ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: **«ПІДВИЩЕННЯ ПРОДУКТИВНОСТІ РОБОТИ ВЕБ-СЕРВЕРА НА БАЗІ АНАЛІЗУ ЗАПИТІВ HTTP»**

на здобуття освітнього ступеня магістра
зі спеціальності 123 Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерні системи та мережі
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олексій ДОБРОВОЛЬСЬКИЙ
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр.КСДМ-61
Олексій ДОБРОВОЛЬСЬКИЙ
(ім'я, ПРІЗВИЩЕ)

Керівник: Андрій ЛЕМЕШКО
доктор філософії (PhD) *(ім'я, ПРІЗВИЩЕ)*

Рецензент: _____
науковий ступінь, вчене звання *(ім'я, ПРІЗВИЩЕ)*

Київ 2023

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерної інженерії
Ступінь вищої освіти «Магістр»

Спеціальність 123 Комп'ютерна інженерія
Освітньо-професійна прорама Комп'ютерні системи та мережі

ЗАТВЕРДЖУЮ
Завідувач кафедрою Комп'ютерної інженерії
Наталія ЛАЩЕВСЬКА
(ім'я, ПРІЗВИЩЕ)
"____" _____2023 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Добровольський Олексій Ігорович
(прізвище, ім'я, по батькові здобувача)

1.Тема кваліфікаційної роботи: Підвищення продуктивності роботи веб-сервера на базі аналізу запитів HTTP
керівник роботи Андрій ЛЕМЕШКО доктор філософії (PhD)
(ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від "19" 10 2023 р. №145

2. Строк подання кваліфікаційної роботи

3. Вихідні дані кваліфікаційної роботи:
3.1. Кешування данних.
3.2. Аналіз продуктивності веб-сервера.
3.3. Науково-технічна література.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити):
4.1. Аналіз методів підвищення продуктивності.
4.2. Дослідження методів підвищення продуктивності особливості функціонування.
4.3. Реалізація методів підвищення продуктивності веб сервера та їх аналіз запитами HTTP

5. Перелік ілюстраційного матеріалу: *презентація* .
6. Дата видачі завдання «19» жовтня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Приміт ка
1	Аналіз наявної науково-технічної літератури	19.10-05.11.23	Виконано
2	Вивчення матеріалів для кешування данних	05.11-12.11.23	Виконано
3	Дослідження технологій кешування	13.11-19.11.23	Виконано
4	Дослідження технологій роботи веб серверів	20.11-25.11.23	Виконано
5	Дослідження технологій підвищення продуктивності	27.11-03.12.23	Виконано
6	Аналіз HTTP запитів	04.12-10.12.23	Виконано
7	Оформлення роботи: вступ, висновки, реферат	11.12-20.12.23	Виконано
8	Розробка демонстраційних матеріалів	21.12-29.12.23	Виконано

Здобувач вищої освіти _____ Олексій ДОБРОВОЛЬСЬКИЙ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Андрій ЛЕМЕШКО__
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня магістра: 104 стор., 77 рис., 22 джерел.

Мета роботи – мета передбачає вивчення, аналіз та дослідження методів, які допоможуть підвищити продуктивність веб-сервера та впевнитися в цьому на базі аналізу HTTP-запитів, що має значний практичний вплив на оптимізацію роботи веб-серверів у сучасних умовах.

Об'єкт дослідження - процес обробки та відповіді на HTTP-запити, а також можливості оптимізації цього процесу через аналіз та застосування відповідних методів та технологій.

Предмет дослідження – процес оптимізації роботи веб-сервера з використанням аналізу запитів HTTP.

Короткий зміст роботи: Робота спрямована на дослідження та покращення продуктивності роботи веб серверу шляхом аналізу запитів HTTP. У роботі проведені методи з налаштування оптимізації веб серверу різними шляхами та проведений аналіз HTTP.

КЛЮЧОВІ СЛОВА: ПРОДУКТИВНІСТЬ ВЕБ СЕРВЕРУ, КЕШУВАННЯ, АНАЛІЗ HTTP, REST.

ABSTRACT

The text part of the qualification work for obtaining a master's degree: 104 pages, 77 figures, 22 sources.

The purpose of the work - the purpose involves the study, analysis and research of methods that will help to improve the performance of the web server and make sure of this based on the analysis of HTTP requests, which has a significant practical impact on the optimization of the work of web servers in modern conditions

The object of research - the process of processing and responding to HTTP requests, as well as the possibilities of optimizing this process through the analysis and application of appropriate methods and technologies.

The subject of research - the process of web server optimization using HTTP request analysis

Summary of the work: The work is aimed at researching and improving the performance of the web server by analyzing HTTP requests. In the work, the methods for setting up the optimization of the web server in various ways and the analysis of HTTP were carried out

KEY WORDS: WEB SERVER PERFORMANCE, CACHING, HTTP ANALYSIS, REST.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

HTTP - Протокол передачі гіпертексту (Hypertext Transfer Protocol)

REST - Представлення стану перенесення (Representational State Transfer)

RESTful - Архітектурний стиль веб-сервісів, який використовує принципи REST.

DevTools - Інструменти розробника (Developer Tools)

XML - Розширена мова розмітки (Extensible Markup Language)

PHP - Рекурсивний акронім "PHP: Hypertext Preprocessor", популярна мова програмування для розробки веб-додатків.

API - Інтерфейс програмування застосунків (Application Programming Interface)

cURL - Утиліта командного рядка для передачі даних за URL-адресою.

FreeBSD - Операційна система з відкритим кодом (Free Berkeley Software Distribution)

TLS - Протокол захищеного транспортного шару (Transport Layer Security)

TCP - Протокол керування передачею (Transmission Control Protocol)

CSS - Каскадні таблиці стилів (Cascading Style Sheets)

WiFi - Бездротові мережі (Wireless Fidelity)

Web - Глобальна павутиння (World Wide Web)

Nginx - Веб-сервер та проксі-сервер з відкритим кодом, який забезпечує швидку обробку запитів і високу масштабованість.

JS - JavaScript, популярна мова програмування, яка використовується для створення веб-сайтів та веб-додатків.

Java - Об'єктно-орієнтована мова програмування, що використовується для створення різноманітних додатків, включаючи веб-додатки.

ASCII - Американський стандарт кодування інформації, який використовується для представлення текстових символів у вигляді числових значень.

WAP - Протокол доступу до бездротових мереж (Wireless Application Protocol), який використовується для доступу до Інтернету на мобільних пристроях.

Apache - Веб-сервер з відкритим кодом, один із найпопулярніших в світі, використовується для веб-хостингу та обробки запитів на сервері.

ВСТУП

У сучасному цифровому світі потік і доступність інформації визначають успіх багатих організацій, тому підтримка продуктивності веб-провайдерів стала дуже важливою. Це не має нічого спільного зі швидкістю обробки запитів, але одним з основних аспектів оптимізації продуктивності веб-сервера є аналіз і оптимізація HTTP-запитів.

HTTP-запити є основною формою взаємодії клієнт-сервер в архітектурі веб-сервера. Міцність і структура цих додатків можуть значно вплинути на продуктивність сервера. У цьому контексті вступає в гру контроль: як ми можемо оптимізувати та покращити продуктивність веб-сервера шляхом детального аналізу та оптимізації HTTP-запитів?

У цій темі розглядаються очікувані показники продуктивності комплексного веб-сервера, а також аналіз і оптимізація вбудованих програм. Дотримуючись передових методів обробки HTTP-запитів, можна лише змінити годинник підключення до сервера та покращити продуктивність системи. У цьому контексті важливо розглянути різні стратегії для маршрутизації запитів, кешування та оптимізації для оптимізації виведення сервером для клієнтських запитів.

Дослідження та впровадження стратегій підвищення продуктивності веб-серверів на основі аналізу HTTP-запитів є необхідним з огляду на збільшення кількості Інтернет-ресурсів. Зусилля щодо забезпечення доступу до інформації залежать від належної оптимізації процесу обробки запитів на рівні веб-сервера. Аналіз вхідних HTTP-запитів показує нам можливість зменшити обробку ресурсів, зменшити затримку та оптимізувати шляхи передачі даних. Переглядаючи схеми використання ресурсів і тенденції погодинного попиту, ми можемо визначити слабкі місця серверних ботів і розробити ефективні стратегії їх усунення.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ

1.1 Веб сервери та їх роль у сучасному Інтернеті

Веб-сервери - це комп'ютери або спеціальні програми, які виконують роль серверів. Коли користувач намагається завантажити HTML-документ через адресний рядок, браузер надсилає запит через протокол передачі даних HTTP. Якщо запит досягає потрібних веб-серверів, HTTP-сервер (програмне забезпечення безпеки) надішле запитуваний документ також через HTTP.

Веб-сервер може бути статичним і динамічним. Статичний сервер завантажує лише файли, необхідні для перегляду. Dinamic також надсилає файли до браузера, але ця програма має встановлене програмне забезпечення для зміни вихідних файлів перед надсиланням їх у браузер. Фактично звітність відбувається на ходу — надсилається статистика, витягуються дані з бази даних тощо.

Apache є найпопулярнішим веб-сервером у світі. Однак багато великих маркетингових веб-сайтів вибирають Nginx або їх комбінацію. Наприклад, Nginx приймає запити у формі статичного файлу (файл зображення, CSS, JavaScript або XML), який вставляється, і, наприклад, скрипт PHP, він надсилає файл на сервер. Apache може обробляти PHP .

Якщо ви можете обійтися без локального HTTP-сервера в першу чергу для зовнішньої розробки, тоді він все одно підходить для серверної частини.

Поточний сайт — це не лише колекція HTML-документів, а й багато інших технологій, баз даних тощо.

Для використання серверних технологій важко використовувати правильний доступний в мережі Інтернет-сервер, тому необхідно встановити відповідний набір програм на локальний комп'ютер і розробити на ньому всі системи керування базами даних PHP і MySQL.

1.2 Принципи роботи HTTP протоколу

HTTP — це протокол для завантаження ресурсів, наприклад документів HTML. Він є основою для передачі всіх даних в Інтернеті та протоколу клієнт-клієнт, що означає, що запити ініціюються одержувачем, зазвичай веб-браузером. Ціла стаття складається з різних типів вмісту, як-от текст, опис макета, зображення, відео, статті тощо [5].

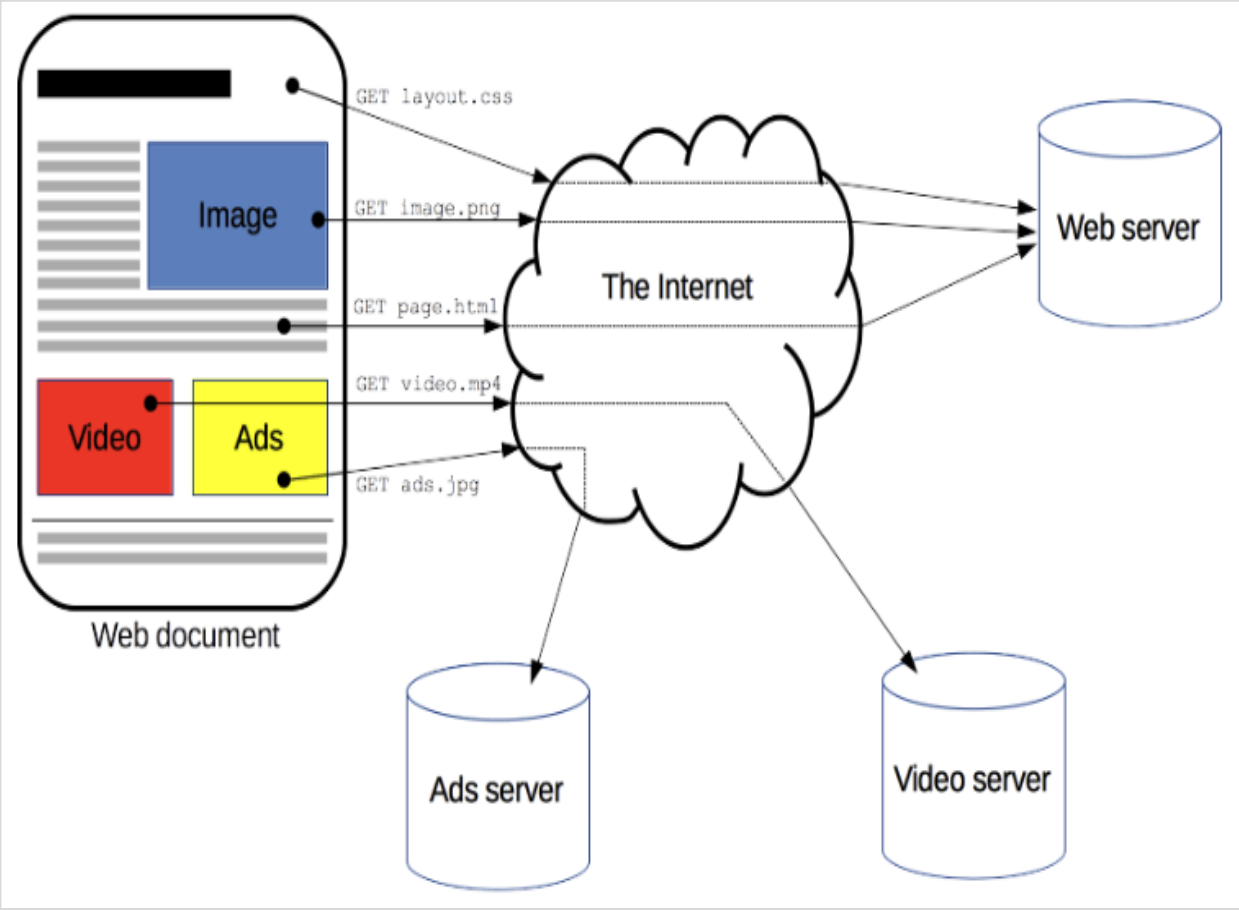


Рисунок 1.1 – Схема обміну даними через HTTP протокол

Клієнти та сервери спілкуються типом обміну конкретнимим повідомленнями (на відміну від потіку даних). Повідомлення, які надсилає клієнт, як правило, веб-браузер, називаються запитами , а повідомлення, які надсилає сервер як відповідь, називаються відповідями. Розробка в початку 90-х років HTTP є розширюваним протоколом, який з часом розвивався. Це протокол прикладного рівня, який надсилается через TCP або через TCP -з'єднання, зашифроване TLS, хоча теоретично можна використовувати будь-який надійний TCP протокол.

Завдяки своїй розширюваності він використовується не лише для отримання гіпертекстових документів, але й для зображень і відео або для публікації вмісту на серверах, як із результатами HTML-форм. HTTP також можна використовувати для отримання частин документів для оновлення веб-сторінок на вимогу [5].

HTTP (протокол передачі гіпертексту) є основним протоколом Всесвітньої павутини. Розроблений Тімом Бернерсом-Лі та його командою в період з 1989 по 1991 роки, протокол HTTP зазнав багатьох змін, які допомогли зберегти його простоту, одночасно сформувавши його гнучкість. Продовжуйте читати, щоб дізнатися, як HTTP розвинувся з протоколу, призначеного для обміну файлами в напівнадійному лабораторному середовищі, до сучасного інтернет-лабіринту, який передає зображення та граф контенту при великому розширенні.

1.2.1 Повідомлення HTTP

Повідомлення HTTP є засобом обміну даними між сервером і клієнтом. Існує два типи повідомлень: запити, які надсилаються клієнтом для ініціювання операції на сервері, і відповіді, які є відповідями від сервера. Повідомлення HTTP – це багаторядкові текстові повідомлення, закодовані ASCII. У HTTP/1.1 і попередніх версіях протоколу ці повідомлення надсилалися відкрито через з’єднання. У HTTP/2 зрозумілі людині повідомлення тепер розділені на кадри HTTP, що забезпечує оптимізацію та підвищення продуктивності. Веб-розробники та власники веб-сторінок рідко самі створюють ці текстові повідомлення HTTP: це роблять комп’ютери, веб-браузери, проксі-сервери чи веб-сервери. Він надсилає HTTP-повідомлення через файли конфігурації (для проксі або серверів), API (для браузерів) та інші інтерфейси [6].

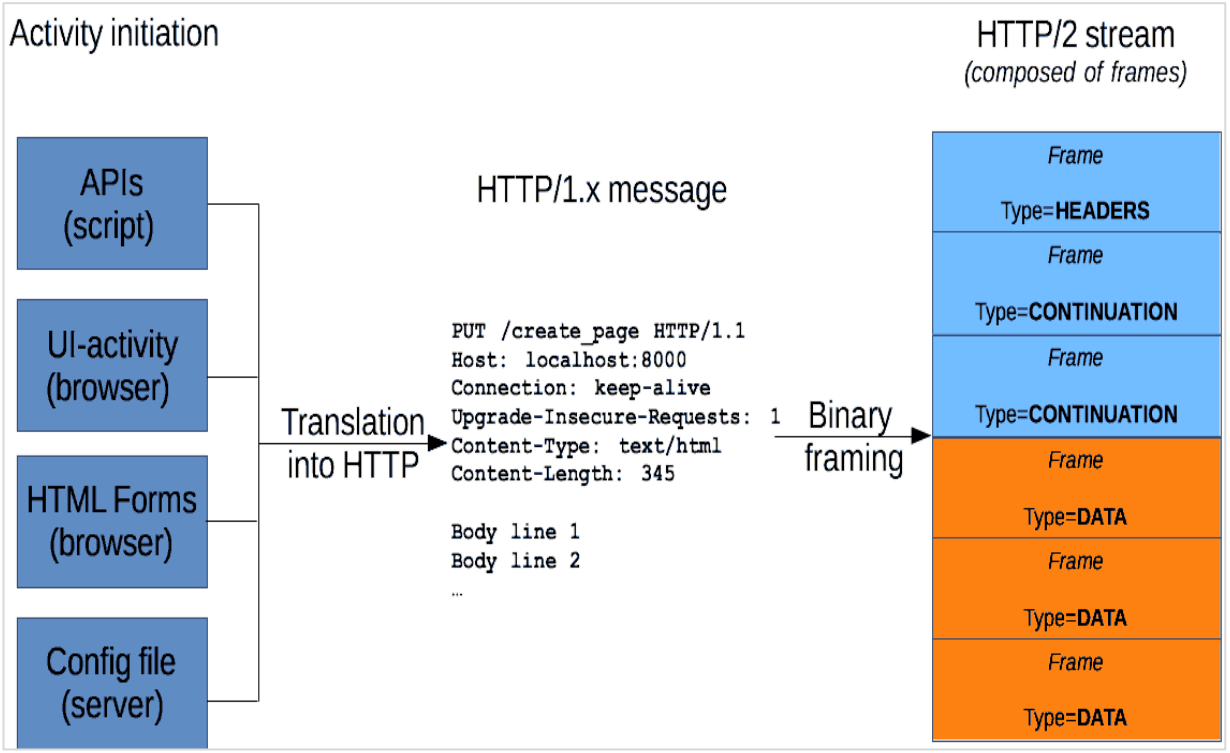


Рисунок 1.2 - Механізм двійкового фреймування HTTP/2

Механізм із двома фреймами HTTP/2 розроблений таким чином, щоб не вимагати власної мережі у реалізованих API або файлах конфігурації: він відкритий для користувача. HTTP-запити та перегляди структуровані подібним чином і включають:

a) початковий рядок, що описує запит, який буде виконано, його успіх чи невдачу. Цей перший рядок є одним рядком.

b) додатковий набір HTTP-заголовків , що вказує запит або описує тіло повідомлення.

c) передачу рядку із переліком усієї метаінформації про запит.

d) додаткове тіло, яке містить дані, пов’язані із запитом (наприклад, вміст форми HTML), або текст, пов’язаний із відповіддю. Наявність тіла та його розмір визначаються початковим рядком і HTTP-заголовками.

Початковий рядок і HTTP-заголовки HTTP-повідомлення спільно відомі як голова запитів , тоді як його корисне навантаження відоме як тіло[6].

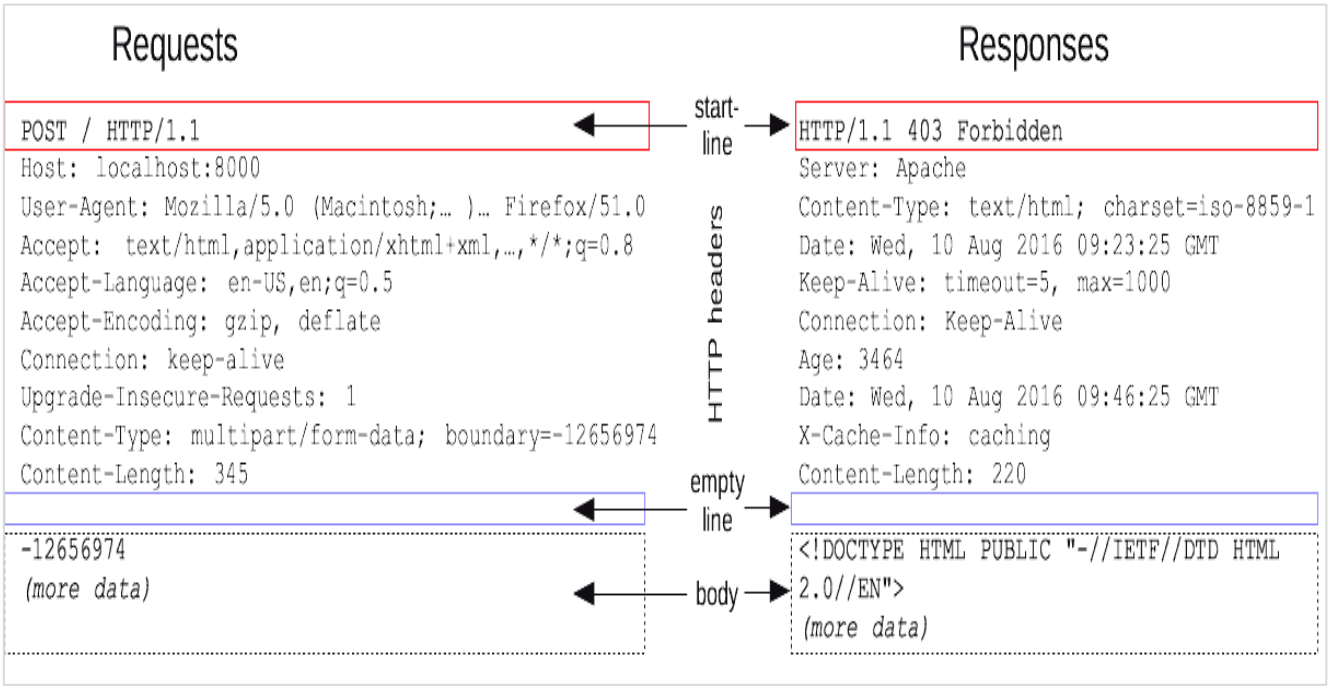


Рисунок 1.3 – Структура HTTP запити Requests/Responses

HTTP-запити — це повідомлення, надіслані клієнтом для ініціювання дії на сервері. Їх стартова лінія містить елементи:

1) метод HTTP, дієслово (наприклад, GET, PUT або POST) або іменник (наприклад, HEAD або OPTIONS), що описує дію, яку потрібно виконати. Наприклад, метод GET отримує ресурс, а метод POST надсилає дані на сервер (створюючи, змінюючи ресурс або створюючи тимчасовий документ для повернення);

2) адресат запиту, як правило, URL-адреса або точний шлях протоколу, порту та домену, вказаний у контексті запиту;

3) формат цільового запиту залежить від різних методів HTTP. Це може бути абсолютний шлях, за яким слідує '?'рядок запиту. Це є звичайна форма, відома як форма початкова , і використовується з методами GET, POST, HEADi .OPTIONS;

a) POST / HTTP/1.1

b) GET /background.png HTTP/1.0

c) HEAD /test.html?query=alibaba HTTP/1.1

d) OPTIONS /anypage.html HTTP/1.0

4) повний URL, також відомий як абсолютний шаблон, використовується GET під час підключення до проксі. Завантажити https://developer.mozilla.org/en-US/docs/Web/HTTP/Messages HTTP/1.1

5) частина перевірки URL-адреси, ім'я домену та порт (позначений «:»), називається шаблоном перевірки. Він використовується лише під час CONNECT під час налаштування HTTP-тунелю. ПРЕДСТАВЛЯЄМО developer.mozilla.org:80 HTTP/1.1

6) зірочка, проста зірочка ('*'), використовується з OPTIONS для позначення всього сервера. ПАРАМЕТРИ * HTTP/1.1

7) версія HTTP, яка описує структуру решти повідомлення, служить індикатором версії, яка, як очікується, буде використана у відповіді [7].

1.3 Клієнт серверна архітектура

З розвитком інструментів проектування програмного забезпечення та правильною розробкою того, як веб-програми та інші ресурси, до яких потрібен доступ. Розроблено користувацький інтерфейс для взаємодії з користувачем та вдосконалено підходи до розробки таких систем. Веб-додаток — це інтерактивний ресурс, який залучає користувача вводити, отримувати та керувати даними, а також виконувати ряд завдань, ряд завдань користувача та завдання в конкретному ресурсі. На відміну від традиційних веб-сайтів, вони в основному призначені для відображення вмісту та, звичайно, для показу більше. Іншими словами, веб-сервіси є програмними платформами [8].

По-перше, веб-служба за своєю суттю є програмою на стороні клієнта функціонал знаходиться на віддаленому сервері і повністю запускається на сервері, а візуальна частина - веб-інтерфейс запускається і відображається в браузері (Рисунок. 1.4) [8,9].

Створення веб-ресурсів передбачає розробку та використання наступного одиниці:

a) розробка архітектури веб-додатків;

b) техніки спілкування;

c) зберігання даних.

Усі сучасні веб-додатки є розподіленими системами.

Розподілена система - це система, в якій частини розподілені між загальними об'єктами вузли виконують певні завдання в залежності від своїх можливостей. Таким чином, структура веб-додатків поділяється на дві частини - клієнтську і серверну. Клієнт робить запит, отримує відповідь і генерується серверним компонентом. Натомість сторона сервера відповідає за обробку запитів, створення відповідей і надсилання відгуків клієнту. Таким чином, відповідь для клієнта веб-додатку буде у формі відповідної графічної форми (HTML, XML) [22].

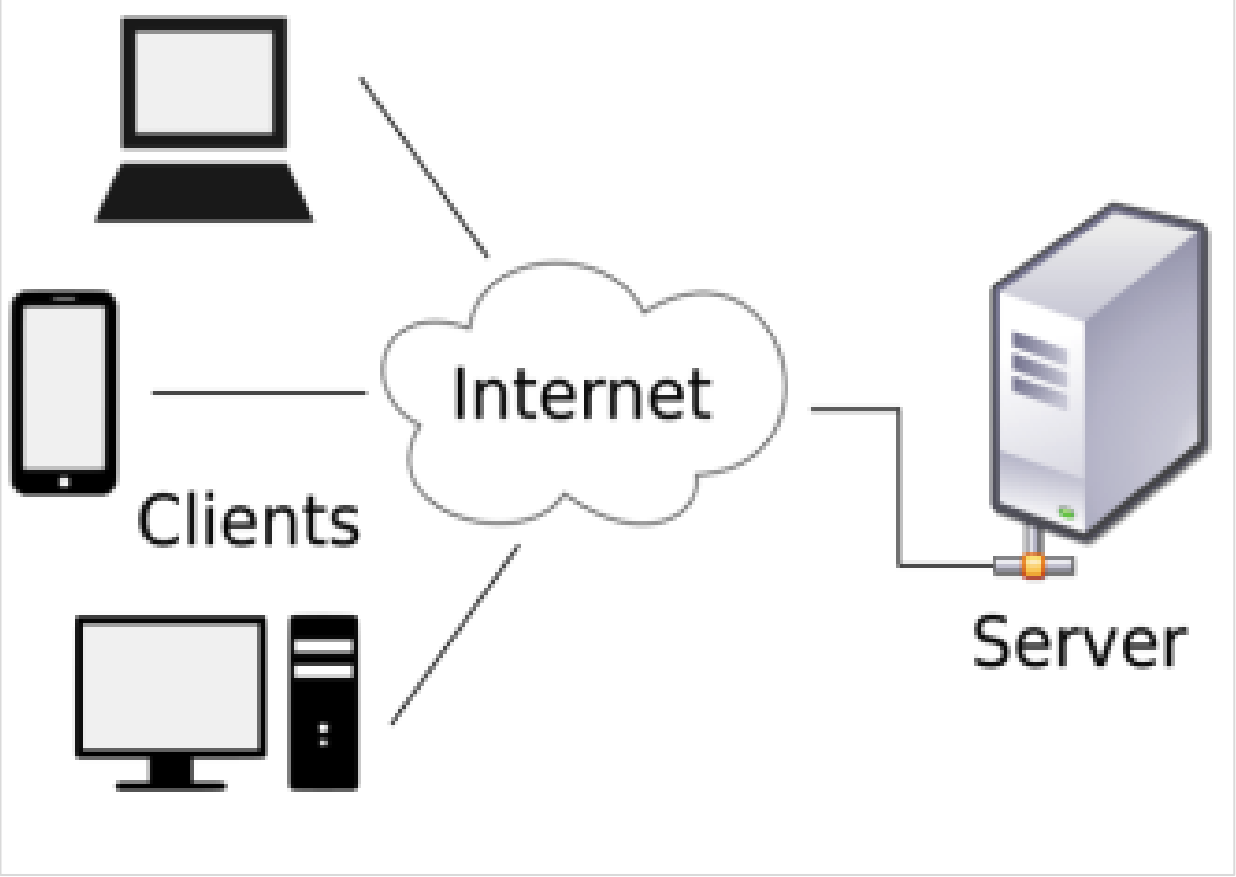


Рисунок 1.4 - Клієнт-серверна взаємодія

Якщо ви уважно подивитеся на модель клієнт-сервер, ви можете розбити цю модель та прийняти той факт, що є два одночасних процеси, що виконують події - один на машині клієнта, а інший на машині сервера.

Клієнтська частина, або просто клієнт, — це «програма, яка з'являється», тобто ті, хто бачить користувача, а серверна частина виконує головний комп'ютер — і мозок, який зберігає, обчислює та візуалізує інформацію про Дані. Для багатьох сервер може створити велику кількість клісів (сотні тисяч) за ніч. Комунікація - це процес, за допомогою якого клієнт отримує інформацію. Клієнтський процес перевіряє згенеровані дані. Якщо процес Servernij закриває елемент керування, виберіть цю функцію або перевірте дані та знову перевірте результат. Ця архітектура забезпечує спільний доступ до зони обслуговування менші та більш спеціалізовані системи. Немає необхідності переписувати серверну частину на основі типу клієнта [10]. Однак, якщо ви змініте внутрішню логіку сервера, усі CLI все одно будуть використовуватися. Підпишіться, коли ми завершимо встановлення API. Таким чином, клієнт-серверна архітектура Vicorist використовується роботом для передачі веб-плагіна Vicorist до ядра клієнта браузера відповідно до протоколу [9].

Програми HTTP-потоків надсилають запити (HTTP-запити) на сервер і там обробляють їх .

1.3.1 Основні компоненти веб-застосунків

Для доступних функцій розрізняють деякі такі типи веб-клієнтів :

a) використовувати протокол для передачі даних за протоколом HTTP, WAP;

b) за типом програми, що використовується – веб-браузер або будь-яке інше.

c) клієнтська програма;

d) за типом відображення даних;

e) щляхом обробки логіки програми.

Важливо дотримуватися способу подання даних.

Дозволено розрізняти доставку на стороні клієнта та доставку на стороні сервера, при цьому ситуація даних відома як «товстий клієнт — тонкий сервер» і «тонкий клієнт — товстий сервер», відповідно.

Архітектура клієнт-сервер не є строго типізованою архітектурою, але клієнт і роботодавець можуть відрізнятися від запропонованої концепції, незалежно від типу плагінів розробки, технологічної платформи, мов програмування та фреймворків, які використовувалися для розробки.

Тоді ви запитуєте про розподіл роботи між постачальниками веб-додатків. Підхід «товстий клієнт — тонкий сервер» має на меті розділити представлення даних і логічну обробку цього плагіна та їх реалізацію переважно на стороні клієнта. Інструменти програмування сучасних мов програмування, тобто велика кількість JavaScript і його свого роду надійна структура, можуть перенести виконання важливого проекту на зовнішні частини бізнес-логіки комп'ютера безпосередньо в диспетчері. Коли ви відображаєте JavaScript на стороні клієнта, запущений у веб-браузері користувача, він відповідає за отримання даних із сервера та взаємодію із сайтом.

Наприклад, якщо корисктувач вводить недійсне значення у форму, код клієнта може просто оновити сторінку з повідомленням про помилку, не надсилаючи новий запит на сервер, тому він працює без перезавантаження сторінки. Тонкий клієнт - товста серверна архітектура. Створено класичний вхід до веб-сервісу. Наявність «товстого сервера» свідчить про реалізацію на ньому додаткової веб-логіки, обробки даних та інших безпосередніх функцій розташування сервера.

При рендерингу на стороні сервера клієнт хоче зробити прямий запит серверу для рендерингу або взаємодії з корисктувачем із веб-сторінки частин.

1.4 REST API як спосіб спілкування компонент веб-додатків

REST API – це скорочення від Representational State Transfer Application Programming Interface. Насправді неважливо, пам'ятаєте ви цю довгу абревіатуру чи ні. Важливо розуміти, що це спосіб для різних компонентів веб-програми спілкуватися один з одним. Якщо ви коли-небудь взаємодіяли з веб-сайтом, наприклад надсилали запит на сервер для отримання даних або надсилали дані на сервер, тоді ви стикалися з REST API. Він використовується скрізь у веб-розробці, і знання, як це зробити, є ключовим навиком для розробників [17].

Щоб краще зрозуміти, як працює інша частина API, уявімо, що у нас є веб-додаток. Ця програма складається з двох основних частин: інтерфейсу (клієнтська частина) і бекенда (серверна частина) REST API як міст між цими двома частинами. Інтерфейс може надсилати запити до серверу та отримувати дані з нього.

Рисунок 1.5 – Проблеми котрі може вирішити REST

1) **запити від клієнта до сервера:** коли користувач додає якийс товар припутімо до кошика на вебсторінці UI то це (frontend), JavaScript на сторінці створює HTTP-запит і відправляє його на сервер (backend) частину . Запит може містити інформацію про товар, його кількість та інші деталі котрі були продумані логікою розробника та наповнені тех райтером.

2) **обробка запитів на сервері:** сервер, отримавши запит, використовує REST API архітектура для інтерпретації його вмісту та виконання відповідної операції. Наприклад, він може перевірити доступність товару, зменшити кількість товару на складі та створити замовлення.

3) **відповідь від сервера до клієнта:** після опрацювання запиту сервер створює HTTP-відповідь і надсилає її назад на клієнтську сторону UI частину. Відповідь може містити інформацію про статус операції, оновлений стан даних або інші відомості котрі потрібно передати .

4) **оновлення інтерфейсу на клієнті:** на клієнтському боці JavaScript код обробляє HTTP-відповідь від серверуі, за необхідності, оновлюєUI. Наприклад, він може оновити кошик користувача котрий на сайті, типу показати повідомлення про успішну покупку тощо.

Такий обмін даними між передньою та серверною частиною через REST API дозволяє програмі працювати ефективно та результативно. REST API визначає правила для створення, надсилання та обробки запитів і відповідей, а процес взаємодії є стандартизованим і надійним [21].

REST API побудовано на основних принципах, які роблять його ефективним і гнучким інструментом обміну даними.

1) **stateless:** (без стану) чому всі HTTP-запити не мають стану?

2) REST API працює в асинхронному режимі, що означає, що кожен запит HTTP, надісланий на сервер, містить дані, необхідні для його обробки. Сервер не зберігає інформацію про попередні запити клієнта. Це підвищує надійність і масштабованість системи.

3) **Client-Server:** незалежність клієнта та сервера та їх взаємодія

4) REST API повністю розділяє клієнтську та серверну частини програми. Це означає, що вони можуть розвиватися незалежно один від одного. Клієнтом може бути веб-браузер, мобільний додаток або інший клієнтський додаток, а сервер може бути написаний будь-якою мовою програмування [14].

5) **Uniform Interface:** чотири відповідні інтерфейси REST API спирається на чотири основні зв’язки: ресурси, методи HTTP (GET, POST, PUT, DELETE),

представлення ресурсів (як представити дані клієнту) і зв’язки між матеріалами. Це робить взаємодію між клієнтом і сервером більш узгодженою.

6) **Cacheable:** як кешування покращує продуктивність програми

7) REST API підтримує кешування, що означає, що клієнт може тимчасово зберігати дані, щоб зменшити навантаження на сервер і підвищити продуктивність.

8) **Layered System:** переваги багатошарової системи архітектури

9) Сервери REST API можна налаштовувати на рівні, що робить їх більш гнучкими та масштабованими. Кожен рівень виконує певні функції, які можна додавати або видаляти, не змінюючи код клієнта.

10) **Code on Demand:** є обмеження, що ви можете завантажити код клієнта. Ця політика є необов’язковою та дозволяє серверу надсилати активований код клієнту. Він використовується рідко і вимагає додаткових функцій безпеки. В цілому інші принципи API забезпечують ефективність і простоту взаємодії між клієнтом і сервером, включаючи стандартизацію, надійність і стабільність системи.

Основними HTTP-методами, які використовуються в REST API, є:

a) GET: Цей метод використовується для отримання даних із сервера. Коли клієнт надсилає запит GET, сервер повертає запитані дані без змін. Наприклад, можна зробити запит GET, щоб отримати інформацію про продукт на веб-сторінці магазину.

b) POST: ці команди використовуються для надсилання даних на сервер. Цей метод дозволяє клієнту створювати нові ресурси на сервері або надсилати дані для обробки. Наприклад, запит POST можна використовувати для створення нового запису даних.

c) PUT: призначений для оновлення існуючих даних на сервері. Клієнт надсилає дані для заміни існуючих даних на сервері. Наприклад, команду PUT можна використовувати для оновлення інформації про користувача.

d) DELETE: використовується для видалення ресурсів на сервері. Цей метод дозволяє клієнту видаляти дані або ресурси, які більше не потрібні.

Наприклад, команду DELETE можна використовувати для видалення облікового запису користувача.

РОЗДІЛ 2. АНАЛІЗ ІСНУЮЧИХ РІШЕНЬ

2.1 Огляд популярних веб серверів котрі розповсюджені у світі

Огляд популярних веб-серверів нам дає змогу розібратися в різних варіантах, які можна використовувати для обслуговування веб-запитів та розглянути їхні основні характеристики. Ось кілька популярних веб-серверів я наведу як приклад побачите нижче нижче:

1) **Apache HTTP Server (Apache):**

a) характеристики: Відомий своєю надійністю та стабільністю за роки , підтримує багато різних модулів в собі та інтеграції і різні можливості.

b) використання: Широко використовується на серверах з великим обсягом контенту (такий тип серверів використовується в компаніях котрі мають нахил на софт та залізо і заробляють на цьому).

2) **Nginx:**

a) характеристики: Орієнтований на високий рівень здатності обробки паралельних з'єднань, відмінно підходить для обслуговування великого числа запитів (на одній із робіт в нас сервера Nginx).

b) використання: Зокрема ефективний для використання в якості веб-проксі та балансувальника навантаження.

3) **Microsoft Internet Information Services (IIS):**

a) характеристики: Інтегрований в середовище Windows Server, підтримує ASP.NET та інші технології Microsoft (дуже рідко зустрічав даний тип веб серверу).

b) використання: Зазвичай використовується в середовищах, де використовується інфраструктура Microsoft.

4) **LiteSpeed Web Server:**

a) характеристики: Дуже відомий веб сервер своєю високою продуктивністю та швидкодією, підтримує кешування та балансування навантаження (що є дуже цікаво для адміна в якості налаштування).

b) використання: Використовується для прискорення роботи веб-сайтів.

5) Caddy:

a) характеристики: Спрощений веб сервер та легкий у використанні веб-сервер з плюшок він автоматично надає SSL-сертифікати.

b) використання: Ідеально підходить для невеликих проектів та тих, хто шукає простоту конфігурації (підходить для навчання).

6) Tomcat:

a) характеристики: Розроблений для використання з Java-додатками, надає середовище для виконання Java Servlets та JSP (більший нахи веб сервер має під Java).

b) використання: Широко використовується для розгортання веб-додатків на основі Java.

7) HAProxy:

a) характеристики: Використовується як балансувальник навантаження для розподілу трафіку між серверами.

b) використання: Ефективний для забезпечення високої доступності та балансування навантаження.

Вибір конкретного веб-сервера залежить від потреб вашого проекту, його розміру, технічних можливостей і ваших повноважень. Кожен має свої переваги та характеристики, які роблять його популярним у різних ситуаціях [13].

2.1.1 NGINX – веб сервер як машина для роботи

Nginx (Engine x) — це HTTP-сервер і проксі IMAP/POP3 для UNIX-подібних платформ (FreeBSD і GNU/Linux).

Для балансування навантаження захист від DDoS діє як резервний агент. Зворотний проксі зазвичай визначається як процес, у якому сервер, який отримує запит від клієнта, не обробляє його повністю, а частково або повністю надсилає запит іншим (вищим) серверам для роботи. Тобто він не перенаправляє клієнта, а автоматично надсилає запит і повертає отриману відповідь клієнту [12,11].

Завдяки низькому використанню системних ресурсів і швидкості роботи, а також гнучкості веб-сервер Nginx часто використовується як перше місце на важких серверах, таких як Apache, для більш вимогливих проектів. Класичним варіантом є підключення Nginx - Apache - FastCGI. Працюючи з такою метою, сервер Nginx приймає всі запити, отримані через HTTP, і вирішує, залежно від конфігурації та самого запиту, чи обробити запит самостійно та надати відповідну відповідь клієнту, чи надіслати запит на бекенд (Apache). або FastCGI) для обробки [12].

2.1.2 Apache – веб сервер як універсал

Apache - це веб-сервер NTTR. Веб-сервер Apache — це веб-програмне забезпечення. Завдяки обширній документації та хорошій інтеграції зі стороннім програмним забезпеченням Apache став дуже популярним. Він підтримує наступні ОС - Linux, BSD, Mac OS, Microsoft Windows, Novell NetWare, BeOS та ще багато іншого. Apache веб сервер потрібен для:

a) відкривання динамічних PHP-сторінок;

b) роподіляння наватаження що знаходится на сервері;

c) забезпечування відмовостійкість сервера (для зберігання своєї працездатності після відмови однієї чи кількох складових частин системи);

d) вправляння у запуску PHP-скриптів та налаштуванні сервера;

Apache має базу, повністю написану мовою C, і модулі, вибір яких визначається конкретною інформацією, що зберігається на сервері. Модулі ширять

фічі Apache Core. Зміни параметрів ядра можливі шляхом зміни конфігураційних файлів:

a) httpd.conf – файл відповідає за налаштування параметрів рівня сервера;

b) extra/httpd-vhosts.conf – файл відповідає за налаштування параметрів віртуального хоста;

c) .htaccess – цей стек відповідає за налаштування параметрів рівня каталогу. Ці файли забезпечують децентралізацію конфігурації веб-сервера. Завдяки визначенню на льоту можна вносити зміни до налаштувань сервера без перезавантаження. Файли .htaccess дозволяють керувати окремими функціями веб-ресурсів певним групам користувачів, які не мають прав адміністратора. Зміна налаштувань модуля можлива лише такими способами:

d) шляхом зміни конфігураційних файлів модуля;

e) зміна параметрів конфігураційних файлів операційної системи;

f) для введення параметрів за допомогою командного рядка.

Для обробки запитів користувачів використовуються такі модулі:

1) mpm_prefork - створює однопотоковий процес для кожного запиту.

2) mpm_worker - породжує багатопотокові процеси, кожен з яких може обробляти лише одне з'єднання. Характеризується високою швидкістю.

3) mpm_event - аналог попереднього модуля, адаптований для роботи з налаштуванням підключення.

4) вибір модулів визначається особливостями самого веб-проекту та аудиторії користувачів.Функціональність веб-сервера Apache розширена модулями, яких понад 500 (деякі розроблені розробниками Apache, інші — сторонніми розробниками). Збірка може бути створена під час конфігурації програмного забезпечення (Apache Web Server — ядро + модулі), але використовує потужну систему підключення модулів. Розробка модулів здійснюється для таких цілей:

a) додавання підтримки для певних мов програмування;

b) для розширення наявного функціоналу;

c) для додавння різних імпрувів до основних функцій для фіксингу багів;

d) для поліпшення системи безпеки продукте.

e) модулі CGI і FastCGI використовуються для взаємодії з різним програмним забезпеченням. Для найпопулярніших мов програмування (наприклад, apache_asp, mod_php та інших) створені окремі модулі.

2.2 Огляд інструментів для аналізу запитів по HTTP

До інструментів аналазу трафіку я хочу віднести декілька котрі на ринку є найпопулярніші.

Вони використуватся у більшості сучасних компаній і у звичайному житті людей котрі хвилюються за свій трафік.

Ці інструменти використовуються у ряді компаній котрі займаются мережами та розробкою для фільтрування свого трафку та його проксіювання.

Аналізатори трафіку – це інструменти, призначені для моніторингу, аналізу та управління потоками даних у комп'ютерних мережах. Вони дозволяють виявляти реєструвати та аналізувати всі дані, що проходять через мережу, блокуючи веб-трафік, пакети даних, HTTP-запити та лінії. Ці інструменти дають розробникам програмного забезпечення можливість збирати статистику, виявляти проблеми зі з'єднаннями, визначати потоки трафіку та надавати важливі дані для ретельної безпеки та управління даними.

2.2.1 Fidler

Fiddler - Проксі-сервер для налаштування , котриі використовується для реєстрації, перевірки та зміни трафіку HTTP та HTTPS між комп’ютером і веб-сервером. Fiddler спочатку був написаний Еріком Лоуренсом, коли він працював менеджером проекту в групі розробників Internet Explorer у Microsoft.[1]

Використання назви «Fiddler» було розширено, щоб включити інші продукти та інструменти, надані Progress Telerikruen, зокрема Fiddler Classic, Fiddler Everywhere, Fiddler Core, Fiddler Cap і Fiddler Jam [19].

Дуже часто під подібну мету використовується інструмент Fiddler - классік проксі, котрий здатний перехоплювати HTTP (S) трафік і забезпечувати роботу з ним [20].

За його допомогою можна записувати , проставляти контрольні точки, оперувати вхідною та вихідною інформацією через нього. Найчастіше тестувальники та розробники використовують його як свого роду проміжну ланку між кінцевим користувачем та цільовим сервером. Запустивши цей інструмент, ми можем побачити, що саме відбувається на веб-сторінці після переходу на неї

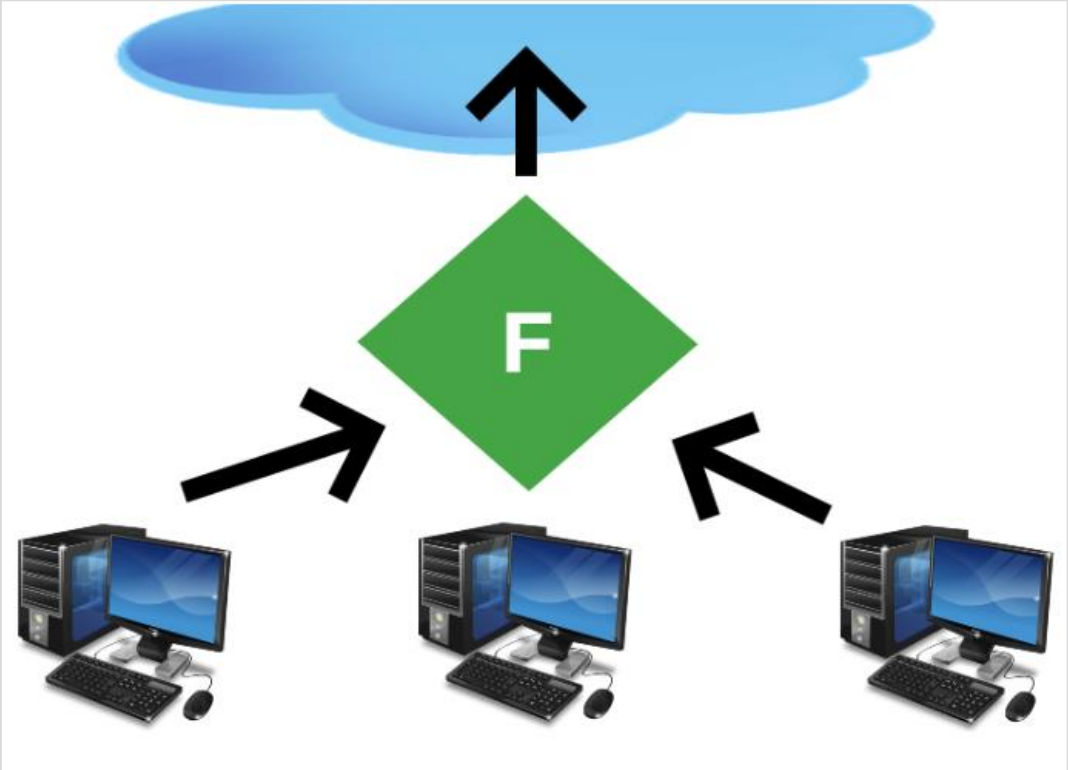


Рисунок 2.1 – Абревіатурний вигляд інструменту

2.2.2 Wireshark

Wireshark – це безкоштовна програма аналізу мережевого трафіку, що дозволяє спостерігати, захоплювати та аналізувати пакети даних, переміщуючи їх у вашій мережі. Одним з основних додатків Wireshark є аналіз мережевих протоколів, що дозволяє виявляти та виправляти помилки у додатках чи протоколах.

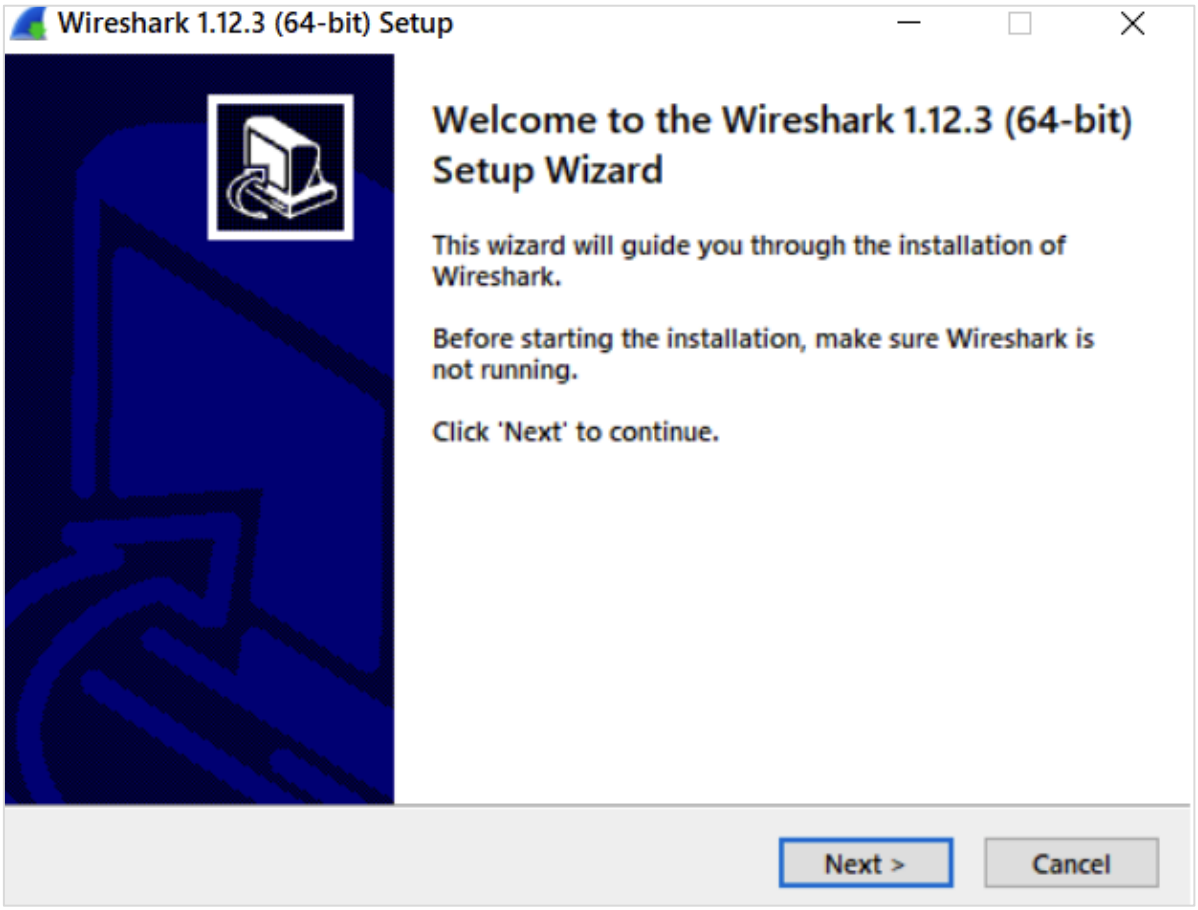


Рисунок 2.2 – Вікно встановлення програми

Мережевий інтерфейс – це програмне забезпечення, яке взаємодіє з мережним драйвером та з рівнем IP. Він забезпечує рівню IP доступ до всіх наявних мережних адаптерів, трафік яких ми будемо перехоплювати. Найчастіше у програмі Wireshark можна зустріти мережний інтерфейс бездротової (Wi-Fi) та кабельний (Ethernet).

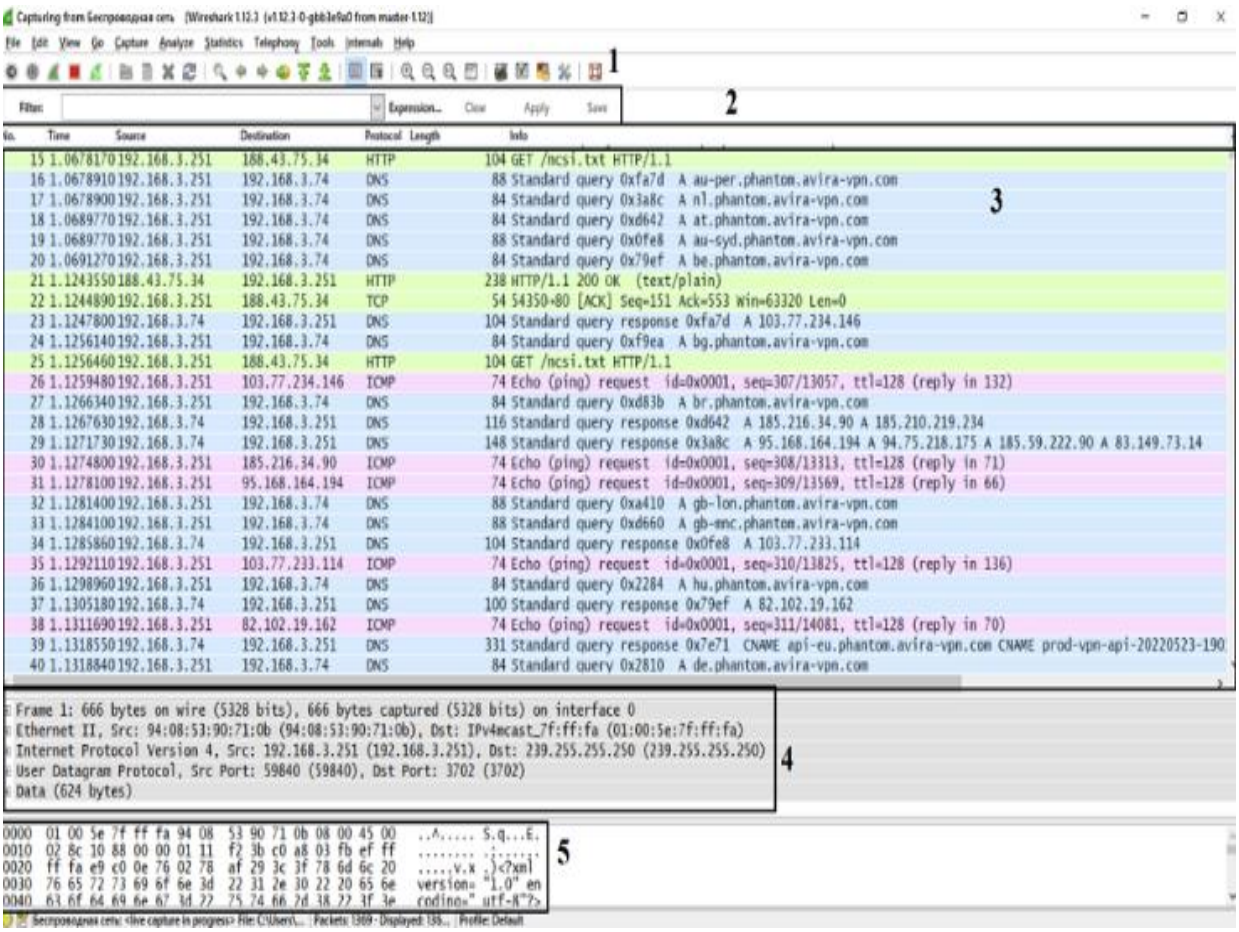


Рисунок 2.3 – Вікно аналізу трафіку

2.2.3 Devtools

Сучасні браузери Safari, Firefox, Microsoft Edge, Chrome та інші мають вбудовані засоби розробника, які дозволяють переглядати вихідний код сайту. Віддержно установка їх не регулювати. З їх допомогою ви можете переглядати та налагоджувати HTML, CSS і Javascript сайту. Також ви можете перевірити мережевий трафік, споживаний сайтом, його швидкість і багато інших параметрів.

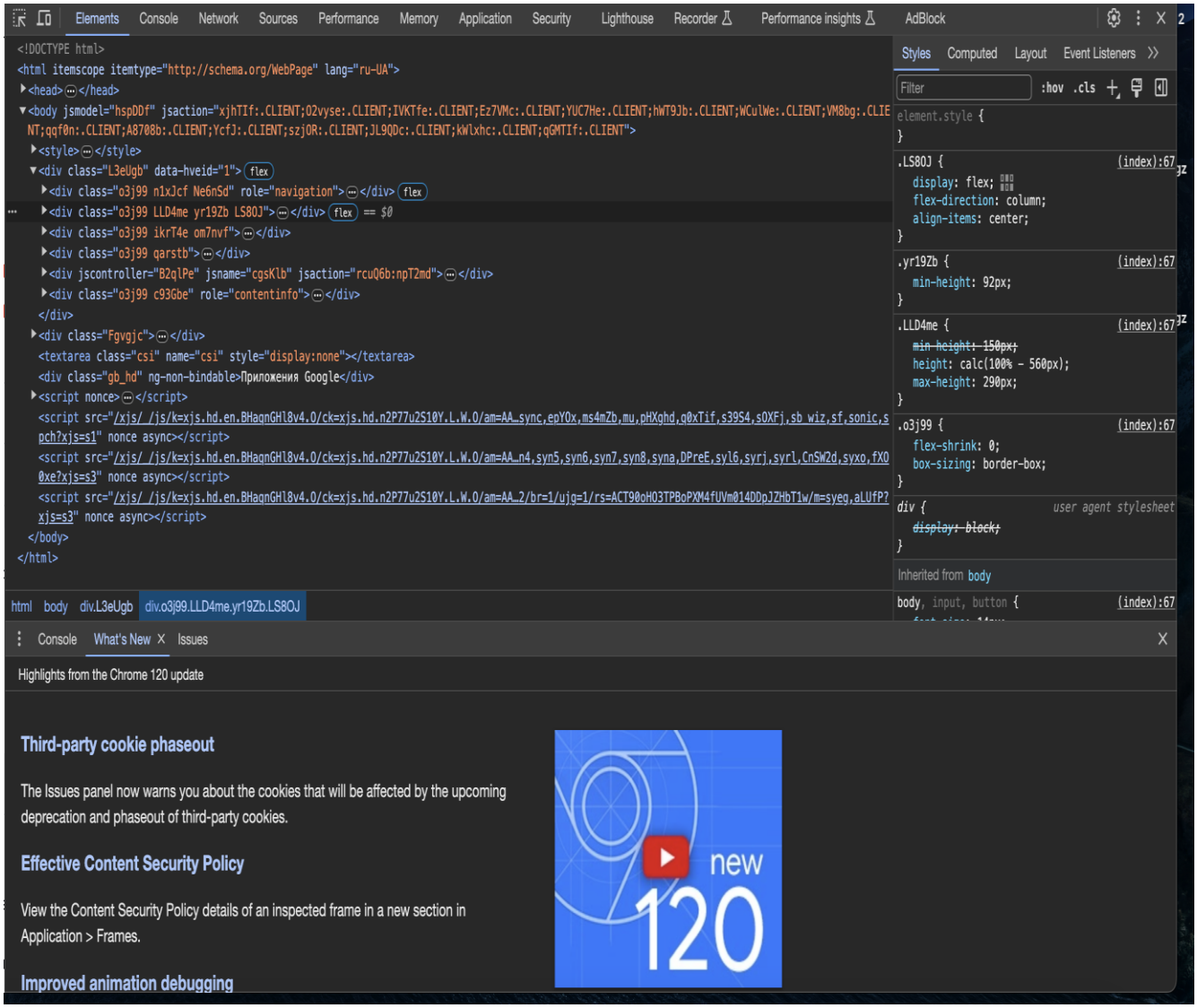


Рисунок 2.4 – Головний Інтерфейс девтулза хром

Інструменти розробника Chrome пропонує великій набір способів перевірки та оптимізації коду, а також боротьби з помилками в коді.

Веб-розробники зазвичай зберігають повідомлення на консолі або деяіки типи розробників пишуть код в блокноті з хардкодом щоб переконатися, що JavaScript працює належним чином. Для зберегання повідомлення треба додати фразу, для прикладу я візьму console.журнал ('Consol, hello.'). Якщо браузер запускає JavaScript та бачe цей вираз, то я можу побачити, що повідомлення має бути записано на консоль. Наприклад, для виводу HTML та JavaScript на сторінці:

```
<!doctype html>
<html>
  <head>
    <title>Console Demo</title>
  </head>
  <body>
    <h1>Hello, World!</h1>
    <script>
      console.log('Loading!');
      const h1 = document.querySelector('h1');
      console.log(h1.textContent);
      console.assert(document.querySelector('h2'), 'h2 not found!');
      const artists = [
        {
          first: 'René',
          last: 'Magritte'
        },
        {
          first: 'Chaim',
          last: 'Soutine'
        },
        {
          first: 'Henri',
          last: 'Matisse'
        }
      ];
      console.table(artists);
      setTimeout(() => {
        h1.textContent = 'Hello, Console!';
        console.log(h1.textContent);
      }, 3000);
    </script>
  </body>
</html>
```

Рисунок 2.5 – Консоль

На рисунку 2.5. показано вигляд консолі після завантаження сторінки та очікування протягом 3 секунд.

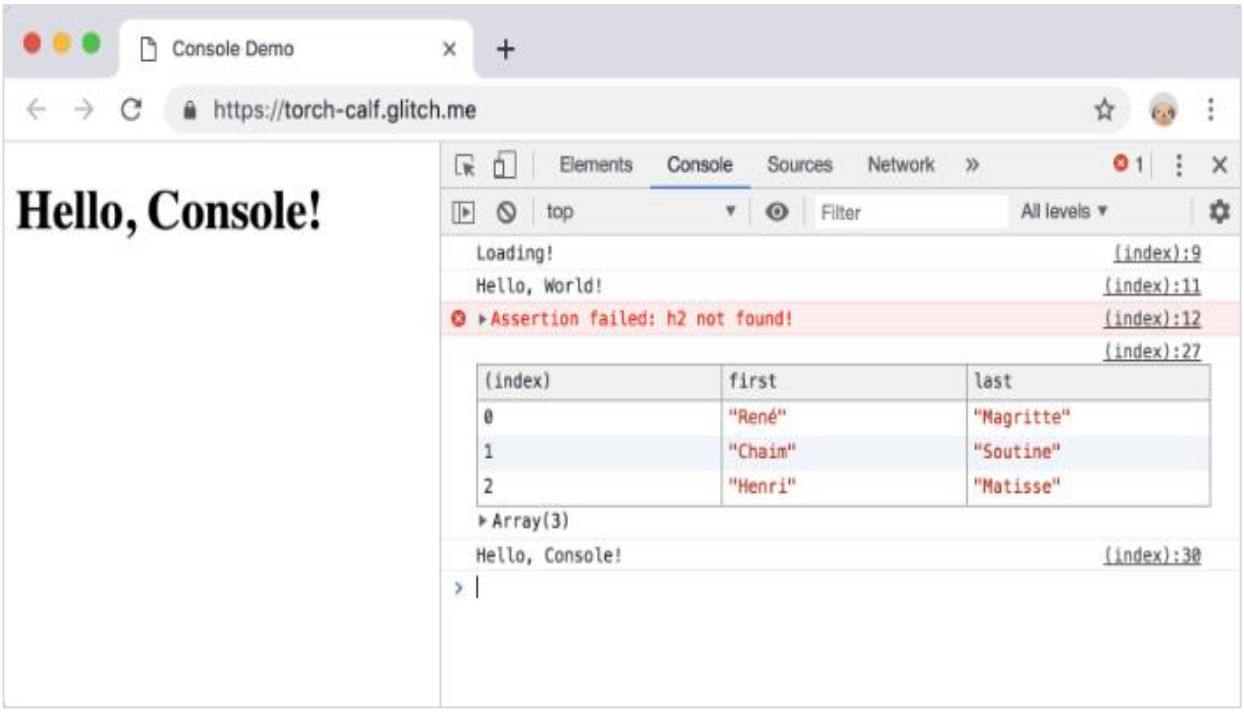


Рисунок 2.6 - Консольна панель

Основними причинами, чому веб-розробники реєструють повідомлення, є:

1) переконання, що код працює належним чином.

2) перевірка значення змінної в певний час.

3) основна відмінність декомунізації між методами полягає в перегляді данних, які реєструються.

Запуск JavaScript

Консоль також є repl. Можна запустити JavaScript у консолі для взаємодії з контрольованою сторінкою. Наприклад, на рисунку 2.5 показана консоль поруч із домашньою сторінкою DevTools, а на рисунку 2.6 показана та сама сторінка після використання консолі для зміни заголовка сторінки.

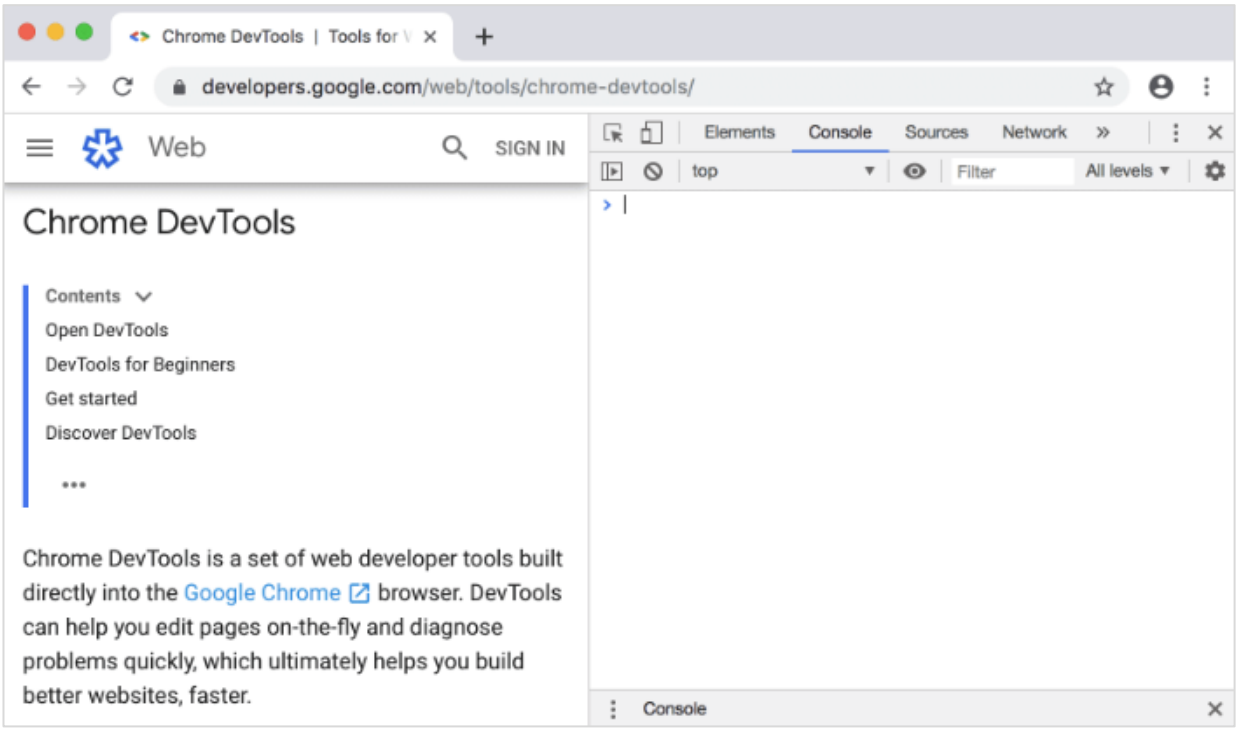


Рисунок 2.7 - Використовується консоль, щоб змінити заголовок сторінки

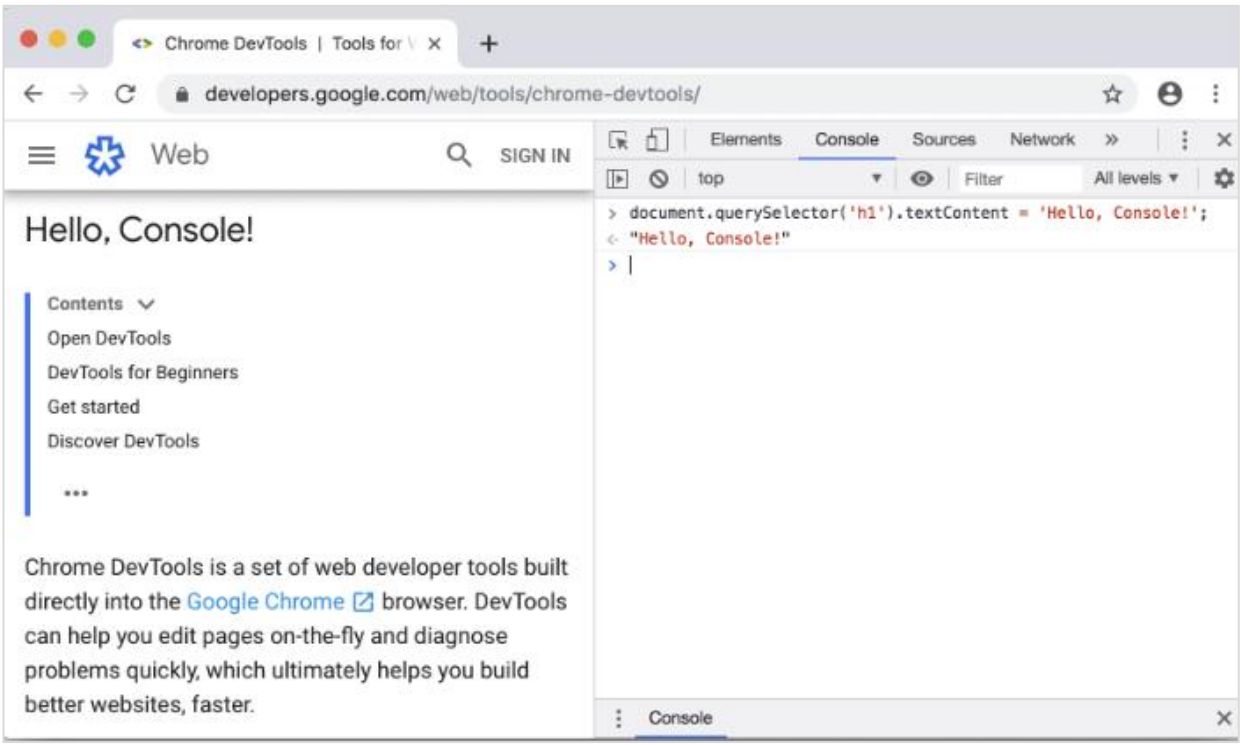


Рисунок 2.8 - Панель консолі поруч із домашньою сторінкою DevTools

Консоль має повний доступ до сторінки вікна, тому можна змінити сторінку з консолі. DevTools має кілька корисних функцій, які полегшують перегляд

сторінки. Наприклад, припустимо, що javascript містить функції з hidemodal. Коли я запускаю Дека (hideModal), код у першому рядку hidemodal зупиняється під час наступного виклику.

Коли запускаю JavaScript, не потрібно взаємодіяти зі сторінкою. Можна використовувати консоль для тестування нового коду, не пов'язаного зі сторінкою. Наприклад, припустимо, щойно вивчено вбудований метод JavaScript Array map() і хочеться спробувати його. Консоль-гарне місце, щоб спробувати цю функцію.

Використовується панель"page", щоб переглянути всі ресурси, завантажені сторінкою.

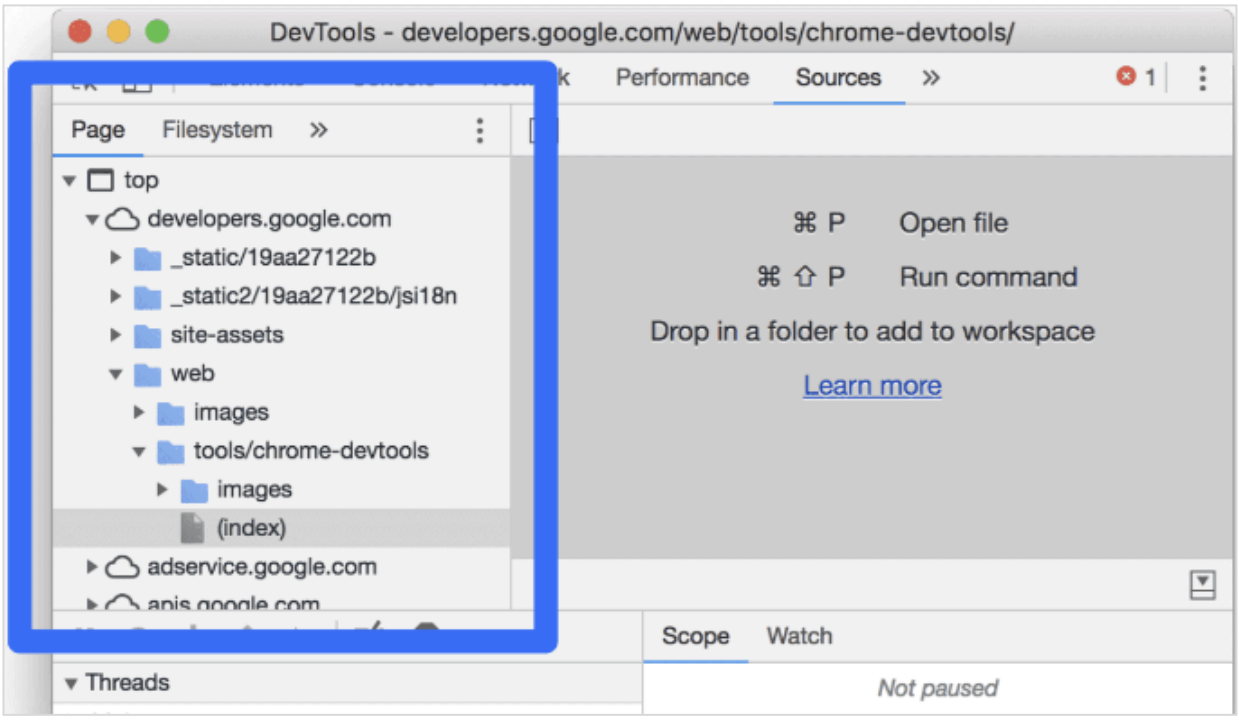


Рисунок 2.9 - Редагування панілі Page

Для редагуванні панелі ‘page’:

1) це HTML - фреймворк верхнього – високого рівня, котрий як top на скріншоті вище. На кожній сторінці, яку відвідую, можна знайти найвищий рейтинг. Верхня частина - це основний фрейм документа.

2) другий рівень, для прикладу, developers.google на скріншоті вище є джерелом.

3) 3-4 рівень, тощо, представляє каталоги та ресурси, завантажені з цього джерела. Наприклад, на скріншотах вище. Коли натискаю на файл на панелі "page", то вміст відображається на панелі "Редактор". Можна переглянути файли будь-якого типу. Для зображень, можна побачити попередній перегляд зображення.

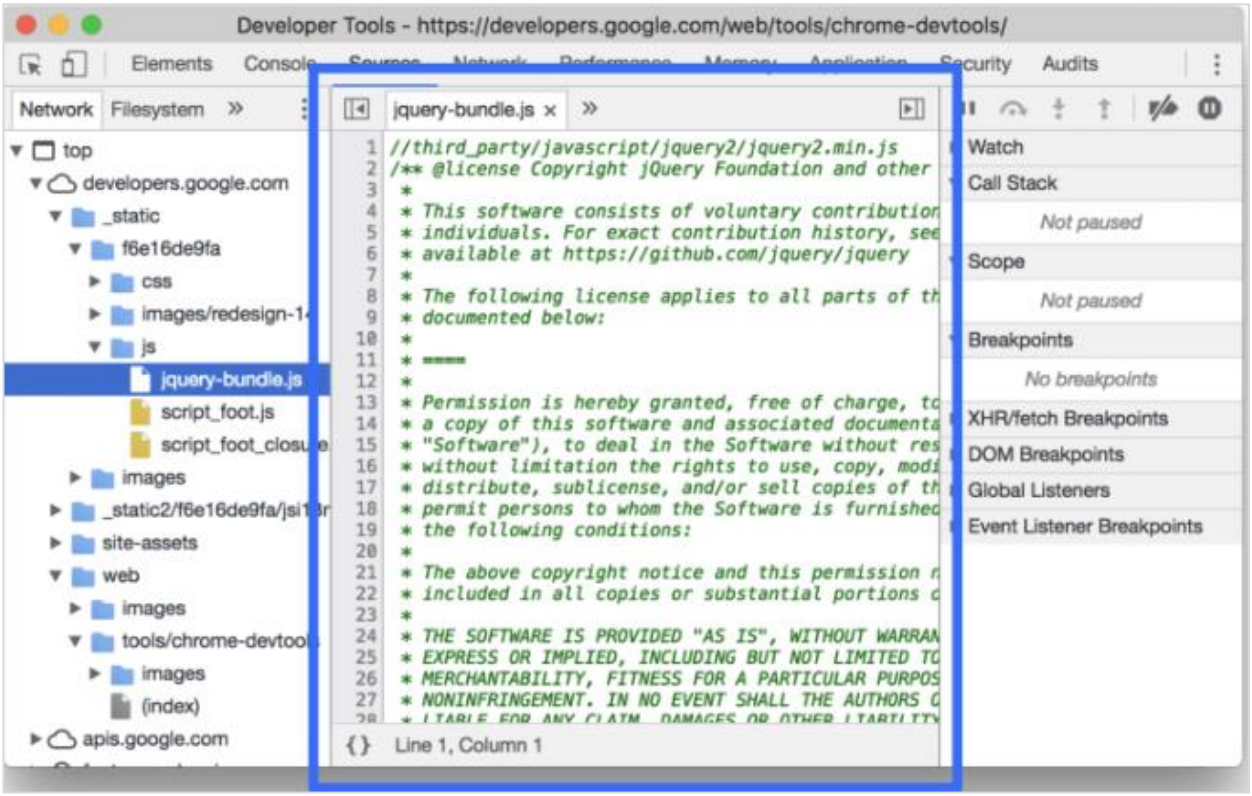


Рисунок 2.10 - Редагування CSS та JavaScript

Використовується панель редактора для редагування CSS та JavaScript. DevTools

Редактор також корисний для налагодження. Наприклад, синтаксичні помилки та інші проблеми, такі як невдалі оператори CSS@import або url(), а також виділення та відображення вбудованих підказок поруч із атрибутами HTML href і з недійсними URL-адресами.

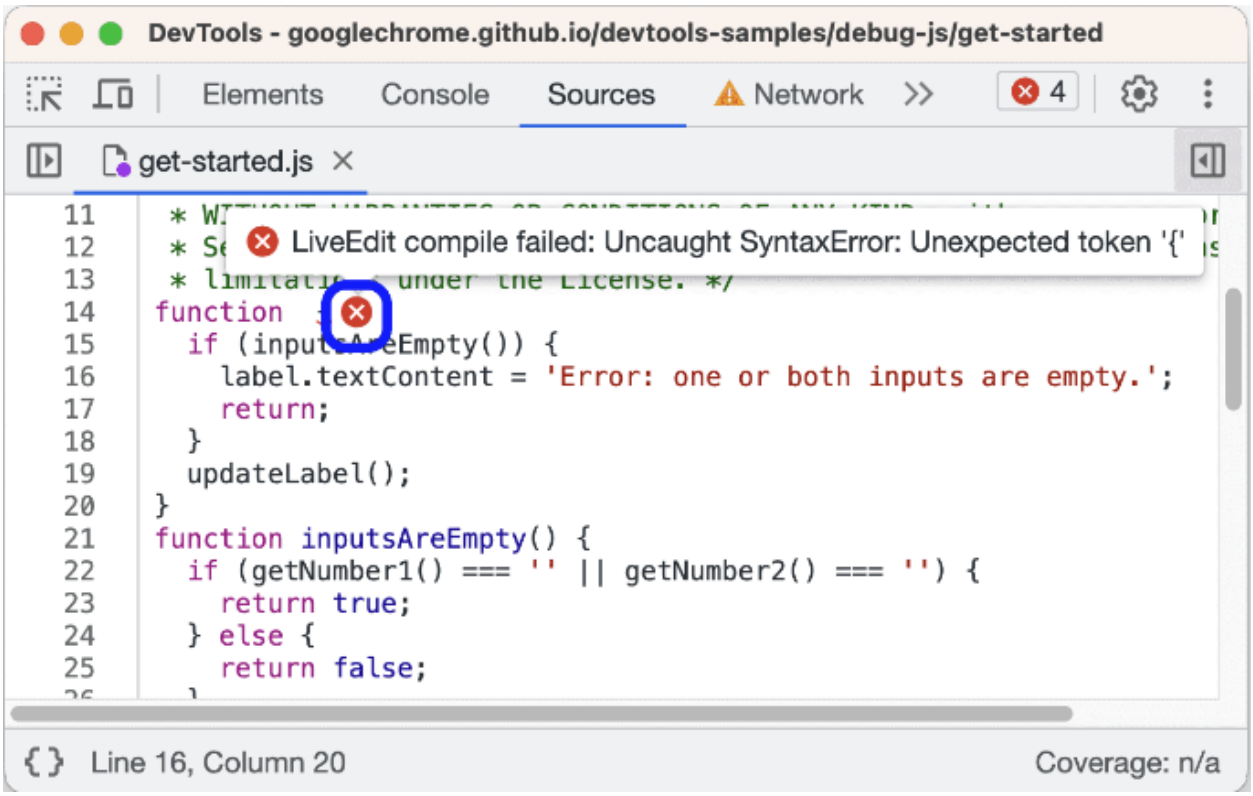


Рисунок 2.11 - Зміни редагуваннння js

Коли редагується елемент колір фону, можна відразу побачити, що зміни вступили в силу.

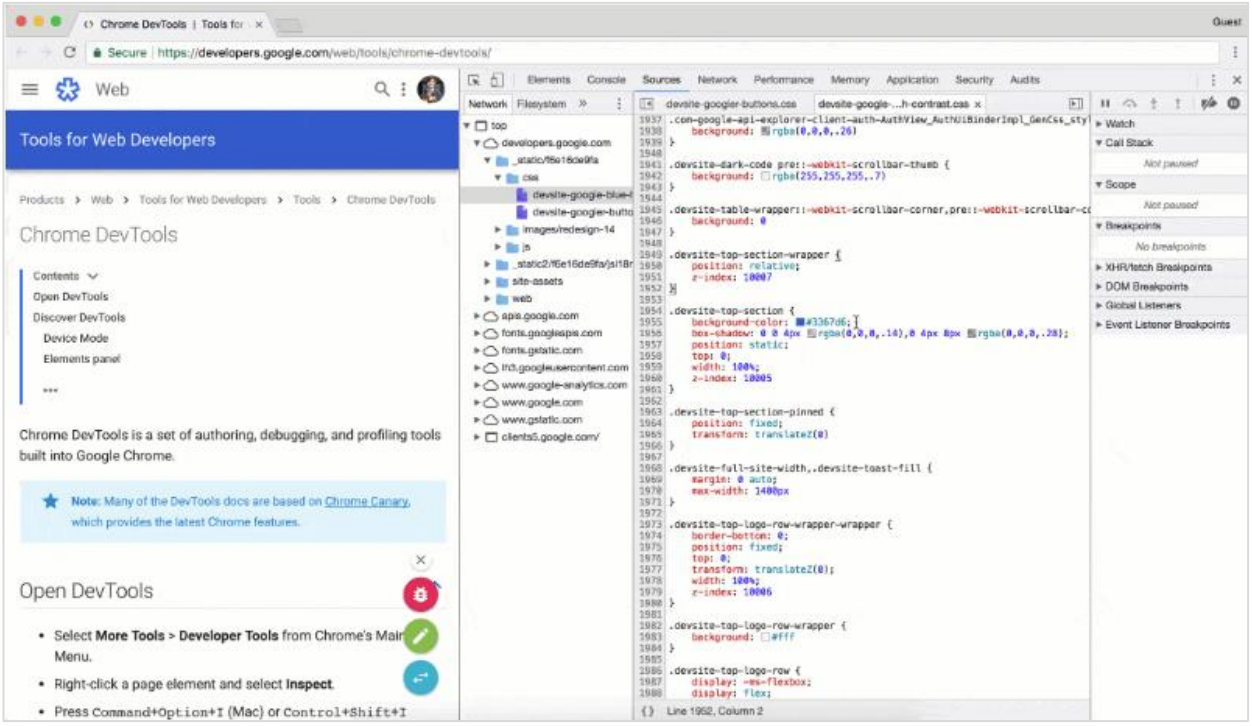


Рисунок 2.12 - Запуск функцій JS

Щоб змінити JavaScript, натискається Command+s(Mac) або Control+s (Windows, Linux). DevTools не перезапускає скрипт, тому вступають в силу тільки зміни JavaScript, внесені в функцію. Наприклад, зверніть увагу на консоль log ('a') не запускається, але консоль log ('B') працює.

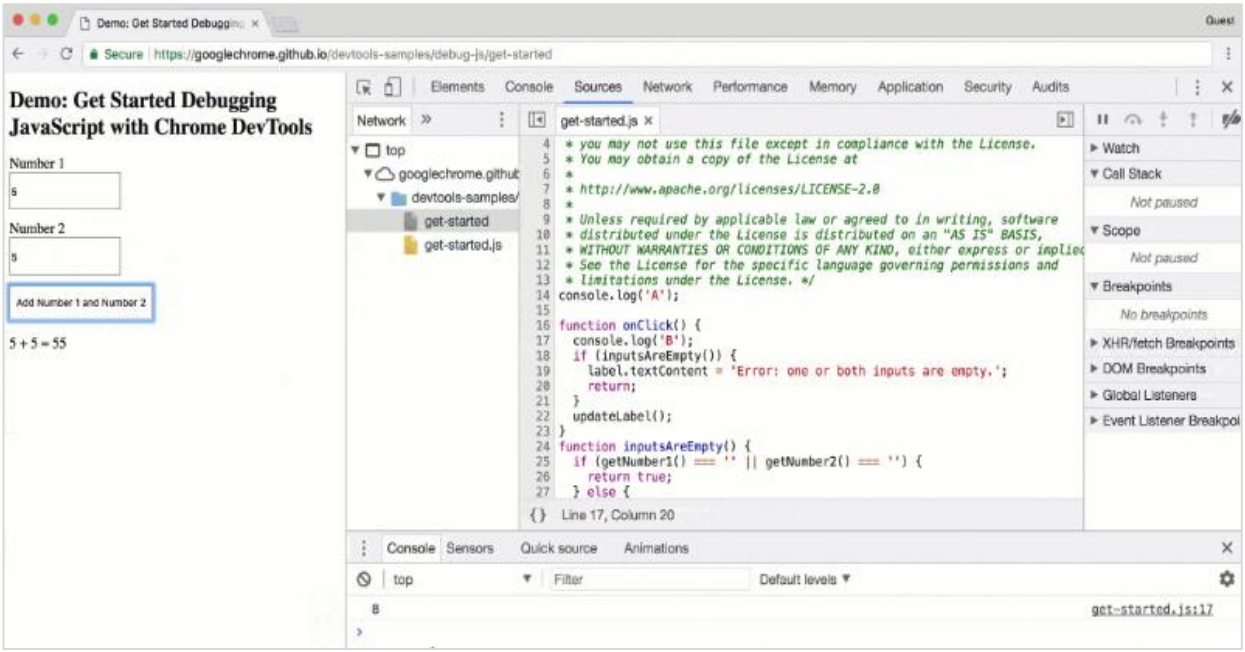


Рисунок 2.13 - Запис функцій на консоль

Якщо DevTools повторно запускає весь сценарій після внесення змін, текст Abu записується на консоль.

DevTools видалить зміни CSS та JavaScript під час перезавантаження сторінки. Фрагмент – це скрипт, який можна запустити на будь-якій сторінці. Припустимо, вводиться код неодноразово у консоль, щоб додати бібліотеку jQuery на сторінку.

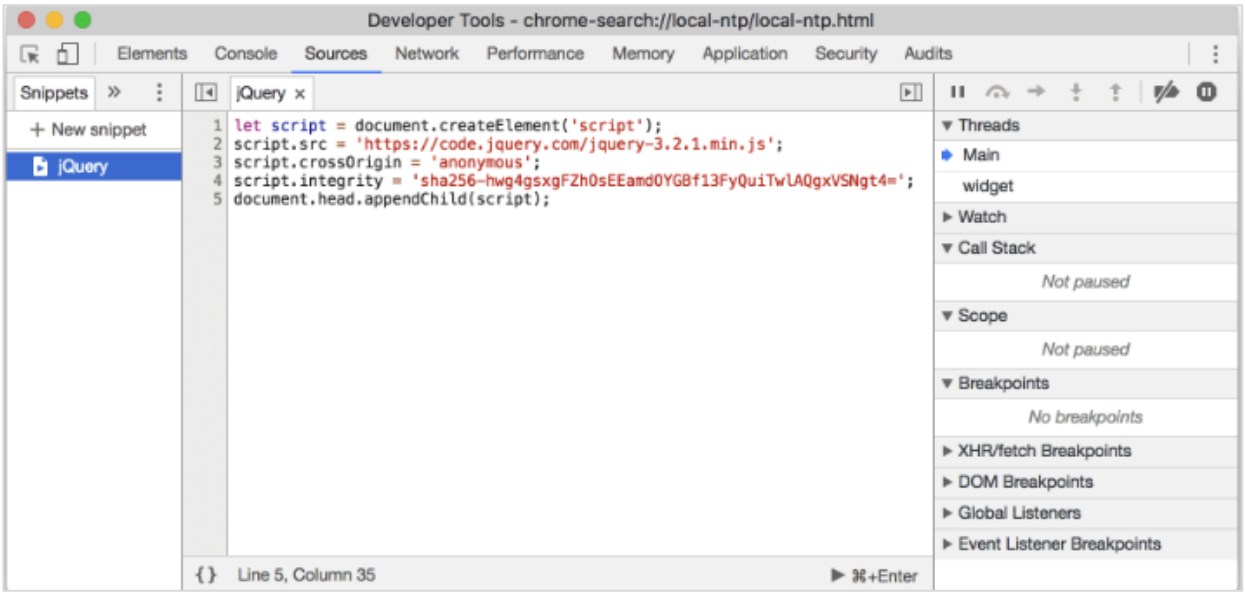


Рисунок 2.14 - Запуск фрагменту

Запускаючи фрагмент потрібно:

Першим ділом відкрити файл на панелі "деталі " та натиснути "Виконати". На нижній панелі дій.

Далі треба відкрити командне меню, видалити символ > і ввести !, ввести назву фрагмента та натиснути клавішу ведення Enter.

Налагодження в JavaScript

Замість використання console. log() позначається, де JavaScript не працює. Загальна ідея полягає в тому, щоб встановити точку зупинки, яка є навмисною точкою зупинки у коді, і покроково виконує 1 рядок коду за раз.

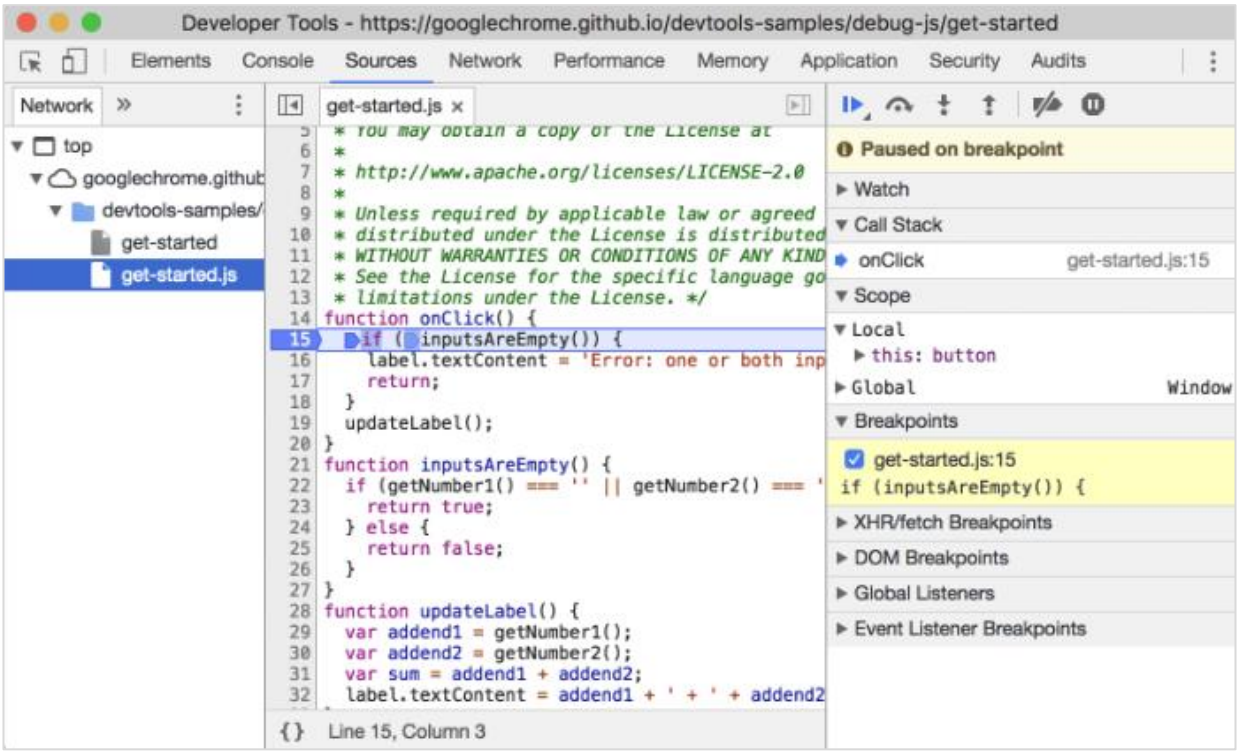


Рисунок 2.15 - Зміни всіх властивостей коду

Після виконання коду я можу переглядати та змінювати значення всіх властивостей та змінних, котрі були визначені на даний момент, або запускати JavaScript у консолі.

Запис мережевої активності

Для перегляду network, викликану сторінкою:

1) перезавантажується сторінка. Мережева панель записує всю мережеву активність у мережевий журнал.

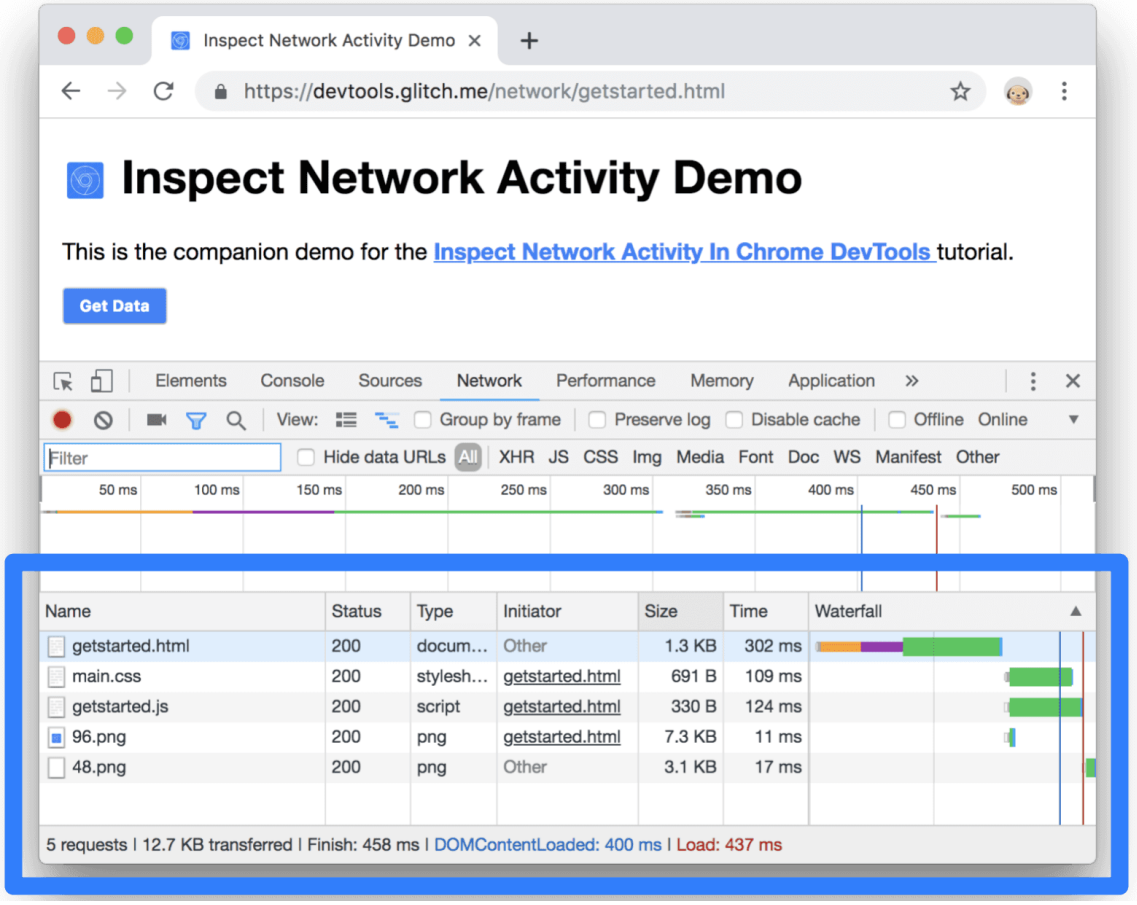


Рисунок 2.16 - Мережеві журнали

Кожен рядок у мережевому журналі представляє ресурс. За замовчуванням джерела перераховані в хронологічному порядку. Верхнє джерело – це основний HTML-документ. Наступне джерело є останнім запитуваним ресурсом.

У кожному стовпці відображається інформація про джерело. На рисунку 2.16 показані стовпці за замовчуванням:

a. Код відповіді HTTP;

b. Література. Тип джерела;

c. Ініціатор. Причина запиту джерела. Натискання посилання в стовпці ініціалізатора приводить до вихідного коду, який викликав запит;

d. Час. Як довго тривав запит;

e. Водоспад. Він графічно представляє різні етапи запиту. Наводиться курсор миші на водоспад, щоб побачити несправність.

2) коли DevTools увімкнено, мережева активність реєструється в мережевому журналі. Щоб проілюструвати це, спочатку потрібно переглянути під мережевий журнал і запам'ятати останню дію.

3) натиснути кнопку "Отримати дані" в демо-версії.

4) в мережевим журналі існує новий ресурс під назвою Getstarted.json... Коли натискається кнопка"Отримати дані", сторінка запитує цей файл.

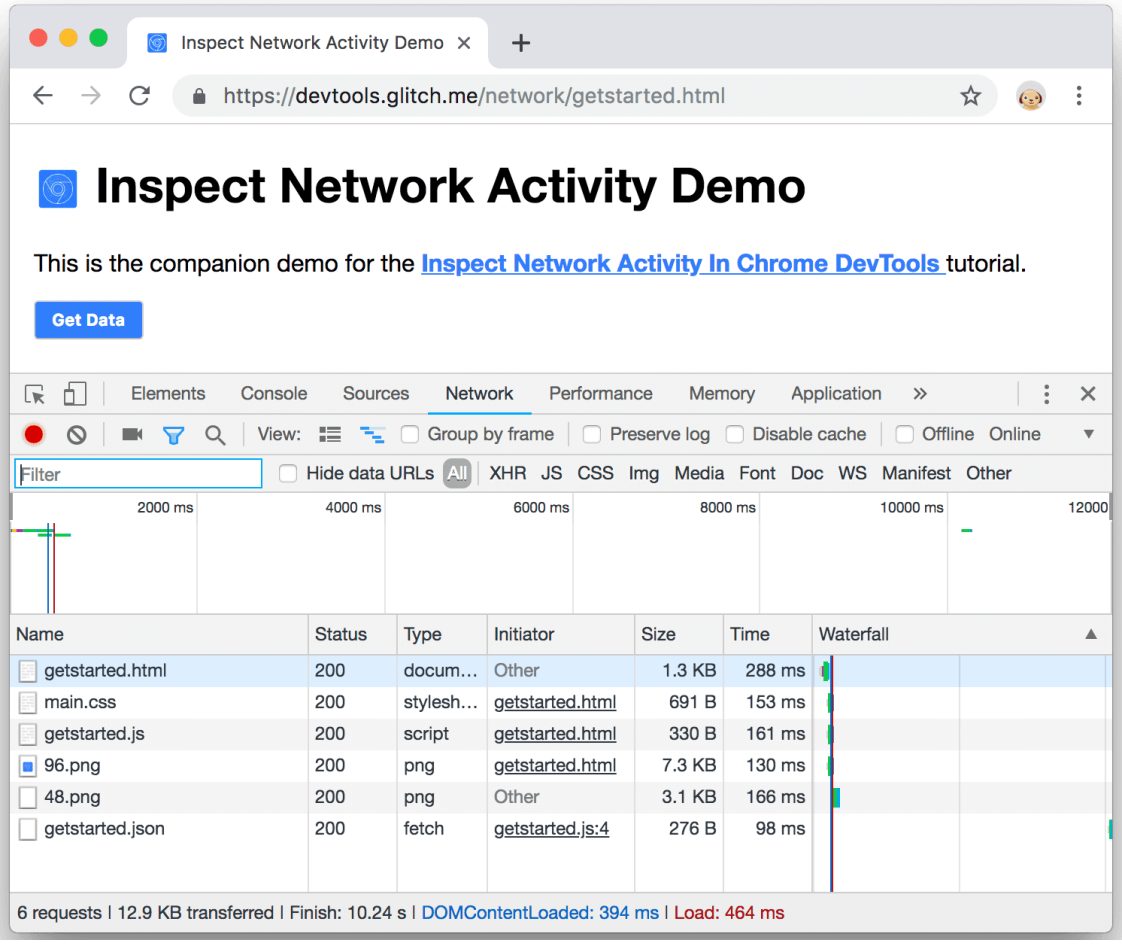


Рисунок 2.17 - Нові ресурси для онлайн-журналів

Можна налаштувати стовпці мережевого журналу. Можна їх приховати, які я не використовую. Є багато стовпців, які за замовчуванням приховані і можуть бути корисними.

1. Клацніть правою кнопкою миші заголовок таблиці мережевого журналу та виберіть домен. Тепер можна побачити домен кожного ресурсу.

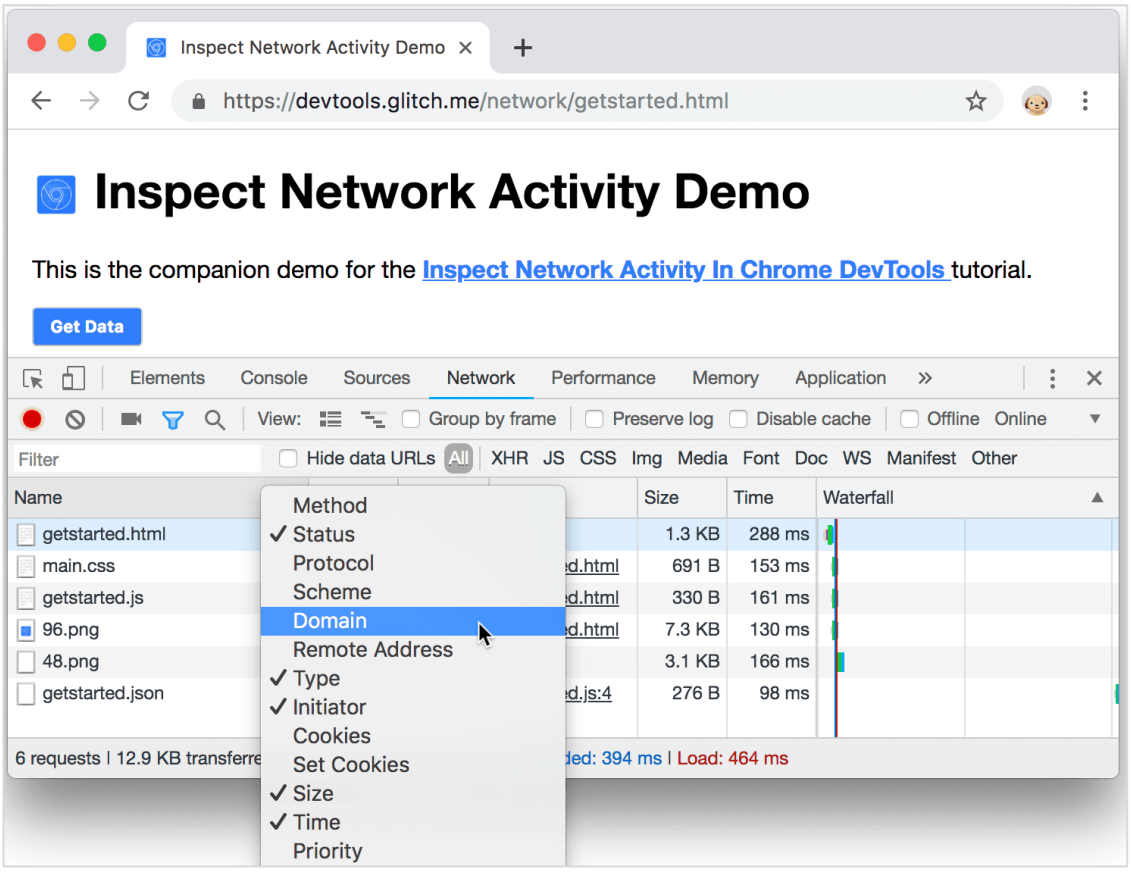


Рисунок 2.18 - Включення стовпців домену

Повну URL - адресу ресурсу можна переглянути, навівши курсор миші на клітинку в стовпці "Ім'я".

Імітація повільного мережевого підключення

Мережеве з'єднання на комп'ютері, який використовується для створення веб-сайту, швидше за все, швидше, ніж мережеве з'єднання на мобільному пристрої ‘Користувача’. Налаштування сторінки дозволяє краще зрозуміти, скільки часу потрібно для завантаження сторінки на мобільному пристрої.

1) натиснути спадне меню налаштувань, у якому за замовчуванням встановлено значення "Користувач";

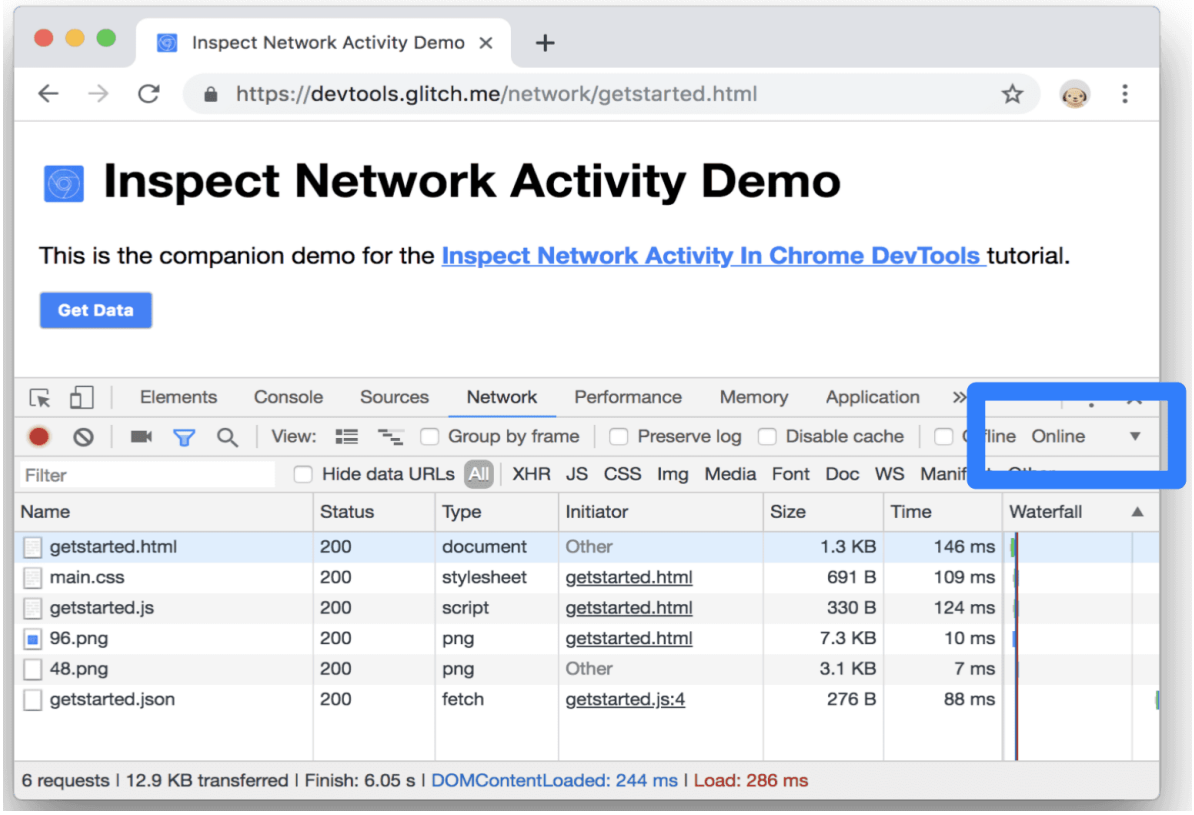


Рисунок 2.19 - Включення коригування

2) вибрати повільний 3G;

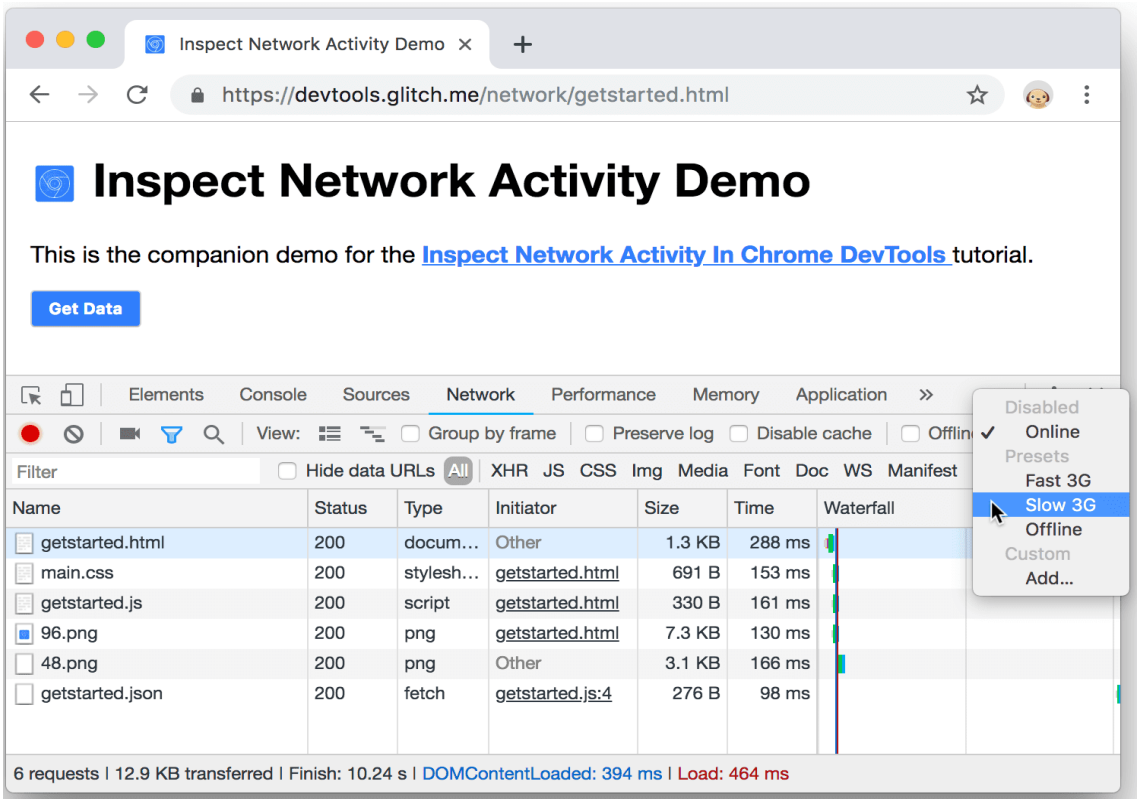


Рисунок 2.20 - Повільний 3G

3) натискаю і утримую кнопку перезапуску та вибираю "Очистити кеш", щоб виконати повне перезавантаження.

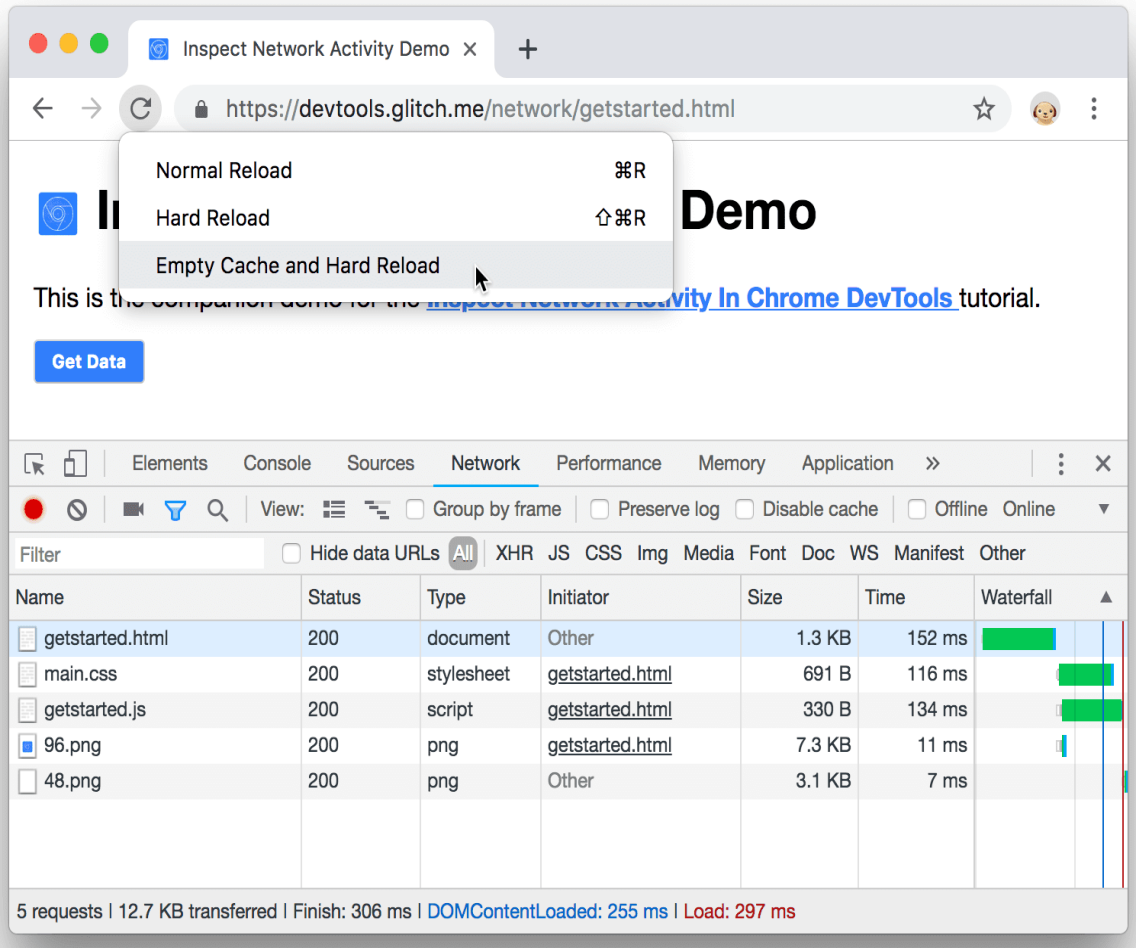


Рисунок 2.21 - Очистка кеш і жорсткий перезапуск

При повторному відвідуванні браузер зазвичай пришвидшує завантаження сторінки та пропонує деякі файли зі свого кешу. Якщо очистити кеш і перезавантажитися, браузер підключиться до мережі всіх ресурсів. Це корисно, якщо потрібно побачити, як відвідувачі завантажують сторінку під час першого відвідування веб-сайту.

Використовуючи скріншоти, можна побачити, як виглядала сторінка з часом під час завантаження.

1) натиснути кнопку "Зробити знімок екрана";

2) використати "Очищення кешу", щоб перезавантажити сторінку та повністю перезапустити робочий процес. Якщо потрібно нагадати, як це зробити,

змоделюю більш повільне з'єднання. На панелі скріншотів відображаються мініатюри того, як сторінка переглядалася в різні моменти процесу завантаження;

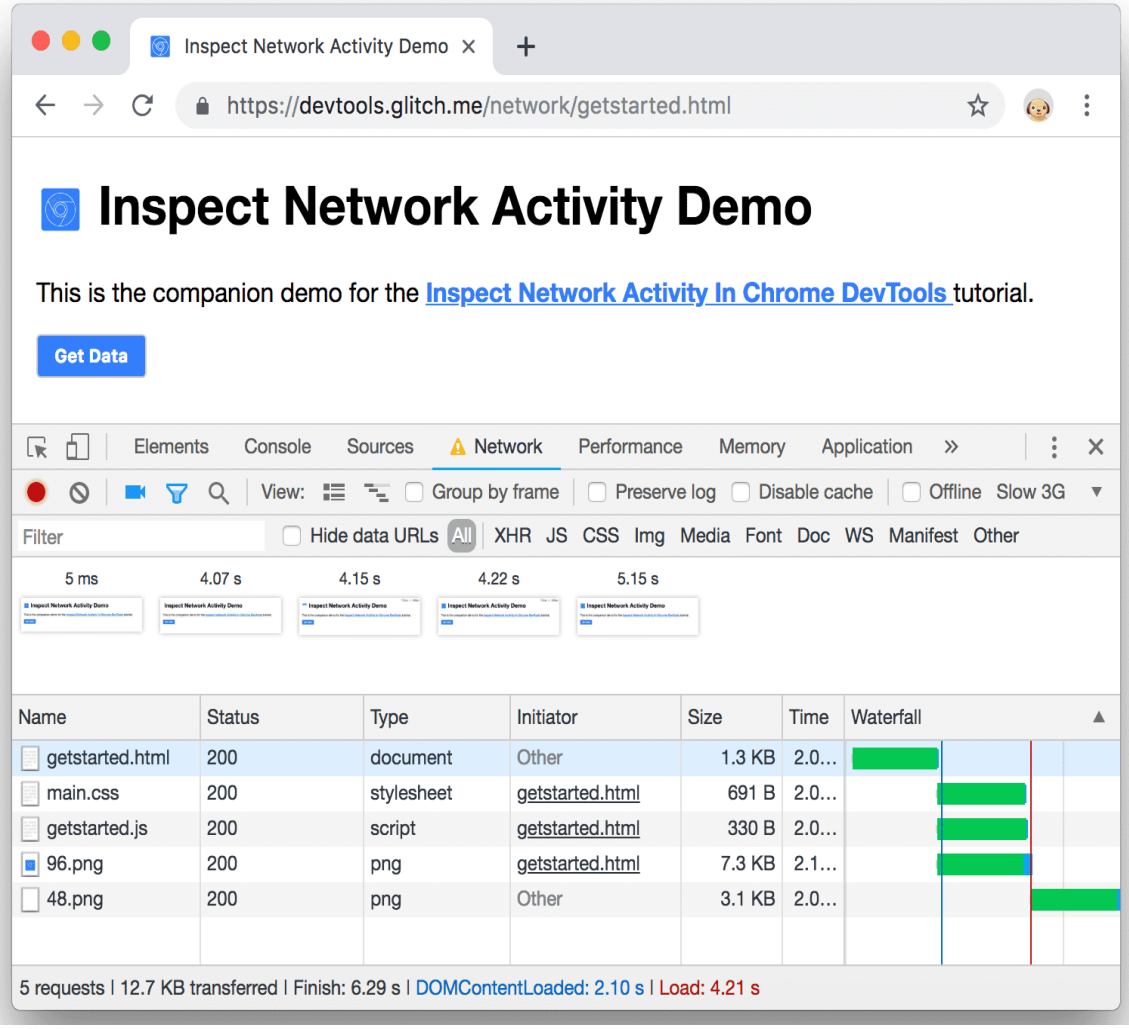


Рисунок 2.22 - Скріншот завантаження сторінки

3) натиснути на першу мініатюру. DevTools показує мережеву активність, яка відбувалася в той час;

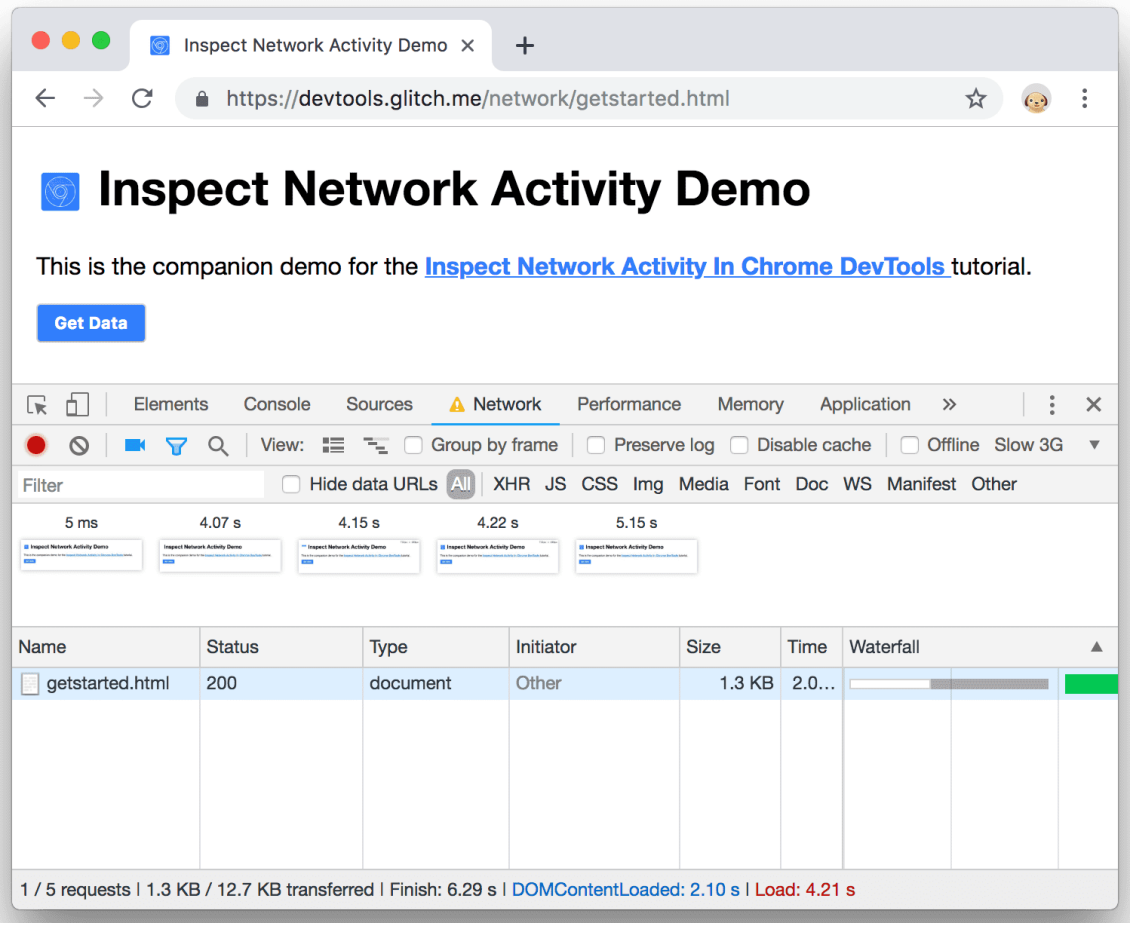


Рисунок 2.23 - Мережева активність, що виникає під час першого знімка екрана

4) натиснути кнопку "Зробити знімок екрана". Знову закрити панель знімків екрана;

5) перезавантажити сторінку.

Переглянути інформацію про джерело

Натиснути на джерело, щоб знайти додаткову інформацію про нього:

а) натиснути "Почати".html. З'явиться вкладка Заголовки. Використаю цю вкладку для керування заголовками HTTP;

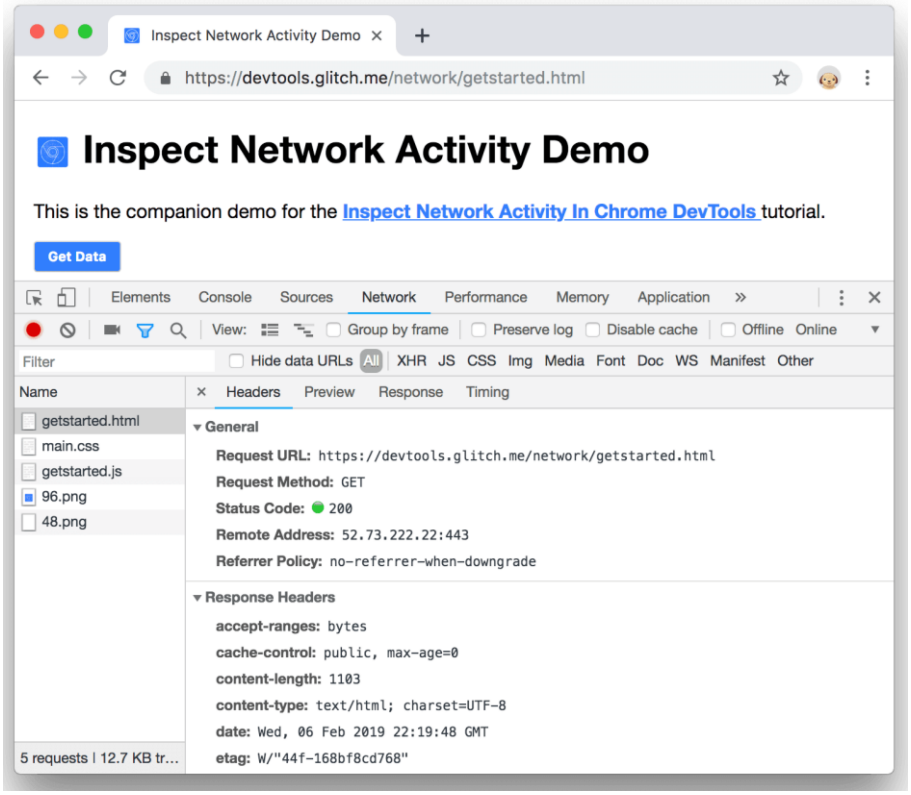


Рисунок 2.24 - Заголовки вкладок

b) перейду на вкладку Попередній перегляд. Там показано базовий рендеринг HTML;

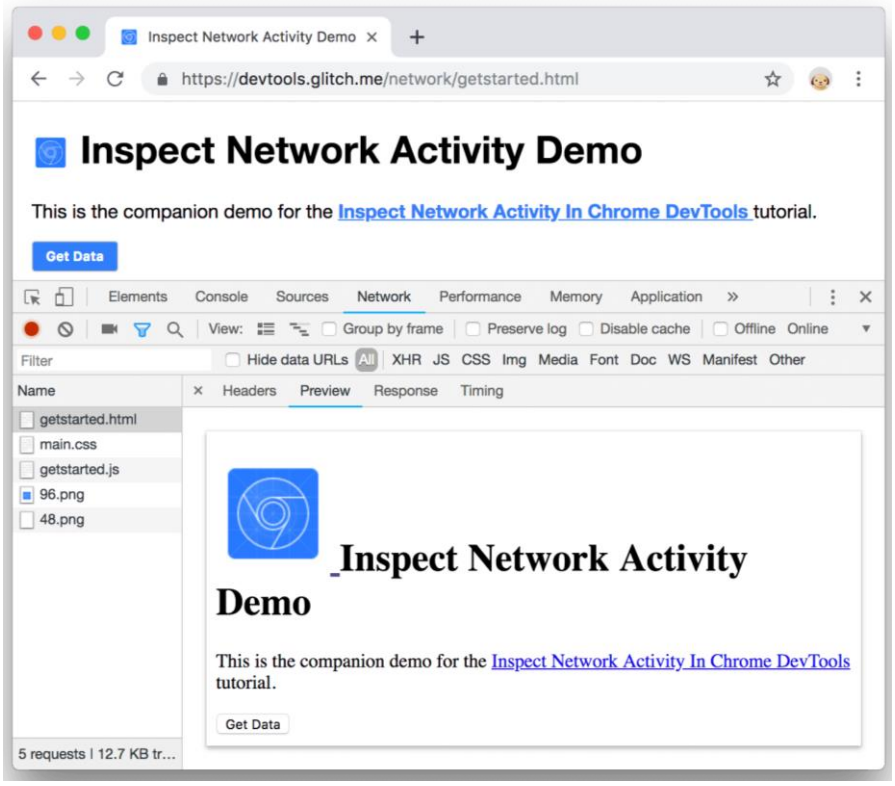


Рисунок 2.25 - Вкладка Попередній перегляд

Ця вкладка корисна, якщо API повертає код помилки HTML, а відображений HTML є більш читабельним, ніж вихідний код HTML, або якщо керувати зображеннями.

c) перейду на вкладку "відповіді". Відобразиться вихідний код HTML;

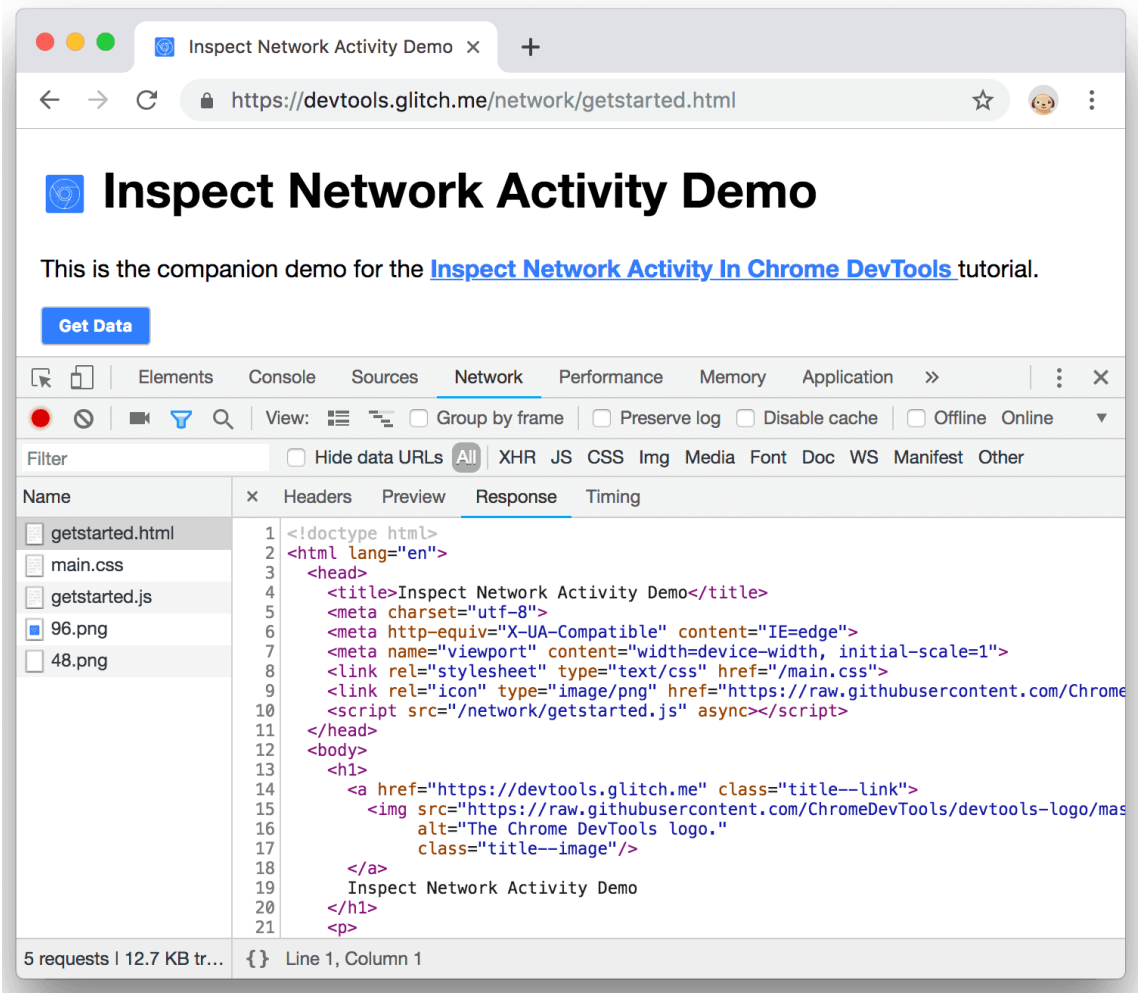


Рисунок 2.26 - Вкладка "Відповіді"

d) перейду на вкладку "час". Відображається дамп мережевої активності для цього ресурсу;

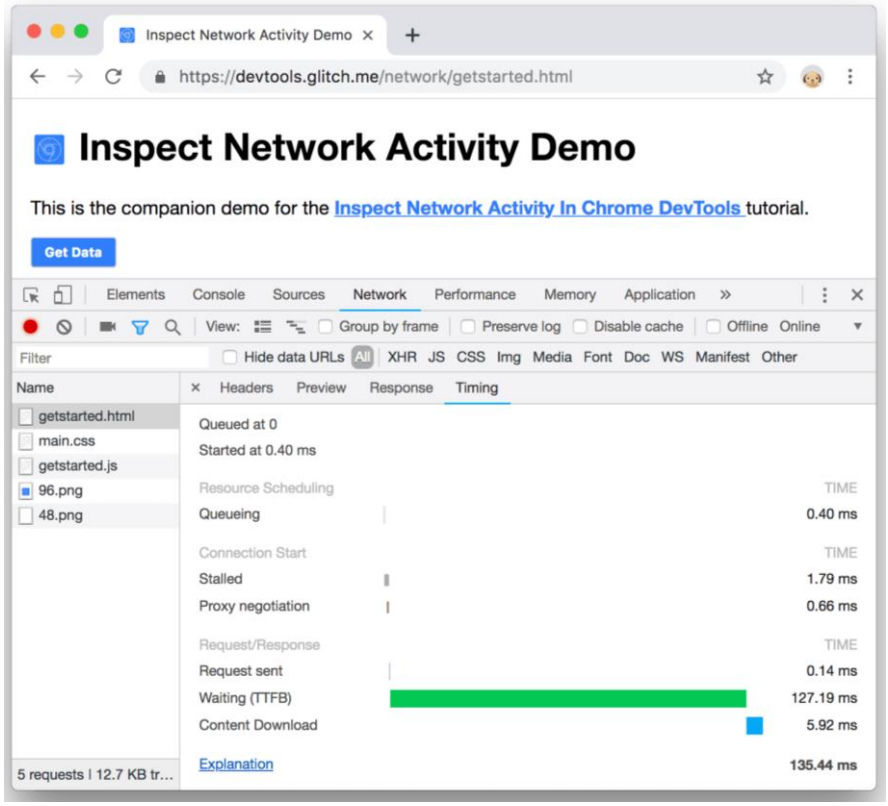


Рисунок 2.27 - Вкладка"Час"

е) натисну кнопку Закрити, щоб знову переглянути мережевий журнал.

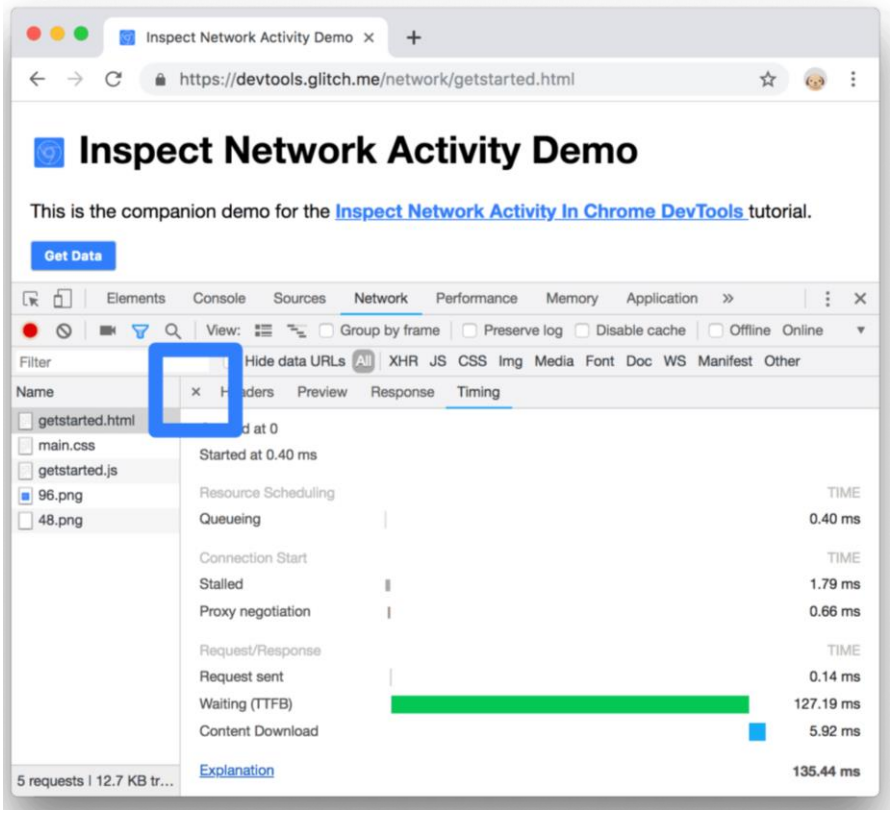


Рисунок 2.28 - Кнопка"Закрити"

Пошук мережевих заголовків та відповідей

Використовую рядок пошуку, щоб знайти заголовки та відповіді HTTP для всіх джерел для декомунізованого рядка або регулярного виразу.

Наприклад, треба перевірити, чи використовує ресурс розумну політику кешування.

1) натиснувши "Пошук". У лівій частині мережевого журналу відкривається дека пошуку;

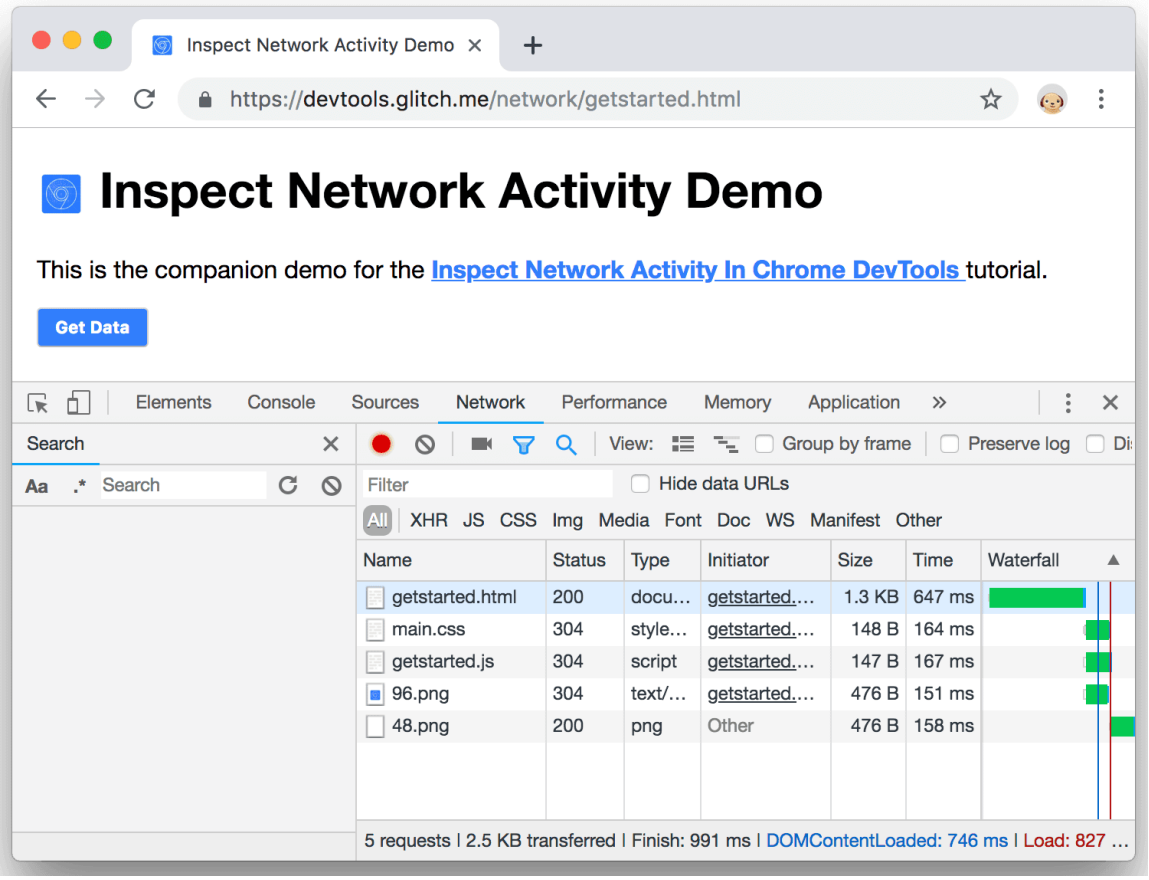


Рисунок 2.29 - Панель пошуку

2) вводиться елемент керування кешем і натискається клавіша ENTER. У рядку пошуку перераховані всі екземпляри деки кешу в заголовку джерела або вмісту;

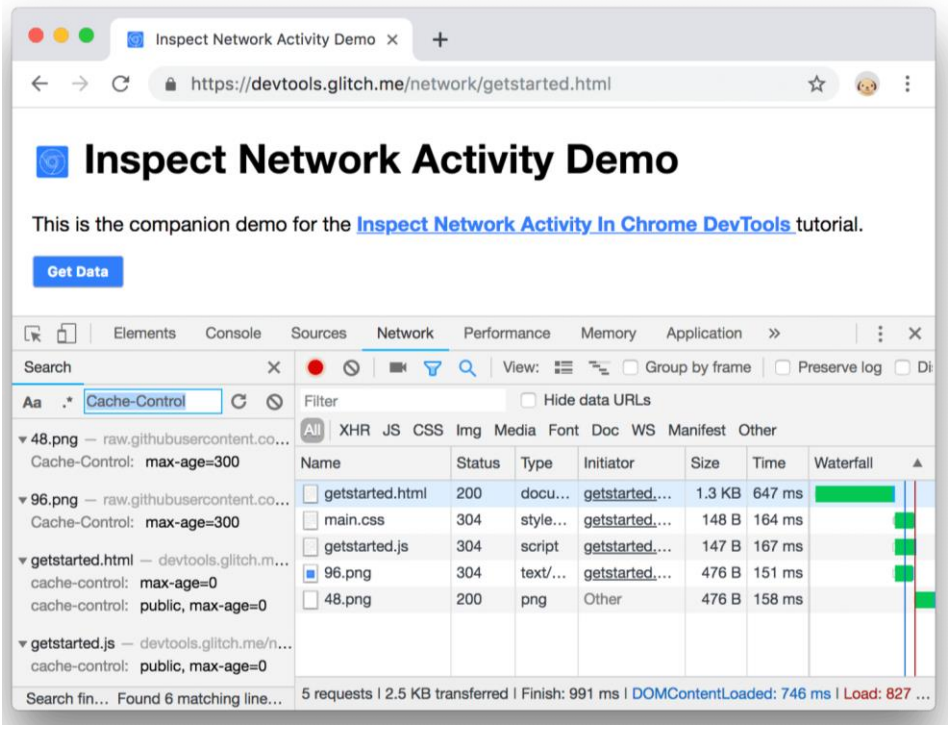


Рисунок 2.30 - Результати пошуку "Cache Control"

3) натисну на результат, щоб переглянути його. Якщо запит знаходиться в заголовку, відкриється вкладка Заголовок. Якщо запит знайдено у вмісті, відкриється вкладка відповідь.

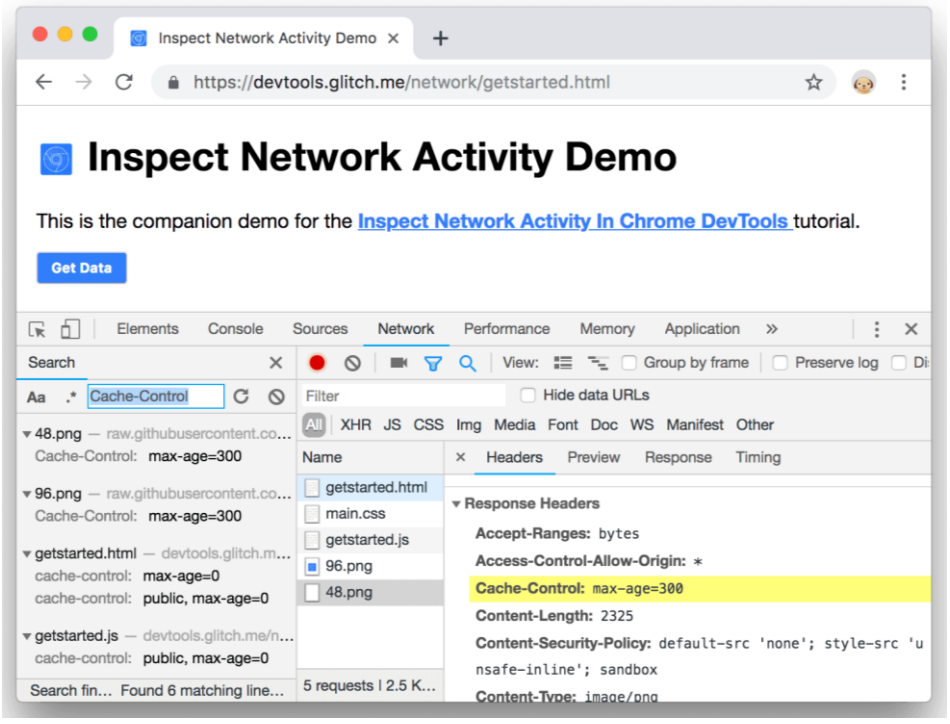


Рисунок 2.31 - Результати пошуку, виділені на вкладці "Заголовок"

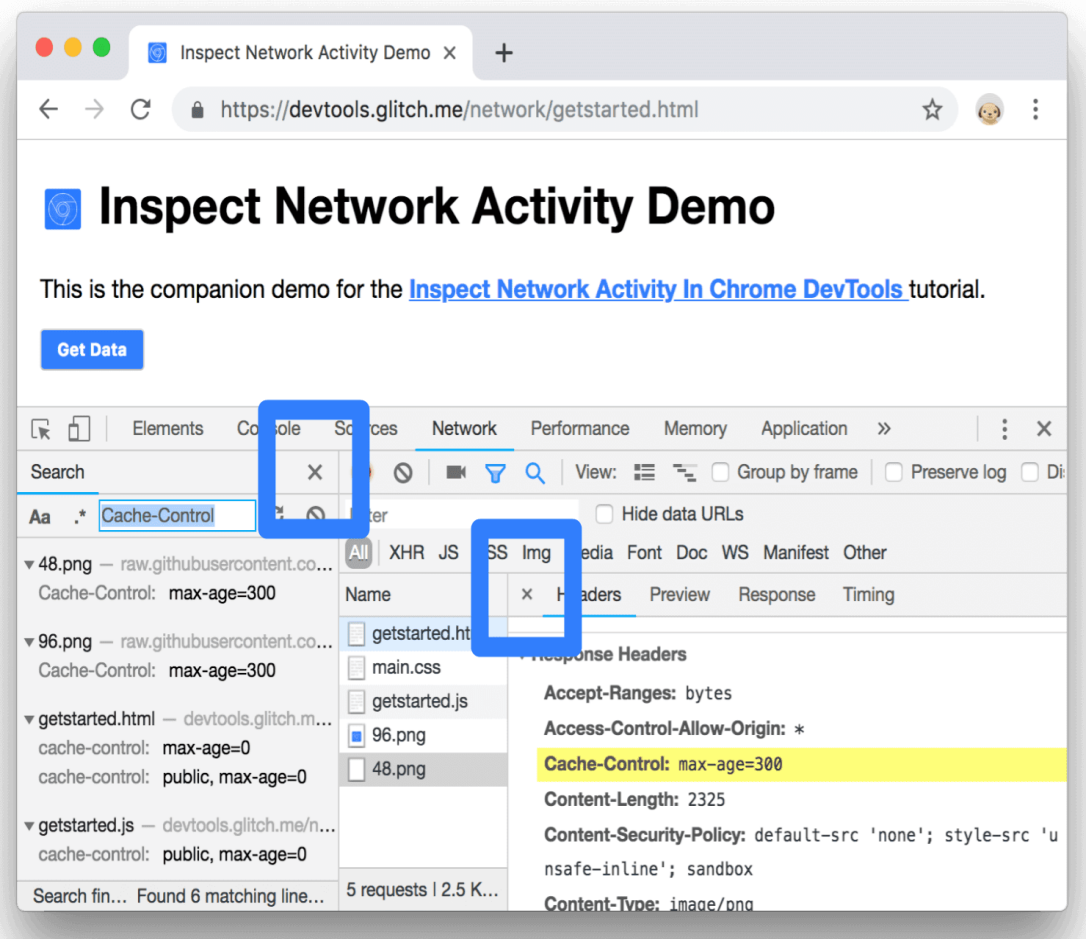


Рисунок 2.32 - Кнопка закриття

Фільтрація ресурсів

DevTools надає різні робочі процеси для фільтрації ресурсів, не пов'язаних із завданням.

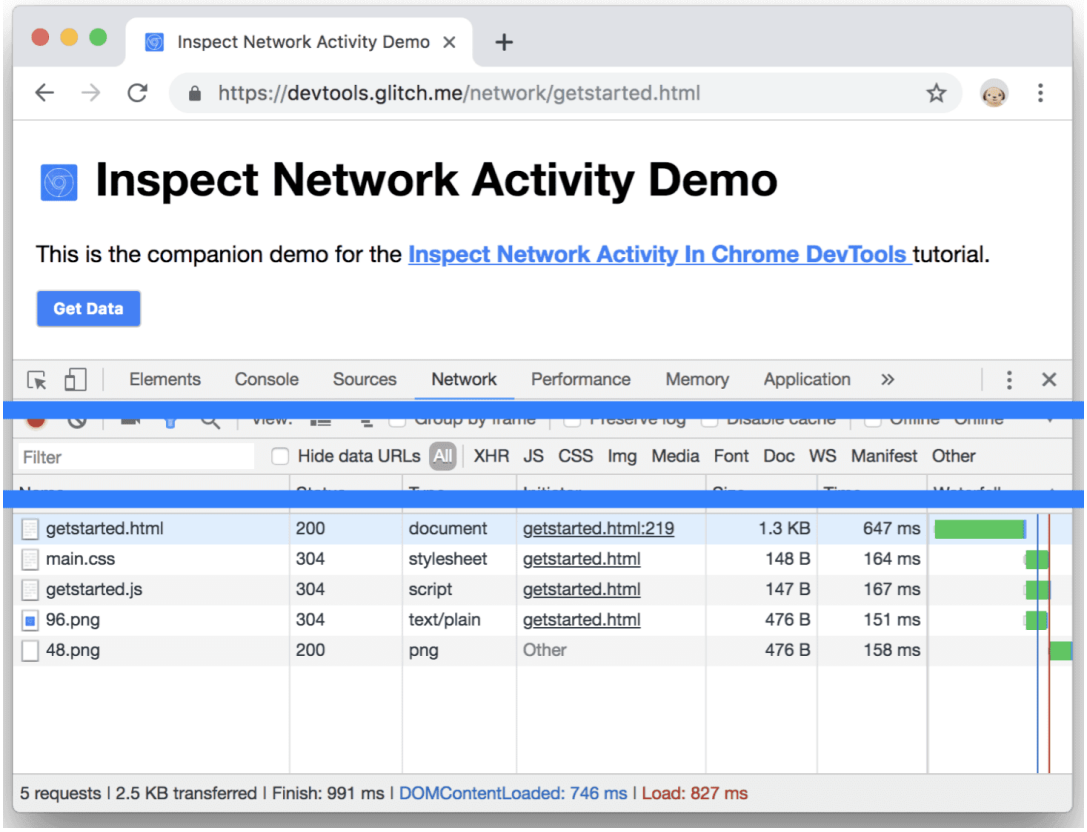


Рисунок 2.33 - Панель інструментів фільтра

Панель інструментів «Фільтр» має бути ввімкнено за умовчанням. Якщо ні:

1) натискаю «Фільтр», щоб відобразити його. Фільтрувати за рядком, регулярним виразом або атрибутом;

Текстове поле «Фільтр» підтримує багато різних типів фільтрації.

2) введу текстове поле png «Фільтр». Відображатимуться лише файли, що містять текст у форматі png. У цьому випадку єдиним файлом, який відповідає фільтру, є зображення PNG;

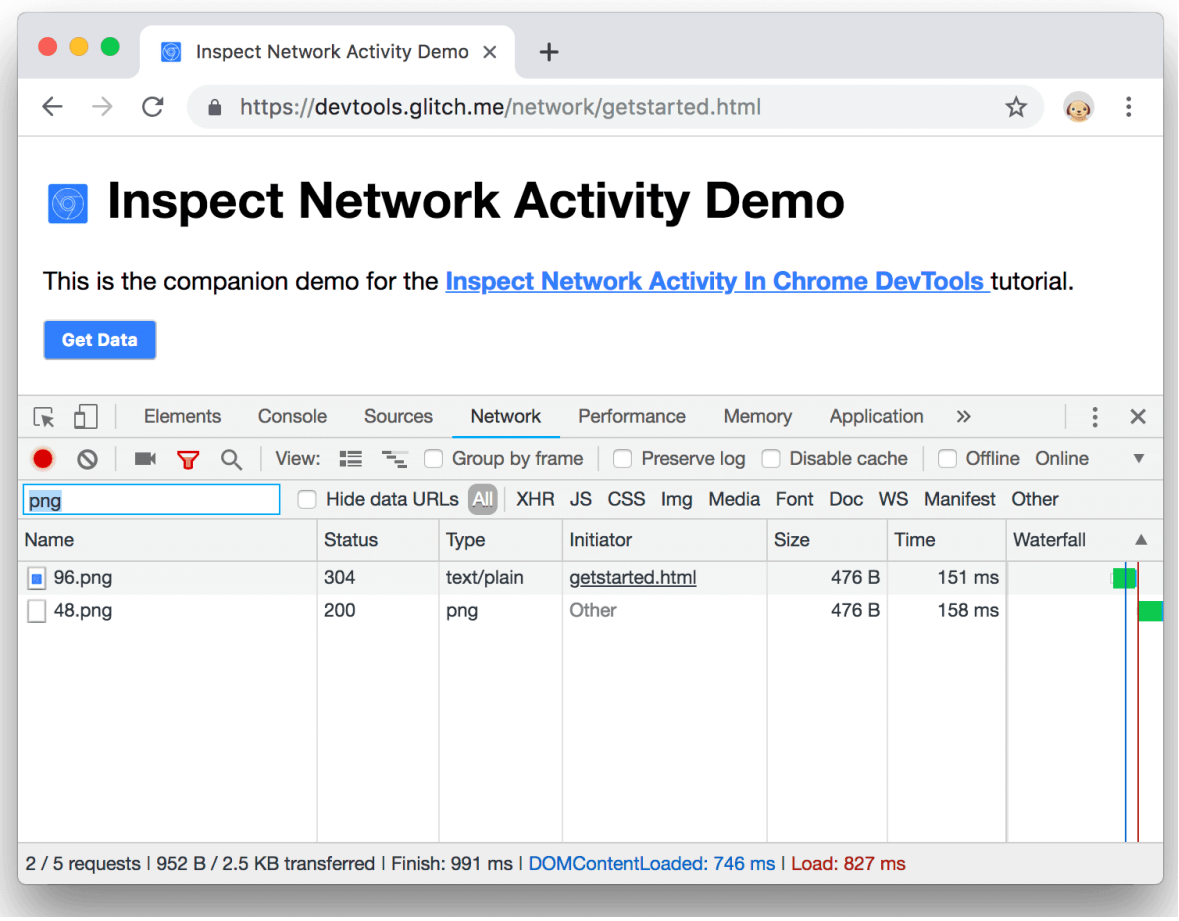


Рисунок 2.34 - Строковий фільтр

3) / Література.*\.[cj]s+∞/. DevTools виключає ресурси, імена файлів яких не закінчуються на J або c, а потім принаймні 1 або більше символів S.

РОЗДІЛ 3. МЕТОДОЛОГІЯ ДОСЛІДЖЕННЯ

3.1 Товстий клієнт та тонкий клієнт

У комп'ютерних технологіях – науках використовують термін «клієнт» то має на увазі програмне або апаратне забезпечення (термін), яке взаємодіє з сервером на сенісі запитів щоб отримати інфй. Клієнт є основною - важливою частиною клієнт-серверної архітектури. Прикладами клієнтів я можу сказати то є web-браузери. Вони виступають в якості web-клієнтів і відправляють запити на web-сервер, отримуючи у відповідь потрібну web-сторінку коли сервер це все обробив.

Клієнтів (Web-clint) у моделі клієнт-серверної архітектури я можу розділити на 2 категорії: тонкі та товсті. Також існують архітектури систем, які об'єднують можливості і тонких клієнтів, і товстих клієнтів та гібридні клієнти [19].

Товстий клієнт – це такий клієнт, який проводитиме запитувані користувачем операції незалежно від центрального сервера. Центральний сервер у такому варіанті архітектури використовується як сховище даних, обробка та надання яких переноситься на робочу машину клієнта.

Товстий клієнт — це робоча станція, або комп’ютер, або смартфон (планшет), який працює під керуванням власної операційної системи та має все програмне забезпечення, необхідне для реалізації завдань користувача.

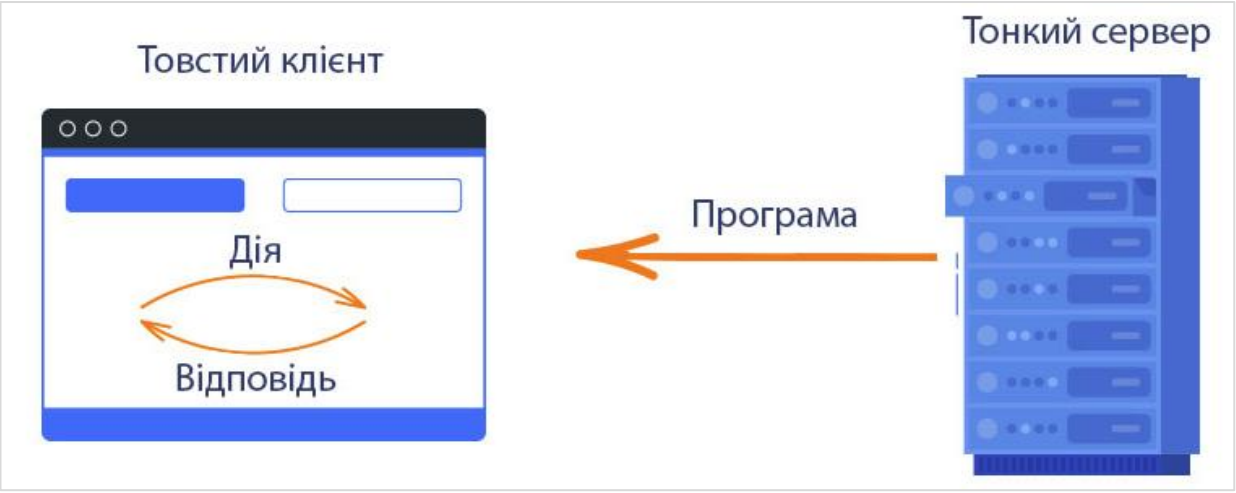


Рисунок 3.1 – Схема роботи товстого клієнту та тонкого серверу

Плюси товстих клієнтів:

a) широка функціональність системи;

b) режим з багатьма користувачами у використанні;

c) робота в режимі оффлайн того що дані на клієнті збіргаются;

d) висока швидкодія всеедині;

e) зменшення залежності від дорогих і складних серверів.

Мінуси:

a) кожна робоча машина потребує постійного обслуговування;

b) вдосконалення індивідуального обладнання до рівня додатків, які будуть використовуватися;

c) можливість виникнення проблем з віддаленим доступом до даних;

d) великий розмір розподілу;

e) в залежності від платформи, для якої розрахований клієнт.

Тонкий клієнт — це тип клієнта, який передає завдання з обробки даних на сервер без використання його обчислювальної потужності для їх обробки. Комп'ютерні ресурси такого клієнта дуже обмежені, їх повинно вистачати лише на запуск необхідних мережевих додатків, наприклад, веб-інтерфейсу. [20].

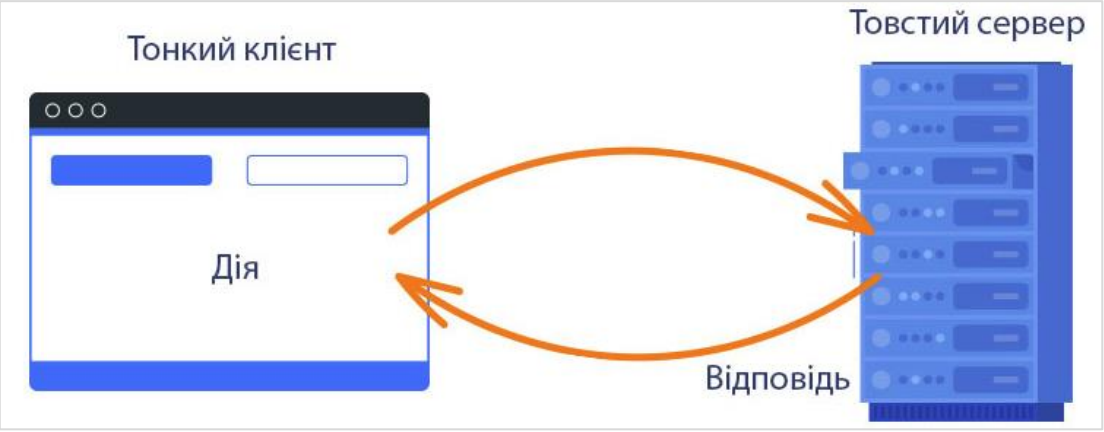


Рисунок 3.2 – Схема роботи тонкого клієнту та товстого серверу

Іншим прикладом використання тонкого клієнта є комп’ютер із встановленим веб-браузером, який використовується для запуску веб-

додатків.Тонкі клієнти характеризуються використанням термінального простору. В данному випадку термінальний сервак використовується щоб надсилання та отримання даних користувача, що це головна відмінність від обробки даних, яка не залежить від щільних клієнтів [21].

На додаток до використання версії програмного забезпечення тонкого клієнта існують апаратні рішення тонкого клієнта. Ці пристрої, які не обов'язково мають власний жорсткий диск, використовують спеціальну локальну операційну систему, основним завданням якої є встановлення з'єднання з сервером, який використовується для вирішення завдань користувача.

Плюси тонких клієнтів:

a) менше обслуговування апаратного та програмного забезпечення користувача;

b) знижує ризик збоїв у роботі, оскільки файли та програми зберігаються на центральному сервері;

c) нижчі вимоги до обладнання порівняно з товстими клієнтами.

Недоліки:

a) спільна точка збою: збій сервера впливає на всіх користувачів;

b) неможливість роботи без підключення до мережі;

c) коли з відео- та аудіоданими виконується багато роботи (особливо створення та редагування), централізація тонких клієнтів може значно знизити продуктивність центрального сервера.

3.1.1 Різниця між тонкими і товстими клієнтами

Основна відмінність між тонкими та товстими клієнтами полягає в способі обробки даних. Товстий клієнт працює з даними, використовуючи свої апаратні та програмні можливості, і підключається до сервера лише для отримання даних із

бази даних. Тонкі клієнти, з іншого боку, використовують для обробки даних центральний сервер, який забезпечує лише інтерфейс користувача, необхідний для роботи користувача. Тому вже застарілі комп'ютери можна використовувати як тонкі клієнти.

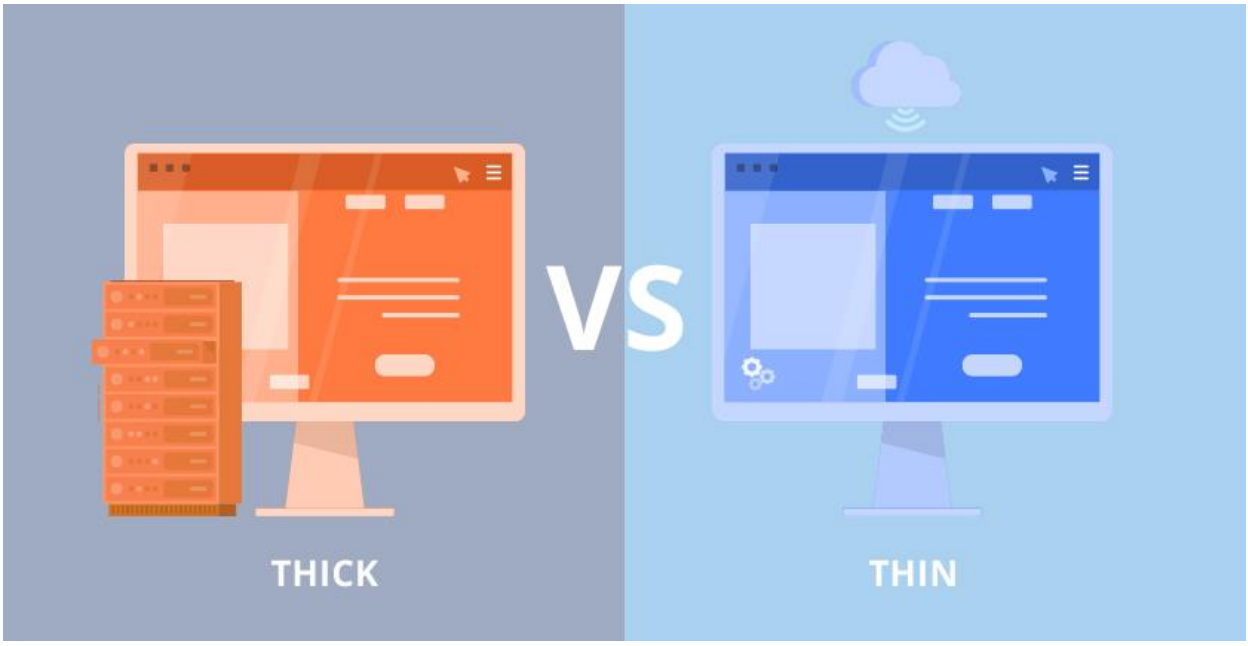


Рисунок 3.3 – Інсталяція порівнянь

Порівняння товстих і худих споживачів за основними ознаками:

Автономність – товсті клієнти працюють без центрального сервера і використовують власні пристрої. Тонкі клієнти майже повністю залежать від центрального сервера та його доступних ресурсів.

Ресурсовитратні – товсті клієнти використовують більше локальних ресурсів, оскільки виконують усі операції самостійно. Локальні ресурси тонких клієнтів призначені лише для встановлення сеансу зв’язку з сервером.

Зберігання даних – товсті дані користувача зберігаються локально на робочому столі.

Підключення до мережі – товсті клієнти можуть працювати в автономному режимі, тоді як тонкі клієнти потребують постійного підключення до Інтернету.

Впровадження - товсті клієнти вимагають великих витрат, оскільки кожна робоча машина повинна оновлюватися окремо для конкретних завдань користувача. Тонкі клієнти не такі дорогі, тому можна використовувати

найпростіше комп’ютерне обладнання. Основна вартість при роботі з тонкими клієнтами - це високопродуктивний сервер і його налаштування.

Безпека – під час роботи з товстими клієнтами можуть виникнути підвищені проблеми безпеки даних, оскільки ризик втрати даних на окремих робочих станціях високий. Тонкі клієнти вважаються більш безпечними, оскільки дані зберігаються на сервері.

3.1.2 Приклади використання та додатки

Всі юзери (користувачі) так чи інакше стикнутся з тонкими і товстими клієнтами під час роботи або просто за допомогою комп'ютера для вирішення своїх завдань.

З апаратної точки зору жирний клієнт вважається функціональною машиною, яку користувач використовує для виконання своїх завдань. Такий комп'ютер оснащений необхідним апаратним забезпеченням для вирішення поставлених завдань. Тонкий клієнт також може працювати як окремий робочий стіл. Такі тонкі клієнти дуже компактні, використовують пасивне охолодження. Тонкі клієнти часто використовуються як офісні машини [9].

З боку програмного забезпечення, прикладами товстих клієнтів є програми та додатки для спільної роботи, особливо якщо вони встановлені та обробляються на певному комп’ютерному пристрої. Деякі приклади таких програм: Microsoft Office 365 і Adobe Creative Cloud, Yahoo Messenger, Microsoft Outlook.

Прикладами тонких клієнтів є браузери та програми та онлайн-ігри. Пристрої, які використовуються для потокової передачі медіа, наприклад Chromecast і Apple TV, із встановленими програмами потокової передачі, такими як Clash of Clance, технічно є прикладами тонких клієнтів. Сайти пошукових систем Google і Yahoo також можна віднести до тонких клієнтів.

Thinstation часто використовується для створення тонких клієнтів. Це спеціалізований дистрибутив Linux для ініціалізації та роботи з тонкими клієнтами.

Сам дистрибутив невеликий за розміром і пропонує можливість роботи з різними протоколами віддаленого доступу (RDP, telnet, SSH, Citrik ICA та інші) [8].

Вибір клієнта для використання часто залежить від того, які цілі переслідує користувач, які апаратні та програмні ресурси доступні. Виходячи з плюсів і мінусів реалізації кожного підходу, можна вибрати найбільш оптимальний варіант.

3.2 Кешування як спосіб оптимізації

Кешування – це спосіб зберігання інформації в пам’яті для швидкого доступу. Цей метод спрямований на підвищення продуктивності системи та її масштабованості. Це корисно, коли вам часто доводиться читати та записувати великі обсяги одних і тих самих даних до основної пам’яті. Прикладом швидкої пам'яті є оперативна пам'ять, але це може бути і звичайна локальна пам'ять, наприклад жорсткий диск клієнта. Кешування може вирішити багато проблем, пов'язаних зі швидкістю та ефективністю програмного забезпечення. Зокрема, цей підхід може:

a) скорочує час відповіді на запити, що потребують великих ресурсів;

b) зменшити навантаження на сервер, кешуючи певні результати запитів замість того, щоб щоразу отримувати їх із бази даних;

c) оптимізувати витрати на обслуговування системи за рахунок зменшення потреби в ресурсах;

d) підвищення стабільності системи, наприклад, під час пікового навантаження або певних типів DDoS-атак;

e) виконайте кілька збережень, щоб уникнути завантаження основної пам’яті. Для кращого розуміння кешування, наведу кілька прикладів, як це працює в нашому застосунку. Для кращого розуміння кешування, наведу кілька прикладів, як це працює.

3.2.1 Кешування на рівні клієнта

Кліентперший раз робить запит на дані, запит надсилається до серверної частини програми, а відповідь кешується на пристрої. І коли користувач знову захоче отримати їх, програма звернеться до внутрішньої пам’яті пристрою [3].

3.2.2 Кешування на рівні бекенду

Особливість кешування бекенду полягає в тому, що все залежить від того, хто в мережі, і постійно змінюється. Ось чому додатку потрібна найновіша фонова інформація. Але формування відповіді на досить складну модель вимагає часу. Щоб серверна частина відповідала якомога швидше, ми кешуємо кожну модель експерта окремо, створюємо колекцію та посилаємося на кеш кожного експерта.

Існують різні способи кешування даних: кешування, читання, повернення, запис тощо. Кожен із них має свої плюси та мінуси. Слід розуміти, що їх можна використовувати як окремо, так і разом для підвищення ефективності.

3.3 Порівняння NGINX та Apache

Основна відмінність між Apache і Nginx полягає в тому, як вони обробляють підключення та трафік. Це впливає на те, як вони витримують різні навантаження. Apache Apache працює в багатопроцесорному режимі. Програма має кілька модулів, які керують обробкою сполук. Ця архітектура дозволяє адміністраторам дуже просто керувати підключеннями. mpm_prefork – цей модуль створює окремий однопотоковий процес для обробки кожного запиту. Кожен дочірній процес може одночасно керувати лише одним підключенням. Поки кількість запитів менша за кількість запущених MPM, веб-сервер працюватиме дуже швидко. Але як тільки

запити перевищують кількість процесів, продуктивність сильно знижується. Тому в багатьох випадках Apache не дуже хороший вибір. Кожен процес споживає оперативну пам'ять, тому важко використовувати MPM на слабкому обладнанні. Але це все одно може бути чудовим вибором для роботи з певними компонентами. Наприклад, php не підтримує потоки, тому це єдиний безпечний спосіб роботи з mod_php [12].

Apache

Apache надає кілька багатопроцесорних модулів (MPM), які відповідають за обробку запиту клієнта. Це дозволяє адміністраторам установлювати політику обробки з’єднань. Нижче наведено список модулів Apache MPM:

a) mpm_prefork – цей модуль створює один потоковий процес для кожного запиту. Кожен процес може обробляти лише одне підключення за раз. Поки кількість запитів менша за кількість процесів, цей MPM працює дуже швидко. Однак продуктивність швидко знижується, коли кількість запитів починає перевищувати кількість процесів, тому в більшості випадків це не найкращий вибір. Кожен процес споживає значну кількість оперативної пам’яті, що ускладнює масштабування цього MPM. Але його можна використовувати з компонентами, які не призначені для роботи в багатопоточному середовищі. Наприклад, PHP не є безпечним, тому цей MPM рекомендується як безпечний метод mod_php;

b) mpm_worker – Цей модуль створює процеси, кожен з яких може керувати кількома потоками. Кожен потік може обробляти одне з'єднання. Потоки набагато ефективніші за процеси, а це означає, що mpm_worker масштабується набагато краще, ніж mpm_prefork. Оскільки існує більше потоків, ніж процесів, це означає, що нове підключення може бути оброблено негайно вільним потоком замість того, щоб чекати, поки процес стане вільним.

c) mpm_event — цей модуль схожий на mpm_worker, але оптимізований для роботи з підключеннями KeepAlive. Коли mpm_worker увімкнено, з’єднання підтримує потік незалежно від того, чи з’єднання активне чи підтримується. Mpm_event резервує окремі потоки для підтримки з’єднань і окремі потоки для

активних з’єднань. Це дозволяє не встановлювати модуль в порти життєзабезпечення, що необхідно для швидкої роботи. Цей модуль позначено як стабільний у Apache версії 2.4.

Як ми бачим що, Apache пропонує гнучкі можливості для вибору різних алгоритмів обробки з'єднань і запитів.

Nginx з’явився пізніше, ніж Apache, тому його розробник був більш обізнаний про конкуруючі проблеми, з якими стикаються сайти під час масштабування. Завдяки цим знанням Nginx спочатку був розроблений навколо асинхронних неблокуючих алгоритмів.

Nginx створює робочі процеси, кожен з яких може обслуговувати тисячі підключень. Співробітники досягають такого результату завдяки механізму, заснованому на швидкому циклі перевірки та обробки транзакцій. Відокремлення основної роботи від обробки з’єднання дозволяє кожному співробітнику виконувати свою роботу та відволікати увагу від обробки з’єднання лише тоді, коли відбувається нова подія.

Кожне з’єднання, оброблене робітником, знаходиться в циклі подій разом з іншими з’єднаннями. У цьому циклі події обробляються асинхронно, що дозволяє обробляти завдання блокуючим способом. Коли з'єднання закрито, воно видаляється з циклу [4].

Такий спосіб обробки підключень дозволяє Nginx неймовірно добре масштабуватися з обмеженими ресурсами. Оскільки сервер є однопотоковим і не запускає процеси для кожного з’єднання, використання пам’яті та ЦП є відносно рівним за великого навантаження.

Інтерпретація базується на файлі та URI.

Спосіб, яким веб-сервер інтерпретує запит і відображає його в системному ресурсі, є ще однією характеристикою двох серверів.

Apache може інтерпретувати запит як фізичний ресурс у файловій системі або як URI, який потребує подальшої обробки. Перший тип запиту використовує блоки специфікації <Directory> або <File>, другий - блоки <Location>. Оскільки Apache спочатку був розроблений як веб-сервер, він інтерпретує запити як ресурси

файлової системи за замовчуванням. Він бере корінь документа веб-сервера та доповнює його частиною запиту, яка слідує за іменем хоста та номером порту, щоб знайти запитаний файл. Загалом, ієрархія файлової системи представлена в Інтернеті та доступна у вигляді дерева документів [3]

Apache надає кілька варіантів, якщо запит не відповідає файлу у файловій системі. Використання блоків <Location> — це спосіб працювати з URI без відображення файлової системи. Ви також можете використовувати регулярні вирази, щоб встановити більш гнучкі параметри для всієї файлової системи.

Оскільки Apache може працювати як з файловою системою, так і з мережевим простором, він в основному покладається на методи файлової системи. Це відображено в деяких дизайнерських рішеннях в архітектурі веб-сервера, наприклад у використанні файлів .htaccess для конфігурації на рівні каталогу. Інструкції Apache не рекомендують використовувати блоки URI для обмеження доступу до запитів файлової системи.

Nginx розроблено для роботи як веб-сервера, так і проксі-сервера. Тому він в першу чергу працює з URI і за необхідності перетворює їх на запити до файлової системи [15].

Ця функція відображається в тому, як Nginx будує та інтерпретує конфігураційні файли. Nginx не має можливості створити конфігурацію для певного каталогу, замість цього він аналізує URI.

Наприклад, основними конфігураційними блоками Nginx є <server> і <location>. Блок <server> вказує хост для обслуговування, блок <location> керує частиною URI, яка йде після імені хоста та номера порту. Тому запит інтерпретується як URI, а не шлях до файлової системи.

Для запитів на статичні файли всі запити мають бути зіставлені з шляхом до файлової системи. Спочатку Nginx вибирає сервер і блоки розташування для обробки запиту, а потім відображає документ на базовий URI, як налаштовано.

Ці підходи (інтерпретація запитів як шляхів файлової системи та URI) можуть здаватися схожими, але той факт, що Nginx обробляє запити як URI замість шляхів файлової системи, дозволяє йому легше обробляти як роль веб-сервера, так

і роль. проксі-сервер Nginx налаштований на відповідь на різні шаблони запитів. Nginx не використовує файлову систему, доки не буде готовий обслуговувати запит, що пояснює, чому він не реалізує нічого подібного до файлів .htaccess [3].

Модулі

Як Apache, так і Nginx можна розширити за допомогою модульної системи, але методи, які використовуються для реалізації модульної системи, принципово відрізняються.

Модульна система Apache дозволяє динамічно завантажувати та вивантажувати модулі відповідно до ваших потреб під час роботи сервера. Ядро Apache завжди доступне, а модулі можна вмикати та вимикати, щоб додавати або видаляти функціональність головного сервера.

Apache використовує цю функцію для виконання різноманітних завдань. Через зрілість платформи багато модулів можуть змінювати ключові функції сервера, наприклад, модуль mod_php дозволяє включати інтерпретатор PHP у кожен робочий файл.

Використання модуля не обмежується динамічною обробкою запитів. Серед інших функцій модуля: зміна URL-адреси (переписування URL-адреси), автентифікація клієнта, безпека сервера, журналювання, кешування, стиснення, проксі, обмеження частоти запитів, шифрування. Динамічні модулі дозволяють значно розширити функціональність ядра.

Nginx також має модульну систему, але вона сильно відрізняється від підходу, який використовується в Apache [2,3].

У Nginx модулі не завантажуються динамічно, але їх потрібно вибрати та скомпілювати за допомогою ядра сервера. Для багатьох користувачів Nginx здається менш гнучким з цієї причини. Це особливо актуально для користувачів із невеликим досвідом ручної розробки програм, які хочуть використовувати системи керування пакетами. Зазвичай розробники дистрибутивів намагаються створювати пакунки для всіх часто використовуваних модулів, але якщо вам потрібен

нестандартний модуль, ви повинні скомпілювати його самостійно з оригінальних джерел.

Однак модулі Ngin дуже корисні та затребувані, ви можете вказати, що хочете від сервера, і включити лише ті модулі, які вам потрібні. Деякі користувачі вважають цей підхід безпечнішим, оскільки довільні модулі не можуть це зробити для підключення до сервера [2].

Модулі Nginx реалізують ті ж функції, що й модулі Apache: проксі, стиснення даних, обмеження частоти запитів, журналювання, модифікація URL-адреси, геолокація, автентифікація, шифрування, потокове передавання, функції електронної пошти.

3.3.1 Підтримка, сумісність, екосистема та документація

У процесі використання програми важлива створена навколо неї екосистема та можливість отримання підтримки.

Оскільки Apache був популярний так довго, проблем із підтримкою немає. Ви просто знайдете велику кількість документації як від розробників Apache, так і від сторонніх авторів. Ця документація охоплює всі можливі сценарії використання Apache, включаючи взаємодію з іншими програмами. Є багато інструментів і веб-проектів, які поставляються з Apache. Це стосується як самих проектів, так і систем управління пакетами.

Загалом Apache отримує більше підтримки, ніж проекти сторонніх розробників, просто тому, що він існує вже давно. Адміністратори також мають більше досвіду роботи з Apache, оскільки більшість людей починають роботу на стороні хостингу, де Apache більш популярний через підтримку файлів .htaccess [2].

Nginx зазвичай використовується там, де вимоги до продуктивності високі, а в деяких областях він все ще повільний. Раніше було важко знайти гідну підтримку цього веб-сервера англійською мовою, оскільки розробка та документація

російською мовою знаходяться на початковій стадії. Разом із зростанням інтересу до проекту документація була перекладена англійською мовою, і тут ви можете знайти достатню кількість документації як від розробників веб-серверів, так і від незалежних авторів.

Сторонні розробники програмного забезпечення також починають підтримувати Nginx, і деякі з них пропонують користувачеві вибрати конфігурацію на основі Apache або Nginx. Навіть якщо програма не підтримує Nginx, важко написати власну конфігурацію для інтеграції програми з Nginx.

3.3.2 Спільне використання Apache та Nginx

Вивчивши плюси і мінуси Apache і Nginx, ви повинні мати уявлення про те, який сервер краще підходить для ваших завдань. Однак кращих результатів можна досягти, використовуючи обидва сервери разом. Зазвичай Nginx розміщують перед Apache як зворотний проксі. У такій конфігурації Nginx називається інтерфейсом, а Apache — сервером. Завдяки такому підходу Nginx обслуговує всі запити клієнтів, і ми отримуємо переваги від його здатності обробляти багато конкуруючих запитів.

Nginx обслуговує статичний вміст незалежно, а для динамічного вмісту, наприклад запитів зі сторінок PHP, він передає запит до Apache, який відтворює сторінку, повертає його до Nginx, який, у свою чергу, пересилає його користувачеві.

Ця конфігурація дуже популярна, Nginx використовується для сортування запитів. Він сам обробляє запити та пересилає Apache лише ті запити, які не може обробити сам, таким чином зменшуючи навантаження на Apache. Ця конфігурація дозволяє горизонтально масштабувати вашу програму: ви можете встановити кілька серверних модулів за одним зовнішнім інтерфейсом, і Nginx розподілить навантаження між ними, підвищуючи відмовостійкість програми.

3.4 Балансувальник як спосіброзподілення навантаження

Балансування навантаження стосується ефективного розподілу вхідного мережевого трафіку між серверною групою серверів. А роль контролера полягає в розподілі навантаження між декількома встановленими бек-енд-серверами. Існує кілька типів пристроїв розподілу навантаження:

a) програмний балансувальник навантаження;
b) балансувальник навантаження мережі;
c) балансувальник навантаження шлюзу;
d) класичний балансир навантаження.

Також існує багато пристроїв для розподілу навантаження, і всі вони мають різне призначення:

a) Гапроксі;
b) Nginx;
c) mod_athena;
d) Light;
e) Balance;
f) Віртуальний сервер Linux (LVS).

3.4.1 Балансування навантаження Nginx

Nginx — це високопродуктивний веб-сервер, який також можна використовувати як балансир навантаження, який є процесом розподілу веб-трафіку між кількома серверами за допомогою Nginx [1].

Це гарантує, що сервер не буде перевантажений і всі запити оброблятимуться вчасно. Nginx використовує різні алгоритми для визначення оптимального

розподілу трафіку, а також може бути налаштований для забезпечення відновлення після відмови, якщо один із серверів виходить з ладу.

Для збалансування у HTTP-трафіку у групі серверів, я буду використовувати Nginx з відкритим кодом або Nginx Plus [1].

Точніше, я використовую відкритий вихідний код р Nginx для налаштування своїх балансувальників навантаження, і саме це я збираюся показати вам дипломній роботі.

Переваги балансування навантаження допомагає масштабувати програми, справляючись зі стрибками трафіку без збільшення витрат на хмару. Це також допомагає усунути проблему єдиної точки відмови. Оскільки навантаження розподілене, служба продовжує працювати, якщо один із серверів виходить з ладу.

РОЗДІЛ 4. ПРАКТИЧНА РЕАЛІЗАЦІЯ

4.1. Налаштування Nginx як балансувальник навантаження

Першим кроком є встановлення Nginx. Він може бути встановлений у Debian, Ubuntu або CentOS системи. Я збираюся використовувати Ubuntu, яку вже налаштував на своєму віртуальному сервері.

```
sudo apt-get update
sudo apt-get install nginx
```

Рисунок 4.1 – Команди установка Nginx

Далі я створюю файл конфігурації

```
cd /etc/nginx/sites-available/
sudo nano default
```

Рисунок 4.2 – Команда для створення файлу конфігурацііі серверу

```
http {
   upstream app{
      server 10.2.0.100;
      server 10.2.0.101;
      server 10.2.0.102;
   }
   server {
      listen 80;
      server_name mydomain.com;
      location / {
          include proxy_params;
          proxy_pass http://app;
          proxy_redirect off;
          proxy_http_version 1.1;
          proxy_set_header Upgrade $http_upgrade;
          proxy_set_header Connection "upgrade";
      }
   }}
```

Рисунок 4.3 – Налаштування балансувальника

У файлі я визначаю директиву upstream та server. Upstream визначає, куди Nginx передаватиме запити після їх отримання. Вона містить IP-адреси групи серверів (бекенди серверів), на які можуть бути надіслані запити в залежності від вибраного методу регулювання навантаження. За промовчанням Nginx використовує метод балансування round-robin (за дефотом) для розподілу потужності між серверами [1].

Сегмент server визначає порт 80, через який Nginx сервер буде отримувати requests. Він також містить змінну proxy_pass (вказівник).

Змінна proxy_pass використовується для вказівки NGINX, куди відправляти одержуваний трафік. У цьому випадку змінна proxy_pass вказує на 3 сервери. Це дозволяє NGINX направляти одержуваний трафік на будь-яку з IP-адрес вищих серверів. Nginx одночасно виступає в ролі зворотного проксі та балансувальника навантаження.

Зворотний проксі – це сервер, який знаходиться між бекенд-серверами та перехоплює запити від клієнтів.

Наступним кроком буде визначення методу розподілу навантаження. Я знаю кілька методів, які ми можемо використати, до них відносяться:

Round Robin — це метод балансування навантаження, коли кожен сервер у кластері має можливість обробляти запити. Цей метод часто використовується на веб-серверах, де кожен запит сервера рівномірно розподіляється між серверами. Потужність розподіляється по черзі, що означає, що кожен сервер має свій час для виконання запиту. Наприклад, якщо у вас є три головні сервери, A, B і C, балансувальник навантаження спочатку розподілить навантаження на A, потім B і, нарешті, C, перш ніж знову розподілити навантаження на A. Цей спосіб досить простий. Одним з обмежень є те, що деякі сервери простоюють просто тому, що чекають своєї черги. У нашому прикладі, якщо A отримує завдання та виконує його за секунду, це означає, що він неактивний до наступного завдання. За замовчуванням Ngink використовує циклічний метод для розподілу навантаження між серверами.

Кругова система з додатковою вагою.Щоб вирішити проблему простою сервера, ми можемо використовувати ваги серверів, щоб повідомити Ngink, які сервери мають мати найвищий пріоритет. Weighted Round Robin — один із найпопулярніших методів балансування навантаження, який використовується сьогодні.

Цей метод передбачає призначення ваг кожному серверу та розподіл трафіку між серверами на основі цих ваг. Це гарантує, що сервери з більшою пропускною здатністю отримують більше трафіку, і допомагає запобігти перевантаженню сервера. Цей метод часто використовується в поєднанні з іншими методами, такими як збереження сеансу, щоб забезпечити рівномірний розподіл потужності між серверами. Сервер додатків із найвищим параметром ваги має пріоритет (більший трафік) над сервером із найменшою (вагою).

```
http {
    upstream app{
        server 10.2.0.100 weight=5;
        server 10.2.0.101 weight=3;
        server 10.2.0.102 weight=1;
    }

   server {
        listen 80;

        location / {
            proxy_pass http://app;
        }
    }
}
```

Рисунок 4.4 – Оновлення конфігурації Nginx

Least Connection - це метод котрий має найменше числа з'єднань (Least Connection) – це популярна техніка, що використовується для рівномірного розподілу робочого навантаження між кількома серверами. Метод працює шляхом маршрутизації кожного нового запиту на з'єднання – на сервер із найменшою кількістю активних з'єднань. Це гарантує, що всі сервери використовуються однаково, і жоден з них не перевантажений.

```
http {
    upstream app{
        least_conn;
        server 10.2.0.100;
        server 10.2.0.101;
        server 10.2.0.102;
    }
    server {
        listen 80;

        location / {
            proxy_pass http://app;
        }
    }
}
```

Рисунок 4.5 – Least Connection метод

Least Connection з додаванням ваги - цей метод використовується для розподілу робочого навантаження між кількома обчислювальними ресурсами (такими як сервери) з метою оптимізації продуктивності серверу та мінімізації часу відгуку серверу. Даний метод враховує кількість активних з'єднань на кожному сервері та надає відповідні ваги їх. Метою є розподіл робочого навантаження таким чином, щоб збалансувати навантаження та мінімізувати час відгуку.

```
http {
    upstream app{
        least_conn;
        server 10.2.0.100 weight=5;
        server 10.2.0.101 weight=4;
        server 10.2.0.102 weight=1;
    }
    server {
        listen 80;

        location / {
            proxy_pass http://app;
        }
    }
}
```

Рисунок 4.6 – Метод Least Connection з додаванням ваги

IP Hash - метод балансування використовує алгоритм хешування визначення того, який сервер повинен отримати кожен із вхідних пакетів. Це корисно, коли за однією IP-адресою знаходиться кілька серверів, і ми хочемо переконатися, що кожен пакет з IP-адреси клієнта направляється на той самий сервер. Цей метод бере IP-адресу джерела та IP-адресу призначення – кінцева точка і створює унікальний хеш-ключ. Потім використовується для розподілу клієнта між певними серверами.

Це важливий момент у разі розгортання стабілного енву. Це дозволяє нам, розробникам, випускати зміни для окремої частини користувачів, щоб вони могли все протестувати та надати відгуки, перш ніж надсилати їх у реліз.

Плюси цього підходу в тому, що можна забезпечити більшу продуктивність, ніж інші методи, такі як round-robin.

```
http {
    upstream app{
        ip_hash;
        server 10.2.0.100;
        server 10.2.0.101;
        server 10.2.0.102;
    }
    server {
        listen 80;

        location / {
            proxy_pass http://app;
        }
    }
}
```

Рисунок 4.7 – Метод IP Hash

Після того як я налаштував регулятор навантаження з потрібним вам методом балансування, ми можемо перезапустити Nginx, щоб зміни примінилися

4.1.1 Проксіювання HTTP-трафіку на групу серверів

Проксі-сервер — це сервер, який діє як посередник між клієнтом та іншим сервером. Проксі-сервер можна використовувати для надання клієнтам доступу до групи серверів, наприклад до групи веб-серверів, шляхом спрямування запитів від клієнта до відповідного сервера.

Проксі-сервер також може забезпечити кешування для підвищення продуктивності, зменшуючи потребу надсилати запит на сервер для кожного перегляду сторінки. Ngink може діяти як проксі або зворотний проксі. Різниця між

прямим і зворотним проксі полягає в тому, що проксі є проксі перед декількома клієнтами, а зворотний проксі – перед декількома серверними модулями. Проксі-сервер захищає особу користувача, тоді як зворотний проксі-сервер захищає інформацію на внутрішніх серверах.

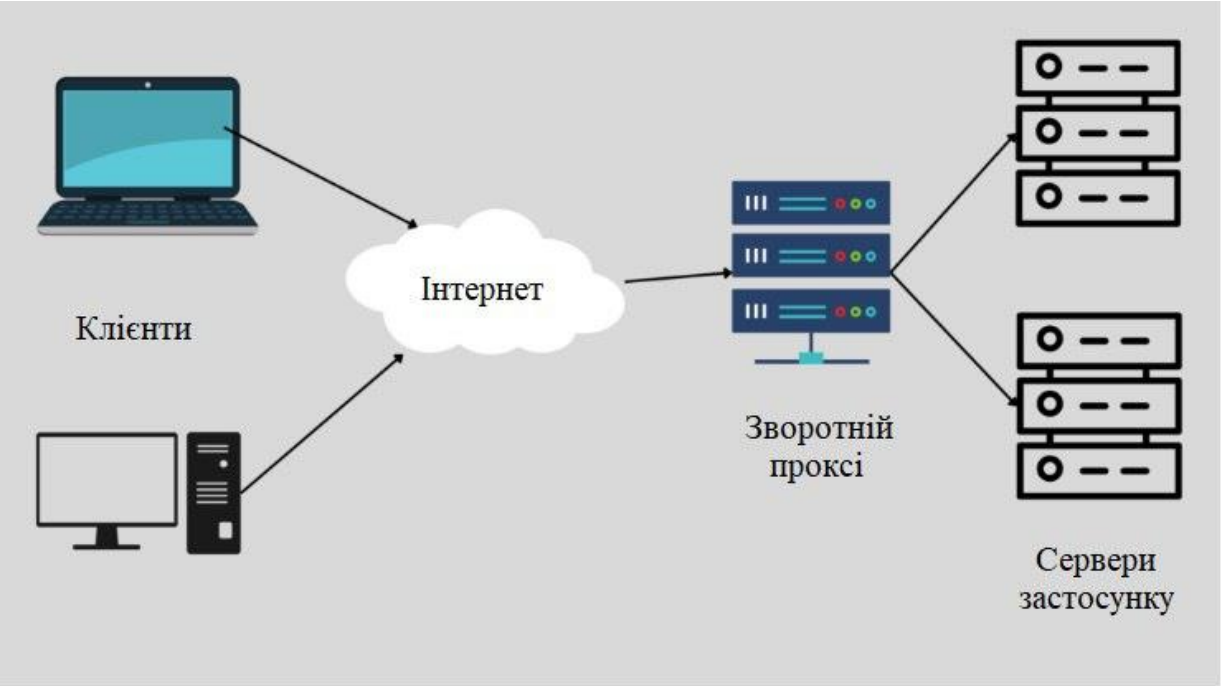


Рисунок 4.8 – Схема прямого проксі

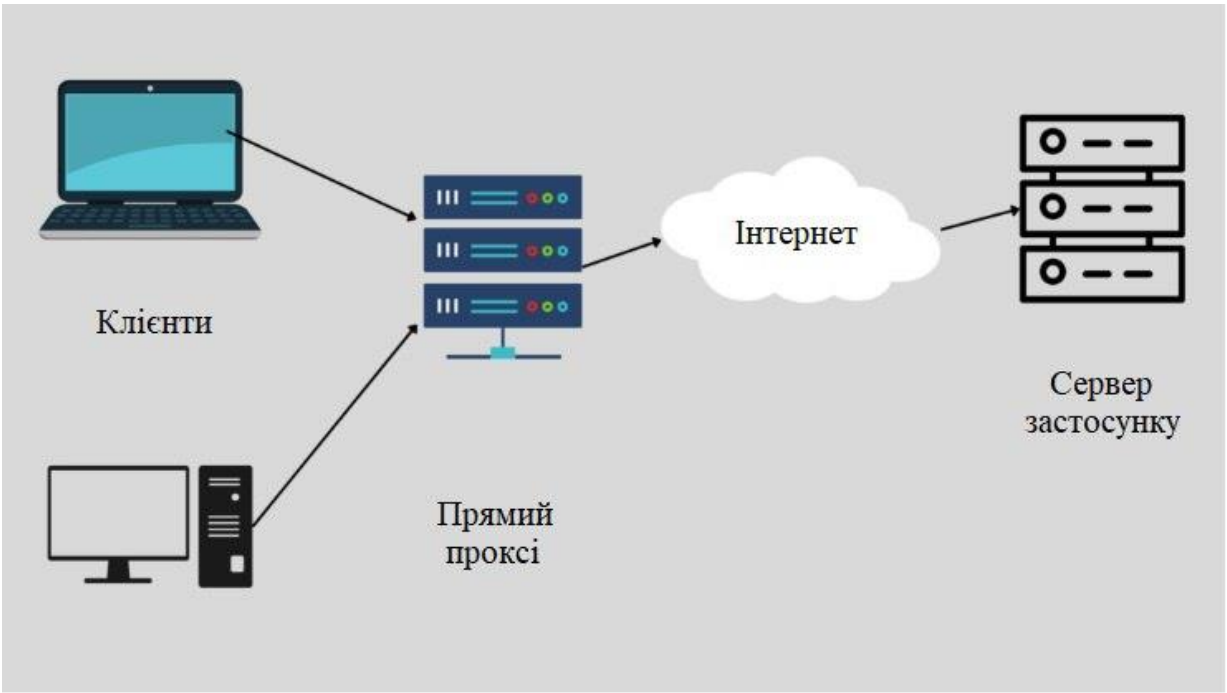


Рисунок 4.9 – Схема зворотнього проксі

4.2 Cache-Aside

Напевно я вважаю, це найрозповсюдженіший патерн кешування. Його суть відбувається в тому, що коли ми хочемо отримати дані, а їх немає в кеші, ми звертаємося до основного сховища. Далі записуємо в кеш і тоді віддаємо результат. Наступні запити вже використовують кеш замість основного сховища.

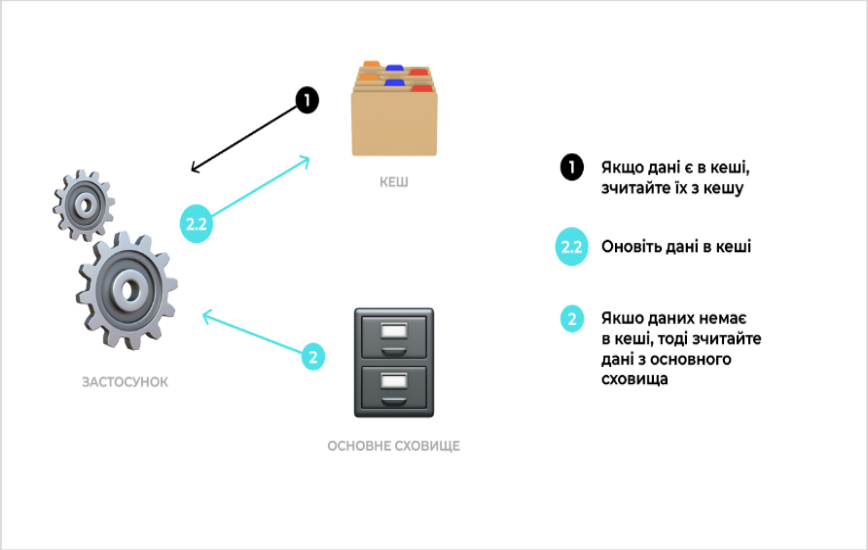


Рисунок 4.10 – Cache-Aside схема

```
import * as Memcached from 'memcached';

class Expert {
 constructor(public id: string, public name: string, public reviews: number) {}
}

class CacheAside {
 constructor(private readonly cache: Memcached) {
 }
 public async getExpert(id) {
   let expert = await this.getExpertFromCache(id);
   if (!expert) {
     //Не знайшли експерта в кеші, пробуємо дістати зі сховища
     expert = await this.getExpertFromDatabase(id);
     // Кешуємо експерта по ID
     this.cache.set(`expert_${id}`, expert, 3600, (err: any) => {
       if (err) {
         throw Error(`Error setting messages in memcached: ${err}`);
       }
     });
   }


   return expert;
 }


 private async getExpertFromCache(id: string) {
   return new Promise((resolve, reject) => {
     this.cache.get(`expert_${id}`, (err: any, data: any) => {
       if (err) {
         reject(Error(`Error setting messages in memcached: ${err}`));
       } else {
         resolve(data ? new Expert(data.id, data.name, data.reviews) : null);
       }
     });
   });
 }


 private async getExpertFromDatabase(id: string) {
   // Замість реального запиту до БД повертаємо тестові дані
   return new Expert(id, `Expert with ID ${id}`, 4);
 }
}
```

Рисунок 4.11 – Приклад реалізації

У такий спосіб можна кешувати дані експерта на строку в одну годину. Щойно час сплайн і кеш очиститься, дані з основного сховища оновлюються, його переваги:

a) просто і зрозуміле рішення;

b) конфлікт проблему інтенсивного зчитування;

c) при вилученні даних з кешу можна працювати далі без помилок;

d) реалізація з TTL дає змогу тимчасово утримувати дані в кеші, коли їм потрібно, що дозволяє зберегти пам’ять.

Недоліки цього методу:

а) Дані кешу не завжди будуть актуальними, тому його слід використовувати з обережністю і тільки в тих випадках, коли це не буде критичним фактором для системи.

4.2.1 Read-Through Cache

Підхід Read-Through схожий на підхід Cache-Aside, його основна мета полягає в тому, щоб допомогти ключовому значенню читатися часто. Різниця лише в двох речах: програма завжди використовує кеш-пам'ять. Тобто зв'язок з основною пам'яттю передається на рівень провайдера кешу, який часто є окремою бібліотекою. Це дозволяє спростити код і зосередитися лише на бізнес-логіці - кеш зберігає ту саму модель даних, що й основна пам'ять.

Рисунок 4.12 – Підхід Read-Through

4.2.2 Write-back Cache

Часто кеш асоціюється з отриманими даними, але при такому підході він готовий більш ефективно записати їх в основну пам'ять. Зворотний запис — це техніка кешування даних, при якій зміни вносяться в кеш і не зберігаються в основній пам’яті, доки не буде зроблено явний запит на збереження змін.

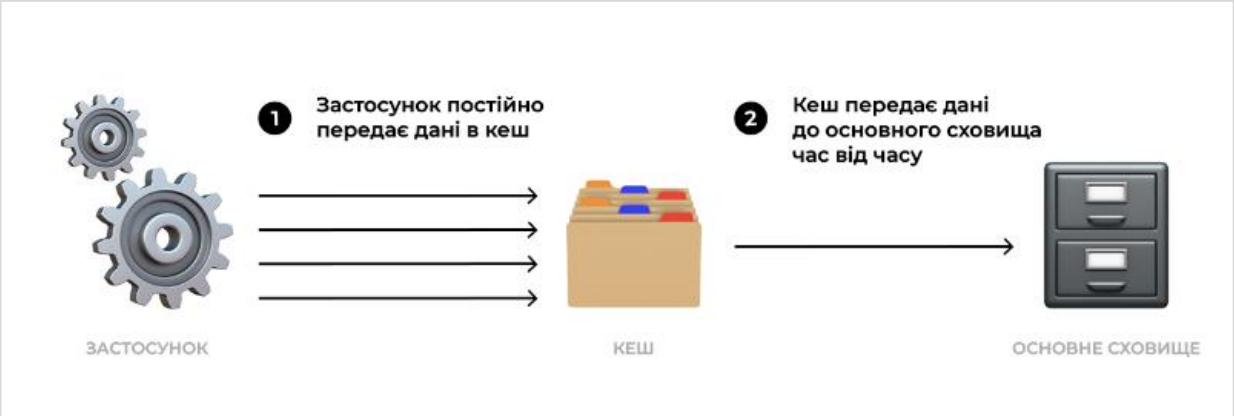


Рисунок 4.13 – Підхід Write-back Cache

Завдяки такому підходу ми завжди маємо актуальну інформацію про кеш. Ми також можемо зробити запис даних в основну пам’ять більш ефективним, особливо якщо він підтримує кілька записів.

```
export interface Message {
 id: string;
 text: string;
 isDirty?: boolean;
}


export class MessagesCache {
 constructor(private readonly cache: Memcached) {}


 public async addMessage(chatId: string, message: Message): Promise<Message> {
   message.isDirty = true;
   const messages = await this.getMessages(chatId);
   messages.push(message);
   await this.replaceInCache(chatId, messages);
   return message;
 }


 public async getMessages(chatId: string): Promise<Message[]> {
   return new Promise<Message[]>((resolve, reject) => {
     this.cache.get(chatId, (err: any, data: any) => {
       if (err) {
         reject(err);
       } else {
         resolve(data ?? []);
       }
     });
   });
 }


 public async getDirtMessages(chatId: string): Promise<Message[]> {
   const messages = await this.getMessages(chatId);
   return messages.filter((message) => message.isDirty === true);
 }


 public async flushDirtMessages(chatId: string, messageIds: string[]): Promise<void> {
   let messages = await this.getMessages(chatId);
   messages = messages.map((message) => {
     if (messageIds.includes(message.id)) {
       message.isDirty = false;
     }
     return message;
   });
   await this.replaceInCache(chatId, messages);
 }


 private async replaceInCache(chatId: string, messages: Message[]): Promise<void> {
   return new Promise<void>((resolve, reject) => {
     if (messages.length <= 1) {
       this.cache.add(chatId, messages, 0, (err) => {
         if (err) {
           reject(Error(`Error setting messages in memcached: ${err}`));
         } else {
           resolve();
         }
       });
     } else {
       this.cache.replace(chatId, messages, 0, (err) => {
         if (err) {
           reject(Error(`Error setting messages in memcached: ${err}`));
         } else {
           resolve();
         }
       });
     }
   });
 }
}
```

Рисунок 4.14 – Реалізація кешування повідомляє у чаті

Також можна реалізувати асинхронне до основного збереження.

```
setTimeout(async () => {
 const chatId = '123';
 const cache = new MessagesCache(new Memcached('localhost:11211'));
 const dirtMessages = await cache.getDirtMessages(chatId);
 // тут код, який зберігає в БД, якщо є можливість реалізувати множинний запис
 await cache.flushDirtMessages(
 chatId,
 dirtMessages.map((message) => message.id),
);
}, 60000);
```

Рисунок 4.15 – Приклад реалізації асинхронного до основного значення

4.2.3 Write-Through Cache

Модель Write-Through стосується не лише читання, але й запису даних. Процедура проста: ми записуємо дані в кеш, які відразу переносимо в основну пам'ять.

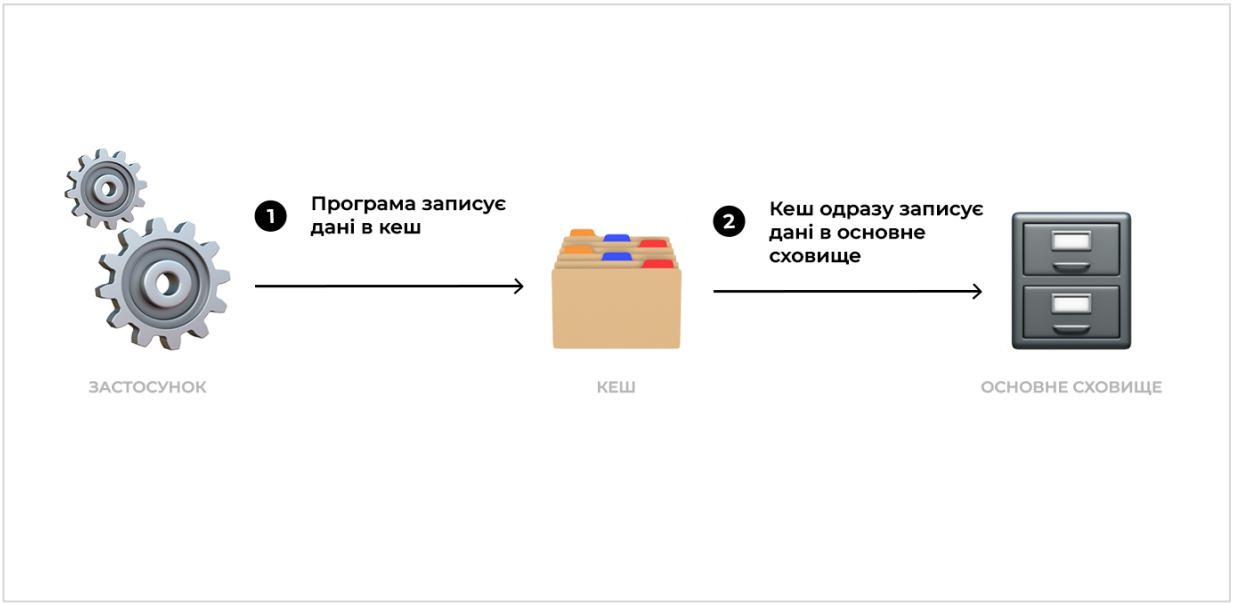


Рисунок 4.16 – Патерн Write-Through

Перевага цього підходу полягає в тому, що століття позначає відступ інформації, і ніколи не потрібно посилатися на основне представлення під час читання.

Недоліком є те, що вам доведеться виконувати дві записи одночасно: в кеш, а потім в основну пам'ять. Коли вам потрібно писати багато і шрифтом, це може бути проблемою. На додаток до такого підходу додаткових витрат, важко мати великий обсяг пам'яті.

4.3 Кешування для підвищення продуктивності

Проблеми з продуктивністю можуть виникати зі зростанням популярності. Деякими типовими прикладами є відсутність каталогів бази даних або забагато запитів SQL на сторінку. З порожньою базою у вас не буде проблем, але при збільшенні трафіку та обсягу даних вони можуть з'явитися в будь-який момент.

Використання стратегії кешування HTTP є чудовим способом досягти найкращої продуктивності для кінцевих користувачів без особливих зусиль. Додайте зворотний кеш-проксі до виробництва, щоб увімкнути кешування, і використовуйте CDN для кращої продуктивності. Кешуємо головну сторінку на годину:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -30,7 +30,7 @@ class ConferenceController extends AbstractController
     {
         return $this->render('conference/index.html.twig', [
             'conferences' => $conferenceRepository->findAll(),
-        ]);
+        ])->setSharedMaxAge(3600);
     }

     #[Route('/conference/{slug}', name: 'conference')]
```

Рисунок 4.17 – Закешована головна сторінка на 1 год

Метод setSharedMakAge() встановлює час життя кешу для зворотних проксі. Використовуйте setMakAge() для керування кеш-пам’яттю браузера. Час встановлюється в секундах (1 година = 60 хвилин = 3600 секунд).

Сторінку конференції важче кешувати, тому вона більш динамічна. Кожен може додати коментар у будь-який час, і ніхто не хоче чекати годину, щоб побачити його на сайті. У таких випадках використовуйте стратегію перевірки HTTP.

4.3.1 Активація ядра HTTP-кешу Symfony

Для стратегії кешування HTTP вимкніть зворотний HTTP-проксі Symfony, але лише в тестовому середовищі «розробки» (у «виробничому» середовищі ми використовуємо більш «надійне» рішення:

```
--- a/config/packages/framework.yaml
+++ b/config/packages/framework.yaml
@@ -23,3 +23,7 @@ when@test:
         test: true
         session:
             storage_factory_id: session.storage.factory.mock_file
+
+when@dev:
+    framework:
+        http_cache: true
```

Рисунок 4.18 – Зворотний зворотний HTTP-проксі Symfony

Крім того, що Symfony HTTP Proxy є повним зворотним HTTP-проксі, він надає корисну інформацію про якість HTTP-заголовків (через клас HttpCache). Це дуже добре для перевірки заголовків кешу, які ми встановили.

```
$ curl -s -I -X GET https://127.0.0.1:8000/
```

```
HTTP/2 200
age: 0
cache-control: public, s-maxage=3600
content-type: text/html; charset=UTF-8
date: Mon, 28 Oct 2019 08:11:57 GMT
x-content-digest: en63cef7045fe418859d73668c2703fb1324fcc0d35b21d95369a9ed1aca48e73e
x-debug-token: 9eb25a
x-debug-token-link: https://127.0.0.1:8000/_profiler/9eb25a
x-robots-tag: noindex
x-symfony-cache: GET /: miss, store
content-length: 50978
```

Рисунок 4.19 – Get request

У першому запиті кеш-сервер повідомляє, що сталася помилка, і кешує відповідь. Перевірте заголовок керування кешем, щоб побачити вказану стратегію кешу.

Відповіді на наступні запити зберігаються в кеші (час (вік) також оновлюється):

```
HTTP/2 200
age: 143
cache-control: public, s-maxage=3600
content-type: text/html; charset=UTF-8
date: Mon, 28 Oct 2019 08:11:57 GMT
x-content-digest: en63cef7045fe418859d73668c2703fb1324fcc0d35b21d95369a9ed1aca48e73e
x-debug-token: 9eb25a
x-debug-token-link: https://127.0.0.1:8000/_profiler/9eb25a
x-robots-tag: noindex
x-symfony-cache: GET /: fresh
content-length: 50978
```

Рисунок 4.20 – Запит де закешований (період (вік) також оновлено)

4.3.2 Уникнення SQL-запитів за допомогою ESI

Слухач TwigEventSubscriber вносить глобальні зміни в Twig для всіх об’єктів конференції. Це відбувається на кожній сторінці веб-сайту.

Ви не додаєте нові конференції щодня, тому код запитує ті самі дані з бази даних знову і знову.

Нам подобається кешувати назви конференцій і «слики» за допомогою кешу Symfony, але я вважаю за краще покладатися на інфраструктуру HTTP для кешування, коли це можливо.

Якщо ви хочете кешувати частину сторінки, перемістіть її за межі поточного запиту HTTP, створивши підзапит. ESI ідеально підходить для цієї мети. ESI — це спосіб вбудувати результат одного HTTP-запиту в інший. Давайте створимо контролер, який повертає лише фрагмент HTML, який відображає конференцію:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -33,6 +33,14 @@ class ConferenceController extends AbstractController
         ])->setSharedMaxAge(3600);
     }

+    #[Route('/conference_header', name: 'conference_header')]
+    public function conferenceHeader(ConferenceRepository $conferenceRepository): Res
+    {
+        return $this->render('conference/header.html.twig', [
+            'conferences' => $conferenceRepository->findAll(),
+        ]);
+    }
+
     #[Route('/conference/{slug}', name: 'conference')]
     public function show(
         Request $request,
```

Рисунок 4.21 – Контролер який повертає лише фрагмент HTML, що відображає конференції:

Тепер щоразу, коли ви переходите на сторінку в браузері, підтримуються два HTTP-запити: один для заголовка та один для головної сторінки. HTTP-виклик до

заголовка конференції наразі здійснюється в Symfony, тому жодної події HTTP не відбувається. Це також означає, що заголовки HTTP не можна кешувати.

Перетворимо виклик на "справжній" HTTP за допомогою ESI.

```
--- a/config/packages/framework.yaml
+++ b/config/packages/framework.yaml
@@ -13,7 +13,7 @@ framework:
         cookie_samesite: lax
         storage_factory_id: session.storage.factory.native

-    #esi: true
+    esi: true
     #fragments: true
     php_errors:
         log: true
```

Рисунок 4.22 – Увімкнення підтримки ESI

```
--- a/templates/base.html.twig
+++ b/templates/base.html.twig
@@ -16,7 +16,7 @@
     <body>
         <header>
             <h1><a href="{{ path('homepage') }}">Guestbook</a></h1>
-            {{ render(path('conference_header')) }}
+            {{ render_esi(path('conference_header')) }}
             <hr />
         </header>
         {% block body %}{% endblock %}
```

Рисунок 4.23 – Заміна на render_esi

Якщо Symfony виявляє зворотний проксі, який може обробляти ESI, він автоматично вмикає підтримку (якщо ні, він переходить до підзапиту синхронної обробки). Simfoni підтримує ESI через зворотний проксі-сервер, перевірте його журнали (спочатку очистіть кеш - див. Очищення нижче):

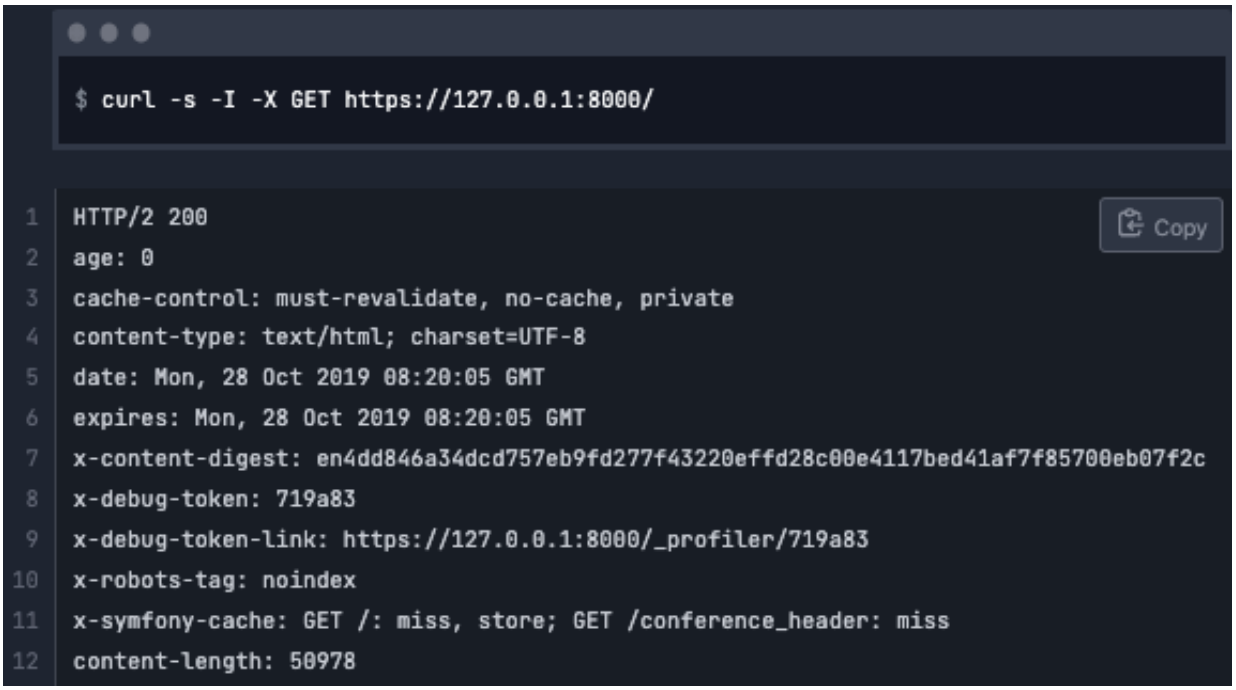


Рисунок 4.24 – Видалення кешу

Оновіть кілька разів: відповідь на /link кешується, а /conference_header – ні. Ми досягли чогось чудового: якщо ми кешуємо всю сторінку, одна частина залишається динамічною. Але це зовсім не те, чого ми хочемо. Кешувати назву сторінки протягом години незалежно від інших:

```
--- a/src/Controller/ConferenceController.php
+++ b/src/Controller/ConferenceController.php
@@ -38,7 +38,7 @@ class ConferenceController extends AbstractController
     {
         return $this->render('conference/header.html.twig', [
             'conferences' => $conferenceRepository->findAll(),
-        ]);
+        ])->setSharedMaxAge(3600);
     }

     #[Route('/conference/{slug}', name: 'conference')]
```

Рисунок 4.25 – Кешування сторінки на годину

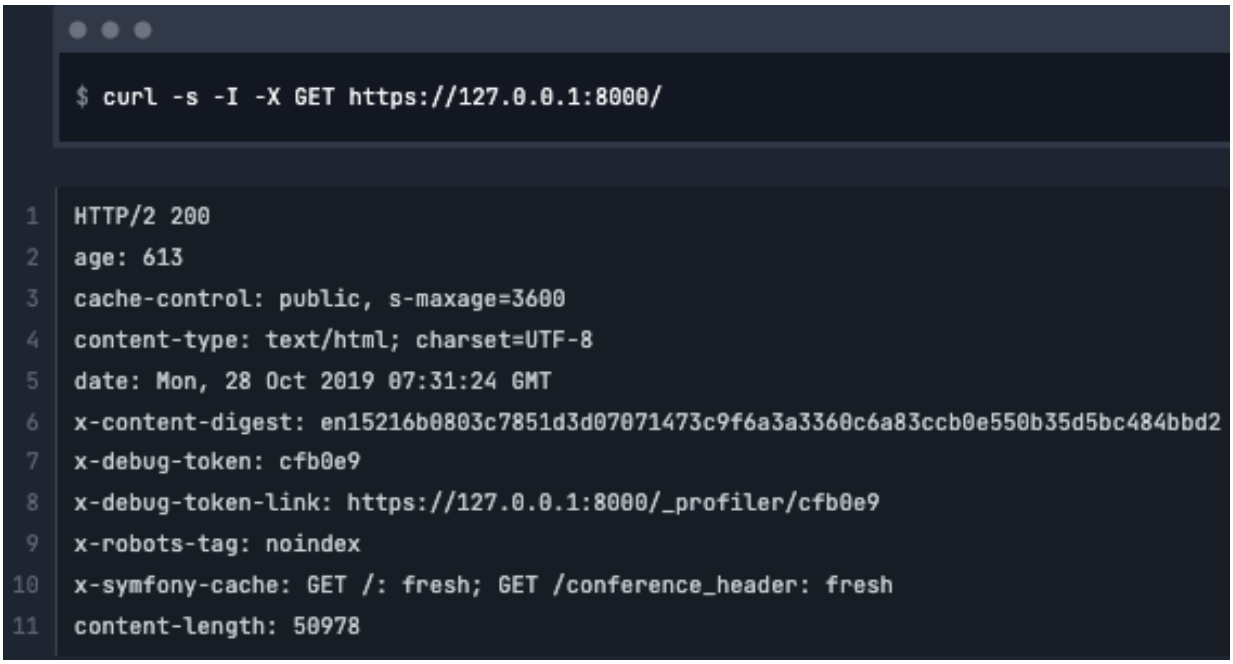


Рисунок 4.26 – Увімкненя кешу для обох запитів

Заголовок Xs-symfoni-cache містить два елементи: основний запит / і підзапит (ESI Conference_header). Обидва кешовані (свіжі). Стратегія кешування головної сторінки та її ESI можуть бути ефективнішими. Якщо у нас є сторінка «про нас», ми можемо зберігати її в кеш-пам’яті протягом тижня й оновлювати назву щогодини. Видалимо слухача, бо він нам більше не потрібен

4.3.3 Кешування інтенсивних операцій ЦП/пам’яті

На нашому веб-сайті немає алгоритмів, які інтенсивно використовують процесор або пам’ять. Я хочу поговорити про локальні кеші, я створюю команду, яка представляє крок, над яким я зараз працюю (точніше, назву тегу Git, пов’язаного з поточним комітом Git). Компонент Symfony Process, який дозволяє запускати команду та отримувати вихідні дані (стандартний вивід і виведення помилок).

src/Command/StepInfoCommand.php

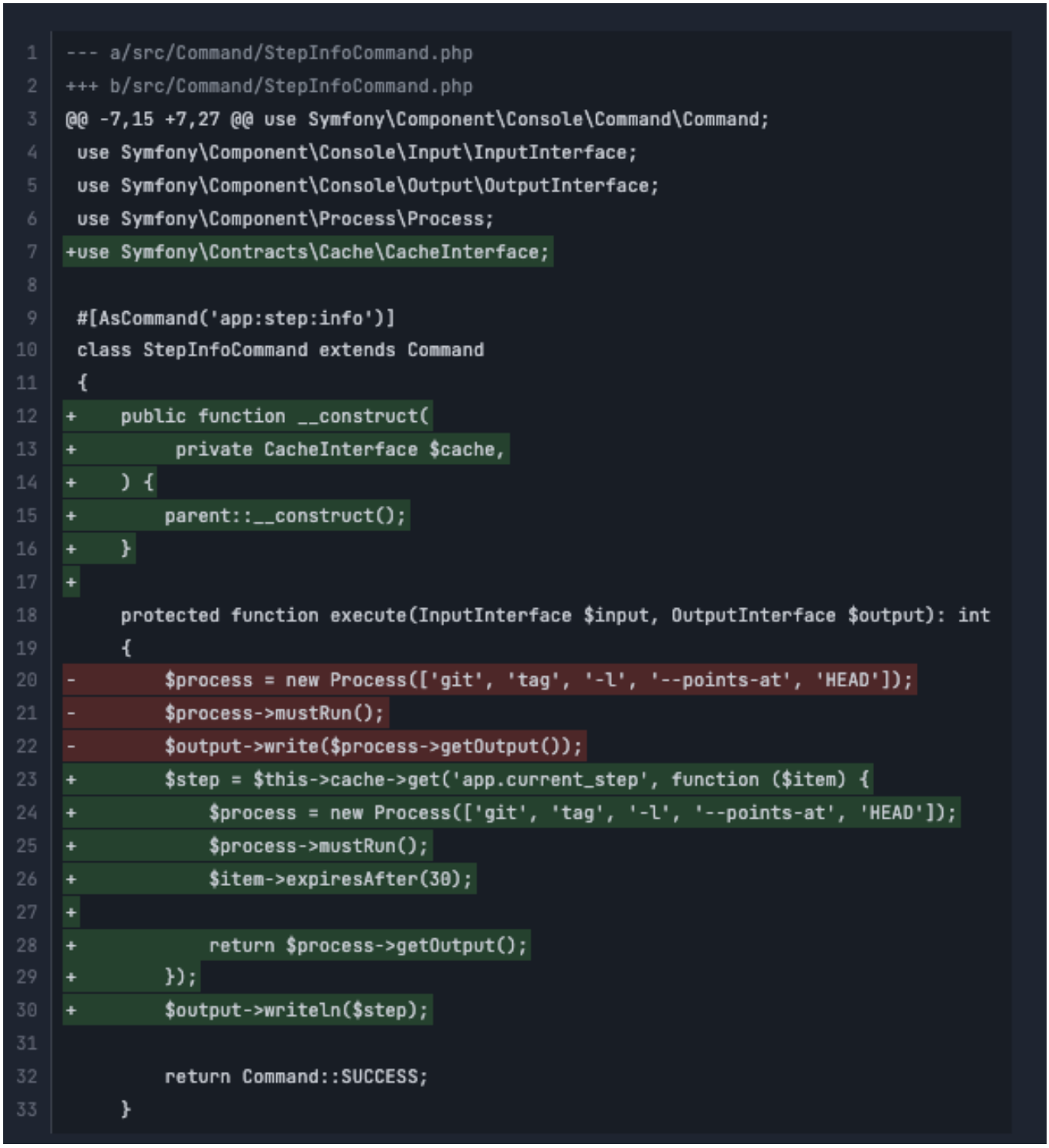
```php
namespace App\Command;

use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;
use Symfony\Component\Process\Process;

#[AsCommand('app:step:info')]
class StepInfoCommand extends Command
{
    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $process = new Process(['git', 'tag', '-l', '--points-at', 'HEAD']);
        $process->mustRun();
        $output->write($process->getOutput());

        return Command::SUCCESS;
    }
}
```

Рисунок 4.27 – Команда (стандартний вивід і вивід помилок)



```
--- a/src/Command/StepInfoCommand.php
+++ b/src/Command/StepInfoCommand.php
@@ -7,15 +7,27 @@ use Symfony\Component\Console\Command\Command;
 use Symfony\Component\Console\Input\InputInterface;
 use Symfony\Component\Console\Output\OutputInterface;
 use Symfony\Component\Process\Process;
+use Symfony\Contracts\Cache\CacheInterface;

 #[AsCommand('app:step:info')]
 class StepInfoCommand extends Command
 {
+    public function __construct(
+         private CacheInterface $cache,
+    ) {
+        parent::__construct();
+    }
+
     protected function execute(InputInterface $input, OutputInterface $output): int
     {
-        $process = new Process(['git', 'tag', '-l', '--points-at', 'HEAD']);
-        $process->mustRun();
-        $output->write($process->getOutput());
+        $step = $this->cache->get('app.current_step', function ($item) {
+            $process = new Process(['git', 'tag', '-l', '--points-at', 'HEAD']);
+            $process->mustRun();
+            $item->expiresAfter(30);
+
+            return $process->getOutput();
+        });
+        $output->writeln($step);

         return Command::SUCCESS;
     }
```

Рисунок 4.28 – Кешування виводу на декілька хвилин

4.4 Аналіз продуктивності на базі HTTP запитів

В даному пункті четвертого розділу дипломної роботи буде розглянутий

порівняльний аналіз роботи веб серверу – буде узятий для прикладу налаштований сайт котрий я раніше налаштував та залив на хостінг. Даний аналіз буде проходити в web dev tools через браузер на вкладці Perfomance.

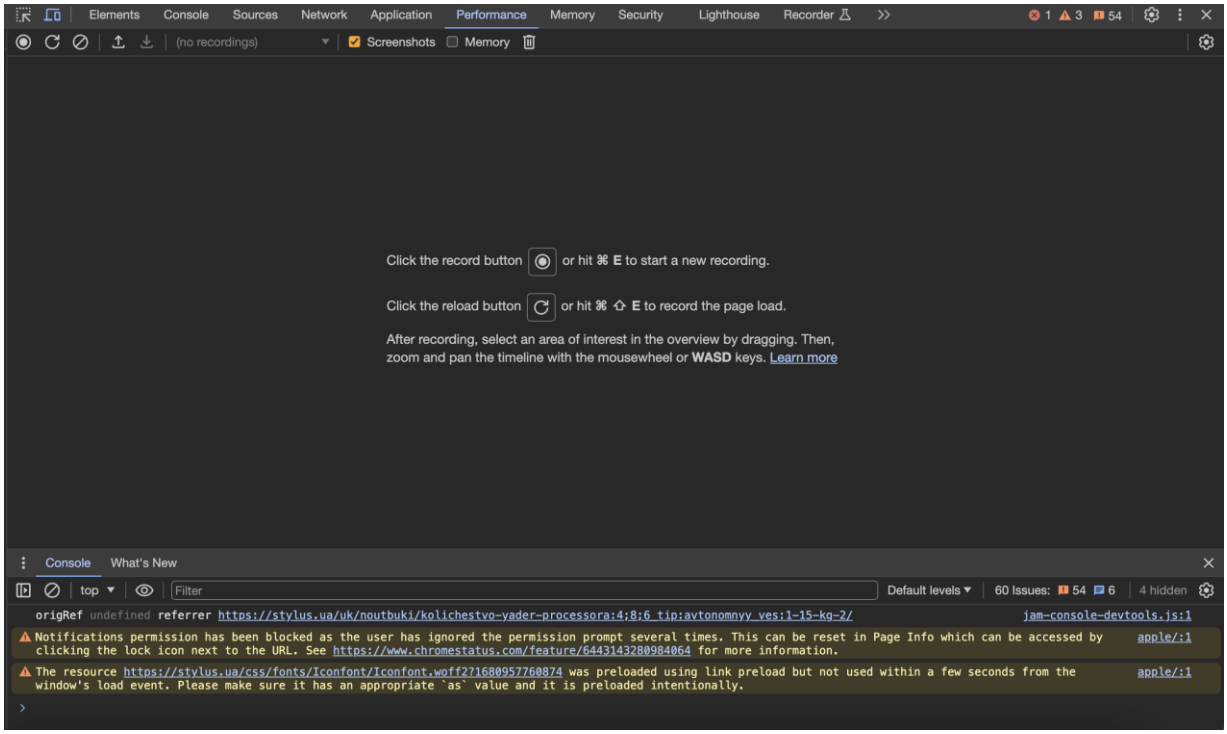


Рисунок 4.29 – Вкладка Performance у dev tools

4.4.1 Аналіз продуктивності веб сервера з увімкненим кешем

Для налаштування кешування я перейшов в адміністрування на хостингу – обрав що мені потрібно кешувати та увімкнув кешування на UI.

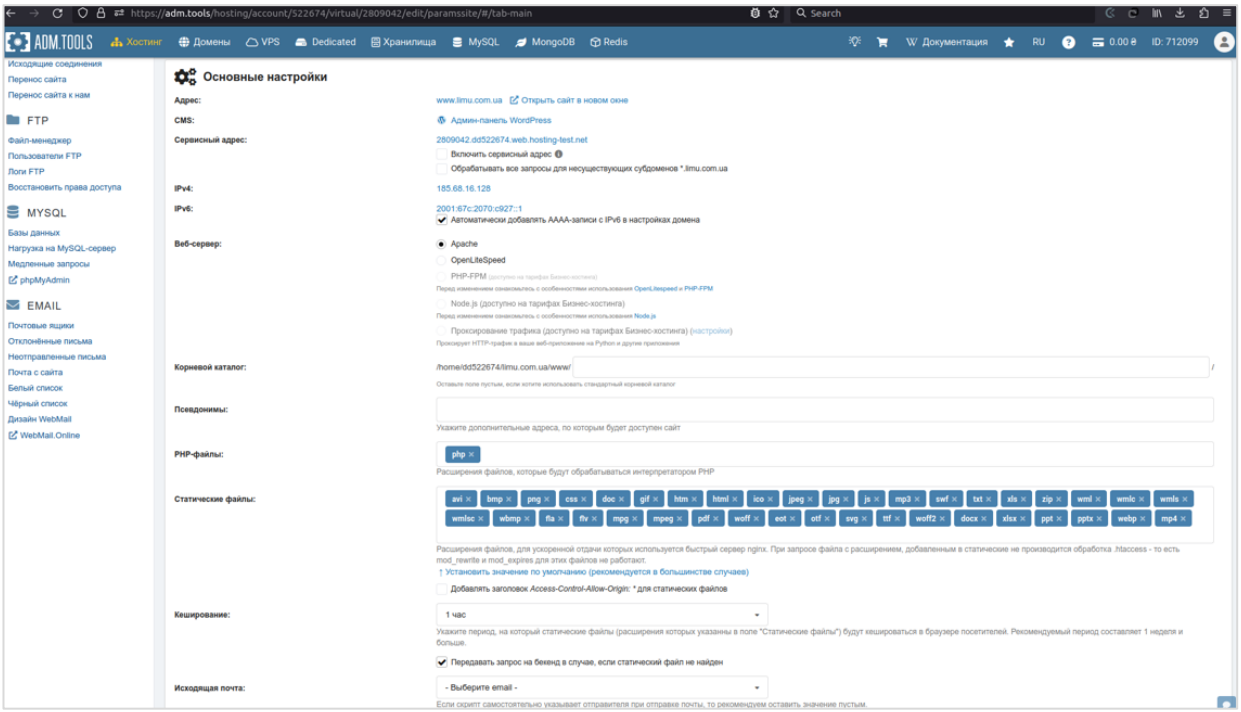


Рисунок 4.30 – Адмін панель хостінгу та увімкнення кешування

Для тестування швидкості була обрана перша головна сторінка де включений кеш на кешування статичних даних.

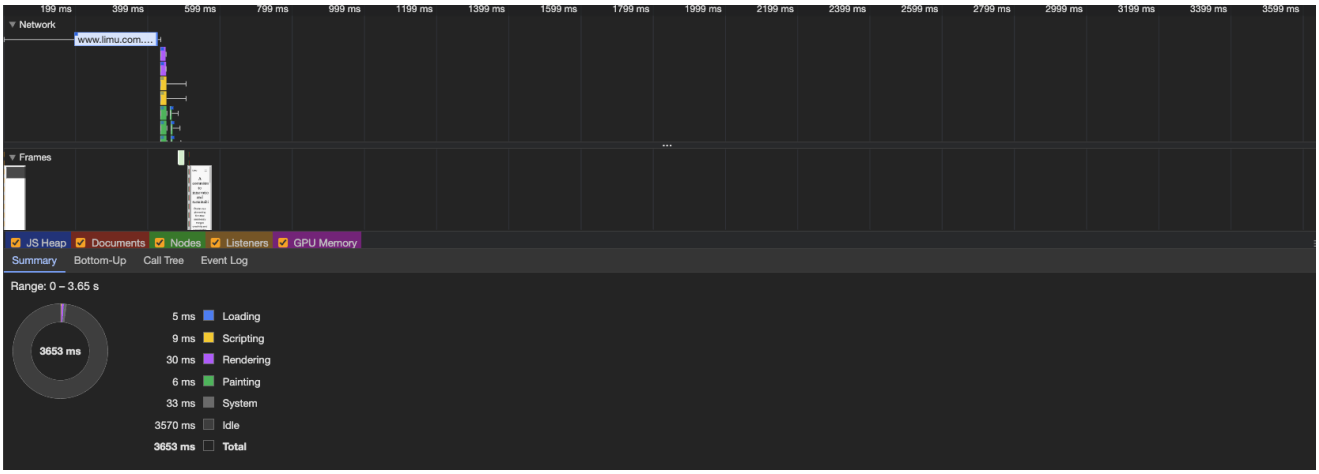


Рисунок 4.31 – Performance вкладка в Chrome сторінки з увімкнутим кешуванням

Для наступного тесту буде використаний браузер Firefox де кеш буде включений

Ми можем побачити що сторінка повністю завантажилася за 3653 ms – 3.6 s

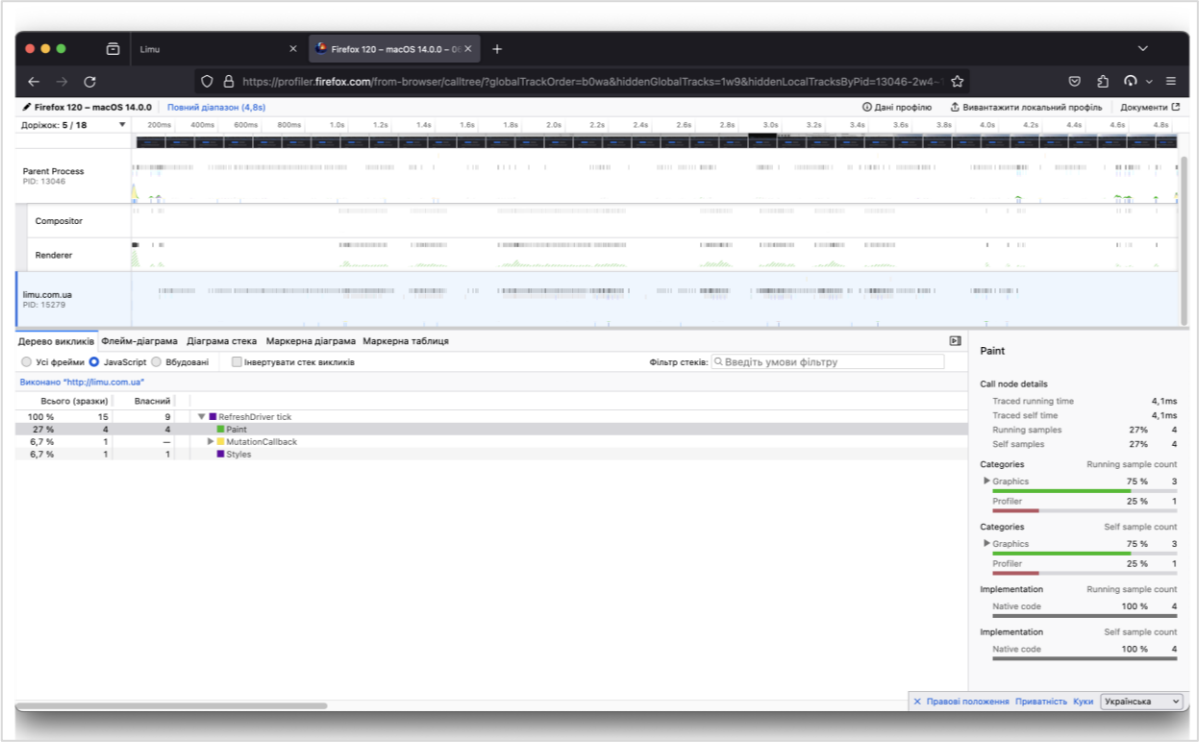


Рисунок 4.32 – Performance вкладка у Firefox сторінки з увімкнутим кешем

Ми можем побачити результат 4.8 секунд з увімкненим кешем на серверній стороні.

4.4.2 Аналіз продуктивності веб сервера без увімкненого кешування

Зараз буде аналіз сторінки без увімкнення кешування – тобто дані будуть завантажуватися при кожному відкритті сторінки і логічно що швидксіть буде не така як з кешуванням.

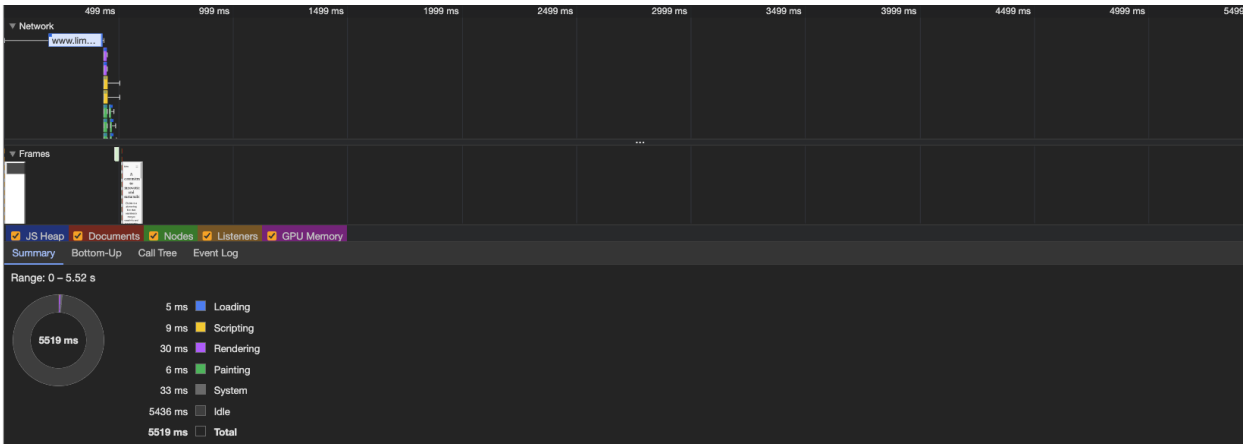


Рисунок 4.33 – Performance вкладка в Chrome сторінки без увімкнення кешування

Зараз буде аналіз сторінки без увімкнення кешування – тобто дані будуть завантажуватися при кожному відкритті сторінки і логічно що швидксіть буде не така як з кешуванням. Я бачу, що сторінка повністю завантажилася за 5519 ms – 5.6 s.

Тобто кешування на рівні вебсерверу статичних даних зменьшує час обробки запитів в нашому випадку майже в двічі а при великих системах де багато даних це рятує кліента.

Для наступного тесту буде використаний браузер Firefox де кеш буде не включено

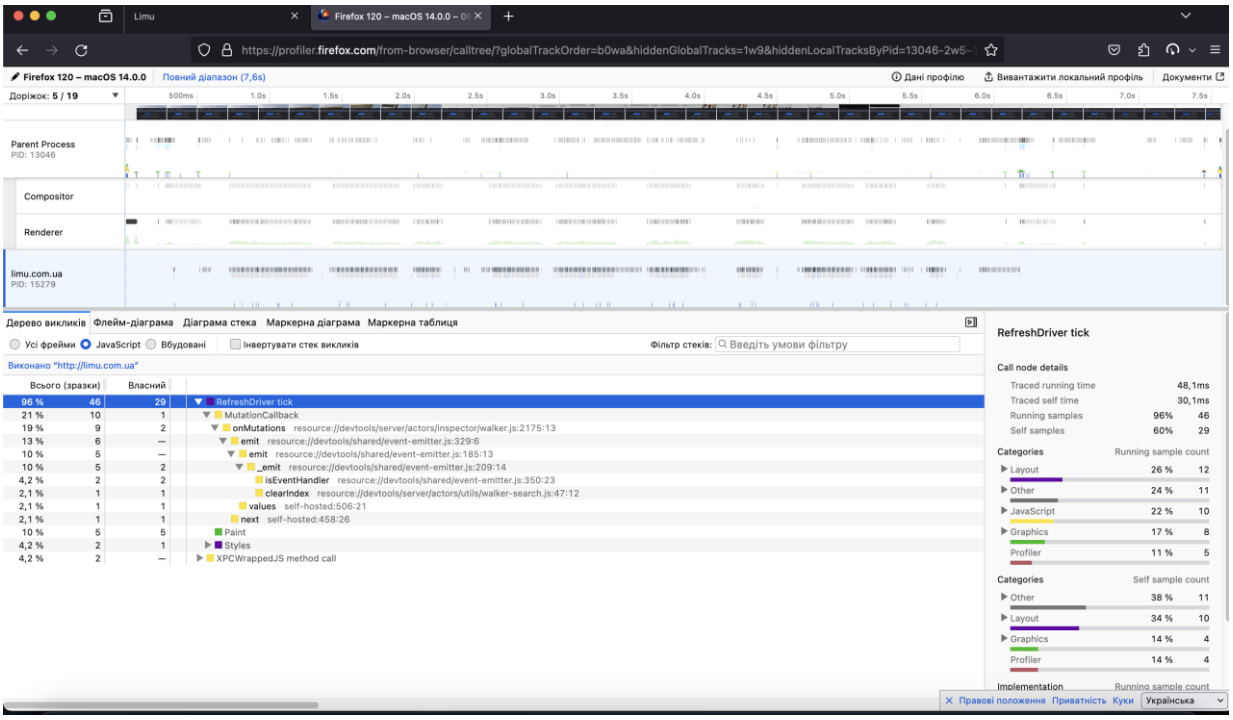


Рисунок 4.34 – Performance вкладка у Firefox сторінки без увімкнення кешу

Firefox може дати більш розгорнути відповідь ніж хром та багато плюсів в плані тестування продуктивності та звичайного тестування запитів на бек частину.

Результат я маю 7.5 секунд запуску сторінки без увімкнутого кешування на сервері.

Через консоль я зробив запит на конкретний Curl щоб перевірити час відповіді з кешем та без підключення БД на сервері для сайту. Щоб було детально зрозуміло що таке cURL то нижче буде детальна інформація. cURL — назва проекту та багатоплатформна комп'ютерна система, що працює на передачі даних через Інтернет. cURL — це утиліта, яка організовує завантаження даних із веб-сайту, тому її можна легко створити, вказавши такі параметри, як cookie, client_user, server та інші заголовки. cURL - це додаткова можливість керування файлами на стороні сервера Інтернет-сторінок для деяких розширень, які можуть бути передані в URL. За допомогою cURL можна, наприклад, створити HTML-сторінку, для якої не потрібен браузер.

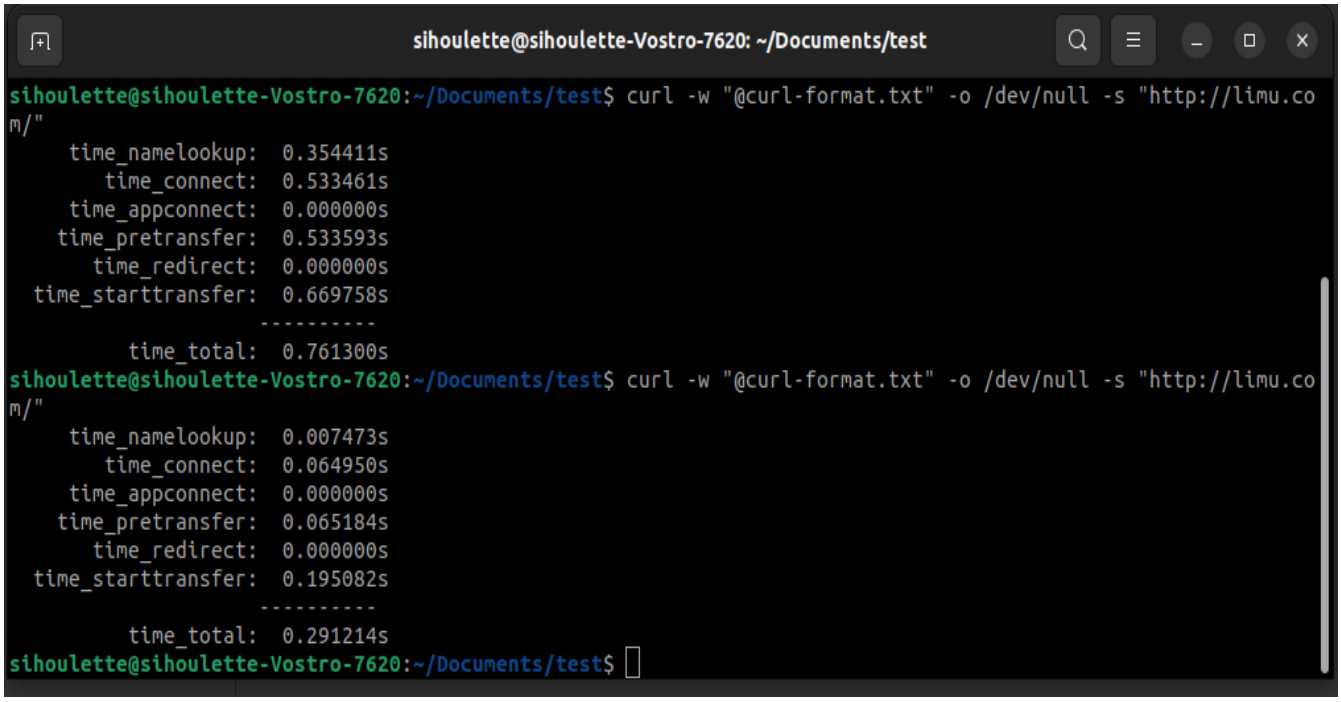


Рисунок 4.35 – Результат відповіді з кешем та без, без запитів до БД

Я бачу, що різниця з включеним та виключеним кешом майже 2 рази З вкюченим кешом час 0.76 секунди а з виключеним 0.29 секунди тобто кеш сеттево впливае на продуктивність серверу.

ВИСНОВКИ

Даний аналіз продуктивності HTTP-запитів і стратегій ефективності веб-сервера показує, що оптимізація різних аспектів відіграє ключову роль у досягненні високої продуктивності. Використання ефективних методів, таких як оптимізація коду для зміни для зменшення навантаження, забезпечення швидкого доступу до ресурсів, оптимізація ресурсів і оновлення апаратного забезпечення демонструє великий потенціал для підвищення швидкості та ефективності веб-серверів.

Цей масштабний процес підтверджує, що інтеграція підходу до оптимізації враховує кожен аспект функціонування сервера, дозволяючи досягти певного рівня продуктивності. Важливим елементом є не тільки введене програмне забезпечення, але й новий пристрій автоматичного відключення, що може призвести до швидкого кодування обробки запитів на сервер.

Через значну складність веб-серверів, де навантаження на сервер і швидкість реагування відіграють важливу роль у конкретних потребах користувачів, результати цього дослідження вимагають постійного доступу та оптимізації веб-серверів для забезпечення стабільності. і швидко працювати в онлайн-середовищі. Такий підхід є ключовим моментом у конкурсі веб-проектів і необхідності постійного інтернет-клієнта.

ПЕРЕЛІК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Technical documentation and resources from web server developers such as Apache, Nginx, Microsoft IIS, where you can find specific tips and tricks for optimizing specific servers.

2. "High Performance Browser Networking" by Ilya Gradina - this book covers the topics of profiling and optimizing network activity, which can be useful for understanding the impact of network factors on the performance of web servers 11-13.

3. "Designing Web APIs: Building APIs That Developers Love" by Brenda Rosenthal is a book that examines the process of building web APIs from a productivity and efficiency perspective 27-34.

4. "Pro Website Development and Operations: Streamlining DevOps for large-scale websites" by Matt Richardson is a book that offers an approach to website development and operations for increased efficiency and scalability.

5. "Architecting Modern Web Applications with Node.js" by Lux Wilson is a book that addresses the architectural aspects of Node.js web applications with performance and optimization in mind 49-52.

6. Technical blogs and resources from companies that specialize in web development and scalability, such as Shopify Engineering Blog, GitHub Engineering Blog, sharing their expertise in web server optimization.

7. "Performance Analysis and Tuning for General Purpose Graphics Processing Units (GPGPU)" by Haidarabed Missioner - This book looks at the use of GPUs to improve computing performance, which can be useful when optimizing some aspects of web servers 102-104.

8. "Web Performance: The Definitive Guide" by Daniel Abramov is a book that covers a wide range of topics on improving the performance of websites and web applications.

9. Building Microservices by Sam Newman - This book focuses on increasing productivity by building and scaling microservices.

10. "Web Operations: Keeping the Data On Time" by John Allsp is a book that will help you to understand the optimization and increasing the speed of web servers in connection with the operational aspects.

11. "High Performance MySQL: Optimization, Backups, and Replication" by Baron Schwartz and Peter Zeigel is a book that addresses the issue of optimizing MySQL to improve the performance of web applications 56-74.

12. Academic articles and publications in scientific journals such as "ACM Transactions on the Web" and "Journal of Web Engineering" that contain current research on improving the performance of web servers.

13. "HTTP: The Definitive Guide" by David Gelders and Brian Ting - This book provides an in-depth overview of the HTTP protocol, including its basics, request and response structure, and examines the different versions of the protocol 32-35.

14. "High-Performance Browser Networking" by Ilya Gradina is a book that, in addition to HTTP, also covers network activity and its effect on website speed 217-223.

15. "HTTP/2 in Action" by Barrett Simson and Mike McFarland - This book examines the latest technologies and improvements offered by HTTP/2 over previous versions 256-261.

16. Official HTTP documentation and specifications at the Internet Engineering Task Force (IETF) site, where you can find current standards and details about the protocol.

17. "Designing Evolvable Web APIs with ASP.NET" by Glenn Blockl - this book focuses on the creation and development of web services based on the principles of HTTP and REST architecture 140-146.

18. "HTTP Pocket Reference" is a book by Ijo Griffin that provides a compact overview of the HTTP protocol, describing the various request methods, status codes, and headers 88-89.

19. "RESTful Web APIs" by Leonard Richardson and Mike Amundsen - This book focuses on developing web services using the REST architecture and HTTP principles 34-37.

20. "HTTP/3 Explained" by Daniel Steinberg - This book covers the new version of the HTTP/3 protocol, which uses UDP instead of TCP to improve data transfer rates 63-66.

21. "Learning HTTP/2: A Practical Guide for Beginners" by Stefan Thom and Daniel Michalis is a book that looks at the main aspects of HTTP/2 and its impact on network speed 79-81.

22. "RESTful Web Services Cookbook" by Subra Ramamurthy - This book provides practical examples and recipes for developing RESTful web services, including using HTTP 101-107.

ДОДАТОК А. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ