МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютерних інформаційних технологій

ДОПУСТИТИ ДО ЗАХИСТУ

Завідувач кафедри

_____Аліна САВЧЕНКО

«___»_____2021 р.

ДИПЛОМНА РОБОТА

(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ «МАГІСТРА»

ЗА СПЕЦІАЛІЗАЦІЄЮ «ІНФОРМАЦІЙНІ УПРАВЛЯЮЧІ СИСТЕМИ ТА ТЕХНОЛОГІЇ (ЗА ГАЛУЗЯМИ)»

Тема: «Система контейнеризації мікросервісів на базі Docker»

Виконавець: Кондратенко Кирило Олександрович

Керівник: к.т.н., доцент Райчев Ігор Едуардович

Нормоконтролер: _____ Ігор РАЙЧЕВ

Київ 2021

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп'ютерної та програмної інженерії

Кафедра комп'ютерних інформаційних технологій

Галузь знань, спеціальність, спеціалізація: 12 "Інформаційні технології", 122 "Комп'ютерні науки", "Інформаційні управляючі системи та технології (за . галузями)"

ЗАТВЕРДЖУЮ
Завідувач випускової кафедри
_____Аліна САВЧЕНКО
«____» _____ 2021 р.

ЗАВДАННЯ

на виконання дипломної роботи студента

Кондратенка Кирило Олександровича

(П.І.Б. випускника)

1. **Тема роботи**: «Система контейнеризації мікросервісів на базі Docker» затверджена наказом ректора від 12.10.2021 за №2228/ст.

2. **Термін виконання роботи**: з 12.10.2021 по 31.12.2021

3. **Вихідні дані роботи**: тема, перелік літератури, календарний план-графік написання дипломної роботи.

4. **Зміст пояснювальної записки**:

Вступ
Розділ 1 Властивості структури веб-додатків
Розділ 2 Розгляд мікросервісної будови
Розділ 3Спосіб контейнеризації мікросервісів- Docker
Розділ 4 Система контейнеризації мікросервісів
Висновки
Список використаних джерел
Застосунок А
Застосунок Б
Застосунок В

5. **Перелік обов’язкового ілюстративного матеріалу**: рисунки,слайди.

6. **Календарний план-графік**:

№ з/п	Завдання	Термін виконання	Підпис керівника
1	Аналіз і опрацювання літератури	12.10.2021 – 15.10.2021	
2	Привести консультацію з науковим керівником щодо розділів дипломної роботи	17.10.21	
3	Підготовка та написання розділу 1	18.10.2021 – 29.10.2021	
4	Підготовка та написання розділу 2	29.10.2021 – 15.11.2021	
5	Підготовка та написання розділу 3	15.11.2021 – 24.11.2021	
6	Підготовка та написання розділу 4	24.11.2021 – 3.12.2021	
7	Оформлення пояснювальної записки	3.12.2021 – 4.12.2021	
8	Оформлення графічної частини роботи	4.12.2021 – 8.12.2021	
9	Подати дипломну роботу керівнику	10.12.2021	
10	Підготовка до захисту дипломної роботи	12.12.2021 – 20.12.2021	

7. **Дата видачі завдання**: 12.10.2021 р.

Керівник дипломної роботи: _____ Ігор РАЙЧЕВ
(підпис керівника)

Завдання прийняв до виконання: _____ Кирило КОНДРАТЕНКО
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до дипломної роботи «Система контейнеризації мікросервісів на базі Docker» містить: 113 сторінок, 44 рисунки, 3 додатки та 4 таблиці.

Ключові слова: МІКРОСЕРВІС, DOCKER, КЛАСТЕРИЗАЦІЯ, ОРКЕСТРАЦІЯ, МОНІТОРИНГ.

Об'єкт дослідження: система контейнеризації.

Мета дипломної роботи: розробка мікросервісної архітектури, кластеризація, оркестрація, моніторинг на базі Docker.

Спосіб дослідження: дослідження існуючих концепцій проектування мікросервісної архітектури.

Область застосування: веб-додатки.

Під час роботи над дипломною роботою, було проведено дослідження принципів побудови мікросервісної архітектури, проаналізовано основні можливості і функції системи контейнеризації Docker.

Було спроектовано масштабовану стабільну кластерну інфраструктуру мікросервісів на базі Docker. Також було реалізовано основну концепцію безперервної доставки, налаштовано формат взаємодії між мікросервісами, підключено моніторинг системи.

Результати роботи: створена інфраструктура може бути створена для реалізації продуктів. На базі створеної інфраструктури можна побудувати продукт із мінімальними оновленнями архітектури. Дану роботу можна використовувати у якості допоміжного матеріалу розробникам ПЗ при проектуванні і розробці мікросервісних архітектур.

Розробка та дослідження проводилися під управлінням ОС Windows 11 у інтегрованому середовищі розробки PHPSTORM, мова програмування – PHP.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ.

DDD	—	Domain Driven Design – предметно-орієнтована архітектура
ОС	—	Операційна система
IDE	—	Integrated development environment – інтегроване середовище розробки
ПЗ	—	Програмне забезпечення
CI	—	Continuous Integration – безперервна інтеграція
CD	—	Continuous Delivery – безперервна доставка
VM	—	Virtual Machine – віртуальна машина
LB	—	Load Balancer – балансер навантаження
LXC	—	Linux Containers – контейнери Linux
REST	—	Representational State Transfer – передача репрезентативного стану
API	—	Application Programming Interface – прикладний програмний інтерфейс

ВСТУП

Багато додатків, з якими ми регулярно перетинаємося (інтернет-банки, розважальні сервіси на кшталт YouTube і так далі), часто створені з використанням безлічі технологій, які якось уживаються під одним дахом і не виглядають розрізнено.

Мікросервісна архітектура - це підхід, при якому єдина програма будується з безлічі слабозв'язаних компонентів меншого розміру, що підтримують незалежне розгортання.

Мікросервіси – це різновид сервіс-орієнтованої архітектури (SOA), що використовується для формування розподілених програмних систем. Модулі в мікросервісній архітектурі взаємодіють за допомогою мережу, при цьому виконуючи єдину мету. На даний момент мікросервіси поступово витісняють монолітні програми та перетворюються на стандарт розвитку програмних систем.

Важливо розуміти, що під сервісом розуміється цілий набір послуг та певний функціонал, що його надають споживачеві. А мікросервіси – це дроблення функціоналу те щоб він був доступний іншим частинам системи. Причому дроблення функціоналу настільки дрібне, що всередині кожного мікросервісу реалізовано дуже невелику кількість функцій.

Метою роботи є зниження ресурсних та тимчасових витрат у розробці, тестуванні та розгортанні додатків, що використовують мікросервісну архітектуру. Об'єктом дослідження є технологія контейнеризації.

Архітектура мікрослужб (MSA) дозволяє командам, які розробляють програмне забезпечення, оптимізувати робочі процеси випуску релізів. До компаній, які відкрито вибирають цей метод розробки програмного забезпечення, входять Amazon, Netflix та eBay. Прагнучи зробити свій внесок у роботу спільноти, вони поділилися своїм досвідом та інструментами розробки, щоб допомогти іншим впровадити цей метод.

В даний час багато компаній, такі як Netflix, Apple, Instagram і Pinterest перенесли свої програми та системи на мікросервіси, оскільки дана архітектура

дозволяє цим компаніям масштабувати свої обчислювальні ресурси відповідно до їх використання. Ця необхідність виникла внаслідок того, що сучасні веб-програми мають високі вимоги для повноцінної роботи, такі як можливість надання програмного інтерфейсу, обробка великої кількості запитів, масштабованість, забезпечення високої швидкості доступу до даних, забезпечення високої надійності відмовостійкості. Перелічені гіганти зіткнулися з різними проблемами, які вирішено завдяки переходу на мікросервіси.

Netflix зіштовхнувся зі складністю перетворення інформації величезної кількості клієнтів. Коли компанії виявили, що його зростання випереджає можливості традиційної монолітної архітектури, мікросервіси виявилися вірним рішенням для масштабованості. У середньому вечорами у будні на Netflix припадає майже третина всього інтернет-трафіку в Північній Америці, а опівночі падає до мінімуму. Мікросервісна архітектура стала порятунком від падіння сервісу при такому нестабільному навантаженні, дозволивши розгортати додаткові сервери в піковий годинник.

Будучи одним з найбільш швидко зростаючих сайтів в інтернеті, Pinterest використовує мікросервіси, щоб пристосуватися до різних рівнів трафіку. У цьому зберігає неймовірно малу команду. Їхній онлайн-сервіс спеціально розроблений для агрегації великих обсягів даних, що стало можливим завдяки використанню мікросервісів.

Instagram звернувся до мікросервісів для підтримки свого зростання та задоволення великої потреби у масштабуванні. Сервіс вперше був запущений у 2010 році як простий монолітний застосунок. Як результат, протягом декількох годин сервер виявився перевантаженим, а Instagram довелося розгорнути мікросервіси. За допомогою шість місяців компанія вже працювала з трьома мільйонами постійних користувачів, не маючи проблем з трафіком.

Apple звернулася до мікросервісної архітектури, щоб залишатися на передовій позиції у сфері технологій. Саме це сталося під час управління випуском Siri - програми, що імітує здатність людини слухати та відповідати на запитання користувачів. Мікросервіси дозволили Apple розширювати і адаптувати Siri набагато швидше і без перерв обслуговування користувачів.

Останні навіть не помітили, що протягом тривалого часу компанія робила оновлення версій та впроваджувала новий функціонал.

Дані компанії є гарним прикладом того, як гнучкість мікросервісів можна використовувати для впровадження інновацій та оптимізації витрат. Тисячі інших брендів у всьому світі роблять те саме, відкриваючи нові можливості мікросервісної архітектури.

Docker – це програмне забезпечення для автоматизації розгортання і управління додатками в середовищах з підтримкою контейнеризації.

Docker — це програмне забезпечення, яке дає можливість на певній ділянці пам'яті ізольовано встановити необхідну ОС (операційну систему), версію Java, настроїти змінні оточення, встановити різні залежності та надати доступ лише за певних умов. При цьому цю програму зовсім не хвилюватиме, що відбувається довкола.

Тепер простими словами: це можливість, образно кажучи, відокремити собі кімнату, зробити там ремонт, який вам подобаються, купити меблі та техніку до смаку, встановити замок на двері та ключ видати тільки братові. При цьому якщо ви подивитеся в іншу кімнату, все буде інакше і жодних проблем не буде.

Також уточню, що Docker не впливає на код програми. Можливо, для зручнішого застосування цієї технології доведеться дуже уважно ставитися до архітектури проекту, але це не означає, що якщо проект старий, то контейнеризацію в ньому не можливо застосувати.

Як правило мікросервісна архітектура використовується як один з способів масштабування додатків:

- Sharding (*розбиття*) – розміщення на різних вузлах;
- Mirroring (*створення дзеркал*) – дублювання на декількох однакових вузлах;
- Модульність – визначається поділом функціональності на декілька сервісів, при чому кожен сервіс може бути створений своїми засобами розробки, а також написаний на різних мовах програмування.

Останнім часом тренд на використання мікросервісів набув популярності. Це очевидно, так як підприємства хочуть, щоб їх програмне забезпечення було більш зручнішим та дешевшим. У мікросервісній архітектурі додаток розбивається на колекцію не дуже пов'язаних служб.

Використовуючи мікросервісний підхід можна: збільшити можливість бізнес-додатків в кілька разів; протягом короткого часу встановлювати декілька оновлень, не ризикуючи порушити цілістність корпоративної інформаційної системи; розробляти оновлення паралельно силами декількох людей, що створюють версії не кооперуючись один з одним.

Позитивні сторони

Чіткий розподіл за модулями. Завжди було зрозуміло, як створена частина коду. Просто додавання нових функції.

Висока доступність. Якщо якась частина проєкту зламається – зламається весь мікросервіс. З мікросервісами інакше: послуги можуть працювати не всі (не критичні, на кшталт авторизації), але програма при цьому залишиться доступною.

Різноманітні технології. При розробці кожного сервісу ви можете вибирати інструменти, які найкраще підходять для конкретної цілі в цьому сервісі. Вибрати оптимальну базу даних та комфортні інструменти для роботи. Мікросервісна архітектура дозволяє створити якусь нову технологію на окремому сервісі, не переробляючи всю програму.

Простота розгортання. Кожен сервіс можна ввімкнути самостійно, що робить процес розгортання та налагодження чистішим.

Недоліки

Складність розробки. Якщо вам потрібне швидше рішення (інший тип, невелика програма, стислі терміни), то мікросервіси можуть не підійти. Швидкість розробки – висока плата за ціну та роздрібленність.

Складність підтримки. Кожен мікросервіс потребує свого обслуговування, тому потрібен постійний ручний або автономний моніторинг.

Орієнтована на служби архітектура (SOA) та мікрослужби - це два різновиди архітектури вищого порядку (архітектури веб-служб). Мікрослужби можна назвати полегшеною версією SOA. Різниця між двома типами архітектур полягає у формальній класифікації типів служб. SOA включає чотири базові типи служб: бізнес-служби, корпоративні служби, служби додатків та інфраструктурні служби. Відповідно до типу служби визначається її відповідальність, пов'язана із конкретною областю. У свою чергу, архітектура мікрослужб включає лише два типи служб: функціональні та інфраструктурні.

В обох архітектурах використовується той самий набір стандартів, що діють на різних рівнях компанії. Модель MSA зобов'язана своїм існуванням успіху моделі SOA. Отже, модель MSA є підмножиною моделі SOA. У ній основна увага приділяється автономному виконанню кожного сервісу.

Мікрослужби поділяють великі завдання, притаманні конкретного бізнесу, на кілька незалежних баз коду. Монолітна архітектура програм є протилежністю мікрослужб. Моноліт - це єдина база коду, в якій об'єднані всі бізнес-завдання. Використовувати моноліти зручно на початкових етапах проектів для скорочення розумових зусиль з управління кодом та полегшення розгортання. Це дозволяє одразу випускати все, що є в монолітному додатку.

Багато проектів розпочинаються як монолітні, а потім, у міру розвитку, переходять до архітектури мікрослужб. У міру додавання до монолітного проекту нових можливостей рано чи пізно виникають складнощі при роботі кількох розробників з єдиною базою коду. Почастішають конфлікти в коді і збільшується ризик того, що при оновленні однієї можливості з'являться баги в іншій незв'язаній можливості. Якщо такі небажані ситуації виникають, можливо, настав час обговорити перехід на мікрослужби.

Давайте як приклад розглянемо гіпотетичний проект розробки програмного забезпечення для електронної комерції. У цій сфері є деякі чітко визначені області. Сайти електронної комерції мають систему автентифікації

для входу користувачів до системи та виходу із неї. Передбачено також кошик для збереження списку товарів, що зацікавили користувача. Білінгова система дозволяє користувачам оплачувати свої покупки.

В архітектурі мікрослужб зазначеним областям відповідали б незалежні служби. Як конкретний приклад можна навести білінгову систему. За умови достатньої кількості співробітників у компанії може бути сформована «білінгова команда», яка відповідає за розробку та контроль якості цієї білінгової мікрослужби. Для білінгової мікрослужби складалися б графіки релізів та плани розгортання. Для білінгової служби також був створений задокументований API з підтримкою різних версій, щоб служби могли підключатися до неї та використовувати її функціональні можливості.

Завдяки можливостям, наданим ядром Linux, і перевагам методу віртуалізації на рівні операційної системи було розроблено програмне забезпечення для вирішення вищевказаних проблем. Ця програма Docker розроблена однойменною компанією. Docker — це безкоштовна програма, заснована на ядрі Linux і просторах імен ядра, групах і LXC. Механізм групування дозволяє ізолювати центральний процесор, мережеві ресурси, а також ресурси пам'яті та вводу-виводу для груп процесів, які забезпечують такі функції:

• обмеженість ресурсів;

• Пріоритетне застосування - різним групам процесів може бути виділений різний обсяг ресурсів ЦП і пропускна здатність підсистеми введення-виводу.

• облік – розрахунок вартості окремих ресурсів для групи процесів;

• ізоляція - поділ поля імені для груп процесів, щоб одна група не мала доступу до процесів, мережевих з'єднань і файлів іншої групи;

• управління - зупинка груп процесів, створення контрольних точок і його перезапуск.

Використання цих механізмів дозволяє створювати ієрархічні групи процесів з певними властивостями ресурсів і забезпечувати управління

програмним забезпеченням, запускати безліч ізольованих прикладів Linux в одному вузлі. Такими прикладами є віртуальні середовища з власною областю процесів і мережевим стеком, який є механізмом контейнерів LXC-Linux. Використання таких середовищ усуває проблему монолітності серверів, пов’язану з непередбачуваною поведінкою процесів, що призводить до виведення з експлуатації всього сервера. Розподіл процесів в ізольованих середовищах не допускає відмови однієї програми, відмови інших програм і сервера в цілому, але також забезпечує необхідну толерантність до збоїв.

Рішення, запропоноване Docker, полягає у використанні віртуальних контейнерів на основі ізольованих процесних середовищ, розширення функціональності та можливостей LXC, а також створення повноцінної платформи для вирішення цілого ряду завдань, від розробки до міграції та вдосконалення програмного забезпечення. . На малюнку нижче показано порівняння повної віртуалізації з віртуальними контейнерами Docker.

РОЗДІЛ 1
ВЛАСТИВОСТІ СТРУКТУРИ ВЕБ-ДОДАТКІВ

1.1. Характеристика веб-додатків

Web-додаток – програма з певним набором функціоналу, що використовує як клієнт браузер. Іншими словами, якщо програмі для здійснення бізнес-логіки потрібне мережне з'єднання та наявність на стороні користувача браузера, його відносять до веб-додатку.

Усі дані зберігаються та перетворюються на серверах, отримати доступ до яких можна за допомогою браузер, мобільний додаток і тд, за допомогою протоколи HTTP або HTTPS (протокол із шифруванням інформації на базі TLS).

Візуально веб-застосунок схожий звичайний комп'ютерний застосунок, окрім того, що він працює за допомогою Інтернет.

За структурою архітектура веб-застосунків показує взаємодію між додатками, базами даних та іншими проміжними елементами.

Архітектура веб-програми в основному представляє відносини та взаємодії між такими компонентами, як інтерфейси користувача, монітори обробки транзакцій, бази даних та інші.

Основна мета – переконатися, що всі елементи правильно працюють кооперуючись.

Всі програми складаються з двох частин - клієнтської (front-end) та серверної (back-end).

Інтерфейс – це візуальна частина програми. Користувачі можуть бачити інтерфейс та взаємодіяти з ним. Клієнтський код дає відповідь на дії в застосунку. Серверна частина не є видимою для користувачів, але змушує їх дії працювати. Він обробляє логічні завдання та відповідає на HTTP-запити.

Тому, якщо ви вводите свої дані в реєстраційну форму, ви працюєте із зовнішнім інтерфейсом, коли ви натискаєте "введення" і реєструєтеся - це серверна частина змушує його працювати.

При правильній роботі клієнтська та серверна сторони становлять архітектуру програмного забезпечення веб-програми.

У будь-якому типовому веб-застосунку є два об’єкти (рис.1.1):

- програмний код з боку клієнта (*frontend складова*)
- програмний код з боку сервера (*backend складова*)

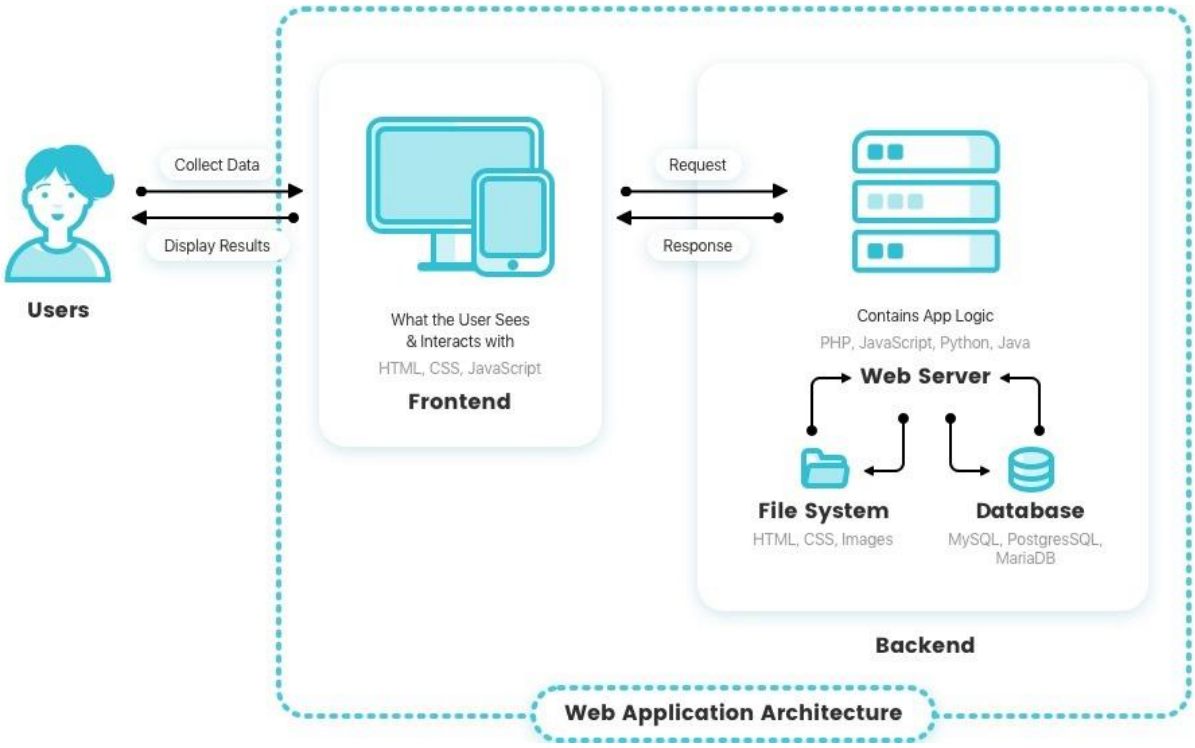


Рис.1.1 Взаємодія користувача із веб-додатком

Шари та компоненти архітектури веб-додатків

Щоб краще зрозуміти архітектуру веб-застосунку, вам слід зануритися в його компоненти та рівні. Веб-програми поділяють свої основні функції на рівні. Це дозволяє замінити чи оновлювати кожен шар незалежно.

Базові компоненти архітектури веб-додатків

Веб-архітектура має компоненти інтерфейсу користувача і структурні компоненти. Останні також поділяються на клієнтські та серверні.

Компоненти інтерфейсу користувача

Компоненти інтерфейсу користувача позначають всі елементи інтерфейсу, такі як журнали активності, інформаційні панелі, повідомлення, налаштування та багато іншого. Вони є частиною макету інтерфейсу веб-програми.

Структурні компоненти складаються з клієнтської та серверної сторін:

Клієнтський компонент розроблений з HTML, CSS чи JavaScript. Веб-браузери запускають код і перетворюють його на інтерфейс, тому немає необхідності в налаштуванні операційної системи.

Щодо серверного компонента, він побудований на Java, .Net, Node.JS, Python та інших мовах програмування. Сервер складається з двох частин - логіки програми та бази даних. Логіка програми – це центр керування веб-програмою. База даних відповідає за зберігання інформації (наприклад, ваші облікові дані).

1.2. Компоненти веб-механізму.

Існує чотири загальні рівні веб-додатків:

- Рівень вистави (PL)
- Рівень обслуговування даних (DSL)
- Рівень бізнес-логіки (BLL)

- Рівень доступу до даних (DAL)

Рівень вистави

PL відображає інтерфейс користувача і спрощує взаємодію з користувачем. Рівень представлення має компоненти інтерфейсу користувача, які візуалізують і показують дані для користувачів. Також існують компоненти процесу користувача, які задають взаємодію з користувачем. PL надає всю потрібну інформацію клієнтській стороні. Основна задача рівня представлення – отримати вхідні дані, обробити запити користувачів, надіслати їх у службу даних та показати результати.

Шар бізнес-логіки

BLL відповідає за належний обмін даними. Цей рівень визначає логіку бізнес-операцій та правил. Вхід на сайт – це приклад рівня бізнес-логіки.

Рівень служби даних

DSL передає дані, опрацьовані рівнем бізнес-логіки, на рівень представлення. Данний рівень підтримує безпеку даних, ізолюючи бізнес-логіку клієнта.

Рівень доступу до даних

DAL пропонує спрощений доступ до даних, що зберігаються у постійних сховищах, таких як бінарні файли та файли XML. Рівень доступу до даних також управляє операціями CRUD – створення, читання, оновлення, видалення.

Найчастіше веб-програми складаються як мінімум з трьох основних компонентів:

Клієнтська частина веб-програми - це графічний інтерфейс. Візуальний вигляд. Графічний інтерфейс відображається у браузері. Користувач взаємодіє з веб-програмою саме за допомогою браузер, клацаючи за посиланнями та кнопками.

Серверна частина веб-застосунку — це програма або скрипт, що знаходиться на сервері, який аналізує запити користувача (точніше, запити браузера). Як правило, серверна частина веб-програми програмується на PHP.

При кожному переході користувача за посиланням браузер надсилає запит на сервер. Сервер обробляє цей запит, викликаючи певний PHP-скрипт, який формує веб-сторінку, описану мовою HTML, і надсилає клієнту за допомогою мережу. Браузер відразу відображає отриманий результат у вигляді чергової веб-сторінки.

Між цими запитами і відповідями існують численні посередники, так звані проксі (*proxy*), які виконують різні операції і працюють як шлюзи або кеш (рис. 1.2).

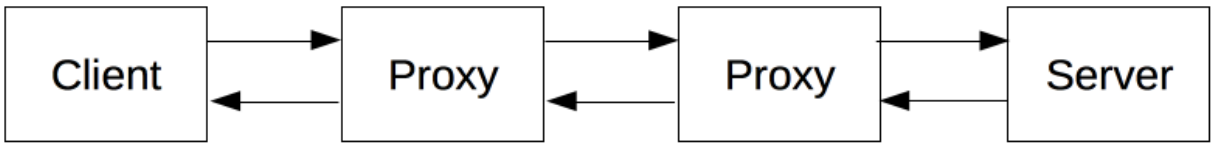


Рис. 1.2. Посередники між клієнтом і сервером

Зв'язок між клієнтами та серверами здійснюється за допомогою запитів та відповідей (рис. 1.3):

- Браузер за допомогою Інтернет надсилає HTTP-запити веб-серверу
- Веб-сервер викликає PHP-скрипт
- PHP-скрипт надсилає запит до бази даних за необхідності
- В результаті PHP-скрипт повертає клієнту веб-сторінку, якувізуалізує браузер.

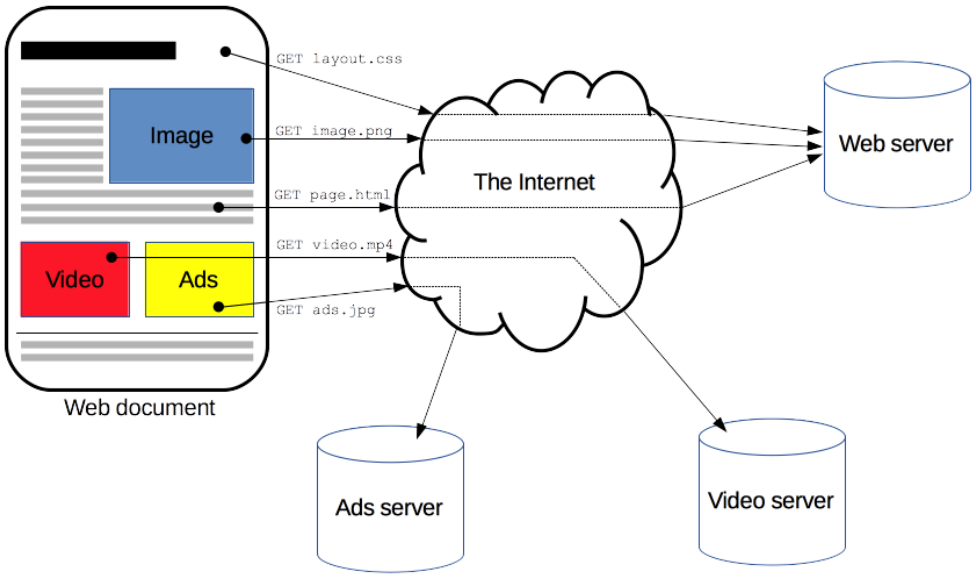


Рис. 1.3. Взаємодія клієнта і сервера

HTTP є протоколом прикладного рівня, який використовує властивості іншого протоколу – TCP (або TLS – захищений TCP) – для пересилки своїх повідомлень (рис. 1.4). HTTP — широко поширений протокол передачі даних, спочатку призначений надсилання гіпертекстових документів (тобто документів, які зазвичай містять посилання, дозволяють організувати перехід до інших документів).

Відповідно до специфікації OSI, HTTP є протоколом прикладного (верхнього, 7-го) рівня. Актуальна на даний момент версія протоколу HTTP 1.1 описана в специфікації RFC 2616.

Власне протокол HTTP передбачає використання клієнт-серверної структури передачі. Клієнтська програма формує запит і надсилає його на сервер, вже після чого серверне програмне забезпечення оброблює цей запит, та компонує відповідь і передає його назад клієнту. Тільки після цього клієнтська програма може продовжити надсилати інші запити, які будуть оброблені аналогічним чином.

А ще HTTP часто використовується як протокол передавання інформації для інших протоколів прикладного рівня, таких як SOAP, XML-RPC та WebDAV. У такому разі кажуть, що протокол HTTP використовується як транспорт.

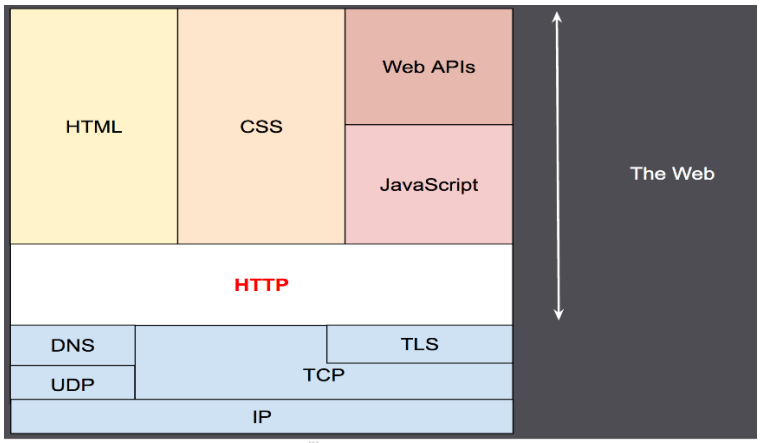


Рис. 1.4. Складові HTTP протоколу

Приклад взаємодії клієнта із веб-сервером і його компонентами показано на рисунку 1.5..

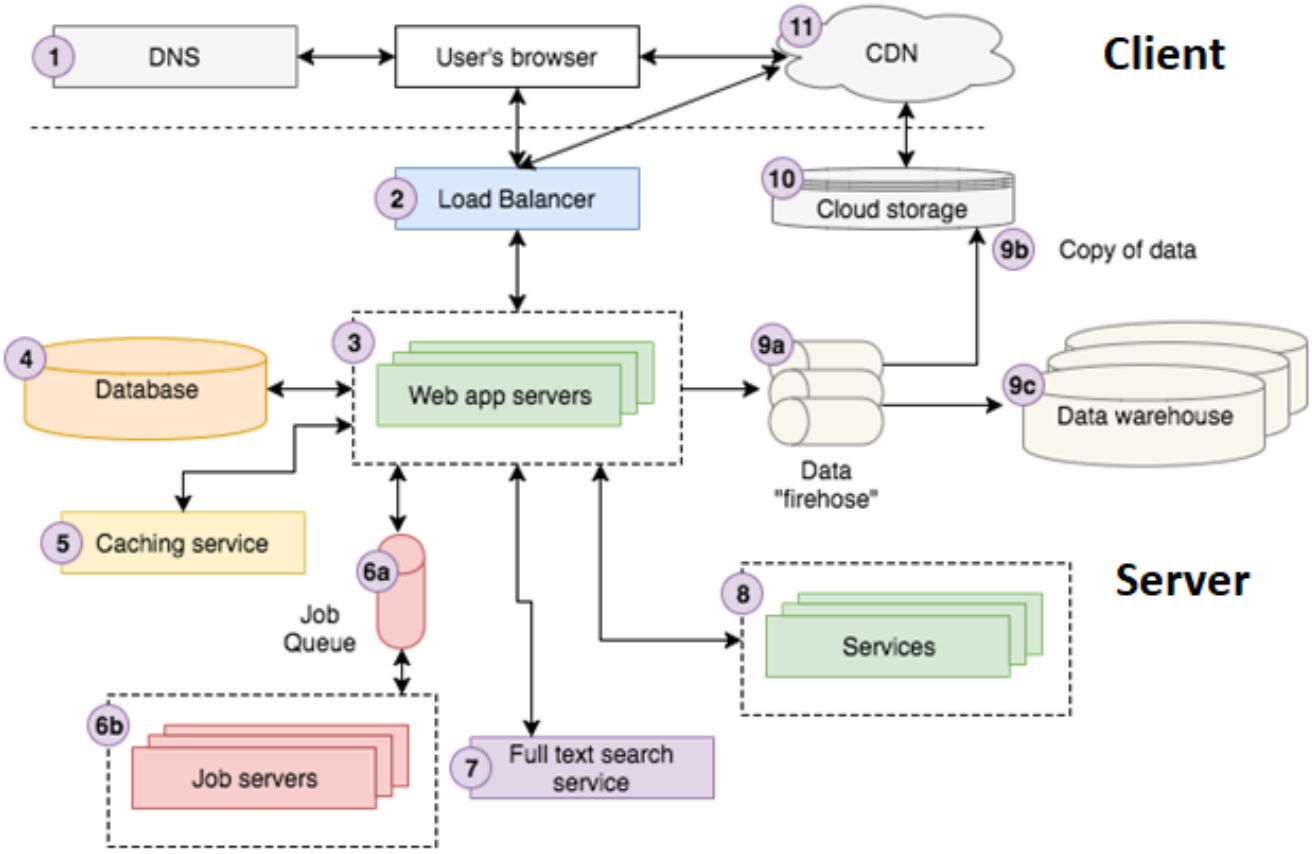


Рис. 1.5. Взаємодія клієнта із веб-сервером і його компонентами

Для роботи необхідно розглянути поетапно, як клієнт взаємодіє із веб-сервером.

Під час роботи з браузером, користувач виконує пошук інформації. В той момент, коли користувач клікає на посилання, веб-браузер направляє його на сторінку веб-сайту. В цей момент, браузер надсилає запит на DNS-сервер (**1**), для того щоб дізнатися, як отримати зв'язок веб-сайтом, тобто йому потрібно відшукати ір адресу веб-сервера, за допомогою використання доменного ім'ені. І лише після цих кроків надсилає запит.

Сам запит надсилається до балансера навантаження (*load balancer*, *LB*) (**2**), що випадково обирає один із декількох веб-серверів (**3**). Обирає один із тих, що задіяні для перетворення запитів.

Далі веб-сервер збирає інформацію у базі даних (**4**) після чого отримує деякі дані із служби кешування (*caching service*) (**5**).

Наступним кроком веб-сервер надсилає задачу (*job*) до черги задач (**6a**), яку *job servers* (**6b**) мають опрацювати асинхронно. Як приклад, це може бути відправка email листа або обробка відео-файлу, що може зайняти деякий час.

Після цього, веб-сервер надсилає запит до *full text search service* (**7**), для того щоб швидко відшукати необхідні дані із великого обсягу інформації.

Після цього, веб-сервер встановлює зв'язок із допоміжними сервісами (**8**), що потрібні для того щоб надавати потрібну інформацію.

Дані від клієнта надсилаються у шину даних (**9a**), яка забезпечує потоковий інтерфейс для прийому та перетворення інформації. Певна частина опрацьованих, агрегованих даних (**9b**) надсилається у хмарне сховище (**10**) для зберігання, інша частина – у локальне (**9c**).

Після цього, сервер надсилає відповідь у вигляді HTML і відправляє її назад у браузер користувача, за допомогою балансер навантаження (**2**).

HTML-сторінка зберігає JavaScript код та CSS, які підгружаються із CDN сховища (**11**).

Варто розглянути більш детально кожен із компонентів сторінки видної користувачеві.

1. **Система доменних імен (DNS)** – це ієрархічна та децентралізована система імен комп’ютерів, служб чи інших ресурсів, підключених до Інтернету чи приватної мережі.

Це і є «книга контактів» інтернету. DNS-сервер – це спеціалізований комп'ютер, який зберігає IP-адреси сайтів. Останні, у свою чергу, прив'язані до імен сайтів та опрацьовують запити користувача. В інтернеті багато DNS-серверів, вони мають кожного провайдера і обслуговують їх користувачів.

2. **Балансер навантаження** забезпечує ефективний розподіл вхідного мережевого трафіку заза допомогою групу серверів, також відомих як ферма серверів або пул серверів.

Балансування навантаження – це процес розподілу запитів між пулом додаткових серверів. Нашим клієнтам доступна послуга балансування HTTP/HTTPS (Layer 7) та TCP (Layer 4).

Використання балансу має дві основні переваги: масштабованість і надійність. У міру зростання кількості користувачів ви можете легко додавати програмні сервери до свого пулу. І якщо один сервер виходить з ладу, інші забезпечують безперервну роботу програми. До додаткових переваг можна віднести передачу навантаження TLS із серверів додатків на плаваючий IP без балансувальника та домену L2.

Ви створюєте приклад балансувальника навантаження на панелі керування. У процесі створення балансувальника навантаження вам надається його загальнодоступна IP-адреса. Потім ви використовуєте IP-адресу балансувальника навантаження, щоб вказати, чи є адреса вашого сервера додатків. Балансувальник навантаження потім розподіляє вхідні запити по пулу ваших реальних серверів програмного забезпечення.

3. **Веб-сервери** виконують головну логіку бізнесу, яка обробляє запит користувача та пересилає HTML назад у браузер користувача.

Поняття «веб-сервер» може стосуватися як апаратної начинки, так і програмного забезпечення. Або навіть до обох частин, що працюють разом.

З точки зору ПЗ, веб-сервер включає кілька компонентів, які контролюють доступ веб-користувачів до розміщених на сервері файлів, як мінімум - це HTTP-сервер. HTTP-сервер - це частина ПЗ, яка розуміє URL-адреси (веб-адреси) і HTTP (протокол, який ваш браузер використовує для перегляду веб-сторінок).

На самому базовому рівні, коли браузер потрібен файл, розміщений на веб-сервері, браузер запитує його за допомогою HTTP-протокол. Коли запит досягає потрібного веб-сервера ("залізо"), сервер HTTP (ПЗ) приймає запит, знаходить запитуваний документ (якщо ні, повідомляє про помилку 404) і відправляє назад, також за допомогою HTTP.

4. **Бази даних** ініціалізують структури даних, вставки нових даних, пошуку даних, оновлення або видалення існуючих даних, проведення обчислень над даними та інше.

5. **Служба кешування** надає ключ-значення, що дозволяє зберігати та шукати інформацію із витратами часу близько *O(1).* Якщо говорити про сферу обчислювальної обробки даних, то так кеш – це високошвидкісний рівень зберігання, у якому необхідний набір даних, зазвичай, тимчасового характеру. Доступ до даних цьому рівні здійснюється набагато швидше, ніж до основного місця їх зберігання. За допомогою кешування стає можливим ефективне повторне використання раніше отриманих даних.

6. У **черзі задач** зберігається список задач, які необхідно виконати асинхронно.

Кожного разу, коли додатку необхідна до виконання робота за певним регулярним графіком, або як це визначено діями користувача, сервер додає відповідну задачу до черги.

«Сервери роботи» обробляють завдання, опитуючи чергу, щоб визначити, чи є задача, яку потрібно зробити.

7. **Сервіс повнотекстового пошуку** використовує індексування даних для швидкого пошуку інформації по ключовим словам.

Повнотекстовий пошук – це пошук слів чи фраз у текстових даних. Зазвичай такий вид пошуку використовується для пошуку тексту у великому обсязі даних, наприклад, таблиця з мільйоном і більше рядків, тому що він значно швидший за звичайний пошук, який можна здійснити, використовуючи конструкцію LIKE.

Повнотекстовий пошук має на увазі створення спеціального індексу (він відрізняється від звичайних індексів) текстових даних, який є певним словником слів, які зустрічаються в цих даних.

За допомогою повнотекстового пошуку можна реалізувати свого роду пошукову систему документів (тобто рядків), за словами чи фразами у базі даних свого підприємства. Оскільки крім своєї швидкої роботи він має ще й можливість ранжувати знайдені документи, тобто. виставляти ранг кожному знайденому рядку, тобто, можна знайти найрелевантніші записи, тобто. найвідповідніші під Ваш запит.

8. **Допоміжні сервіси (служби)** надають потрібну інформацію за запитом від сервера. Наприклад, служба облікових записів зберігає дані користувачів, що дозволяє отримувати усю необхідну інформацію про клієнтів додатку.

9. **Конвеєр даних** потрібен для того, щоб забезпечити можливість збирання, перетворення, фільтрування, зберігання та аналізу даних.

10. **Хмарне сховище даних** – це простий і масштабований спосіб зберігання, доступу та обміну даними за допомогою мережу Інтернет. Для доступу до даних використовується REST API.

Хмарне сховище — це служба, яка дозволяє зберігати дані шляхом їх передачі за допомогою Інтернет або іншу мережу в систему зберігання, що обслуговується третьою стороною. Існують сотні різних хмарних систем зберігання - від особистих сховищ, що містять та/або архівуючі повідомлення електронної пошти, відео та інші особисті файли користувачів, до корпоративних хмарних сховищ, які компанії можуть використовувати як комерційне рішення для віддаленої архівації, що дозволяє безпечно передавати та зберігати файли даних, а також використовувати їх із різних прихильностей.

Хмарні сховища зазвичай масштабуються відповідно до потреб зберігання даних користувача або організації та доступні з будь-якого розташування та пристрою, не вимагаючи використання програми. Компанії можуть вибирати одну з трьох основних моделей: загальнодоступна хмара - служба зберігання, що підходить для неструктурованих даних, приватна хмара - служба зберігання, яка знаходиться в захищеному розташуванні за корпоративним брандмауером для більшого контролю над даними, гібридна хмара - служба зберігання, що поєднує приватну та загальнодоступну хмарні служби підвищення гнучкості.

11. **CDN** (*Content Delivery Network, Мережа Доставки Контенту*) – технологія, що забезпечує можливість доступу до даних, таких як статичний HTML, CSS, Javascript і зображення в Інтернеті, набагато швидше, ніж їх обслуговування з одного сервера-джерела.

Швидкість завантаження контенту – показник, який впливає на лояльність користувачів та позицію сайту у пошукових системах. Згідно з інформацією Google, повільна швидкість завантаження збільшує кількість відмов (доглядів користувачів). Так, якщо сторінка завантажується 6 секунд, вона сягає 106%.

Щоб сайти з великою кількістю даних відкривалися швидше, використовують технологію CDN. У статті розповімо, у яких ситуаціях потрібен CDN-хостинг та які проблеми він допомагає вирішити.

DN-хостинг (Content Delivery Network) додає в це просте рівняння ще один компонент - сервери, на яких кешується частина контенту або сторінка повністю. Вони знаходяться між сервером та кінцевим користувачем, зберігають інформацію різних сайтів для швидкого завантаження та передають її один одному.

CDN-хостинг надають провайдери. Вони розміщують мережу взаємопов'язаних CDN-серверів, що кеширують, у різних точках світу. За рахунок цього відстань між клієнтами та основним сервером не впливає на швидкість передачі даних. Сайти, які використовують CDN-хостинг, завантажуються швидше.

1.3. Шаблони створення веб-додатків

Шаблони проектування веб-додатків, подібні за принципами шаблонів проектування веб-сайтів та програмного забезпечення, пропонують безліч ефективних рішень. У книзі наведені шаблони проектування веб-застосунків виходячи не тільки з проблем взаємодії користувачів, але й розглядаючи їх ефективність та показуючи, як вони повинні застосовуватися.

З книги ви зможете дізнатися, як швидше та якісніше проектувати відмінні інтерфейси, як застосовувати на практиці та покращувати надалі численні готові рішення. Ви зможете досягти небувалих висот у нелегкій справі веб-програмування.

Монолітний застосунок повністю замкнутий в контексті поведінки. Під час роботи застосунок може взаємодіяти з іншими службами або сховищами даних, проте уся поведінка реалізується у власному, одиничному процесі, а застосунок зазвичай розгортається як один елемент. Для горизонтального масштабування такий застосунок зазвичай цілком дублюється на декількох серверах або віртуальних машинах.

Термін «моноліт» означає – «усе складене в одне ціле», у якому різні компоненти об'єднуються в єдину програму з однієї платформи.

На рис. 1.6 зображено приклад монолітної архітектури: застосунок складається із декількох сервісів, які взаємодіють із БД. Клієнти, використовуючи браузер або мобільні пристрої, взаємодіють із веб-сервером.

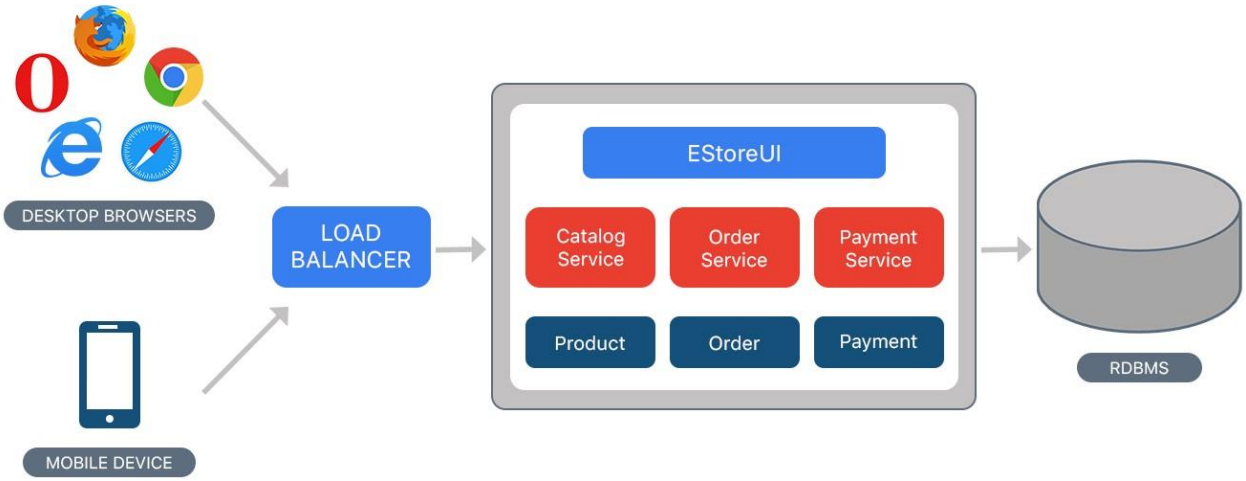


Рис.1.6. Приклад монолітної архітектури

Переваги:

- мала кількість ролей при розробці та експлуатації;
- простота у розгортанні – потрібно лише скопіювати «упаковану» програму на веб-сервер;
- легко забезпечувати транзакційну цілісність даних;
- простота у розробці – новий проект набагато простіше розробляти опираючись на монолітну архітектуру;
- легко масштабувати горизонтально, запустивши декілька копій додатку і балансер навантаження.

Недоліки:

- технічне обслуговування – якщо застосунок занадто великий і складний, обслуговування може ускладнитись;
- єдиний технологічний стек технологій для усіх компонентів додатка;
- великі часові і людські витрати при модифікації і рефакторингу додатку із складною бізнес-логікою;
- перезапуск усього додатку при появі нової функціональності;
- великий «розмір» програми може сповільнити час запуску і розгортання;

- монолітні програми можуть бути складними для масштабування, коли модулі мають різні вимоги до апаратних ресурсів;
- ненадійність – помилка в будь-якому модулі (наприклад, витік пам'яті) може потенційно привести до збою усього додатку.

Щоб зробити правильний вибір, потрібно передусім замислитися, навіщо все це потрібно, необхідно конкретно розуміти бізнес-завдання. Адже рішення на користь мікросервісів є дуже відповідальним кроком. Справа в тому, що в мікросервісах все значно складніше, ніж зазвичай, тобто ми можемо зіткнутися з такою ситуацією:

Ось не можна просто взяти і розпиляти все на якісь шматки і сказати, що це тепер мікросервіси. Інакше вам доведеться дуже несолодко.

Мікросервісна архітектура – це тип розробки додатків, у якому великий застосунок складається із набору модульних служб, сервісів, мало пов’язаних модулів / компонентів(рис. 1.7).

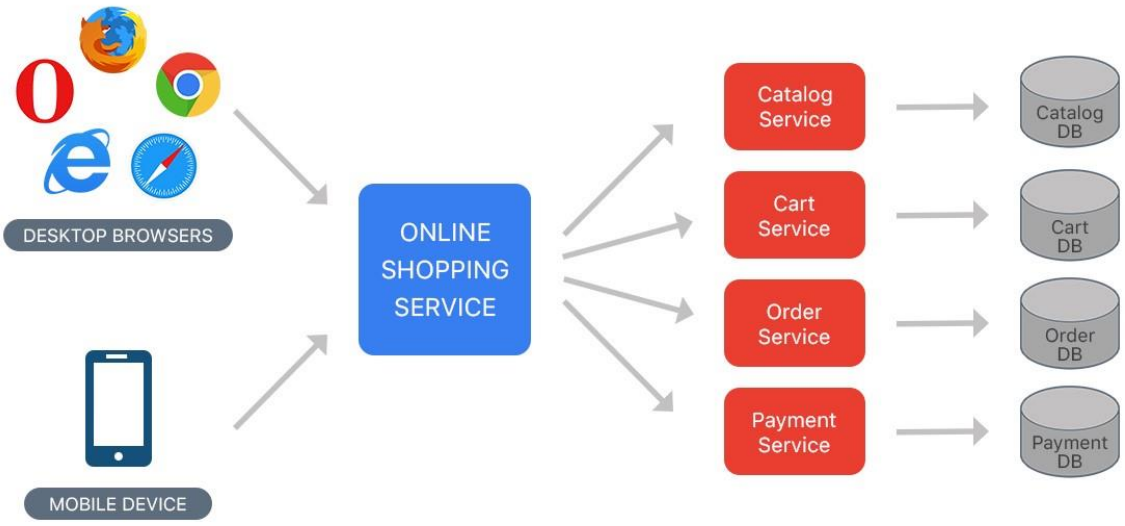


Рис. 1.7. Приклад мікросервісної архітектури

Замість обміну інформацією з єдиною базою даних, як у монолітній архітектурі, кожен мікросервіс має свою власну базу даних. Єдина база даних для служби забезпечує погане з’єднання між програмними компонентами. Крім того, сервіс може використовувати той тип бази даних, який найбільше відповідає його потребам (наприклад, SQL або NoSql).

Безсерверна архітектура – це спосіб створювати та запускати програми та служби без необхідності керувати інфраструктурою. Програма все ще працює на серверах, але цими серверами керує стороння служба. Це позбавляє від необхідності виділення, масштабування та обслуговування серверів для запуску програм, баз даних і систем зберігання даних.

Безсерверні обчислення — це модель виконання комп'ютерного коду, в якій розробники звільняються від зберігання компонентів системної інфраструктури. Ця тенденція також відома як функція обслуговування (FaaS), коли постачальник хмарних послуг відповідає за активацію та деактивацію функції контейнерної платформи, перевірку безпеки інфраструктури, зниження витрат на обслуговування та покращення масштабованості.

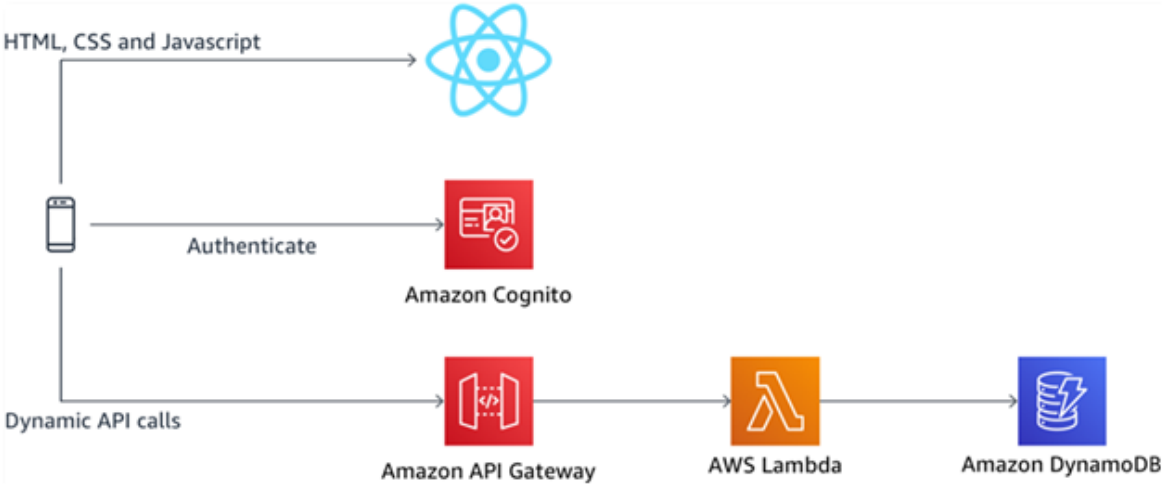


Рис. 1.8. Приклад безсерверної архітектури

У безсерверних додатках спеціальні програмні компоненти активізуються тільки тоді, коли надходить запит. Після перетворення запиту компонент «засинає». Дані компоненти часто розташовуються у керованому середовищі, яке відповідає за масштабованість і стежить за життєвим циклом.

Як висновок до вище розглянутих технологій створимо порівняльну характеристику за визначеними критеріями.

Теорема CAP (*теорема Брюєра*) – евристичне твердження про те, що в будь-який реалізації розподілених обчислень можливо забезпечити не більше двох з трьох наступних властивостей (табл. 1.1):

- узгодженість даних (*consistency*) – у всіх вузлах розподіленої системи в один момент часу усі дані однакові і не суперечать один одному;
- доступність (*availability*) – на будь-який запит до кожного вузла розподіленої системи надходить очікувана відповідь, але немає гарантії, що всі відповіді в один момент часу будуть співпадати;
- стійкість до розподілу (*partition tolerance*) – поділ вузлів розподіленої системи на кілька ізольованих сегментів не призводить до неочікуваної роботі системи.

Таблиця 1.1

Характеристика CAP

Архітектура	Узгодженість	Доступність	Стійкість до розподілу
Моноліт	+	−	−
Мікросервіс	−	+	+
Безсерверна	−	+	+

Вертикальне масштабування (*scaling up*) – збільшення кількості доступних для ПО ресурсів за рахунок збільшення потужності з серверів. У даному випадку перевагу віддають потужнішому апаратному забезпеченню.

Горизонтальне масштабування (*scaling out*) – збільшення кількості серверів, об'єднаних в кластер серверів або процесів за рахунок ОС.

У табл. 1.2 показано допустимість типу масштабування у залежності від виду архітектури.

Таблиця 1.2

Характеристика по типу масштабування

Архітектура	Вертикальне	Горизонтальне

Моноліт	+	−
Мікросервіс	+	+
Безсерверна	+	+

У табл. 1.3 показана орієнтована оцінка вимог апаратних ресурсів інфраструктури до архітектурного рішення.

Позначення: S – small (*невисокі вимоги*), M – middle (*середні вимоги*), L – large (*значні вимоги*).

Типи ресурсів:

- Пам'ять (*RAM*);
- Ресурси процесора (*CPU*);
- Місце на диску (*HDD / SSD*);
- Комунікація (*LAN*);

Таблиця 1.3

Характеристика по типам ресурсів

Архітектура	RAM	CPU	HDD/SSD	LAN
Моноліт	**M**	**M**	**M**	**S**
Мікросервіс	**L**	**L**	**S**	**L**
Безсерверна	**L**	**L**	**S**	**L**

Висновок

Високі темпи змін і висока складність можуть бути факторами, які змушують вибрати архітектуру мікросервіса. Можливість безперервно розробляти та розгортати великі, складні програми буде тільки сприяти активному росту бізнесу.

І навпаки, якщо немає конкретної предметної області, було б непогано почати з моноліту, але варто модулювати послуги. Якщо прийнято рішення розділити моноліт на кілька мікросервісів, це полегшить роботу.

Завдяки безсерверній архітектурі немає необхідності виділяти ресурси, масштабувати та підтримувати сервери для виконання програм, тому ви можете зосередити всю свою увагу на бізнес-вимогах та розробці додатків.

РОЗДІЛ 2
РОЗГЛЯД МІКРОСЕРВІСНОЇ БУДОВИ

2.1. Властивості і засади архітектури мікросервісів

Що таке мікросервіси?

Це архітектурний шаблон, в якому послуги:

- маленькі (small),
- сфокусовані (focused),
- слабопов'язані (loosely coupled),
- високоузгоджені (highly cohesive).

Тепер розберемо ці поняття окремо.

Що означає «маленький» сервіс? Це означає, що сервіс у мікросервісній архітектурі не може розроблятися більш ніж однією командою. Зазвичай одна команда розробляє десь 5 – 6 сервісів. При цьому кожен сервіс вирішує одне бізнес-завдання, і його здатна зрозуміти одна людина. Якщо ж не здатний, сервіс настав час пиляти. Тому що, якщо одна людина здатна підтримувати всю бізнес-логіку одного сервісу, вона побудує справді ефективне рішення. Адже буває так, що часто люди, приймаючи рішення в процесі написання коду, просто не розуміють, що саме роблять — не знають, як поводиться система в цілому. А якщо сервіс невеликий, все набагато простіше. Цей підхід, до речі, ми можемо застосовувати окремо, навіть не дотримуючись мікросервісної архітектури загалом.

Що означає «сфокусований» сервіс? Це означає, що сервіс вирішує лише одне бізнес-завдання і вирішує його добре. Такий сервіс має сенс у відриві від інших сервісів. Іншими словами, ви його можете викласти в інтернет, дописавши security-обгортку, і він буде приносити людям користь.

Що таке «слабозв'язаний» сервіс? Це коли зміна одного сервісу не потребує змін в іншому. Ви пов'язані за допомогою інтерфейсів, у вас є рішення за допомогою DI та IoC – це зараз стандартна практика, застосовувати яку потрібно обов'язково. Зазвичай розробники знають чому :)

Що таке «високоузгоджений» сервіс? Це означає, що клас або компонент містить усі необхідні спосіби вирішення поставленого завдання. Однак, тут часто виникає питання, чим висока узгодженість (high cohesion) відрізняється від SRP? Допустимо, у нас є клас, який відповідає за керування кухнею. У випадку SRP такий клас працює тільки з кухнею і більше ні з чим, але при цьому він може містити не всі спосіби управління кухнею. У випадку ж високої узгодженості, всі спосіби управління кухнею містяться тільки в цьому класі, і більше ніде. Це важлива відмінність.

Характеристики мікросервісів

- Поділ на компоненти (сервіси).
- Угруповання з бізнес-завдань.
- Сервіси мають бізнес-сенс.
- Розумні сервіси та прості комунікації.
- Децентралізоване керування.
- Децентралізоване керування даними.
- Автоматизація розгортання та моніторингу.
- Design for failure (Chaos Monkey).

Мікросервісний підхід має декілька моделей дизайну архітектури, їх можна розділити на п’ять моделей (рис. 2.1).

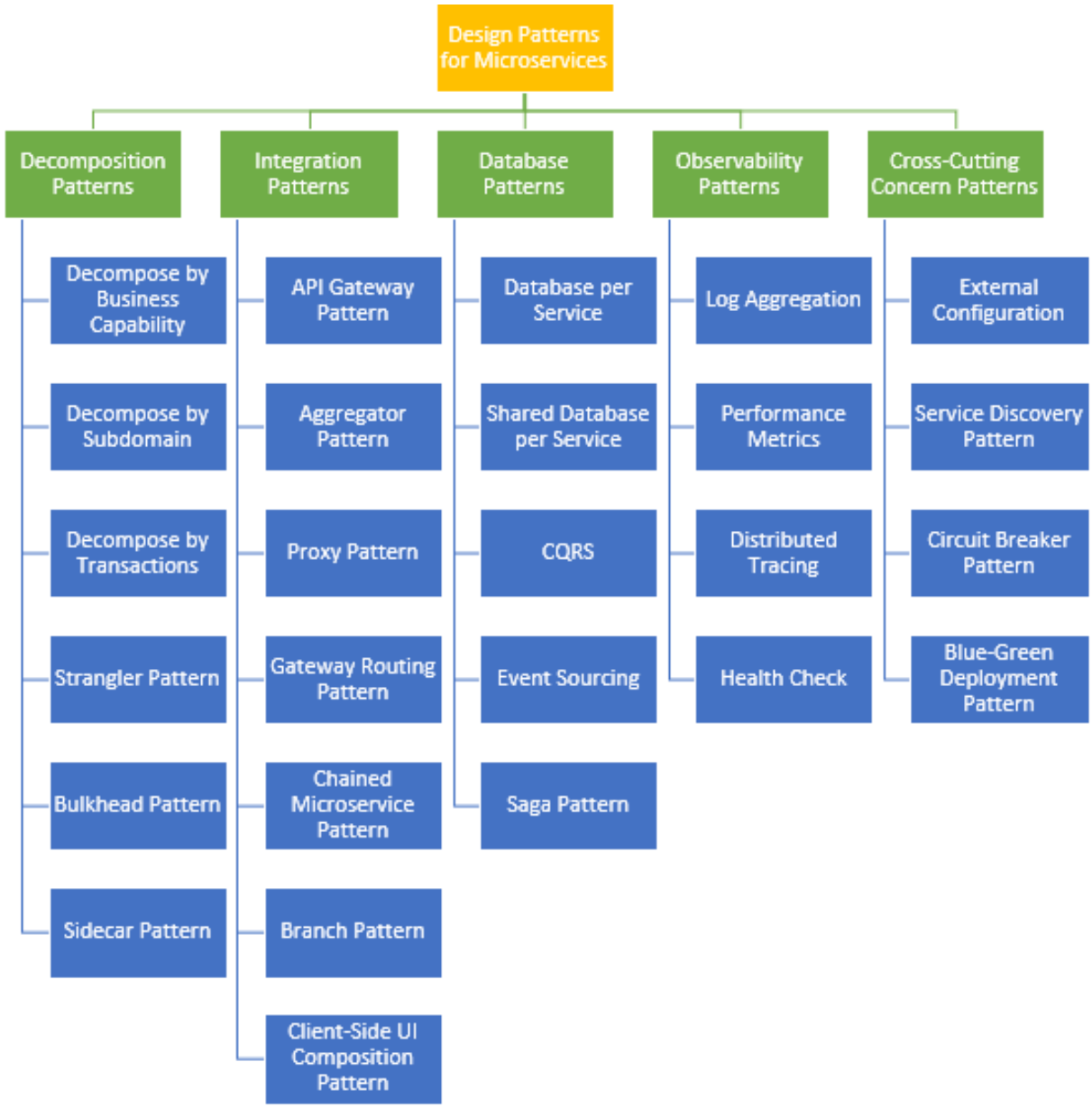


Рис. 2.1. Патерни мікросервісної архітектури

Бізнес-вимога – те що виконує організація, а це у свою чергу сприяє досягненню її бізнес-цілей. Для підвищення ефективності розробки ви також часто змушені ділити по цих шарах і команди: є команда, яка займається UI, є команда, яка займається ядром, і є команда, яка розуміється на БД.

Якщо ж ви переходите до мікросервісної архітектури, сервіси та команди поділяються на бізнес-завдання.

Наприклад, може бути група, яка займається управлінням замовленнями, вона може обробляти транзакції, робити за ними звіти і т. д. Така група займатиметься і відповідними БД, і відповідною логікою, і, можливо, навіть UI. Втім, у моєму досвіді UI розпилювати поки що не вдавалося - його доводилося

залишати монолітним. Можливо, нам вдасться зробити це в майбутньому, тоді обов'язково розкажіть іншим, як ви цього досягли. Як би там не було, навіть якщо UI залишається монолітним, все одно набагато краще, коли решта розбита на компоненти. Тим не менш, повторюся, дуже важливо розуміти, НАВІЩО ви це робите - інакше одного разу доведеться все переробляти назад.

Декомпозуючи систему таким чином, застосовують принцип єдиної відповідальності. Система розподіляється на основі бізнес-вимог та визначаються послуги, що задовольняють вимоги бізнесу (рис. 2.2).

Набір вимог для даного бізнесу залежить від типу бізнесу. Наприклад, робота страхової компанії зазвичай включає у себе продажі, маркетинг, обробка пропозицій і претензій, виставлення рахунків, відповідність. Отже система, декомпозується на сервіси: сервіс продажу, сервіс маркетингу, сервіс рахунків і тд.

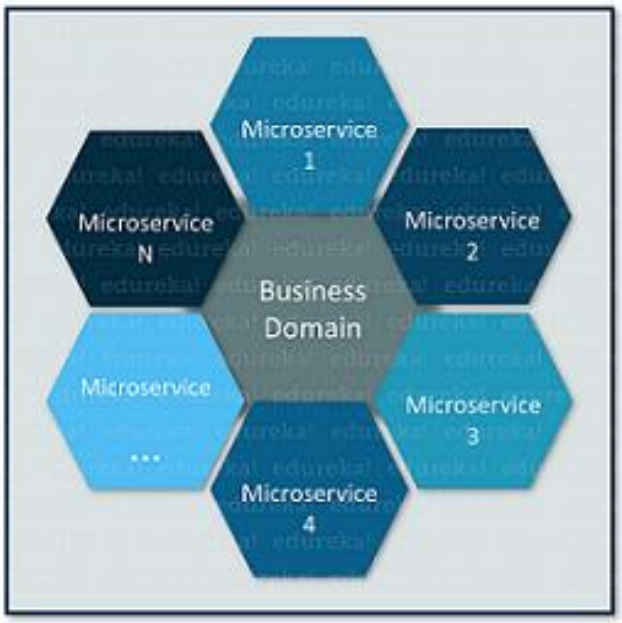


Рис. 2.2. Декомпозиція на основі бізнес-вимог

Предметно-орієнтоване проектування (*domain-driven design, DDD*) – підхід до проектування програмного забезпечення, заснований на моделюванні предметної області.

Але повна заміна складної системи часто пов'язана з величезними труднощами. У багатьох випадках краще переходити на нову систему поступово, зберігаючи стару систему для перетворення тих можливостей, які ще не реалізовані в новій (рис. 2.3).

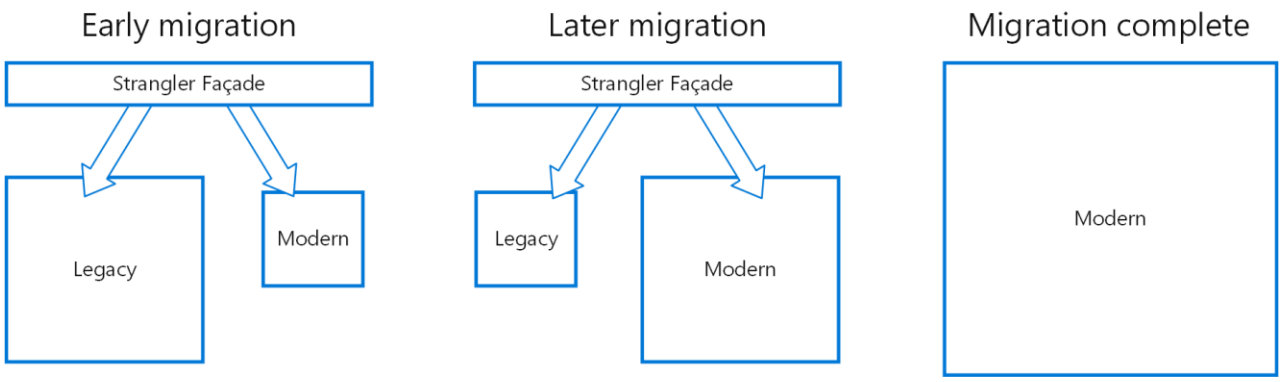


Рис. 2.3. Поетапний перехід на новий компонент

Цей шаблон використовується при переході від монолітної системи до мікросервісу.

Шаблон Bulkhead — це тип архітектури програмного забезпечення, стійкої до збоїв. Відповідно до цієї архітектури елементи програми поділяються на групи, якщо одні не працюють, інші продовжуватимуть працювати.

2.4 показує клієнти, які звертаються до служби. Кожному клієнту надається окрема копія цієї послуги. Клієнт номер 1 звертається до пошкодженої служби. Оскільки кожен екземпляр служби ізольований від інших, інші клієнти продовжують спокійно працювати з сервісом.

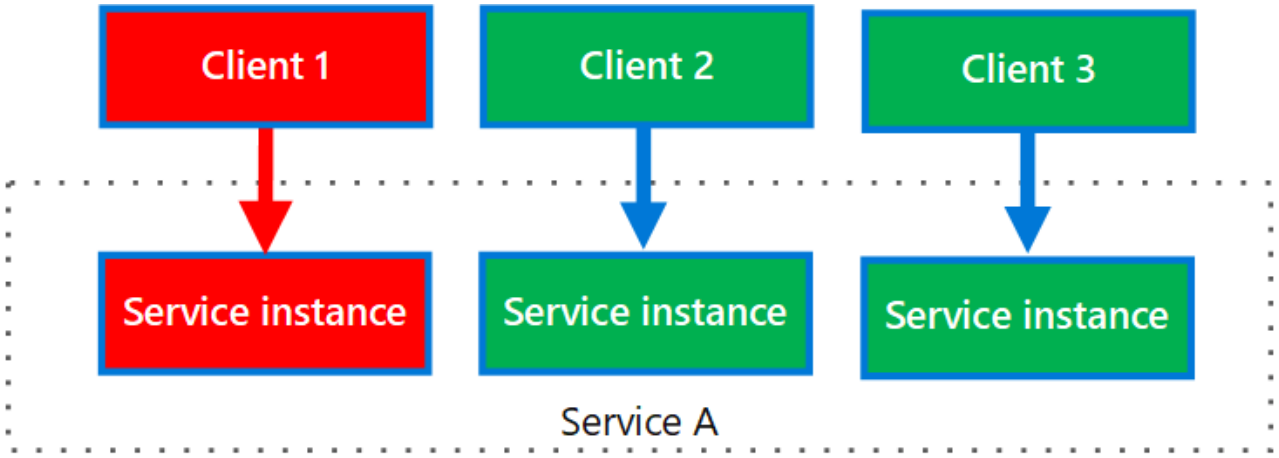


Рис. 2.4. Ізоляція відмов

Згідно із шаблоном, рішенням є розділення сервісу по окремим, власним процесам або контейнерам, надаючи однорідний інтерфейс для служб на платформі для різних мов (рис. 2.5).

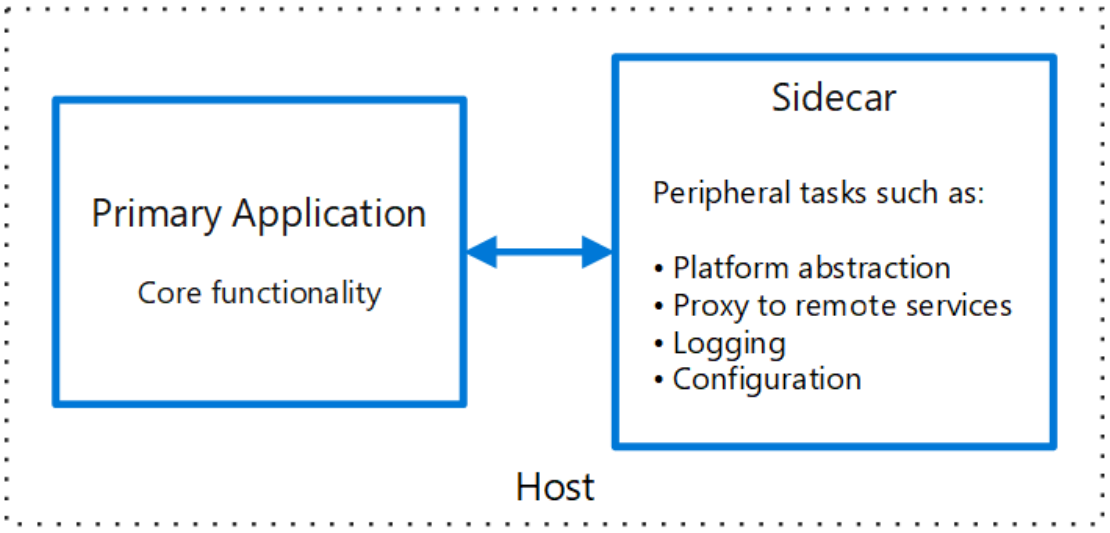


Рис. 2.5. Розділення служб по різним процесам

2.2. Міжпроцесорна співдія

Візьмемо приклад простої програми. Картинки, наведені нижче, я взяв із блога Кріса Річардсона на NGINX – там детально розповідається, що таке мікросервіси.

Отже, припустимо, у нас є якийсь клієнт (необов'язково навіть UI-ний), який, щоб надати комусь потрібні дані, взаємодіє із сукупністю інших сервісів.

Начебто все просто - клієнт може отримати доступ до всіх цих послуг. Але насправді це призводить до дуже великої конфігурації клієнтів. Тому він має дуже простий шаблон API шлюзу

API шлюзу – це перше, що потрібно враховувати при налаштуванні архітектури мікросервісів. Якщо у вас є кілька сервісів у бекенді, поставте перед ними найпростіший сервіс, завдання — зібрати ділові дзвінки до цільових служб. Тоді ви зможете намалювати карту транспорту (транспорт буде будь-яким, а не REST API, як показано на малюнку). API шлюзу надає інформацію у формі, необхідному для цього типу користувачів. Наприклад, якщо у вас є веб-додаток і мобільний додаток, у вас буде два API шлюзу, які збирають інформацію від служб і надають їх дещо іншим способом. API шлюзу ніколи не повинен містити серйозну бізнес-логіку, інакше він буде повторюватися скрізь, і його буде важко підтримувати. API шлюзу передає лише дані і все.

Програмне забезпечення на основі мікросервісів — це розподілена система, яка іноді працює на кількох процесах чи службі, а не навіть на кількох серверах чи хостах. Як правило, кожен екземпляр служби є процесом.

Таким чином, служби повинні бути з’єднані між собою через протокол зв’язку в процесі, такий як HTTP, AMQP, або двійковий протокол, такий як TCP, залежно від характеру кожної служби.

Залежно від сценарію та цілей клієнти та служби можуть взаємодіяти за допомогою різних типів спілкування. Ці види комунікації можна розділити на два напрямки.

Перша група визначає, чи є протокол синхронним чи асинхронним (Малюнок 2.6):

• синхронний протокол. HTTP – це синхронний протокол. Клієнт відправляє запит і чекає відповіді від сервісу. Тут важливо, щоб протокол (HTTP

/ HTTPS) був синхронним і щоб код клієнта міг продовжити виконання завдання тільки після отримання відповіді від HTTP-сервера;

• асинхронний протокол. Інші протоколи, такі як AMQP (протокол, який підтримується багатьма операційними системами та хмарними середовищами), використовують асинхронний обмін повідомленнями. Людина, яка надсилає код або повідомлення клієнта, зазвичай не очікує відповіді. RabbitMQ надсилає лише одне повідомлення, як під час надсилання повідомлення черзі чи іншому посереднику повідомлень.

Друга група визначає, чи є запит від одного або кількох покупців:

• один одержувач - кожен запит повинен оброблятися тільки одним отримувачем або службою;

• Кілька одержувачів – кожен запит може оброблятися різною кількістю одержувачів – від одного до кількох. Цей тип взаємодії має бути асинхронним. Наприклад, механізм підписки на публікацію, який використовується в таких шаблонах, як архітектура на основі подій. Він покладається на інтерфейс шини подій або посередника повідомлень, коли події оновлюють дані в багатьох мікросервісах. Зазвичай це робиться за допомогою службової шини або подібного об’єкта, наприклад, службової шини Azure з темами та підписками.

Додатки на основі мікросервісів часто використовують комбінацію цих стилів взаємодії. Найпоширенішим типом є взаємодія з одним одержувачем за допомогою синхронного протоколу, такого як HTTP або HTTPS, під час виклику звичайної служби веб-API HTTP. Протоколи повідомлень зазвичай

використовуються для асинхронної взаємодії між мікросервісами.

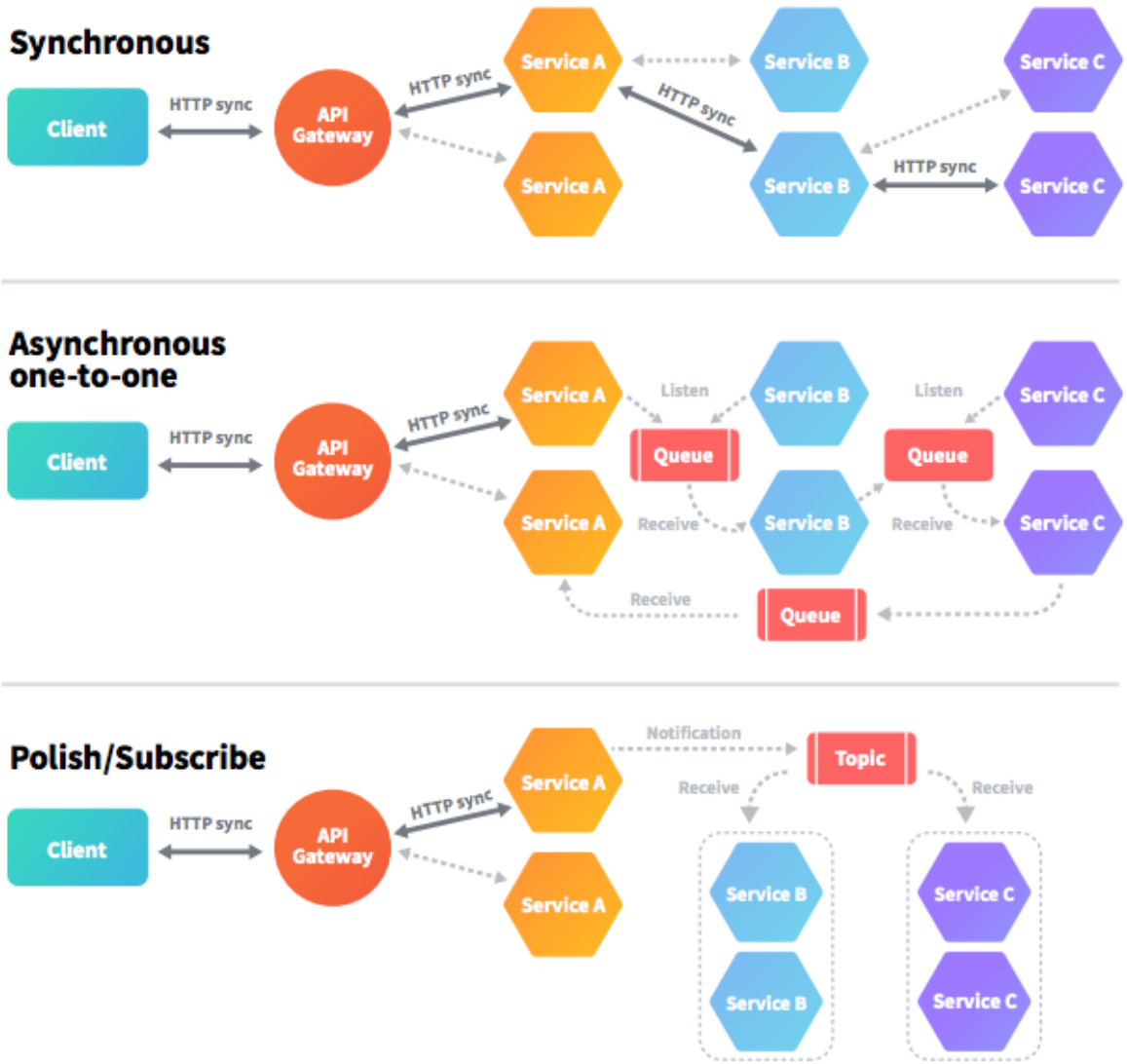


Рис. 2.6. Типи взаємодій мікросервісів

Service Discovery створено для того, щоб з мінімальними витратами можна підключити нову програму у вже існуюче наше оточення. Використовуючи Service Discovery, ми можемо максимально розділити або контейнер як докера, або віртуальний сервіс від оточення, в якому він запущений.

Як це виглядає? На класичному прикладі в Інтернеті - це фронтенд, який приймає запит користувача. Далі виконує маршрутизацію його на backend. На даному прикладі це load-balancer балансує на два backend.

Service Discovery дає змогу виявити збій, виявити відмови.

Service Discovery зі свого боку опитує програму на факт доступності.

Або ж використовується сторонній скрипт або програма, яка перевіряє нашу програму на доступність і повідомляє Service Discovery, що все добре і можна працювати, або, навпаки, що все погано і необхідно цей екземпляр програми виключити з балансування.

На рис. 2.7 зображений реєстр сервісів який виконує функції виявлення сервісів. Запит від клієнта проходить за допомогою балансер навантаження, який зв'язується із реєстром сервісів, щоб отримати розташування (ip та порт) усіх екземплярів додатку, після, направляє запит до потрібного екземпляру.

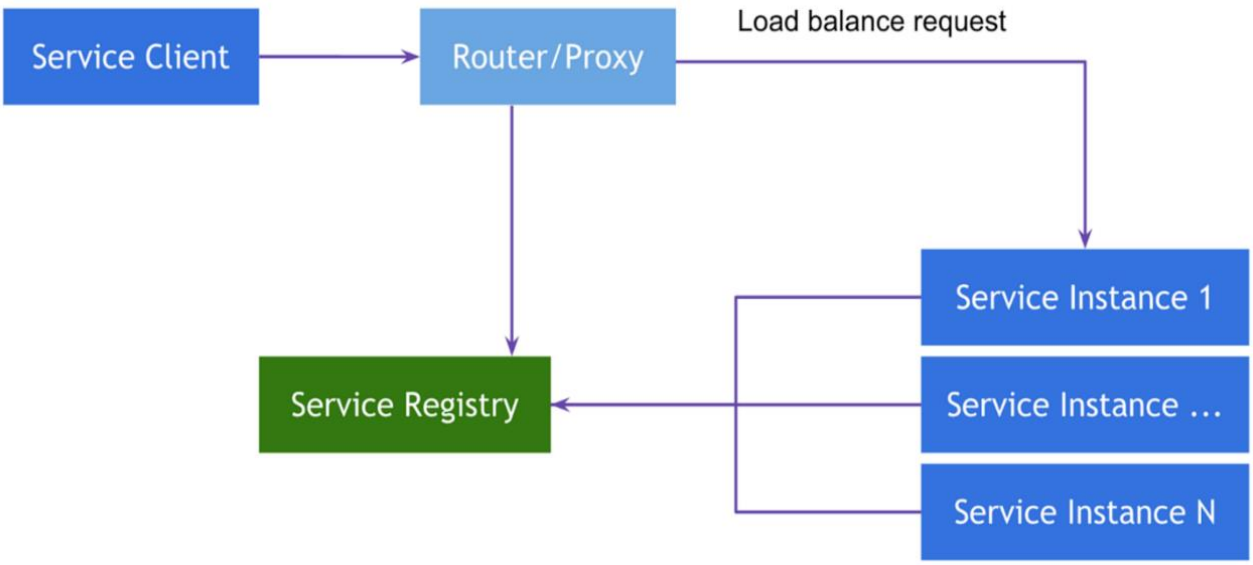


Рис. 2.7. Реєстр сервісів

RabbitMQ – це застосунок для роботи з асинхронними чергами повідомлень (*message-queueing*), ще його називають брокер повідомлень (*message broker*) або менеджер черг (*queue manager*). Тобто, це програмне забезпечення в якому можуть бути створені черги до яких можуть підключатись різні додатки і передавати та отримувати повідомлення.

На рис. 2.8 зображена схема взаємодії видавця (*producer*) і підписника (*consumer*). Видавець передає подію брокеру повідомлень, який публікує її у

чергу повідомлень. Таким чином, підписники, які підписались на деяку чергу, отримають своє повідомлення.

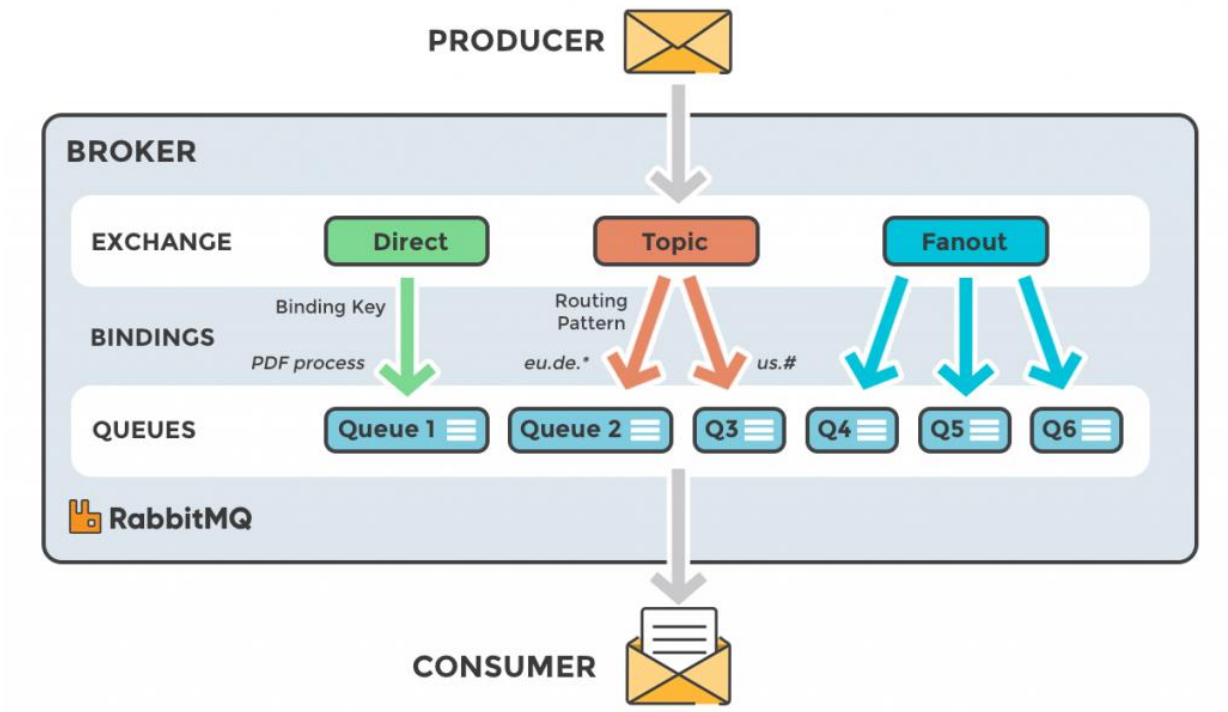


Рис. 2.8. Принцип взаємодії RabbitMQ

RabbitMQ - це платформа, яка дозволяє обмінюватися повідомленнями. Що може бути у повідомленні – вирішувати тільки тобі. Обмінюватись можна як на одному сервері, так і з одного на інший. Це відмінний спосіб масштабування, тому що з добре налаштованим RabbitMQ ми можемо просто підключати нові сервери, налаштувавши на них потрібне ПЗ і прописавши конфігурацію, а RabbitMQ сам делегуватиме роботу між усіма серверами. На офіційному сайті є мануали за основними способами використання кролика – ознайомся. Я не описуватиму основні сутності, які є в RabbitMQ і на основі якого будується протокол AMQP — це можна легко знайти в мережі.

В процесі розробки додатків з мікросервісною архітектурою неминуче виникає проблема вибору формату обміну даними між модулями. Вирішення цієї проблеми може мати значний вплив як на роботу програми, так і на

трудомісткість подальшої модернізації. Час відгуку, обсяг переданої інформації по каналу зв'язку, розширюваність і портованість, необхідні ресурси і інші параметри можуть залежати від формату обміну даними.

Найчастіше в співтоваристві розробників, перевагу віддають одному з трьох найбільш використовуваних форматів обміну даними: XML, JSON, YAML.

Крім HTML, картинок та відео на сайті необхідно передавати та відображати різну інформацію.

Зараз я говорю про масиви даних, про складну ієрархічну структуру.

Для передачі як інтеграції, так сайтів використовуються певні формати даних.

JSON і XML використовуються для отримання та надсилання даних з веб-сервера.

JSON (англ. JavaScript Object Notation) - простий формат обміну даними, що базується на мові програмування JavaScript. Використовує текст людини для передачі об'єктів даних.

Приклад синтаксису:

«employees»:[

{"firstName":"Lev", "lastName":"Tolstoy"},

{"firstName": "Anna", "lastName": "Karenina"},

{"firstName":"Aleksey", "lastName":"Vronsky"},

]

Синтаксичні правила JSON

Дані вказуються в парах ім'я/значення, що розділяються двокрапкою «firstName»: «Lev»

Дані розділяються комами "firstName": "Anna", "lastName":

«Karenina»

Фігурні дужки утримують об'єкти «firstName»: «Lev», «lastName»: «Tolstoy»},

Квадратні дужки містять масиви

Переваги JSON

Меньше слів більше діла

XML вимагає відкриття та закриття тегів, а JSON використовує пари ім'я / значення, чітко позначені «{«і»}» для об'єктів, «[«і»]» для масивів, «,» (кома) для розділення пари та «:» (двокрапка) для відокремлення імені від значення.

Розмір має значення

При однаковому обсязі інформації JSON майже завжди значно менше, що призводить до більш швидкої передачі та обробки.

Близькість до javascript

JSON є підмножиною JavaScript, тому код для його аналізу та пакування цілком природно вписується в код JavaScript.

XML

XML — це мова розмітки, яка визначає набір правил для кодування документів у форматі, який читається людиною та читається машиною. Але що більше інформації (вкладень, коментарів, варіантів тегів тощо.) в xml, то складніше її читати людині.

XML зберігає дані у текстовому форматі. Це забезпечує незалежний від програмного та апаратного забезпечення спосіб зберігання, транспортування та обміну даними. XML також полегшує розширення або оновлення до нових операційних систем, нових програм або нових браузерів без втрати даних.

JSON (*Java Script Object Notation*) – являє собою полегшений формат обміну даними між комп'ютерами. JSON більш компактний, ніж XML, його конструкції легше аналізуються засобами JavaScript, для якого JSON є внутрішнім використовуваним типом даних. Основна сфера застосування JSON – програмування веб-додатків, де він служить альтернативою XML.

YAML – це мова для зберігання інформації у форматі зрозумілій людині. Його назва розшифровується як «Ще одна мова розмітки». Однак, пізніше розшифровку змінили на «YAML не мова розмітки», щоб відрізняти його від справжніх мов розмітки. Мова схожа на XML і JSON, але використовує більш мінімалістичний синтаксис при збереженні аналогічних можливостей. YAML зазвичай застосовують для створення конфігураційних файлів у програмах типу Інфраструктура як код (Iac), або управління контейнерами в роботі DevOps.

Найчастіше за допомогою YAML створюють протоколи автоматизації, які можуть виконувати послідовності команд, записані в YAML-файлі. Це дозволяє вашій системі бути більш незалежною та чуйною без додаткової уваги розробника. У табл. 2.1 зображена порівняльна оцінка форматів даних [11].

Таблиця 2.1

Порівняльна оцінка форматів даних за шкалою зручності 1-5

Критерій	XML	JSON	YAML
Зручність читання	4	5	4
Простота серіалізації	5	5	5
Простота десеріалізації	5	5	5
Простота перевірки вхідних даних	4	4	3
Ефективність стиснення даних	3	5	1

Отже, аналізуючи характеристики кожного із представлених форматів даних, можна зробити висновок, що формат даних JSON, є найбільш оптимальним для використання у міжпроцесерній взаємодії мікросервісів.

2.3. Розкривання додатків

CI (Continuous Integration) – безперервна інтеграція. Під час написання коду розробники постійно вносять зміни, що підвантажуються до репозиторію. Для автоматичного тестування та перевірки використовують спеціалізовані сервіси (наприклад, Github), що створюють скрипти. Ведеться балка, в якій запротоколовані всі модифікації.

CD (Continuous Delivery) – безперервна доставка. Відповідає за автоматичне розгортання збірки у будь-якому оточенні: продакшн, середовище тестування чи розробки. Наприклад, після редагування коду він автоматично вміщується в область тестування.

Спосібологія використовується в компаніях, які під час розробки часто модифікують код, але випускають стабільні релізи. Розробники отримують автоматичний процес тестування та розгортання програми, що дозволяє зосередитись на постійному покращенні ПЗ.

На рис. 2.9 зображена схема безперервної інтеграції: спочатку розробник фіксує свої зміни у систему контролю версій (*git*), завантажує зміни у репозиторій контролю версій (*GitHub*), після чого система CI проводить модульне тестування коду (*unit test*), робить синтаксичний аналіз коду (*code analysis*), вимірює відсоток покриття тестами (*test coverage*).

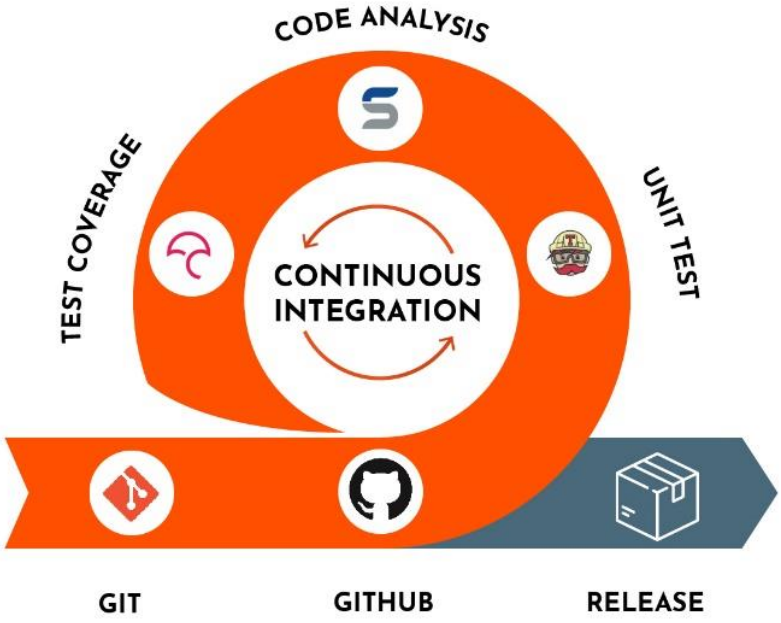


Рис. 2.9. Схема безперервної інтеграції

Збірка за розкладом, зазвичай, проводяться в неробочий час, вночі, та плануються так, щоб до початку чергового робочого дня були готові результати тестування. Для відмінності окремо вводиться система нумерації збірок –кожна збірка нумерується натуральним числом, яке зазвичай збільшується з кожною новою збіркою.

Окремо, вихідні тексти та інші вихідні дані при взятті їх зі сховищ системи контролю версій позначаються номером збірки. Завдяки цьому, точно така ж збірка може бути точно відтворена в майбутньому – досить взяти вихідні дані по потрібної мітці і запустити процес знову. Це дає можливість повторно випускати навіть дуже старі версії програми з невеликими виправленнями.

Популярних системи CI є CircleCI, TeamCity, Jenkins.

CI допомагає розширити персонал та наростити ефективність команд інженерів. Впровадження CI за згаданим сценарієм дозволяє розробникам програмного забезпечення паралельно вести незалежну роботу над функціями. Так вони зможуть самостійно і без затримок поєднати ці функції в кінцевий продукт, коли все буде готове. CI – важлива та надійна технологія, що використовується сучасними, високоефективними організаціями у сфері розробки ПЗ.

CI зазвичай використовується разом із робочим процесом Agile-розробки ПЗ. Організація готує список завдань, що становлять дорожню карту продукту. Ці завдання потім розподіляються між учасниками команди інженерів із метою поставки. За допомогою CI відповідальні розробники можуть виконувати ці завдання з розробки ПЗ незалежно та паралельно з колегами. Після завершення завдання розробник внесе виконану роботу до системи CI для інтеграції з рештою проекту.

Безперервна інтеграція, розгортання та доставка утворюють три стадії автоматизованого конвеєра випуску ПЗ (з урахуванням конвеєра DevOps) Ці три стадії охоплюють розробку програмного забезпечення від ідеї до доставки кінцевому користувачеві. Стадія інтеграції – це перший крок у цьому процесі. Безперервна інтеграція охоплює етап, у якому кілька розробників намагаються поєднати зміни коду з головним репозиторієм коду проекту.

Безперервне постачання є продовженням безперервної інтеграції. На стадії поставки відбувається пакетування робочого продукту доставки кінцевим користувачам. На цьому етапі запускаються автоматизовані інструменти збирання для створення такого продукту. Ця стадія складання вважається «зеленою». Це означає, що продукт має бути готовим до розгортання для користувачів у будь-який момент.

Безперервне розгортання – це остання стадія конвеєра. Стадія розгортання відповідає за автоматичний запуск та розповсюдження робочого продукту серед кінцевих користувачів. На момент розгортання продукт має успішно пройти стадії інтеграції та постачання. Тепер потрібно автоматично розгорнути або розповсюдити продукт. Це буде проведено за допомогою скриптів або інструментів, які автоматично розмістять продукт на загальнодоступних серверах або в іншій системі розповсюдження, як-от магазин додатків.

На рис. 2.10 зображена схема безперервної інтеграції. Етапи розробки (*Develop*) та збірки (*Build*) уже проведені на стадії CI. На етапі CD проводяться автоматизовані тести (*Automated Test*), та розгортання додатку на основному сервері (*Automated Deploy*), після чого вирішується чи запускати даний розгорнутий застосунок у роботу (*Automated/Controller Release*).

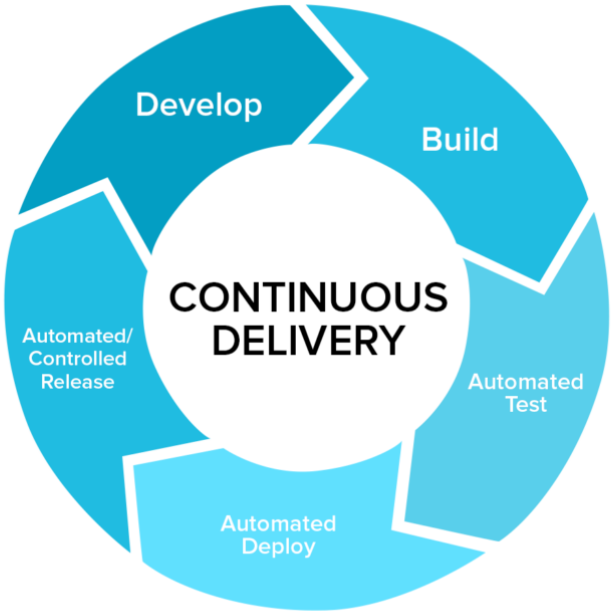


Рис. 2.10. Схема безперервної доставки

Потрібно зыбрати кожен сервіс як образ віртуальної машини (VM), зокрема, такий як Amazon EC2 AMI. Кожен компонент сервісу – це віртуальна машина (наприклад, екземпляр EC2), яка запускається з використанням даного образу віртуальної машини.

Нижче, на рис. 2.11 зображену схема розгортання сервісів на окремих серверах (*хостах*) із використанням VM.

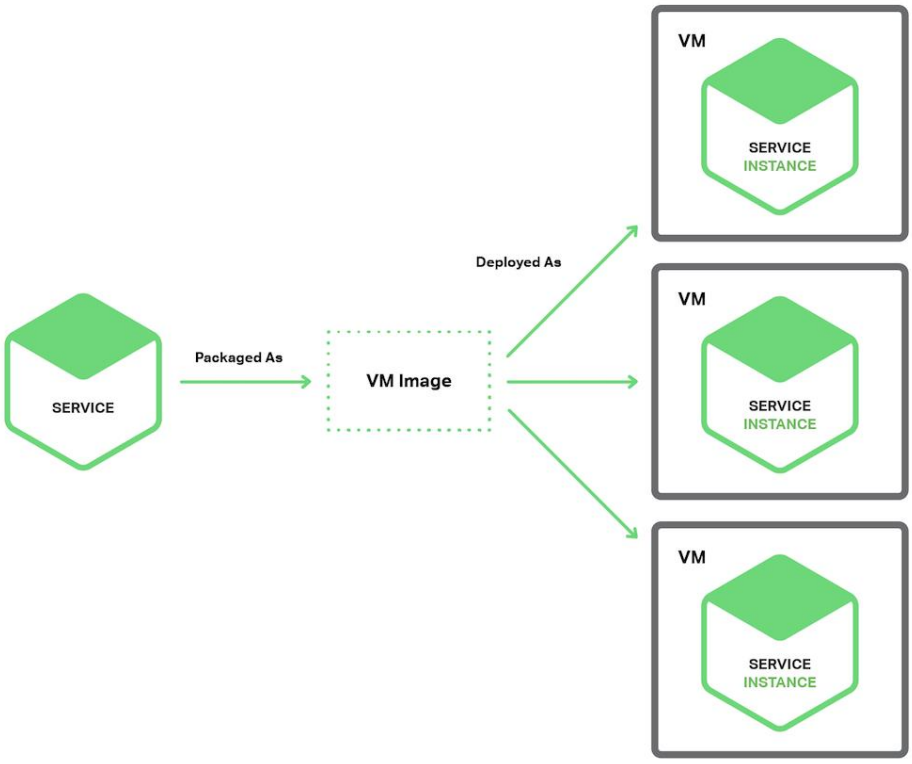


Рис. 2.11. Схема розгортання сервісів із використанням VM

Безперервна інтеграція – важлива частина DevOps та високоефективних команд з розробки програмного забезпечення. У цьому переваги CI приносять користь як команді інженерів, а й решті частинам організації. CI підвищує прозорість, а також допомагає аналізувати процес розробки та постачання ПЗ. Ці переваги дозволяють іншим співробітникам організації краще планувати та реалізовувати ринкові стратегії. Нижче наведено деякі із загальних переваг CI для організації.

CI дозволяє організаціям масштабувати команду інженерів, кодову базу та інфраструктуру. CI допомагає створювати робочі процеси DevOps та Agile, мінімізуючи бюрократію при інтеграції коду та надмірну комунікацію. Завдяки цьому кожен учасник команди може зробити зміну коду аж до випуску. CI дозволяє виконувати масштабування завдяки видаленню організаційних залежностей розробки окремих функцій. Розробники можуть створювати функції самостійно і з повною впевненістю в тому, що злиття їхнього коду з рештою кодової бази пройде без проблем. Це основний процес DevOps.

2.4. Калібрування послуг

Масштабованість при хмарних обчисленнях - це можливість швидко і легко збільшити або зменшити розмір або потужність IT-рішення або ресурсу. У той час як термін "масштабованість" може означати здатність будь-якої системи впоратися зі зростаючим обсягом роботи, у контексті горизонтального та вертикального масштабування мова часто йде про бази даних та великі обсяги даних.

Забезпечення масштабованості бази даних — пріоритетне завдання для розробників сучасних додатків. Припустимо, новий додаток стає популярним. Попит на нього зростає від декількох користувачів до мільйонів користувачів у всьому світі. На цьому етапі ефективне масштабування критично необхідне розробникам додатків, щоб адаптуватися до попиту і мінімізувати час простою.

Вертикальне масштабування використовується, коли необхідно швидко реагувати на проблеми з продуктивністю, які не можна вирішити за допомогою класичної методики оптимізації бази даних, наприклад, зміни запитів або індексування. Вертикальне масштабування допомагає впоратися з піками у робочих навантаженнях, коли поточний рівень продуктивності не може задовольнити всі вимоги. Вертикальне збільшення масштабу дозволяє додавати додаткові ресурси, щоб легко адаптуватися до пікових робочих навантажень. Потім, якщо ресурси більше не потрібні, можна виконати вертикальне зменшення масштабу, щоб повернутися до початкового стану та скоротити витрати на хмару.

Розробники програм починають застосовувати горизонтальне масштабування, якщо їм не вдається отримати достатньо ресурсів для робочих навантажень навіть на найвищих рівнях продуктивності. При горизонтальному масштабуванні дані розбиваються кілька баз даних (або сегментів) між серверами. Масштаб кожного сегмента можна вертикально збільшувати чи зменшувати окремо.

Як секціонування даних підвищує масштабованість? При вертикальному збільшенні масштабу окремої бази даних шляхом додавання таких ресурсів, як віртуальні машини, буде досягнуто фізичне обмеження обладнання. Кожна секція даних розміщується на окремому сервері, тому якщо розділити дані на кілька сегментів, можна практично без обмежень горизонтально збільшувати масштаб системи.

Деякі типи технологій баз даних, особливо нереляційні бази даних або бази даних NoSQL розробляються з унікальними можливостями горизонтального збільшення масштабу даних шляхом сегментування. Це дозволяє таким баз даних керувати об'ємними, незв'язаними, невизначеними або швидко змінюються.

Крім того, деякі реляційні служби баз даних (SQL), що спочатку пропонували послуги вертикального збільшення або зменшення масштабу, починають надавати цікаві можливості, дозволяючи досягти рівня масштабування нереляційних баз даних. Служби гіпермасштабування, такі як Гіпермасштабування Бази даних SQL Microsoft Azure та Гіпермасштабування Бази даних Azure PostgreSQL, дають користувачам можливість швидко масштабувати сховище до 100 ТБ, забезпечують орієнтовану на хмару гнучку архітектуру, дозволяючи збільшувати обсяг сховища відповідно до потреб, а також включати резервного копіювання та швидкі операції відновлення баз даних за кілька хвилин.

Це обговорення відмінностей між горизонтальним та вертикальним масштабуванням зосереджено на способах, за допомогою яких масштабованість дозволяє адаптуватися до величезних обсягів різноманітних даних та керувати ними, змінюючи обсяги даних та шаблони робочих навантажень, що створюються у хмарі, на мобільних пристроях, у соціальних мережах та джерелах великих даних. На рис. 2.12 зображена схема із трьох серверів і балансера навантаження. Запити від користувачів надходять до LB, який рівномірно їх розподіляє по наявним серверам.

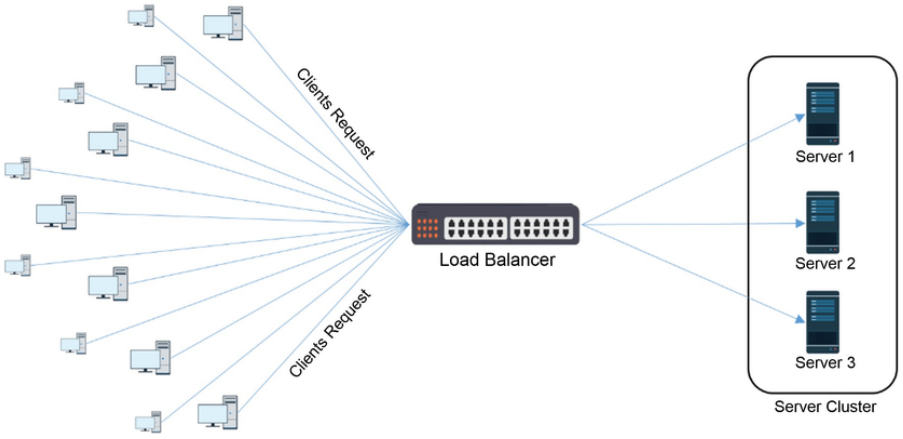


Рис. 2.12. Схема кластеризації серверів

Автоматичне масштабування – це процес автоматичного та динамічного узгодження ресурсів із вимогами до продуктивності системи. У міру збільшення обсягу роботи програм можуть знадобитися додаткові ресурси для підтримки необхідних рівнів продуктивності або задоволення зростаючих потреб. Якщо потреба зменшується і додаткові ресурси більше не потрібні, ви можете скоротити витрати на хмару за допомогою автоматичної служби, яка скасує виділення ресурсів, що не використовуються.

Автоматичне масштабування використовує переваги еластичності хмарного середовища. Це спрощує управління, зменшуючи необхідність для системних операторів постійно приймати рішення щодо додавання або видалення ресурсів або перевірки продуктивності системи.

Є два основних способи масштабування додатків: вертикальне та горизонтальне. Вертикальне масштабування рідше виконується автоматично, тому що при ньому часто потрібно, щоб система була тимчасово недоступна під час повторного розгортання.

Автоматичне масштабування найчастіше виконується при горизонтальному масштабуванні, оскільки горизонтальне збільшення або зменшення масштабу означає просто додавання або видалення екземплярів ресурсу. При цьому ваша програма продовжує роботу без переривання під час підготовки нових ресурсів. Якщо потреба зменшується, ви можете

безперешкодно і без простоїв завершити роботу ресурсів, а також скасувати виділення.

Багато постачальників хмарних систем, таких як Microsoft Azure, підтримують автоматичне горизонтальне масштабування. Оркестровка — це координація взаємодії кількох контейнерів. Оркестровка дозволяє створювати інформаційні системи з кількох контейнерів, кожен з яких відповідає лише за одне конкретне завдання, а зв'язок здійснюється через мережеві порти та спільні каталоги. При необхідності кожен такий контейнер можна замінити іншим, що дозволяє, наприклад, при необхідності швидко перейти на іншу версію бази даних.

Існують різноманітні платформи для організації контейнерів, що дозволяє реалізувати зручні та ефективні методи розміщення контейнерних систем, налагодити єдиний централізований сервіс для реалізації політик управління. Найпопулярнішими є: Kubernetes, Docker Swarm і Apache Mesos.

Оркестрація — це єдиний централізований виконуваний бізнес-процес (Orchestrator), який координує взаємодію між різними службами (рисунок 2.13). Оркестр відповідає за виклик і суміщення послуг. Взаємодія між усіма сервісами описується в одній кінцевій точці - складовій службі (composite service).

Оркестровка передбачає управління взаємодією та координацією між окремими службами. Іншими словами, можна зробити висновок, що оркестровка використовує централізований підхід до управління послугами.

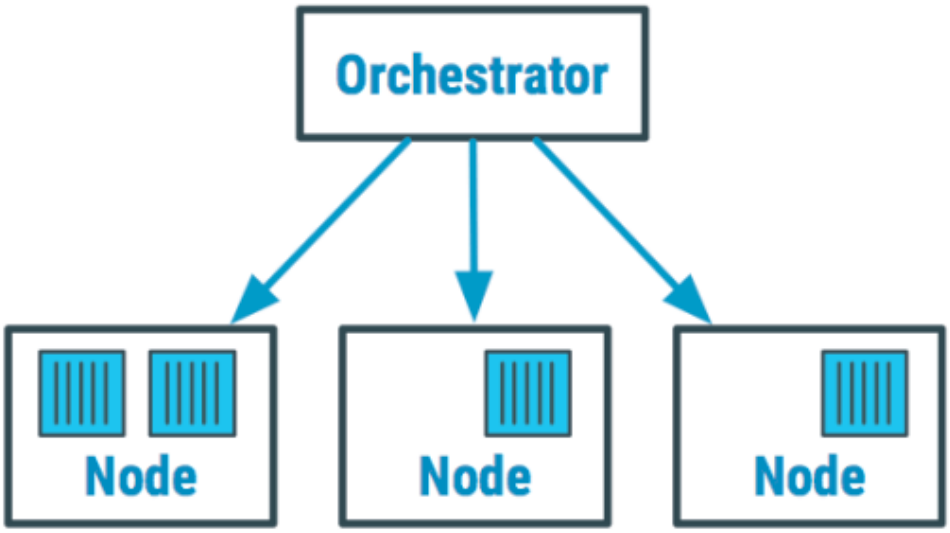


Рис. 2.13. Приклад оркестрації сервісів

2.5. Перевірка серверів

Сучасні корпоративні інформаційні системи за своєю природою завжди є розподіленими системами. Робочі станції користувачів, сервери програм, сервери баз даних та інші мережні вузли розподілені по великій території. У великій компанії офіси та майданчики з'єднані різними видами комунікацій, що використовують різні технології та мережеві пристрої. Головне завдання мережевого адміністратора – забезпечити надійну, безперебійну, продуктивну та безпечну роботу всієї цієї складної системи.

Розглянемо мережу як сукупність програмних, апаратних та комунікаційних засобів, що забезпечують ефективний розподіл обчислювальних ресурсів. Всі мережі можна умовно поділити на 3 категорії:

- локальні мережі (LAN, Local Area Network);
- глобальні мережі (WAN, Wide Area Network);
- міські мережі (MAN, Metropolitan Area Network).

Глобальні мережі дозволяють організувати взаємодію між абонентами великих відстанях. Ці мережі працюють на відносно низьких швидкостях і можуть вносити значні затримки передачі інформації. Протяжність глобальних

мереж може становити тисячі кілометрів. Тому вони так чи інакше інтегровані із мережами масштабу країни.

Міські мережі дозволяють взаємодіяти на територіальних утвореннях менших розмірів та працюють на швидкостях від середніх до високих. Вони менше уповільнюють передачу даних, ніж глобальні, але можуть забезпечити високошвидкісне взаємодія великих відстанях. Протяжність міських мереж знаходиться в межах від кількох кілометрів до десятків та сотень кілометрів.

Локальні мережі забезпечують найвищу швидкість обміну інформацією між комп'ютерами. Типова локальна мережа займає простір в один будинок. Протяжність локальних мереж становить близько кілометра. Їхнє основне призначення полягає в об'єднанні користувачів (як правило, однієї компанії або організації) для спільної роботи.

Механізми передачі даних у локальних та глобальних мережах суттєво відрізняються. Глобальні мережі орієнтовані на з'єднання — до початку передачі між абонентами встановлюється з'єднання (сеанс). У локальних мережах використовуються методи, що не вимагають попередньої установки з'єднання, пакет з даними посилається без підтвердження готовності одержувача до обміну.

Крім різниці у швидкості передачі даних, між цими категоріями мереж існують інші відмінності. У локальних мережах кожен комп'ютер має мережевий адаптер, який з'єднує його із середовищем передачі. Міські мережі містять активні пристрої, що комутують, а глобальні мережі зазвичай складаються з груп потужних маршрутизаторів пакетів, об'єднаних каналами зв'язку. Крім того, мережі можуть бути приватними або мережами загального користування.

РОЗДІЛ 3
СПОСІБ КОНТЕЙНЕРИЗАЦІЇ МІКРОСЕРВІСІВ -DOCKER

3.1. Принцип контейнеризації

Легкість, швидкодія та можливість працювати на високому рівні абстракції, делегуючи проблеми із залізом та ОС провайдеру, — це переваги контейнерів, що дозволяють знизити операційні витрати, пов'язані з розробкою та експлуатацією додатків, які роблять рішення на їх основі настільки привабливими для бізнесу.

Технічним спеціалістам контейнери насамперед полюбилися за можливість упаковати додаток разом з його середовищем запуску, вирішуючи цим проблему залежностей у різних оточеннях. Потрібно витратити час на їх аналіз, а як максимум вирішувати проблему багів, що проникли в продакшен. Використання контейнерів вирішує проблему «А на моїй машині все працювало!

Перевагою контейнеризації є масштабованість. Головним є те, що можливість швидко здійснювати горизонтальне масштабування, нові контейнери будут створені для короткострокових завдань. З точки зору програми, створення нового екземпляра образу (контейнера) рівноцінно створенню екземпляра процесу, яке буде створено для служби або веб-додатку.

Дозволяє легко розміщувати контейнери. У класичному підході вам може знадобитися виконати кілька дій для встановлення програми: запустити скрипт, змінити файли конфігурації тощо. У цьому процесі не виключається можливість людської помилки: користувач двічі запустить скрипт, переплутає послідовність або щось не зрозуміє. Контейнери дозволяють повністю автоматизувати цей процес, оскільки включають всі необхідні залежності та процедури.

Контейнери також полегшують розміщення на кількох серверах. У класичному підході вам доведеться повторити ті самі кроки, щоб розмістити ту саму програму на більш ніж одній машині. Контейнери позбавляють вас цієї рутини і дозволяють автоматизувати розміщення.

Контейнери добре вписуються у мікросервісну архітектуру. Це підхід до розробки, при якому програма розбивається на невеликі компоненти, по можливості незалежні. Зазвичай протиставляється монолітної архітектури, де всі частини системи сильно пов'язані одна з одною.

Це дозволяє розробляти нову функціональність швидше, адже у випадку з монолітною архітектурою зміна якоїсь частини може торкнутися всієї іншої системи.

В кожен контейнер можливо вміщати цілий веб-застосунок або службу, як показано на рис. 2.1. У даному прикладі вузол Docker – це вузол контейнерів, а *App1*, *App2*, *Svc*1 і *Svc*2 – контейнерні додатки або служби.

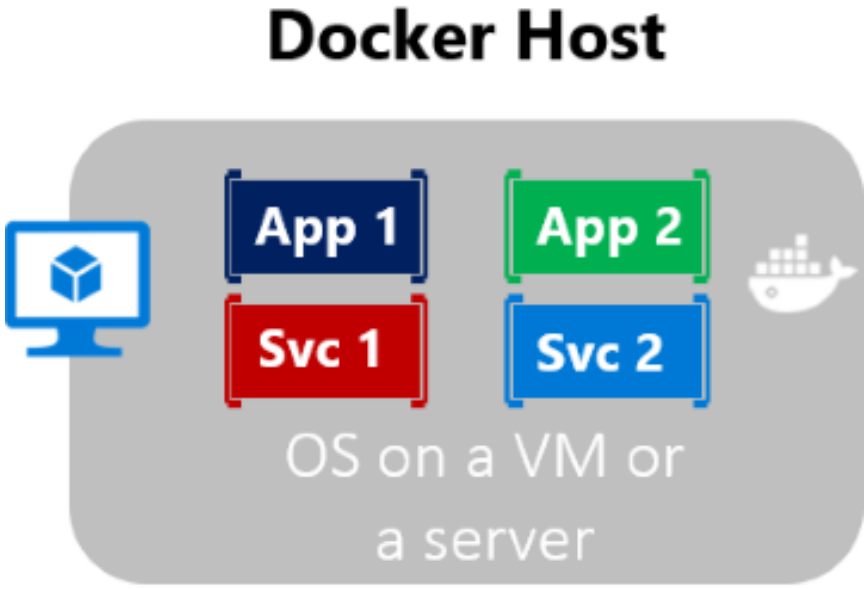


Рис. 3.1. Принцип роботи контейнерів

Це означає, що контейнери забезпечують такі переваги, як ізоляція, портативність, маневреність, масштабованість і контроль протягом усього терміну дії програми. Основна перевага — ізольованість середовища розробки від робочого середовища.

Найпоширенішим питанням при виборі середовища запуску програми є різниця між контейнерами та віртуальними машинами – двома найпопулярнішими на даний момент варіантами. Між ними є важлива різниця. Контейнер — це в основному обмежений простір в ОС, який використовує ядро хост-системи для доступу до апаратних ресурсів. VM – це машина з усіма необхідними для її роботи пристроями. Це створює відмінності, які мають практичне значення:

Контейнери вимагають значно менше грошей на свою роботу, що позитивно позначається на продуктивності та бюджеті.

Технологія контейнерної віртуалізації

Віртуалізація на рівні ОС дозволяє віртуалізувати фізичні сервери на рівні ядра операційної системи (рис. 3.2).

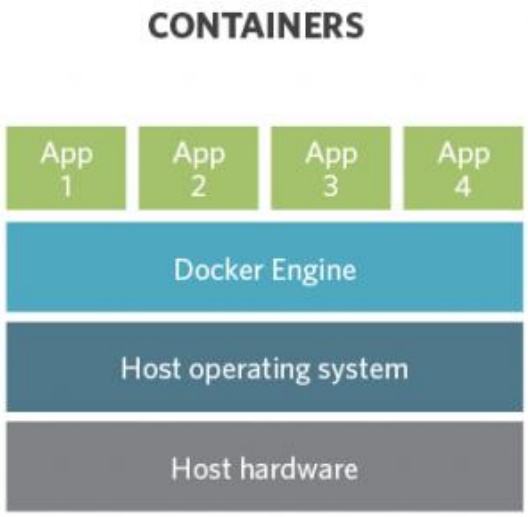


Рис. 3.2. Принцип роботи контейнерної віртуалізації

Для ефективної роботи програми у контейнерах недостатньо просто створити образ контейнера та запустити його. Потрібно подбати про те, щоб архітектура програми та контейнера відповідала базовим принципам контейнеризації, які добре виклала компанія RedHat.

1 контейнер - 1 послуга. Контейнер повинен виконувати лише одну функцію - не варто розміщувати всі сутності, з яких походить додаток. Дотримання цього принципу дозволяє досягти більшого повторного використання зображень і, головне, дозволяє тонко налаштувати програму - вузьке місце вашого сервісу може бути лише частиною купи використовуваних технологій, а множення всіх його частин у різних контейнерах збільшить ефективність Вашого обслуговування.

Інваріантність зображення. Усі зміни всередині танка необхідно вносити на етапі збору зображень – цей принцип захищає вас від втрати даних при знищенні танка. Container також дозволяє виконувати паралельні завдання в системах CI/CD - наприклад, можна виконувати різні тести одночасно, прискорюючи процес розробки продукту.

Утилізованість контейнерів. Цей принцип є яскравим прикладом сучасної концепції «Звертайся з інфраструктурою як зі худобою, не як з вихованцями». Це означає, що будь-який контейнер може бути у будь-який момент знищено та замінено на інший без зупинки обслуговування. Конфігурація контейнера у

вигляді його образу сутнісно відокремлена від безпосередньо виконує роботу екземпляра контейнера, що дозволяє «пускати під ніж» екземпляри, коли потрібно — при збої перевірки стану контейнера, масштабуванні на зниження і т. д. Відповідність цьому принципу означає, що вихід контейнерів з ладу не повинен бути новиною для вашої програми: ротація контейнерів має стати однією з вимог до розробки.

Звітність. Контейнер повинен мати точки перевірки стану його готовності (readiness probe) та життєздатності (liveness probe), надавати логи для відстеження стану запущеного у ньому додатка.

Керованість. Додаток у контейнері повинен мати можливість взаємодіяти з процесом, що його контролює, — наприклад, для коректного завершення своєї роботи по команді ззовні. Це дозволить акуратно закривати транзакції, перешкоджаючи втраті даних у результаті зупинки або знищення контейнера.

Самодостатність. Образ з додатком повинен мати всі необхідні залежності для роботи - бібліотеки, конфіги та інше. Сервіси до цих залежностей не відносяться, інакше це суперечило б принципу «1 контейнер — 1 сервіс». Зв'язковість контейнерів, що залежать один від одного, можна визначити за допомогою інструментів оркестрування, про що буде розказано нижче.

Технологія апаратної віртуалізації

Щоб надати ресурси віртуальним машинам, забезпечується їх віртуалізація на сервері. Віртуальні машини запускають свою власну копію операційної системи і додатків на віртуалізованому обладнанні (рис.3.3).

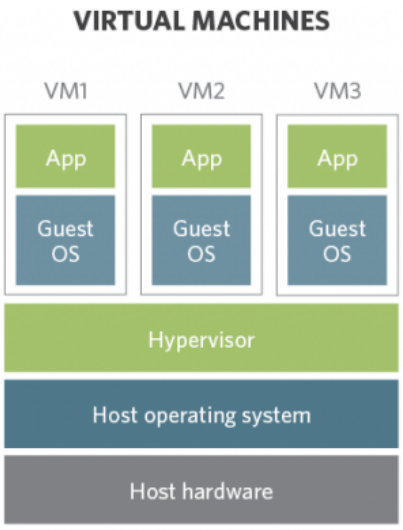


Рис. 3.3. Принцип роботи апаратної віртуалізації

Програмний спосіб передбачає використання хостової ОС, поверх якої запускається спеціальна платформа для емуляції апаратних компонентів і ресурсів. Реалізація платформи для симуляції у разі досить складна процедура, оскільки завжди є зниження продуктивності. Також нерідко виникає загроза безпеці даних, оскільки отримання доступу до хостової операційної системи означає надання доступу всім гостьовим ресурсам.

На відміну від такого способу апаратна віртуалізація пристроїв дозволяє отримати ізольовані гостьові системи. Тобто кожна з них управляється гіпервізором безпосередньо, що загалом підвищує рівень безпеки. Під час такого процесу немає втрати продуктивності. Така віртуалізація відбувається із застосуванням процесорної архітектури.

В основі технології закладено принцип поділу фізичного процесора на два компоненти: моніторну частину (root mode) та гостьову (так звана non-root mode). При перемиканні з базової ОС на гостьову процесор автоматично переходить у гостьовий стан. Тим самим змінюються значення регістру під параметри нової операційної системи. Особливість у тому, що гостьова ОС також може прямо працювати з процесором та переглядати інформацію про нього, чого не відбувається за програмної віртуалізації. Якщо необхідна установка розширень ядра ОС (наприклад для VPN), повний контроль над

ядром ОС, можливість використання Docker, ручний апгрейд самої ОС – варто використовувати апаратну віртуалізацію.

З метою підтримки апаратної віртуалізації суттєво перероблено архітектуру обладнання. В результаті цього було надано безпосередній доступ до всіх ресурсів процесора прямо з гостьових ОС. Було додано додатковий набір інструкцій – VMX, що містить необхідну інформацію для доступу.

В результаті процесор під час віртуалізації працює у кількох режимах – root та non-root operation. У другому випадку використовується спеціальне ПЗ, що є прокладкою між базовою операційною системою (монітором ВМ) і гостьовими ОС.

Переведення процесора в режим віртуалізації відбувається за рахунок виконання певної команди та передачі управління гіпервізору. Кожна з гостьових ОС при цьому запускається та функціонує незалежно від інших.

Методика може реалізовуватись по-різному. Виділяють такі типи технології апаратної віртуалізації:

Повна. Такий варіант дозволяє повністю імітувати обладнання для гостьової ОС. Тобто створюється середовище, повністю аналогічне працюючому на окремому сервері. Але при цьому і гостьова, і основна операційна система розташовуються на одному процесорі. Такий спосіб дозволяє запускати віртуальне середовище будь-якої конфігурації без додаткових налаштувань, програмних та апаратних компонентів.

Паравіртуалізація. За такого підходу на віртуальній машині запускається заздалегідь підготовлена версія гостьової ОС (вона може бути змінена або перекомпільована). Це дозволяє дотримуватися вихідних вимог до обладнання лише частково.

З апаратною підтримкою. Апаратне забезпечення у цьому випадку використовується як «архітектурна підтримка», що дозволяє створити нову віртуальну машину та керувати нею. Такий спосіб віртуалізації сьогодні можна назвати найпоширенішим.

Переваги використання Docker

Ресурс мінімального споживання - це система, яка працює без віртуальних операцій (OS), ядро міцності матки та програми виключаються в процесі процесу.

Приклад безкоштовного встановлення та налаштування константної системи Linux Ubuntu.

Зручне приховування процесів – для кожного контейнера можна використовувати різні методи обробки даних, приховуючи фонові процеси.

Робота з небезпечним кодом – технологія ізоляції контейнерів дозволяє запускати будь-який код без шкоди для ОС.

Просте масштабування – будь-який проект має можливість розширити, впровадивши нові контейнери.

Зручний запуск - програму, яка знаходиться в зоні контейнера, можна запустити на будь-якому docker-хості.

Оптимізація файлової системи - образ складається з декілької шарів, які дозволяють ефективніше використовувати надану файлову систему.

3.2. Будова Docker

Коли ми говоримо про контейнери в сучасних IT-системах, перш за все ми маємо на увазі Docker — open-source-технологію, завдяки своїй популярності слова «контейнер», що стала в IT синонімом.

Основні сутності Docker:

Dockerfile. Текстовий файл, який використовується для створення образу контейнера. Містить посилання на базовий образ, що служить відправною точкою при формуванні нового образу і набір інструкцій для складання, таких як установка залежностей, компіляція програми і копіювання конфігів. Також він містить точку входу в контейнер - команду, що виконується під час його запуску.

Image. Готова файлова система, сформована за інструкціями з Dockerfile і служить прообразом для контейнерів, що запускаються.

Instance. Запущений екземпляр образу, мінімальна одиниця деплою у Docker.

Volume. Файлова система, що підключається до контейнерів, не є їх невід'ємною частиною і існує незалежно від образу. За допомогою об'єктів Volume вирішується проблема збереження даних, записаних у процесі роботи контейнерів у локальній файловій системі після їх знищення.

Registry. Репозиторій, що використовується для зберігання образів Docker. Registry може бути як публічним, так і приватним, захищеним механізмом автентифікації.

Процес розробки в середовищі Docker.

Типовий процес розробки в середовищі Docker виглядає наступним чином: розробники встановлюють на свої машини Docker, завантажують зібраний заздалегідь образ із встановленим середовищем складання та виконання програми, а потім запускають контейнер командою, яка також прокине в нього директорію з вихідними джерелами. Для установки Docker не потрібно особливого заліза - він може бути встановлений на звичайному автомобілі. Однак у нього є обмеження за версіями ОС - Windows 7 64bit або вище для ПК з підтримкою Hyper-V, MacOS Sierra 10.12 для пристроїв від Apple і версією ядра 3.10 для систем з Linux. Контейнери Docker є рідною технологією для ОС Linux та запускаються на інших за допомогою віртуальних машин під її керуванням.

Інструкції зі встановлення Docker на різних платформах: Windows, Mac, Linux Ubuntu.

Щоб запустити ваш перший контейнер на Docker, після його встановлення введіть у командному рядку docker run hello-world - ця команда завантажить образ hello-world з Docker hub'a (публічно доступний Docker registry), створить контейнер, використовуючи цей образ і видасть вітальну фразу:

```
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

У своєму ядрі docker дозволяє запускати практично будь-яку програму, безпечно ізольовану в контейнері. Безпечна ізоляція дозволяє запускати на одному хості багато контейнерів одночасно. Легка природа контейнера, який запускається без додаткового навантаження гіпервізора, дозволяє вам домагатися більше від заліза.

Платформа та засоби контейнерної віртуалізації можуть бути корисними у таких випадках:

- упаковка вашої програми (і так само використовуваних компонент) в контейнери docker;
- роздача та доставка цих контейнерів вашим командам для розробки та тестування;
- викладання цих контейнерів на ваші продакшени, як у дата центри, так і в хмари.

Що використовується для контейнеризації на рівні ядра

Основні технології, які дозволяють створювати ізольований від інших процесів контейнер, це Namespaces і Control Groups.

Namespaces: PID, Networking, Mount та User. Є ще, але для простоти розуміння зупинимося на цих.

PID Namespace обмежує процеси. Коли ми, наприклад, створюємо PID Namespace, поміщаємо туди процес, він стає з PID 1. Зазвичай у системах PID 1 — це systemd чи init. Відповідно, коли ми поміщаємо процес у новий namespace, він також отримує PID 1.

Networking Namespace дозволяє обмежити/ізолювати мережу та всередині вже розміщувати свої інтерфейси. Mount — це обмеження файлової системи. User - обмеження за користувачами.

Control Groups: Memory, CPU, IOPS, Network — лише близько 12 налаштувань. Інакше їх ще називають Cgroups (Сі-групи).

Control Groups керують ресурсами контейнера. За допомогою Control Groups ми можемо сказати, що контейнер не повинен споживати більше кількості ресурсів.

Щоб контейнеризація повноцінно працювала, використовуються додаткові технології: Capabilities, Copy-on-write та інші.

Capabilities це коли ми говоримо процесу, що він може робити, а чого не може. На рівні ядра це просто бітові карти з багатьма параметрами. Наприклад, користувач має повні привілеї, може робити все. Сервер часу може змінювати системний час: у нього є можливості на Time Capsule, і все. За допомогою привілеїв можна гнучко налаштувати обмеження для процесів і тим самим убезпечити себе.

Система Copy-on-write дозволяє нам працювати з образами Docker, використовувати їх ефективніше.

Наразі Docker має проблеми із сумісністю Cgroups v2, тому у статті розглядаються саме Cgroups v1.

Клієнт Docker взаємодіє з демоном Docker, який береться за себе створення, запуск, масштабування контейнерів, організацію томів, підтримку мережі.

Клієнт і сервер спілкуються за допомогою сокет або за допомогою RESTful API.

Docker-image — це шаблон лише для читання з набором інструкцій для створення контейнера. Він складається з шарів, які Docker об’єднує в одне зображення за допомогою допоміжної файлової системи UnionFS. Це вирішує проблему нераціонального використання дискової пам'яті. Параметри зображення визначаються у файлі Docker.

Для багаторазового використання Docker-image необхідно використовувати реєстр образів або Docker-register (Docker-register), що дозволяє завантажувати готові образи із зовнішнього репозиторію сервісу та зберігати їх у реєстрі Docker-host. Рекомендований варіант — офіційний довірений реєстр Docker (DTR).

Docker Engine складається з трьох компонентів (Малюнок 3.4):

• сервер – зображення, контейнери, мережа тощо. демон докера під назвою dockerd, який може створювати й керувати;

• REST API - використовується для зв'язку з демоном докера;

• Інтерфейс командного рядка (CLI) – клієнт, який використовується для введення команд Docker.

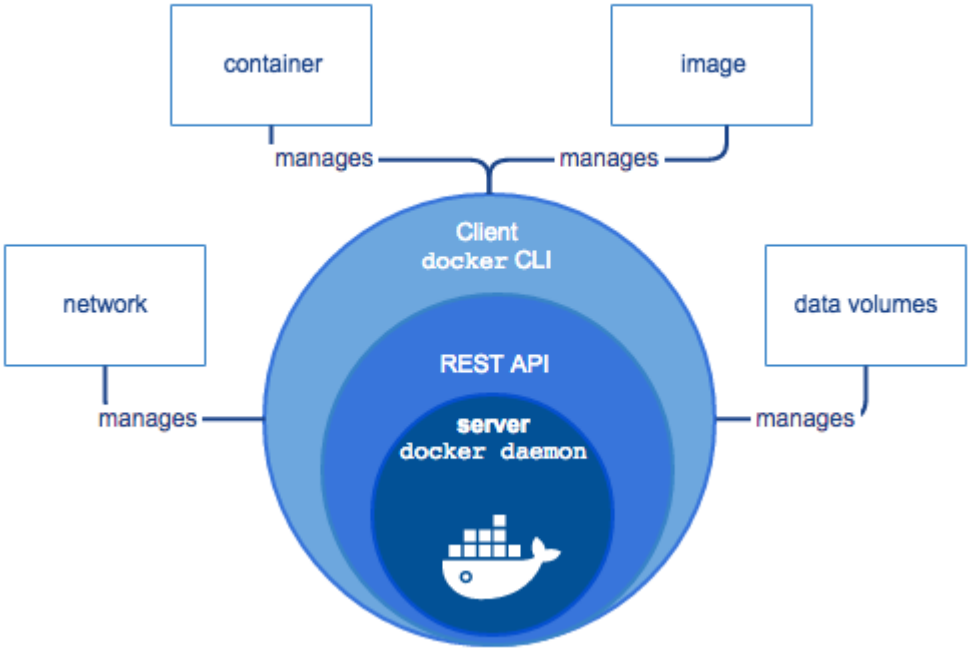


Рис. 3.4. Принцип роботи Docker

Docker демон це мозок, що стоїть за всією операцією, як і сама aws. Коли ви використовуєте команду docker run для запуску контейнера, ваш клієнт docker перетворює цю команду на виклик http API, відправляє її демону docker, потім демон Docker оцінює запит, зв'язується з базовою ОС та готує ваш контейнер.

Зверніть увагу, що docker cli може підключатися до віддаленого docker демона, і ви можете налаштувати свій docker демона на використання tcp IP.

Це залежить від вас, але в більшості випадків у розробників є локальний демон docker і клієнт, який створює образи за допомогою dockerfiles. Якщо їм

потрібно поділитися зображеннями docker, можна надати локальний реєстр docker або використовувати загальнодоступні зображення. Таким чином, скориставшись docker, ви можете мати в розпорядженні розробників таке саме середовище розробки. Це середовище розробки буде аналогічне до робочого середовища.

За допомогою команди dockerd можна запустити сервер Docker. Після введення команди, сервер недійшле інформаційні повідомлення про успішний запуск сервера:

```
$ dockerd
INFO[0000] +job init_networkdriver()
INFO[0000] +job serveapi(unix:///var/run/docker.sock)
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
```

Демон Docker за замовчуванням прослуховує Unix-сокет. Для прослуховування іншого порту, наприклад 2375, можна створити конфігураційний файл `/etc/systemd/system/docker.socket.d/socket.conf` із наступним вмістом:

```
[Socket]
ListenStream=0.0.0.0:2375
```

Демон Docker використовує «драйвер під час виконання» для створення контейнера. За замовчуванням це драйвер Docker runc, але він підтримує LXC.

Runc відповідає за функціональність:

• Групи, відповідальні за управління використовуваними ресурсами контейнера (тобто використання ЦП і ОЗП).

• Простіри імен, які відповідають за ізоляцію контейнерів і забезпечення того, щоб файлова система, ім'я хоста, мережеве середовище, список процесів і контейнери були відокремлені від інших частин основної операційної системи.

Клієнт Docker є основним інтерфейсом Docker. Клієнт отримує команди від користувача і взаємодіє з демоном докера.

Клієнт Docker Host використовується для зв’язку з демоном Docker через HTTP. За замовчуванням це з сокетом домену Unix, тобто IPC (Inter-Process Communication Socket), але ви також можете використовувати сокет TCP, який дозволяє створювати клієнтів віддалено.

Таким чином, при роботі з інтерфейсом командного рядка (CLI) Docker термінал вводить команди, починаючи з клавіш слова docker, що означає доступ до клієнта Docker. Потім клієнт Docker використовує API Docker для надсилання команд демона Docker.

3.3. Робота з зображеннями (images)

Образ – це пакет з усіма залежностями і інформацією, яка необхідна для створення контейнера. Образ включає в себе всі залежності (наприклад, платформа, бібліотеки), а також конфігурацію розгортання і виконання для середовища виконання контейнера.

Як правило, образ створюється на основі декількох базових образів, нашарованих один на одного в файлової системі контейнера. Після створення образ залишається незмінним.

ocker-образ — шаблон створення Docker-контейнерів. Являє собою пакет, що містить все необхідне для запуску програми: код, середовище виконання, бібліотеки, змінні оточення і файли конфігурації.

Docker-образ складається із шарів. Кожна зміна записується у новий шар.

При завантаженні або завантаженні Docker-образу операції виконуються тільки з тими шарами, які були змінені.

Шари вихідного Docker-образу є спільними між усіма його версіями і дублюються.

Керування версіями Docker-образу здійснюється за допомогою тегів та хешів.

Тег – присвоюється користувачем. Тег повинен бути унікальним у межах одного репозиторію і може бути змінений. Якщо тег не був вказаний, при

завантаженні образа Docker в реєстр, Docker CLI за замовчуванням встановлює latest.

Рекомендується не перезаписувати теги, а використовувати для кожної версії образа Docker унікальний тег. Це дозволяє використовувати одну версію Docker-образу на всіх ВМ з однаковою специфікацією та полегшує пошук причини проблеми.

Один Docker-образ може мати кілька тегів. Якщо ви завантажуєте нову версію Docker-образу з існуючим тегом, він буде перевикористаний — видалений зі старої версії Docker-образу і записаний на нову.

Хеш – генерується автоматично, є унікальним та однозначно визначає версію Docker-образу.

Звернутися до певної версії Docker-образу можна одним із способів:

<реєстр>/<ім'я образу>:<тег>;

<реєстр>/<ім'я образу>@<хеш>.

Docker-образ і всі його версії зберігаються у репозиторії.

Docker може автоматично створювати образи, читаючи інструкції з Dockerfile. Файл Dockerfile являє собою текстовий документ, що містить всі команди для складання образу. За допомогою команди docker build користувачі можуть виконувати автоматизовану збірку, яка виконує послідовність інструкцій у командному рядку.

На цій сторінці описані команди, які можна використовувати в Dockerfile. Коли ви ознайомитеся з цим розділом, рекомендуємо також прочитати Dockerfile найкраща практика.

Команда docker build створює образ із Dockerfile та контексту. Контекст збирання це файли із заданим місцезнаходженням на локальному комп'ютері (PATH) або URL, що вказує на Git репозиторій.

Контекст опрацьовується рекурсивно. Отже PATH включає всі піддиректорії і URL включає репозиторій і його підмодулі. Проста команда створення образу включає поточну директорію та її контекст:

$ docker build .

Sending build context to Docker daemon 6.51 MB

...

Створення образу запускається Docker демоном, а чи не інтерфейсом командного рядка (CLI). Насамперед процес складання рекурсивно відправляє контекст демону. У більшості випадків, краще почати з порожнього каталогу як контекст і зберегти в нього Dockerfile. Додайте в нього тільки ті файли, які будуть використовуватися в Dockerfile.

Для використання файлу в контексті збірки образу, Dockerfile використовують спеціальну інструкцію, наприклад COPY. Для покращення продуктивності при складанні образу, можна виключити непотрібні файли з контексту, додавши в непотрібний каталог файл .dockerignore. Докладніше про те, як створити файл .dockerignore, перейдіть за посиланням.

Традиційно Dockerfile розміщується до кореня контексту. Використовуйте прапорець -f з командою docker build щоб задати інше розташування Dockerfile у файловій системі.

$ docker build -f /path/to/a/Dockerfile .

Ви можете вказати репозиторій і тег щоб зберегти образ при успішному складанні:

$ docker build -t shykes/myapp .

Для того щоб позначити образ в декількох репозиторіях після складання, додайте кілька параметрів -t коли ви запускаєте команду складання build:

$ docker build -t shykes/myapp:1.0.2 -t shykes/myapp:latest .

Шари у підсумковому образі створюють лише інструкції FROM, RUN, COPY та ADD. Інші інструкції щось налаштовують, описують метадані, або повідомляють Docker про те, що під час виконання контейнера потрібно щось зробити, наприклад відкрити якийсь порт або виконати якусь команду.

Тут ми виходимо з припущення, відповідно до якого використовується образ Docker, що базується на Unix-подібній ОС. Звичайно, тут можна

скористатися і способом, заснованим на Windows, але використання Windows це менш поширена практика, працювати з такими образами складніше. В результаті, якщо у вас є така нагода, користуйтеся Unix. Створюючи новий контейнер, додається новий шар для запису над нижчими шарами. Цей шар часто називають контейнерним шаром. Всі зміни, внесені до запущеного контейнера, такі як запис нових файлів, зміна існуючих файлів та видалення файлів, записуються на цей тонкий шар контейнера. На рис.3.5 показаний контейнер на основі образу Ubuntu 15.04.

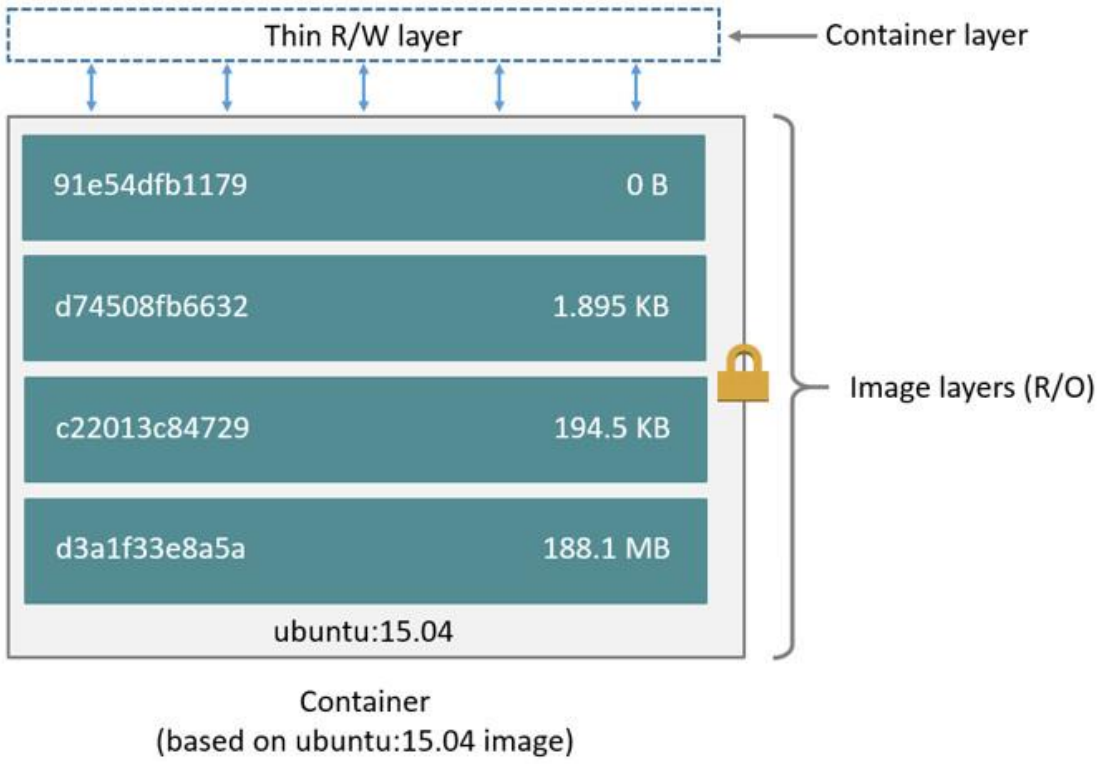


Рис. 3.5. Схема шарів образу

Репозиторій являє собою набір пов’язаних зображень Docker, позначених версією зображення. Деякі депо мають кілька варіантів зображення, наприклад, набір інструментів розробки та тестування (більший обсяг), образ лише із середовищем виконання (менший обсяг) тощо. Репозиторій може мати такі параметри платформи, як образ Linux і образ Windows.

Реєстр зображень — це служба, яка надає доступ до сховищ. Стандартним реєстром для більшості загальнодоступних образів є Docker Hub. Зазвичай

реєстр містить кілька сховищ команд. Компанії часто використовують особисті реєстри для зберігання та керування своїми зображеннями.

Тобто реєстр Docker є віддаленим місцем, яке містить усі образи Docker. Розробники завантажують туди зображення або навпаки. Ви можете використовувати як свій власний реєстр, так і реєстр будь-якого постачальника, наприклад AWS або Google Cloud.

3.4. Основи роботи контейнерів

Вона з основних особливостей контейнерів — ефемерність. Це означає, що контейнери можуть бути зупинені, перезапущені або знищені. При цьому всі накопичені дані у контейнері будуть втрачені. Тому програми потрібно розробляти так, щоб вони не покладалися на сховище даних у контейнері, це називається принципом Stateless.

Це добре підходить для програм або сервісів, які не зберігають результати своєї роботи. Наприклад, функції розрахунку або перетворення даних: їм на вхід надійшов один набір даних, вони його перетворили або розрахували та повернули результат. Все нічого нікуди зберігати не потрібно.

Але далеко не всі програми такі, і є багато даних, які потрібно зберегти. У контейнерах цього передбачено кілька способів.

Тома (Docker volumes)

Це спосіб, коли докер сам створює директорії для зберігання даних. Їх можна зробити доступними для різних контейнерів, щоб вони могли обмінюватись даними. За замовчуванням ці директорії створюються на хост-машині, але можна використовувати віддалені сховища: файловий сервер або об'єктне сховище.

Монтування каталогу (bind mount)

У цьому випадку директорія спочатку створюється в хост-системі, а вже потім вмонтовується в докер контейнери.

Але цей спосіб не рекомендується, тому що він ускладнює резервне копіювання, міграцію та спільне використання даних кількома контейнерами.

На рис. 3.6 зображено вивід терміну при запуску цього контейнера. Командою `docker ps` можна перевірити список усіх потрібних контейнерів, наявних на хост машині.

```
$ docker run -it ubuntu /bin/bash
root@af588b25a4ad:/#

$ docker ps
CONTAINER ID   IMAGE    COMMAND      CREATED         STATUS         PORTS   NAMES
af588b25a4ad   ubuntu   "/bin/bash"  24 seconds ago  Up 23 seconds          jovial
```

Рис. 3.6. Вивід у термінал при створенні контейнера

Docker Swarm — це система кластеризації для Docker, яка перетворює набори хостів Docker в єдиний послідовний кластер під назвою Swarm.

Основні особливості Docker Swarm:

• Баланс навантаження: Docker Swarm відповідає за балансування навантаження та призначення нових імен DNS, щоб програму, розміщену в цьому кластері, можна було використовувати як програму, розміщену на одному хості Docker;

• Динамічне керування ролями: хости Docker часто додаються до кластера Swarm без необхідності перезапускати кластер. Роль вузла (менеджера або співробітника) також може змінюватися;

• Динамічне масштабування послуг: служба може динамічно масштабуватися за зростанням або спаданням. Механізм управління піклується про додавання або вилучення контейнерів до вузлів;

• Контроль збоїв: вузли постійно контролюються вузлом керування, і якщо будь-який вузол виходить з ладу, нові завдання виконуються на інших робочих вузлах.

• Docker Swarm дозволяє створювати нові керуючі вузли, щоб запобігти помилкам кластера в разі відмови одного вузла керування;

• Швидкі оновлення: оновлення служби можна застосовувати поступово; На рис. 3.7 показана можлива схема архітектури Docker Swarm.

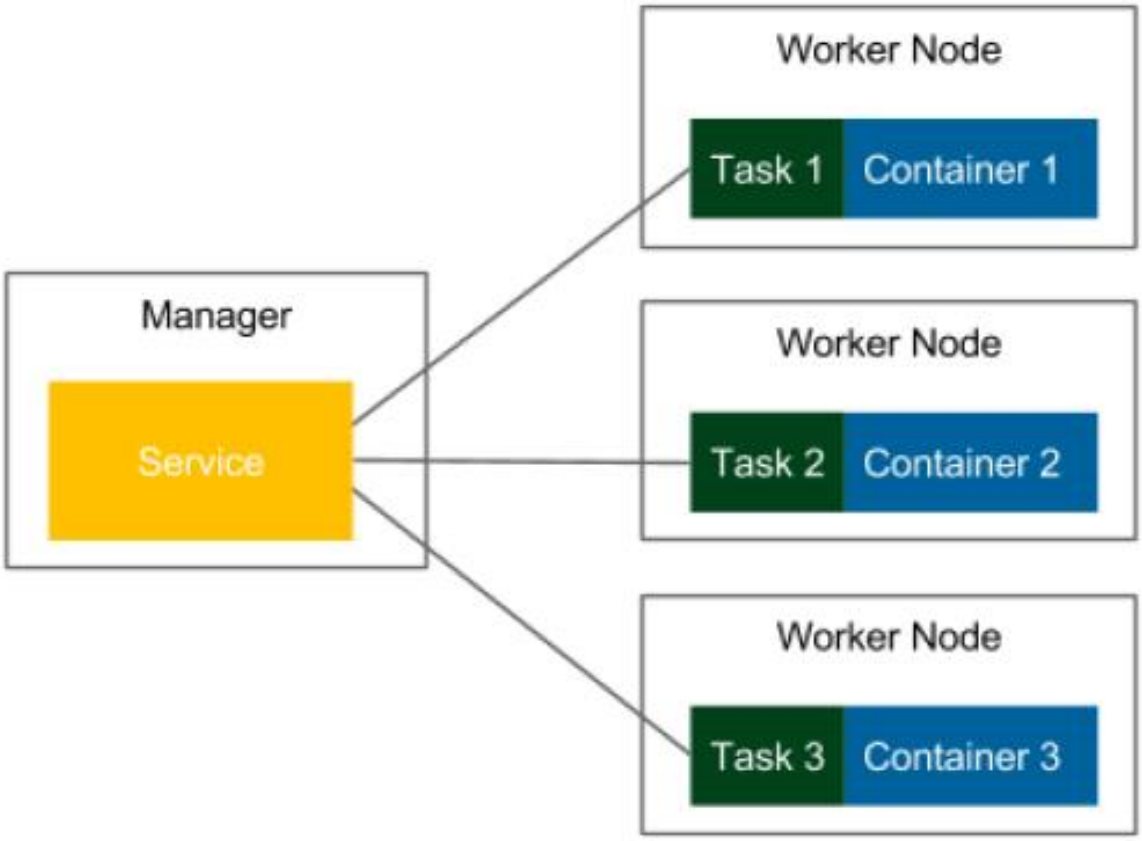


Рис. 3.7. Структура Docker Swarm

3.5. Система Docker

Зрозуміло, не буває ідеальних рішень, і мікросервіси теж мають слабкі місця. Головний недолік мікросервісів криється в самій їхній природі: розподілена система з безліччю незалежних елементів складніша як в

організаційному, так і в архітектурному плані. Ускладнюється управління командами розробки та розгортання, і тут не обійтися без методологій Agile та DevOps. Розподілений доступ до сервісів означає збільшення мережевих затримок та потенційних збоїв, з чим можна боротися шляхом асинхронності та скорочення кількості викликів.

Не варто забувати і про необхідність забезпечувати узгодженість програми: через децентралізацію і модульність можливе виникнення неконсистентності даних, що також призводить до збоїв і недоступності програми в цілому. В цьому випадку варто шукати компроміси між доступністю сервісів та консистентністю.

Через ці та інші подібні обмеження фахівці не рекомендують[6] переходити на мікросервіси просто в гонитві за модою: бізнес повинен мати грамотну і підготовлену команду розробників і адміністраторів, а також достатню інфраструктуру. Крім того, експерти не радять використовувати цю архітектуру без хмарних технологій, а також практик Agile та DevOps. Мережа Docker побудована на моделі Container Network (CNM), яка дозволяє кожному створити власний мережевий диск. Таким чином, контейнери мають доступ до різних типів мереж і можуть бути підключені до кількох мереж одночасно. На додаток до різних мережевих драйверів сторонніх розробників, сам Docker має 4 встановлені:

Міст – у цій мережі контейнери працюють стандартно. Зв'язок встановлюється за допомогою інтерфейсу мосту на хості. Контейнери, що використовують одну мережу, мають власну підмережу, і вони можуть передавати дані один одному стандартно;

• Хост – диск надає контейнеру доступ до області хоста (контейнер бачитиме та використовувати той самий інтерфейс, що й хост);

• Macvlan – надає водієві нові контейнери доступ до створеного хостом інтерфейсу та підінтерфейсу (vlan).

• Overlay – цей драйвер дозволяє Docker працювати в мережі на кількох хостах (зазвичай це кластер Docker Swarm). Контейнери мають власні мережеві

адреси та підмережі, які можуть обмінюватися інформацією безпосередньо, якщо вони фізично знаходяться на різних хостах.

Найбільш часто використовуваний драйвер мосту. Ви можете створити власні мостові мережі за допомогою команди docker network create, вказавши параметр --driver bridge.Наприклад, команда `docker network create --driver bridge --subnet 192.168.100.0/24 --ip-range 192.168.100.0/24 my-bridge-network` створює bridge-мережу з ім'ям `my-bridge-network` і підмережею `192.168.100.0/24`.

Контейнери на спільному хості здатні просити доступ до сервісів, хост-система просто буде відправляти запити до інтерфейсу docker0 в потрібне місце.

Контейнери можуть знаходити свої порти для хоста, на які вони можуть приймати трафік, що приходить із зовнішнього світу. Відчинені порти можуть бути відображені (mapped) на хост-систему або шляхом перебору конкретного порту, або дозволом Docker вибрати випадковий вільний порт. У цих випадках Docker подбає про всіх правилах переадресації та конфігурації iptables для коректної маршрутизації пакетів.

На рис.3.8 зображена схема мережевої архітектури Docker із двома ізольованими мережами контейнерів.

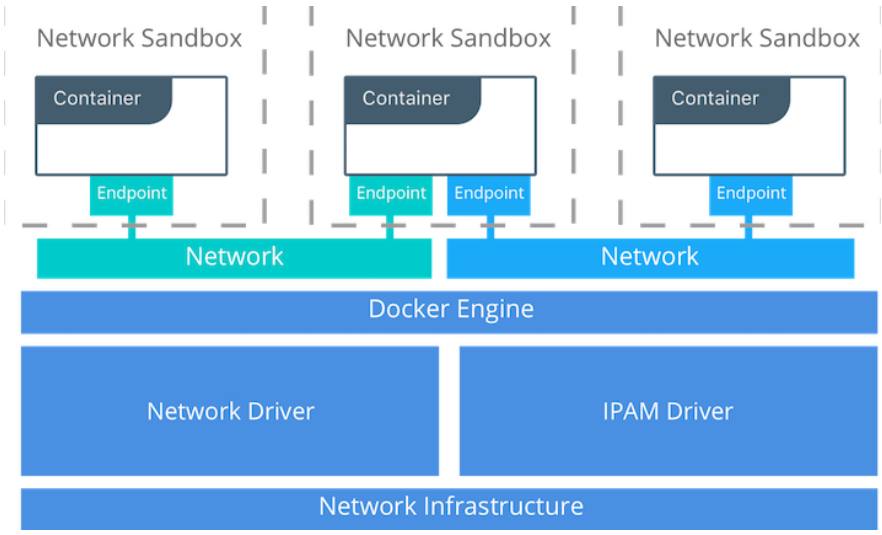


Рис. 3.8. Архітектура мережі Docker

Висновок

Docker розроблений для більш швидкого розгортання додатків. За допомогою Docker можна відокремити застосунок від інфраструктури і розгортати застосунок на будь-якому хості, який підтримує Docker.

Docker допомагає розгортати, швидше тестувати, зменшити час між написанням коду і запуску коду. Docker робить це за допомогою «легкої» платформи контейнерної віртуалізації.

За допомогою Docker можна запускати будь-який застосунок, який безпечно ізольований у контейнері.

РОЗДІЛ 4
СИСТЕМА КОНТЕЙНЕРИЗАЦІЇ МІКРОСЕРВІСІВ

4.1. Розгляд Принципів

Архітектура на базі мікросервісів будується на основі атомарних, модульних сервісів, які виконують обмежену кількість задач.

Отже, базовою вимогою є незалежність сервісів один від одного.

JSON API – це специфікація інтерфейсу взаємодії мій клієнтом і сервером. JSON API призначений для мінімізації як кількості запитів, так і кількості переданих даних між клієнтами та серверами Специфікація JSON API вирішує низку проблем - загальну угоду для всіх. Якщо є загальна угода, то ми не сперечаємось усередині команди — велосипедний сарай задокументований. У нас є угода, з яких матеріалів робити велосипедний сарай та як його фарбувати.

Тепер, коли розробник робить щось неправильно і я це бачу, то не починаю дискусію, а говорю: «Не за JSON API!» та показую на місце у специфікації. Мене ненавидять у компанії, але поступово звикають і JSON API всім почав подобатися. Нові сервіси за замовчуванням ми робимо за цією специфікацією. Ми маємо ключ date, ми готові додавати ключі meta, include. Для фільтрів є зарезервований GET-параметр filters. Ми не сперечаємося, як назвати фільтр — використовуємо цю специфікацію. У ній описано, як робити URL-адресу.

Оскільки ми не сперечаємося, а робимо бізнес завдання, продуктивність розробки вища. У нас специфікації описані, бекенд розробник прочитав, зробив API, ми його прикрутили – замовник щасливий.

Мінуси JSON API

Об'єкт розрісся (date, attributes, included та ін.) - фронтенду треба парсить відповіді: вміти перебирати масиви, ходити по об'єкту і знати, як працює reduce. Не всі розробники-початківці знають ці складні речі. Є бібліотеки серіалізатори/десеріалізатори, можна користуватися ними. Взагалі, це просто робота з даними, але об'єкти великі.

А у бекенда починається біль:

- Контроль вкладеності – include можна залізти дуже далеко;
- Складність запитів до БД вони будуються іноді автоматично, і виходять дуже важкими;
- Безпека - можна залізти в нетрі, особливо якщо підключити бібліотеку;
- Специфікація складно читається. Вона англійською, і це декого відлякало, але поступово всі звикли;
- Не всі бібліотеки реалізують специфікацію добре – це проблема Open Source. За допомогою REST підходу буде вирішено завдання стандартизації інтерфейсу керування ресурсами.

REST – Це англійська абревіатура, яка розшифровується та перекладається як передача стану уявлення. Web-служби, які користуються системою Representational State Transfer, застосовують термін RESTful. Відмінність цього архітектурного стилю від інших полягає в тому, що він не має єдиного стандарту, проте при цьому допустимо використовувати XML, HTTP, JSON та URL.

Representational State Transfer розробили ще в 2000 році, але з того моменту він дуже розвинувся і зараз став одним із найпопулярніших, відсунувши на задній план аналогічні.

Щоб пояснити суть Restful API для чайників, можна подати калькулятор на будь-якому комп'ютері. Коли ми натискаємо на кнопки, бажаючи отримати розрахунки, також починають діяти і приховані функції, які в результаті допомагають отримати результат. А коли сервіс отримує відповідь, він виводить його на екран у вигляді готової цифри у графічному інтерфейсі.

Тут архітектура працює аналогічно. При натисканні на кнопку виконуються різні операції з обробки та передачі інформації. Вони можуть не просто отримувати дані з однієї мережі, а здатні викликати та звертатися до віддалених серверів, щоб взяти потрібне у них.

Як приклад варто навести кнопку Facebook, яка вміє задіяти соцмережу, або відео на Youtube, його теж запускає веб-версія API.

Як працює

- Насамперед варто розібратися, як діє підхід:
- компоненти систем взаємодіють у значно більшому масштабі;
- всі спільні інтерфейси;
- частини можна запроваджувати незалежно одну від іншої;
- є проміжні елементи, які знижують відсоток затримки та посилюють безпеку з'єднання.

4.2. Вживані спеціальні та програмні ресурси

Docker

Docker – це програмна платформа для швидкої розробки, тестування і розгортання додатків. Docker упаковує ПО в стандартизовані блоки, які називаються контейнерами.

Використана версія Docker – 19.03.5.

PHP

PHP — це мова сценаріїв загального призначення, що широко використовується для розробки веб-додатків.

Використана версія PHP - 7.2.26.

Laravel

Laravel MVC (Model View Controller) — це безкоштовний веб-фреймворк з відкритим кодом для розробки веб-додатків з використанням архітектурної моделі. Laravel має вбудовані модулі для роботи з базами даних, HTTP-запитами та сеансами.

Використана версія Laravel - 6.10.1.

PHPSTORM

PhpStorm — це інтегроване кросплатформне середовище розробки PHP.

PhpStorm включає в себе інтелектуальний редактор для PHP, HTML і JavaScript, миттєвий аналіз коду, запобігання помилкам коду та автоматизовані інструменти рефакторингу для PHP і JavaScript. Є повноцінний редактор SQL, який може редагувати результати запитів.

Використана версія PHPSTORM - 2019.3.

MySQL

База даних MySQL — це система управління, тобто підключена база даних. Дані зберігаються в окремих таблицях у зв’язаній базі даних, що забезпечує швидкість і спритність.

Як частину системи MySQL, SQL можна описати як структуровану мову запитів і найпоширенішу стандартну мову, яка використовується для доступу до бази даних.

Використана версія MySQL - 5.7.21.

VirtualBox

VirtualBox — це програмний продукт віртуалізації для різних операційних систем. Тобто програма, яка моделює реальний комп’ютер, що дозволяє користувачеві встановлювати, запускати та використовувати інші операційні системи як зазвичай. Такий комп'ютер на комп'ютері.

Використана версія VirtualBox - 6.1.2.

Прометей

Програмне забезпечення для збору та аналізу показників стану

веб-додатки, тобто оперативна пам’ять, центральний процесор, кількість помилок та інші окремі параметри.

Використана версія Prometheus - 2.15.2.

Графана

Інструмент візуалізації метрики, який дозволяє створювати гнучкі графіки, діаграми та виконувати обчислення на основі метричних даних.

Використана версія Grafana - 6.5.3.

4.3. Контейнеризація мікросервісів

Docker дозволяє упаковувати всі залежні служби в образи, що дозволяє передавати ці образи через мережу і працювати на будь-якому сервері, де встановлено Docker.

Ми вже знаємо, що зображення — це просто читабельний шаблон і з нього створюється контейнер. Кожне зображення складається з кількох рівнів. Docker використовує файлову систему union, щоб об’єднати ці рівні в одне зображення. Одинична файлова система дозволяє файлам і папкам з різних файлових систем (різних гілок) прозоро перекриватися і створює послідовну файлову систему.

Одна з причин необережності Doker – використання таких рівнів. Коли ви змінюєте зображення, наприклад, оновлюєте програму, створюється новий рівень. Так, лише рівень додається або оновлюється, оскільки ви можете підключитися до віртуальної машини, не змінюючи чи перекомпоновуючи весь

образ. І вам не потрібно поширювати повністю нове зображення, лише оновлення, які дозволяють поширювати зображення легше та швидше.

У центрі кожного зображення основне зображення. Наприклад, ubuntu, основний опис Ubuntu або Fedora, основний опис дистрибутива Fedora. Ви також можете використовувати зображення як основу для створення нових зображень. Наприклад, якщо у вас є зображення Apache, ви можете використовувати його як базове зображення для ваших веб-програм.

Ми створимо кілька мікросервісів для реалізації постановки проблеми: api-gateway, users-api, document-api, statistics-api.

API шлюзу надає зовнішнім клієнтам загальний інтерфейс для взаємодії з системою. Крім того, цей шлюз часто виступає як єдина точка доступу для зовнішніх запитів. У цьому випадку шлюз може відповідати за забезпечення безпеки транспортного рівня за допомогою різних каналів безпеки.

До переваг використання підходу api-шлюзу також можна віднести: ізоляцію клієнтів від структури сервісу, єдиний API для клієнтів, автоматичне розпізнавання місцезнаходження необхідної послуги.

Завдання мікросервісу users-api — зберігати інформацію про користувачів.

Функція мікросервісу document-api полягає в наданні інформації про документи.

Завданням мікросервісу statistics-api є узагальнення статистики відвідувань.

Кожен мікросервіс має свою власну базу даних MySQL.

сумнів. 4.1. представлений архітектурною схемою розробленого додатка.

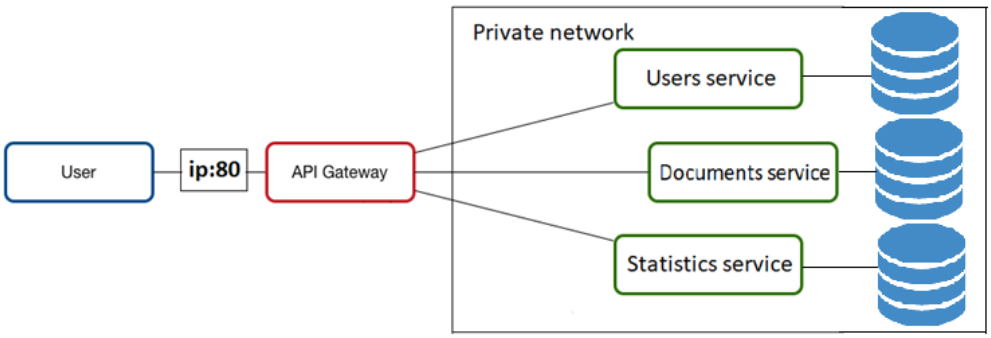


Рис. 4.1. Архітектура додатку

Служба api-шлюзу заснована на веб-сервері nginx, який прослуховує порт 80 (порт HTTP) і отримує запити користувачів (веб-браузер, мобільний додаток тощо). На основі URL-адреси вирішується, до якого мікросервісу буде перенаправлено запит.

URL-адреса, що починається з api / v1 / users - перенаправлено на microservice users-api, api / v1 / documents - перенаправлено на microservice documents-api, api / v1 / statistics - перенаправлено на microservice statistics-api.

Конфігурація веб-сервера знаходиться у файлі default.conf, який аналізується nginx під час запуску служби. Інструкції з налаштування див. у веб-сервері.

Мікросервіс User-api включає веб-сервер nginx і службу php-fpm для перетворення php-скриптів.

На малюнку 4.2 показаний інтерфейс, сумісний з REST, для взаємодії сервісів. Тобто URL-запит GET [app_ip]: 80 / api / v1 / дозволяє користувачам отримати доступ до програми. сумнів. 4.3 наведено приклад відповіді на цей запит.

```
+--------+----------+------------------+------+-------------------------------------------+------------+
| Domain | Method   | URI              | Name | Action                                    | Middleware |
+--------+----------+------------------+------+-------------------------------------------+------------+
|        | GET|HEAD | api/v1/users     |      | App\Http\Controllers\UsersController@index   |            |
|        | GET|HEAD | api/v1/users/{id} |      | App\Http\Controllers\UsersController@getById |            |
+--------+----------+------------------+------+-------------------------------------------+------------+
```

Рис. 4.2. Інтерфейс взаємодії із сервісом users-api

```
{
    "data": [
        {
            "id": 1,
            "name": "magnus.bashirian",
            "email": "gislason.loy@gmail.com",
            "created_at": "2020-01-19 13:22:17",
            "updated_at": null
        },
    ],
    "meta": {
        "container_id": "6ae6484f505a",
        "service_name": "users-api"
    }
}
```


Рис. 4.3. Приклад відповіді на API запит

Зображення збирається за допомогою Dockerfile, в якому є всі інструкції для складання кожного шару зображення. Кожна інструкція додає новий шар до зображення, а це означає, що ви повинні мінімізувати їх кількість, щоб зменшити розмір зображення.

У Додатку В перераховано вміст файлу Dockerfile для служб users-api, document-api, statistics-api.

Оператор FROM вказує, яке базове зображення буде використовуватися, в даному випадку встановлене зображення php. Під цим посібником відбувається встановлення всіх необхідних пакетів.

Оператор CMD вказує, що запущені програмні служби: nginx і php-fpm.

Мікросервіс Document-api включає веб-сервер і службу nginx

На малюнку 4.4. Відображає REST-сумісний інтерфейс для взаємодії зі службою document-api. Тобто за допомогою певних запитів GET можна отримати інформацію про документи.

```
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
| Domain | Method   | URI                      | Name | Action                                              | Middleware |
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
|        | GET|HEAD | api/v1/documents         |      | App\Http\Controllers\DocumentsController@index      |            |
|        | GET|HEAD | api/v1/documents/user/{id} |    | App\Http\Controllers\DocumentsController@getByUserId |            |
|        | GET|HEAD | api/v1/documents/{id}    |      | App\Http\Controllers\DocumentsController@getById    |            |
+--------+----------+--------------------------+------+-----------------------------------------------------+------------+
```

Рис. 4.4. Інтерфейс взаємодії із сервісом documents-api

По аналогії із іншими сервісами, мікросервіс *statistics-api* включає у себе веб-сервер *nginx* і службу *php-fpm* для перетворення php-скриптів.

Код сервіса розміщений у публічному репозиторії за посиланням: https://github.com/zhenia97/statistics-api.

На рисунку 4.5. зображено REST-сумісний інтерфейс взаємодії із сервісом *statistics-api*. Тобто, за допомогою GET запитів можна отримати статистику відвідувань по кожному користувачу.

```
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
| Domain | Method   | URI                       | Name | Action                                              | Middleware |
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
|        | GET|HEAD | api/v1/statistics         |      | App\Http\Controllers\StatisticsController@index     |            |
|        | GET|HEAD | api/v1/statistics/user/{id} |    | App\Http\Controllers\StatisticsController@getByUserId |          |
+--------+----------+---------------------------+------+-----------------------------------------------------+------------+
```

Рис. 4.5. Інтерфейс взаємодії із сервісом statistics-api

За допомогою специфікації JSON API було стандартизовано формат повідомлень для обміну між мікросервісами.

Для усіх API відповідей від сервісів, слідуючи стандарту, було додано заголовок: *Content-Type: application/json; charset=UTF-8.*

Була створена бібліотека «JsonApi» на мові PHP для уніфікації формату повідомлень, що дало змогу спростити підтримку сервісів і покращило модульність коду. Усі повідомлення були стандартизовані у формат виду:

```
[
      'data' => [],
      'meta' => [],
      'errors' => [],
 ]
```

Де, в ключ *data* записується інформація про запитуваний ресурс, *meta* – додаткова, довідкова інформація про ресурс, *errors* – інформація у разі виникнення помилки.

```
public function index(): JsonResponse
{
    $data = new DataObject(
        AppUserModel::all()->toArray()
    );
```

```
        jsonApi->toArray(), 200, [],
        JSON_PRETTY_PRINT
         'container_id' => getenv('HOSTNAME'),
             'service_name' => self::SERVICE_NAME,
         ]);
         $jsonApi = new JsonApi($data, $meta);
         return response()->json(
              'container_id' => getenv('HOSTNAME'),
             'service_name' => self::SERVICE_NAME,

             $jsonApi->toArray(), 200, [], JSON_PRETTY_PRINT
         );
    }
```

Для фіксації змін у коді сервісів було застосовано підхід семантичного версіонування. Даний підхід допомагає попереджувати ситуації із можливою несумісністю служб, особливо коли йде мова про десятки взаємопов'язаних сервісів.

Загальною практикою при реалізації версіонування API є включення номера версії в URL-адресу виклику API-спосібу. Наприклад, `[app_ip]:80/api/v1/users` означає виклик API-спосібу сервісу *users-api* із версією 1.

Версіонування також було застосовано до Docker образів, наприклад, один із перших образів мікросервісу *users-api* був позначений тегом (версією) *users-api:1.0.0.*

Версія образу із тегом *latest*, наприклад, *users-api:latest*, означає що використовується нестабільна версія, яка ще у процесі розробки.

Було проаналізовано тривалість збірки кожного мікросервісного образу. Із табл. 4.1 видно, що сервіс *api-gateway*, має мінімальне значення, це пояснюється тим, що кодова база даного образу мінімальна.

Отже, можна зробити висновок, що тривалість збірки є прямопропорційною величиною до розміру образу, тому для швидкого розгортання та збірки потрібен мінімальний розмір образу.

Таблиця 4.1

Аналіз тривалості збірки образів

Сервіс	Кількість вимірів	Розмір образу (мегабайт)	Середня тривалість збірки (секунд)
users-api	20	204,7	195,4
documents-api	20	201,4	191,8
statistics-api	20	203,1	194,3
api-gateway	20	48,5	54,2

4.4. Здійснення безперебійного доставлення

Система реєстру зображень Docker Hub може автоматично створювати образ із вихідного коду у зовнішньому репозиторії (наприклад, GitHub) і автоматично перенаправляти запакований образ у сховище.

Під час налаштування автоматизованих збірок вам потрібно створити список гілок і етикеток для упаковки. Після того, як нова версія коду увійде в систему контролю версій, DockerHub автоматично почне створювати образ на основі вихідного коду. Після завершення встановлення зображення поміщається в репозиторій із призначеним тегом.

Також є можливість визначити значення змінних середовища, які використовуються в процесі складання, що дозволяє динамічно виконувати певні інструкції під час установки. Якщо установка завершена неправильно, процес можна перезапустити.

сумнів. 4.6 показує налаштовану конфігурацію автоматизованих колекцій зображень.

Завдяки опції Build Caching час встановлення зменшується за рахунок захисту проміжних шарів зображення. Кожен шар зображення поміщається в репозиторій DockerHub, а наступна структура використовує шар сховища замість створення нового, якщо хеш-сума поточного шару не змінилася.

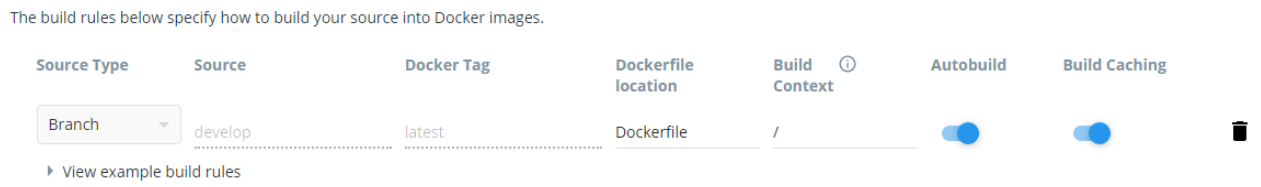


Рис. 4.6. Конфігурація автоматизованих збірок

Підхід на основі безперервної доставки дав змогу автоматизовано створювати версії Docker образів без особистого запуску збірки.

4.5. Гарантування стійкості до відмов додатку

Відмовні кластери широко використовуються для підтримки важливих баз даних, мережевої пам’яті, бізнес-пропозицій і систем обслуговування клієнтів, таких як веб-сайти електронної комерції.

Ось концепції, які використовуються при проектуванні відмовостійкої інфраструктури.

Defect Tolerance (FT) - здатність продовжувати роботу після відмови будь-якого елемента системи.

Кластер - група серверів (блоків), з'єднаних між собою каналами зв'язку.

Безперебійне обслуговування клієнтів можливе лише в будь-який час, якщо є точна копія сервера (фізичного чи віртуального), на якому працює сервіс. Якщо зробити копію після апаратного збою, це займе час, тобто буде перерва в наданні послуг. Крім того, після аварії неможливо буде отримати вміст оперативної пам’яті з проблемної машини, а отже, дані там будуть втрачені.

Є два способи реалізації СА: апаратний і програмний. Поговоримо докладніше про кожну з них.

Апаратний метод — це «розділений» сервер: усі компоненти дублюються, а обчислення виконуються одночасно та незалежно. Вузол відповідає, серед іншого, за синхронність, яка порівнює результати половин. Якщо є розбіжності,

шукається причина і намагаються виявити помилку. Якщо помилка не виправлена, несправний модуль вимикається.

Нещодавно у Хабре була стаття про апаратні сервери ЦС. Виробник, описаний в матеріалі, гарантує, що річна перерва складе трохи більше 32 секунд. Тому для досягнення таких результатів необхідно купувати обладнання. Російський партнер Stratus повідомив, що вартість двопроцесорного СА-сервера для кожного синхронізованого модуля становить близько 160 000 доларів в залежності від конфігурації. Всього на кластер потрібно 1 600 000 доларів.

Програмний метод.

На момент написання статті найпопулярнішим інструментом для розгортання кластера сталого розвитку був vSphere від VMware. Технологія сталого доступу називається Fault Tolerance.

На відміну від апаратного методу, використання цієї опції має обмеження. Основними з них є:

Фізичний хост повинен мати процесор:

Архітектура Intel Sandy Bridge (або новіша). Avoton не підтримується.

AMD Bulldozer (або пізніше).

Машини, які використовують відмовостійкість, повинні бути інтегровані в 10-гігабітну мережу з низькою затримкою. VMware рекомендує виділену мережу.

• ВМ не повинна мати більше 4 віртуальних процесорів.

• Не більше 8 віртуальних процесорів на кожен фізичний хост.

• Не більше 4 віртуальних машин для кожного фізичного хоста.

• Не можна використовувати знімки віртуальної машини.

• Memory vMotion не можна використовувати. сумнів. 4.7 показана архітектура проектного кластера. Єдиною точкою доступу до програми є служба api-шлюзу: відкритий сервер nginx

порт 80. Тобто програма знаходиться в приватній мережі, яка недоступна ззовні, за винятком порту 80 (HTTP). У свою чергу, копії трьох мікросервісів для кожного сервісу знаходяться в приватній мережі.

Три балансувальники навантаження є єдиною точкою доступу для мікросервісів: запит надходить до балансувальника навантаження, а потім пересилається на будь-який, зазвичай менш завантажений, екземпляр.

Зв'язок між мікросервісами здійснюється через імена DNS, призначені кожному балансировщику навантаження, тобто якщо служба users-api запитує службу document-api, ви повинні використовувати ім'я DNS документів служби балансування навантаження-api.

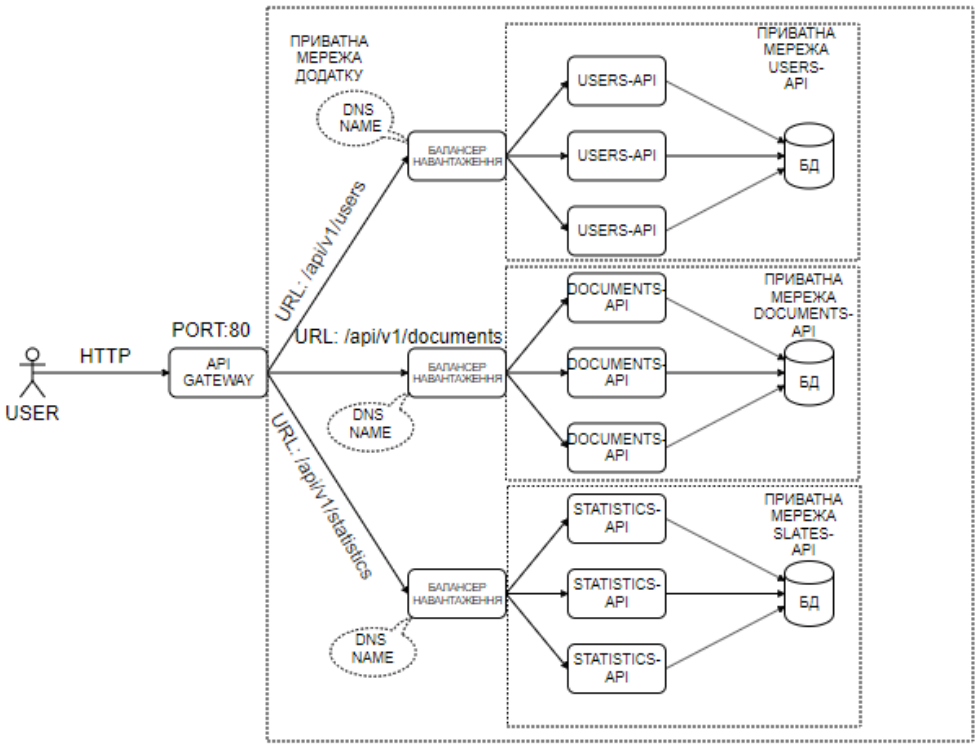


Рис. 4.7. Проектована кластерна інфраструктура

Використовуючи технологію Docker Swarm, яка вбудована в Docker, можна створити масштабований кластер на базі декількох хостів.

Використовуючи VirtualBox, було створено чотири віртуальні машині (рис. 4.8), які будуть об’єднані в кластер. Машина із назвою *default* буде виступати у якості менеджера, тобто головної ноди, яка буде керувати кластером і збалансовувати навантаження між іншими нодами.

```
C:\Users\Evgeniy>docker-machine.exe ls
NAME             ACTIVE   DRIVER       STATE     URL                         SWARM   DOCKER     ERRORS
default          *        virtualbox   Running   tcp://192.168.99.100:2376           v19.03.5
swarm-worker     -        virtualbox   Stopped                                       Unknown
swarm-worker-2   -        virtualbox   Stopped                                       Unknown
swarm-worker-3   -        virtualbox   Stopped                                       Unknown
```

Рис. 4.8. Список віртуальних машин

Запуск кластера відбувається за допомогою команди, де `swarm-docker-compose.yml` – конфігураційний файл кластеру:

```
docker stack deploy --compose-file swarm-docker-compose.yml app
```

Тривалість розгортання кластеру: 7.31 секунди.

У додатку А приведений вміст конфігураційного файлу кластера.

Docker Swarm автоматично займається розподіленням сервісів, і їх екземплярів у вигляді контейнерів, по чотирьом запущеним віртуальним машинам, які є вузлами кластеру.

Із виводу команди (рис. 4.9) видно, що сервіси *users-api, documents-api, statistics-api* бути створені у кількості три контейнери на кожен сервіс. Сервіс api-gateway та БД кожного сервіса були створені одиничними контейнерами.

```
docker@default: $ docker service ls
ID              NAME                MODE          REPLICAS  IMAGE                                  PORTS
r8mazuiasx87    app_api-gateway     replicated    1/1       zhenia97chap/api-gateway:latest        *:80->80/tcp
qk93dfs6qpps    app_documents-api   replicated    3/3       zhenia97chap/documents-api:latest
ooapjb9mdzbj    app_documents-db    replicated    1/1       mysql:5.7.21
cidkfcovzlnp    app_statistics-api  replicated    3/3       zhenia97chap/statistics-api:latest
y65uludclat8    app_statistics-db   replicated    1/1       mysql:5.7.21
nxrpk81hhzcl    app_users-api       replicated    3/3       zhenia97chap/users-api:latest
biiqm82dej2h    app_users-db        replicated    1/1       mysql:5.7.21
```

Рис. 4.9. Список сервісів кластеру

Набір масштабується за допомогою команди:

масштаб служби docker app_users-api = 5

Час масштабування сервісу: 3,89 секунди.

У цьому випадку сервіс user-api вимірювався від трьох до п'яти контейнерів.

Коли вам потрібно зупинити запущений вузол, ви можете використовувати вузол дренажу Swarm, наприклад, для обслуговування або просто видалити його з кластера. Перехід зі статусу вузла в режим активного розвантаження змушує блок керування переміщувати всі запущені завдання / контейнери з дренажного вузла до інших активних вузлів, зупиняючи такі контейнери на дренажному вузлі та використовуючи їх на активних вузлах.

Наприклад, щоб перевести вузол робочого стада в режим обслуговування, необхідно ввести команду:

Оновлення докерного вузла -- доступність drain swarm-worker

Наприклад, наступна команда змінює вузол стадного працівника з «запущеного» на «контролюючий»: вузол докера заохочує ройового працівника.

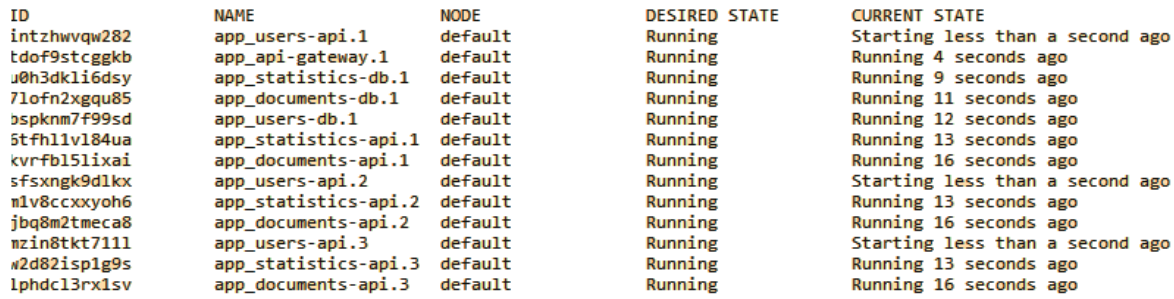
Docker сам по собі гарантує толерантність до помилок кластера. Частково це пов’язано з тим, що кластер може одночасно керувати декількома керуючими вузлами, що в будь-який момент може замінити невдалого лідера. Використовується так званий алгоритм підтримки розподіленого консенсусу - Raft.

Пліт реалізовано на наборі слабо зв’язаних вузлів, кожен з яких управляє кінцевим автоматом, тому кожен вузол гарантовано відповідає іншим вузлам.

Відмовостійкість сервісів, тобто зразків контейнерів цих сервісів, забезпечується тиражуванням: контейнери одного сервісу можуть бути розміщені на кількох вузлах.

Наприклад, використовуйте docker-machine.exe stop [node_name], щоб вимкнути всі вузли, крім типових. Після зупинки вузлів усі контейнери перемістилися до вузла, що залишився – як показано на рисунку 4.10.

Тривалість міграції на доступну ноду: 12.67 секунд.



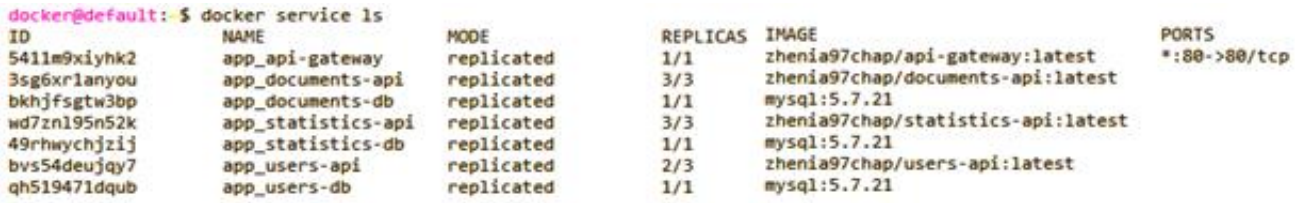
ID	NAME	NODE	DESIRED STATE	CURRENT STATE
intzhwvqw282	app_users-api.1	default	Running	Starting less than a second ago
tdof9stcggkb	app_api-gateway.1	default	Running	Running 4 seconds ago
u0h3dkli6dsy	app_statistics-db.1	default	Running	Running 9 seconds ago
71ofn2xgqu85	app_documents-db.1	default	Running	Running 11 seconds ago
bspknm7f99sd	app_users-db.1	default	Running	Running 12 seconds ago
5tfhl1v184ua	app_statistics-api.1	default	Running	Running 13 seconds ago
kvrfbl5lixai	app_documents-api.1	default	Running	Running 16 seconds ago
sfsxngk9dlkx	app_users-api.2	default	Running	Starting less than a second ago
m1v8ccxxyoh6	app_statistics-api.2	default	Running	Running 13 seconds ago
jbq8m2tmeca8	app_documents-api.2	default	Running	Running 16 seconds ago
mzin8tkt7111	app_users-api.3	default	Running	Starting less than a second ago
w2d82isp1g9s	app_statistics-api.3	default	Running	Running 13 seconds ago
lphdcl3rx1sv	app_documents-api.3	default	Running	Running 16 seconds ago

Рис. 4.10. Забезпечення відмовостійкості сервісів

Для перевірки відмовостійкості окремо взятого контейнера будемо використовувати симуляцію нештатної зупинки контейнера. Для зупинки контейнера використовується команда *stop*, наприклад: `docker stop 783ee96f6500`, де, `783ee96f6500` – унікальний ідентифікатор контейнера.

На рис. 4.11 видно, що сервіс *users-api*, включає тільки два контейнера.

Тривалість запуску нового контейнера: 4.3 с.



```
docker@default: $ docker service ls
ID              NAME                MODE          REPLICAS  IMAGE                                PORTS
5411m9xiyhk2    app_api-gateway     replicated    1/1       zhenia97chap/api-gateway:latest      *:80->80/tcp
3sg6xr1anyou    app_documents-api   replicated    3/3       zhenia97chap/documents-api:latest
bkhjfsgtw3bp    app_documents-db    replicated    1/1       mysql:5.7.21
wd7zn195n52k    app_statistics-api  replicated    3/3       zhenia97chap/statistics-api:latest
49rhwychjzij    app_statistics-db   replicated    1/1       mysql:5.7.21
bvs54deujqy7    app_users-api       replicated    2/3       zhenia97chap/users-api:latest
qh519471dqub    app_users-db        replicated    1/1       mysql:5.7.21
```

Рис. 4.11. Аварійна зупинка контейнера

Проведемо дослідження тривалості затримки відповіді для REST-ресурсу: `api/v1/users/[user_id]`.

У даному випадку запит перенаправляється на сервіс *users-api*, який взаємодіє і отримує дані із сервісів *documents-api* і *statistics-api*.

У табл. 4.2 приведено аналіз часу очікування відповіді від додатку. Загалом час очікування склав 1,09 с. При чому, найшвидше запит пройшов за допомогою сервіс *api gateway*, а найдовше – *users-api*.

Отже, можна зробити висновок, що на маршрутизацію запиту витрачається найменше часу.

Таблиця 4.2

Аналіз часу очікування відповіді

Сервіс	Тривалість, секунди
Api Gateway	0,19
Users-api	0,38
Documents-api	0,22
Statistics-api	0,3
Усього	1,09

4.6. Спостереження за системою

Моніторинг програм і програмних серверів є важливою частиною будь-якої інфраструктури. Контейнери, сервери, використання ЦП, споживання пам’яті, відновлення диска тощо. Необхідний постійний моніторинг ситуації. Додатково потрібно повідомляти розробників, коли доступна пам’ять на сервері закінчується або програма перестає відповідати. запобігти проблемам, які можуть призвести до збою сервера в довгостроковій перспективі.

Prometheus — це система моніторингу різних систем і сервісів, які системні адміністратори можуть налаштувати для збору інформації про поточні налаштування та отримання інших повідомлень (наприклад, сповіщень про обмеження перевантаження ЦП).

За допомогою Prometheus ми опишемо необхідні групи вимірювань:

• основні показники верстата;

• розміри контейнера;

• розміри шлюзу api (єдина точка доступу для програми).

Нижче наведено вміст файлу конфігурації Prometheus. Цільовий блок описує, де будуть отримані метрики, у цьому випадку контейнери експортерів показників. Оператор scrape_interval вказує інтервал, у якому будуть запитуватися експортери.

```
global:

scrape_configs:
  - job_name: 'node-exporter'
    scrape_interval: 10s
    static_configs:
      - targets: ['node-exporter:9100']
  scrape_interval: 5s
  evaluation_interval: 5s
  external_labels:
    monitor: 'prometheus-grafana-exporter'



  - job_name: 'containers-exporter'
    scrape_interval: 10s
  - job_name: 'nginx-exporter'
    static_configs:
      - targets: ['cadvisor:8080']

    scrape_interval: 10s
    static_configs:
      - targets: ['nginx-exporter:9113']
```

Експортер — це програмне забезпечення, яке бере існуючі вимірювання зі сторонньої системи та експортує їх на сервер Prometheus у зрозумілому форматі сумнів.

На малюнку 4.12 наведено стандартний графік використання хоста ЦП. На графіку видно, що навантаження на процесор залишається стабільною на рівні ~ 12%. Це пов’язано з тим, що система часто не працює через відсутність трафіку користувачів.

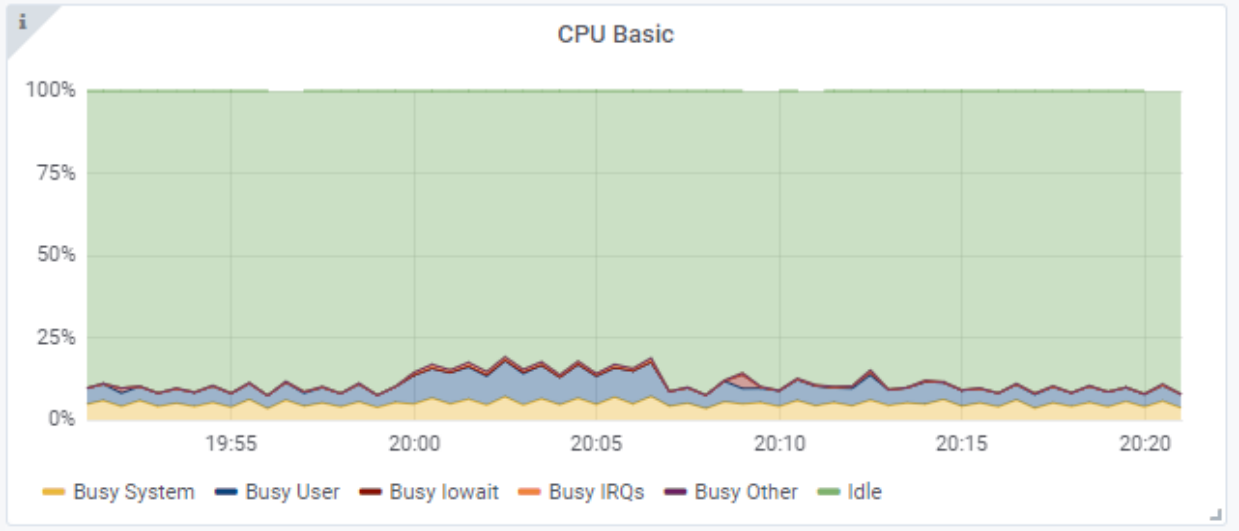


Рис. 4.12. Графік використання CPU хост машиною

На рис. 4.13 зображено графік використання CPU кожним запущеним контейнером.

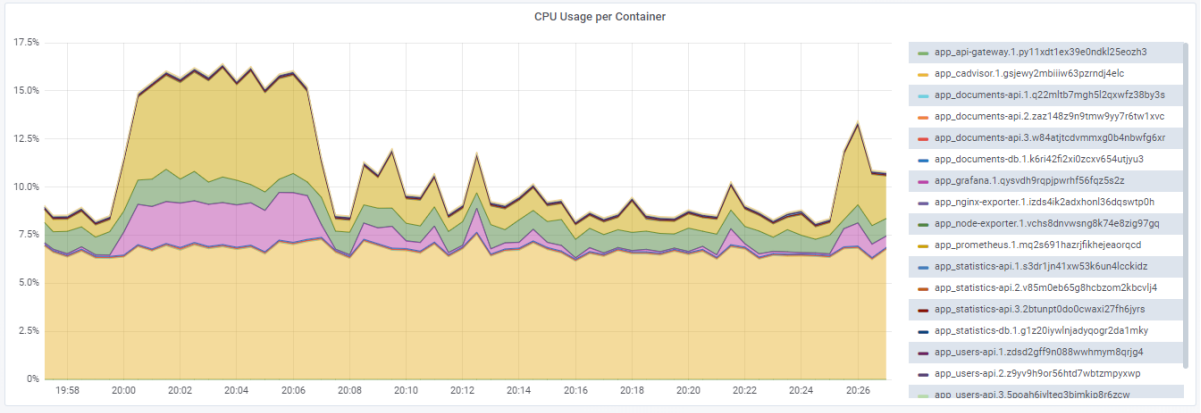


Рис. 4.13. Графік використання CPU контейнерами

Використовуючи даний графік можна побачити, який із контейнерів найбільше навантажує процесор і прийняти міри.

Наприклад, додати кілька екземплярів сервісу на інший хост.

Запит на отримання метрик для рис. 4.13 наведений нижче:

```
sum(rate(container_cpu_usage_seconds_total{name=~".+"}[$interval])) by (name) * 100
```

На рис. 4.14 зображено таблицю використання RAM контейнерами на якій показано що система найбільше нагружена моніторинговими сервісами, які сканують метрики хоста, а інші сервіси не навантажують систему, так як мережевого трафіку немає.

Запит для отримання метрик використання RAM контейнерами виглядає так:

```
container_memory_usage_bytes{name=~".+"}
```

Metric	Current
app_statistics-api.2.v85m0eb65g8hcbzom2kbcvlj4	12.60 MB
app_statistics-api.1.s3dr1jn41xw53k6un4lcckidz	13.00 MB
app_prometheus.1.mq2s691hazrjfikhejeaorqcd	94.88 MB
app_node-exporter.1.vchs8dnvwsng8k74e8zig97gq	12.75 MB
app_nginx-exporter.1.izds4ik2adxhonl36dqswtp0h	6.97 MB
app_grafana.1.qysvdh9rqpjpwrhf56fqz5s2z	48.51 MB
app_documents-db.1.k6ri42fi2xi0zcxv654utjyu3	77.32 MB
app_documents-api.3.w84atjtcdvmmxg0b4nbwfg6xr	14.75 MB
app_documents-api.2.zaz148z9n9tmw9yy7r6tw1xvc	12.72 MB
app_documents-api.1.q22mltb7mgh5l2qxwfz38by3s	13.15 MB
app_cadvisor.1.gsjewy2mbiiiw63pzrndj4elc	49.65 MB
app_api-gateway.1.py11xdt1ex39e0ndkl25eozh3	2.34 MB

Рис. 4.14. Використання RAM контейнерами

На рис. 4.15 зображено графік із інформацією про кількість прийнятих з’єднань сервісом *api-gateway* за визначений період часу.

Рис. 4.15. Кількість прийнятих з’єднань nginx

Висновок

Docker Swarm дозволяє створювати розширювану та швидку інфраструктуру кластера.

Було створено і підключено чотири мікросервіси.

Розроблено платформу DockerHub на якій реалізовано концепцію безперебійної доставки, що значно спрощує та усуває програмні збої.

Було проаналізовано видимість системи та встановлено, що кластер візуально посилений за рахунок видимості університету.

ВИСНОВКИ

У ході виконання дипломного проекту було реалізовано і спроектовано систему контейнеризації мікросервісів.

Було проаналізовано складові та компоненти веб-інфраструктури, принципи взаємодії клієнт-сервер, визначено базові технології і служби за допомогою яких організовано роботу веб-додатків.

У цій магістерській роботі було вивчено поняття масштабованої розподіленої веб архітектури, її типи, принципи проектування та інфраструктурні вимоги. Зокрема, було досліджено процес міграції застарілого ПЗ на мікросервісну архітектуру при побудові масштабованого веб-додатка на базі системи відпімізації логістики. По-перше, було детально розібрано питання переходу від моноліту до мікросервісної архітектури, проаналізовано складності та специфіку даного процесу. У ході роботи були проведені наступні роботи:

● Порівняльний аналіз різних типів архітектур - моноліт, SOA та мікросервіси

● Порівняння сильних та слабких сторін монолітної архітектури та мікросервісів

● Порівняльний аналіз монолітної архітектури та мікросервісів з урахуванням основних критеріїв проектування системи

● Порівняльний аналіз монолітної архітектури та мікросервісів на основі стану проекту та команди

● Розглянуто основні спосіби міграції від моноліту до мікросервісів з урахуванням переваг та недоліків По-друге, були вивчені сучасні інструменти для вирішення проблеми маршрутизації транспортних засобів (VRP). У ході роботи були проведені наступні роботи:

● Аналіз інструментів OR tools, Jsprit, OptsPlanner розв'язання задачі VRP за основними критеріями оцінки ПЗ

● Порівняльний інструмент OR tools, Jsprit, OptsPlanner на підставі можливостей вирішення задачі VRP , черга повідомлень та система оркестрації та відповідні інструменти. У ході роботи були проведені наступні роботи:

● Порівняльний аналіз черги повідомлень (MQ) та REST API за основними критеріями оцінки ПЗ

● Порівняльний аналіз брокер повідомлень Kafka та RabbitMQ з урахуванням основних критеріїв проектування системи

● Аналіз систем оркестрації контейнерів Kubernetes, Docker Swarm та Apache Mesos 75 У заключній стадії роботи було реалізовано мікросервіс планування маршрутів у системі оптимізації транспортної логістики. У ході роботи були проведені наступні роботи:

● Піднято мікросервіс планування з використанням передових технологій (RabbitMQ, Docker, Kubernetes), який працює незалежно від решти системи, що робить його більш гнучким та доступним

● Повністю розділений процес розгортання, тестування та розробки

● Реалізована можливість керувати та масштабувати мікросервіс незалежно від інших частин системи

● Для планування маршрутів впроваджено інструмент Jsprit, який зробив процес планування швидшим та якіснішим. Отримані результати в ході цієї магістерської роботи мають науково-практичну цінність у процесі міграції від застарілого додатка до масштабованої розподіленої веб архітектури, які можуть бути використані на виробництві для вирішення різноманітних транспортних завдань.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ

1. Микросервисные паттерны проектирования [Electronic resource]. Access mode: https://habr.com/ru/company/piter/blog/275633/ (lastaccess:22.11.21).

2. Розгортання монолітного додатку [Electronic resource]. Access mode: https://aws.amazon.com/ru/getting-started/container-microservices-tutorial/module-two/ (lastaccess:20.11.21).

3. Ньюмен С. Создание микросервисов. — СПб.: Питер, 2016. — 304 с.: ил. — (Серия «Бестселлеры O'Reilly»).

4. Шаблон отсеков [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/bulkhead (lastaccess:07.11.21).

5. Шаблон подавления [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/strangler (lastaccess:08.11.21).

6. Шаблон расширения [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/azure/architecture/patterns/sidecar (lastaccess:08.11.21).

7. Взаимодействие в архитектуре микрослужб [Electronic resource]. Access mode: https://docs.microsoft.com/ru-ru/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture (lastaccess:14.12.21).

8. Хорсдал К. Микросервисы на платформе .NET. — СПб.: Питер, 2018. — 352 с.: ил. — (Серия «Для профессионалов»).

9. Microservice Architecture and Design Patterns for Microservices [Electronic resource]. Access mode: https://dzone.com/articles/microservice-architecture-and-design-patterns-for (lastaccess:10.12.21).

10. Экосистема Docker: обнаружение сервисов (Service Discovery) и распределённые хранилища конфигураций (Distributed Configuration Stores) [Electronic resource]. Access mode:

https://www.digitalocean.com/community/tutorials/docker-service-discovery-distributed-configuration-stores-ru (lastaccess:11.12.21).

11. Сравнительный анализ форматов обмена данными, используемых в приложениях с клиент-серверной архитектурой [Electronic resource]. Access mode: https://www.fundamental-research.ru/ru/article/view?id=38464 (lastaccess:12.12.21).

12. Непрерывная интеграция [Electronic resource]. Access mode: https://ru.wikipedia.org/wiki/Непрерывная_интеграция (lastaccess:08.13.21).

13. Выбор стратегии деплоя микросервисов [Electronic resource]. Access mode: https://bool.dev/blog/detail/vybor-strategii-deploya-mikroservisov (lastaccess:02.12.21).

14. Ричардсон Крис Микросервисы. Паттерны разработки и рефакторинга. — СПб.: Питер, 2019. — 544 с.: ил. — (Серия «Библиотека программиста»).

15. Кластер серверов микросервисов [Electronic resource]. Access mode: https://www.stekspb.ru/autsorsing-it-infrastruktury/it-glossary/server-cluster/ (lastaccess:22.13.21).

16. Основы мониторинга и сбора метрик [Electronic resource]. Access mode: https://www.8host.com/blog/osnovy-monitoringa-i-sbora-metrik/ (lastaccess:10.14.21).

17. Docker и технология контейнеров Linux [Electronic resource]. Access mode: https://vps.ua/blog/docker-and-linux-containers/ (lastaccess:13.18.21).

18. Технология контейнеризации [Electronic resource]. Access mode: https://www.cloud4y.ru/about/news/obzor-tekhnologii-konteynerizatsii/ (lastaccess:20.19.21).

19. Моуэт Э. Использование Docker / пер. с англ. А. В. Снастина; науч. ред. А. А. Маркелов. – М.: ДМК Пресс, 2017. – 354 с.: ил.

20. About images, containers, and storage drivers [Electronic resource]. Access mode: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers (lastaccess:22.12.21).

21. Основы Kubernetes [Electronic resource]. Access mode: https://habr.com/ru/post/258443/ (lastaccess:21.13.21).

22. Взаимодействие docker контейнеров [Electronic resource]. Access mode: https://dotsandbrackets.com/communication-between-docker-containers-ru/ (lastaccess:21.13.21).

23. Экосистема Docker: сетевое взаимодействие [Electronic resource]. Access mode: https://www.digitalocean.com/community/tutorials/docker-ru-992094e0-5e33-49a5-b30f-f9bfa371aeab (lastaccess:21.11.21).

Застосунок А

Код конфігураційного файлу кластера Docker Swarm

```
version: "3.7"

services:
  api-gateway:
    image: zhenia97chap/api-gateway:latest
     depends_on:
    ports:
      - 80:80

      replicas: 3
    depends_on:
      - users-db

  documents-api:
    image: zhenia97chap/documents-api:latest
    deploy:
      - users-api
      - documents-api
      - statistics-api
deploy:
      replicas: 1



    depends_on:
      - statistics-db
      replicas: 3
    depends_on:
      - documents-db


  users-api:
    image: zhenia97chap/users-api:latest
    deploy:


    environment:
       statistics-api:
    image: zhenia97chap/statistics-api:latest
    deploy:
      replicas: 3

      MYSQL_USER: root
  users-db:
    image: mysql:5.7.21
    deploy:
```

```
      replicas: 1
      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: users-api
    volumes:
      - db_users_data:/var/lib/mysql



  documents-db:

      MYSQL_ROOT_PASSWORD: password
      MYSQL_DATABASE: documents-api
    volumes:
      - db_documents_data:/var/lib/mysql
image: mysql:5.7.21
    deploy:
      replicas: 1

      MYSQL_DATABASE: statistics-api
    volumes:
      - db_statistics_data:/var/lib/mysql
       environment:
      MYSQL_USER: root

      MYSQL_USER: root
      MYSQL_ROOT_PASSWORD: password


  statistics-db:
    image: mysql:5.7.21
    deploy:
      replicas: 1
- db_statistics_data:/var/lib/mysql
       environment:
      MYSQL_USER: root

      MYSQL_USER: root
      MYSQL_ROOT_PASSWORD: password

    environment:

      placement:
        constraints:
          - node.role == manager
       prometheus:
    image: prom/prometheus:v2.15.2
    deploy:

    ports:
    volumes:
```

```
    - ./prometheus/:/etc/prometheus/
    - prometheus_data:/prometheus
  command:
    - '--storage.tsdb.retention=200h'
    - '--web.enable-lifecycle'

    - 9090:9090



node-exporter:
  image: prom/node-exporter:v0.18.1
    - '--path.sysfs=/host/sys'
    - '--collector.filesystem.ignored-mount-points=^/(sys|proc|dev|host|etc)($$|/)'
  user: root
    - '--web.console.templates=/etc/prometheus/consoles'
    - '--config.file=/etc/prometheus/prometheus.yml'
    - '--storage.tsdb.path=/prometheus'
    - '--web.console.libraries=/etc/prometheus/console_libraries'

  volumes:
    - /proc:/host/proc:ro
    - /sys:/host/sys:ro
    - /:/rootfs:ro
  command:
    - '--path.procfs=/host/proc'


  logging:
    driver: "json-file"
    options:
      max-size: "5m"
nginx-exporter:
  image: nginx/nginx-prometheus-exporter:0.5.0
  environment:



  image: google/cadvisor:latest
  volumes:
    - /:/rootfs:ro
    - /var/run:/var/
     run:rw
    - TELEMETRY_PATH=/metrics
    - NGINX_RETRIES=10
       - SCRAPE_URI=http://192.168.99.100/nginx_status
    - /sys:/sys:ro
    - /var/lib/docker/:/var/lib/docker:ro
cadvisor:
```

```
      - GF_SECURITY_ADMIN_PASSWORD=admin
      - GF_USERS_ALLOW_SIGN_UP=false
    ports:
      - GF_SECURITY_ADMIN_USER=admin

      - 3000:3000

volumes:
  db_users_data:
    driver: local
  db_documents_data:
    driver: local
    db_statistics_data:
    driver: local
  prometheus_data:
      - grafana_data:/var/lib/grafana
    environment:

    driver: local
  grafana_data:
    driver: local
       db_users_data:
    driver: local
  db_documents_data
```

Застосунок Б

Вміст Dockerfile мікросервісів users-api, documents-api, statistics-api

```
FROM php:7.2-fpm

RUN apt-get update \

    && apt-get -y install git \

    && apt-get -y install zip \

    && apt-get -y install procps \

    && apt-get -y install htop \

    && apt-get -y install nano \

    && apt-get -y install supervisor \


    && apt-get -y install net-tools \

    && apt-get -y install nginx


RUN docker-php-ext-install pdo_mysql


COPY docker/nginx/conf.d/default.conf /etc/nginx/conf.d

COPY . /var/www

COPY docker/supervisor/conf.d /etc/supervisor/conf.d/


RUN chown -R www-data:www-data /var/www

RUN rm /etc/nginx/sites-available/default /etc/nginx/sites-enabled/default


WORKDIR /var/www


RUN composer install -n --prefer-dist --ignore-platform-reqs


RUN mv .env.example .env

RUN php artisan key:generate

RUN curl -sS https://getcomposer.org/installer | php -- --install-dir=/usr/local/bin --filename=composer


CMD ["/usr/bin/supervisord", "-n"]
```

Застосунок В

Конфігурація веб-сервера сервісу api-gateway

```
server {
    listen 80;
    index index.php index.html;
    root /var/www/public;


    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME ${document_root}/index.php;
    fastcgi_param PATH_INFO $fastcgi_path_info;
access_log /var/log/nginx/access.log;
    error_log /var/log/nginx/error.log;

    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_index index.php;



    location / {

        rewrite ^(.*)/$ $1 permanent;
        fastcgi_pass users-api:9000;
    }
return 200 'App is alive';
        add_header Content-Type text/plain;
    }

    location /api/v1/users {


    location /api/v1/statistics {
        rewrite ^(.*)/$ $1 permanent;
        fastcgi_pass statistics-api:9000;
    }
    location /api/v1/documents {
        rewrite ^(.*)/$ $1 permanent;
        fastcgi_pass documents-api:9000;
    }

}
```