

## Лабораторна робота №3.

**Мета роботи:** вивчити структуру мікросервісного додатку та принцип комунікації сервісів у ньому. Створити власний примітивний мікросервіс.

**Задача:** на основі прикладу, описаного у Лабораторній роботі №2, створити власний мікросервіс. Необхідно розгорнути кілька мікросервісів і показати, що вони між собою взаємодіють. Налаштувати автоматизоване розгортання всіх контейнерів, продемонструвати механізм **горизонтального масштабування**.

**Завдання**

- ☒ Додати до прикладу, описаного у попередній роботі, власний мікросервіс.
- ☒ Підготувати опис кроків, які було зроблено при підключенні власного сервісу.
- ☒ Підключити інфраструктурні сервіси (edge, discovery, and so on)
- ☒ Протестувати роботу власного сервісу та проілюструвати це. Підняти декілька інстансів одного сервісу, показати механізм горизонтального масштабування і пояснити механізми балансування.
- ☒ Вивчити основні підходи до мікросервісної архітектури.

**Хід роботи**

Для тестування було створено простий мікросервіс – калькулятор, який вираховує вираз переданий по GET запиту.

```
@app.get("/calculate")
def get_records(expression: str = '5+5/5-3'):
    print(f"calculating {expression}")
    return {
        "host": myhost,
        "result": eval(expression)
    }
```

Також було додано loadbalancer який базується на nginx:

```
user nginx;
events {
    worker_connections 1000;
}
http {
    server {
        listen 8080;
        location / {
            proxy_pass http://calculator:81;
        }
    }
}
```

Даний конфіг описує правила розподілу трафіку по копіям мікросервісу. Nginx використовує циклічну реалізацію для розподілу трафіку.

Крім того, було додано ще один сервіс, котрий отримує запит від користувача і передає його до load balancer.

```
@app.get("/calculate")
async def calculate(expression: str = '5+5/5-3'):
    async with aiohttp.ClientSession() as session:
        async with session.get(f'http://nginx:8080/calculate',
                                params={'expression': expression}) as resp:
            return await resp.json()
```

Варто зазначити, що в docker-compose calculator не прокидує порти на ззовні, вони доступні лише через внутрішню dns docker, яка використовується Nginx.






```
services:
  api:
    build: api
    ports:
      - "80:80"
    depends_on:
      - nginx

  calculator:
    build: calculator
    expose:
      - "81"
    deploy:
      mode: replicated
      replicas: 2

  nginx:
    build: nginx
    ports:
      - 8080:8080
    depends_on:
      - calculator
```

А також вказано тип deploy для створення необхідних копій.

При docker-compose up буде піднято два контейнера calculator

<input type="checkbox"/>	Name	Image	Status	CPU (%)	Port(s)
<input type="checkbox"/>	 lab_3		Running (4/4)	0%	
<input type="checkbox"/>	 calculator-1	lab_3-calculator	Running	0%	
<input type="checkbox"/>	 calculator-2	lab_3-calculator	Running	0%	
<input type="checkbox"/>	 nginx-1	lab_3-nginx	Running	0%	8080:8080 <a href="#">↗</a>
<input type="checkbox"/>	 api-1	lab_3-api	Running	0%	80:80 <a href="#">↗</a>

Які вже можна розширити завдяки команді `docker-compose up --scale calculator=3`

```

○ (study) → lab_3 git:(soa_lab_3-5) ✖ docker-compose up --scale calculator=3
[+] Running 5/0
  ✓ Container lab_3-calculator-2 Running
  ✓ Container lab_3-calculator-1 Running
  ✓ Container lab_3-calculator-3 Created
  ✓ Container lab_3-nginx-1 Running
  ✓ Container lab_3-api-1 Running
Attaching to lab_3-api-1, lab_3-calculator-1, lab_3-calculator-2, lab_3-calculator-3, lab_3-nginx-1
lab_3-calculator-3 | INFO: Will watch for changes in these directories: ['/server']
lab_3-calculator-3 | INFO: Uvicorn running on http://0.0.0.0:81 (Press CTRL+C to quit)
lab_3-calculator-3 | INFO: Started reloader process [1] using StatReload
lab_3-calculator-3 | INFO: Started server process [8]
lab_3-calculator-3 | INFO: Waiting for application startup.
lab_3-calculator-3 | INFO: Application startup complete.

```

Або знизити `docker-compose up --scale calculator=2`:

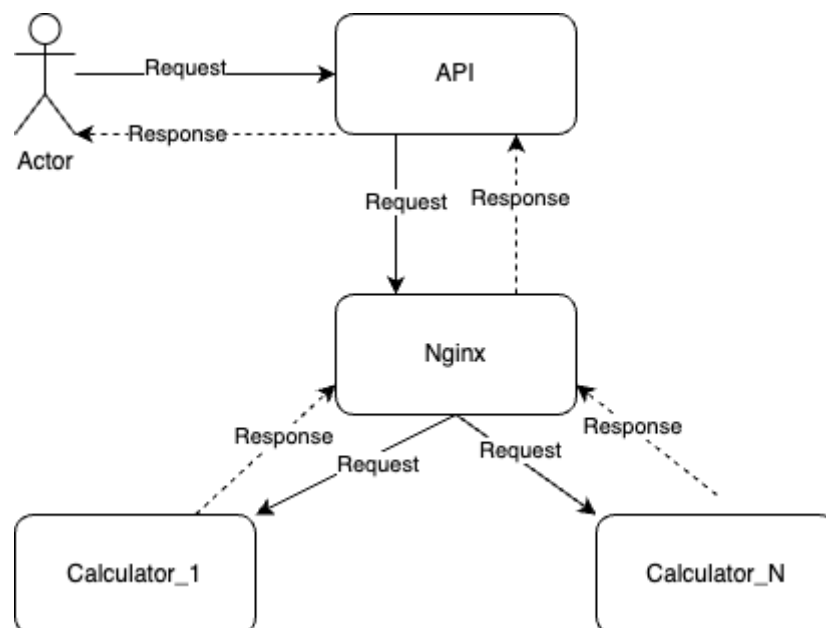
```

.0s Container lab_3-api-1 Running
.0s Attaching to lab_3-api-1, lab_3-calculator-1, lab_3-calculator-2, lab_3-calculator-3, lab_3-nginx-1
lab_3-calculator-3 | INFO: Will watch for changes in these directories: ['/server']
lab_3-calculator-3 | INFO: Uvicorn running on http://0.0.0.0:81 (Press CTRL+C to quit)
lab_3-calculator-3 | INFO: Started reloader process [1] using StatReload
lab_3-calculator-3 | INFO: Started server process [8]
lab_3-calculator-3 | INFO: Waiting for application startup.
lab_3-calculator-3 | INFO: Application startup complete.
lab_3-calculator-3 | INFO: Shutting down
lab_3-calculator-3 | INFO: Waiting for application shutdown.
lab_3-calculator-3 | INFO: Application shutdown complete.
lab_3-calculator-3 | INFO: Finished server process [8]
lab_3-calculator-3 | INFO: Stopping reloader process [1]
lab_3-calculator-3 exited with code 0
lab_3-calculator-3 exited with code 0

○ (base) → lab_3 git:(soa_lab_3-5) ✖ docker-compose up --scale calculator=2
[+] Running 4/2
  ✓ Container lab_3-calculator-2 Running 0.0s
  ✓ Container lab_3-calculator-1 Running 0.0s
  ✓ Container lab_3-nginx-1 Running 0.0s
  ✓ Container lab_3-api-1 Running 0.0s
Attaching to lab_3-api-1, lab_3-calculator-1, lab_3-calculator-2, lab_3-nginx-1

```

У загальному архітектурі даної системи виглядає так:



Як бачимо, клієнт звертається до одного мікросервісу, котрий відправляє запит на load balancer, який вже розподіляє запити між іншими мікросервісами.

Для тестування було написано тестову програму, яка відправляє купу запитів одночасно на api:

```
● (study) → lab_3 git:(soa_lab_3-5) x python test.py
5+5-3 => {"host":"2d151a066967","result":7}
5+5 => {"host":"cad0faf7f0fd","result":10}
5+5//5 => {"host":"2d151a066967","result":6}
5+5-2 => {"host":"cad0faf7f0fd","result":8}
5+11 => {"host":"cad0faf7f0fd","result":16}
5+5+3 => {"host":"cad0faf7f0fd","result":13}
5+5/5 => {"host":"2d151a066967","result":6.0}
5+5-3 => {"host":"cad0faf7f0fd","result":7}
5+5-5 => {"host":"2d151a066967","result":5}
5+5%3 => {"host":"2d151a066967","result":7}
(5+5)*8 => {"host":"cad0faf7f0fd","result":80}
5+5-10 => {"host":"2d151a066967","result":0}
```

У результаті ми бачимо, що host (індикатор контейнера) є різним для певних запитів, отже балансування працює.

**Висновки:** на лабораторній роботі було досліджено мікросервісну архітектуру, розроблено систему, яка складається з двох мікросервісів та loadbalancer. Вдало протестовано роботу розподілу запитів.

Код можна знайти на [GITHUB](#).

### Відповіді на контрольні питання:

1. Розповісти про призначення основних компонентів мікросервісної архітектури.

**Мікросервіси (Microservices):** Мікросервіси представляють собою невеликі, автономні та самостійні компоненти програмного забезпечення, які виконують конкретні функції. Кожен мікросервіс може бути розгорнутий, оновлений та масштабований незалежно від інших, що дозволяє забезпечити гнучкість та швидку реакцію на зміни.

**Шлюз API (API Gateway):** Шлюз API виступає в ролі вхідної точки для клієнтів та мікросервісів. Він забезпечує централізоване управління, маршрутизацію, аутентифікацію та авторизацію запитів, а також може виконувати функції кешування та балансування навантаження.

Служба виявлення (Service Discovery): Цей компонент допомагає мікросервісам взаємодіяти, реєструючи свої адреси та забезпечуючи знаходження доступних сервісів.

База даних (Database): Мікросервіси можуть використовувати свої власні бази даних або ділити спільні бази даних. Це дозволяє кожному сервісу зберігати свої дані незалежно та оптимізувати доступ до них.

Централізовані журнали (Centralized Logging): Забезпечують централізований збір та аналіз журналів подій від усіх мікросервісів. Це сприяє відслідковуванню та діагностиці помилок, виявленню неполадок та взаємодії сервісів.

Моніторинг та трасування (Monitoring and Tracing): Дозволяють відстежувати продуктивність та поведінку мікросервісів. Моніторинг надає інформацію про використання ресурсів, а трасування – про шляхи викликів сервісів в системі.

Конфігураційний сервер (Configuration Server): Забезпечує централізоване управління конфігурацією мікросервісів. Це дозволяє змінювати налаштування без перезапуску служб та допомагає в управлінні різними середовищами.

2. Зобразити архітектуру інтернет магазину (чи іншого сервісу) в термінах мікросервісів, враховуючі інфраструктурні компоненти.

Рекомендаційна система:

- Провайдер – отримання контенту та запис його в бд.
- Скорер – оцінка елементів контенту і збереження в бд.
- Фільтр – отримання даних з бд, фільтрація (по часу публікацій).
- Групувальник – отримання даних з фільтру, групування та видача клієнту.

3. Розповісти про основні патерни проектування мікросервісних архітектур.

Сервісні міжфейси (Service API): Кожен мікросервіс повинен мати чітко визначений API, який надається для взаємодії з іншими сервісами. REST API

База даних за сервісом (Database per Service).

Автоматизоване розгортання (Automated Deployment).

Незалежність сервісів (Service Independence):

Кожен сервіс повинен бути самодостатнім і не повинен сильно залежати від внутрішньої реалізації інших сервісів.

Моніторинг (Monitoring).

Логування (Logging).

#### 4. Orchestration vs Choreography.

**Оркестрація (Orchestration)** передбачає наявність центрального контролера, який відповідає за координацію дій і взаємодії між мікросервісами. Цей центральний компонент визначає порядок та момент виконання кожного мікросервісу, забезпечуючи відповідність встановленому порядку.

Контроль: Логіка контролю централізована, часто виконується окремим сервісом або компонентом, відомим як оркестратор.

Комунікація: Мікросервіси спілкуються з оркестратором, і оркестратор визначає порядок та час їх взаємодії.

**Хореографія (Choreography):** відсутній центральний контролер. Логіка координації розподілена між учасниками.

Контроль: Логіка контролю розподілена та реалізована у кожному мікросервісі. Кожен сервіс знає свою роль і спілкується з іншими на основі подій чи повідомлень.

Комунікація: Мікросервіси спілкуються напряму один з одним, і кожен сервіс відповідає на події та визначає подальші дії.

#### Порівняння:

- Гнучкість:
  - Оркестрація: Централізований контроль забезпечує чітке управління процесом та спрощує зміни, але може вимагати оновлення оркестратора.
  - Хореографія: Розподілений контроль забезпечує більшу гнучкість, але може ускладнити розуміння системи як цілісної єдності.
- Складність:
  - Оркестрація: контрольована система легше розуміється як єдина управлінська структура. Проте це вводить єдину точку виходу та можливі проблеми продуктивності.
  - Хореографія: Кожен мікросервіс має власну логіку, що може ускладнити розуміння системи. Проте це уникне єдиної точки виходу та може бути більш масштабованим.
- Комунікація:
  - Оркестрація: Спілкування здійснюється через оркестратора, що може призвести до більшої ясності управління, але вносить шар непрямого обміну повідомленнями.

- Хореографія: Спілкування відбувається безпосередньо, що може призвести до більшого неявного та розподіленого управління.

#### 5. Описати реалізацію паттерну Circuit breaker.

Паттерн Circuit Breaker – це механізм контролю помилок в мікросервісних архітектурах, який дозволяє програмі визначити, коли і як реагувати на помилки та збої в системі.

Реалізація паттерну "Circuit Breaker" включає такі елементи:

- Лічильник помилок (Error Count).
- Час відкриття (Open Timeout).
- Поріг помилок (Error Threshold).
- Автоматичний повернення в стан "Closed".

#### 6. Описати базові команди docker-compose, або іншого пакету, який застосовувався в роботі.

`docker-compose up --build --scale calculator=3` – перестворює іміджі, якщо є зміни, незалежно від фактора копій в конфіг файлі, буде створено, збільшено або зменшено кількість копій контейнера до 3.

#### 7. Описати механізм знаходження мікросервісів одне одним, як відбувається їх горизонтальне масштабування.

Кожен мікросервіс отримує свій унікальний ідентифікатор та належність до мережі, де вже через DNS можна до нього відправити запит.

Завдяки балансувальнику навантаження потік запитів рівномірно розподіляється між мікросервісами, але якщо навантаження збільшується певної межі, то можливе Автоматичне масштабування – підняття додаткових контейнерів.

#### 8. Роль Service discovery та принцип роботи балансувальника на обраній технології при горизонтальному масштабуванні.

Роль служби полягає в автоматизованому виявленні та реєстрації мікросервісів в розподіленій системі. Це важливий компонент для побудови мікросервісних архітектур, оскільки дозволяє ефективно здійснювати комунікацію між сервісами та автоматично адаптуватися до змін в конфігурації та структурі системи. Nginx використовує циклічну реалізацію для розподілу трафіку.