Практична робота №5

КЕШУВАННЯ У NGINX: PROXY CACHE, БРАУЗЕРНЕ КЕШУВАННЯ ТА FASTCGI CACHE

Мета заняття: набути практичних навичок з налаштування різних механізмів кешування у Nginx: браузерного кешування через директиви Cache-Control, Expires, ETag та Last-Modified; серверного кешування відповідей через proxy_cache з перевіркою HIT/MISS; кешування PHP-відповідей через FastCGI cache; дослідити вплив кешування на продуктивність за допомогою навантажувального тестування; навчитися налаштовувати обхід кешу, керування застарілим кешем та розширене логування.

Теоретичні відомості

Кешування — це механізм збереження копій відповідей у тимчасовому сховищі для повторного використання без звернення до оригінального джерела даних. У вебінфраструктурі виділяють два основних рівні кешування: клієнтське (браузерне) та серверне. Правильно налаштоване кешування може зменшити навантаження на бекенд-сервери у 5–100 разів і суттєво скоротити час відповіді для кінцевого користувача.

Браузерне кешування (client-side caching) контролюється HTTP-заголовками відповіді: Cache-Control визначає стратегію кешування (max-age, no-cache, no-store, public, private); Expires задає абсолютний час закінчення дії кешу (застарілий механізм, Cache-Control має пріоритет); ETag — унікальний ідентифікатор версії ресурсу для умовних запитів (If-None-Match); Last-Modified — час останньої зміни ресурсу для умовних запитів (If-Modified-Since). Браузер самостійно вирішує, чи відправляти новий запит, чи використовувати кешовану копію.

Proxy cache (серверне кешування) у Nginx зберігає відповіді від upstream-бекендів у файловій системі або RAM. Nginx перевіряє, чи є відповідь у кеші: якщо так — повертає її клієнту без звернення до бекенду (HIT); якщо ні —

запитує бекенд, зберігає відповідь у кеші і повертає її клієнту (MISS). Статус кешу відображається через змінну $upstream_cache_status.

FastCGI cache — аналог proxy_cache, але для застосунків, що спілкуються з Nginx через протокол FastCGI (PHP-FPM, Python через uwsgi). Конфігурується директивами fastcgi_cache_path і fastcgi_cache замість proxy_cache_path і proxy_cache. FastCGI cache особливо ефективний для PHP-сайтів з інтенсивними обчисленнями або зверненнями до бази даних, де кешування PHP-відповіді може зменшити час генерації сторінки з 200 мс до кількох мілісекунд.

Завдання на роботу

1. Підготовка інфраструктури: Node.js та PHP-FPM бекенди у Docker.

Для дослідження кешування розгортаються два різних типи бекендів. Node.js-бекенд слугуватиме для тестування proxy_cache: його відповідь містить випадкове число та timestamp, що дозволяє наочно відрізнити закешовану відповідь від свіжої. PHP-FPM-бекенд слугуватиме для тестування FastCGI cache.

a. Встановити Docker, Node.js, PHP-клієнт та тестові утиліти.

Для цієї практичної роботи знадобляться Docker для запуску контейнерів, jq для аналізу JSON-відповідей, curl для HTTP-запитів з аналізом заголовків, apache2-utils для навантажувального тестування утилітою ab (Apache Bench). Також встановлюється php-fpm на хості для демонстрації роботи FastCGI-сокету.

```
sudo apt update && sudo apt install -y docker.io nodejs npm curl jq
apache2-utils
sudo systemctl enable --now docker
sudo usermod -aG docker vagrant && exit
```

Після exit виконати `vagrant ssh` для відновлення сесії з членством у групі docker.

Перевірити застосовану конфігурацію, виконавши команду:

```
docker --version && ab -V | head -1
```

Зробити висновки за отриманими результатами.

b. Створити Node.js-бекенд з випадковим числом у відповіді та зібрати Docker-образ.

Відповідь містить поле random — нове випадкове число при кожному зверненні до бекенду. Якщо Nginx повертає кешовану відповідь, поле random не змінюватиметься між запитами — це є головним індикатором роботи кешу.

Створити директорію та файл `~/node-backend/app.js` (рис. 1.b):

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

const PORT = parseInt(process.env.PORT || '3001', 10);

const server = http.createServer((req, res) => {
  if (req.method !== 'GET' || req.url !== '/') {
    res.writeHead(404);
    res.end(JSON.stringify({ error: 'Not Found' }));
    return;
  }
  const body = JSON.stringify({
    hostname:  os.hostname(),
    timestamp: new Date().toISOString(),
    random:    Math.floor(Math.random() * 1000000)  // індикатор
кешування
  }, null, 2);
  res.writeHead(200, {
    'Content-Type':   'application/json',
    'Content-Length': Buffer.byteLength(body)
  });
  res.end(body);
});

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

Рис. 1.b – Файл app.js — Node.js бекенд з полем random для тестування кешу

```
mkdir -p ~/node-backend && sudo vi ~/node-backend/app.js
```

Створити `~/node-backend/Dockerfile`:

```
FROM node:alpine
WORKDIR /app
COPY app.js .
EXPOSE 3001
CMD ["node", "app.js"]
```

Рис. 1.b-df – Dockerfile для Node.js бекенду

```
sudo vi ~/node-backend/Dockerfile
cd ~/node-backend && docker build -t node-backend .
docker run -d --name node-app -p 3001:3001 node-backend
```

Перевірити застосовану конфігурацію, виконавши команду:

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

Зробити висновки за отриманими результатами.

c. Розгорнути PHP-FPM бекенд у Docker-контейнері з монтуванням PHP-скрипту.

PHP-FPM (FastCGI Process Manager) — менеджер процесів для виконання PHP-скриптів через протокол FastCGI. На відміну від Node.js, PHP-FPM не надає HTTP-сервер — Nginx підключається до нього безпосередньо через FastCGI-сокет або TCP-порт і передає запити для виконання PHP-коду.

Створити директорію та PHP-скрипт `~/php-backend/index.php` (рис. 1.c):

```
<?php
// Діагностичний PHP-скрипт для тестування FastCGI cache
header('Content-Type: application/json');

echo json_encode([
    'hostname'    => gethostname(),
    'timestamp'   => date('c'),
    'php_version' => phpversion(),
    'random'      => rand(0, 999999),  // індикатор кешування
], JSON_PRETTY_PRINT);
```

Рис. 1.c – Файл index.php — PHP-скрипт для тестування FastCGI cache

```
mkdir -p ~/php-backend && sudo vi ~/php-backend/index.php
```

Запустити контейнер `php:fpm-alpine` з монтуванням директорії скрипту.

PHP-FPM слухатиме на порту 9000 (стандартний FastCGI-порт):

```
docker run -d --name php-fpm -p 9000:9000 -v ~/php-
backend:/var/www/html php:fpm-alpine
```

Перевірити застосовану конфігурацію, виконавши команду:

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

Зробити висновки за отриманими результатами.

d. Додати доменне ім'я до `/etc/hosts` та підготувати статичний сайт для тестування браузерного кешування.

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

Створити директорію статичного сайту та набір файлів для тестування різних типів кешу. Кожен тип файлів (HTML, CSS, JS, зображення) матиме різні правила кешування:

```
sudo mkdir -p /var/www/cache-site/{css,js,img}
```

Створити тестову HTML-сторінку `/var/www/cache-site/index.html` (рис. 1.d):

```
<!DOCTYPE html>
<html lang="uk">
<head>
    <meta charset="UTF-8">
    <title>Тест кешування Nginx</title>
    <link rel="stylesheet" href="/css/style.css">
</head>
<body>
    <h1>Практична робота №5 — Кешування Nginx</h1>
    <p>Статичний HTML — кешується браузером (no-cache).</p>
    <script src="/js/app.js"></script>
</body>
</html>
```

Рис. 1.d-html – Тестова HTML-сторінка для перевірки браузерного кешування

```
sudo vi /var/www/cache-site/index.html
```

Створити CSS-файл `/var/www/cache-site/css/style.css`:

```
echo "body { font-family: Arial, sans-serif; margin: 2em; }" | sudo
tee /var/www/cache-site/css/style.css
```

Створити JS-файл `/var/www/cache-site/js/app.js`:

```
echo "console.log('Cache test app loaded');" | sudo tee
/var/www/cache-site/js/app.js
```

Скопіювати будь-яке зображення для тестування або створити тестовий файл:

```
sudo dd if=/dev/urandom bs=1024 count=10 2>/dev/null | base64 | head -
20 | sudo tee /var/www/cache-site/img/test.jpg > /dev/null
```

Перевірити застосовану конфігурацію, виконавши команду:

```
ls -la /var/www/cache-site/ && ls -la /var/www/cache-
site/{css,js,img}/
```

Зробити висновки за отриманими результатами.

2. Браузерне кешування (клієнтське): Cache-Control, Expires, ETag, Last-Modified.

Браузерне кешування контролюється HTTP-заголовками відповіді сервера. Коли браузер отримує ресурс із заголовком Cache-Control: max-age=604800, він зберігає копію локально на 7 днів і не надсилає запит до сервера протягом

цього часу. Після закінчення терміну браузер виконує умовний запит (conditional request) з заголовком If-None-Match або If-Modified-Since — сервер повертає 304 Not Modified без тіла відповіді, якщо ресурс не змінився, що економить трафік.

Nginx надає директиву expires для встановлення терміну дії кешу та директиву add_header для довільних HTTP-заголовків. ETag та Last-Modified Nginx генерує автоматично для статичних файлів — вмикати їх окремо не потрібно. Стратегія кешування залежить від типу ресурсу: статичні активи з хешем у імені (style.abc123.css) можуть кешуватися роками; HTML-документи зазвичай не кешуються або кешуються коротко, щоб браузер завжди отримував актуальну структуру сторінки.

a. Створити конфігурацію Nginx з правилами браузерного кешування для різних типів файлів.

   Директива `expires` встановлює заголовки `Expires` та `Cache-Control: max-age`. Значення `30d` означає 30 днів, `7d` — 7 днів. Значення `-1` встановлює `Expires: -1` та `Cache-Control: no-cache`, примушуючи браузер перевіряти свіжість ресурсу.

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

    root /var/www/cache-site;
    index index.html;

    # Для HTML — завжди перевіряти свіжість (no-cache не означає "не
кешувати",
    # а означає "перевіряти перед використанням")
    location ~* \.html$ {
        expires -1;
        add_header Cache-Control "no-cache, must-revalidate";
    }

    # CSS і JavaScript — кешувати 7 днів
    location ~* \.(css|js)$ {
        expires 7d;
        add_header Cache-Control "public, max-age=604800, immutable";
    }

    # Зображення — кешувати 30 днів
    location ~* \.(jpg|jpeg|png|gif|ico|svg|webp)$ {
```

```
        expires 30d;
        add_header Cache-Control "public, max-age=2592000";
    }

    # API — ніколи не кешувати
    location /api/ {
        expires off;
        add_header Cache-Control "no-store, no-cache, must-revalidate";
        add_header Pragma "no-cache";
        proxy_pass http://127.0.0.1:3001/;
    }

    # Статичний контент за замовчуванням
    location / {
        try_files $uri $uri/ =404;
    }
}
```

Рис. 2.a – Конфігурація Nginx з правилами браузерного кешування для різних типів файлів

```
sudo vi /etc/nginx/sites-available/cache-xxx-yyy-zzz
sudo ln -s /etc/nginx/sites-available/cache-xxx-yyy-zzz
/etc/nginx/sites-enabled/
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo nginx -T 2>/dev/null | grep -A 5 expires
```

Зробити висновки за отриманими результатами.

b. Перевірити HTTP-заголовки кешування для різних типів файлів.

Прапорець `-I` у curl виводить тільки HTTP-заголовки (HEAD-запит). Перевірити заголовки для CSS, JS, зображення та HTML окремо — кожен тип має отримати різне значення Cache-Control та Expires.

```
curl -I http://cache-xxx-yyy-zzz.local/css/style.css
curl -I http://cache-xxx-yyy-zzz.local/js/app.js
curl -I http://cache-xxx-yyy-zzz.local/img/test.jpg
curl -I http://cache-xxx-yyy-zzz.local/index.html
```

Порівняти значення Cache-Control та Expires між типами файлів:

```
for f in css/style.css js/app.js img/test.jpg index.html; do echo "===
$f ==="; curl -sI http://cache-xxx-yyy-zzz.local/$f | grep -i "cache-
control\|expires\|etag\|last-modified"; done
```

Очікувані результати: для `style.css` та `app.js` — `Cache-Control: public, max-age=604800, immutable` та `Expires` через 7 днів; для `test.jpg` — `Cache-Control: public, max-age=2592000` та `Expires` через 30 днів; для `index.html` — `Cache-Control: no-cache, must-revalidate` та `Expires: -1`.

Директива `immutable` у заголовку `Cache-Control` є розширенням, яке сигналізує браузеру, що ресурс не змінюватиметься протягом усього терміну кешування. Браузери, що підтримують це розширення (Chrome, Firefox), не надсилатимуть умовних запитів (If-None-Match) навіть після очищення кешу, якщо термін не минув. Це підходить лише для файлів з хешем у назві (наприклад, `style.a1b2c3.css`).

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -sI http://cache-xxx-yyy-zzz.local/css/style.css | grep -i cache-control
```

Зробити висновки за отриманими результатами.

c. Перевірити механізм умовних запитів: ETag, If-None-Match та відповідь 304 Not Modified.

Nginx автоматично генерує заголовок ETag для статичних файлів на основі розміру файлу та часу останньої зміни. При повторному запиті браузер надсилає значення ETag у заголовку If-None-Match. Якщо файл не змінився, Nginx повертає 304 Not Modified без тіла відповіді — це суттєво економить трафік для великих файлів.

Виконати перший запит та зберегти значення ETag:

```
ETAG=$(curl -sI http://cache-xxx-yyy-zzz.local/css/style.css | grep -i etag | awk '{print $2}' | tr -d '\r')
echo "ETag: $ETAG"
```

Надіслати умовний запит з `If-None-Match`: очікується відповідь `304 Not Modified`:

```
curl -I --header "If-None-Match: $ETAG" http://cache-xxx-yyy-zzz.local/css/style.css
```

Аналогічно перевірити умовний запит через `If-Modified-Since`:

```
LM=$(curl -sI http://cache-xxx-yyy-zzz.local/css/style.css | grep -i last-modified | cut -d: -f2- | tr -d '\r')
curl -I --header "If-Modified-Since:$LM" http://cache-xxx-yyy-zzz.local/css/style.css
```

Відповідь `304 Not Modified` не містить тіла — браузер використовує локально збережену копію файлу. Для файлу розміром 1 МБ це означає економію 1 МБ трафіку на кожен повторний запит. На відміну від `200 OK` з тілом відповіді, `304` передає лише заголовки (зазвичай менше 1 КБ),

що критично для сайтів з великою кількістю статичних ресурсів і мобільних користувачів з обмеженим трафіком.

Важливо розрізняти два механізми умовних запитів. `ETag / If-None-Match` — сильна валідація: порівнюється точний ідентифікатор версії файлу. `Last-Modified / If-Modified-Since` — слабка валідація: порівнюється час зміни з точністю до секунди. Nginx генерує обидва заголовки автоматично для статичних файлів. При наявності обох, `ETag` має пріоритет відповідно до специфікації HTTP/1.1.

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -sI http://cache-xxx-yyy-zzz.local/css/style.css | grep -i "etag\|last-modified"
```

Зробити висновки за отриманими результатами.

d. Перевірити, що маршрут `/api/` проксюється до Node.js-бекенду без кешування.

Два послідовних запити до `/api/` повинні повертати різні значення поля `random` — підтвердження того, що відповідь не кешується і кожен запит проходить до бекенду. Заголовок `Cache-Control: no-store` забороняє збереження відповіді у будь-якому кеші.

```
curl -s http://cache-xxx-yyy-zzz.local/api/ | jq .random
curl -s http://cache-xxx-yyy-zzz.local/api/ | jq .random
```

Порівняти два значення random — вони мають відрізнятися:

```
curl -sI http://cache-xxx-yyy-zzz.local/api/ | grep -i "cache-control\|pragma"
```

Перевірити застосовану конфігурацію, виконавши команду:

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

Зробити висновки за отриманими результатами.

3. Proxy cache: серверне кешування відповідей від upstream-бекенду.

Proxy cache у Nginx зберігає відповіді HTTP-бекендів на диску або в RAM. Це принципово відрізняється від браузерного кешування: браузерний кеш зберігається на машині кінцевого користувача і недоступний іншим

користувачам; proxy cache — на сервері Nginx, тому одна закешована відповідь обслуговує всіх клієнтів, що запитують той самий ресурс.

Налаштування proxy cache складається з двох рівнів: глобального (у блоці http {}) — де оголошується зона кешу через proxy_cache_path; та рівня location — де вмикається кешування для конкретного шляху через proxy_cache і встановлюються правила TTL через proxy_cache_valid. Змінна $upstream_cache_status повертає статус кожного запиту: MISS (перший запит, збережено у кеш), HIT (відповідь з кешу), EXPIRED (термін минув), BYPASS (обхід кешу), STALE (застарілий кеш при помилці бекенду).

a. Додати директиву `proxy_cache_path` до блоку `http {}` у файлі `/etc/nginx/nginx.conf`.

   Директива **proxy_cache_path** оголошує зону кешу. Параметри: `/var/cache/nginx/proxy` — шлях до директорії кешу на диску; `levels=1:2` — дворівнева ієрархія директорій (розподіляє файли, щоб уникнути тисяч файлів в одній директорії); `keys_zone=PROXY:10m` — ім'я зони та розмір shared memory для зберігання ключів (10 МБ = ~80 000 ключів); `max_size=100m` — максимальний розмір кешу на диску; `inactive=60m` — видаляти елементи, до яких не зверталися 60 хвилин; `use_temp_path=off` — записувати безпосередньо у кеш (без тимчасового файлу).

```
# Додати у блок http { } файлу /etc/nginx/nginx.conf
proxy_cache_path /var/cache/nginx/proxy
    levels=1:2
    keys_zone=PROXY:10m
    max_size=100m
    inactive=60m
    use_temp_path=off;
```

Рис. 3.a – Оголошення зони proxy_cache у блоці http {}

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

Створити директорію кешу та встановити правильного власника:

```
sudo mkdir -p /var/cache/nginx/proxy
sudo chown www-data:www-data /var/cache/nginx/proxy
sudo nginx -t && sudo systemctl reload nginx
```

Перевірити застосовану конфігурацію, виконавши команду:

```
ls -la /var/cache/nginx/
```

Зробити висновки за отриманими результатами.

Зверніть увагу на структуру зберігання кешу. Директорія `/var/cache/nginx/proxy` є коренем кешу. Усередині Nginx автоматично створює дворівневу ієрархію: наприклад, файл із ключем, що завершується на `ab`, буде розміщений у шляху `/var/cache/nginx/proxy/b/a/<full_hash>`. Такий розподіл дозволяє уникнути проблем продуктивності файлової системи при зберіганні мільйонів файлів в одній директорії.

Параметр `keys_zone` є найважливішим: він визначає не лише ім'я зони, а й виділяє пам'ять для зберігання хеш-таблиці ключів. Ця хеш-таблиця дозволяє Nginx за мікросекунди перевірити, чи існує запис у кеші, без читання файлів з диску. 10 МБ shared memory достатньо приблизно для 80 000 ключів — цього цілком вистачає для більшості production-сайтів.

Параметр `inactive=60m` слід відрізняти від `proxy_cache_valid`. `proxy_cache_valid` визначає, коли запис вважається застарілим (expired) і вимагає оновлення від бекенду. `inactive` визначає, через який час запис видаляється з кешу взагалі, якщо до нього не зверталися — незалежно від того, чи є він ще валідним.

b. Додати блок `location /cache-api/` з увімкненим proxy cache до конфігурації сайту.

Директиви proxy cache у блоці location: **proxy_cache PROXY** — вказати ім'я зони (оголошено у nginx.conf); **proxy_cache_valid 200 5m** — кешувати успішні відповіді 5 хвилин; **proxy_cache_valid 404 1m** — кешувати відповіді 404 на 1 хвилину; **proxy_cache_key** — ключ кешу (унікальний ідентифікатор записуваної відповіді); **add_header X-Cache-Status** — передати статус кешу клієнту для діагностики.

```
# Додати у блок server {} файлу sites-available/cache-xxx-yyy-zzz

# Проксі з кешуванням — тестування proxy_cache
location /cache-api/ {
    # Вказати зону кешу (оголошена у nginx.conf)
    proxy_cache            PROXY;

    # TTL кешу: 200 OK → 5 хвилин, 404 → 1 хвилина
    proxy_cache_valid      200 5m;
    proxy_cache_valid      404 1m;

    # Ключ кешу: схема + хост + URI
    proxy_cache_key        "$scheme$host$request_uri";

    # Передати статус кешу у заголовку відповіді
    add_header             X-Cache-Status $upstream_cache_status;

    # Заголовки для бекенду
    proxy_set_header       Host              $host;
    proxy_set_header       X-Real-IP         $remote_addr;

    # Передати запит до Node.js бекенду
    proxy_pass             http://127.0.0.1:3001/;
}
```

Рис. 3.b – Location з proxy cache для проксювання до Node.js бекенду

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

Зверніть увагу на директиву `proxy_cache_key`. Ключ кешу — це рядок, за яким Nginx ідентифікує запит у кеші. Якщо два різних запити мають однаковий ключ, Nginx поверне для них одну й ту саму закешовану відповідь. Тому до ключа обов'язково включають `$scheme` (щоб розрізняти HTTP і HTTPS), `$host` (для підтримки кількох віртуальних хостів) і `$request_uri` (шлях і параметри запиту).

Якщо не включити `$scheme` до ключа кешу, то закешована відповідь для `http://example.com/api` буде повернута і для `https://example.com/api`, що може призвести до проблем з безпекою (HTTPS-відповідь, що сигналізує браузеру про використання HSTS, не повинна потрапляти до кешу для HTTP-запитів).

c. Перевірити роботу proxy cache: спостерігати перехід MISS → HIT.

Перший запит до закешованого location завжди MISS: Nginx звертається до бекенду, отримує відповідь, зберігає у кеш і повертає клієнту. Наступні запити до закінчення TTL повертають HIT: Nginx читає з кешу, не звертаючись до бекенду. Поле random у відповіді не змінюватиметься при HIT.

Перший запит (очікується `X-Cache-Status: MISS`):

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
curl -s http://cache-xxx-yyy-zzz.local/cache-api/ | jq .random
```

Другий запит (очікується `X-Cache-Status: HIT`, random не змінився):

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
curl -s http://cache-xxx-yyy-zzz.local/cache-api/ | jq .random
```

Надіслати 5 запитів і спостерігати статус кешу:

```
for i in $(seq 1 5); do echo -n "Запит $i: "; curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep -i x-cache-status; done
```

Переглянути структуру кешу на файловій системі:

```
sudo find /var/cache/nginx/proxy -type f | head -5
sudo du -sh /var/cache/nginx/proxy
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
```

Зробити висновки за отриманими результатами.

d. Налаштувати обхід кешу через `proxy_cache_bypass` та `proxy_no_cache`.

proxy_cache_bypass визначає умови, за яких Nginx не повертає відповідь з кешу (завжди йде до бекенду, але може зберегти нову відповідь у кеш). proxy_no_cache визначає умови, за яких відповідь не зберігається у кеш. Обидві директиви приймають змінні: якщо значення змінної не порожнє і не "0" — умова активна.

Додати обхід кешу при наявності параметра запиту `nocache=1` або cookie `nocache` (рис. 3.d):

```
location /cache-api/ {
    proxy_cache             PROXY;
    proxy_cache_valid       200 5m;
    proxy_cache_valid       404 1m;
    proxy_cache_key         "$scheme$host$request_uri";
    add_header              X-Cache-Status $upstream_cache_status;

    # Обхід кешу за параметром URL або cookie
    # $arg_nocache — значення параметра nocache= у URL
    # $cookie_nocache — значення cookie з іменем nocache
    proxy_cache_bypass      $arg_nocache $cookie_nocache;
    proxy_no_cache          $arg_nocache $cookie_nocache;

    proxy_set_header        Host              $host;
    proxy_set_header        X-Real-IP         $remote_addr;
    proxy_pass              http://127.0.0.1:3001/;
}
```

Рис. 3.d – Обхід proxy cache через параметр URL та cookie

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

Перевірити: запит з `?nocache=1` повинен повертати `BYPASS`, навіть якщо відповідь є у кеші:

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
curl -sI "http://cache-xxx-yyy-zzz.local/cache-api/?nocache=1" | grep X-Cache-Status
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -sI "http://cache-xxx-yyy-zzz.local/cache-api/?nocache=1" | grep X-Cache-Status
```

Зробити висновки за отриманими результатами.

e. Очистити кеш вручну та спостерігати повторний MISS.

Nginx не має вбудованої команди для очищення окремих записів кешу у відкритій версії (це функція Nginx Plus). Найпростіший спосіб очистити кеш — видалити файли у директорії кешу та перевантажити Nginx. Після очищення перший запит до будь-якого закешованого URL знову поверне MISS.

```
sudo rm -rf /var/cache/nginx/proxy/*
sudo systemctl reload nginx
```

Переконатися, що після очищення перший запит знову MISS:

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
```

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
```

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo du -sh /var/cache/nginx/proxy
```

Зробити висновки за отриманими результатами.

4. Тонке налаштування proxy cache: min_uses, lock, stale, background_update.

Базове налаштування proxy cache підходить для більшості випадків, але production-середовища потребують додаткових налаштувань. proxy_cache_min_uses запобігає кешуванню рідкісних запитів. proxy_cache_lock вирішує проблему cache stampede (шторм кешу), коли при закінченні TTL сотні паралельних запитів одночасно потрапляють до бекенду. proxy_cache_use_stale дозволяє Nginx повертати застарілі дані при помилках бекенду, забезпечуючи часткову відмовостійкість.

a. Налаштувати `proxy_cache_min_uses` — кешувати після N запитів.

Директива **proxy_cache_min_uses=3** означає: кешувати відповідь тільки після того, як URL запитано тричі. Перші два запити завжди ідуть до бекенду (статус MISS). Лише з третього запиту відповідь зберігається у кеш — наступні запити повернуть HIT. Це запобігає заповненню кешу рідкісними запитами, залишаючи місце для популярних URL.

```
location /cache-api/ {
    proxy_cache             PROXY;
    proxy_cache_valid       200 5m;
    proxy_cache_valid       404 1m;
    proxy_cache_key         "$scheme$host$request_uri";
    add_header              X-Cache-Status $upstream_cache_status;

    # Кешувати тільки після 3-го запиту до одного URL
    proxy_cache_min_uses    3;

    proxy_cache_bypass      $arg_nocache $cookie_nocache;
    proxy_no_cache          $arg_nocache $cookie_nocache;

    proxy_set_header        Host              $host;
    proxy_set_header        X-Real-IP         $remote_addr;
    proxy_pass              http://127.0.0.1:3001/;
}
```

Рис. 4.a – Директива proxy_cache_min_uses=3

```
sudo vi /etc/nginx/sites-available/cache-xxx-yyy-zzz
sudo nginx -t && sudo systemctl reload nginx
sudo rm -rf /var/cache/nginx/proxy/*
```

Надіслати 5 запитів та спостерігати: перші два MISS, третій та далі — HIT:

```
for i in $(seq 1 5); do echo -n "Запит $i: "; curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep -i x-cache; done
```

Перші два запити поверне `MISS` — Nginx звертається до бекенду, але ще не кешує відповідь, оскільки мінімальна кількість звернень (proxy_cache_min_uses=3) ще не досягнута. Третій запит знову `MISS`, але цього разу відповідь зберігається у кеш. Четвертий і п'ятий запити повертають `HIT` — Nginx повертає кешовану відповідь без звернення до бекенду.

Значення `proxy_cache_min_uses` слід підбирати виходячи з характеру трафіку. Для API з великою кількістю унікальних URL (пошукові запити, персональні дані) рекомендується встановлювати значення 3–5, щоб уникнути зберігання рідкісних запитів і ефективніше використовувати ресурси кешу.

Перевірити застосовану конфігурацію, виконавши команду:

```
for i in $(seq 1 4); do curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep -i x-cache-status; done
```

Зробити висновки за отриманими результатами.

b. Увімкнути `proxy_cache_lock` для запобігання cache stampede.

Cache stampede (шторм кешу, або thundering herd) — проблема, коли термін дії популярного запису у кеші закінчується і тисячі паралельних запитів одночасно намагаються отримати нову відповідь від бекенду, перевантажуючи його. Директива proxy_cache_lock вирішує цю проблему: тільки один запит проходить до бекенду, решта чекає завершення та отримує відповідь з кешу.

```
location /cache-api/ {
    proxy_cache            PROXY;
    proxy_cache_valid      200 5m;
    proxy_cache_key        "$scheme$host$request_uri";
    add_header             X-Cache-Status $upstream_cache_status;
    proxy_cache_min_uses   3;
```

```
    # Запобігання cache stampede:
    # тільки 1 запит іде до бекенду, решта чекають результату
    proxy_cache_lock       on;

    # Максимальний час очікування для заблокованих запитів
    proxy_cache_lock_timeout 5s;

    proxy_cache_bypass     $arg_nocache $cookie_nocache;
    proxy_no_cache         $arg_nocache $cookie_nocache;
    proxy_set_header       Host              $host;
    proxy_set_header       X-Real-IP         $remote_addr;
    proxy_pass             http://127.0.0.1:3001/;
}
```

Рис. 4.b – Директива proxy_cache_lock для запобігання cache stampede

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

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo nginx -T 2>/dev/null | grep cache_lock
```

Зробити висновки за отриманими результатами.

c. Налаштувати `proxy_cache_use_stale` — повернення застарілого кешу при помилці бекенду.

proxy_cache_use_stale дозволяє Nginx повертати клієнту застарілий (прострочений) запис з кешу замість помилки, якщо бекенд недоступний або повертає помилку. Статус у відповіді при цьому — STALE. Це підвищує відмовостійкість: при тимчасовій недоступності бекенду користувачі бачать "дещо застарілий" контент замість сторінки 502 Bad Gateway.

```
location /cache-api/ {
    proxy_cache            PROXY;
    proxy_cache_valid      200 5m;
    proxy_cache_key        "$scheme$host$request_uri";
    add_header             X-Cache-Status $upstream_cache_status;
    proxy_cache_min_uses   3;
    proxy_cache_lock       on;
    proxy_cache_lock_timeout 5s;

    # Повертати застарілий кеш при цих умовах:
    # error  — помилка підключення до бекенду
    # timeout — таймаут очікування відповіді від бекенду
    # http_502 http_503 http_504 — HTTP-помилки від бекенду
    proxy_cache_use_stale  error timeout http_502 http_503 http_504;

    # Оновити кеш у фоні при зверненні до застарілого запису
    proxy_cache_background_update on;

    proxy_cache_bypass     $arg_nocache $cookie_nocache;
    proxy_no_cache         $arg_nocache $cookie_nocache;
    proxy_set_header       Host              $host;
    proxy_set_header       X-Real-IP         $remote_addr;
    proxy_pass             http://127.0.0.1:3001/;
}
```

Рис. 4.c – Повна конфігурація proxy cache з use_stale та background_update

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

Прогріти кеш (забезпечити min_uses запитів):

```
for i in $(seq 1 4); do curl -s http://cache-xxx-yyy-zzz.local/cache-api/ > /dev/null; done
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
```

Зупинити Node.js бекенд та переконатися, що Nginx повертає `STALE` замість 502:

```
docker stop node-app
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep -i "x-cache\|http"
curl -s http://cache-xxx-yyy-zzz.local/cache-api/ | jq .random
```

Відновити бекенд після тестування:

```
docker start node-app
```

Директива `proxy_cache_background_update on` дозволяє Nginx оновлювати застарілий запис у кеші у фоновому режимі, не змушуючи користувача чекати. Коли користувач запитує застарілий запис, Nginx: (1) негайно повертає застарілу відповідь із статусом `STALE`; (2) одночасно запускає фоновий запит до бекенду для оновлення кешу. Наступний запит вже отримає свіжу відповідь із статусом `HIT`.

Поєднання `proxy_cache_use_stale` і `proxy_cache_background_update` є рекомендованою практикою для high-availability сервісів. При тимчасових збоях бекенду користувачі продовжують отримувати відповіді (хоч і дещо застарілі), а після відновлення бекенду кеш автоматично оновлюється у фоні без додаткових запитів.

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -sI http://cache-xxx-yyy-zzz.local/cache-api/ | grep X-Cache-Status
```

Зробити висновки за отриманими результатами.

d. Оголосити окрему зону кешу для API та перевірити ізоляцію зон.

Різні зони кешу (keys_zone) дозволяють налаштовувати незалежні параметри для різних груп URL: різний TTL, розмір, поведінку при помилках. Наприклад, зона для статичного API з великим TTL та зона для динамічних даних з малим TTL. Оголошення двох зон виконується у блоці http {} файлу nginx.conf (рис. 4.d):

```
# Зона для статичного API (довгий TTL)
proxy_cache_path /var/cache/nginx/static_api
    levels=1:2 keys_zone=STATIC_API:5m max_size=50m inactive=1h
use_temp_path=off;

# Зона для динамічних даних (короткий TTL)
proxy_cache_path /var/cache/nginx/dynamic_api
    levels=1:2 keys_zone=DYNAMIC_API:5m max_size=50m inactive=5m
use_temp_path=off;
```

Рис. 4.d – Оголошення двох незалежних зон proxy cache у nginx.conf

```
sudo vi /etc/nginx/nginx.conf
sudo mkdir -p /var/cache/nginx/static_api /var/cache/nginx/dynamic_api
sudo chown www-data:www-data /var/cache/nginx/static_api
/var/cache/nginx/dynamic_api
sudo nginx -t && sudo systemctl reload nginx
```

Для використання зони `STATIC_API` у конфігурації сайту необхідно змінити директиву `proxy_cache` у відповідному блоці location. Наприклад, для статичних ресурсів API, що рідко змінюються, слід вказати `proxy_cache STATIC_API;` та встановити довгий TTL: `proxy_cache_valid 200 1h;`. Для ендпоінтів з динамічними даними — `proxy_cache DYNAMIC_API;` та короткий TTL: `proxy_cache_valid 200 30s;`.

Ізоляція зон кешу також надає операційну перевагу: можна очистити лише одну зону, не зачіпаючи іншу. Наприклад, при деплої нових версій API достатньо видалити вміст директорії відповідної зони, тоді як статичні ресурси продовжують обслуговуватися з кешу. Це особливо цінно у великих проектах, де різні команди відповідають за різні частини API.

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo nginx -T 2>/dev/null | grep keys_zone
```

Зробити висновки за отриманими результатами.

5. Налаштувати FastCGI-кеш для PHP-FPM бекенду.

FastCGI-кеш призначений для зберігання відповідей від FastCGI-застосунків, таких як PHP-FPM. На відміну від proxy_cache, який кешує відповіді HTTP-проксі, fastcgi_cache працює безпосередньо на рівні протоколу FastCGI, що дозволяє уникнути зайвих HTTP-перетворень і ефективніше кешувати динамічно згенерований PHP-контент.

Основна відмінність між proxy_cache і fastcgi_cache полягає у способі взаємодії: proxy_cache використовується для запитів до HTTP-серверів (upstream), тоді як fastcgi_cache — для FastCGI-процесів. Обидва механізми мають схожу архітектуру зберігання та схожі директиви управління, тому отримані в попередніх завданнях навички повністю застосовні і тут.

a. Оголосити зону fastcgi_cache_path у блоці http{ } файлу /etc/nginx/nginx.conf (рис. 5.a):

   Відкрити конфігураційний файл:

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

   Додати директиву `fastcgi_cache_path` усередині блоку `http { ... }` після існуючого оголошення proxy_cache_path (рис. 5.a):

```
# Зона FastCGI-кешу
fastcgi_cache_path /var/cache/nginx/fastcgi
    levels=1:2
    keys_zone=FASTCGI:10m
    max_size=100m
    inactive=30m
    use_temp_path=off;
```

Рис. 5.a – Оголошення зони fastcgi_cache_path у блоці http

Параметр `keys_zone=FASTCGI:10m` виділяє 10 МБ оперативної пам'яті для зберігання ключів кешу. Параметр `max_size=100m` обмежує розмір кешу на диску. Параметр `inactive=30m` видаляє записи, до яких не було звернень протягом 30 хвилин.

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo nginx -t
```

b. Налаштувати location для PHP-файлів із підтримкою FastCGI-кешу у файлі `/etc/nginx/sites-available/cache-xxx-yyy-zzz.local` (рис. 5.b):

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

Додати або замінити блок location для PHP (рис. 5.b):

```
location ~ \.php$ {
    # Підключення FastCGI-кешу
    fastcgi_cache            FASTCGI;
    fastcgi_cache_valid       200 5m;   # Кешувати успішні відповіді на 5
хвилин
    fastcgi_cache_valid       404 1m;   # Кешувати "не знайдено" на 1
хвилину
    fastcgi_cache_key         "$scheme$request_method$host$request_uri";
    fastcgi_cache_min_uses    1;        # Кешувати після першого запиту
    fastcgi_cache_lock        on;       # Захист від cache stampede

    # Обхід кешу за параметром або cookie
    fastcgi_cache_bypass      $arg_nocache $cookie_nocache;
    fastcgi_no_cache          $arg_nocache $cookie_nocache;

    # Заголовок статусу кешу
    add_header X-FastCGI-Cache $upstream_cache_status;

    # FastCGI-параметри для PHP-FPM
    include        fastcgi_params;
    fastcgi_pass   127.0.0.1:9000;
    fastcgi_index  index.php;
    fastcgi_param  SCRIPT_FILENAME /var/www/html$fastcgi_script_name;
}
```

Рис. 5.b – Конфігурація location ~ \.php$ із FastCGI-кешем

Директива `fastcgi_pass 127.0.0.1:9000` передає запити до контейнера php-fpm, який прослуховує порт 9000. Директива `fastcgi_param SCRIPT_FILENAME` повідомляє PHP-FPM, який файл виконувати — шлях відповідає директорії, змонтованій у контейнер.

c. Застосувати конфігурацію та переконатися у коректності налаштувань:

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

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep -i "x-fastcgi-cache"
```

Зробити висновки за отриманими результатами.

Перший запит повинен повернути статус MISS (запис ще відсутній у кеші). Повторний запит через кілька секунд повинен повернути статус HIT.

d. Перевірити послідовність MISS → HIT для FastCGI-кешу:

Виконати два послідовних запити та порівняти заголовки:

```
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep -E "X-FastCGI-Cache|random"
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep -E "X-FastCGI-Cache|random"
```

Перший запит: `X-FastCGI-Cache: MISS` — Nginx звертається до PHP-FPM і зберігає відповідь у кеші. Другий запит: `X-FastCGI-Cache: HIT` — Nginx повертає кешовану відповідь, не звертаючись до PHP-FPM. Значення поля `random` у відповідях має бути однаковим, що підтверджує роботу кешу.

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep "X-FastCGI-Cache"
```

Зробити висновки за отриманими результатами.

e. Перевірити механізм обходу FastCGI-кешу:

Для примусового обходу кешу передати параметр запиту nocache=1:

```
curl -si "http://cache-xxx-yyy-zzz.local/index.php?nocache=1" | grep -E "X-FastCGI-Cache|random"
```

Відповідь повинна містити `X-FastCGI-Cache: BYPASS`, а значення `random` — відрізнятися від попередніх, що свідчить про прямий виклик PHP-FPM в обхід кешу.

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -si "http://cache-xxx-yyy-zzz.local/index.php?nocache=1" | grep "X-FastCGI-Cache"
```

Зробити висновки за отриманими результатами.

f. Перевірити фізичне розташування файлів FastCGI-кешу на диску:

```
sudo find /var/cache/nginx/fastcgi -type f | head -10
```

Виведені шляхи демонструють дворівневу структуру каталогів (параметр levels=1:2). Кожен файл містить серіалізовану HTTP-відповідь разом із службовими метаданими (час закінчення дії, ключ кешу).

Для перегляду метаданих конкретного файлу кешу можна скористатися командою: `sudo head -20 /var/cache/nginx/fastcgi/<path_to_file>`. Перші рядки файлу містять заголовок із ключем кешу, часом закінчення дії (TTL) у форматі Unix timestamp, а далі — серіалізована HTTP-відповідь: рядок статусу, заголовки та тіло відповіді.

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo du -sh /var/cache/nginx/fastcgi
```

Зробити висновки за отриманими результатами.

g. Налаштувати `fastcgi_cache_use_stale` — повернення застарілого FastCGI-кешу при помилці PHP-FPM.

Аналогічно до proxy_cache_use_stale у попередньому завданні, директива fastcgi_cache_use_stale дозволяє повертати застарілий вміст кешу при помилках PHP-FPM. Це важливо для PHP-сайтів: якщо PHP-FPM тимчасово перевантажений або недоступний, Nginx поверне закешовану сторінку замість помилки 502 Bad Gateway, забезпечуючи безперервність сервісу для кінцевого користувача.

Доповнити блок `location ~ \.php$` директивами відмовостійкості (рис. 5.g):

```
location ~ \.php$ {
    fastcgi_cache            FASTCGI;
    fastcgi_cache_valid       200 5m;
    fastcgi_cache_key         "$scheme$request_method$host$request_uri";
    fastcgi_cache_min_uses    1;
    fastcgi_cache_lock        on;
    fastcgi_cache_bypass      $arg_nocache $cookie_nocache;
    fastcgi_no_cache          $arg_nocache $cookie_nocache;
    add_header X-FastCGI-Cache $upstream_cache_status;

    # Повертати застарілий кеш при помилках PHP-FPM
    fastcgi_cache_use_stale   error timeout invalid_header updating
                              http_500 http_503;

    include        fastcgi_params;
    fastcgi_pass   127.0.0.1:9000;
    fastcgi_index  index.php;
    fastcgi_param  SCRIPT_FILENAME /var/www/html$fastcgi_script_name;
}
```

Рис. 5.g – Директива fastcgi_cache_use_stale для відмовостійкого FastCGI-кешу

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

Прогріти FastCGI-кеш, зупинити контейнер PHP-FPM і переконатися, що Nginx повертає застарілу відповідь:

```
for i in $(seq 1 3); do curl -s http://cache-xxx-yyy-
zzz.local/index.php > /dev/null; done
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep "X-FastCGI-
Cache"
docker stop php-fpm
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep -E "X-
FastCGI-Cache|HTTP"
```

При зупиненому PHP-FPM відповідь повинна мати заголовок `X-FastCGI-Cache: STALE` замість 502. Це підтверджує, що Nginx повертає застарілий вміст кешу і не пропускає помилку до кінцевого користувача.

Відновити PHP-FPM після тестування:

```
docker start php-fpm
```

Перевірити застосовану конфігурацію, виконавши команду:

```
curl -si http://cache-xxx-yyy-zzz.local/index.php | grep "X-FastCGI-
Cache"
```

Зробити висновки за отриманими результатами.

6. Провести навантажувальне тестування та порівняти продуктивність із кешем і без нього.

Навантажувальне тестування є обов'язковим етапом оцінки ефективності кешування. Утиліта ab (Apache Bench) генерує серію HTTP-запитів та вимірює ключові показники: Requests per second (кількість запитів за секунду) і Time per request (середній час обробки одного запиту). Порівнюючи результати до і після прогріву кешу, можна кількісно оцінити приріст продуктивності.

a. Встановити утиліту ab (Apache Bench) зі складу пакета apache2-utils:

```
sudo apt install -y apache2-utils
```

Перевірити застосовану конфігурацію, виконавши команду:

```
ab -V
```

Зробити висновки за отриманими результатами.

b. Розширити формат access-логу Nginx для запису статусу кешу та часу відповіді бекенду (рис. 6.b):

Відкрити файл `/etc/nginx/nginx.conf` та додати новий формат логу в блок `http { ... }`:

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

```
# Розширений формат логу з даними кешування
log_format cache_log '$remote_addr - $remote_user [$time_local] '
                     '"$request" $status $body_bytes_sent '
                     '"$http_referer" "$http_user_agent" '
                     'cache_status=$upstream_cache_status '
                     'upstream_time=$upstream_response_time';
```

Рис. 6.b – Формат логу з полями upstream_cache_status та upstream_response_time

Активувати новий формат логу у блоці `server` файлу `cache-xxx-yyy-zzz`:

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

```
server {
    listen 80;
    server_name cache-xxx-yyy-zzz.local;
    access_log /var/log/nginx/cache-access.log cache_log;
    error_log  /var/log/nginx/cache-error.log;
    # ... решта конфігурації ...
}
```

Рис. 6.b2 – Підключення формату cache_log у блоці server

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

Перевірити застосовану конфігурацію, виконавши команду:

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

Зробити висновки за отриманими результатами.

c. Виміряти базову продуктивність proxy_cache БЕЗ прогріву кешу:

Очистити кеш та виконати тест:

```
sudo rm -rf /var/cache/nginx/proxy/*
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/cache-api/
```

Параметр `-n 200` задає загальну кількість запитів, `-c 10` — кількість одночасних з'єднань. Записати значення `Requests per second` та `Time per request` для порівняння.

Перевірити застосовану конфігурацію, виконавши команду:

```
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/cache-api/ 2>&1 | grep
-E "Requests per second|Time per request"
```

Зробити висновки за отриманими результатами.

d. Прогріти proxy_cache та виміряти продуктивність ІЗ кешем:

Виконати кілька запитів для наповнення кешу, після чого повторити тест:

```
for i in $(seq 1 5); do curl -s http://cache-xxx-yyy-zzz.local/cache-
api/ > /dev/null; done
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/cache-api/
```

Після прогріву кешу показник `Requests per second` повинен суттєво зрости, а `Time per request` — значно зменшитися порівняно з результатами з підзавдання c.

Типові результати при тестуванні на локальній Vagrant VM: без кешу — 50–200 RPS (залежить від Node.js і ресурсів VM); з прогрітим кешем — 500–5000 RPS, оскільки Nginx обслуговує запити з диска або RAM без запуску Node.js. Зростання у 10–50 разів є типовим для proxy_cache з простими JSON-відповідями.

Зверніть увагу на рядок `Transfer rate` у виведенні ab. При обслуговуванні з кешу пропускна здатність значно вища, оскільки Nginx не чекає на Node.js-процес. Також рядок `Percentage of the requests served within a certain time` (таблиця перцентилів) показує

максимальний час відповіді — з кешем хвіст розподілу (95-й, 99-й перцентиль) суттєво коротший.

Перевірити застосовану конфігурацію, виконавши команду:

```
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/cache-api/ 2>&1 | grep
-E "Requests per second|Time per request"
```

Зробити висновки за отриманими результатами.

e. Виміряти базову продуктивність FastCGI-кешу БЕЗ прогріву:

```
sudo rm -rf /var/cache/nginx/fastcgi/*
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/index.php
```

Записати отримані показники для порівняння.

Перевірити застосовану конфігурацію, виконавши команду:

```
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/index.php 2>&1 | grep -
E "Requests per second|Time per request"
```

Зробити висновки за отриманими результатами.

f. Прогріти FastCGI-кеш та виміряти продуктивність ІЗ кешем:

```
for i in $(seq 1 5); do curl -s http://cache-xxx-yyy-
zzz.local/index.php > /dev/null; done
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/index.php
```

Порівняти отримані значення з попередніми та зробити висновок про ефективність FastCGI-кешу для PHP-вмісту.

Перевірити застосовану конфігурацію, виконавши команду:

```
ab -n 200 -c 10 http://cache-xxx-yyy-zzz.local/index.php 2>&1 | grep -
E "Requests per second|Time per request"
```

Зробити висновки за отриманими результатами.

g. Проаналізувати записи access-логу після навантажувального тестування:

Підрахувати кількість запитів із різними статусами кешу:

```
sudo awk -F'cache_status=' '{print $2}' /var/log/nginx/cache-
access.log | awk '{print $1}' | sort | uniq -c | sort -rn
```

Вивести останні записи розширеного логу для ручного аналізу:

```
sudo tail -20 /var/log/nginx/cache-access.log
```

У виведенні слід звернути увагу на співвідношення HIT / MISS / BYPASS. Високий відсоток HIT свідчить про ефективну роботу кешу. Поле `upstream_time` показує реальний час відповіді бекенду — для HIT-

запитів це поле зазвичай дорівнює - або 0, оскільки Nginx не звертається до бекенду.

Для розрахунку середнього часу відповіді бекенду (лише для MISS-запитів) можна скористатися командою:

```
sudo awk '/cache_status=MISS/ {n++; split($NF, a, "="); sum+=a[2]} END {if(n>0) print "Avg upstream time (MISS):", sum/n "s, Requests:", n}' /var/log/nginx/cache-access.log
```

Ця команда виводить середній час відповіді бекенду виключно для запитів зі статусом MISS, що відображає реальну продуктивність бекенду без впливу кешу. Порівнявши цей показник із середнім часом відповіді для HIT (яке наближається до нуля), можна обчислити коефіцієнт прискорення для запитів, що обслуговуються з кешу.

Перевірити застосовану конфігурацію, виконавши команду:

```
sudo du -sh /var/cache/nginx/proxy/ /var/cache/nginx/fastcgi/
```

Зробити висновки за отриманими результатами.

7. Зупинити Docker-контейнери та завершити роботу віртуальної машини.

Після завершення практичної роботи необхідно коректно зупинити всі запущені сервіси, щоб звільнити ресурси хост-машини. Зупинка контейнерів виконується командою docker stop, після чого завершується робота самої віртуальної машини.

a. Переконатися, що обидва контейнери поточно запущені:

```
docker ps
```

У виведенні повинні бути наявні контейнери `node-app` і `php-fpm` зі статусом `Up`.

Перевірити застосовану конфігурацію, виконавши команду:

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

Зробити висновки за отриманими результатами.

b. Зупинити контейнер Node.js-бекенду:

```
docker stop node-app
```

Перевірити застосовану конфігурацію, виконавши команду:

```
docker ps -a --filter name=node-app --format "table {{.Names}}\t{{.Status}}"
```

Зробити висновки за отриманими результатами.

Статус контейнера має змінитися з `Up` на `Exited`.

c. Зупинити контейнер PHP-FPM-бекенду:

```
docker stop php-fpm
```

Перевірити застосовану конфігурацію, виконавши команду:

```
docker ps -a --filter name=php-fpm --format "table
{{.Names}}\t{{.Status}}"
```

Зробити висновки за отриманими результатами.

d. Перевірити, що всі контейнери зупинено, та очистити кеш:

```
docker ps
```

Список запущених контейнерів повинен бути порожнім (або не містити контейнерів практичної роботи). Очистити залишки кешу:

```
sudo rm -rf /var/cache/nginx/proxy/* /var/cache/nginx/fastcgi/*
```

Перевірити застосовану конфігурацію, виконавши команду:

```
docker ps && sudo du -sh /var/cache/nginx/
```

Зробити висновки за отриманими результатами.

e. Завершити роботу віртуальної машини Vagrant:

Виконати команду на хост-машині (не всередині VM) з директорії проекту Vagrant:

```
vagrant halt
```

Контрольні запитання

1. Яку роль відіграє директива fastcgi_cache_path у конфігурації Nginx і які параметри (keys_zone, max_size, inactive) вона приймає?
2. У чому полягає принципова відмінність між proxy_cache і fastcgi_cache з точки зору протоколу взаємодії Nginx з бекендом?
3. Як відбувається формування ключа кешу за допомогою директиви fastcgi_cache_key і чому важливо включати до нього змінні $scheme, $request_method і $host?
4. Які значення може повертати змінна $upstream_cache_status (HIT, MISS, BYPASS, EXPIRED, STALE, UPDATING) і в яких ситуаціях кожне з них виникає?
5. Яким чином директиви fastcgi_cache_bypass і fastcgi_no_cache дозволяють обходити кеш за певними умовами, наприклад за наявністю параметра запиту або cookie?
6. Як відбувається налаштування браузерного кешування в Nginx за допомогою директив expires і Cache-Control для різних типів ресурсів (HTML, CSS/JS, зображення)?
7. Яку роль відіграє механізм ETag і заголовок If-None-Match у процесі умовних HTTP-запитів, і як Nginx підтримує ці механізми за замовчуванням?
8. Які переваги надає директива proxy_cache_use_stale і в яких умовах повернення застарілої кешованої відповіді є прийнятним рішенням?
9. Яким чином утиліта ab (Apache Bench) вимірює показники Requests per second і Time per request, і як інтерпретувати різницю в цих показниках до та після прогріву кешу?
10. Як відбувається сегментація кешу на диску (параметр levels=1:2) і чому дворівнева структура каталогів є більш ефективною порівняно зі зберіганням усіх файлів в одному каталозі?