МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

.

**Основи сервісно-орієнтованих обчислень і архітектур**

**Лабораторний практикум**

Київ

КПІ ім. Ігоря Сікорського

2021

Зміст

[Лабораторна робота №1. Налаштування середовища для роботи з мікросервісами 3](#_Toc524080166)

[Хід роботи: 3](#_Toc524080167)

[Опис Docker 3](#_Toc524080168)

[Установка компонентів 4](#_Toc524080169)

[Контрольні питання: 6](#_Toc524080170)

[Лабораторна робота №2. Запуск тестового мікросервісного додатку. 7](#_Toc524080171)

[Хід роботи: 7](#_Toc524080172)

[Контрольні запитання: 8](#_Toc524080173)

[Лабораторна робота № 3. Створення власного мікросервісу. 9](#_Toc524080174)

[Хід роботи: 9](#_Toc524080175)

[Основні компоненти мікросервісної архітектури 10](#_Toc524080176)

[Комунікація між сервісами 10](#_Toc524080177)

[Контрольні питання: 12](#_Toc524080178)

[Лабораторна робота № 4. Розгортання додатку, в мікросервісах якого застосовуються різні технології 13](#_Toc524080179)

[Хід роботи: 13](#_Toc524080180)

[Контрольні питання: 13](#_Toc524080181)

# Лабораторна робота №1. Налаштування середовища для роботи з мікросервісами

*Мета роботи*: Підготувати середовище розробки для виконання наступних лабораторних робіт.

*Задача*: Підготувати середовище для розробки додатків з використанням мікросервісів: встановити і налаштувати Docker на обраній ОС.

## Хід роботи:

### Опис Docker

Docker — інструментарій для управління ізольованими Linux-контейнерами. Docker доповнює інструментарій LXC більш високорівневим API, що дозволяє керувати контейнерами на рівні ізоляції окремих процесів. Зокрема, Docker дозволяє не переймаючись вмістом контейнера запускати довільні процеси в режимі ізоляції і потім переносити і клонувати сформовані для даних процесів контейнери на інші сервери, беручи на себе всю роботу зі створення, обслуговування і підтримки контейнерів.

Сирцевий код Docker написаний мовою Go і поширюється під ліцензією Apache 2.0. Інструментарій базується на застосуванні вбудованих в ядро Linux штатних механізмів ізоляції на основі просторів імен (namespaces) і груп управління (cgroups). Для створення контейнерів використовуються скрипти lxc. Для формування контейнера досить завантажити базовий образ оточення (команда docker pull base), після чого можна запускати в ізольованих оточеннях довільні програми (наприклад, для запуску bash можна виконати docker run -i -t base/bin/bash).

Основні можливості Docker:

1. Можливість розміщення в ізольованому оточенні різнорідної начинки, що включає різні комбінації виконуваних файлів, бібліотек, файлів конфігурації, скриптів, файлів jar, gem, tar тощо
2. Підтримка роботи на будь-якому комп'ютері на базі архітектури x86\_64 з системою на базі ядра Linux, починаючи від ноутбуків, закінчуючи серверами та віртуальними машинами. Можливість роботи поверх немодифікованих сучасних ядер Linux (без накладення патчів) і в штатних оточеннях всіх великих дистрибутивів Linux, включаючи Fedora, RHEL, Ubuntu, Debian, SUSE, Gentoo і Arch;
3. Використання легковагих контейнерів для ізоляції процесів від інших процесів і основної системи.
4. Оскільки контейнери використовують свою власну самодостатню файлову систему, не важливо де, коли і в якому оточенні вони запускаються.
5. Ізоляція на рівні файлової системи: кожен процес виконується у повністю окремій кореневій ФС;
6. Ізоляція ресурсів: споживання системних ресурсів, таких як витрата пам'яті і навантаження на CPU, можуть обмежуватися окремо для кожного контейнера з використанням cgroups;
7. Ізоляція на рівні мережі: кожен ізольований процес має доступ тільки до пов'язаного з контейнером мережевого простору імен, включаючи віртуальний мережевий інтерфейс і прив'язані до нього IP-адреси;
8. Коренева файлова система для контейнерів створюється з використанням механізму copy-on-write[en] (окремо зберігаються тільки змінені і нові дані), що дозволяє прискорити розгортання, знижує витрату пам'яті і економить дисковий простір;
9. Всі стандартні потоки (stdout/stderr) кожного виконуваного в контейнері процесу накопичуються і зберігаються у вигляді логу;
10. Змінена файлова система одного контейнера може використовуватися як основа для формування нових базових образів і створення інших контейнерів, без необхідності оформлення шаблонів або ручного налаштування складу образів;
11. Можливість використання інтерактивної командної оболонки: до стандартного вводу будь-якого контейнера може бути прив'язаний псевдо-tty для запуску shell.
12. Підтримка використання різних систем зберігання, які можуть підключатися як плаґіни. Серед підтримуваних драйверів зберігання заявлені aufs[en], device mapper[en] (використовуються снапшоти LVM), vfs (на основі копіювання директорій) і Btrfs. Очікується поява драйверів для ZFS, Gluster і Ceph;
13. Можливість створення контейнерів, що містять складні програмні стеки, через зв'язування між собою вже існуючих контейнерів, що містять складові частини формованого стека. Зв'язування здійснюється не через злиття вмісту, а через забезпечення взаємодії між контейнерами (створюється мережевий тунель).
14. [<https://uk.wikipedia.org/wiki/Docker>]

### Установка компонентів

Розглянемо налаштування для ОС Linux (для Mac OS X процес буде аналогічним)

#### Додамо репозиторій

1. Оновимо індекс пакетів:

$ sudo apt-get update

1. Install packages to allow apt to use a repository over HTTPS:

$ sudo apt-get install \

apt-transport-https \

ca-certificates \

curl \

software-properties-common

1. Add Docker’s official GPG key:

$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -

1. Додамо репозиторій

$ sudo add-apt-repository \

"deb [arch=amd64] https://download.docker.com/linux/ubuntu \

$(lsb\_release -cs) \

stable"

#### Встановимо Docker CE

$ sudo apt-get update

$ sudo apt-get install docker-ce

**Запуск докера не від рута:**

1. Створимо групу docker.

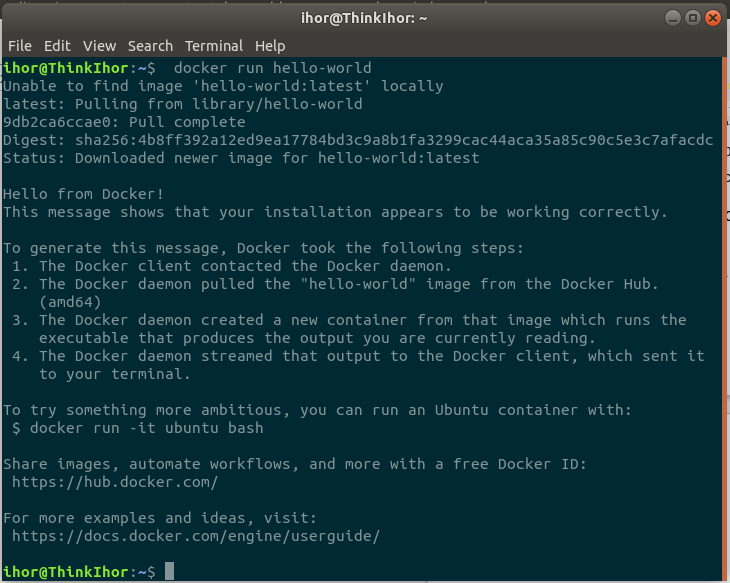
$ sudo groupadd docker

1. Додамо до неї поточного користувача.

$ sudo usermod -aG docker $USER

Верифікуємо правильність виконання попередніх етапів:

$ docker run hello-world



Детальну інформацію по роботі з докер через командний рядок можна отримати за посиланням <https://docs.docker.com/engine/reference/commandline/docker/#child-commands>

Для користувачів Windows Home та в випадку помилок встановлення на OS X рекомендується використовувати Docker Toolbox, котрий являє собою віртуальну машину з Убунту та налаштованим докером.

Користувачі MS Windows 10 PRO мають змогу встановити нативний докер-демон та пов’язати його з докер демоном ubuntu-підсистеми і отримати більш нативну підтримку linux-контейнерів.

Для базової оркестрації контейнерів можна застосувати docker-compose (опис установки за посиланням <https://docs.docker.com/compose/install/#install-compose>)

sudo curl -L "https://github.com/docker/compose/releases/download/1.22.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose

sudo chmod +x /usr/local/bin/docker-compose

## Контрольні питання:

1. Що таке Docker Image
2. Що таке Docker Container
3. Як перестворити і перезапустити контейнер з командного рядка, що при цьому станеться з даним ви ньому?

# Лабораторна робота №2. Запуск тестового мікросервісного додатку.

*Мета роботи*: набути практичних навичок при розгортанні мікросервісного додатку на локальному комп’ютері.

*Задача*: розгорнути базований на Spring (або іншому зручному) фреймворку сервіс на локальному комп’ютері із встановленим та налаштованим у Лабораторній роботі №1 середовищем.

## Хід роботи:

Рекомендовано ознайомитись з best practices створення докерфайлів за посиланням:

<https://docs.docker.com/v17.09/engine/userguide/eng-image/dockerfile_best-practices/#use-multi-stage-builds>

Детальна інформація по всіх інструкціях докерфайлу доступна за посиланням <https://docs.docker.com/engine/reference/builder/>

Розглянемо типову структуру докерфайлу:

FROM ubuntu:15.04

COPY . /app

RUN make /app

CMD python /app/app.py

Кожна інструкція створює новий docker-layer (котрий кешується):

* FROM створює layer з базового образу ubuntu:15.04.
* COPY додає файли з поточної лиректорії клієнта (також є команда ADD, котра інкапсулює роботу з архівами і веб-ресурсами).
* RUN виконує наступну команду, в даному випадку білд за допомогою make.
* CMD команда, котру виконує контейнер.

В випадку наявного спрінг-бут додатку матимемо наступну структуру докер файлу:

FROM java:8-jre # Базовий образ зі встановленим JRE

MAINTAINER Ihor Pysmennyi <ihor.pismennyy@gmail.com>

COPY ./build/libs/api\*.jar /app/api.jar # копіюємо сервер ігноруючи версію білда в файл api.jar

#ARG JAR\_FILE – для передачі uri файлу через аргумент командного рядка

#ADD ${JAR\_FILE} /app/api.jar

CMD ["java", "-Xmx200m", "-jar", "/app/api.jar"] # запускаємо веб-сервер

EXPOSE 8082 #в рантаймі контейнер слухає запити на порту 8082

## Контрольні запитання:

1. Запустити додаток.
2. Питання по основним командам докерфайлу.
3. Що таке сервіс-орієнтована архітектура.
4. Що таке мікросервісна архітектура, які її основні ознаки і як відрізнити від звичайної СОА.
5. Domain Driven Design (DDD) і його зв’язок з мікросервісними архітектурами.

# Лабораторна робота № 3. Створення власного мікросервісу.

*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 (наприклад клієнт зробив запит з якимось рядком, а кожен з сервісів до нього щось допише, головне показати протоколи спілкування). Налаштувати автоматизоване розгортання всіх контейнерів, продемонструвати механізм горизонтального масштабування.

*Завдання*

1. Додати до прикладу, описаного у попередній роботі, власний мікросервіс.

2. Підготувати опис кроків, які було зроблено при підключенні власного сервісу.

3. Підключити інфраструктурні сервіси (edge, discovery, and so on)

4. Протестувати роботу власного сервісу та проілюструвати це. Підняти декілька інстансів одного сервісу, показати механізм горизонтального масштабування і пояснити механізми балансування.

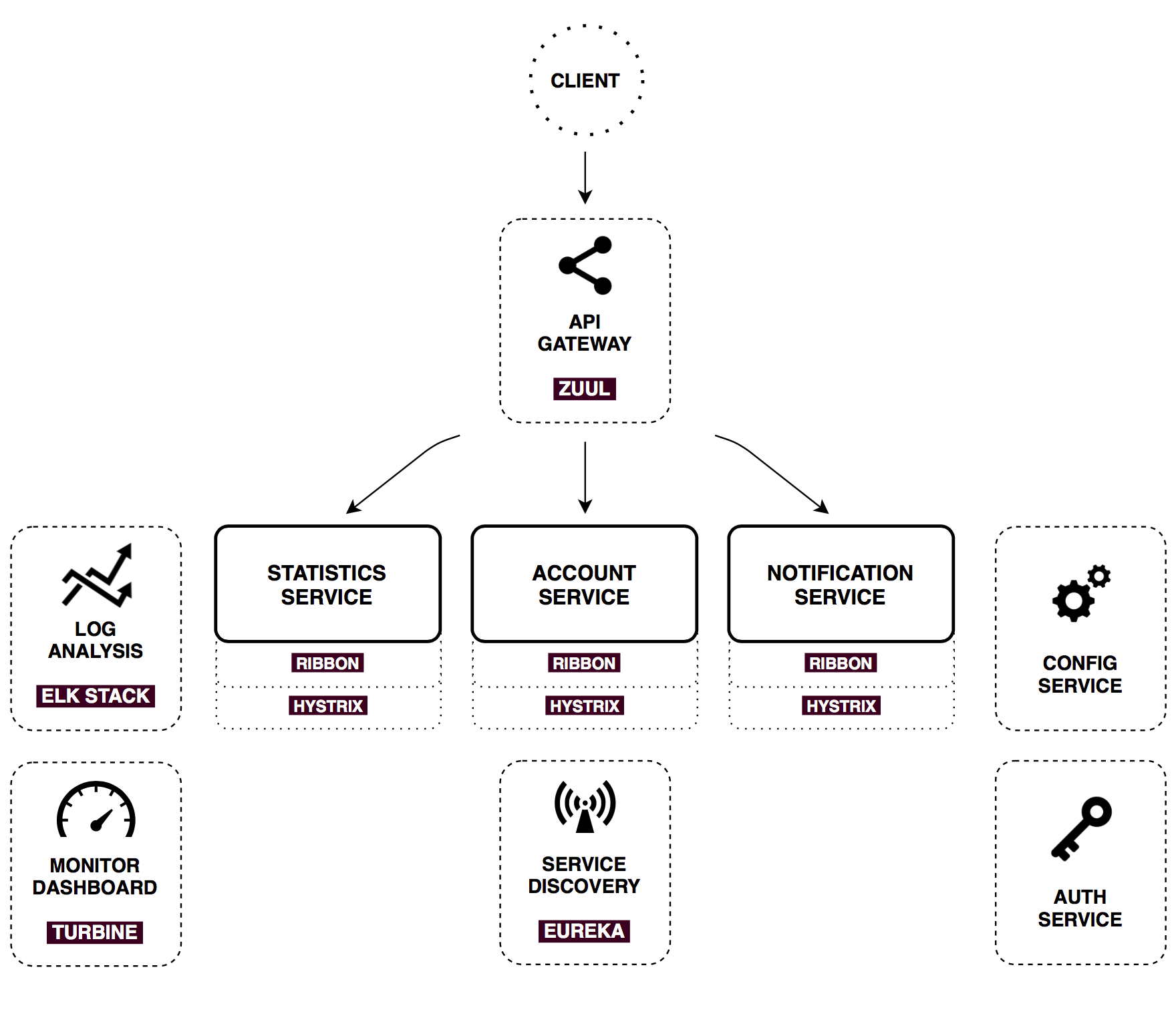
5. Вивчити основні підходи до мікросервісної архітектури.

## Хід роботи:

Ознайомитись з реалізацією мікросервісних архітектур. Презентація: <https://goo.gl/HFduA6>

<https://github.com/sqshq/PiggyMetrics> – репозиторій з якісно реалізованим прикладом на стекові Spring Cloud Netflix

<https://cloud.spring.io/spring-cloud-netflix/>

Компонентний приклад реалізації [https://github.com/sqshq/PiggyMetrics]

### Основні компоненти мікросервісної архітектури

1. Discovery service – реєстр сервісів в системі
2. Edge service – API шлюз
3. Load balancers – балансувальники навантаження між активними інстансами сервісів, бувають client-side, server side
4. Config service
5. Circuit breaker – призначений для мінімізації негативного ефекту від запитів до “поламаних” сервісів , ізолюючи їх.

### Комунікація між сервісами

В дані лабораторній роботі необхідно налаштувати комунікацію між мікросервісами інкапсульованими в докер контейнери, для цього в навчальних цілях можна використовувати docker-compose, на продакшен серидовищах краще користуватись k8s [https://kubernetes.io](https://kubernetes.io/) чи docker-swarm

Нижче наведено простий приклад compose-файлу:

eddie:

restart: always

build: ../eddie/

links: #задаємо резолв адрес між контейнерами

- calculator

- api

ports: #мапінг портів

- 8080:8080

extra\_hosts: #додаємо зовнішні сервісі до хостів

- "extra\_host:external.service.com"

env\_file: #вичитуємо змінні серидовища доступні в контейнері з файлу

- local\_var.env

calculator:

restart: always

build: ../calculator/

links:

- minio

ports:

- 8083

extra\_hosts:

- "extra\_host:external.service.com"

env\_file:

- local\_var.env

api:

restart: always

build: ../api/

links:

- minio

- pgdb

ports:

- 8082:8082

minio:

image: minio/minio

ports:

- 8090:9000

volumes: # монтуємо директорію хоста в контейнер для перзисту даних

- ./media/data:/data

- ./media/config:/root/.minio

env\_file:

- local\_var.env

command: server /data

pgdb:

restart: always

image: 172.16.100.75:5000/build\_pgdb:latest #забираємо образ з локального репозиторію

#build: ./Database/

ports:

- 5432:5432

volumes: # монтуємо директорію хоста в контейнер для перзисту даних

- /srv/docker/pgdata:/var/lib/postgresql/data

Також для створення мережі можна використовувати networks замість links

Детальний опис доступний за посиланням <https://docs.docker.com/compose/compose-file/#compose-file-structure-and-examples>

Масштабування можна дослідити за допомогою docker-compose scale:

docker-compose scale web=2 worker=3

## Контрольні питання:

1. Розповісти про призначення основих компонентів мікросервісної архітектури.
2. Зобразити архітектуру інтернет магазину (чи іншого сервісу) в термінах мікросервісів, враховуючі інфраструктурні компоненти.
3. Розповісти про основні патерни проектування мікросервісних архітектур.
4. Orcherstration vs Choreography.
5. Описати реалізацію паттерну Circuit breaker.
6. Описати базові команди docker-compose, або іншого пакету, який застосовувався в роботі.
7. Описати механізм знаходження мікросервісів одне одним, як відбувається їх горизонтальне масштабування.
8. Роль Service discovery та принцип роботи балансувальника на обранй технології при горизонтальному масштабуванні.

# Лабораторна робота № 4. Розгортання додатку, в мікросервісах якого застосовуються різні технології

*Мета роботи:* ознайомитись з гетерогенністю мікросервісної архітектури, яка дозволяє застосовувати свою технологію кожному сервісу.

*Задача:* на основі попередніх робіт розгорнути «багатомовний» додаток на локальному комп’ютері, протестувати його та дослідити принцип роботи з різними технологіями на рівні мікросервісів. Необхідно розгорнути кілька мікросервісів на різних технологіях і показати, що вони між собою взаємодіють (наприклад клієнт зробив запит з якимось рядком, а кожен з сервісів до нього щось допише, головне показати протоколи спілкування).

## Хід роботи:

Додати новий сервіс на відмінній від попередніх технології до лабораторної 3. Дослідити технології комунікації між сервісами.

## Контрольні питання:

1. Принципи і протоколи комунікації між сервісами,

2. Яким чином сервіси знаходять екземпляри один одного?

3. (\*) Механізм відслідковування тайм-ауту при ланцюжковому (послідовному) виклику мікросервісів.

# Лабораторна робота № 5. Квотування частоти доступу до ресурсів (API-throttling) в сервіс-орієнтованих архітектурах

*Мета роботи:* ознайомитись з різними типами API throttling, алгоритмами їх роботи. Програмно реалізувати обраний механізм.

*Задача:* розглянути підходи до реалізації API throttling. Обмежити частоту викликів між сервісами (з боку сервера та клієнта) в системі розробленій в попередніх роботах.

## Хід роботи:

### Ресурси для ознайомлення:

* <https://konghq.com/blog/how-to-design-a-scalable-rate-limiting-algorithm/>
* <https://medium.com/@animirr/fixed-window-rate-limiter-is-slightly-better-than-token-bucket-here-is-why-bc769c0bdd9>
* <https://nordicapis.com/everything-you-need-to-know-about-api-rate-limiting/>
* <https://www.progress.com/blogs/how-to-rate-limit-an-api-query-throttling-made-easy>
* <https://www.tibco.com/reference-center/what-is-api-throttling>

### Теоретичні відомості

Основою СОА є взаємодія різних сервісів, реалізація їх композиції. При інтеграції сторонніх *сервісів* в систему (або при наданні доступу клієнтам до власних сервісів) постає питання квотування частоти доступу до них – API throttling. Наприклад, AWS Dynamo DB обмежує кількість операцій доступу до БД в залежності від тарифу клієнта (Server-side API throttling). Прикладом необхідності «самообмеження» клієнта в доступі до зовнішнього сервера є виконання клієнтом АПІ інтеграційних SLA типу «Обраний пакет дозволяє максимум 200 запитів за хвилину» (Rate Limitting).

#### Основні алгоритми квотування частоти запитів:

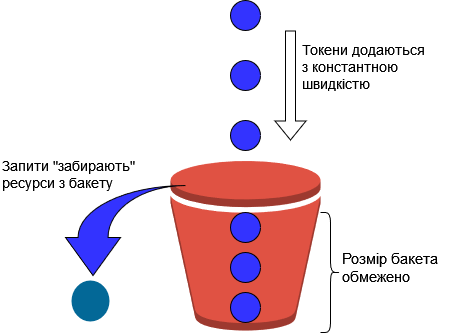
##### Leaking bucket:

По суті є реалізацією FIFO. При реєстрації запиту система додає його в кінець черги. Якщо черга заповнена, додаткові запити відкидаються (або витікають). При цьому обробка запитів з черги відбувається з фіксованим рейтом.

Основною перевагою даного алгоритму є те, що частота відправки запитів на сервер є постійною і не залежить від навантаження клієнта. Недоліком є нижча пріоритетність нових запитів порівняно з вже наявними в черзі (за визначенням FIFO).

##### Token bucket:

Фактично є інверсією leaking bucket: в бакеті зберігаються токени використання ресурсів, які «забираються» клієнтом, квоти додаються в bucket з заданим рейтом.



##### Фіксоване вікно (Fixed window):

Дозволяється N викликів API від користувача за певний період. Часовий проміжок розділений на фіксовані фрейми, кожен з яких триває хвилину. На початку хвилини лічильник встановлюється на нуль. З кожним запитом користувача лічильник збільшується. Якщо лічильник досягає верхньої межі до закінчення часового вікна, нові запити відхиляються. На початку кожної хвилини лічильник скидається на нуль. Скидання лічильника нівелює проблему накопичення запитів в черзі з депріорітизацією нових запитів, проте навантаження на сервер є нерівномірним (call bursts).

##### Ковзке вікно (Sliding window):

На відміну від фіксованого вікна, тут лічильник перераховується що разу при отриманні нового запиту, що дозволяє більш рівномірно розподіляти навантаження/

### Хід роботи

* 1. Обрати сервіс-сервер. Квотувати частоту доступу до нього на рівні 5 запитів в хвилину (не зважаючи на клієнтські додатки). Запити понад цю квоту отримують помилку 429 (https://developer.mozilla.org/ru/docs/Web/HTTP/Status/429).
     1. На додатковий бал – обмеження стосується кожного конкретного сервісу-клієнта і може бути зміненим “per client”
  2. На сервісі клієнті обмежити частоту доступу до ресурсів на рівні 4 запити в хвилину. При перевищенні квоти запити стають в чергу.
     1. Існує досить велика кількість готових клієнтських бібліотек, які реалізують наведені вище алгоритми (наприклад, <https://guava.dev/releases/19.0/api/docs/index.html?com/google/common/util/concurrent/RateLimiter.html>)
  3. Завдання на додатковий бал – реалізувати попередній пункт з врахуванням горизонтального масштабування клієнту (в різні екземпляри клієнта мають квоту в 4 запити).

## Контрольні питання:

1. Основні алгоритми квотування частоти запитів. Їх переваги та недоліки.

2. Які переваги забезпечує квотування частоти запитів. Для яких сервісів є доцільним?

3. (\*) Квотування та горизонтальне масштабування клієнта.