Практична робота №4

NGINX ЯК БАЛАНСУВАЛЬНИК НАВАНТАЖЕННЯ

Мета заняття: набути практичних навичок з налаштування Nginx як балансувальника навантаження; навчитися розгортати бекенд-додатки у Docker-контейнерах; дослідити різні алгоритми балансування навантаження (Round Robin, Weighted Round Robin, Least Connections, IP Hash); відпрацювати механізми health checks, автоматичного failover та резервних серверів; ознайомитися з методами розширеного моніторингу upstream-бекендів засобами Nginx; навчитися налаштовувати таймаути з'єднань, буферизацію відповідей та кастомні сторінки помилок.

Теоретичні відомості

Балансування навантаження (load balancing) — це метод розподілу вхідного мережевого трафіку між кількома серверами (бекендами) з метою підвищення продуктивності, доступності та надійності застосунку. Основна ідея полягає в тому, що жоден окремий сервер не отримує надмірного навантаження, а відмова одного сервера не призводить до повної недоступності сервісу.

Виділяють два основних типи балансування за рівнем OSI-моделі. Балансування на рівні транспортного протоколу (Layer 4, TCP/UDP) маршрутизує трафік виключно на основі IP-адреси та порту, не аналізуючи вміст пакетів. Такий підхід менш гнучкий, але швидший і застосовується для небазованих на HTTP протоколів (бази даних, SMTP, LDAP).

Балансування на прикладному рівні (Layer 7, HTTP/HTTPS) аналізує вміст HTTP-запиту: заголовки, URI, тіло, cookie. Це дозволяє реалізувати складні стратегії маршрутизації — наприклад, направляти запити до /api/ до одного кластера серверів, а до /static/ до іншого (CDN або файловий сервер). Nginx реалізує балансування Layer 7 через директиви upstream та proxy_pass, а Layer 4 — через модуль stream.

Ключові метрики ефективності балансувальника: пропускна здатність (RPS — запитів за секунду), затримка (latency — час від запиту до відповіді), доступність (uptime — частка часу, коли сервіс доступний). Правильно налаштований балансувальник дозволяє горизонтально масштабувати систему: замість одного потужного сервера (vertical scaling) використовуються кілька менш потужних (horizontal scaling), що також підвищує відмовостійкість.

Nginx у ролі балансувальника реалізує кілька алгоритмів розподілу трафіку. Round Robin (за замовчуванням) послідовно обходить список серверів. Weighted Round Robin враховує різну потужність серверів через параметр weight. Least Connections направляє до сервера з мінімумом активних з'єднань. IP Hash забезпечує прив'язку клієнта до конкретного сервера для підтримки сесій. Кожен алгоритм має свої переваги та обмеження залежно від характеристик трафіку.

Архітектура тестового стенду цієї практичної роботи: на одній Vagrant VM встановлений Nginx та запущені три Docker-контейнери з ідентичним Node.js-додатком. Nginx слухає порт 80 і приймає HTTP-запити за іменем домену lb-xxx-yyy-zzz.local, яке резолюється через /etc/hosts у 127.0.0.1. Запити проксюються до контейнерів на портах 3001, 3002, 3003. Цей стенд відтворює реальну виробничу схему балансування між кількома екземплярами одного мікросервісу.

Завдання на роботу

1. Підготовка робочого середовища та встановлення необхідного програмного забезпечення.

Перед виконанням основних завдань необхідно підготувати Vagrant VM: встановити Docker Engine, Node.js та допоміжні утиліти, а також зібрати Docker-образ бекенд-додатку і запустити три контейнери на різних портах. Nginx вже встановлений на VM з попередніх практичних робіт.

a. Перевірити та при необхідності оновити Vagrantfile для поточної роботи.

Для цієї практичної роботи достатньо базового Vagrantfile з попередніх занять. Переконатися, що VM запущена і пам'ять сконфігурована не менше 1024 МБ (Docker потребує додаткових ресурсів). При необхідності відредагувати Vagrantfile:

```
Vagrant.configure("2") do |config|
  config.vm.box = "ubuntu/focal64"

  config.vm.provider "virtualbox" do |vb|
    vb.memory = "1024"  # мінімум 1 ГБ для Docker
    vb.cpus   = 2
  end

  # Приватна мережа для прямого доступу з хост-машини
  config.vm.network "private_network", ip: "192.168.56.10"
end
```

Рис. 1.a – Vagrantfile з налаштуванням пам'яті та приватної мережі

```
vagrant up
vagrant ssh
```

Перевірити застосовану конфігурацію, виконавши команду:

```
free -h && nproc
```

b. Встановити Docker та підготувати оточення контейнеризації.

Docker — платформа контейнеризації, що дозволяє пакувати застосунок разом із усіма залежностями у стандартизований образ (image). Контейнери ізольовані між собою та від хост-системи за допомогою механізмів ядра Linux: cgroups (обмеження ресурсів) та namespaces (ізоляція процесів, мережі, файлової системи).

У репозиторії Ubuntu пакет docker.io містить стабільну версію Docker Engine без необхідності підключення зовнішніх репозиторіїв. Для production-середовищ рекомендується встановлювати Docker CE безпосередньо з репозиторію docker.com, але для навчальних цілей docker.io з ubuntu-репозиторію є достатнім.

Оновити список пакетів та встановити Docker:

```
sudo apt update && sudo apt install -y docker.io
```

Запустити службу Docker та увімкнути її автозапуск при завантаженні системи:

```
sudo systemctl enable --now docker
```

Перевірити статус служби:

```
sudo systemctl status docker --no-pager
```

Додати користувача `vagrant` до групи `docker`, щоб виконувати команди Docker без `sudo`. Група docker надає доступ до unix-сокета демона: `/var/run/docker.sock`.

```
sudo usermod -aG docker vagrant
```

Важливо: після додавання до групи необхідно завершити поточну SSH-сесію та під'єднатися знову. Виконати `exit`, потім `vagrant ssh`.

Перевірити застосовану конфігурацію, виконавши команду:

```
docker --version && docker ps
```

c. Встановити Node.js та необхідні утиліти для розробки та тестування.

Node.js — середовище виконання JavaScript поза браузером, засноване на движку V8. Воно підходить для розробки швидких I/O-інтенсивних вебсерверів завдяки асинхронній, неблокуючій архітектурі на основі event loop. У цій роботі Node.js використовується для запуску бекенд-додатку безпосередньо, а також буде упакований у Docker-образ.

Утиліта jq — легкий процесор JSON для командного рядка, що дозволяє форматувати, фільтрувати та трансформувати JSON-відповіді безпосередньо у терміналі. curl — інструмент для надсилання HTTP-запитів та отримання відповідей.

```
sudo apt install -y nodejs npm curl jq
```

Перевірити застосовану конфігурацію, виконавши команду:

```
node --version && npm --version && jq --version
```

Зробити висновки за отриманими результатами.

d. Створити директорію проєкту `~/backend-app` та підготувати її структуру.

Усі файли бекенд-додатку (вихідний код app.js та Dockerfile) розміщуються в одній директорії. Це є build context для команди docker

build — лише файли з цієї директорії будуть доступні при виконанні інструкцій Dockerfile (наприклад, COPY).

```
mkdir -p ~/backend-app && cd ~/backend-app
ls -la ~/backend-app
```

e. Створити файл `app.js` — мінімальний HTTP-сервер на Node.js з діагностичним JSON-відгуком.

Додаток реалізує HTTP-сервер тільки на вбудованих модулях Node.js (http, os) — без зовнішніх залежностей (npm install не потрібен). Порт читається зі змінної середовища PORT, що дозволяє запускати кілька ідентичних контейнерів з одного образу на різних портах.

На GET-запит до кореневого шляху / сервер повертає JSON з чотирма полями: hostname — ім'я хосту контейнера (унікальне, генерується Docker); port — порт прослуховування; timestamp — час обробки запиту у форматі ISO 8601; requests — лічильник запитів на поточний екземпляр. Ці поля дозволяють однозначно визначити, який бекенд обробив запит.

Створити файл `~/backend-app/app.js` з таким вмістом (рис. 1.e):

```
'use strict';
const http = require('http');
const os   = require('os');

// Порт із змінної середовища або 3000 за замовчуванням
const PORT    = parseInt(process.env.PORT || '3000', 10);
let requestCount = 0;

const server = http.createServer((req, res) => {
  // Обробляти тільки GET /
  if (req.method !== 'GET' || req.url !== '/') {
    res.writeHead(404);
    res.end(JSON.stringify({ error: 'Not Found' }));
    return;
  }
  requestCount++;
  const body = JSON.stringify({
    hostname:  os.hostname(),            // унікальне ім'я контейнера
    port:      PORT,                     // порт прослуховування
    timestamp: new Date().toISOString(), // час обробки запиту
    requests:  requestCount              // лічильник запитів до цього
екземпляра
  }, null, 2);
  res.writeHead(200, {
    'Content-Type':   'application/json',
    'Content-Length': Buffer.byteLength(body)
```

```
  });
  res.end(body);
});

server.listen(PORT, () => {
  console.log(`Backend listening on port ${PORT}`);
});
```

Рис. 1.e – Файл app.js — діагностичний Node.js-бекенд

```
nano ~/backend-app/app.js
```

f. Створити файл `Dockerfile` для збірки Docker-образу бекенду.

Dockerfile — текстовий файл з покроковими інструкціями для Docker щодо збірки образу. Кожна інструкція (FROM, WORKDIR, COPY, EXPOSE, CMD) створює новий незмінний шар образу. Шарова архітектура дозволяє повторно використовувати незмінені шари при перебудові образу.

Базовий образ node:alpine обирається свідомо: Alpine Linux має розмір близько 5 МБ, що дає фінальний образ ~60 МБ замість ~350 МБ при node:latest (Ubuntu/Debian-base). Це критично для Production-середовищ, де образи зберігаються та передаються по мережі. Для мінімізованих образів без npm-залежностей node:alpine є оптимальним вибором.

Створити файл `~/backend-app/Dockerfile` (рис. 1.f):

```
# Базовий образ: Node.js на Alpine Linux (~60 МБ)
FROM node:alpine

# Встановити робочу директорію всередині контейнера
WORKDIR /app

# Копіювати файл додатку до образу (з build context)
COPY app.js .

# Документаційний порт (реальне відкриття при docker run -p)
EXPOSE 3000

# Команда запуску при старті контейнера
CMD ["node", "app.js"]
```

Рис. 1.f – Dockerfile для збірки мінімального бекенд-образу

```
nano ~/backend-app/Dockerfile
```

Переглянути структуру директорії перед збіркою:

```
ls -la ~/backend-app/
```

g. Зібрати Docker-образ з тегом `backend-app`.

Прапорець `-t backend-app` присвоює образу тег (ім'я) для зручного посилання. Крапка наприкінці вказує на поточну директорію як build context — Docker читатиме Dockerfile та матиме доступ до всіх файлів у цій директорії під час COPY.

Виконати з директорії `~/backend-app`:

```
cd ~/backend-app && docker build -t backend-app .
```

Переглянути список локальних образів та розмір створеного образу:

```
docker images backend-app
```

Перевірити застосовану конфігурацію, виконавши команду:

```
docker images --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
```

h. Запустити три екземпляри бекенду як Docker-контейнери на портах 3001, 3002, 3003.

Три контейнери запускаються з одного образу `backend-app`, але з різними параметрами. Прапорці команди docker run: **-d** — detached mode (фоновий режим, не блокує термінал); **--name** — унікальне ім'я контейнера; **-p 3001:3001** — прив'язка порту 3001 хоста до порту 3001 контейнера (host:container); **-e PORT=3001** — встановлення змінної середовища, яку app.js читає для вибору порту прослуховування.

```
docker run -d --name backend1 -p 3001:3001 -e PORT=3001 backend-app
docker run -d --name backend2 -p 3002:3002 -e PORT=3002 backend-app
docker run -d --name backend3 -p 3003:3003 -e PORT=3003 backend-app
```

Переконатися, що всі три контейнери запущені зі статусом Up:

Перевірити застосовану конфігурацію, виконавши команду:

```
docker ps --format "table {{.Names}}\t{{.Status}}\t{{.Ports}}"
```

i. Перевірити роботу кожного контейнера окремо через прямі curl-запити.

Прямі запити до кожного контейнера обходять Nginx-балансувальник і звертаються безпосередньо до кожного бекенду. Це дозволяє підтвердити, що всі три бекенди запущені і відповідають коректно перед налаштуванням балансування. Поле hostname у відповіді буде унікальним для кожного контейнера.

```
curl -s http://localhost:3001/ | jq .
curl -s http://localhost:3002/ | jq .
curl -s http://localhost:3003/ | jq .
```

Переглянути логи контейнера для перевірки успішного запуску:

```
docker logs backend1
```

Порівняти hostname у відповідях трьох контейнерів:

```
for p in 3001 3002 3003; do echo "Port $p:"; curl -s
http://localhost:$p/ | jq .hostname; done
```

Зробити висновки за отриманими результатами.

2. Round Robin — базове рівномірне балансування навантаження.

Round Robin (кругова черга) — найпростіший і найпоширеніший алгоритм балансування. Nginx почергово направляє кожен новий запит до наступного бекенду у списку: запит 1 → server1, запит 2 → server2, запит 3 → server3, запит 4 → знову server1. Цикл повторюється нескінченно. Алгоритм гарантує математично рівномірний розподіл при однаковому часі обробки запитів на всіх бекендах.

Round Robin є алгоритмом за замовчуванням у блоці upstream Nginx — його не потрібно вказувати явно жодною директивою. Щоб увімкнути інший алгоритм, необхідно додати відповідну директиву (least_conn, ip_hash тощо) у блок upstream. Блок upstream визначає іменовану групу серверів; директива proxy_pass у location передає запити до цієї групи.

a. Додати доменне ім'я балансувальника до файлу `/etc/hosts` на VM.

Файл `/etc/hosts` — локальна таблиця відповідності доменних імен та IP-адрес. Його записи мають пріоритет над DNS-резолюцією (конфігурується через `/etc/nsswitch.conf`). Це стандартний інструмент тестування вебсерверів за іменем домену без реєстрації DNS.

Замінити `xxx-yyy-zzz` у команді на реальний ідентифікатор: xxx — шифр навчальної групи, yyy — номер варіанта, zzz — ініціали студента латиницею.

```
echo "127.0.0.1  lb-xxx-yyy-zzz.local" | sudo tee -a /etc/hosts
```

Перевірити застосовану конфігурацію, виконавши команду:

```
grep lb-xxx-yyy-zzz.local /etc/hosts
```

b. Створити файл конфігурації балансувальника `/etc/nginx/sites-available/lb-xxx-yyy-zzz`.

Конфігурація складається з двох блоків: upstream визначає групу бекенд-серверів, server описує віртуальний хост, що приймає запити і передає їх до upstream. Директиви proxy_set_header передають бекенду важливу метаінформацію про оригінальний запит клієнта, яка інакше була б втрачена через проксювання.

Значення директив proxy_set_header: `Host $host` — оригінальний HTTP-заголовок Host (домен, на який звертався клієнт); `X-Real-IP $remote_addr` — реальна IP-адреса клієнта (не Nginx); `X-Forwarded-For $proxy_add_x_forwarded_for` — ланцюжок всіх проксі; `X-Forwarded-Proto $scheme` — протокол (http або https).

```
# Група бекенд-серверів (Round Robin за замовчуванням)
upstream backend_pool {
    server 127.0.0.1:3001;  # backend1
    server 127.0.0.1:3002;  # backend2
    server 127.0.0.1:3003;  # backend3
}

server {
    listen 80;
    server_name lb-xxx-yyy-zzz.local;

    location / {
        proxy_pass         http://backend_pool;
        proxy_set_header   Host              $host;
        proxy_set_header   X-Real-IP         $remote_addr;
        proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }
}
```

Рис. 2.b – Конфігурація Nginx — Round Robin балансування

c. Активувати конфігурацію через символічне посилання та перевантажити Nginx.

Nginx використовує паттерн sites-available / sites-enabled: у `sites-available` зберігаються всі конфігурації (в т.ч. вимкнені), а `sites-enabled` містить символічні посилання (symlinks) тільки на активні. Такий

підхід дозволяє легко вмикати/вимикати сайти без редагування або видалення файлів.

```
sudo ln -s /etc/nginx/sites-available/lb-xxx-yyy-zzz /etc/nginx/sites-enabled/
```

Перевірити синтаксис конфігурації перед перевантаженням. Команда `nginx -t` завантажує конфігурацію та виводить діагностику без фактичного перезапуску:

```
sudo nginx -t
```

Команда `systemctl reload` застосовує нову конфігурацію без переривання поточних з'єднань (graceful reload), на відміну від restart:

```
sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
ls -la /etc/nginx/sites-enabled/ | grep lb
```

Зробити висновки за отриманими результатами.

d. Перевірити рівномірний розподіл Round Robin: надіслати 9 запитів та проаналізувати розподіл.

Для спостереження балансування надсилається серія запитів до балансувальника і фіксується hostname або port бекенду, що відповів. При Round Robin кожен бекенд повинен отримати рівно третину від загальної кількості запитів.

```
# Надіслати 9 запитів і відобразити розподіл між бекендами
for i in $(seq 1 9); do
  echo -n "Запит $i → ";
  curl -s http://lb-xxx-yyy-zzz.local/ | jq -r '"hostname: " + .hostname
+ ", port: " + (.port|tostring)';
done
```

Рис. 2.d – Скрипт перевірки розподілу Round Robin (9 запитів)

Виконати скрипт з рис 2.d та спостерігати почергове чергування бекендів:

Підрахувати, скільки разів кожен порт (бекенд) відповів — очікується рівно по 3:

```
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Переглянути поточний стан Nginx та перевірити конфігурацію:

```
sudo nginx -T 2>/dev/null | grep -A 10 upstream
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -s http://lb-xxx-yyy-zzz.local/ | jq .
```

Зробити висновки за отриманими результатами.

е. Перевірити лічильник запитів на кожному бекенді після серії запитів.

Поле requests у відповіді бекенду є лічильником запитів на конкретний екземпляр, що зберігається у пам'яті процесу. Після 9 запитів при Round Robin кожен бекенд повинен мати лічильник, приблизно рівний 3 (допустиме відхилення ±1). Прямі запити до кожного бекенду дозволяють переглянути реальний стан лічильника.

```
curl -s http://localhost:3001/ | jq '{port: .port, requests:
.requests}'
curl -s http://localhost:3002/ | jq '{port: .port, requests:
.requests}'
curl -s http://localhost:3003/ | jq '{port: .port, requests:
.requests}'
```

Зверніть увагу: лічильник requests зберігається у пам'яті Node.js-процесу і скидається при перезапуску контейнера. При кожному новому запуску він починається з 0.

Зробити висновки за отриманими результатами.

3. Weighted Round Robin — зважене балансування навантаження.

Weighted Round Robin розширює базовий алгоритм можливістю призначення ваги (параметр weight) кожному серверу у блоці upstream. Вага визначає відносну частку трафіку, яку отримуватиме сервер. Сервер з weight=5 отримає у 5 разів більше запитів, ніж сервер з weight=1. Загальна кількість «слотів» у циклі дорівнює сумі ваг всіх серверів.

Зважене балансування застосовується в двох основних сценаріях. Перший — гетерогенна інфраструктура: якщо один бекенд має вдвічі більше CPU та RAM, йому призначають удвічі більшу вагу для ефективного використання ресурсів. Другий — Canary Deployment (канаркові розгортання): новій версії застосунку спочатку призначають малу вагу (weight=1 при решті weight=10) для тестування під мінімальним навантаженням з поступовим збільшенням.

a. Відредагувати блок `upstream`: додати параметр `weight` до кожного сервера.

У конфігурації нижче backend1 отримує вагу 5, backend2 та backend3 — по 1. Загальна сума ваг = 5+1+1 = 7. Розподіл при 14 запитах: backend1 отримає 10 запитів (5/7 × 14 ≈ 10), backend2 та backend3 — по 2 (1/7 × 14 ≈ 2). Параметр weight за замовчуванням дорівнює 1 і може бути пропущений.

```
# Weighted Round Robin — розподіл 5:1:1
upstream backend_pool {
    server 127.0.0.1:3001 weight=5;  # основний  — 5/7 трафіку
    server 127.0.0.1:3002 weight=1;  # резервний — 1/7 трафіку
    server 127.0.0.1:3003 weight=1;  # резервний — 1/7 трафіку
}
```

Рис. 3.a – Блок upstream з вагами серверів Weighted Round Robin

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

b. Перевірити розподіл запитів відповідно до налаштованих ваг.

Надіслати 14 запитів та підрахувати кількість відповідей від кожного порту. Очікуваний результат: порт 3001 отримає ~10 відповідей, порти 3002 та 3003 — приблизно по 2 відповіді.

```
for i in $(seq 1 14); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Переглянути повну JSON-відповідь для підтвердження коректності балансування:

Зробити висновки за отриманими результатами.

c. Проаналізувати вплив ваги на розподіл трафіку: перевірити лічильники на кожному бекенді.

Після серії з 14 запитів через балансувальник перевірити лічильники запитів безпосередньо на кожному бекенді. Різниця у значеннях підтверджує нерівномірний розподіл відповідно до ваг: backend1 отримав найбільше, backend2 та backend3 — однаково мало.

```
for p in 3001 3002 3003; do echo -n "port $p requests: "; curl -s
http://localhost:$p/ | jq .requests; done
```

Зважений Round Robin корисний не лише для гетерогенних серверів, а й для Canary Deployment (поступове введення нової версії): новій версії призначається weight=1, решті — weight=10. Так тільки 9% трафіку отримає нову версію. При відсутності проблем вага поступово збільшується до рівного значення.

Перевірити застосовану конфігурацію, виконавши команду:

```
for i in $(seq 1 7); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

d. Повернути рівні ваги перед наступним завданням.

Видалити параметри weight або встановити `weight=1` для всіх серверів, щоб повернутися до стандартного Round Robin. Перевантажити Nginx після редагування:

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
for i in $(seq 1 6); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

4. Least Connections — балансування за кількістю активних з'єднань.

Алгоритм Least Connections (найменша кількість з'єднань) направляє кожен новий запит до бекенду з мінімальною кількістю поточних активних з'єднань. На відміну від Round Robin, цей алгоритм адаптується до реального навантаження: якщо один бекенд обробляє повільні запити і має багато відкритих з'єднань, нові запити підуть до вільніших серверів.

Least Connections є більш справедливим алгоритмом за Round Robin у сценаріях, де запити мають суттєво різний час обробки. Типовий приклад: веб-API, де 80% запитів повертають кешовані результати за 5 мс, а 20% виконують важкі обчислення по 500 мс. Round Robin може перевантажити один бекенд повільними запитами; Least Connections автоматично компенсує

це, направляючи більше швидких запитів до зайнятого сервера лише після того, як він звільниться.

Директива least_conn вказується у блоці upstream і замінює алгоритм Round Robin. Цей алгоритм можна комбінувати з параметром weight: зважений Least Connections нормалізує кількість з'єднань на вагу сервера (connections / weight).

a. Додати директиву `least_conn` до блоку `upstream`.

Директива `least_conn` повністю замінює Round Robin. Nginx автоматично відстежує кількість активних з'єднань до кожного бекенду і балансує трафік відповідно. Додаткових параметрів для директиви не потрібно.

```
# Алгоритм Least Connections
upstream backend_pool {
    least_conn;  # направляти до бекенду з мінімумом активних з'єднань

    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
    server 127.0.0.1:3003;
}
```

Рис. 4.a – Блок upstream з алгоритмом Least Connections

```
sudo vi  /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

b. Перевірити роботу алгоритму Least Connections.

При однаковому (миттєвому) часі обробки запитів Least Connections поводиться подібно до Round Robin — рівномірний розподіл. Відмінність помітна при паралельних довготривалих з'єднаннях, але для навчальних цілей достатньо переконатися, що алгоритм вмикається і запити не зосереджуються на одному бекенді.

```
for i in $(seq 1 9); do echo -n "Запит $i → "; curl -s http://lb-xxx-
yyy-zzz.local/ | jq -r '"port: " + (.port|tostring)'; done
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

c. Порівняти поведінку Round Robin та Least Connections при послідовних запитах.

Для наочного порівняння двох алгоритмів виконати серію запитів при Least Connections і порівняти розподіл з попередньо отриманими результатами Round Robin. При низькому навантаженні (послідовні короткі запити) обидва алгоритми дають схожий результат — приблизно рівномірний розподіл. Різниця проявляється тільки при паралельних запитах з різним часом обробки.

```
for i in $(seq 1 12); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Least Connections є стандартним вибором для API-шлюзів у production, де запити мають різний час обробки. Nginx Plus (комерційна версія) додатково підтримує активні health checks, що дозволяють виявляти проблемні бекенди проактивно, не чекаючи реальних помилок клієнтів.

Зробити висновки за отриманими результатами.

d. Видалити директиву `least_conn` для повернення до Round Robin.

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
for i in $(seq 1 6); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

5. IP Hash — прив'язка сесії клієнта до конкретного бекенду.

IP Hash — алгоритм, що забезпечує стійку прив'язку (session persistence / sticky sessions) клієнта до конкретного бекенду на основі хешу його IP-адреси. Nginx обчислює хеш від перших трьох октетів IPv4-адреси клієнта (наприклад, для 192.168.1.5 хешується 192.168.1 — вся підмережа /24 потрапляє до одного бекенду). Для IPv6 хешуються перші 64 біти.

IP Hash вирішує проблему зберігання стану (state) у пам'яті бекенду. Традиційні застосунки (PHP із $_SESSION, деякі Java-фреймворки) зберігають сесійні дані у RAM конкретного процесу. Якщо наступний запит клієнта потрапить до іншого бекенду, сесія буде втрачена. IP Hash гарантує,

що всі запити з однієї IP йдуть до одного бекенду, зберігаючи неперервність сесії.

Обмеження IP Hash: якщо багато клієнтів знаходяться за NAT (корпоративний шлюз, мобільний оператор), всі вони матимуть однакову зовнішню IP і потраплять до одного бекенду — що знищує сенс балансування. Сучасна альтернатива — зберігання сесій у централізованому Redis-сховищі з Round Robin/Least Connections для балансування.

a. Додати директиву `ip_hash` до блоку `upstream`.

Директива `ip_hash` несумісна з `least_conn` і `hash`. Вона перевизначає весь алгоритм балансування, забезпечуючи deterministic routing (детерміноване маршрутизування): один і той самий клієнт завжди потрапляє до одного бекенду.

```
# IP Hash — sticky sessions на основі IP-адреси клієнта
upstream backend_pool {
    ip_hash;  # хешувати IP → завжди один і той самий бекенд

    server 127.0.0.1:3001;
    server 127.0.0.1:3002;
    server 127.0.0.1:3003;
}
```

Рис. 5.a – Блок upstream з алгоритмом IP Hash

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

b. Перевірити прив'язку клієнта до бекенду при IP Hash.

Оскільки всі curl-запити надходять з однієї IP-адреси (127.0.0.1), всі вони повинні отримувати відповідь від одного і того самого бекенду. Підрахунок показує, що лише один порт з'являється у результатах — це підтверджує роботу sticky session.

```
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Очікуваний результат: один рядок з кількістю 9 — всі запити до одного бекенду.

Зробити висновки за отриманими результатами.

c. Розглянути обмеження алгоритму ip_hash та сучасні альтернативи.

Директива ip_hash має важливе обмеження: якщо значна частина клієнтів знаходиться за одним NAT-шлюзом (корпоративна мережа, мобільний оператор), всі вони матимуть однакову зовнішню IP-адресу і будуть завжди направлятись до одного бекенду. Це нівелює сенс балансування і може перевантажити один сервер.

Сучасна рекомендована альтернатива для sticky sessions — централізоване сховище сесій (Redis, Memcached) з Round Robin або Least Connections для балансування. При такому підході будь-який бекенд може обробити запит будь-якого клієнта, оскільки сесійні дані доступні з Redis, а не зберігаються у RAM конкретного процесу.

Якщо перенесення сесій до Redis неможливе (legacy-додаток), альтернативою є sticky cookie: Nginx Plus може додавати до відповіді cookie з ідентифікатором бекенду і в наступних запитах використовувати його для маршрутизації. Це надійніше за ip_hash, оскільки прив'язує конкретного клієнта, а не підмережу.

Перевірити застосовану конфігурацію, виконавши команду:

```
for i in $(seq 1 6); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

d. Видалити директиву `ip_hash` та повернутися до Round Robin перед наступним завданням.

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

6. Health checks та автоматичний failover бекендів.

Nginx Open Source підтримує пасивні health checks через параметри max_fails та fail_timeout у блоці upstream. «Пасивний» означає, що Nginx не надсилає спеціальних тестових запитів — він аналізує реальні запити клієнтів: якщо бекенд повертає помилки підключення або не відповідає протягом заданого

часу, Nginx фіксує невдачу. Після досягнення порогу max_fails сервер тимчасово виводиться з ротації на час fail_timeout.

Директива backup позначає сервер як резервний (hot standby): він не отримує запитів в нормальному режимі, але автоматично активується, якщо всі основні бекенди стануть недоступними. Директива down повністю і статично виключає сервер з балансування незалежно від стану інших серверів — на відміну від backup, down-сервер не активується навіть при відмові всіх інших бекендів. Обидва параметри набирають чинності при reload Nginx, а не автоматично при зміні стану бекенду.

a. Налаштувати параметри `max_fails` та `fail_timeout` для кожного бекенду у блоці upstream.

   Параметр **max_fails=3** вказує максимальну кількість невдалих спроб з'єднання з бекендом за час fail_timeout. Після досягнення цього порогу сервер вважається тимчасово недоступним (down). Параметр **fail_timeout=15s** виконує дві ролі: задає тривалість вікна підрахунку невдалих спроб, і тривалість «карантину» сервера після виключення. Через 15 секунд Nginx спробує направити запит до виключеного сервера знову.

```
# Round Robin з пасивними health checks
upstream backend_pool {
    # max_fails: кількість помилок за fail_timeout для визнання сервера
недоступним
    # fail_timeout: вікно підрахунку помилок і тривалість паузи сервера
    server 127.0.0.1:3001 max_fails=3 fail_timeout=15s;
    server 127.0.0.1:3002 max_fails=3 fail_timeout=15s;
    server 127.0.0.1:3003 max_fails=3 fail_timeout=15s;
}
```

Рис. 6.a – Параметри max_fails та fail_timeout для пасивних health checks

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

b. Змоделювати відмову бекенду та спостерігати автоматичний failover.

   Зупинити контейнер `backend2`, щоб змоделювати відмову сервера. Nginx продовжить надсилати перші 3 запити до недоступного backend2 (поки

не зафіксує max_fails), потім виключить його з ротації. Після виключення запити розподілятимуться лише між backend1 та backend3.

```
docker stop backend2
```

Надіслати серію запитів і спостерігати за автоматичним виключенням бекенду:

```
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Перевірити лог помилок Nginx на наявність записів про недоступний upstream:

```
sudo tail -20 /var/log/nginx/error.log
```

Відновити `backend2` та переконатися, що балансування повертається до трьох серверів (після закінчення fail_timeout Nginx повторно включить backend2):

```
docker start backend2
```

Після відновлення backend2 надіслати запити та перевірити перерозподіл:

```
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

c. Позначити `backend3` як резервний (backup) сервер і перевірити його поведінку.

Директива `backup` налаштовує сервер як «гарячий резерв». У нормальному режимі він не бере участі у балансуванні та не отримує жодних запитів. Активується тільки тоді, коли всі основні сервери позначені як недоступні — виключені через max_fails або явно зазначені як down.

```
upstream backend_pool {
    server 127.0.0.1:3001 max_fails=3 fail_timeout=15s;  # основний
    server 127.0.0.1:3002 max_fails=3 fail_timeout=15s;  # основний
    server 127.0.0.1:3003 backup;  # резервний — активується при відмові
основних
}
```

Рис. 6.c – Блок upstream з директивою backup для резервного сервера

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

У нормальному режимі backend3 (порт 3003) не повинен з'являтися у відповідях:

```
for i in $(seq 1 6); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зупинити обидва основних бекенди та переконатися, що backup-сервер взяв трафік:

```
docker stop backend1 backend2
curl -s http://lb-xxx-yyy-zzz.local/ | jq .port
```

Відновити основні бекенди після тестування:

```
docker start backend1 backend2
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -s http://lb-xxx-yyy-zzz.local/ | jq .
```

Зробити висновки за отриманими результатами.

d. Використати директиву `down` для виключення бекенду під час планового обслуговування.

Директива `down` статично виключає сервер з балансування. На відміну від backup, down-сервер не активується навіть якщо всі інші бекенди недоступні. Директива down застосовується для планового технічного обслуговування: оновлення застосунку, перевірки конфігурації, дебагу. Зміна набирає чинності при перевантаженні Nginx (reload), а не при docker stop.

```
upstream backend_pool {
    server 127.0.0.1:3001 max_fails=3 fail_timeout=15s;
    server 127.0.0.1:3002 max_fails=3 fail_timeout=15s;
    server 127.0.0.1:3003 down;  # виключено для технічного
обслуговування
}
```

Рис. 6.d – Директива down для виключення сервера з балансування

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити, що backend3 (порт 3003) відсутній у розподілі:

```
for i in $(seq 1 6); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Видалити директиву `down` та відновити повне балансування після тестування:

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
```

```
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ | jq -r
.port; done | sort | uniq -c
```

Зробити висновки за отриманими результатами.

7. Додаткові налаштування балансування та моніторинг.

Конфігурація production-балансувальника включає ряд важливих параметрів, що виходять за межі базового балансування. Кастомні сторінки помилок покращують UX при аваріях, таймаути запобігають зависанню з'єднань при повільних або недоступних бекендах, буферизація підвищує пропускну здатність при роботі з повільними клієнтами, а розширені логи з upstream-змінними є основою для моніторингу та відлагодження.

a. Налаштувати кастомну сторінку помилки 502 Bad Gateway.

Помилка 502 Bad Gateway виникає, коли Nginx не може встановити з'єднання з жодним із серверів у блоці upstream — всі недоступні або повертають помилки. За замовчуванням Nginx показує просту службову сторінку «502 Bad Gateway». Для кращого користувацького досвіду рекомендується замінити її на кастомну HTML-сторінку з інформативним повідомленням мовою сайту.

Директива `proxy_intercept_errors on` дозволяє Nginx перехоплювати HTTP-помилки (4xx, 5xx) від upstream і обробляти їх через власний механізм error_page замість передачі клієнту «як є». Без цієї директиви error_page не спрацює для upstream-помилок.

Створити файл `/var/www/html/502.html`:

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Сервіс тимчасово недоступний</title>
    <style>
        body { font-family: Arial, sans-serif; text-align: center;
padding: 60px; }
        h1   { color: #e74c3c; font-size: 2em; margin-bottom: 0.3em; }
        p    { color: #555; font-size: 1.1em; }
        code { background: #f4f4f4; padding: 2px 6px; border-radius: 3px;
}
```

```
    </style>
</head>
<body>
    <h1>502 — Сервіс тимчасово недоступний</h1>
    <p>Бекенд-сервери не відповідають. Спробуйте через кілька хвилин.</p>
    <p><code>Error 502 Bad Gateway</code></p>
</body>
</html>
```

Рис. 7.a-html – Кастомна HTML-сторінка помилки 502

```
sudo vi /var/www/html/502.html
```

Додати директиви до блоку `server {}` у конфігурації (рис. 7.a):

```
server {
    listen 80;
    server_name lb-xxx-yyy-zzz.local;

    # Дозволити Nginx перехоплювати upstream-помилки
    proxy_intercept_errors on;

    # Прив'язати код 502 до файлу 502.html
    error_page 502 /502.html;

    # internal — файл доступний тільки для внутрішніх перенаправлень
    location = /502.html {
        root /var/www/html;
        internal;
    }

    location / {
        proxy_pass         http://backend_pool;
        proxy_set_header   Host              $host;
        proxy_set_header   X-Real-IP         $remote_addr;
        proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
        proxy_set_header   X-Forwarded-Proto $scheme;
    }
}
```

Рис. 7.a – Конфігурація кастомної сторінки помилки 502

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

Протестувати кастомну сторінку — зупинити всі бекенди та надіслати запит:

```
docker stop backend1 backend2 backend3
curl -s http://lb-xxx-yyy-zzz.local/
```

Відновити всі бекенди після тестування:

```
docker start backend1 backend2 backend3
```

Зробити висновки за отриманими результатами.

b. Налаштувати таймаути проксі-з'єднань для запобігання зависанню запитів.

Без явних таймаутів Nginx може чекати відповіді від бекенду необмежено довго, блокуючи з'єднання та пам'ять. Три директиви таймаутів охоплюють різні фази взаємодії з бекендом і застосовуються незалежно.

proxy_connect_timeout — максимальний час (секунди) для встановлення TCP-з'єднання з бекендом. Якщо бекенд не відповідає на SYN-пакет, Nginx через цей час спробує наступний бекенд. Рекомендоване значення для локальної мережі: 3–10 секунд.

proxy_send_timeout — максимальний час між двома послідовними операціями запису у з'єднання з бекендом (передача тіла запиту). Не є загальним часом передачі — таймер скидається кожного разу, коли дані передаються.

proxy_read_timeout — максимальний час між двома послідовними операціями читання відповіді від бекенду. Таймер скидається при кожному отриманому пакеті. Повинен бути достатньо великим для обробки найповільніших запитів у системі.

```
location / {
    proxy_pass         http://backend_pool;
    proxy_set_header   Host              $host;
    proxy_set_header   X-Real-IP         $remote_addr;
    proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;

    # Час на встановлення TCP-з'єднання з бекендом
    proxy_connect_timeout  5s;

    # Час між операціями запису до бекенду
    proxy_send_timeout    10s;

    # Час між операціями читання відповіді від бекенду
    proxy_read_timeout    10s;
}
```

Рис. 7.b – Конфігурація таймаутів proxy_connect/send/read_timeout

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

c. Налаштувати буферизацію відповідей від upstream для підвищення продуктивності.

Буферизація (proxy_buffering) — режим, коли Nginx отримує повну відповідь від бекенду у пам'ять або тимчасовий файл, звільняє з'єднання з бекендом, і лише потім починає передавати відповідь клієнту. Це дозволяє бекенду обробити запит і повернути відповідь максимально швидко, не чекаючи доки повільний клієнт отримає всі дані по мережі.

Без буферизації (proxy_buffering off) з'єднання з бекендом залишається відкритим протягом усього часу передачі відповіді клієнту. При повільному клієнті це може займати секунди та споживати ресурси бекенду. Буферизація особливо важлива для мобільних клієнтів та клієнтів з повільним з'єднанням.

```
location / {
    proxy_pass         http://backend_pool;
    proxy_set_header   Host              $host;
    proxy_set_header   X-Real-IP         $remote_addr;
    proxy_set_header   X-Forwarded-For   $proxy_add_x_forwarded_for;
    proxy_set_header   X-Forwarded-Proto $scheme;

    proxy_connect_timeout  5s;
    proxy_send_timeout    10s;
    proxy_read_timeout    10s;

    # Увімкнути буферизацію відповідей від бекенду
    proxy_buffering      on;

    # Буфер для першої частини відповіді (заголовки, зазвичай < 4 КБ)
    proxy_buffer_size    4k;

    # 8 буферів × 16 КБ = 128 КБ для тіла відповіді
    proxy_buffers        8 16k;
}
```

Рис. 7.c – Конфігурація буферизації відповідей від upstream

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

d. Налаштувати розширений формат логів з upstream-змінними.

Стандартний формат логів Nginx (combined) не містить жодної інформації про upstream: не видно, який бекенд відповів, скільки часу

витратив, і чи була помилка на боці бекенду. Для повноцінного моніторингу балансувальника необхідні спеціальні змінні Nginx.

Ключові upstream-змінні Nginx: `$upstream_addr` — адреса та порт бекенду, що обробив запит (може бути кілька через кому, якщо Nginx робив retry); `$upstream_status` — HTTP-статус відповіді від бекенду; `$upstream_response_time` — час відповіді бекенду у секундах (дробове число). Аналіз цих значень дозволяє виявляти повільні бекенди, помилки та аномалії у розподілі.

Додати новий формат логів до `/etc/nginx/nginx.conf` у блок `http {}` (рис. 7.d):

```
# Розширений формат логів з інформацією про upstream
log_format upstream_info
    '$remote_addr - $remote_user [$time_local] '
    '"$request" $status $body_bytes_sent '
    '"$http_referer" "$http_user_agent" '
    'upstream: $upstream_addr '
    'upstream_status: $upstream_status '
    'upstream_time: $upstream_response_time s';
```

Рис. 7.d – Формат логів upstream_info у /etc/nginx/nginx.conf

```
sudo vi /etc/nginx/nginx.conf
```

Застосувати розширений формат у `sites-available/lb-xxx-yyy-zzz` — повна фінальна конфігурація (рис. 7.d-site):

```
upstream backend_pool {
    server 127.0.0.1:3001 max_fails=3 fail_timeout=15s;
    server 127.0.0.1:3002 max_fails=3 fail_timeout=15s;
    server 127.0.0.1:3003 max_fails=3 fail_timeout=15s;
}

server {
    listen 80;
    server_name lb-xxx-yyy-zzz.local;

    # Розширений лог з upstream-змінними
    access_log /var/log/nginx/lb-access.log upstream_info;
    error_log  /var/log/nginx/lb-error.log  warn;

    proxy_intercept_errors on;
    error_page 502 /502.html;
    location = /502.html { root /var/www/html; internal; }

    location / {
        proxy_pass            http://backend_pool;
        proxy_set_header      Host             $host;
```

```
        proxy_set_header       X-Real-IP         $remote_addr;
        proxy_set_header       X-Forwarded-For
$proxy_add_x_forwarded_for;
        proxy_set_header       X-Forwarded-Proto $scheme;
        proxy_connect_timeout  5s;
        proxy_send_timeout    10s;
        proxy_read_timeout    10s;
        proxy_buffering        on;
        proxy_buffer_size      4k;
        proxy_buffers          8 16k;
    }
}
```

Рис. 7.d-site – Повна фінальна конфігурація балансувальника з upstream-логуванням

```
sudo vi /etc/nginx/sites-available/lb-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
```

e. Перевірити розширене логування: надіслати серію запитів та проаналізувати лог.

Надіслати 9 запитів для генерації достатньої кількості записів у новому форматі:

```
for i in $(seq 1 9); do curl -s http://lb-xxx-yyy-zzz.local/ >
/dev/null; done
```

Переглянути останні записи лога `/var/log/nginx/lb-access.log`:

```
sudo tail -10 /var/log/nginx/lb-access.log
```

Витягти лише upstream-адреси для аналізу розподілу запитів між бекендами:

```
sudo tail -20 /var/log/nginx/lb-access.log | grep -oP 'upstream: \K[^
]+'
```

Витягти часи відповідей від upstream (у секундах) для аналізу продуктивності:

```
sudo tail -20 /var/log/nginx/lb-access.log | grep -oP 'upstream_time:
\K[0-9.]+'
```

Порахувати кількість запитів до кожного upstream-порту (аналіз балансування):

```
sudo tail -20 /var/log/nginx/lb-access.log | grep -oP 'upstream:
127\.0\.0\.1:\K[0-9]+' | sort | uniq -c
```

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo tail -5 /var/log/nginx/lb-access.log
```

Зробити висновки за отриманими результатами.

8. Зупинка контейнерів та завершення роботи.

Після завершення практичної роботи необхідно коректно зупинити та видалити всі Docker-контейнери для звільнення ресурсів системи (пам'ять, процесор, мережеві порти). Docker-образ backend-app залишається у локальному реєстрі і може бути використаний у наступних практичних роботах без повторної збірки.

Розуміння різниці між docker stop та docker rm є важливим: docker stop надсилає сигнал SIGTERM контейнеру і чекає до 10 секунд на graceful shutdown, потім надсилає SIGKILL. docker rm видаляє зупинений контейнер і звільняє його ресурси, але не видаляє образ. docker rmi видаляє образ (не використовується тут для збереження образу).

a. Зупинити та видалити всі бекенд-контейнери.

Команда `docker stop` коректно завершує контейнери (SIGTERM, потім SIGKILL через 10 с). Команда `docker rm` видаляє зупинені контейнери: звільняє ресурси і дозволяє знову використовувати ті самі імена при наступному запуску. Docker-образ `backend-app` залишається у локальному реєстрі.

```
docker stop backend1 backend2 backend3
docker rm backend1 backend2 backend3
```

Перевірити, що жодного контейнера з іменем backend не залишилося:

Перевірити застосовану конфігурацію, виконавши команду:

```
docker ps -a --filter name=backend
```

Зробити висновки за отриманими результатами.

b. Завершити SSH-сесію та зупинити Vagrant VM.

Вийти з SSH-сесії та повернутися на хост-машину:

```
exit
```

Зупинити VM командою з директорії, де розміщений Vagrantfile:

```
vagrant halt
```

Контрольні запитання

1. Яку роль відіграє директива upstream у конфігурації Nginx? Перелічити та описати параметри кожного сервера у блоці upstream (weight, max_fails, fail_timeout, backup, down) і пояснити їх взаємодію.
2. У чому полягає відмінність між алгоритмами Round Robin та Least Connections? За яких умов Least Connections є ефективнішим і чому Round Robin може спричинити нерівномірне навантаження?
3. Як відбувається автоматичне виключення та відновлення бекенду при використанні параметрів max_fails та fail_timeout? Опишіть покроковий механізм пасивного health check у Nginx.
4. Яким чином директива backup відрізняється від директиви down? У яких виробничих сценаріях доцільно використовувати кожну з них? Наведіть конкретні приклади.
5. Які переваги та обмеження алгоритму ip_hash порівняно з Round Robin? Яка проблема виникає при відмові бекенду в схемі з ip_hash та як її вирішити архітектурно?
6. Яку роль відіграють змінні $upstream_addr, $upstream_response_time та $upstream_status у логуванні Nginx? Як ці дані використовуються для аналізу продуктивності та діагностики балансувальника?
7. У чому полягає призначення директив proxy_buffering, proxy_buffer_size та proxy_buffers? Як буферизація впливає на тривалість з'єднань з бекендом та продуктивність при роботі з повільними клієнтами?
8. Яким чином Dockerfile дозволяє запускати кілька ідентичних контейнерів на різних портах з одного образу? Поясніть роль змінної середовища PORT та прапорців -p і -e у команді docker run.
9. Що таке proxy_intercept_errors і для чого застосовується директива error_page у контексті проксі-балансування? Яка різниця між помилками, що генерує сам Nginx, і помилками від upstream?
10. Порівняйте три таймаути Nginx: proxy_connect_timeout, proxy_send_timeout та proxy_read_timeout. Що відбувається при перевищенні кожного з них і яке значення рекомендоване для типового API-сервісу?