculated the complexity of software development, the cost of creating the software and the duration of its development, and marketing studies market created by the software.

List of keywords: MSA, API, CPU, DEVOPS, DOCKER, HTTP, HYPER-V, SSH, MICROSERVICES, CONTAINERIZATION, OPERATING SYSTEM, DATABASE.

Зміст

Перелік скорочень

MSA – MicroService Architecture (мікросервісна архітектура)

SOA – Service-Oriented Architecture (сервіс-орієнтована архітектура)

REST – Representational State Transfer, архітектурний стиль

ПЗ – Програмне забезпечення RPC Remote Procedure Call, протокол для взаємодії між комп'ютерами

БД – База даних

API – Application Programming Interface, набір інтерфейсів для взаємодії різнотипного програмного забезпечення

HTTP – Hyper Text Transfer Protocol, протокол передачі даних, на якому побудована мережа Інтернет

HTTPS – Hyper Text Transfer Protocol S ecure, захищений HTTP

JSON – JavaScript Object Notation, текстовий формат обміну даними

SOAP – Simple Object Access Protocol, протокол обміну структурованими повідомленнями в розподілених системах

CI – Continuous Integration CD Continuous Delivery

ООП – Об'єктно-орієнтоване програмування

AWS – Amazon Web Services, інфраструктура хмарних платформ

CGI – Common Gateway Interface, стандарт інтерфейсу, який використовується для зв'язку зовнішньої програми та серверу.

J2EE – Java Enterprise Edition, java платформа для корпоративних додатків

OSGi – Open Service Gateway Initative, специфікація модульної системи

GWT – Google Web Toolkit, Java-фреймворк

XML – eXtensible Markup Language, розширювана мова розмітки

Вступ

Актуальність роботи. Сьогодні процес проектування, розробки та функціонування програмного забезпечення значно еволюціонував в порівнянні із попередніми десятиліттями і характеризується значним ускладненням вимог в зв'язку із зростанням складності бізнес-процесів та зростанням кількості даних в обчислювальних системах. Вимогами до сучасних програмних систем є не тільки вимоги функціональності, але й вимоги до продуктивності, надійності та ефективності рішень.

Виходячи з цього, велику роль у задоволенні вимог до програмних систем відіграє вибір архітектури реалізації, оскільки вона значною мірою впливає на алгоритмічні та математичні рішення в процесі розробки. При створенні розподілених обчислювальних систем, зокрема грід та хмарних сервісів для розв'язання ресурсоємних задач щодо створення власних IT- рішень та керування бізнес-процесами, доцільно розробити власну архітектуру програмної системи, або ж обґрунтувати вибір однієї з наявних.

Архітектурний стиль мікросервісів - це підхід, при якому програмний додаток будується як набір невеликих сервісів, кожен з яких працює у власному процесі і виконує комунікацію з рештою сервісів використовуючи мережні механізми, як правило HTTP. Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованого середовища. Самі по собі ці сервіси можуть бути написані на різних мовах і використовувати різні технології зберігання даних.

Подібні сервіси будують завдяки потребам користувача і розгортають незалежно з використанням повністю автоматизованого середовища. Вони можуть бути написані з використанням різних мов програмування високого рівня і технологій зберігання даних. Зростаюча складність систем, навантаження на окремі її частини, а також необхідність відповідати вимогам бізнесу потребує нових, гнучкіших підходів до проектування програмного забезпечення та інфраструктури для його розгортання.

Мікросервісна архітектура підходить для процесів безперервного постачання. Мікросервісну архітектуру спрямовано на створення одного чи

кількох застосунків, на відміну від сервіс-орієнтованої архітектури (архітектурний шаблон програмного забезпечення, модульний підхід до розроблення програмного забезпечення, оснований на використанні розподілених компонентів, забезпечених стандартними інтерфейсами для взаємодії застосунків за визначеними протоколами).

Відомо низку проектів, що використовують мікросервісну архітектуру. Їх порівнюють з потужним застосунком, що побудований як єдине ціле. Застосунки на рівні підприємств базовано з трьох основних підходів: дружній інтерфейс, база даних (БД) та сервер. Серверна частина обробляє HTTP запити, оновлює дані в БД, заповнює HTML сторінки, які потім мають бути відправлені браузеру клієнта. Зміни в системі потребують розгортання нової версії серверної частини програми. Бізнес-логіку з оброблення запитів виконують в єдиному місці, при цьому можна скористатися наявними можливостями мови програмування для поділу прикладного застосунку на класи, функції та області дії імен. Можна масштабувати великі застосунки горизонтально через запуск кількох фізичних серверів за балансувальником навантаження.

Цілі і задачі дослідження. Метою даної магістерської роботи є аналіз основних ідей і принципів мікросервісної архітектури, виділення проблем які виникають при створенні мікросервісних додатків, створення вдосконаленої методики проектування мікросервісних систем, дослідження існуючих методів котролю якості мікросервісних додатків.

Об'єкт дослідження - методологія проектування та тестування мікросервісних додатків.

Предметом дослідження - процес створення та тестування мікросервісних додатків.

Мета роботи полягає у створення вдосконаленої методики проектування мікросервісних систем, дослідження існуючих методів котролю якості мікросервісних додатків.

Методи дослідження. При рішенні поставленої задачі використовувалися наукові досягнення в областях розробки мікросервісних систем та програмного забезпечення.

Наукові положення, очікувані наукові результати.

1. Сформований аналіз традиційного підходу до розробки мікросервісних додатків, а також виявлення недоліків;

2. Створення методики проектування, розгортання та масштабування мікросервісних додатків за допомогою різних програмних засобів та розробка тестової системи на основі мікросервісного підходу.

Обгрунтованість і достовірність наукових положень.

Обгрунтованість і достовірність наукових положень, висновків і рекомендацій магістерської роботи обгрунтована коректністю поставлених проблем і прийнятих припущень при математичному описі процесів, обґрунтованістю вихідних даних, достатнім обсягом вибірки даних і верифікованими на модельних об'єктах результатами обчислень.

Наукова новизна результатів, що очікуються, полягає у проведені аналізу та виявленні недоліків традиційного підходу до розробки мікросервісних додатків та у створенні методики проектування мікросервісних додатків на різних мовах програмування.

Практична цінність результатів полягає у розробленні методики для сторення, розгортання та масштабування мікросервісних додатків за допомогою різних програмних засобів та створення тестової системи на основі мікросервісного підходу.

Особливий внесок магістра полягає в:

- виборі методів досліджень і технологій реалізації;
- створення методики проектування, розгортання та масштабування мікросервісних додатків
- створення тестової системи, що реалізує механізми мікросервісного підходу;
- розробці теоретичної частини роботи, в якій досліджені і систематизовані знання про існуючі підходи розробки мікросервісних додатків;
- оцінці отриманих результатів.

Апробація результатів магістерської роботи.

Основні положення і результати були докладені та обговорені на студентській науковій конференції.

Структура і обсяг роботи. Робота складається з вступу, трьох розділів та висновків. Містить 106 сторінок друкованого тексту з 27 рисунками, списку використаних джерел з 78 найменуваннями на 5 сторінках, 3 додатків на 11 сторінках.

РОЗДІЛ 1
АНАЛІЗ АРХІТЕКТУРНИХ СТИЛІВ СТВОРЕННЯ ВЕБ-ДОДАТКІВ

В ході розвитку інформаційних технологій вимоги по продуктивності і доступності до інформаційних систем, що тільки розробляються, так і вже існуючим, постійно росли. Для того, щоб системи відповідали таким постійно зростаючим вимогам, відбувалося нарощування обчислювальних потужностей, тобто застосовувалося так зване вертикальне масштабування. Проте ефективність вертикального масштабування виявилася дуже обмеженою, оскільки приріст обчислювальних потужностей конкретного сервера не давав необхідного приросту продуктивності інформаційної системи, що спричинило появу горизонтального масштабування. Слід зазначити, що для забезпечення ефективності застосування горизонтального масштабування, необхідно закладати його можливість в архітектуру системи ще на етапі її проектування.

1.1. Горизонтальне масштабування та архітектура веб-додатків

Горизонтальне масштабування здійснюється не за рахунок збільшення обчислювальних потужностей конкретного сервера, а за рахунок додавання нових вузлів в систему. Існує три стратегії горизонтального масштабування додатків.

Модель стратегій масштабування можна представити у вигляді тривимірного куба, в якому кожна із сторін є певною стратегією (рис. 1.1).

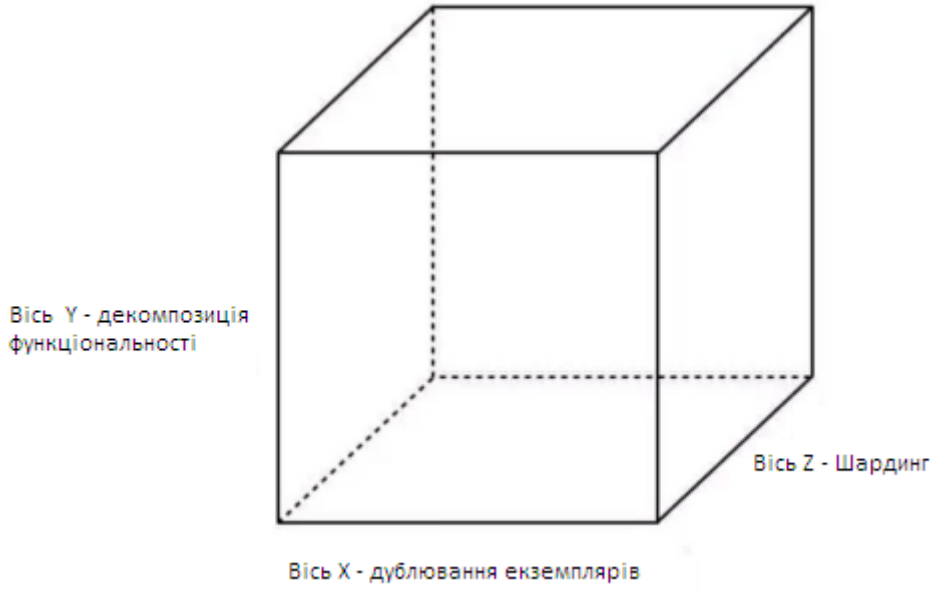


Рис. 1.1. Модель масштабування застосунків

Масштабування по осі X - це класична стратегія горизонтального масштабування, яка використовує балансування навантаження (рис 1.2).

Це простий і ефективний спосіб масштабування для багатьох типів додатків. Цей підхід перевірений часом і простий в реалізації, він працює навіть в тих випадках, коли при розробці додатку спочатку не малося на увазі його масштабувати. Монолітні додатки в основному масштабуються за допомогою цього підходу, оскільки під час їх розробки не враховувалися інші стратегії і їх залежність від стану користувацьких сесій, роблять неможливим використати інші моделі.

Модель масштабування по осі X може бути легко реалізована за допомогою балансування навантаження на 4 рівні моделі OSI (TCP) якщо стан не важливий, але часто потрібно балансування, яке працює на 7 рівні OSI (HTTP) для перевірки http-заголовків або інших складових запиту для того, щоб перенаправити його в потрібний сервіс. На рисунку 1.2 зображена модель масштабування по осі X.

Рис 1.2. Модель масштабування по осі X

Масшабування по осі Y є декомпозицією додатка на різні сервіси. Це найкращий спосіб масштабування для сервісів, які групують свою функціональність. Ця модель реалізується з використанням механізму (зазвичай із застосуванням балансування навантаження на 7 рівні моделі OSI), який переглядає URL або заголовки запиту і перенаправляє запит в потрібний сервіс (рис. 1.3).

Рис. 1.3. Модель масштабування по осі Y

Хоча даний шаблон можна використати і в монолітних застосунках, в яких є можливість розділити внутрішню функціональність на різні сервіси за допомогою розбору і аналізу даних в URL або в заголовках, робити це треба обережно і стежити за їх станом (дані сесії в пам'яті). Наприклад, не можна перенаправити запит до сервісу B для оплати якщо він посилається на сесії, які зберігаються в сервісі A або C. Масштабування по осі Z поводиться як масштабування по осі X, оскільки воно обгрунтоване на клонуванні серверів додатків. Різниця між цими підходами полягає в тому, що запити перенаправляються залежно від даних, а не від шляху в URL або даних в заголовках (рис. 1.4).

Рис. 1.4. Модель масштабування по осі Z

Інший шаблон використання масштабованості по осі Z полягає в тому, що привілейований користувач може отримати відповідь на свій запит швидше, ніж звичайний користувач, оскільки для його запитів буде виділений окремий сервер.

1.2. Монолітна архітектура

Монолітна архітектура передбачає реалізацію всіх сервісів ресурсу як єдиної програмної системи. Тобто всі сервіси реалізовані за допомогою одного набору технологій (і мови програмування) і використовують загальні бібліотеки коду. Всі сервіси працюють з одним сервером баз даних, що дозволяє кожному сервісу звертатися до бази даних безпосередньо (рис. 1.5).

Вся логіка по обробці запитів виконується в єдиному процесі, при цьому ви можете використати можливості вашої мови програмування для розділення додатка на класи, функції та namespace. Ви можете запускати і тестувати додаток на машині розробника і використати стандартний процес розгортання для перевірки змін перед викладанням їх в продакшн. Ви можете масштабувати монолітні застосунки горизонтально шляхом запуску декількох фізичних серверів з балансувальником навантаження.

Монолітні додатки можуть бути успішними, але все більше людей розчаровуються в них, особливо у світлі того, що все більше додатків розгортаються за допомогою хмарних технологій. Будь-які зміни, навіть самі невеликі, вимагають перезбірки і розгортання усього моноліту. З часом, стає важче зберігати функціональну модульну структуру, зміни логіці одного модуля мають тенденцію впливати на код інших модулів.

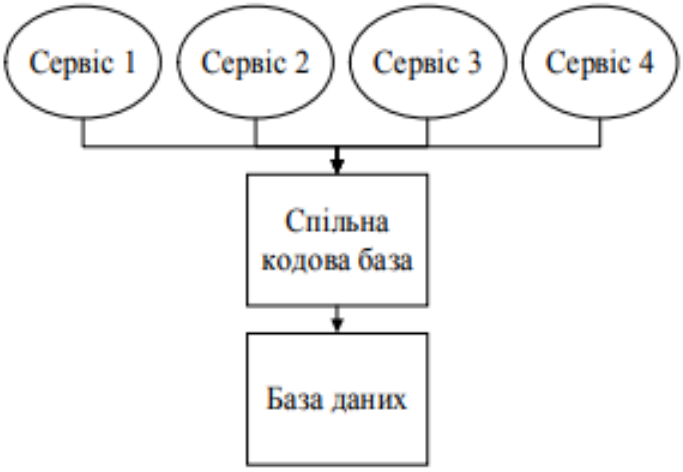


Рис. 1.5. Програмна реалізація монолітної архітектури

1.3. Сервіс-орієнтована архітектура (SOA)

SOA замість монолітної пропонує блокову систему із взаємодіючих компонентів, у якій різні функціональні модулі додатків (сервіси), призначені для управління бізнес-процесами, що пов'язані між собою за допомогою чітко визначених інтерфейсів. Інтерфейси самі по собі незалежні від оточення і платформи, і тому така модель отримала назву моделі «слабкого зв'язку». Фактично суть концепції SOA полягає в уніфікації та автоматизації бізнес-процесів за допомогою типових компонентів — сервісів (наприклад, веб- сервіси використовують один стандарт — розширювану мову розмітки XML). При цьому створення, впровадження або зміна бізнес-процесу є компоновкою (оркестровкою) раніше розроблених сервісів, призначених для автоматизації бізнес-функцій.

Для бізнесу SOA означає збільшення задоволеності клієнтів, реальну гнучкість бізнесу, швидкий час виходу на ринок, простоту співробітництва і низьку вартість бізнесу. Для IT-організацій SOA означає більшу продуктивність, зниження витрат на IT за рахунок прискорення розробки додатків, більш м'якого повторного використання сервісів, більш високої якості додатків, і в цілому більш швидкого реагування на запити бізнес-клієнтів для поліпшення і модифікації системи. На додаток до цього існує можливість використання сервісів незалежних постачальників, що забезпечує ще більшу цінність SOA.

Слово «гнучкість» часто згадується у ході обговорення переваг SOA і може бути інтерпретовано, з одного боку, як можливість змінювати бізнес-процеси відповідно до змін вимог ринку і вимог клієнтів і, з другого, як здатність виконувати бізнес-процеси швидше або запускати швидше нові процеси, продукти і сервіси. Гнучкість і швидкість — реальні й відчутні переваги переходу на SOA і багаторазове використання сервісів.

SOA є архітектурним підходом, який дозволяє розкласти функціональність додатків на множину сервісів, розміщених у мережі. Веб-сервіси є технологію реалізації концепції дизайну SOA. Існує велика кількість досліджень, присвячених вимогам надійності в SOA і, більш конкретно, веб- сервісів. Веб-спільнота розробила ряд специфікацій, які підтримують надійний обмін повідомленнями, управлінням транзакціями, реплікаціями і безпекою. Сервіси взаємодіють один з одним за допомогою шаблону синхронного запиту і відповіді, що забезпечує щільне зчеплення між компонентами системи (рис.1.6).

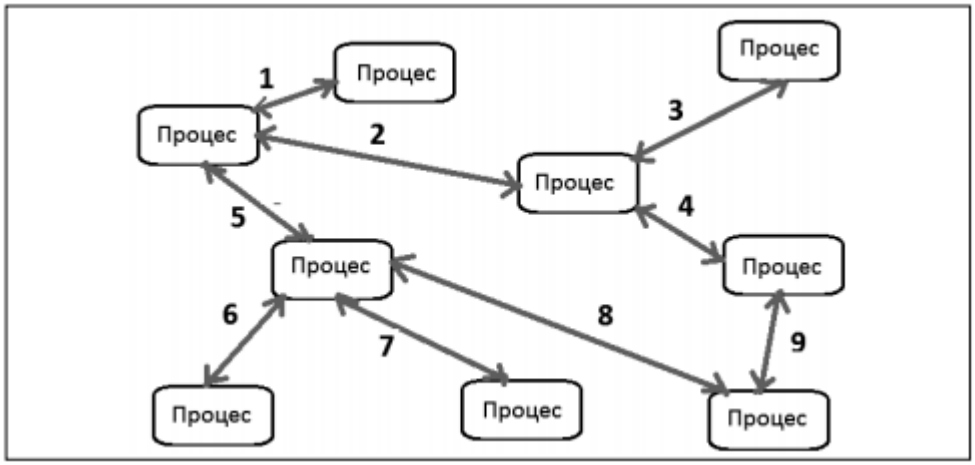


Рис. 1.6. Архітектура SOA, керована запитами

До базових принципів SOA треба віднести:

- Слабке зв’язування сервісів — одна з найбільших переваг SOA. Слабко зв’язані модулі мають кілька добре відомих залежностей, а тісно пов’язані модулі мають багато невідомих залежностей. У суперечність зі слабким зв’язком між сервісами і можливістю їх повторного використання, застосування сервісу може бути ґрунтовним і щільним.

- Стандартизація — стандарти SOA відкриті в тому сенсі, що будь-який виробник програмного забезпечення має право використовувати ці стандарти у процесі розробки архітектури SOA. Крім того, процес створення та перегляду стандартів є більш-менш демократичним, де будь-яка зацікавлена сторона має право брати участь у всіх засіданнях, які призводять до рішень про стандарт.
- Модульність — SOA реалізує сервіси, що підтримують чітко визначені модульні бізнес-функцій та інформаційні процеси. Ці модулі можуть надалі бути використані повторно і бути частиною бізнес-процесу. Модульність сервісів призводять до поділення додатків на безліч дрібних модулів. Кожен модуль відповідає за одну окрему функцію у додатку.
- Скомпонованість — здатність ефективно складати сервіси є найважливішою вимогою для досягнення деяких із найбільш фундаментальних цілей сервіс-орієнтованих застосувань. Сервіси, які складаються, здатні брати участь у якості ефективних модулів. Модульна сервісна структура дозволяє створювати нові інформаційні системи, про які розробники сервісів можуть не мати жодного уявлення під час проектування сервісів.
- Повторюваність — повторне використання сервісів як основна частина аналізу обслуговування та процесів проектування систем сервісів. Згідно з цим принципом сервіс може бути використаний більш ніж за одним сценарієм у різних бізнес-процесах або інформаційних системах.
- Відкритість — сервіси мають бути легко ідентифіковані й зрозумілі, щоб забезпечити можливість їх повторного використання. Тому у процесі проектування сервісів необхідно враховувати «якість доступу» до сервісів та їх індивідуальні властивості незалежно від того, чи вони зареєстровані в депозитарії, чи ні.
- Абстракція — цей принцип підкреслює необхідність приховувати так багато з основних деталей сервісів, як це можливо. Це забезпечує безпосередньо описане раніше слабке зв’язування сервісів.
- Зв’язок «один-на-один» — один із конкретних сервісів викликається у часі тільки одним із споживачів, але зв’язок є двонаправленим.

- Синхронність — відповіді на запити відправляються назад до споживача в синхронному режимі.
- Запуск процесу — потік управління зініціюється клієнтом (споживачем послуг).
- Зернистість сервісів — рівень деталізації обслуговування. Сервіси в SOA є модулями бізнес-логіки досить високого рівня, завдяки чому взаємодія між ними зводиться до обмеженого числа повідомлень за змістом бізнес-логіки замість безлічі низькорівневих викликів, що враховують деталі реалізації сервісів. Такий підхід знижує навантаження на мережу і сприяє більш високій продуктивності системи.
- Відсутність стану — сервіси проектуються так, щоб залишитися зі станом тільки за необхідності. Сервісам не варто покладатися на тривалий зв'язок між споживачем і постачальником, вони не мають також покладатися на попередні виклики.

Орієнтація на сервіси та SOA може краще використовуватися тоді, коли процеси або їх частини є стандартними, коли вони часто повторюються без змін, або коли декільком користувачам потрібно той же компонент процесу для виконання своїх завдань. Виклик (споживання) сервісів у SOA реалізується віддалено за допомогою віддаленого виклику процедури RPC (remote procedure call — віддалений виклик процедур) на вимогу споживача сервісів. SOA добре зарекомендувала себе для побудови великих корпоративних програмних додатків. Ціла низка розробників та інтеграторів пропонують інструменти і рішення на основі SOA (наприклад, платформи Intel SOA Expressway, JBoss SOA Platform, IBM WebSphere, Software AG webMethods, Oracle, BEA Aqualogic, Microsoft Windows Communication Foundation, SAP NetWeaver, TIBCO).

1.4. Подійно-орієнтована архітектура (EDA)

EDA (event-driven architecture — подійно-орієнтована архітектура) містить в собі три базові компоненти: генератор події (датчик), оброблювач події і менеджер подій (відповідач). Менеджер подій реєструє всі події, які виникають в системі (зовнішні і внутрішні), відповідним чином ідентифікує і передає

оброблювачу подій. Якщо потрібний оброблювач за яких-небудь при- чин недоступний, подія в залежності від конкретної реалізації або ставиться в чергу, або передається іншому оброблювачу. Завдяки такій схемі система стає максимально гнучкою і чутливою до змін інформаційного середовища — у разі виникнення принципово нової події досить підключити новий оброблювач, не зачіпаючи ядра системи. Подібні системи зазвичай будуються на тригерах, що реагують на певні події та ініціюють відповідні веб-сервіси (рис. 1.7).

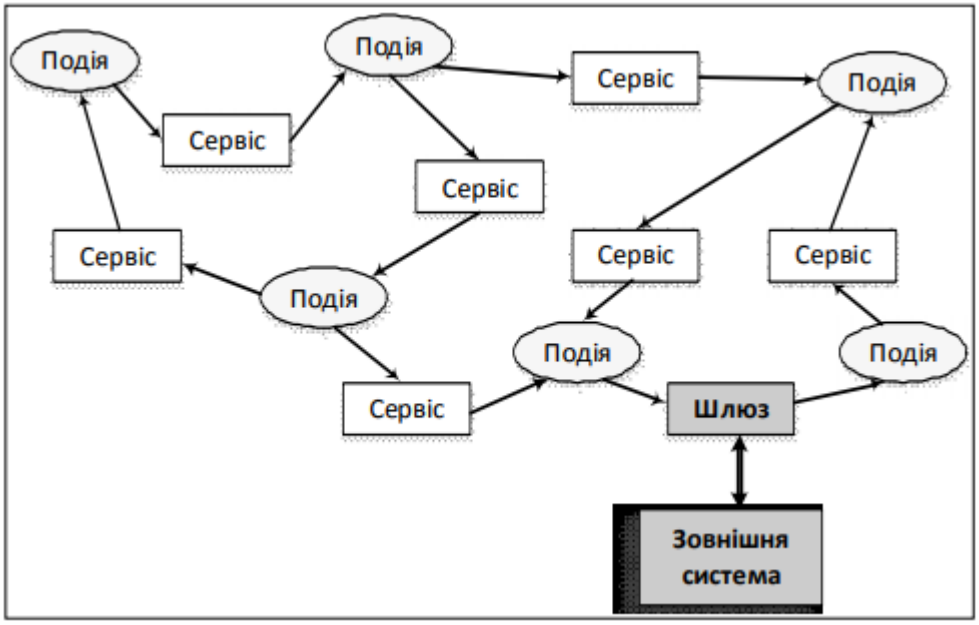


Рис. 1.7. EDA архітектура з подіями

Обидві нові архітектури SOA й EDA відрізняються від традиційної ар- хітектури тим, що бізнес-процес розбивається на малі й повторно використовувані процедури. Ці процедури формалізуються і реалізуються у вигляді IT-компонентів, які слабо пов’язані і можуть бути інтегровані в динамічний і гнучкий засіб. SOA і EDA намагаються замінити і розбити старі і великі успадковані системи й архітектури на більш гнучкі і багаторазово використовувані бізнес та IT-компоненти. Співвідношення між SOA й EDA є актуаль- ною темою для обговорення. Споживачі і дослідники дотримуються протилежних думок з приводу того, як сервіси та події мають взаємодіяти на
різних рівнях, включаючи бізнес, інформацію та інформаційні системи. До базових принципів EDA треба віднести:

- Роз’єднаність — джерело події знає тільки створену подію і не має жодного уявлення щодо її подальшого оброблення заходу або про зацікавлені сторони. Цей принцип використовується у ході інтеграції роз’єднаних додатків і бізнес-процесів. Процес складається з декількох етапів. У EDA етапи не залежать фізично або логічно один від одного, так що кожен може бути змінено, не викликаючи побічних ефектів, на інші, поки повідомлення щодо подій не змінюються.
- Повторюваність — вирішальним фактором у проектуванні подій є те, що вони мають бути спроектованими таким чином, щоб відповідати не тільки нинішнім вимогам, а й майбутнім потребам і поза сферою використан- тання. Події мають бути розроблені в загальному вигляді, який забезпечує їх повторне використання.
- Повідомлення у режимі реального часу — технічна інфраструктура має підтримувати створення в режимі реального часу подій і їх доставку. Персонал інфраструктури забезпечує в режимі реального часу перетворення інформації в знання, а потім в інтелектуальні наступні дії.
- Свобода діяти — повідомлення не вказує, які дії виконуватиме споживач події. Це звіт, а не вимога. Споживачу притаманна логіка, яка визначає, як він буде реагувати. Величезну кількість подій може бути вироблено, але тільки деякі з них будуть споживаними.
- Зв’язок «один-до-багатьох» — на одну конкретну подію може відгукнутися багато передплатників.
- Асинхронність — підтримка асинхронних операцій через повідомлення про події.
- Запуск процесу — потік управління, який визначається одержувачем, базується на розміщених подіях.
- Відсутність стану компонентів обробки подій — компоненти оброблення подій не мають стану, але сама подія буде мати стан, визначений у ній.

1.5. Мікросервісна архітектура

Мікросервісна архітектура - це архітектура, яка враховує можливість горизонтального масштабування ще на етапі проектування системи. Архітектурний стиль мікросервісів - це підхід, при якому єдине застосування будується як набір невеликих сервісів, кожен з яких працює у власному процесі і комунікує з іншими, використовуючи легковагі механізми, як правило, HTTP (рис 1.8). Процес розбиття монолітного додатку, на менші сервіси дозволяє природним чином розробляти додаток так, щоб використати стратегію горизонтального масштабування по осі Y. Ці сервіси побудовані навколо бізнес-потреб і розгортаються незалежно з використанням повністю автоматизованого середовища. Існує абсолютний мінімум централізованого управління цими сервісами. Самі по собі ці сервіси можуть бути написані на різних мовах і використати різні технології зберігання даних.

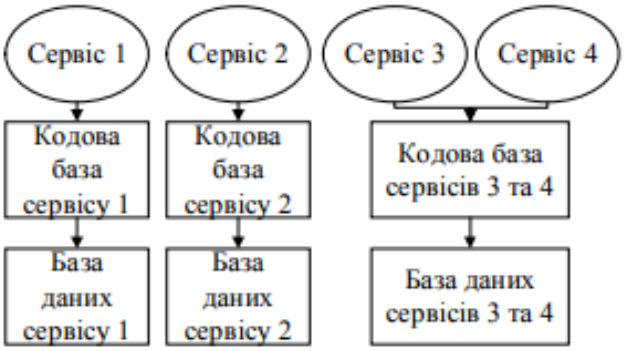


Рис. 1.8. Програмна реалізація мікросервісної архітектури

Мікросервісна архітектура використовує бібліотеки, але їх основний спосіб розбиття додатка - шляхом ділення його на сервіси. Розробники визначають бібліотеки як компоненти, які підключаються до програми і викликаються нею в тому ж процесі, тоді як сервіси - це компоненти, що виконуються в окремому процесі і комунікують між собою через веб-запити або видалені виклики процедур.

Порівнюючи мікросервісний підхід із іншими розглянутими раніше технологіями, можна виділити ряд визначальних переваг:

- Мікросервіси дозволяють збільшити продуктивність та ефективність систем завдяки зменшенню порогу обмежень відносно паралелізму та розподіленості реалізації окремих сервісів.
- Надійність мікросервісної архітектури та напрацювання на відмову значно вищі, оскільки домен поширення помилок обмежено окремим сервісом.
- Розподіл функціональності та масштабування рівнів виконується у відповідності до потреб бізнесу, виходячи із конкретних вимог до системи.

Мікросервіси описують способи дизайну додатків у вигляді набору незалежно розгорнутих сервісів, якому характерні організація сервісів навколо бізнес-потреб, автоматичне розгортання, перенесення логіки від шини повідомлень до приймачів та децентралізований контроль над мовами і даними. На додаток до можливості незалежного розгортання і масштабування кожен сервіс також отримує чітку фізичну межу, яка дозволяє різним сервісам бути написаними на різних мовах програмування. Архітектура мікросервісів використовує бібліотеки, але їх основний спосіб розбиття додатку здійснюється завдяки ділення його на сервіси. У свою чергу сервіси - це компоненти, що виконуються в окремому процесі та взаємодіють між собою через веб-запити або remote procedure call (RPS), а також містить безліч процесів, які завжди розробляються і розгортаються паралельно. Додатки, побудовані з використанням мікросервісної архітектури, містять власну доменну логіку. Принцип їх дії складається з отримання запиту, застосування логіки і надсилання відповіді. Замість складних протоколів, таких як WS або BPEL, вони використовують прості REST- протоколи. Виходячи з вищезазначеного, мікросервіси варто розглядати як пріоритетне архітектурне рішення при розробці сучасних обчислювальних систем. Проте існує потреба в розробці алгоритмічного, математичного та програмного забезпечення для спрощення використання мікросервісного підходу в реальних рішеннях.

1.5.1 Збірка і розгортання мікросервісів

У будь-якому архітектурному рішенні існують компроміси. Зокрема, мікросервісна архітектура спричиняє за собою серйозну зміну процесу

розгортання . Доводиться обслуговувати цілу екосистему невеликих сервісів, а не єдиний, чітко визначений монолітний додаток. Наслідком використання сервісів як компонентів є необхідність проектування додатків так, щоб вони могли працювати при відмові окремих сервісів. Будь-яке звернення до сервісу може не спрацювати через його недоступність. Клієнт повинен реагувати на це настільки терпимо, наскільки це можливо. Це є недоліком мікросервісної архітектури в порівнянні з монолітною, оскільки це вносить додаткову складність в розробку. Як наслідок, при розробці мікросервісної архітектури слід постійно думати над тим, як недоступність сервісів повинна впливати на досвід використання системи. Для симуляції збоїв вимагається штучно викликати відмови сервісів, і навіть дата центрів для тестування відмовостій- кості додатку і служб моніторингу.

Оскільки сервіси можуть відмовити у будь-який час, дуже важливо мати можливість швидко виявити неполадки і, якщо можливо, автоматично відновити працездатність сервісу.

Мікросервісна архітектура робить великий акцент на моніторингу додатку в режимі реального часу, перевірці як технічних елементів (наприклад, як багато запитів в секунду отримує база даних), так і бізнес-метрик (наприклад, як багато замовлень в хвилину отримує додаток). Семантичний моніторинг може попередити про проблемні ситуації, дозволяючи команді розробки підключитися до дослідження проблеми на самих ранніх стадіях.

Це особливо важливо у випадку з мікросервісною архітектурою, оскільки розбиття на окремі процеси і комунікація через події призводять до несподіваної поведінки. Моніторинг украй важливий для виявлення небажаних випадків такої поведінки і швидкого їх усунення.

Поміщення компонентів в сервіси додає можливість точнішого плану-вання реліза системи. З монолітом будь-які зміни вимагають перезбірки і розгортання усього додатку. У разі мікросервісної архітектури вимагається розгорнути тільки ті сервіси, які змінилися. Це дозволяє спростити і прискорити процес релізу.

Недолік такого підходу полягає в тому, що, що зміни в одному сервісі можуть бути не погоджені з іншими сервісами. Для забезпечення узгодженості змін в сервісах, і працездатності системи в цілому, вимагається безперервно здійснювати збірку, тестування, перевірку інтеграції.

Для побудови мікросервісної інфраструктури вимагається інвестувати час в автоматизацію процесів розробки (виділення ресурсів, складання, автоматичне тестування, перевірка інтеграції, розгортання). Це дозволить зробити процес однаковим і прозорим, так щоб розгортання системи з однієї складової (монолітні додатки) або з багатьох (мікросервіси) не відрізнялися по своїй складності.

1.5.2. Автоматизація процесів

Техніки автоматизації інфраструктури значно еволюціонували за останні декілька років. Еволюція хмарних технологій зменшила операційну складність побудови, розгортання і функціонування мікросервісної інфраструктури. Компанія повинна мати можливість розгорнути і ініціалізувати новий сервер впродовж декількох годин. Найпростіший спосіб зробити це - використати технології хмарних обчислень. Для того, щоб розгорнути новий сервер в вказаний час без використання хмарних технологій вимагається повністю автоматизувати цей процес у власній інфраструктурі. На перших кроках розробки мікросервісної системи цього не вимагається, але надалі, на етапах тестування і впровадження цей пункт є обов'язковим.

Автоматизація збірки та перевірка. Для перевірки інтеграції нового коду зі старим вимагається забезпечити можливість його перевірки. Це досягається за рахунок неперервних процесів. Оскільки збірка і перевірка здійснюється після кожного комміту коду в системі контролю версій, вимагається забезпечити мінімальний часовий проміжок між фіксацією коду і його перевіркою.

Автоматизація розгортання додатка. Необхідно забезпечити можливість швидкого розгортання великої кількості сервісів, як на тестовому, так і на промисловому оточеннях. На початкових етапах розробки ці операції можна

виконувати в ручному режимі, проте, надалі, ці процеси необхідно автоматизувати в обов'язковому порядку.

Невеликою кількістю серверів можливо управляти вручну, вивантажувати файли журналів, розгортати додатки і перевіряти утилізацію ресурсів в запущених процесах. При управлінні серверами вручну, адміністратор значно обмежений у своїх діях. Для кожного сервера має бути відкрите термінальне вікно, підключене до сервера по SSH протоколу. І чим більше серверів, за якими потрібно стежити, тим нижче швидкість роботи адміністратора.

Автоматизація також є процесом, при якому можна бути упевненим, що розробники зберігають високу продуктивність, концентруються тільки на імплементації бізнес вимог в додатку. Одним із способів для збереження продуктивності є забезпечення можливості розробникам самостійно забезпечувати сервіси або групи сервісів. В ідеальному випадку, розробники повинні користуватися тим же самим набором інструментів для розгортання, що і для промислової версії, щоб мати можливість виявляти проблеми раніше. Вибір технології для автоматизації украй важливий. Культура автоматизації ключовий момент, якого потрібно досягти, щоб не збільшувати складність управління мікросервісною архітектурою.

Висновки

В даному розділі були проаналізовані основні методи для побудови вебдодатків, розглянуті переваги та недоліки монолітної архітектури, сервіс-орієнтованої архітектури (SOA), подійно-орієнтованої архітектури, мікросервісної архітектури. На основі цього аналізу були сформулювані наступні задачі, для подальшого дослідження:

- Дослідження оптимізації процесів розгортання мікросервісів.
- Технічний розляд основних методів використання неперервної інтеграції в мікросервісній архітектурі.
- Аналіз способів розгортання сервісів та розгляд шаблонів проектування мікросервісів.

РОЗДІЛ 2
АНАЛІЗ ІНФРАСТРУКТУРИ ТА ОПТИМІЗАЦІЯ ПРОЦЕСІВ СТВОРЕННЯ І РОЗГОРТАННЯ МІКРОСЕРВІСІВ

Розгортання мікросервісів є нетривіальним процесом. Якщо не пропрацювати процес розгортання якісно, то з часом проблеми в цій області будуть рости експоненціально. Технології в області розгортання зробили крок далеко уперед за останні декілька років.

2.1. Неперервна інтеграція

Неперервна інтеграція (Continuous integration) існує вже досить давно. Ця практика розробки націлена на те щоб усі компоненти системи були синхронізовані між собою. Коли в системі контролю версій фіксується новий код, починається процес різних перевірок, наприклад, чи компілюється код і чи успішно завершується модульне тестування.

Одним з етапів цього процесу є створення артефактів, які використовуються в подальших перевірках. Наприклад, в розгортанні і старті сервісу для запуску інтеграційних тестів. В ідеальному випадку бажано збирати ці артефакти тільки одного разу і використати їх для усіх подальших розгортань цієї версії коду. Це робиться для того, щоб упевнитися в тому, що розгорнутий артефакт є тим самим, який був протестований. Для втілення цієї ідеї в життя вимагається поміщати ці артефакти в різні репозиторії, що надаються або самою системою неперервної інтеграції, або іншою системою.

Використання неперервної інтеграції дає велику кількість переваг. За допомогою неперервної інтеграції можна отримати швидку відповідь на питання про якість коду, який потрапив в систему контролю версій. Також ця практика дозволяє створювати артефакти, які у свою чергу теж діляться на версії. Існує можливість також перетворювати артефакт.

Новий код повинен працювати зі старим без конфліктів. Якщо перевірка коду відбувається рідко, то подальша перевірка інтеграції буде набагато складніша.

При безперервній інтеграції без автоматичних тестів перевірка коду здійснюється лише синтаксично, тобто можна бути упевненим тільки в тому, що код компілюється, але немає упевненості в тому, що функціональність, яка додається з цим кодом, не нашкодить іншим сервісам.

Успішно завершене збірки означає, що новий код успішно інтегрований. Збірка, завершене з помилкою навпроти, означає, що код не інтегрується. Вимагається припинити подальше додавання нового коду, який не спрямований на виправлення помилки, при якій збірка не проходить усі тести. Якщо дозволити внести подальші зміни час на виправлення істотно зросте.

2.1.1. Використання неперервної інтеграції в мікросервісній архітектурі

При використанні неперервної інтеграції в мікросервісній архітектурі вимагається обдумати, як збірки співвідноситимуться з окремими сервісами. Вимагається переконатися в можливості зміни окремого сервісу і його незалежного розгортання від усієї системи.

Існують різні способи співвідношення збіркок і сервісів. У простому випадку увесь код сервісів може зберігатися в одному репозиторії, до якого відноситься єдиний процес збірки (рис 2.1).

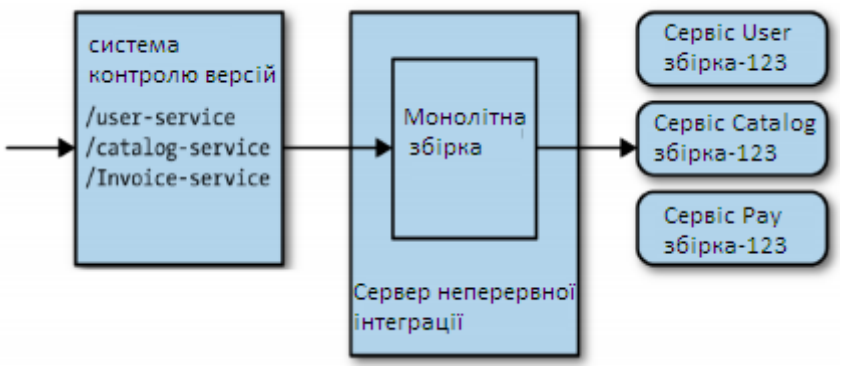


Рис. 2.1. Монолітна збірка

Будь-які зміни в вихідному коді будь-якого сервісу почнуть процес неперервної інтеграції, який запустить усі перевірки і створить безліч артефактів, по кожному на сервіс.

Цей спосіб найбільш простий в реалізації. Менше репозиторіїв, за якими вимагається стежити і процес збірки концептуально виглядає простіше. З точки зору розробника все також виглядає досить прямолінійно. Досить просто зафіксувати новий код в системі контролю версій. Якщо розробник працює відразу над декількома сервісами, то для запуску процесу складання усіх сервісів потрібно одну фіксацію змін.

Ця модель працює добре, якщо не вимагається розгортати сервіси окремо. На практиці цю модель слід уникати. Нею можна користуватися тільки на самих ранніх етапах розробки або якщо тільки одна команда розробляє усі сервіси. Навіть зміна єдиного рядка в вихідному коді сервісу запускає процес збірки і перевірки усіх сервісів.

Це займе значно більше часу, ніж збірка і перевірка одного сервісу, де сталися зміни. Також цей підхід уповільнює цикл, в якому усі зміни після збірки і перевірки потрапляють в промислову експлуатацію. Набагато важливіша проблема в цьому підході полягає в тому, що невідомо який артефакт потрібно розгорнути, всі або тільки той сервіс, в якому сталися зміни. У такому підході зазвичай розгортаються усі артефакти відразу. Таким чином, будь-яка зміна в окремому сервісі, яка не пройшла перевірку, обмежує подальшу розробку інших сервісів, поки зміна того, що викликало помилку інтеграції не буде виправлена.

Інша варіація першого підходу полягає в тому, щоб мати єдиний репозиторій з вихідним кодом, що складається з модулів і робити збірки неперервної інтеграції по кожному з модулів окремо (рис. 2.2).

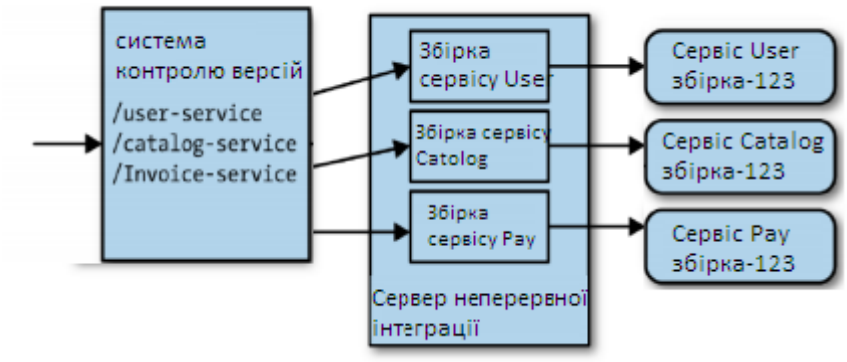


Рис. 2.2. Модульна збірка

З одного боку, цей підхід вигідний тим, що процес фіксації змін залишається таким же простим. Існує тільки один репозиторій. З іншого боку, існує вірогідність того, що часто зміни можуть вноситися відразу в декілька сервісів, що тягне за жорсткіше зв'язування сервісів між собою. Є і третій варіант підходу до співвідношення сервісів і збірок, один до одного. Почат-ковий код окремо взятого сервісу зберігається у своєму репозиторії і для кожного сервісу існує своя сбірка (рис. 2.3).

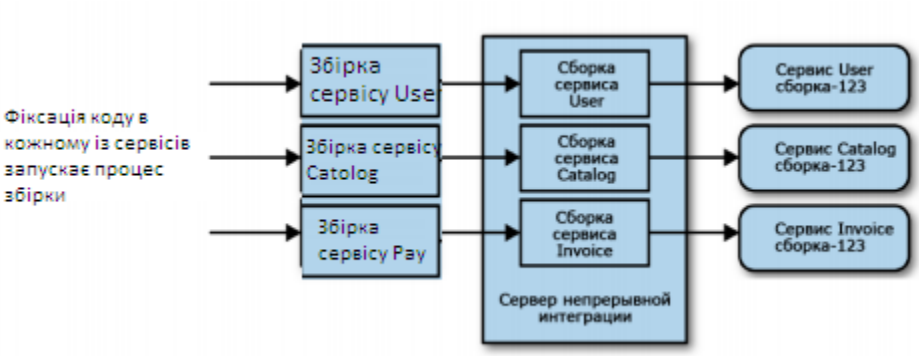


Рис 2.3. Роздільна збірка

При зміні вихідного коду окремого сервісу виконується тільки його збірка і тестування. Таким чином, виходить тільки єдиний артефакт для розгортання. Зіставлення з командами розробки при цьому підході стає прозоріше. У кожної команди, розробляючої сервіс, є свій репозиторій з вихідними кодами і своя збірка. Вносити зміни між різними репозиторіями стає складніше, але управляти процесом складання і виправленням помилок під час інтеграції стає легше. Тести для конкретного мікросервісу повинні зберігатися в тому ж репозиторії, де зберігається його вихідний код**.**

За допомогою підходу неперервної інтеграції процес збірки сервісу, який потрібно розгорнути, виконується повністю автоматично.

2.1.2. Етапи збірки та незперервне постачання

На самому початку практичного використання неперервної інтеграції було відмічено наявність декількох етапів при збірці проекту. Найбільш яскравий приклад - це етап тестування. У збірці може бути безліч швидких тестів і невелика кількість довгих тестів, які охоплюють усю бізнес-логіку сервісу. Якщо

запускаються усі тести підряд не оглядаючись на їх час виконання, неможливо отримати швидкий зворотний зв'язок, коли тести виконуються з помилкою. Для відповіді потрібно дочекатися, коли усі тести виконаються. Таким чином, якщо швидкий тест виконався з помилкою, вимагається дочекатися виконання усіх повільних тестів перш ніж отримати результат. Вирішення цієї проблеми полягає в тому, щоб додати різні етапи тестування в процес збірки. Один етап для швидких тестів, інший - для повільних. Концепція етапів збірки дає прекрасну можливість спостерігати прогрес збірки в цілому, даючи зворотний зв'язок про якість функціонування нового коду.

Процес неперервного постачання будується на концепції етапів збірки. Неперервне постачання - це підхід, в якому є можливість постійно отримувати зворотний зв'язок про кожну зміну в коді і таким чином вважати чи являється ця зміна кандидатом в реліз, який буде випущений в промислову експлуатацію. Щоб повністю охопити цю концепцію вимагається моделювати усі процеси від фіксації змін в коді до виходу сервісу в промислову експлуатацію. При неперервному постачанні розширюється ідея етапів збірки, щоб повністю охопити процес, включаючи автоматичне і ручне тестування. На рис. 2.4 зображені етапи збірки сервісу.

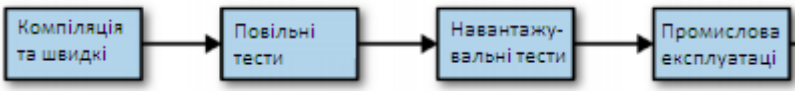


Рис. 2.4. Етапи збірки

Як тільки версія вихідного коду проходить етапи автоматичного тестування, вона потрапляє на етап ручного тестування (UAT). Це повинно бути показано в спеціальних інструментах, які автоматизують процес неперервного постачання. Після моделювання кожного етапу, який розпочинається з фіксацій зміни в коді, а закінчується розгортанням в промислову експлуатацію можна точно судити про якість програмного забезпечення і значно понизити тимчасові витрати між різними релізами системи, оскільки є можливість централізова-ного спостереження за кожним етапом збірки модуля. Таким чином, можна точково вносити удосконалення в процес розробки.

2.2. Артефакти збірки

При використанні мікросервісної архітектури вимагається бути упевненим в тому, що сервіси незалежні один від одного. Вимагається мати окремий набір етапів для кожного сервісу, також, як і мати свою збірку під кожен сервіс при використанні неперервної інтеграції. Слід зазначити, що по етапах складання рухається деякий артефакт, який може мати різний вигляд і розміри. Нижче розглядаються форми, які може приймати артефакт.

2.2.1. Платформо-залежні артефакти

Кожен технологічний стек має на увазі свій набір артефактів, разом з інструментами, які створюють і встановлюють ці артефакти. Розробники на якійсь певній платформі добре знайомі з інструментами специфічними для збірки артефактів, які відносяться до їх платформ.

З точки зору мікросервісної архітектури для запуску окремих сервісів тільки артефактів недостатньо. Потрібно певний спосіб установки і конфігурування окремого програмного забезпечення для повноцінного розгортання і запуску артефактів. Безпосередньо для таких завдань існують інструменти автоматичної конфігурації, такі як Puppet.

Інший складний момент полягає в тому, що кожен артефакт зав'язаний на конкретному технологічному стеку. Ця особливість може значно ускладнити процес розгортання при змішуванні технологій. Такий аспект можна співвіднести зі спробою розгортання декількох сервісів спільно. Це істотно ускладнює процес розгортання. Автоматизація процесу розгортання може значно полегшити цей процес. Puppet підтримує можливість автоматизувати процес розгортань артефактів для різних платформ, але існують інші види артефактів, з якими набагато легше працювати.

2.2.2. Артефакти операційної системи

Один із способів позбавитися від проблем при використанні артефактів конкретної платформи - це використати артефакти специфічні для операційної системи, в якій здійснюватиме процес розгортання. Перевага при використанні

артефактів операційної системи полягає в тому, що з точки зору розгортання не вимагається турбуватися про технологічний стек вмісту в артефакті. Використовуються стандартні інструменти для установки артефакту на операційну систему. Також є інструменти для видалення і отримання інформації про встановлені пакети. Окрім цього, операційна система може надати репозиторії, в які потраплятимуть артефакти, створені на етапі неперервної інтеграції. Вбудований модуль диспетчера пакетів операційної системи може полегшити автоматичне створення артефактів, також, як і інструмент Puppet при побудові платформо-залежних артефактів.

Один з недоліків цього підходу полягає у відмінності механізмів управління залежностями пакетів в різних ОС. Так, для Linux платформи артефакти містять в собі посилання на залежні модулі, які будуть автоматично встановлені з артефактом. Windows платформа припускає, що артефакт вже містить в собі усі зовнішні залежності.

Інший недолік цього підходу проявляється, якщо розгортання здійснюється на декілька різних ОС. Для різних операційних систем процес складання артефакту виглядає по-різному. Якщо компанія створює ПЗ, яке сама не встановлюватиме, то вибору як-такого немає, але, якщо ПЗ встановлюється самою компанією, вимагається уніфікувати або хоча б зменшити набір різних операційних систем.

При переході від платформо-залежних артефактів до артефактів специфічних для операційної системи значно спрощується процес розгортання. Відсутні складні скрипти розгортання. Зокрема, це помітно при використанні Linux. Цей спосіб значно спрощує розгортання мікросервісів, що використовують різні технологічні стеки.

2.2.3. Образ віртуальної машини як артефакт

Одним з критичних при використанні інструментів автоматизації конфігурації параметрів є час, що витрачається на виконання скриптів.

Розглянемо приклад забезпечення сервера і налаштування його для розгортання Java додатка. Впершу чергу вимагається забезпечити наявність

сервера. При використанні хмарних технологій це можливо зробити за декілька хвилин. Далі вимагається встановити на сервер JVM, що ще займає до десяти хвилин, і тільки після цього починається сам процес розгортання додатка.

Наведений вище приклад досить тривіальний. Зазвичай на сервер вимагається встановити ще декілька десятків часто використовуваних застосувань, служб моніторингу або журналювання. Через деякий час додається ще ряд додатків обов'язкових до установки, що веде за собою ще більші витрат часу на розгортання. Також вимагається постійно стежити за скриптами автоматичної конфігурації, часто змінюючи їх і не допускати випадків ресинхронизації конфігурації працюючого сервера від скриптів конфігурації. Якщо використовується платформа, яка запускається на вимогу, щодня запускає і зупиняє різні сервери, такій продуктивності інструментів авто-матичної конфігурації може бути недостатньо.

Ця проблема виникає, якщо старт нових серверів зв'язаний з розробкою або неперервною інтеграцією. Це істотно уповільнює зворотний зв'язок. Це також веде до збільшення часу простою системи, коли додаток розгортається в промислову експлуатацію. Якщо до системи пред'являються вимоги, при яких простій заборонений, вимагається отримати налагоджений сервер ще до того, як процес розгортання додатка почався.

Одним із способів зменшити час, що витрачається на забезпечення і налаштування сервера, є використання образу віртуальної машини, яка вже містить усі потрібні залежності. Усі платформи віртуалізації дозволяють створювати власні образи. При використанні такого підходу досить створити образ віртуальної машини і розгорнути в нього нову версію сервісу.

Оскільки образ створюється одного разу, подальші запуски подібного оточення можна використати повторно. Це призводить до значного скорочення часу на забезпечення сервера. Образ вимагається міняти тільки у тому випадку, якщо міняються його залежності.

Цей підхід теж має свої недоліки. Складання образу може займати велику кількість часу. Наступний недолік полягає у розмірі образу, він може досягати декількох гігабайт.

Історично так склалося, що набір інструментів для збірки образів розрізняється і залежить від типу платформи де буде запущений образ. Так, наприклад, процес складання образу VMWare відрізняється від складання образу для AWS, Vagrant або Rackspace. Ця проблема може бути відсутньою, якщо використовується єдина платформа.

Існують різні інструменти, які полегшують завдання створення образів, наприклад, Packer. Використовуючи конфігурації Chef або Puppet, ці інструменти створюють образи для різних платформ з єдиної конфігурації. Це означає, що можна створити образ для розгортання в промислову експлуатацію з використанням AWS, а для розробки і тестування образ Vagrant з однієї конфігурації.

Таким чином, можна створювати образи віртуальних машин, які містять усі залежності, щоб прискорити зворотний зв'язок під час процесу неперервної інтеграції. Але цей спосіб створення оточення можна оптимізувати і під час збірки розгортати нову версію сервісу безпосередньо в образ віртуальної машини. З'являється новий тип артефакту - образ віртуальної машини з інкапсульованим в нього сервісом. При запуску образу нова версія сервісу вже встановлена.

Також, як і з артефактами специфічними для операційної системи, образи віртуальних машин мають достатній рівень абстракції від різних стеків технологій використаних при створенні сервісів.

2.3. Оточення функціонування сервісів

Для коректного функціонування процесу неперервного постачання потрібно, щоб сервіси, що розробляються, розгорталися в різному оточенні:

- оточення для повільних тестів;
- оточення для ручного тестування;
- оточення для перевірки продуктивності;
- оточення промислової експлуатації.

Артефакт сервісу має бути одним і тим же в усьому оточенні. У найпростішому випадку це оточення розрізнятиметься конфігураціями і

хостами. Але часто оточення відрізняються в значно більшому ступені. Наприклад, в оточенні промислової експлуатації хости розподілені на декілька дата-центрів, пере якими стоїть балансувальник навантаження, а на тестовому оточенні усі сервіси розгорнуті на одному хості.

У таких ситуаціях вимагається, щоб оточення для ручного тестування якомога більше відповідало оточенню промислової експлуатації, оскільки помилки, які відтворюються тільки на певному оточенні досить складно виправити. Управляти оточенням в монолітній системі досить складно, особливо в тих системах, розгортання яких складно автоматизувати. Коли мова заходить про різне оточення для мікросервісів, це завдання стає таким, що ще більше лякає.

2.4. Методи розгортання сервісів на хостах

Одним з перших питань, що виникають при розробці мікросервісної архітектури, є питання: "скільки мікросервісів має бути на одному хості"?

2.4.1 Декілька сервісів на хост

Модель розгортання декілька сервісів в одному хості виглядає досить привабливо з декількох причин. У тому разі якщо одна команда займається інфраструктурою, а інша розробкою програмного забезпечення, завантаження команди інфраструктури збільшується в стільки разів, скільки хостів потрібно обслуговувати. На рис 2.5 зображена модель розгортання декількох сервісів на хост.

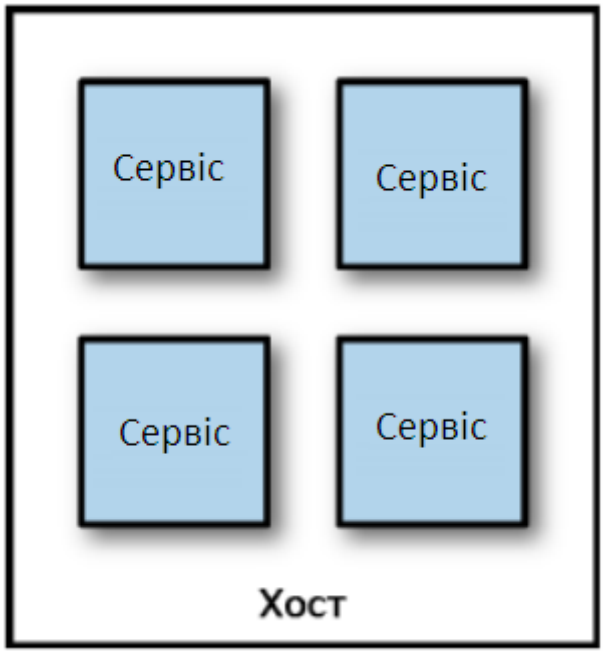


Рис. 2.5. Модель розгортання декілька сервісів на хост

Якщо сервіси знаходяться на одному хості, то збільшення кількості сервісів ніяк не позначиться на завантаженні команди, яка підтримує інфраструктуру. Інша причина - це вартість. Навіть якщо є доступ до платформи віртуалізації, яка дозволяє забезпечувати і міняти розміри віртуальних хостів, віртуалізація додає накладні витрати на ресурси реальної машини.

Ця модель чимось нагадує контейнери додатків. У деякому роді використання контейнера додатків - це окремий випадок моделі декілька сервісів на хост. Ця модель також спрощує життя розробникам. Розгортання декількох сервісів на один хост промислової експлуатації нічим не відрізняється від розгортання сервісів на тестове оточення або оточення розробки. Для того, щоб поглянути на альтернативну моделі вимагається зберегти таку ж просту концепцію для розробників.

Ця модель також не позбавлена вад. Істотно ускладнюється моніторинг системи. Наприклад, при відстежуванні роботи CPU вимагається відстежувати роботу CPU на окремому сервісі або відразу на усіх? Якщо один сервіс навантажений сильніше за інших, це може істотно зменшити кількість доступних ресурсів для інших сервісів. Спочатку успішно працюючі разом сервіси на одному хості, при нерівномірному навантаженні можуть несприятливо впливати на інші сервіси на цьому ж хості. Це робить аналіз проблем набагато складніше.

Розгортання сервісів теж може бути ускладнене, оскільки вимагається завжди бути упевненим в тому, що розгортання одного сервісу не вплине на інші вже розгорнуті сервіси. Так, наприклад, якщо для налаштування хоста використовується Puppet, але кожен сервіс має різні конфліктуючі залежності, ці проблеми вирішуються нетривіальним чином.

Таким чином, це спрощення позбавляє мікросервісну архітектуру однієї з основних переваг - незалежний випуск нових версій сервісів. Якщо все ж позбавитися від розгортання декількох сервісів на одному хості неможливо, вимагається завжди тримати в голові правило, що сервіси можуть розгортатися незалежно і стежити за цим вручну.

Це обмеження також порушує автономність команд, оскільки всі конфлікти при розгортанні їм доведеться вирішувати спільно.

Факт розгортання на одному хості різних сервісів негативно позначається на масштабованості того сервісу, який реально цього потребує. Якщо один мікросервіс обробляє велику кількість даних або його функціональність досить важлива, можливо, знадобиться додаткове налаштування хоста або винесення його в окремий сегмент мережі. Якщо усі сервіси знаходяться на одному хості, то всі вони вимушені знаходитися в одному оточенні.

Насправді, усі напрацьовані практики по відношенню до розгортання сервісів і управління хостом грунтуються на оптимізації дефіциту ресурсів. Раніше, для того, щоб отримати новий хост вимагалося купити або орендувати нову фізичну машину.

Цей процес був досить тривалий і вимагав немало фінансових ресурсів. Але так звані on - demand комп'ютерні платформи кардинально понизили вартість комп'ютерних ресурсів і удосконалили технології віртуалізації.

2.4.2. Один сервіс на хост

З моделлю одним сервісом на хост, є можливість позбавитися від недоліків моделі безлічі сервісів на один хост. Проблема з моніторингом значна полегшується, а масштабування стає значно простіше. Також гіпотетично зникає

проблема єдиної точки відмови. На рис. 2.6 зображена модель розгортання одного сервісу на хост.

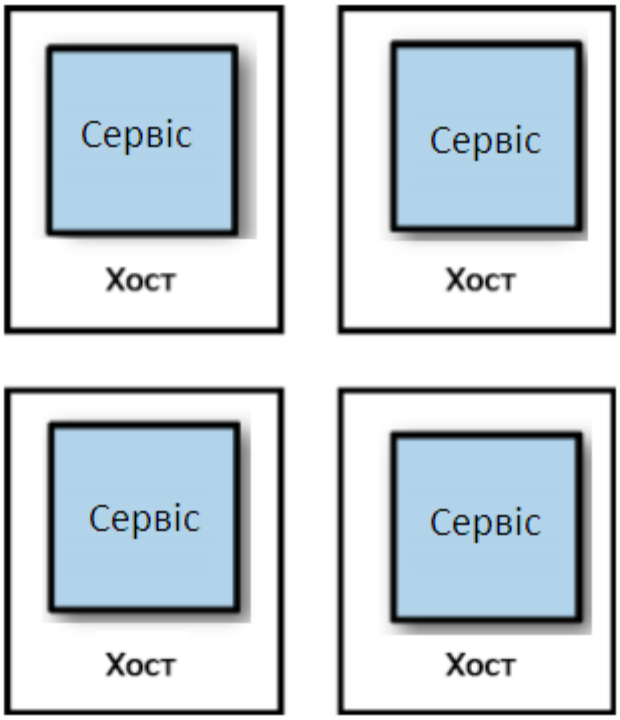


Рис. 2.6. Модель розгортання одного сервісу на хост

Вихід з ладу одного хоста повинен впливати тільки на один сервіс, хоча це не завжди відповідає дійсності, якщо використовується платформа віртуалі-зації. Масштабування відбувається значно простіше, а проблеми з безпекою вирішуються точково, концентруючись на якомусь конкретному сервісі або хості.

Найголовніше - стають доступні альтернативні способи розгортання, наприклад, образи віртуальних машин як артефакти або заздалегідь підготовлені образи під кожен сервіс для подальшого їх розгортання.

Ця модель значно знижує складність для мікросервісної архітектури. Модель один сервіс на один хост значна простіше за рахунок відсутності загального оточення і можливості незалежного масштабування. З іншого боку, збільшення кількості хостів спричиняє за собою негативні моменти. Вимагається підтримувати більше серверів, а також збільшуються витрати на обслуговування. Незважаючи на це, саме ця модель вважається еталонною для микроервисной

архітектури. Надалі будуть розглянуті моменти знижуючі негативні аспекти цієї моделі.

2.5. Віртуалізація інфраструктури

Один з основних способів для можливості управління великою кількістю хостів, є спосіб ділення існуючих фізичних машин на дрібніші частини. Традиційна віртуалізація така як VMWare або що використовується в AWS принесла величезні вигоди в зменшенні накладних витрат при управлінні хостами. Проте з'явилися нові досягнення в цій області, які розкривають нові можливості для мікросервісної архітектури.

Виконання одного екземпляра мікросервісу на сервері без віртуалізації не економічно. Таким чином, у більшості випадків, розгортання безлічі мікросервісів здійснюється на звичайному сервері без віртуалізації. Виконання декількох сервісів на звичайному сервері без віртуалізації може привести до конфліктів налаштувань оточення між сервісами. Мікросервіси не ізольовані один від одного якщо вони виконуються на одній і тій же машині. Також, сервіси, розгорнуті на одному сервері, можуть споживати ресурси інших сервісів, знижуючи їх продуктивність.

2.5.1. Традиційна віртуалізація

Обслуговувати велике число хостів дорого. При використанні фізичного сервера на хост, це твердження очевидне. При використанні фізичного сервера модель один сервіс на один хост є єдиним прийнятним. Віртуалізація дозволяє розділити фізичний сервер на незалежні хости, кожен з яких виконуватиме різні сервіси. Таким чином якщо потрібно модель один сервіс на один хост, досить розділити фізичну інфраструктуру на дрібні частини. Цей підхід має істотний мінус. Розділення фізичного сервера має свої накладні витрати. На рис. 2.7 зображені різні слої, залучені в процесі традиційної віртуалізації.

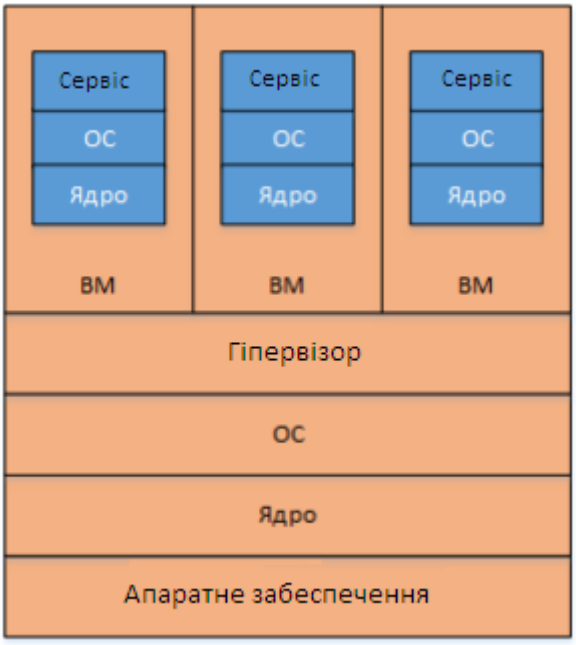


Рис. 2.7 Слої традиційної виртуализації

Цей вид віртуалізації використовується в AWS, VMWare, Xen. Він називається віртуалізація другого типу. Віртуалізація першого типу відноситься до технології де віртуальна машина виконується безпосередньо на апаратному забезпеченні, а не на іншій операційній системі. На фізичній інфраструктурі виконується код основної операційної системи. На цій операційній системі виконується ПЗ, яке називається гіпервізор, що виконує 2 основні функції. По-перше, він співвідносить ресурси CPU і пам'яті фізичної машини на віртуальний хост. По-друге, є слоєм, що управляє, дозволяє маніпулювати віртуальними машинами.

Всередині віртуальної машини працює абсолютно ізольована система, в якій може виконуватися абсолютно інша операційна система зі своїм ядром. Віртуальна машина повністю відокремлена від фізичного хоста, що пролягає нижче, і інших віртуальних машин за допомогою гіпервізора.

Одна з проблем при використанні віртуальних машин полягає в необхідності виділення ресурсів для її функціонування. Повністю обмежуючи перевикорстання виділених ресурсів, таких як CPU, I/O і пам'яті, іншими віртуальними машинами або основною операційною системою. Чим більше віртуальних машин управляється гіпервізором, тим більше ресурсів потрібно.

Віртуальні машини досить ваговиті. Таким чином запуск безлічі невеликих віртуальних машин на фізичній машині не ефективний оскільки має великі накладні витрати, по суті даремна витрата ресурсів, на забезпечення роботи гіпервізора. У випадку якщо розгортання мікросервісів здійснюється в одній віртуальній машині, виникатимуть ті ж проблеми, які були описані вище.

Наприклад, при розробці мікросервісів створених з використанням технології Java, загальне використання сервісами віртуальної машини або чистого сервера, означає використання декількома сервісами однієї JRE (середовище виконання Java додаток). Будь-яка зміна встановленою на хості JRE зачіпає усі сервіси, розгорнуті на цій машині. Також, якщо вказані параметри специфічні для операційної системи або бібліотеки, або певні оптимізації необхідні тільки конкретного мікросервісу, налаштуваннями складніше управляти, оскільки вони застосовуються відразу для усіх сервісів.

2.6. Технологія контейнеризації

Один з принципів мікросервісної архітектури свідчить про те, що сервіс має бути ізольованим і автономним, повністю інкапсулюючи оточення виконання. Для дотримання цього принципу усі компоненти, такі як операційна система, середовище виконання і виконуваний код мікросервісу має бути автономним і ізольованим. Єдиний спосіб досягти цього - наслідувати підхід один мікросервіс на одну віртуальну машину. Проте це приведе до недостатньої утилізації ресурсів віртуальної машини. Також у багатьох випадках через додаткові накладні витрати можуть звестись усі переваги мікросервісів нанівець.

Технологія контейнеризації далеко не нова і не новаторська технологія. Вона використовується вже досить тривалий час. Проте, ця технологія стала набирати значну популярність з приходом хмарних технологій. Недоліки традиційних віртуальних машин стали каталізатором зростання популярності контейнерів. Постачальники інструментів для роботи з контейнерами, наприклад, Docker, значною мірою спростили технологію контейнеризації, що сприяло впровадженню цієї технології в широкі маси. Популярність DevOps і

мікросервісна архітектура також прискорили переродження технології контейнеризації.

Технологія контейнеризації надає приватне оточення в операційній системі. Ця технологія також називається віртуалізація операційної системи. У цьому підході, ядро операційної системи надає ізольований віртуальний простір. Кожен з віртуальних просторів називається контейнером. Контейнери дозволяють процесам створювати ізольоване оточення в операційній системі що містить ці контейнери. На рис. 2.8 зображені різні слої, залучені в процес контейнеризації.

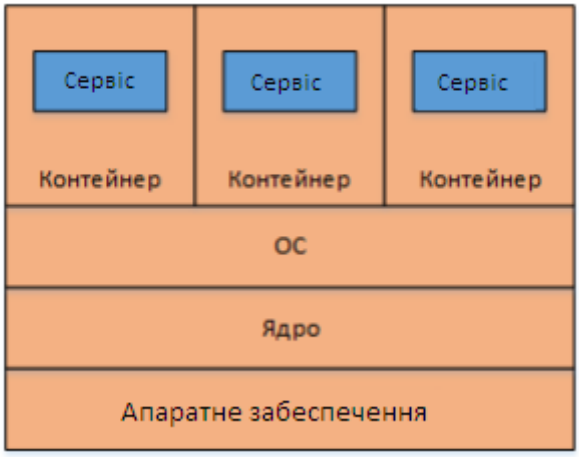


Рис. 2.8. Слої контейнеризації

Контейнери, це простий механізм для складання і постачання слабозв'язаних компонентів програмного забезпечення. У загальному випадку, контейнери упаковують усі виконувані файли і бібліотеки, які потрібні для запуску додатка. Контейнери повністю ізолюють наступні елементи:

- файлову систему;
- IP адресу;
- мережеві інтерфейси;
- внутрішні процеси;
- простори імен;
- бібліотеки операційної системи;
- бінарні файли додатка;
- залежності;

- файли конфігурації додатка.

Усі інструменти контейнеризації грунтуються на функціональності ядра Linux. Основні компоненти ядра Linux для контейнеризації перераховані нижче:

- простори імен;
- групи, що управляють.

Простори імен (namespaces) використовуються для створення ізольованого оточення, яке і називається контейнер. Коли запускається контейнер, докер створює нові простори імен для цього контейнера. Простір імен представляють собою слой ізоляції. Кожен аспект роботи контейнера виконується в окремому просторі.

Нижче перераховані групи просторів імен :

- PID (process identifier);
- мережева взаємодія;
- міжпроцесова взаємодія;
- точки монтування файлової системи;
- ізольоване ядро і ідентифікатори версій.

Групи (control groups), що управляють, також, як і простори імен містяться в ядрі Linux. Групи, що управляють, обмежують ресурси, споживані додатком. Також вони дозволяють ядру докера ділити доступні апаратні ресурси між контейнерами і обмежувати або розширювати ці ресурси за потреби. Наприклад, можна обмежити використання пам'яті для якогось певного контейнера.

Різні організації використовують мільярди контейнерів. Окрім цього, багато великих компаній інвестують в технологію контейнеризації. Docker значно випереджає конкурентів і підтримується багатьма великими постачальниками операційних систем, а також хмарними провайдерами. Також в поточний час розробляється відкрита специфікація контейнерів.

2.6.1. Система Docker

Найпопулярнішим інструментом для контейнеризації є Docker. Контейнеризація як альтернатива віртуалізації, завжди мала потенціал змінити шлях розгортання додатків. Docker як реалізація інструменту контейнеризації, часто порівнюється з віртуальними машинами. Віртуальні машини були створені з метою оптимізації використання обчислювальних ресурсів. На одному сервері можливо запустити декілька віртуальних машин та розгорнути окремі додатки на кожній з віртуальних машин. За цією моделлю, кожна з віртуальних машин надає стабільне програмне середовище для кожного додатку. Але при масштабуванні додатку ми отримуємо значні проблеми з продуктивністю, оскільки віртуальна машина споживає багато системних ресурсів.

Оскільки мікросервіси є відносно маленькими програмами, які необхідно розміщати в окремому програмному середовищі, то створення цілої віртуальної машини не є ефективним підходом. З Docker можливо зменшити витрати на продуктивність та розгортати велику кількість сервісів на єдиному сервері, тому що Docker-контейнер вимагає набагато менше ресурсів.

Основні переваги використання Docker:

- Швидкий час запуску. Контейнер запускається в межах декількох секунд, оскільки він є процесом операційної системи, в той час запуск окремої віртуальної машини займе декілька хвилин.
- Швидше розгортання. Для контейнеру немає потреби кожен раз налаштовувати середовище, а лише необхідно завантажити відповідний образ.
- Просте управління та масштабування контейнерів.
- Краще використання обчислювальних ресурсів.
- Підтримка великої кількості різних операційних систем.

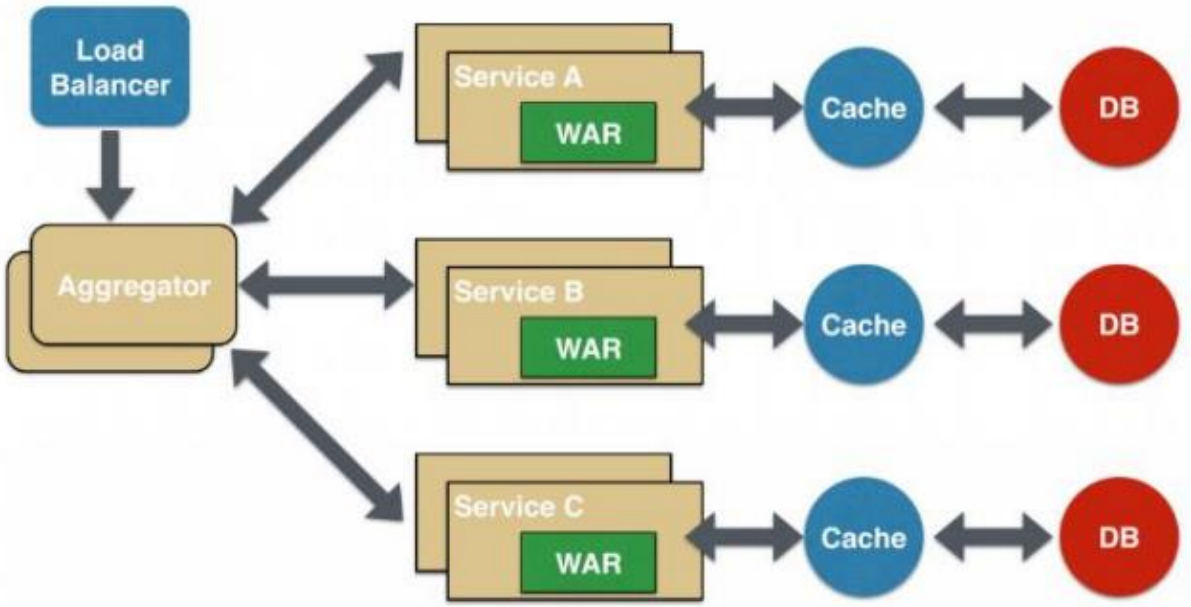
2.7. Шаблони проектування мікросервісів

1. Шаблон «Агрегатор».

Перший та, мабуть, найбільш поширений шаблон проектування при створенні мікросервісів – «агрегатор». У найпростішому випадку, агрегатор є

звичайною веб-сторінкою, яка викликає безліч сервісів для реалізації функціональності, необхідної додатком. Схема патерну наведена на рис. 2.9. Оскільки всі сервіси (Service A, Service B і Service C) надаються за допомогою легкого REST-механізму, веб-сторінка може отримати дані й опрацювати їх як потрібно. Якщо необхідне будь-яке додаткове опрацювання, наприклад, застосувати бізнес-логіки до даних, отриманих від окремих сервісів, то для цього у вас може бути CDI-компонент, що перетворює дані таким чином, щоб їх можна було показати на веб-сторінці. Агрегатор може використовуватися і в тих випадках, коли не потрібно нічого відображати, а потрібен лише більш високорівневий мікросервіс, який може використовувати інші сервіси.

Рис. 2.9. Схема шаблону «Агрегатор»



Цей патерн дотримується принципу DRY. Якщо існує безліч сервісів, які повинні звертатися до сервісів A, B і C, то рекомендується абстрагувати цю логіку в мікросервіс і агрегувати її у вигляді окремого сервісу. Перевага абстрагування на цьому рівні полягає в тому, що окремі сервіси, скажімо, A, B і C, можуть розвиватися незалежно, а бізнес-логіку буде як і раніше виконувати композитний мікросервіс.

2. Шаблон «Посередник».

Патерн «посередник» при роботі з мікросервісами – це окремий варіант агрегатора. В такому випадку агрегація повинна відбуватися на клієнті, але в

залежності від бізнес-вимог при цьому може викликатися додатковий мікросервіс.

На рис. 2.10 наведена схема даного шаблону, як і агрегатор, посередник може незалежно масштабуватися по горизонталі і по вертикалі. Це може знадобитися в ситуації, коли кожен окремий сервіс потрібно не надавати споживачеві, а запускати через інтерфейс.

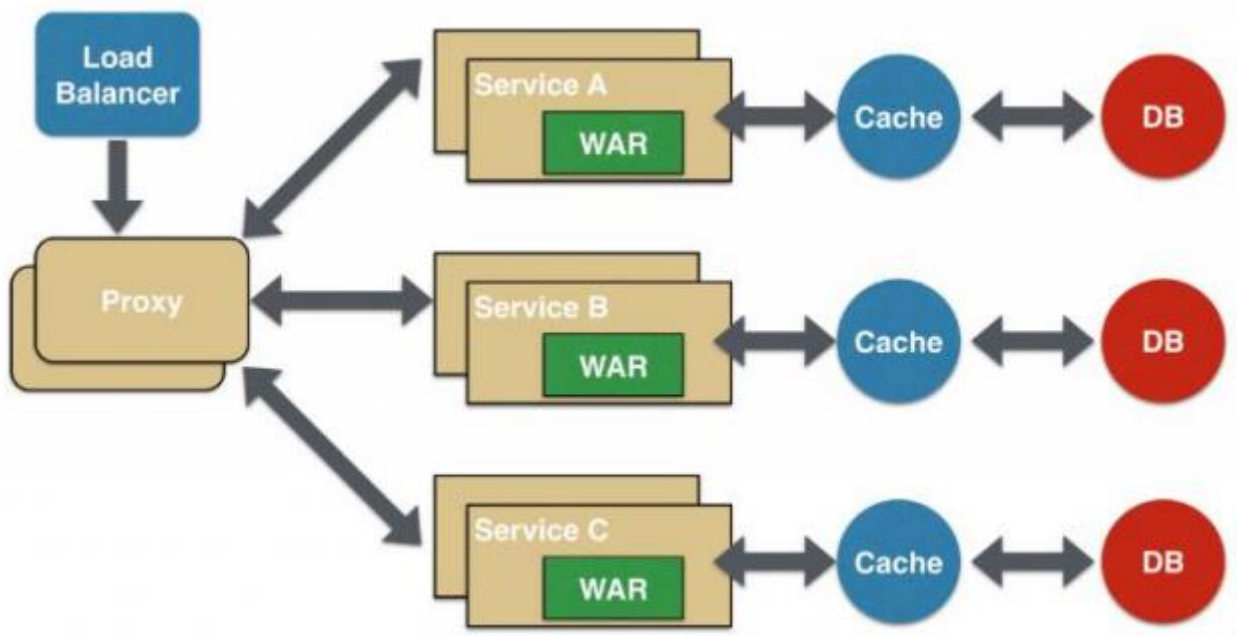


Рис. 2.10. Схема шаблону «Посередник»

Посередник може бути формальним, в такому випадку він просто делегує запит одному з сервісів. Він може бути і розумним, в такому випадку дані перед відправкою клієнту піддаються тим чи іншим перетворенням. Наприклад, рівень представлення для різних пристроїв може бути інкапсульований в розумного посередника.

3. Шаблон «Ланцюг».

Мікросервісний патерн проектування «Ланцюг» видає єдину консолідовану відповідь на запит. В даному випадку сервіс А отримує запит від клієнта, зв'язується з сервісом В, який, в свою чергу, може зв'язатися з сервісом С.

Архітектура побудови мікросервісів за моделлю «Ланцюг» наведена на рис. 2.11. Всі ці сервіси, як правило, обмінюються синхронними повідомлен-нями «запит/відповідь» по протоколу HTTP. Найважливішим моментом є те, що клієнт блокується до тих пір, поки не виконається вся комунікаційна послідовність запитів і відповідей, тобто Service A - Service B і Service B - Service C. Запит від Service B до Service C може виглядати зовсім інакше, ніж від Service A до Service B. Це найбільш важливо у всіх випадках, коли бізнес-цінність декількох сервісів додається.

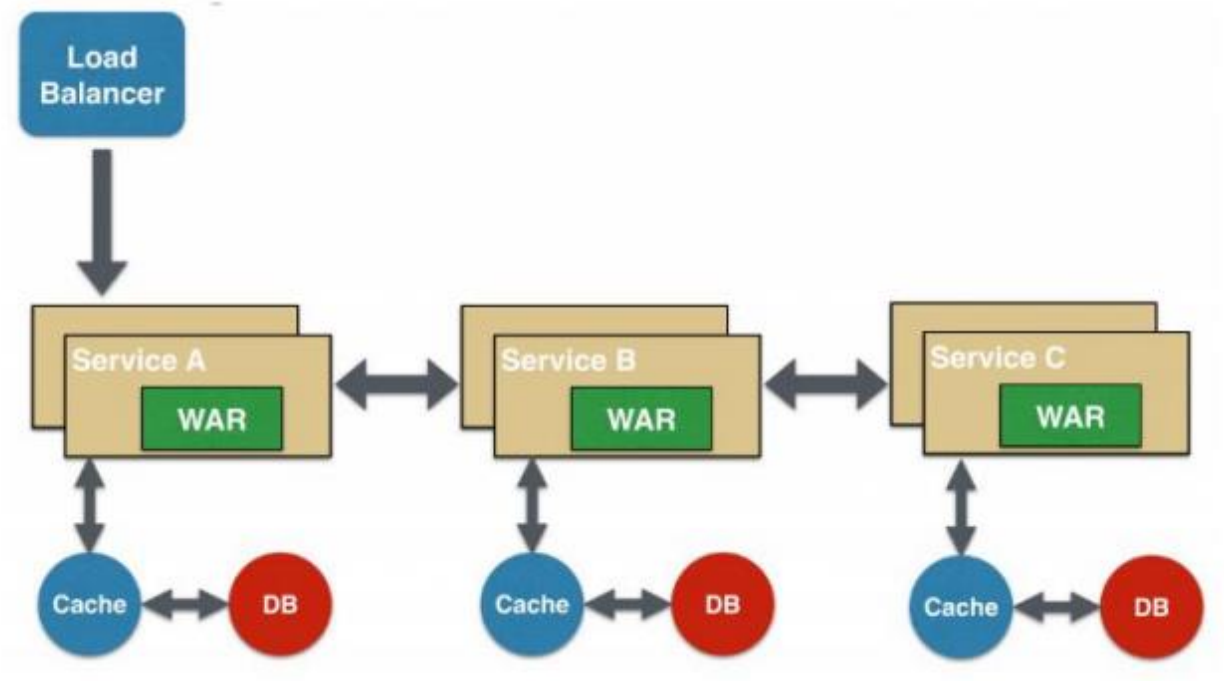


Рис. 2.11. Схема шаблону «Ланцюг»

Також важливо зрозуміти, що не можна робити ланцюг занадто довгим. Це критично, оскільки ланцюг синхронний за своєю природою, і чим він довший, тим довше доведеться чекати клієнтові, особливо якщо відгук полягає у виведенні веб-сторінки на екран. Існують способи обійти такий блокуючий механізм запитів і відгуків, і вони розглядаються в наступному шаблоні.

4. Шаблон «Гілка».

Мікросервісний шаблон проектування «Гілка» розширює шаблон «Агрегатор» і забезпечує одночасну обробку відповідей від двох ланцюгів мікросервісів, які можуть бути взаємовиключними. Цей патерн також може

застосовуватися для виклику різних ланцюгів, або одного і того ж ланцюга - в залежності від потреб. Приклад взаємодії сервісів наведено на рис. 2.12.

В іншому випадку сервіс А може викликати лише один ланцюг в залежності від того, який запит отримає від клієнта. Такий механізм можна конфігурувати, реалізувавши маршрутизацію кінцевих точок JAX-RS, в такому випадку конфігурація повинна бути динамічною.

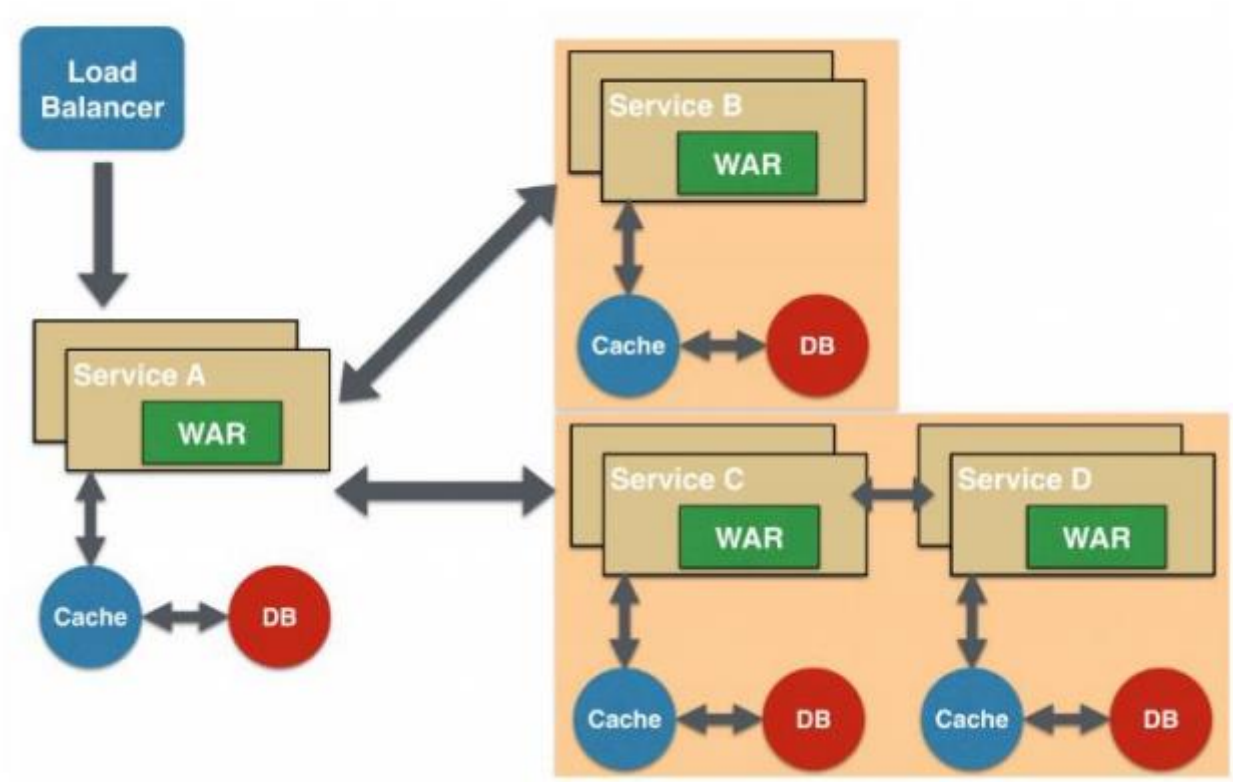


Рис. 2.12. Схема шаблону «Гілка»

5. Шаблон «Дані спільного використання».

Один з принципів проектування мікросервісів — автономність. Це означає, що сервіс повностековий і контролює всі компоненти: інтерфейс призначений для користувача, проміжне ПЗ, транзакції. В такому випадку сервіс може бути багатомовним і вирішувати кожну задачу за допомогою найбільш відповідних інструментів. Наприклад, якщо при необхідності можна застосувати сховище даних NoSQL, то краще зробити саме так, а не додавати всю цю інформацію в базу даних SQL. Однак, типова проблема, особливо при рефакторингу наявного монолітного додатку, пов'язана з нормалізацією бази даних — так, щоб у кожного мікросервісу був строго визначений обсяг інформації. На рис. 2.13 продемонстровано базову схему даного патерну. За цим паттерном кілька

мікросервісів можуть працювати по ланцюгу і спільно використовувати сховища кеша і бази даних. Це доцільно лише в разі, якщо між двома сервісами існує сильний зв'язок. Деякі можуть вбачати в цьому антипаттерн, але в деяких бізнес-ситуаціях такий шаблон дійсно доречний.

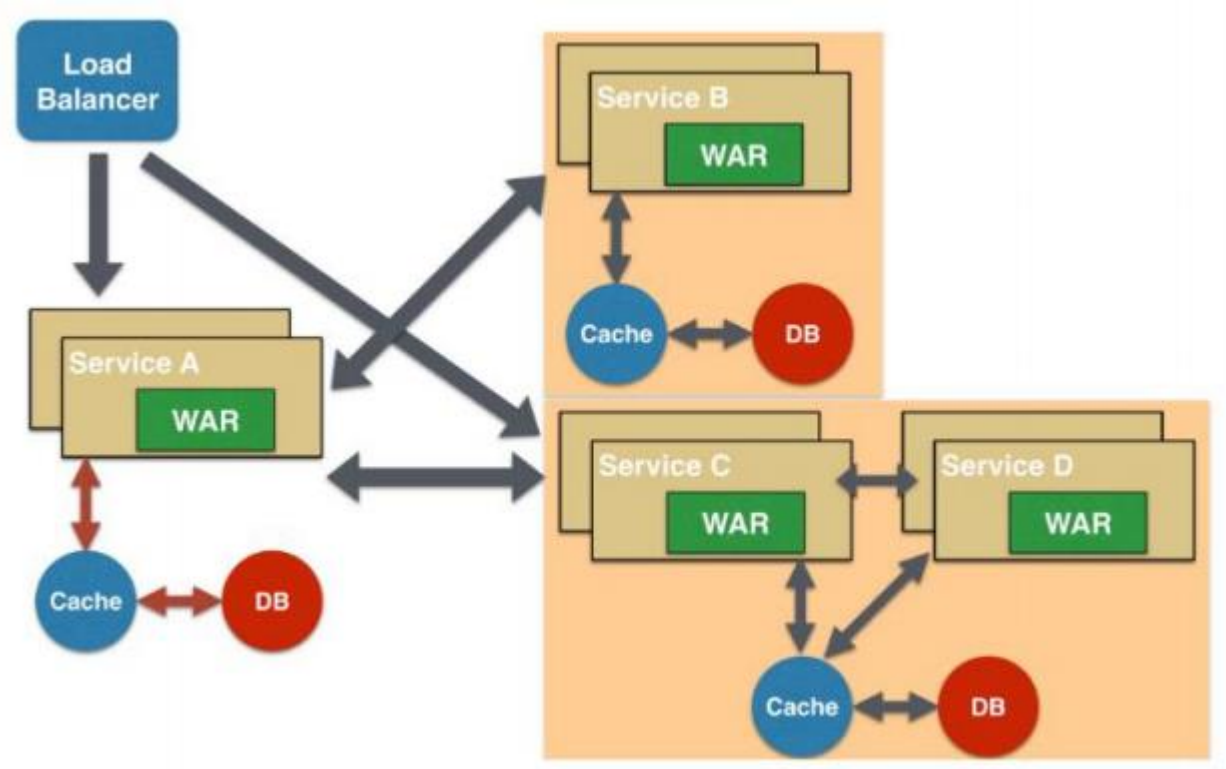


Рисунок 2.13. Схема шаблону «Дані спільного використання»

Крім того, його можна розглядати як проміжний етап, який потрібно подолати, поки мікросервіси не стануть повністю автономними.

6. Шаблон «Асинхронні повідомлення».

При всій поширеності і зрозумілості підходу REST, у нього є важливе обмеження, а саме: він синхронний і, отже, блокуючий. Забезпечити асинхронність можна, але це робиться по-своєму в кожному додатку. Тому в деяких мікросервісних архітектурах можуть використовуватися черги повідомлень, а не модель REST - запит / відповідь.

На рис. 2.14 описано як сервіс А може синхронно викликати сервіс С, який потім буде асинхронно зв'язуватися з сервісами В і D за допомогою черги повідомлень. Комунікація між сервісами А та С може бути асинхронною, скажімо, з використанням веб-сокетів; так досягається бажана масштабованість. Комбінація моделі REST(запит/відповідь) та обміну

повідомленнями (видавник / підписник) також можуть використовуватися для досягнення поставлених цілей.

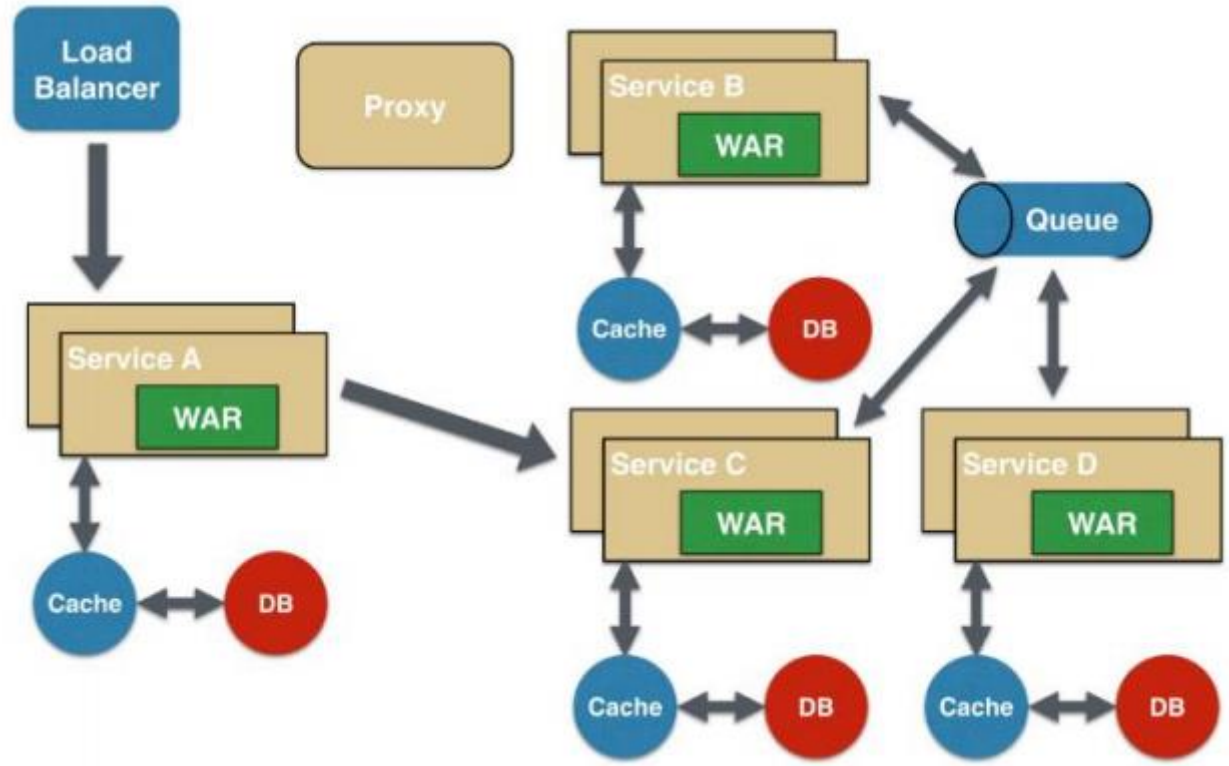


Рис. 2.14. Схема шаблону «Асинхронні повідомлення»

2.8. Типи комунікації мікросервісів

Існує два основних підходи до організації комунікації мікросервісів, які лежать в основі всіх шаблонів наведених вище: синхронний (REST, RPC) та асинхронний обмін повідомленнями.

Розглянемо синхронний підхід на прикладі REST. Передача репрезентативного стану являє собою архітектурний стиль.

У REST-стилю є безліч принципів і обмежень, але ми сконцентруємо увагу на тих з них, які дійсно допоможуть нам при зустрічі з інтеграційними складнощами при побудові мікросервісів та пошуку стилів інтерфейсів для наших сервісів, які виступають в якості альтернативи RPC. Найбільш важливим є поняття ресурсів. Під ресурсами можна розуміти те, про що знає сам сервіс, наприклад сутність «Користувач». У запиті сервер створює різні образи (або представлення) цього об'єкта. Клієнт, наприклад, може запросити JSON-репрезентацію об'єкта «Користувач», навіть якщо він збережений абсолютно в

іншому форматі. Отримавши представлення цього об'єкта, він може робити запити на його зміну і сервер може їх виконувати або не виконувати.

Деякі з властивостей, що надаються протоколом HTTP в якості частини своєї специфікації, спрощують реалізацію REST по HTTP, тоді як при використанні інших протоколів доводиться справлятися з реалізацією подібних властивостей самостійно. У самому протоколі HTTP визначається ряд дуже корисних можливостей, які дуже добре працюють на реалізацію REST-стилю. Наприклад, в HTTP-специфікації методи, такі як GET, POST і PUT, мають цілком зрозумілий сенс, який визначає характер їх роботи з ресурсами. Фактично архітектурний стиль REST підказує нам, що ці методи будуть вести себе так само і по відношенню до всіх ресурсів, і виходить, що HTTP-специфікація вже визначила той набір методів, якими ми можемо скористатися. GET витягує ресурс ідемпотентним способом, а POST створює новий ресурс.

Перейдемо до асинхронного обміну даними на основі подій. Розглянемо два основні моменти: спосіб надання мікросервісами подій і спосіб визначення споживачами моменту настання тієї або іншої події. Традиційно такі брокери повідомлень, як RabbitMQ, намагаються охопити відразу обидві задачі.

Постачальники використовують API для публікації події брокеру. Брокер опрацьовує підписки, дозволяючи споживачам отримати інформацію при настанні тієї чи іншої події. Такі брокери можуть навіть обробляти стан споживачів, наприклад сприяючи відстеженню того, які повідомлення вони бачили раніше.

Будь-яка кількість постачальників може відправляти повідомлення в одну чергу, також будь-яка кількість підписників може отримувати повідомлення з однієї черги. Підписник - програма, яка приймає повідомлення. Зазвичай, підписник знаходиться в стані очікування повідомлень.

Ці системи, як правило, розробляються з можливостями масштабування. Можливо, доведеться поплатитися ускладненням процесу розгортання, оскільки для розробки і тестування сервісів може знадобитися запуск ще однієї системи. Для збереження працездатності цієї інфраструктури можуть також знадобитися додаткові машини і наявність певного досвіду. Але якщо вийде справитися з

усіма труднощами, це може стати дуже ефективним способом реалізації слабо пов'язаних архітектур, керованих подіями.

2.9. Масштабування мікросервісів

Володіти системою здатною автоматично масштабуватись та відповідно реагувати на збільшення навантаження або відмову деяких вузлів є пріоритетною вимогою, але для деяких випадків це виявиться не актуальним і ресурсозатратним.

Коли розглядаються питання про необхідність і способи масштабування системи, що дозволяє краще впоратися з навантаженням або збоями, висуваються наступні вимоги:

- Час відгуку / затримки.
- Доступність.
- Збереження даних.

Важливою частиною створення відмовостійкої системи, особливо коли функціональні можливості розподіляються серед декількох мікросервісів, які можуть перебувати як в робочому, так і в неробочому стані, є забезпечення її спроможності безпечно знижувати рівень функціональності. При роботі з єдиним монолітним додатком нам не доводиться приймати безліч рішень. Працездатність системи залежить від роботи двійкового коду. Але при використанні архітектури мікросервісів потрібно розглядати набагато складніші ситуації. Чим більше один сервіс залежить від залучення інших сервісів, тим більше успішна робота одного сервісу впливає на виконання завдань іншими сервісами. Використання технологій інтеграції, що дозволяють переводити нижчий сервер в режим автономної роботи, може знизити ймовірність впливу простоїв, як планових, так і позапланових збоїв на вищі сервіси.

При проведенні ідемпотентних операцій результат після першого застосування не змінюється, навіть якщо операція послідовно виконується ще кілька разів. Якщо операції є ідемпотентними, ми можемо повторювати виклик кілька разів без негативного впливу. Це нам дуже знадобиться, якщо необхідно повторно відтворити повідомлення, коли немає впевненості, що вони оброблені.

Це є досить поширеним способом відновлення після помилок. В основному масштабування систем виконується з двох причин.

По-перше, для того, щоб легше було впоратися зі збоями: якщо ми переживаємо за відмову будь-якого компонента, то допомогти зможе наявність такого ж додаткового компонента.

По-друге, для підвищення продуктивності, що дозволяє або впоратися з більш високим навантаженням, або знизити час відгуку, або досягти обох результатів.

Розглянемо ряд найбільш поширених технологій масштабування, якими можна буде скористатися, і подумаємо про їх застосування до архітектури мікросервісів.

1. Нарощування потужностей.

Від нарощування потужностей деякі операції можуть тільки виграти. Більш об'ємний корпус з більш швидким центральним процесором і більш ефективної підсистемою введення-виведення часто здатні зменшити затримки і підвищити пропускну здатність, дозволяючи виконувати більший обсяг робіт за менший час.

Але такий різновид масштабування, який часто називають вертикальним масштабуванням, може бути занадто витратним: іноді один великий сервер може коштувати набагато більше, ніж два невеликих сервера нижчої потужності.

2. Розподіл робочих навантажень.

Наявність єдиного мікросервіса на кожному хості, безумовно, краще моделі, яка передбачає наявність на хості відразу декількох мікросервісів. Але спочатку з метою зниження вартості обладнання або спрощення управління хостом багато хто приймає рішення про співіснування кількох мікросервісів на одній фізичній машині. Оскільки мікросервіси запускаються в незалежних процесах, які обмінюються даними по мережі, завдання подальшого їх переміщення на власні хости з метою підвищення пропускної спроможності і масштабування не представляє особливої складності.

3. Балансування навантаження.

Коли сервісу потрібна відмовостійкість, вам знадобляться способи обходу критичних місць збоїв. Для мікросервісу, який надає синхронну кінцеву точку по HTTP, найбільш простим способом вирішення цього завдання (рис. 2.15) буде використання декількох хостів із запущеними на них екземплярами мікросервісу, що знаходяться за балансувальником наванта-ження. Споживачі мікросервісу не знають, чи пов'язані вони з одним його екземпляром або з сотнею таких екземплярів.

Рис. 2.15. Схема розподілу навантаження

4. Системи на основі виконавців.

Застосування балансувальника не є єдиним способом поділу навантаження серед кількох екземплярів сервісу та зменшення їх крихкості. Залежно від характеру операцій настільки ж ефективною може бути і система на основі виконавців. Дана модель також добре працює при пікових навантаженнях, де в міру зростання потреб можуть запускатися додаткові екземпляри для відповідності вхідного навантаження. Поки сама черга робіт буде зберігати стійкість, ця модель може використовувати масштабування для підвищення як пропускної здатності робіт, так і відмовостійкості, оскільки стає простіше впоратися з впливом відмовив (або відсутнього) виконавця. Робота займе більше часу, але нічого при цьому не втратиться.

Також важливим питанням є масштабування баз даних. Масштабування мікросервісів без збереження стану проводиться досить просто. А що робити, якщо ми зберігаємо дані в базі даних? Різні типи баз даних вимагають різних форм масштабування, і розуміння того, яка з цих форм пасуватиме найкращим чином саме для вашого випадку.

Багато сервісів в основному займаються зчитуванням даних. Масштабування для читання дається значно легше масштабування для запису. Тут велику роль може зіграти кешування даних.

В системі керування базами даних (RDBMS), подібної MySQL або Postgres, дані можна буде скопіювати з основного вузла в одну або кілька реплік. Сервіс може направляти всі запити на запис до єдиного основного вузла, але при цьому розподіляти запити на читання між декількома репліками, призначеними для зчитування даних (рис. 2.16).

Створення резервних копій з основної бази даних до реплік відбувається через деякий час після запису. Це означає, що при такій технології зчитування до завершення реплікації дані можуть бути застарілими. Через деякий час для операцій читання стануть доступні вже актуальні дані.

Рис 2.16. Приклад організації взаємодії з БД

2.10. Виявлення сервісів (Service Discovery)

У мікросервісному середовищі, екземпляри регулярно додаються або видаляються при масштабуванні. Оскільки сервери та порти часто призначаються автоматично, клієнти, а також інші мікросервіси повинні бути в змозі знайти та ідентифікувати їх, щоб взаємодіяти. Це завдання вирішується за допомоги концепції виявлення сервісів, в якому ключовим є компонент, який називають реєстром сервісів, що відстежує всі доступні мікросервіси в системі. Кожен сервіс реєструється при запуску, використовуючи клієнт на самому сервісі, який здійснює зв'язок з реєстром, або через сторонній додаток, який контролює екземпляри сервісів в середовищі та зберігає свій статус в реєстрі. Розглянемо детальніше кожен з двох варіантів відкриття сервісів.

1. Виявлення на стороні клієнта.

При використанні даного підходу, клієнт відповідає за визначення місць екземплярів сервісів в мережі та балансування запитів через них. Клієнт запитує реєстр, який є базою даних доступних екземплярів сервісів. Потім клієнт використовує певний алгоритм балансування, щоб обрати один з наявних сервісів та зробити запит до нього. Даний підхід зображено на рис. 2.17.

Розташування в мережі екземпляру сервісу реєструється при його запуску в реєстрі сервісів, а також видаляється з цього реєстру при завершенні роботи відповідного сервісу. Також відбувається періодичне оновлення реєстру сервісів. Патерн виявлення на стороні клієнта має свої переваги та недоліки. Ця модель відносно проста і, за винятком реєстру сервісів, не вимагає інших додаткових частин. Крім того, оскільки клієнт знає про доступні екземпляри сервісів, він може зробити інтелектуальні, специфічні для даного додатка, рішення балансування навантаження, такі як використання послідовного хешування.

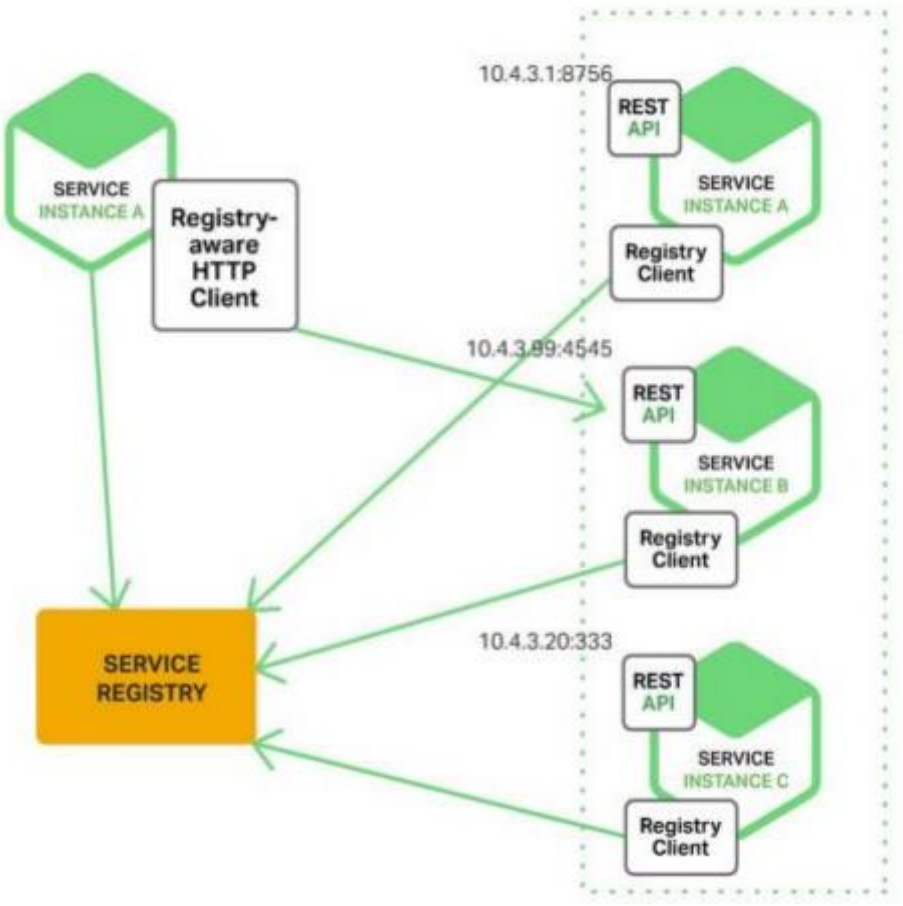


Рис. 2.17. Виявлення на стороні клієнта

Одним із суттєвих недоліків цієї моделі є те, що клієнт пов'язаний з реєстром сервісів. Необхідно реалізувати логіку виявлення сервісів на стороні клієнта для кожної мови програмування і фреймворків, які використовуються в сервісах.

2. Виявлення на стороні сервера.

Клієнт робить запити до сервісу через балансувальник навантаження. Балансувальник навантаження посилає запит до реєстру сервісів та перенаправляє кожен запит до доступного сервісу. Як і в разі виявлення на клієнтській стороні, екземпляри служб реєструються і видаляються у сервісі реєстру сервісів (рис. 2.18). Патерн виявлення на стороні сервера має власні переваги та недоліки. Важливим плюсом є те, що виявлення абстрагується від клієнта. Клієнти нічого не знають додатково, а просто звертаються до балансувальника навантаження.

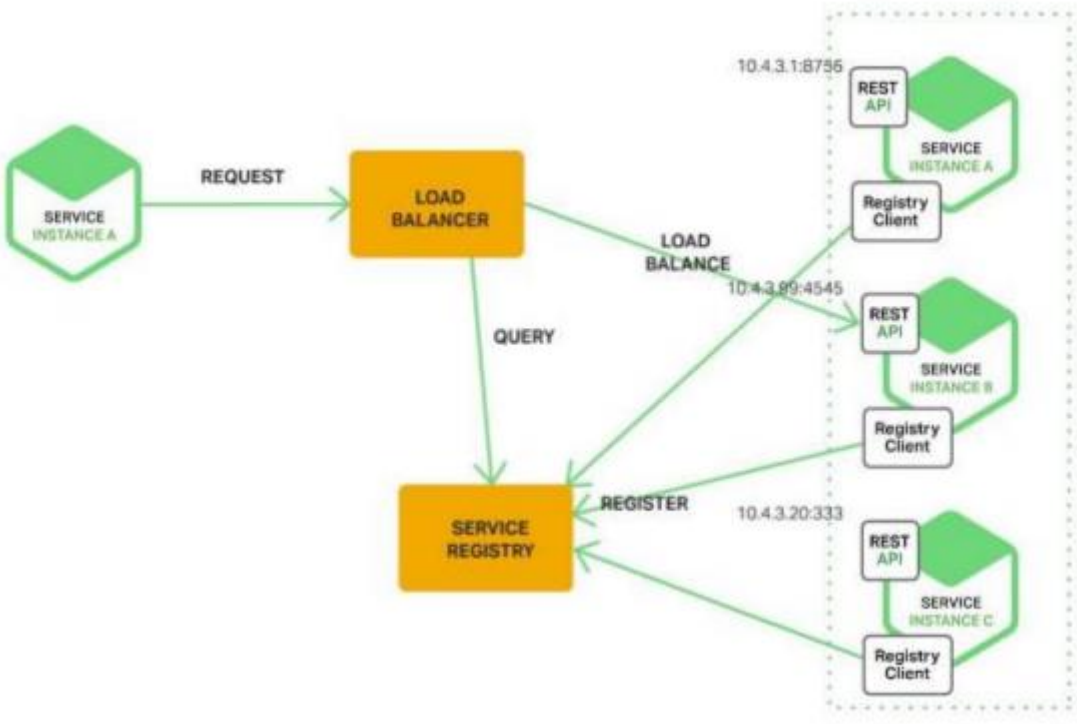


Рис 2.18. Виявлення на стороні сервера

Це усуває необхідність виконання логіки виявлення для кожної мови програмування та використовуваних фреймворків, які використовуються сервісними клієнтами. Крім того, деякі середовища розгортання пропонують дану функціональність безкоштовно.

До недоліків даного шаблону можна віднести те, що необхідно мати та підтримувати ще один додатковий компонент системи – балансувальник навантаження.

Реєстр сервісів є ключовою частиною виявлення сервісу. Він являє собою БД, що містить мережеві розташування екземплярів сервісу. Реєстр сервісів повинен бути постійно доступним і актуальним. Клієнти можуть кешувати мережеві розташування, отримані з реєстру сервісів. Отже, реєстр сервісів складається з кластера серверів, які використовують протокол реплікації для забезпечення узгодженості.

2.11. Автоматичний вимикач (Circuit Breaker)

Мікросервіси часто співпрацюють при опрацюванні запитів. Коли один сервіс викликає синхронно інший, завжди є можливість того, що сервіс буде

недоступним або матиме великий час відгуку. Це може привести до висна-ження ресурсів, які можуть зробити викликаючий сервіс не здатним опрацьовувати інші запити. Відмова одного сервісу може каскадним чином зупинити роботу всієї системи.

Для забезпечення надійності роботи системи використовують шаблон автоматичний вимикач. Необхідно обернути виклик об'єктом автоматичного вимикача, який буде стежити за виключними ситуаціями в системі. Як тільки несправність досягає деякого порогу, то вимикач спрацьовує та всі подальші виклики до вимикача повертаються з помилкою, а перший виклик не виконується взагалі.

Зазвичай, необхідно мати додатковий інструмент, який сповіщає якщо вимикач спрацьовує. Автоматичні вимикачі допомагають зменшити споживання ресурсів, які пов'язані з нестабільними операціями. Ви уникаєте очікування таймаутів, а також вимикач зменшує навантаження на сервер. Даний шаблон можна використовувати не тільки при віддалених викликах, а також в будь-якій програмній системі, щоб забезпечити надійність програми від збоїв в інших частинах. Для мікросервісів даний шаблон особливо цінний, оскільки він надає необхідну стійкість. Якщо він реалізований правильно, то виникнення каскадних збоїв не є можливим.

Висновки

У даному розділі було розглянуто найважливіші аспекти та головні шаблони реалізації мікросервісних додатків, яких рекомендовано дотримуватися для побудови ефективних програм. Також було здійснено огляд основних інструментів для забезпечення правильного функціонування всієї інфраструктури мікросервісів. Використавши всі вище описані концепції, розроблений додаток повинен правильно та ефективно працювати, а також володіти необхідною відмовостійкістю.

РОЗДІЛ 3
ДОСЛІДЖЕННЯ ПЛАТФОРМ ДЛЯ ПОБУДОВИ MSA ТА ПРИКЛАД СТВОРЕННЯ ТЕСТОВОЇ СИСТЕМИ

Для створення мікросервісної архітектури існує багато інструментів для різних мов програмування. Відомо, що мікросервісний підхід дозволяє використання будь-яких технологій, які підтримують певні протоколи та інтерфейси. Для багатьох сучасних мов програмування існують вже готові рішення для створення та розгортання мікросервісів. В даному розділі наведено перелік програмних платформ для найпопулярніших мов програмування, їх особливості при використанні.

3.1. Інструменти для C++

C++ широко використовується для створення продуктивного програмного забезпечення, мова має багату стандартну бібліотеку, яка включає в себе безліч корисних інструментів, які полегшують створення програм. Також C++ поєднує якості високорівневих та низькорівневих мов програмування. Основні переваги, які надає мова для побудови мікросервісів:

- Висока обчислювальна продуктивність. За рахунок того, що C++ має доступ до всіх апаратних ресурсів та компілюється в машинні коди, то він дає перевагу в швидкодії над іншими мовами.
- Підтримка різних методологій програмування, таких як структурне, об’єктно-орієнтоване, узагальнене, функціональне, породжуюче програмуван-ня та інші.
- Наявність великої кількості навчальної літератури
- Широкі можливості даної мови програмування Розглянемо фреймвор-ки для створення мікросервісів на цій мові.

3.1.1. C++ MicroServices

CppMicroServices – це відкрита бібліотека для створення модульних програмних систем, які базуються на ідеях OSGi, які описують модель для побудови додатку із компонентів, які пов’язані друг з другом через сервіси. Дана

бібліотека має набір компонентів для створення модульних, динамічних сервіс-орієнтованих додатків. Даний фреймворк надає наступні можливості:

- Повторне використання компонентів.
- Низька зв’язність між сервісами.
- Розподіл обов’язків відповідно до SOA.
- Чисте API для сервісних інтерфейсів.
- Створення розширюваних та гнучких систем.

По суті, CppMicroServices надає потужний динамічний реєстр сервісів поверх потужного фреймворку, також керує, крім того, логічними модульними одиницями, які називаються пакетами, які розташовуються в загальних або статичних бібліотеках. Кожен пакет в бібліотеці поєднаний з певним контекстом через який доступний реєстр сервісів.

Для створення мікросервісу необхідно реалізувати один чи більше відповідних інтерфейсів, у випадку C++ інтерфейс – це абстрактний клас з одними віртуальними методами, без будь-яких членів класу. Пакет – це набір специфічного ініціалізаційного коду, метаданих, які зберігаються в маніфест файлі та інших ресурсних файлах. Декілька пакетів можуть бути частиною тієї ж самої або різних бібліотек чи виконуваних файлів.

Для створення та використання сервісу, необхідно отримати екземпляр BundleContext, через який кожен пакет має доступ до C++ MicroServices API. Кожен пакет пов’язаний з унікальним контекстом, який доступний з будь-якого місця пакета через GetBundleContext() метод.

Даний фреймворк надає елегантне та чисте API та Discovery Service але багато функцій ще знаходяться в розробці.

3.1.2. Бібліотека UServer / ULib

ULib – це високооптимізована бібліотека для написання C++ додатків. Вона була створена для розробки різноманітних програмних систем. ULib – дуже легка C++ бібліотека, яка полегшує використання шаблонів проектування для систем, які використовують uclibs, а також підтримує потоки через POSIX. Цей фреймворк відключає функції мови, які споживають додаткову пам’ять чи

вводять накладні витрати під час виконання, наприклад RTTI, та опрацювання виняткових ситуацій. Також передбачає, що будуть використані бібліотеки, які базуються на чистому C, а не перевантажені C++ бібліотеки, тому вона має переваги у швидкодії в порівнянні з іншими відомими C++ бібліотеками.

Дана бібліотека підтримує наступні елементи:

- Підтримка протоколів HTTP/1.0 та HTTP/2.
- Підтримка CGI.
- Використання динамічних USP сторінок.

Дана бібліотека надає можливість створення динамічних веб-сторінок подібних до JSP(Java Server Pages) чи ASP(Active Server Pages). ULib рекомендовано використовувати, якщо необхідно мати високу швидкодію та зручний у використані C-сервер.

3.2. Інструменти для Java

Java – одна з найпопулярніших та найпотужніших мов програмування сучасності. Програми на Java транслюються в байт-код, який виконується віртуальною машиною Java(JVM) – програмою, яка опрацьовує байтовий код та передає інструкції обладнанню як інтерпретатор. Перевага подібного способу виконання програм є повна незалежність байт-коду від операційної системи та апаратури, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого існує відповідна віртуальна машина.

Java, без пребільшення, є найпопулярнішою мовою для побудови мікросервісів. Багато відомих компаній створюють свої системи саме на цій мові, прикладом є Netflix, Amazon.

3.2.1. Spring framework

Spring framework – найпотужніша Java-бібліотека, яка дозволяє створювати програмні системи будь-якої складності. Виник як альтернатива J2EE платформи. Для побудови мікросервісів найважливіші модулі Spring це Spring Boot та Spring Cloud.

Spring Boot – складова екосистеми бібліотеки Spring. Spring Boot дозволяє легко створювати повноцінні, ефективні Spring-додатки. Для налаштування програмної системи необхідно відносно мало конфігурацій, в порівнянні з Spring MVC.

Основні особливості Spring Boot:

- Створення повноцінних Spring додатків.
- Вбудований сервер Tomcat або Jetty.
- Автоматична конфігурація Spring framework.
- Забезпечує можливостями моніторингу стану системи.
- Не вимагає написання конфігураційних файлів на XML.

Spring Boot є основою для більш складного фреймворку Spring Cloud, який призначений для створення розподілених систем та має широкий інструментарій для розробки. З основних переваг використання Spring Cloud виділимо наступні:

- наявність реєстру сервісів та системи виявлення сервісів, маршрутизації, міжсервісних викликів;
- балансування навантаження;
- наявність автоматичного вимикача;
- розподілений обмін повідомленнями. Spring Cloud надає декларативний підхід до створення програмного забезпечення.

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

В даному прикладі використовується лише єдина анотація @EnableDiscoveryClient для введення в систему механізму виявлення сервісів. Також Spring Cloud має чудову інтеграцію з Netflix OSS, та надає дані інструменти «з коробки»:

```
@SpringBootApplication
@EnableEurikaServer
public class ServiceRegistrationServer {
    public static void main(String[] args) {
        System.setProperty("spring-config.yml",
"registration-server");

SpringApplication.run(ServiceRegistrationServer.class,
args);
    }
}
```

Вище наведено приклад використання компоненту Netflix OSS через Spring Cloud. Eurika – це сервер, який виконує функцію реєстрації сервісів.

Для включення балансувальника навантаження також необхідно докласти мінімально зусиль, а саме включити анотацію @LoadBalanced над відповідним Spring-компонентом.

```
@Autowired
@LoadBalanced
protected RestTemplate restTemplate;
```

Також Spring Cloud надає горизонтально масштабоване сховище конфігурацій для розподілених систем. Як джерело даних на даний момент підтримується Git, Subversion та прості файли, що зберігаються локально. За замовчуванням Spring Cloud Config віддає файли, які відповідають імені викликаючого Spring додатку.

Spring Cloud надає зручні анотації та автоконфігурації для забезпечення аутентифікації, створення токенів OAuth2 для доступу до ресурсів бекенду. Spring Cloud спрощує підключення до сервісів та отримання можливостей середовища в хмарних платформах, таких як Cloud Foundry і Heroku. Особлива підтримка Spring-додатків через Java і XML-конфігурації робить підключення до хмарних сервісів тривіальним завданням. Ви можете використовувати існуючі хмарні коннектори або написати власний для вашої хмарної платформи. "З коробки" підтримуються найбільш популярні сервіси (реляційні СУБД, MongoDB, Redis, Rabbit), але також можливе розширення для ваших сервісів. Жоден з сервісів не вимагає зміни самого Spring Cloud, досить просто додати необхідну вам jar бібліотеку в область видимості classpath.

3.2.2. Spark framework

Spark framework – це проста та легка Java веб-бібліотека для швидкої розробки програмних систем. Ціль даного фреймворку це забезпечити інструменти для побудови ефективних та елегантних веб-додатків на Java. Основною особливістю є те, що Spark побудований за філософією лямбда-виразів, які були введені в Java 8.

Даний фреймворк є гарним вибором для побудови мікросервісів, оскільки він вимагає написання невеликих об'ємів коду. Spark загалом використовується для створення REST API, але одночасно підтримує велику кількість різноманітних шаблонів. Spark є ідеальним рішенням для невеликих та ефективних веб-додатків.

3.2.3. Бібліотека Restlet

Restlet – це бібліотека для створення потужних RESTful веб-додатків, яка має великий набір інструментів та можливостей. Вона доступна для більшості платформ основаних на JVM (Java SE/EE, Google App Engine, GWT, Android).

Restlet пропонує платформу для побудови REST API за концепцією «Перш за все API» («API First»). Дана концепція передбачає те, що API має найвищий пріоритет та для якої користувач API є головним користувачем системи.

RestComponent володіє всім необхідним для конфігурації серверу та налаштування додаткових функцій, таких як безпека та інше. В даному фреймворку надається високе значення поняттю ресурса, як і для REST стилю в цілому. Приклад простого Restlet-клієнта під платформу Android:

```
    ClientResource resource = new
ClientResource("http://localhost/test")
    resource.setRequestEntityBuffering(true);
    UserResource testResource =
resource.wrap(UserResource.class);
    User user = testResource.retrieve();
```

Даний приклад демонструє отримання ресурсу від віддаленого сервера, який в даному прикладі виконує localhost.

Restlet на даний момент не є настільки популярним як Spring framework чи Spark, але пропонує елегантний підхід до створення REST API та підтримує велику кількість існуючих платформ.

3.3. Інструменти для Python

Python – це мова програмування, яка орієнтована на збільшення продуктивності розробника та читабельність коду. Синтаксис ядра Python мінімалістичний, а стандартна бібліотека включає великий об'єм корисних функцій. Основні напрямки використання Python:

- Розробка веб-додатків(фреймворк Django).
- Аналіз даних та машинне навчання.
- Швидке прототипування ідей в бізнесі за рахунок існування готових. бібліотек, низького порогу входження та високої продуктивності Python-програмістів.
- Написання скриптів для автоматизації Python значно поступається в швидкодії мовам зі статичною типізацією, таким як C/C++ та Java, але його перевага полягає у ефективності та швидкості написання коду. Розглянемо головні фреймворки для створення мікросервісів.

3.3.1. Nameko

Nameko – це бібліотека для створення мікросервісів для Python, яка дозволяє розробникам сервісу сконцентруватися на логіці додатку та закликає до спроможності до тестування. Основні можливості, які підтримує даний фреймворк:

- Підтримка AMQP, RPC та подій.
- Підтримка веб-сокетів.
- Підтримка командного рядку для простої та швидкої розробки.
- Утиліти для інтеграційних та юніт тестів Сервіс написаний на Nameko виглядає як:

```
from nameko.rpc import rpc
class DummyService:
```

```
name = “dummy_service”
@rpc
def hello(self, name):
return “Hello, {}!”.format(name)
```

Щоб запустити сервіс необхідно викликати наступну команду:

```
$ nameko run app * starting services: dummy_service
```

Nameko здатний до масштабування від одного сервісу до кластеру, який складається з багатьох екземплярів різних сервісів. Бібліотека також надає інструменти для клієнтів, які ви можете використовувати для написання Python програм для взаємодії з існуючим кластером Nameko.

3.4. Розробка та оптимізація тестової мікросервісної системи шляхом декомпозиції монолітного додатку

В даному розділі буде розглянуто приклад побудови програмної системи за мікросервісною архітектурою на мовах Java та Python. Для інтеграції сервісів використаємо інструменти компанії Netflix, які знаходяться у вільному доступі, а саме: Zuul, Hystrix, Eureka.

Тестова система надає можливість здійснити реєстрацію користувачу, а також отримати перелік карт/рахунків, створити депозит, виконати переказ коштів.

Дана система вже реалізована в вигляді монолітного додатку, потрібно декомпозувати його та створиті додаток на основі MSA.

Всю систему можна умовно розподілити на 2 основні частини: набір допоміжних сервісів, таких як точка для єдиного входу, автоматичний вимикач, а також незалежні самостійні мікросервіси. Всього в системі чотири мікросервіси: Pay Service, User Service, Card Service, Deposit Service. Система побудована за шаблоном агрегатор. Pay Service надає системі можливість зробити грошовий переказ або виконати оплату в інтернеті, гроші списуються з карти та оновлюються дані в сервісі Card. User Service оперує даними про користувачів та тісно співпрацює із іншими сервісами в системі. Наприклад, при відображенні списку платіжних кард для користувача ми звертаємось до Card

Service, а для п для перегляда депозитів користувача, ми йдемо до Deposit Service. Deposit Service забезпечує систему створення депозитів та зняття коштів на карту. Card Service містить список наявних карт користувача та їх характеристики. Вся взаємодія між сервісами є синхронною та відбувається шляхом звертання до певної точки REST API кожного сервісу. Для розгортання кожного сервісу використовується окремий Docker-контейнер із налаштованим середовищем. Для початку потрібно розбити моноліт на частини.

3.4.1. Розбиття моноліту на частини

Майже всі успішні приклади використання мікросервісної архітектури, розпочиналися з моноліту, який з плином часу розростався. І доволі частими є випадки, коли проект розпочинався з мікросервісів і не досягав своєї мети.

Тому можна зробити висновок, що краще розпочинати новий проект з моноліту, навіть якщо ви знаєте, що ваш проект буде достатньо великим для використання мікросервісів. MSA є корисною та ефективною архітектурою, але всі її переваги доцільні лише для великих та складних систем. Для простих систем набагато краще підходить суцільна монолітна архітектура.

Перша причина дотримуватися принципу «спочатку - моноліт» — класичний принцип «Вам це не знадобиться». Коли ви починаєте розробляти новий додаток, необхідно впевнитись, що він буде корисним для користувачів. Кращий спосіб перевірити, чи буде додаток користуватися попитом — це створення його спрощеної версії. Спочатку на першому місці стоїть швидкість розробки (а значить, і швидкість отримання зворотного зв'язку від потенційних користувачів), а розробка мікросервісів займає набагато більше часу.

Наступна проблема полягає в тому, що мікросервіси працюють добре, якщо ви досягли чітких, стабільних меж між окремими сервісами — для цього потрібно отримати правильний набір обмежених контекстів. Будь-який рефакторинг функціональності між сервісами складніший аналогічного в моноліті.

Побудувавши монолітний додаток, ви зможете визначити вірні межі, перш ніж використовувати мікросервіси. Це також дасть вам час підготувати все

необхідне для створення сервісів з більш чіткими обмеженими контекстами. Існують різні шляхи реалізації стратегії «спочатку - моноліт».

- Логічний шлях – це проектувати моноліт з усією обережністю, звертаючи увагу на модульність в програмному забезпеченні, межі API та спосіб зберігання даних. Якщо зробити це добре, то перехід до мікросервісів буде відносно простим.
- Більш загальний підхід – почати з моноліту і поступово відокремлювати від нього мікросервіси. У цьому випадку значна частина початкового моноліту може залишитися в якості центральної в мікросервісній архітектурі, але основна частина нової розробки буде відбуватися в сервісах, залишаючи моноліт без великих змін.
- Також поширений підхід з повною заміною моноліту. Даний підхід не є дуже ефективним, але він має право на існування.
- Ще один варіант розбиття моноліту – почати з декількох сервісів, які більші за тих, що очікуються в кінці. Використовуйте ці великі сервіси, щоб навчитися працювати в мультисервісному середовищі. Хоча багато переваг надає підхід «спочатку - моноліт», далеко не всі розділяють такий метод. Контраргумент полягає в тому, що починаючи з мікросервісів, ви звикаєте до ритму розробки в такому оточенні. Потрібно дуже багато зусиль, щоб побудувати монолітний додаток в модульному вигляді, достатньому для простого розбиття на мікросервіси. Починаючи з мікросервісів, ви працюєте в невеликих командах, розділених межами сервісів, це дозволить вам прискорити розробку за рахунок додаткових кадрових ресурсів, як тільки це буде потрібно. Такий підхід особливо добре працює в разі заміни системи, коли у вас є більше шансів отримати досить стабільні кордони на ранньому етапі.

3.4.2. Єдина точка входу (API Gateway)

Мікросервісна система може складатися з десятків або, можливо, навіть сотень сервісів, тому клієнту досить тяжко взаємодіяти з всіма мікросервісами індивідуально. Більше того, навіть неможливо знайти всі мікросервіси без додаткових інструментів. У такій ситуації було прийнято використовувати

шаблон API Gateway. Даний патерн виступає в якості єдиної точки входу для набору пов’язаних мікросервісів або навіть всієї системи.

Приклад єдиного шлюзу для входу наведено на рис. 3.1. Цей шлюз визначає та перенаправляє кожен клієнтський запит на відповідний сервіс. Як вхідна точка до системи, API Gateway також відповідає за підтримку безпеки, перевірки OAuth-токенів та перевірки прав доступу клієнтів до конкретних сервісів.

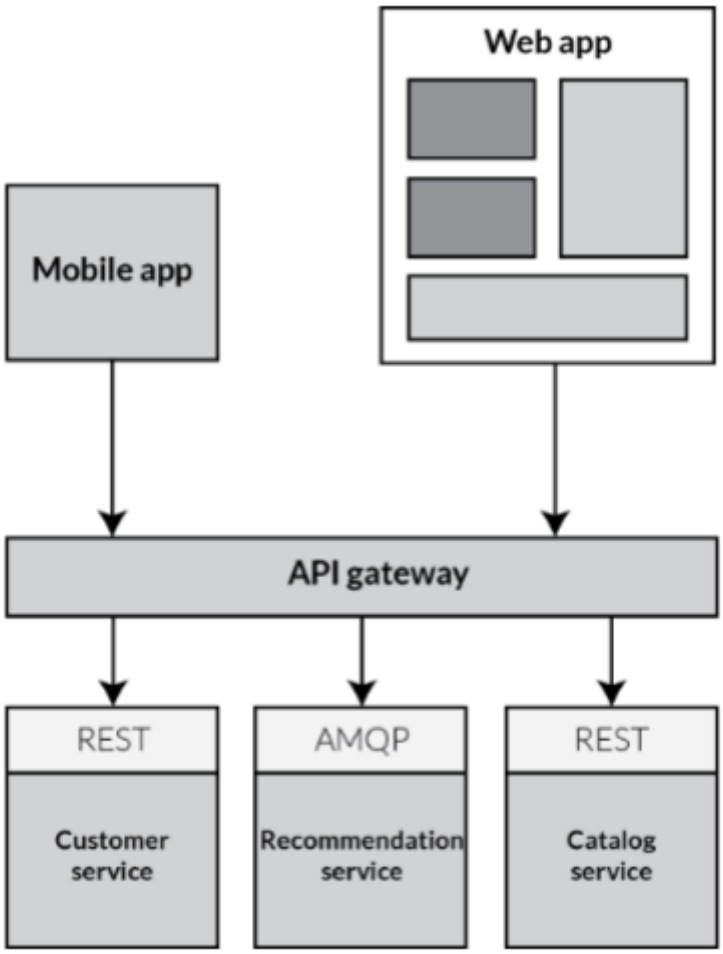


Рис. 3.1. Взаємодія зовнішніх клієнтів з мікросервісами

Переваги використання спільної точки входу:

- Єдиний адрес набагато зручніший сотні індивідуальних адресів API.
- Зручно реалізовувати обмеження на кількість запитів в єдиному місці
- Вся система стає гнучкішою – можна змінювати внутрішню структуру
- Можна кешувати відповіді.

- Можна поєднувати відповіді від різних сервісів. Єдиний шлюз є деяким аналогом стандартного ООП-шаблону фасад, який надає єдиний зручний інтерфейс для роботи зі складною системою.

Даний шаблон має деякі недоліки. З'являється ще одна додатковий сервіс, який має бути створений та розгорнутий. Існує також ризик того, що єдина точка входу стане вузьким місцем для подальшого розвитку. Розробники повинні постійно оновлювати шлюз, щоб додавати кінцеві точки кожного мікросервісу.

3.4.3. Конфігорування Netflix Zuul

Netflix Zuul – це реалізація API Gateway від Netflix. Для налаштування Zuul серверу було використано можливості фреймворку Spring Cloud, який дозволяє створити даний сервер використавши лише одну анотацію над стандартним java класом.

```
@SpringBootApplication
@Controller
@EnableZuulProxy
public class ZuulServerApplication {
    public static void main(String[] args) {
        new
SpringApplicationBuilder(ZuulServerApplication.class).web(t
rue).run(args);
    }
}
```

Zuul працює як інтерфейс, який надає єдиний вхід до всієї системи та делегує виклики до підсистем. Кожен з мікросервісів мають власні REST APIs, які використовуються для маршрутизації даним сервером.

Файл конфігурації серверу application.yml має наступний вигляд:

```
zuul:
    ignoredServices: "*"
    routes:
        cards: /cards/**
        users: /users/**
```

Даний елемент конфігураційного файлу описує як саме необхідно перенаправляти запити, які надходять на Zuul.

3.4.4. Конфігорування Netflix Eureka

Service Discovery є одним з ключових принципів мікросервісної архітектури. Задача виявлення сервісів була вирішена використанням Netflix Eureka. Eureka – це сервер для реєстрації всіх сервісів, які знаходяться в системі. Для запуску серверу треба створити простий Spring Boot додаток з анотацією @EnableEurekaServer:

```
@SpringBootApplication
@EnableEurikaServer
public class EurekaServer {
 public static void main(String[] args) {
 SpringApplication.run(EurekaServer.class, args); }}
```

При правильному налаштуванні серверу, відкриється вікно браузера з панеллю управління. Дана панель демонструє всі сервіси, які зареєстровані в системі. Окрім того, можна отримати доступ до одного із сервісів, наприклад, до сервісу користувачів, виконавши команду:

```
$ open $(echo \”(echo $DOCKER_HOST)/user\”)
\sed ‘s/tcp:\/\//http:\/\//g’|
\sed ‘s/[0-9]\{4,\}/10000/g’|
\sed ‘s/\”//g’)
```

Вище зазначена команда перейде до кінцевої точки Gateway API і проксі REST API, який надається сервісом користувачів. Ці REST API були налаштовані на використання HATEOAS, який підтримує автоматичне виявлення всіх функціональних можливостей сервісу. Отримаємо наступний результат в JSON нотації:

```
{ “_link”: {
 “_self”: {
```

“href”: ”http://127.0.0.1:1000/user”
},
“resume”:{
“href”: ”http://127.0.0.1:1000/user/resume”
},
“pause”: {
“href”: ”http://127.0.0.1:1000/user/pause”
},
“restart”: {
“href”: “http://127.0.0.1:1000/user/restart”
},
“metrics”: {
“href”: “http://127.0.0.1:1000/user/metrics”
},
“env”: [{
“href”: “http://127.0.0.1:1000/user/env”
}],
“beans”: {
“href”: “http://127.0.0.1:1000/user/beans”
},
“info”: {
“href”: “http://127.0.0.1:1000/user/info”
},
“routes”: {
“href”: “http://127.0.0.1:1000/user/routes”} }}

3.4.5. Конфігорування Netflix Hystrix

Також в цьому додатку було використано автоматичний вимикач для кожного сервісу, який реалізовано в Netflix Hystrix Dashboard. Якщо мікросервіс

не відповідає через деякі внутрішні помилки, Hystrix може перенаправити запит на спеціальний внутрішній метод. У випадку, якщо сервіс працює з постійними збоями, Hystrix буде вимикати сервіс та викликати спеціальний метод, який опрацьовує дану ситуацію та не буде викликати даний сервіс поки він не стане знову працездатним. Для того щоб визначити чи доступний сервіс знову, Hystrix дозволяє надсилати декілька запитів, щоб дізнатися про стан мікросервісу. Для створення спеціального методу, який буде опрацьовувати виключні ситуації просто додаємо відповідну анотацію:

```
@HystrixCommand(fallbackMethod = “defaultUsers”)
public Object doPay(final Map<String, Object>
parameters) {
 // do stuff that might fail }
```

Метод doPay() буде викликано лише при виникненні виключних ситуацій, прикладом яких можуть бути збої в мережі або самого серверу.

Після правильного запуску Hystrix переходимо на відповідний сервер та можемо спостерігати наступну панель. Дана панель надає опис станів системи та кожного автоматичного вимикача.

3.4.6. Розробка Card Service

Даний мікросервіс є важливою частиною всієї системи та зберігає інформацію про платіжні карти користувача. Він був створений на платформі Spring framework та розгортається в окремому процесі, використовуючи Docker. Для зчитування та оновляння даних використана база даних Oracle Database.

Сервіс являє собою простий CRUD-додаток, для створення якого було використано інструменти Spring Data, а саме шаблон репозиторій з поєднанням анотації @RepositoryRestResource, яка створює відповідний контролер для взаємодії з певним репозиторієм:

```
@RepositoryRestResource(collectionResourceRel = “genres”,
path = “genres”)
public interface GenreRepository extends
```

PagingAndSortingRepository<Genre, Long> {

List<Card> getCardsByAccount(@Param(“0”) String account);

}

Даний клас дозволяє робити виклики до даного репозиторію не створюючи явно необхідні контролери, приклади HTTP-викликів та їх результати:

$ curl –i –X POST –H “Content-Type:application/json” –d

“{\”account\”:\”38688856\”}” http://localhost/genres

Відповідь:

HTTP/1.1 200 Ok

Content-Length: 0

Date: 15 Jan 2017 00:03:46 GMT

Параметри команди curl :

- -i – гарантує, що ви побачите відповідь із заголовками;
- -X POST – виклик HTTP-POST методу;
- -d – дані, які будуть відіслані серверу/

3.4.7. Розробка User Service

Даний сервіс було створено за допомоги мови програмування Python та фреймворку Flask. Вказаний мікросервіс оперує даними про користувачів та взаємодіє із іншими сервісами в системі через REST по протоколу HTTP. Спілкуючись з Card сервісом, система надає користувачеві список платіжних карт. Також цей сервіс повертає інформацію про користувачів існуючих в системі. В якості сховища даних мікросервіс використовує стандартну реляційну базу даних.

Для реалізації контролерів використаємо можливості фреймворку Flask, приклад реалізації контролера:

@app.route(“/users/<username>”, methods=[‘GET’])

def user_record(username):

if username not in users:

```
 raise NotFound
 return get_user(username)
```

Анотацією @app.route ми вказуємо шлях та HTTP-метод, що викликає вказаний метод. Викликавши curl отримаємо наступний результат:

```
$ curl http://localhost:8888/users/roman_poshtak
{
 “id”: “roman_poshtak”,
 “last_active”: “1360031625”,
 “name”: “John Smith”
“account”: “10239991”
}
```

Слід зауважити, що для python мікросервісу небхідно використовувати додаткові інструменти для інтеграції з Hystrix та Eureka.

Приклад створення автоматичного вимикача для python додатку.

```
from hystrix import Command
class FallbackCommand(Command):
 def run(self):
 return default()
```

Також необхідно використовувати додатковий клієнт для Netflix Eureka:

```
from eureka.client import EurekaClient
ec = EurekaClient(“UserService”,
 eureka_domain_name=”test.domain.net”,
 port=80, secure_port=443)
ec.register()
ec.update_status(“UP”)
```

3.4.8. Розробка Deposit Service

Даний мікросервіс відповідає за створення та конфігорування депозитів. Реалізований на мові програмування Java та фреймворку Spring boot аналогічно мікросервісу Card. Щоб отримати всі відкриті депозити користувача в системі,

треба звернутися до екземпляру сервісу, який знаходиться, наприклад, за адресою http://127.0.0.1:8889, на /deposits, і сервіс поверне список замовлень.

GET /deposits

{

“20170521”: [

“7daf7208-be4d-4944-a3ae-c1c2f516f3e6”,

“267eedb8-0f5d-42d5-8f43-72426b9fb3e6”

],

“20170522”: [

“a8034f44-aee4-44cf-b32c-74cf452aaaae”,

74

“276c79ec-a26a-40a6-b3d3-fb242a5947b6”

]

3.4.9. Розробка Pay Service

Реалізований на мові програмування Java та фреймворку Spring boot, робить запит до сервісу Card для зміни балансу на карті та робить запрос в процесинговий центр.

@RestResource(collectionResourceRel = “pay”,

path = “pay”)

public interface Pay {

List<Responce> doPay(@Param(“0”) Param param);

Висновки

В даному розділі було проаналізовано найвідоміші програмні платформи для деяких мов прорамування. З переглянутих мов найбільше виділяється Java та її фреймворк Spring Cloud, який надає багатий інструментарій у вигляді інфраструктурних сервісів створений компанією Netflix.

Python є гарним рішенням для швидкої розробки та підтвердження концепції майбутньої системи. Найкращим вибором фреймворку під дану мову є бібліотека Flask. Вибір

C++ у якості інструменту для створення мікросервісів має сенс, якщо швидкодія системи має вирішальне значення, в іншому разі краще використати мову зі зручнішими бібліотеками та швидкістю розробки.

Також було розглянуто створення тестового додатку, який складається з декількох сервісів, які написані на різних мовах програмування та на різних фреймворках. Також було використано такі шаблони мікросервісної архітектури як Gateway API, Service Discovery та Circuit Breaker за допомоги відповідних інструментів. Дана система піддається горизонтальному масштабуванню шляхом збільшення вузлів в системі та володіє необхідною відмовостійкістю.

РОЗДІЛ 4
ЕКОНОМІЧНИЙ РОЗДІЛ

4.1. Визначення трудомісткості розробки програмного забезпечення

1) передбачувана кількість операторів – 3000;
2) коефіцієнт складності програми – 1,9;
3) коефіцієнт редагування програми у ході її розробки - 0,07;
4) часова заробітна платня програміста, грн/год - 70,0;
5) вартість машиночасу, грн/год - 20,0.

Нормування праці в процесі створення ПЗ істотно ускладнено в силу творчого характеру праці програміста. Тому трудомісткість розробки ПЗ може бути розрахована на основі системи моделей з різною точністю оцінки.

Трудомісткість розробки ПЗ можна розрахувати за формулою:

$$t = t_o + t_u + t_a + t_n + t_{отл} + t_д, \text{ людино-годин},$$

де t_o - витрати праці на підготовку й опис поставленої задачі (приймається 50);

$t_и$ - витрати праці на дослідження алгоритму рішення задачі;

t_a - витрати праці на розробку блок-схеми алгоритму;

$t_п$ - витрати праці на програмування по готовій блок-схемі;

$t_{отл}$ - витрати праці на налагодження програми на ЕОМ;

$t_д$ - витрати праці на підготовку документації.

Складові витрати праці визначаються через умовне число операторів у ПЗ, яке розробляється. Умовне число операторів (підпрограм):

$$Q = q \cdot C \cdot (1+ p),$$

де q - передбачуване число операторів;

C - коефіцієнт складності програми;

p - коефіцієнт кореляції програми в ході її розробки.

Звідки:

$$Q = 3000 \cdot 1{,}9\ (1 + 0{,}07) = 6099$$

Витрати праці на вивчення опису задачі ti визначається з урахуванням уточнення опису і кваліфікації програміста:

$$t_u = \frac{Q \cdot B}{(75..85) \cdot k}, \text{ людино-годин}$$

де B - коефіцієнт збільшення витрат праці унаслідок недостатнього опису завдання. Коефіцієнт збільшення витрат праці в залежності від складності завдання приймається від 1.2 до 1.5, приймемо B = 1.3.

k - коефіцієнт кваліфікації програміста, який визначається від стажу роботи за даною спеціальністю. Коефіцієнт становить:

для працюючих до двох років – 0.8;

від двох до трьох років – 1.0;

від трьох до п'яти років – 1.1 – 1.2;

від п'яти до семи – 1.3 – 1.4;

понад сім років – 1.5 – 1.6.

Тому приймемо k = 0.8.

Звідки:

$$t_u = \frac{6099 \cdot 1{,}3}{77 \cdot 0.8} = 128.7$$

Витрати праці на розробку алгоритму рішення задачі:

$$t_a = \frac{Q}{(20..25) \cdot k}, \text{ люд-годин}$$

Звідки:

$$t_a = \frac{6099}{22 \cdot 0.8} = 346.5$$

Витрати на складання програми по готовій блок-схемі:

$$t_n = \frac{Q}{(20..25) \cdot k}, \text{ людино-годин}$$

Звідки:

$$t_n = \frac{6099}{22 \cdot 0{,}8} = 346{,}5$$

Витрати праці на налагодження програми на ЕОМ:

- за умови автономного налагодження одного завдання:

$$t_{отл} = \frac{Q}{(20..25) \cdot k}, \text{ людино-годин}$$

- за умови комплексного налагодження завдання:

$$t^k_{отл} = 1.5 \cdot t_{отл}, \text{ людино-годин}$$

Звідки:

$$t_{отл} = \frac{6099}{22 \cdot 0.8} = 346{,}5$$

$$t^k_{отл} = 1.5 \cdot 346{,}5 = 519.75$$

Витрати праці на підготовку документації:

$$t_д = t_{др} + t_{до}, \text{ людино-годин,}$$

де $t_{др}$ - трудомісткість підготовки матеріалів і рукопису.

$$t_{др} = \frac{Q}{(15..20) \cdot k}, \text{ людино-годин}$$

$t_{до}$ - трудомісткість редагування, печатки й оформлення документації

$$t_{до} = 0.75 \cdot t_{др}, \text{ людино-годин,}$$

Звідки:

$$t_д = \frac{6099}{17 \cdot 0,8} + \frac{0,75 \cdot 6099}{17 \cdot 0,8} = 448.5 + 336,3 = 784,8$$

Звідки отримуємо, що трудоміткість розроблення програмного забезпечення складає:

t = 50 + 128.7+ 346,5 + 346,5 + 346,5 + 784,8 = 2003 людино-години.

4.2. Витрати на створення програмного забезпечення

Витрати на створення ПЗ $К_{по}$ включають витрати на заробітну плату виконавця програми $З_{зп}$ і витрат машинного часу, необхідного на налагодження програми на ЕОМ

$$К_{ПО} = З_{ЗП} + З_{МВ}, \text{ грн.}$$

Заробітна плата виконавців визначається за формулою:

$$З_{ЗП} = С_{ПР} \cdot t = 70 \cdot 2003 = 140210 \text{ грн},$$

де: t - загальна трудомісткість, людино-годин;

$С_{пр}$ - середня годинна заробітна плата програміста, грн/година.

Вартість машинного часу, необхідного для налагодження програми на ЕОМ:

$$З_{мв} = t_{отл} \cdot С_{мч} = 346,5 \cdot 20 = 6930 \text{ грн},$$

де $t_{отл}$ - трудомісткість налагодження програми на ЕОМ, год.

$С_{мч}$ - вартість машино-години ЕОМ, грн/год.

Звідки отримуємо, що витрати на створення програмного забезпечення складає:

$$К_{ПО} = 6930 + 140210 = 147140 \text{ грн}.$$

Визначені в такий спосіб витрати на створення програмного забезпечення є частиною одноразових капітальних витрат на створення АСУП.

Очікуваний період створення ПЗ:

$$T = \frac{t}{B_k \cdot F_p}\ , \text{місяців},$$

де B_k - число виконавців;

F_p - місячний фонд робочого часу (при 40 годинному робочому тижні F_p=176 годин).

Звідки:

$$T = \frac{2003}{1 \cdot 176} = 11{,}3 \approx 11 \text{ місяців}.$$

4.3. Маркетингові дослідження ринку збуту розробленого програмного продукту

За допомогою маркетингового дослідження було виявлено, що розробленій методиці проектування на сьогодні немає аналогів. Подібні методики націлені на підвищення ефективності розробки програмного забезпечення на основі мікросервісних систем.

Однією з головних переваг мікросервісної архітектури є її гетерогенність, тому було досліджено базові бібліотеки для створення незалежних сервісних додатків на різних мовах програмування та програмних платформах. Застосовуючі ці дані при розробці мікросервісних систем можна значно прискорити процес вибору технологічного стеку, прискорити процес розробки та розгортання. Також використання даної методики дозволить оптимізувати процес фунціонування мікросервісів, що зменшить витрати на апаратне забезпечення системи та її моніторинг.

Мікросервісна архітектура зараз набирає значних обертів, проте алгоритмів для проектування мікросервісних систем майже немає. Розробника є унікальної та може значно спростити процес створення мікросервісних систем за допомогою різних програмних засобів.

4.4. Оцінка економічної ефективності впровадження програмного забезпечення.

Розроблена методика дозволяє проектувати мікросервісні додатки зі значним скороченням як матеріальних витрат, так і тимчасових, що підтверджується розробленим програмним продуктом в даній магістерській роботі. Зараз багато різних компаній розробляють масштабні мікросервісні додатки, використанн даної методики дозволить їм:

- оптимтимізувати процес розгортання і тим самим знизити вартість розробки;
- знизити витрати на апаратне забезбечення;
- зменшити кількість розробників;
- скоротити час розробки мікросервісів.

4.5. Висновки

Ціна розробки даного програмного забезпечення без урахування ПДВ складає 147140 грн (сто сорок сім тисяч сто сорок гривень).

Час, використаний на розробку складає приблизно одинадцять місяців.

Проект має соціальний ефект, через що складно вирахувати економічну вигоду та строк окупності.

ВИСНОВКИ

В даній дипломній роботі було досліджено основні концепції побудови додатків на базі мікросервісної архітектури. Розглянуто основні особливості, переваги та недоліки даного підходу до побудови програмних систем, порівняно з класичним монолітним рішенням. Монолітна архітектура дуже добре розв'язує свої задачі, але із зростанням складності вона вже не може якісно вирішувати свої функції, тому розвинулися сервіс-орієнтовані архітектури, прикладом якої є мікросервісна архітектура.

Було детально розглянуто основні принципи проектування та розгортання даних систем, основні шаблони інтеграції мікросервісів, технології комунікації. В тестовому прикладі було продемонстровано працездатність вищенаведених концепцій.

Також було розглянуто ключові компоненти для побудови мікросервісних систем, які утворюють інфраструктурний рівень та надають необхідну гнучкість всій системі. До даних компонентів відносяться: сервіс єдиного входу, сервіс відкриття, балансувальних навантаження, автоматичний вимикач. Дані шаблони було перевірено на існуючих відкритих реалізаціях.

Однією з головних переваг мікросервісної архітектури є її гетерогенність, тому було досліджено базові бібліотеки для створення незалежних сервісних додатків на різних мовах програмування та програмних платформах. Значну роль у розвитку та створенні програми є підтримка необхідного середовища, що забезпечується системами віртуалізації та постійної інтеграції. Було розглянуто систему Docker для забезпечення відповідного програмного середовища, а також проаналізовано основні підходи до налаштування процесів постійної інтеграції. Також було наведено основні питання які стосуються проблем безпеки міжсервісного спілкування в системі, а також підходів до масштабування додатку.

Як результат всіх проведених досліджень, було розроблено методику для розробки мікросервісних додатків та тестову систему, яка підтверджує концепцію даної архітектури та демонструє її працездатність. Для реалізації даної системи було використано дві різні мови програмування – Java та Python,

а також інструменти для забезпечення інтеграції та внутрішні бібліотеки, які надають змогу розгортати мережеві розподілені додатки.

Підсумовуючи, можна стверджувати, що мікросервісна архітектура має право на існування, але не претендує стати монополістом для побудови інформаційних систем. Більше того, мікросервіси не рекомендовано створювати без глибокого попереднього аналізу предметної області та чіткого виділення обмежених контекстів, а також пропонується створювати мікросервіси на базі існуючого моноліту.

Дана архітектура є новою та перспективною у сучасному проектуванні інформаційних систем та, цілком можливо, її використання набуде масовий характер.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. М. Фаулер. Архитектура корпоративных программных приложений М. Фаулер. – Издательский дом Вильямс, 2006 – 544 с.

2. Ньюмен С. Создание микросервисов / Ньюмен С. – СПб.: Питер, 2017– 304 с.

3. Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions / Gregor Hohpe, Bobby Woolf – Addison-Wesley, 2004 – 736p.

4. Chris Richardson. From Design to Deployment / Chris Richardson, Floyd Smith, 2016. – 74 p.

5. E. Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software / E. Evans – Addison-Wesley, 2003 – 560 p.

6. Martin Fowler – Microservices – Режим доступу: http://martinfowler.com/articles/microservices.html I. Nadareishvili.

7. Microservice Architecture: Aligning Principles, Practices, and Culture / I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen – O'Reilly Media, 2016 – 146p.

8. Introduction to microservices. – Режим доступу: https://nginx.com/blog/introduction-to-microservices/

9. Using an API Gateway. – Режим доступу: https://nginx.com/blog/buildingmicroservices-using-an-api-gateway/.

10. Service Discovery. – Режим доступу: https://nginx.com/blog/service-discovery-ina-microservices-architecture/

11. Офіційний сайт Docker. – Режим доступу: https://docker.com/ –.

12. Офіційний сайт C++ Micro Services. – Режим доступу: http://cppmicroservices.org/

13. Офіційний сайт Pistache framework. – Режим доступу: http://pistache.io/

14. Офіційний сайт Spring framework. – Режим доступу: https://spring.io

15. Офіційний сайт Spark framework. – Режим доступу: https://sparkjava.com/

16. Офіційний сайт Restlet. – Режим доступу: https://restlet.com/

17. Офіційний сайт Flask. – Режим доступу: http://flask.pocoo.org/

18. Офіційний сайт Tornado. – Режим доступу: http://tornadoweb.org/

19. Офіційний сайт Nameko. – Режим доступу: https://nameko.readthedocs.io

20. Frankel D. Model Driven Architecture: Applying MDA to Enterprise Computing. – Wiley, 2003, 352 p.

21. Grassle P. et al. UML 2.0 in action: a project based tutorial. – Packt Publishing, 2005. – 232 p.

22. Gunasekaran A. (Ed.) Modeling and analysis of enterprise information systems. – IGI Global, 2007. – 392 p.

23. Hruby P. Model–Driven Design Using Business Patterns. – Springer, 2006. – 359 p.

24. Iordache O. Modeling Multi–Level Systems. – Springer, 2011. – 243p.

25. Karakostas B., Zorgios Y. Engineering Service Oriented Systems: A Model Driven Approach. – IGI Global, 2008, 374 p.

26. Kurbel K.E. The Making of Information Systems: Software Engineering and Management in a Globalized World. – Springer, 2008. – 602 p.

27. Langer A.M. Analysis and Design of Information Systems. – Springer, 2008. – 436 p.

28. Lischner R. Delphi in a Nutshell. O'Reilly Media, 2000. – 576 p.

29. Mouheb D. et al. Aspect–Oriented Security Hardening of UML Design Models. – Springer, 2015. – 246 p.

30. Object Management Group. OMG Unified Modeling Language (OMG UML), Infrastructure. Version 2.4.1. – Object Management Group, 2010. – 230 p.

31. Pallab Saha. Handbook of Enterprise Systems Architecture in Practice. – Idea Group Inc (IGI), 2007. – 471 p.

32. Podeswa H. UML for the IT Business Analyst: A Practical Guide to Requirements Gathering Using the Unified Modeling Language. – Course Technology PTR, 2009. – 400 p.

33. Rebovich G.J. Enterprise Systems Engineering: Advances in the Theory and Practice. – CRC Press, 2010. – 477 p.

34. Saaty T.L., Vargas L.G. Models, Methods, Concepts & Applications of the Analytic Hierarchy Process. – Springer US, 2001. – 333 p.

35. Seidl M., Scholz M., Huemer C., Kappel G. UML @ Classroom: An

Introduction to ObjectOriented Modeling. – Springer, 2015. – 215 p.

36. Seruca I., Cordeiro J., Hammoudi S., Filipe J. Enterprise Information Systems VI. – Springer, 2006. –334 p.

37. Stahl Harry. Cross Platform Development with Delphi XE7 & FireMonkey for Windows & MAC OS X. Harry Stahl, 2015. – 312 p.

38. Swart B. Delphi 8: Migrating Delphi applications to the Microsoft. NET Framework with Delphi 8. Borland Software Corporation, 2004. – 22 p.

39. Teti D. Delphi Cookbook: 50 hands–on recipes to master the power of Delphi for cross–platform and mobile development on Windows, Mac OS X, Android, and iOS. Packt Publishing, 2014. – 329 p.

40. Vogel O. Software Architecture: A Comprehensive Framework and Guide for Practitioners. – Springer, 2011. – 550 p.

41. Ward John, Peppard Joe. Strategic Planning for Information Systems. – John Wiley & Sons, 2002. – 641 p.

42. Акимов В.А., Воронов Е.М., Крыжановская Т.Г. Проектирование информационных систем. – М.: МГМУ «МАМИ» (Университет машиностроения), 2013. – 126 с.

43. Арунянц Г.Г. Проектирование информационных систем. – Калининград: Балтийский институт экономики и финансов, 2010. – 254 с.

44. Архангельский А.Я. Delphi 2006. Справочное пособие. – М.: ООО «Бином–Пресс», 2006. – 1152 с.

45. Бабич А.В. Введение в UML. – М.: НОУ ИНТУИТ, 2016. – 209 с.

46. Бакнелл Дж. М. Фундаментальные алгоритмы и структуры данных в Delphi. – М.: ДиаСофт, 2003. – 560 с.

47. Блинков Ю.А. Проектирование информационных систем. – Саратов: Саратовский государственный университет, 2010. – 377 с.

48. Брукс П. Проектирование процесса проектирования. – М.: Вильямс, 2013. – 464 с.

49. Буторин Д. MS Agent и Speech API в Delphi. – СПб: БХВ–Петербург, 2005. – 440 с.

50. Буч Г., Рамбо Д., Якобсон И. Язык UML. Руководство пользователя. 2–е

изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.

51. Варфел Тодд. Прототипирование. Практическое руководство. – М: Манн, Иванов и Фербер, 2013. – 389 с.

52. Гвоздева Т.В., Баллод Б.А. Проектирование информационных систем. – Ростов н/Д.: Феникс, 2009. – 512 с.

53. Головчинер М.Н. Проектирование информационных систем. – Томск: ТГУ, 2015. – 110 с.

54. Гома Х. UML–проектирование систем реального времени параллельных и распределенных приложений. – М.: ДМК–Пресс, 2011. – 704с.

55. Грекул В.И., Денищенко Г.Н., Коровкина Н.Л. Проектирование информационных систем. – М.: Интернет–Университет Информационных Технологий «Интуит», 2016. – 570 с.

56. Грибачев К.Г. Model Driven Architecture. Разработка приложений баз данных.– СПб.: Изд–во Питер, 2004. – 352 с.

57. Коцюба И.Ю., Чунаев А.В., Шиков А.Н. Основы проектирования информационных систем. Учебное пособие. – СПб.: Университет ИТМО, 2015. – 206 с.

58. Кузнецов М.Б. Трансформация UML–моделей и ее применение в технологии MDA. – М.: Институт системного программирования РАН, 2005. – 13 с.

59. Культин Н. Основы программирования в Delphi 2010. – СПб.: БХВ–Петербург, 2010. – 438 с.

60. Ларман Крэг. Применение UML 2.0 и шаблонов проектирования. 3–е изд. – М.: Вильямс, 2013. – 736 с.

61. Martin L. Abbott. The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise. – Addison-Wesley Professional, 2015. – 624 с.

62. Хабрахабр, Микросервисы (Microservices). Режим доступу: – https://habrahabr.ru/post/249183/

63. Martin Fowler, Microservices. Режим доступу: – https://martinfowler.com/articles/microservices.html

64. Vikram Murugesan, Microservices Deployment Cookbook. – Packt Publishing, 2017. – 378 с.

65. Sam Newman. Building Microservices: Designing Fine-Grained Systems. – O'Reilly Media, 2015. – 280 с.

66. Sourabh Sharma, Rajesh RV, David Gonzalez. Microservices: Building Scalable Software. – Packt Publishing Limited, 2017. – 919 с.

67. Rajesh RV, Spring Microservices. – Packt Publishing, 2016. – 436 с.

68. Pethuru Raj. Docker: Creating Structured Containers. – Packt Publishing, 2016. – 320 с.

69. Docker, Docker Documentation. Режим доступу: – https://docs.docker.com/

70. Randall Smith. Docker Orchestration. – Packt Publishing, 2017. – 284 с.

71. Kubernetes, Kubernetes Documentation. Режим доступу: https://kubernetes.io/docs/home/

72. Docker, Swarm mode overview. Режим доступу: – https://docs.docker.com/engine/swarm/

73. Fabrizio Soppelsa, Chanwit Kaewkasi. Native Docker Clustering with Swarm. – Packt Publishing, 2016. – 280 с.

74. Хабрахабр, Основы Kubernetes. Режим доступу: – https://habrahabr.ru/post/258443/

75. Platform9, Container Orchestration Tools: Compare Kubernetes vs Docker Swarm. Режим доступу: – https://platform9.com/blog/compare-kubernetes-vs-docker-swarm/

76. UpCloud, Docker Swarm vs. Kubernetes: Comparison of the Two Giants in Container Orchestration. Режим доступу: – https://www.upcloud.com/blog/docker-swarm-vskubernetes/ 85

77. Хабрахабр, Функциональная безопасность, Часть 2 из 7. МЭК 61508. – Режим доступу: https://habrahabr.ru/post/309636/

78. Viktor Farcic. The DevOps 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices. – Leanpub, 2016. – 430 с.

Додаток А

Лістинг програми:

```
import com.piggymetrics.account.domain.User;
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@FeignClient(name = "auth-service")
public interface AuthServiceClient {

    @RequestMapping(method = RequestMethod.POST, value =
"/uaa/users", consumes = MediaType.APPLICATION_JSON_UTF8_VALUE)
    void createUser(User user);

}
import com.piggymetrics.account.domain.Account;
import org.springframework.cloud.netflix.feign.FeignClient;
import org.springframework.http.MediaType;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@FeignClient(name = "statistics-service")
public interface StatisticsServiceClient {
```

```
        @RequestMapping(method = RequestMethod.PUT, value =
    "/statistics/{accountName}", consumes =
    MediaType.APPLICATION_JSON_UTF8_VALUE)
        void updateStatistics(@PathVariable("accountName") String
    accountName, Account account);

}
    import com.piggymetrics.account.domain.Account;
    import com.piggymetrics.account.domain.User;
    import com.piggymetrics.account.service.AccountService;
    import org.springframework.beans.factory.annotation.Autowired;
    import org.springframework.security.access.prepost.PreAuthorize;
    import org.springframework.web.bind.annotation.*;

    import javax.validation.Valid;
    import java.security.Principal;

    @RestController
    public class AccountController {

        @Autowired
        private AccountService accountService;

        @PreAuthorize("#oauth2.hasScope('server') or #name.equals('demo')")
        @RequestMapping(path = "/{name}", method = RequestMethod.GET)
        public Account getAccountByName(@PathVariable String name) {
            return accountService.findByName(name);
        }

        @RequestMapping(path = "/current", method = RequestMethod.GET)
        public Account getCurrentAccount(Principal principal) {
```

```
        return accountService.findByName(principal.getName());
    }

    @RequestMapping(path = "/current", method = RequestMethod.PUT)
    public void saveCurrentAccount(Principal principal, @Valid
@RequestBody Account account) {
        accountService.saveChanges(principal.getName(), account);
    }

    @RequestMapping(path = "/", method = RequestMethod.POST)
    public Account createNewAccount(@Valid @RequestBody User user) {
        return accountService.create(user);
}

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseStatus;

@ControllerAdvice
public class ErrorHandler {

    private final Logger log = LoggerFactory.getLogger(getClass());

    // TODO add MethodArgumentNotValidException handler
    // TODO remove such general handler
    @ExceptionHandler(IllegalArgumentException.class)
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public void processValidationError(IllegalArgumentException e) {
```

```
                log.info("Returning HTTP 400 Bad Request", e);
            }
}
```

```
import org.codehaus.jackson.annotate.JsonIgnoreProperties;
import org.hibernate.validator.constraints.Length;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import javax.validation.Valid;
import javax.validation.constraints.NotNull;
import java.util.Date;
import java.util.List;

@Document(collection = "accounts")
@JsonIgnoreProperties(ignoreUnknown = true)
public class Account {

    @Id
    private String name;

    private Date lastSeen;

    @Valid
    private List<Item> incomes;

    @Valid
    private List<Item> expenses;

    @Valid
    @NotNull
```

```
private Saving saving;

@Length(min = 0, max = 20_000)
private String note;

public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}

public Date getLastSeen() {
        return lastSeen;
}

public void setLastSeen(Date lastSeen) {
        this.lastSeen = lastSeen;
}

public List<Item> getIncomes() {
        return incomes;
}

public void setIncomes(List<Item> incomes) {
        this.incomes = incomes;
}

public List<Item> getExpenses() {
        return expenses;
```

```
        }

        public void setExpenses(List<Item> expenses) {
            this.expenses = expenses;
        }

        public Saving getSaving() {
            return saving;
        }

        public void setSaving(Saving saving) {
            this.saving = saving;
        }

        public String getNote() {
            return note;
        }

        public void setNote(String note) {
            this.note = note;
        }
}

    public enum Currency {

        USD, EUR, RUB;

        public static Currency getDefault() {
            return USD;
        }
}
```

```
import javax.validation.constraints.NotNull;
import java.math.BigDecimal;

public class Saving {

	@NotNull
	private BigDecimal amount;

	@NotNull
	private Currency currency;

	@NotNull
	private BigDecimal interest;

	@NotNull
	private Boolean deposit;

	@NotNull
	private Boolean capitalization;

	public BigDecimal getAmount() {
		return amount;
	}

	public void setAmount(BigDecimal amount) {
		this.amount = amount;
	}

	public Currency getCurrency() {
		return currency;
```

```
    }

    public void setCurrency(Currency currency) {
        this.currency = currency;
    }

    public BigDecimal getInterest() {
        return interest;
    }

    public void setInterest(BigDecimal interest) {
        this.interest = interest;
    }

    public Boolean getDeposit() {
        return deposit;
    }

    public void setDeposit(Boolean deposit) {
        this.deposit = deposit;
    }

    public Boolean getCapitalization() {
        return capitalization;
    }

    public void setCapitalization(Boolean capitalization) {
        this.capitalization = capitalization;
    }
}
```

```
public enum TimePeriod {

	YEAR, QUARTER, MONTH, DAY, HOUR}
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface AccountRepository extends CrudRepository<Account, String>
{

	Account findByName(String name);

}
@SpringBootApplication
@EnableResourceServer
@EnableDiscoveryClient
@EnableOAuth2Client
@EnableFeignClients
@EnableGlobalMethodSecurity(prePostEnabled = true)
@EnableConfigurationProperties
@Configuration
public class AccountApplication extends ResourceServerConfigurerAdapter {

	@Autowired
	private ResourceServerProperties sso;

	public static void main(String[] args) {
		SpringApplication.run(AccountApplication.class, args);
	}

	@Bean
```

```
	@ConfigurationProperties(prefix = "security.oauth2.client")
	public ClientCredentialsResourceDetails clientCredentialsResourceDetails() {
		return new ClientCredentialsResourceDetails();
	}

	@Bean
	public RequestInterceptor oauth2FeignRequestInterceptor(){
		return new OAuth2FeignRequestInterceptor(new DefaultOAuth2ClientContext(), clientCredentialsResourceDetails());
	}

	@Bean
	public OAuth2RestTemplate clientCredentialsRestTemplate() {
		return new OAuth2RestTemplate(clientCredentialsResourceDetails());
	}

	@Bean
	public ResourceServerTokenServices tokenServices() {
		return new CustomUserInfoTokenServices(sso.getUserInfoUri(), sso.getClientId());
	}

	@Override
	public void configure(HttpSecurity http) throws Exception {
		http.authorizeRequests()
				.antMatchers("/" , "/demo").permitAll()
				.anyRequest().authenticated();
	}
}
```