Міністерство освіти і науки України
Тернопільський національний педагогічний університет
імені Володимира Гнатюка

Фізико-математичний факультет

Кафедра інформатики та методики її навчання

Кваліфікаційна робота

ОСОБЛИВОСТІ ВИКОРИСТАННЯ РІЗНИХ СТРАТЕГІЙ КЕШУВАННЯ ПРИ РОЗРОБЦІ БЛОГІВ НА ОСНОВІ ТЕХНОЛОГІЇ PWA

Спеціальність 122 Комп'ютерні науки
Освітня програма «Комп'ютерні науки»

Здобувача вищої освіти освітньо-кваліфікаційного рівня «магістр»
Базиволяка Максима Івановича
НАУКОВИЙ КЕРІВНИК:
доцент кафедри інформатики та методики її навчання, кандидат біологічних наук
Шмигер Галина Петрівна
РЕЦЕНЗЕНТ:
доцент кафедри комп'ютерних наук Тернопільського національного технічного університету ім. І. Пулюя, кандидат технічних наук
Дмитроца Леся Павлівна

Тернопіль – 2025

АНОТАЦІЯ

Базиволяк М. І. Особливості використання різних стратегій кешування при розробці блогів на основі технології PWA. Кваліфікаційна робота на здобуття освітнього ступеня «магістр» зі спеціальності 122 Комп'ютерні науки. ТНПУ ім. В. Гнатюка. Тернопіль, 2025. 68 с.

Кваліфікаційна робота присвячена дослідженню особливостей використання різних стратегій кешування при розробці блогових вебзастосунків на основі технології Progressive Web Apps (PWA). У роботі розглянуто теоретичні засади кешування, види кешу, принципи їх функціонування та вплив на продуктивність вебресурсів. Особливу увагу приділено аналізу ролі Service Worker, технології Workbox, серверного кешування (Redis, Varnish, Nginx), а також механізмів кешування на рівні фреймворків та CDN.

Практична частина роботи включає розробку прикладного PWA-застосунку типу блогу, конфігурацію кешування на рівні Nginx, реалізацію REST API на основі Laravel з використанням Laravel Cache, а також застосування SSR для досягнення максимальної продуктивності. Проведено експериментальні вимірювання за допомогою Lighthouse, PageSpeed Insights та Chrome DevTools для оцінки ефективності різних підходів до кешування.

Отримані результати підтверджують, що правильна інтеграція багаторівневих механізмів кешування дозволяє суттєво пришвидшити завантаження вебзастосунку, покращити індексацію в пошукових системах, зменшити ресурсні витрати сервера та підвищити загальну конкурентоспроможність бізнес-рішень на основі технології PWA.

Ключові слова: кешування, Progressive Web Apps, Service Worker, Workbox, Redis, Varnish, CDN, Nginx, Laravel Cache, продуктивність вебзастосунків.

ABSTRACT

Bazyvolyak M. I. Features of using different caching strategies when developing blogs based on PWA technology. The qualification work for obtaining a master's degree in the specialty 122 Computer Science. Ternopil Volodymyr Hnatiuk National Pedagogical University. Ternopil, 2025. 68 p.

This thesis is devoted to researching the peculiarities of using various caching strategies when developing blog web applications based on Progressive Web Apps (PWA) technology. The thesis examines the theoretical foundations of caching, types of cache, principles of their functioning, and their impact on the performance of web resources. Particular attention is paid to analysing the role of Service Worker, Workbox technology, server caching (Redis, Varnish, Nginx), as well as caching mechanisms at the framework and CDN levels.

The practical part of the work includes the development of a blog-type PWA application, caching configuration at the Nginx level, implementation of a REST API based on Laravel using Laravel Cache, and the use of SSR to achieve maximum performance. Experimental measurements were carried out using Lighthouse, PageSpeed Insights, and Chrome DevTools to evaluate the effectiveness of different approaches to caching.

The results confirm that the correct integration of multi-level caching mechanisms can significantly speed up the loading of web applications, improve search engine indexing, reduce server resource consumption, and increase the overall competitiveness of PWA-based business solutions.

Keywords: caching, Progressive Web Apps, Service Worker, Workbox, Redis, Varnish, CDN, Nginx, Laravel Cache, web application performance.

ЗМІСТ

ВСТУП

Актуальність теми дослідження. У сучасних умовах стрімкого розвитку вебтехнологій та зростання вимог користувачів до швидкодії, стабільності й доступності вебресурсів особливого значення набуває оптимізація продуктивності вебзастосунків. Збільшення обсягів інформації, кількості інтерактивних елементів і залежності від мережевого з'єднання зумовлює потребу у впровадженні ефективних механізмів кешування даних [1]. Технологія Progressive Web Apps (PWA) поєднує переваги веб- і мобільних застосунків, забезпечуючи швидке завантаження, офлайн-доступність та покращений користувацький досвід [2]. Однією з ключових складових архітектури PWA є використання Service Worker та клієнтських механізмів кешування, які дозволяють перехоплювати мережеві запити, керувати кешем ресурсів і реалізовувати різні стратегії оновлення контенту [3].

Використання Cache API, Service Worker та бібліотеки Workbox дає змогу реалізовувати такі підходи до кешування, як cache-first, network-first та stale-while-revalidate, що безпосередньо впливає на швидкість завантаження сторінок і стабільність роботи вебзастосунку [4–6]. Особливо актуальним це є для блогових платформ, які характеризуються великою кількістю повторюваних запитів і поєднанням статичного та динамічного контенту.

Окрім клієнтського кешування, важливу роль відіграють серверні механізми, зокрема Redis та Varnish Cache, які дозволяють суттєво зменшити час обробки запитів і навантаження на серверну інфраструктуру [7–9]. Поєднання клієнтських і серверних підходів формує багаторівневу модель кешування, що є ефективним рішенням для високонавантажених вебресурсів. Таким чином, дослідження стратегій кешування у PWA, їх порівняльний аналіз та практичне застосування при розробці блогових вебзастосунків є актуальним завданням сучасної вебінженерії.

Мета дослідження – теоретично обґрунтувати ефективні стратегій кешування при розробці блогів на основі технології Progressive Web Apps (PWA), дослідити бізнес-переваги впровадження кешування, розробити прикладний вебзастосунок на базі технології PWA для практичного вивчення та тестування його можливостей та функціоналу.

У даному дослідженні були поставлені такі **завдання:**

1. Теоретично обґрунтувати та проаналізувати технологічні аспекти використання технології PWA. Вивчення та оцінка ключових технічних складових Service Worker, патернів кешування та їх вплив на розробку та швидкодію вебзастосунка.

2. Провести порівняльний аналіз результатів кешування та його ігнорування, порівняння різних інструментів з іншими методами інтеграціями кешування (Redis, Apolo Cache), для визначення переваг та обмежень кожної з них.

3. Дослідити практичні аспекти розробки: інструменти, фреймворки та кращі практик для розробки PWA, включаючи методи розробки для досягнення оптимальних результатів. Вивчити користувацький досвід. Проаналізувати вплив кешування для простого користувача, включаючи швидкість завантаження, швидкість відгуку.

4. Розробити прикладний вебзастосунок на базі технології PWA для практичного вивчення та тестування його можливостей та функціоналу.

Об’єкт дослідження – процес кешування даних при розробці блогів на основі технології PWA.

Предмет дослідження – технології кешування, що реалізуються за допомогою Service Worker у Progressive Web Applications типу блогів.

Для досягнення мети та реалізації поставлених завдань було використано комплекс взаємопов’язаних методів дослідження: теоретичних методів (аналіз, порівняння, узагальнення), логіко-методологічних методів, практично-емпіричних методів (проєктування, моделювання, експериментальна перевірка рішень).

Наукова новизна роботи полягає у комплексному аналізі клієнтських і серверних стратегій кешування в PWA, а також у порівнянні їх впливу на продуктивність і користувацький досвід блогових вебзастосунків на основі сучасних вебстандартів і практичних експериментів.

Практичне значення роботи полягає у можливості використання отриманих результатів і рекомендацій під час розробки та оптимізації реальних PWA-застосунків. Запропоновані підходи до кешування можуть бути застосовані для підвищення швидкодії вебресурсів, покращення офлайн-доступності та зменшення навантаження на сервери.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ЗАСАДИ КЕШУВАННЯ У ПРОГРЕСИВНИХ ВЕБЗАСТОСУНКАХ

1.1 Поняття кешування та його роль у вебзастосунках

«Cache» або ж наділі «кеш» в сфері «обчислень» – це надшвидке сховище, облаштоване фізичним або програмним чином, яке зберігає «наперед» сформовані дані, які можуть використовуватися для майбутніх обчислень чи інших потреб. Зазвичай дані, які зберігаються в кеші, є результатом попередніх обчислень, або ж просто копією даних визначеною будь-де. Використання даних з кешу, яке зазвичай маркується операцією «cache-hit», відбувається в момент, коли необхідні дані знаходяться в межах сховища та успішно віддаються ресурсу, які їх запросив.

Виконання «cache-hit» є простим зчитуванням даних з спеціального сховища, є швидшою операцією, ніж повторне повне зчитування чи обчислення даних з повільного носія даних. Таким чином, чим більше запитів обчислюється за допомогою кешу, тим швидше працює система, в якій інтегрована практика кешування.

Щоб робота кешу оправдала витрачені на нього ресурси, тобто була економічно ефективною з точки зору комп’ютерних обчислень, завчасно збережені дані повинні зберігатися невеликими дозованими даними. Ігнорування цього правила, зазвичай, призводить до збільшення використання апаратних ресурсів і до «негнучких» кешованих даних. Однак варто зазначити, що існують випадки, коли затребувані дані можуть досягати значних розмірів. Подібні випадки зустрічаються при роботі з типом даних «blob», файлоподібний об'єкт незмінного типу.

Тим не менш, використання кешування є ефективним в багатьох областях обчислювальної техніки. Подібна практика перевірена протягом десятків років та була б відсіяна при перших проявах не «ефективності».

За своєю природою кешовані дані є тим часовими, непостійними. Це надає змогу гнучко та головне – швидко реагувати на зміни в даних. У разі довготривалих даних, які повинні існувати значний час та не змінюються, зазвичай створюються відповідні інструменти для їх «invalidate» (знищення).

Зазначимо, що в цій роботі, не будуть братися до розгляду принципи кешування на рівні процесору (CPU cache), кешування графічних процесорів (GPU cache), цифрових сигнальних процесорів (DSPs) чи подібних випадків кешування. Головним для розгляду будуть наступні види, підвиди та практики кешування:

– «In-network cache»;
– «Web Cache»;
– «Memoization»;
– «Content delivery network»;
– «Cloud storage gateway».

Тобто, вищеперелічені види, підвиди, практики кешування часто використовуються при створенні та використанні вебсторінок та вебзастосунків (PWA).

Підсумовуючи вище сказане, можемо сказати, що «кеш» – це прості дані, які завчасно обчислені та зберігаються в спеціалізованому сховищі, та використовуються при потребі. Інтеграція кешування може відбутися на будь-якому «рівні» розробки, який підвладний до розширення та редагування. Відповідно були визначені різні «рівні» кешування. Залежно від використовуваних технологій кількість та варіація «рівнів» кешування можуть різнитися, тому тут немає сталих констант для всієї індустрії комп'ютерних обчислень.

Основною задачею кешування є пришвидшення роботи кінцевої системи, де вона реалізована (у деяких випадках необхідно зменшити

навантаження на систему, якщо обчислення даних займає значний час та ресурсів системи).

1.2 Принципи розміщення кешу

Насамперед варто зазначити, що дані, які необхідно кешувати обирають розробники, які створюють вебсторінки чи програмне забезпечення. Зі сторони простого користувача існує обмежений інструментарій для роботи з кешованими даними. У випадку використання веббраузера користувачі можуть повністю їх очистити за допомогою «hard reload» (рис. 1.1).

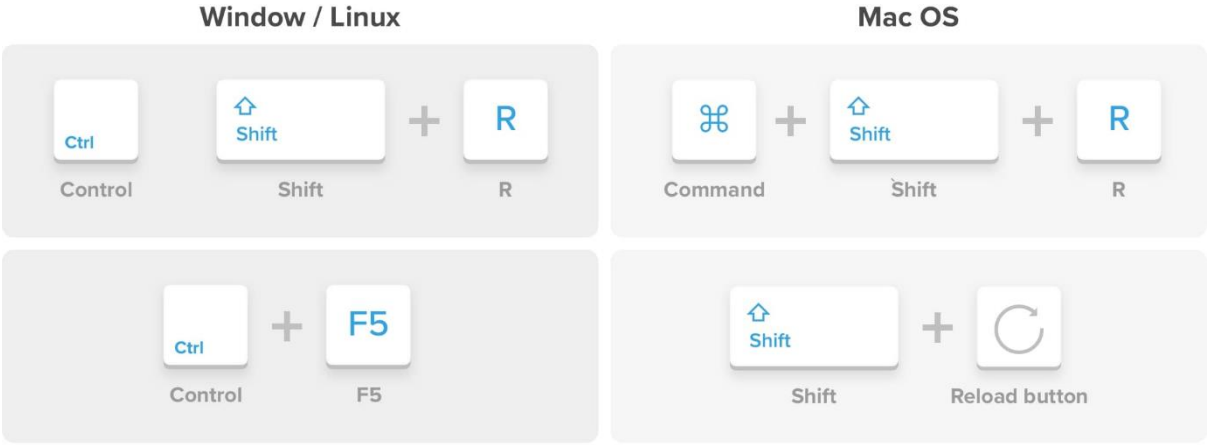

Рис. 1.1. Виконання «hard reload» для повного очищення кешів на вебсторінці

Розглядаючи питання з точки розу розробки, важливо визначити які дані буде ефективно завчасно вирахувати та зберігати для пришвидшення роботи кінцевого сервісу чи цифрового продукту. За два з половиною десятки років були апробовані сотні підходів та варіацій кешування, були створенні певні практики.

У першу чергу кешують дані, які не є динамічними. Щоб з’ясувати, які дані для вебсторінки є сталими, варто розглянути з чого складається типова вебсторінка:

– картинки (*.png, *.jpg, *.jpeg, *.gif, *.webp);

– «Doc» дані, або ж проста HTML сторінка (*.html);

– JavaScript (*.js, *.mjs);

– шрифти (*.woff, *.woff2, *.otf);

– Fetch / XHR (будь які запити за допомогою технології AJAX, fetch API чи XMLHttpRequest API);

– CSS (*.css);

– media (*.og, *.mp4, *.mp3, *.webm).

Здебільшого всі ці дані є статичними та тривалий час залишаються без змін. CSS файли, які містять візуальні правила для відображення кінцевої вебсторінки міняються у випадку редизайну цілої сторінки чи блоку. Чи картинки у вигляді логотипів вебсторінки також рідко зазнають зміни. Подібні «статичні» ресурси складають значну частину трафіку завантаження вебсторінки, тому їх при першій нагоді розробки та сам браузер стараються закешувати, щоб при наступному завантаженні вебсторінки відображення її вмісту відбулося швидше.

Однак на «противагу» статичним ресурсам є динамічні дані, які можуть змінюватися постійно, при кожному новому завантаженні сторінки. Для прикладу візьмемо вебсторінку крамниці, який відображає десятки різних пропозицій з різними зображеннями та описом. Для бізнесу критично аналізувати нові побажання користувача, відповідно постає гостра потреба відображати відповідні пропозиції, які прямо зараз цікавлять користувача. В такому випадку існує наступний варіант розвитку подій: сервер визначив, які пропозиції цікавлять покупця на даний момент та відповідно закешував всі ці пропозиції; проходить певний час (години, день) і користувач знову відвідує вебсторінку у надії знайти нові цікаві для нього пропозиції; при повторному відтворювані вебсторінки сервер помічає, що він вже має обраховані дані для цього користувача та повторно відображає попередні пропозиції; користувач бачить повторні пропозиції та просто ігнорує їх.

Подібний результат не завжди задовільний для власників вебкрамниці, тому щоб уникнути подібних ситуацій, або зменшити їх можливість до мінімуму були створені наступні інструменти та практики:

– Time to Live (надалі TTL) – тривалість життя кешованих даних, в контексті «Application-Level Cache», логіки кешу в вебзастосунку;

– заголовок Expires до HTTP протоколів – містить дату/час, після яких відповідь вважається такою, що закінчилася, в контексті HTTP кешування.

Тобто, прийнято керувати тривалістю життя кешованих даних. При використанні вищеперелічених інструментів розробник може закласти тривалість життя обрахованих пропозицій для користувача, а під час закінчення терміну «валідності» кешу він просто буде знищений. Тобто, для подібних часто змінюваних даних, або даних які повинні показувати найновіші дані, які тільки є, найкращим варіантом є використання мінімальних термінів для кешування (15–60 хв), або ж повністю відмовитися їх кешувати.

Які частини вебсторінки зазвичай не кешуються, або отримують найменший час життя кешів? Це кошик користувача, сторінка користувача, інформаційні графіки та дошки (dashboards). Перелічені компоненти є критичними для негайного відображення даних.

Важливо також зазначити, що розробники не завжди власноруч обирають тривалість життя кеш файлів, зазвичай це вирішує замовник вебсторінки. Тому нерідко можна зустріти випадки, коли закешовані критичні дані живуть понад десятки або й сотні годин. Розробники надають консультацію в цьому питанні та створюють належний функціонал для керування тривалістю життя кешів.

Залежно від виду (рівня) кеш може зберігатися на різних носіях – як на девайсі користувача, так і на програмному забезпеченні сервера вебсторінки. Наприклад, браузерний кеш зберігається локально на девайсі користувача. Браузер «Google Chrome» зберігає всі кешовані дані за наступними шляхами:

– Windows: C:\Users\<user>\AppData\Local\Google\Chrome\UserData\Default\Cache;

– Linux: ~/.cache/google-chrome/Default/Cache;

– macOS: ~/Library/Caches/Google/Chrome/Default/Cache.

Ось приклад кешованих даних браузера Google Chrome (Version 135.0.7049.52 (Official Build) (64-bit)) на операційній системі Linux (Ubuntu 22.04.5 LTS):

```
~/.cache/google-chrome/Default/Cache/Cache_Data$ ls
000ea43d7aa786dd_0  464c53aff3b42111_0  25a2559e623c4ab_0
021b94f887ea5be4_0  432378868fbb900d_0  84a657f6d34c50bf_0
```

1.3. Різновиди кешу

HTTP кеш. Протокол запиту чи відповіді передачі гіпертексту, або ж просто HTTP містить декілька різновидів кешування. Робота HTTP-кешу полягає в збережені відповіді, пов'язаної із запитом, та повторно використати збережену відповідь при потребі, для наступних запитів.

Повторне використання має кілька переваг. По-перше, оскільки немає потреби доставляти запит на вихідний сервер, то чим ближче клієнт і кеш, тим швидшою буде відповідь. Найтиповішим прикладом є те, коли сам браузер зберігає кеш для запитів браузера (рис. 1.2).

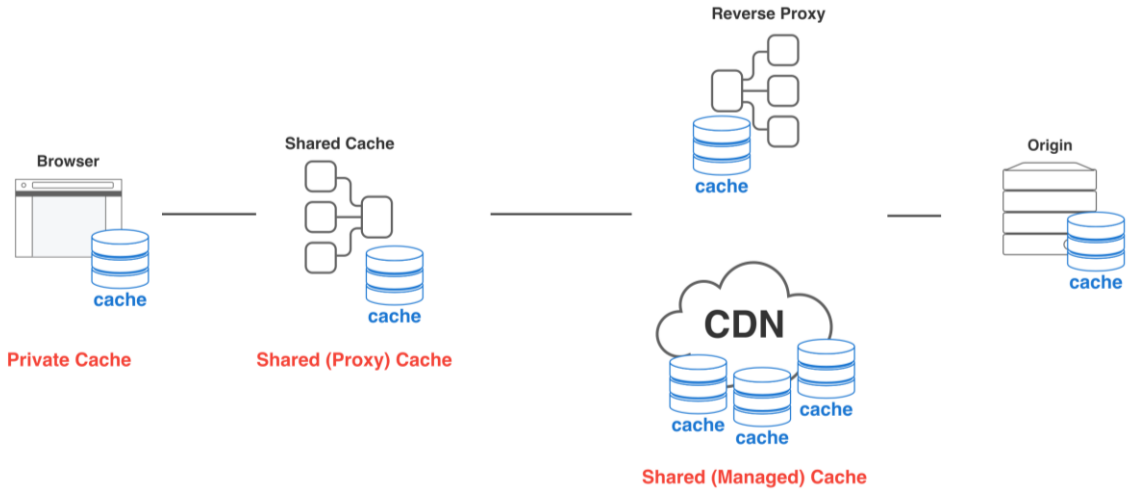

Рис 1.2. Типи HTTP кешу

Крім того, коли відповідь можна використовувати повторно, вихідному серверу не потрібно обробляти запит, тому йому не потрібно аналізувати та маршрутизувати запит, відновлювати сеанс на основі cookie, запитувати результати до бази даних або рендерити механізм шаблонів. Це зменшує навантаження на сервер.

Приватний кеш. Приватний кеш – це кеш, пов’язаний з певним клієнтом, зазвичай це кеш браузера. Оскільки збережена відповідь не передається іншим клієнтам, приватний кеш може зберігати персоналізовану відповідь для цього користувача. З іншого боку, якщо персоналізований вміст зберігається в кеші, відмінному від приватного, інші користувачі можуть отримати цей вміст, що може призвести до ненавмисного витоку інформації.

Персоналізований контент зазвичай контролюється файлами cookie, але наявність файлу cookie не завжди означає, що він є приватним, і тому сам по собі файл cookie не робить відповідь приватною. Такий кеш позначається наступним заголовком в запиті: `Cache-Control: private`

Спільний кеш. Спільний кеш розташований між клієнтом і сервером і може зберігати відповіді, якими можна обмінюватися між користувачами. Спільні кеші можна додатково класифікувати на проксі-кеші та керовані кеші.

Proxy кеш. Окрім функції контролю доступу, деякі проксі-сервери реалізують кешування для зменшення трафіку з мережі. Зазвичай це не контролюється розробником сервісу, тому воно має контролюватися відповідними HTTP-заголовками тощо.

Керований кеш. Керовані кеші розгортаються розробниками сервісів для розвантаження вихідного сервера та ефективної доставки контенту. Прикладами є зворотні проксі-сервери, CDN [12] та сервісні працівники в поєднанні з Cache API.

Характеристики керованих кешів залежать від розгорнутого середовища. У більшості випадків розробник може керувати поведінкою кешу через заголовок Cache-Control та власні файли конфігурації або панелі інструментів.

Наприклад, специфікація HTTP Caching не визначає спосіб явного видалення кешу, але за допомогою керованого кешу збережену відповідь можна видалити будь-коли за допомогою операцій панелі інструментів,

викликів API, перезапусків тощо [13]. Це дозволяє використовувати більш проактивну стратегію кешування.

Також можна ігнорувати стандартні протоколи специфікації HTTP Caching на користь явного маніпулювання [13]. Наприклад, можна вказати наступне, щоб відмовитися від приватного кешу або проксі-кешу, використовуючи власну стратегію кешування лише в керованому кеші.

Наприклад, Varnish Cache використовує логіку VCL (Varnish Configuration Language, тип DSL) для обробки сховища кешу, тоді як сервісні працівники в поєднанні з Cache API дозволяють створювати цю логіку в JavaScript [14]. Це означає, що якщо керований кеш навмисно ігнорує директиву no-store, немає потреби сприймати його як «невідповідний» стандарту. Вам слід уникати використання заголовків kitchen-sink, а уважно прочитати документацію будь-якого механізму керованого кешу, який ви використовуєте, і переконатися, що ви належним чином керуєте кешем способами, передбаченими обраним вами механізмом.

Важливо звернути увагу, що деякі CDN надають власні заголовки, які ефективні лише для обраної CDN (наприклад, Surrogate-Control) [12]. Наразі триває робота над визначенням заголовка CDN-Cache-Control для їх стандартизації.

Серверний кеш. Серверний кеш є ключовим механізмом оптимізації продуктивності у сучасних високонавантажених системах, оскільки забезпечує зберігання попередньо обчислених або раніше отриманих даних безпосередньо на сервері застосунку чи у проміжному рівні обробки. Цей тип кешування зазвичай реалізується у вигляді in-memory структур (наприклад, через Redis [15], Memcached або локальні механізми кешу фреймворку), що забезпечують доступ до даних з мінімальною латентністю. Завдяки значно швидшому часу читання, порівняно з базою даних або зовнішніми API, серверний кеш дозволяє зменшити кількість повторюваних обчислень, оптимізувати використання CPU та мінімізувати кількість дискових операцій.

Ефективність серверного кешу зростає у сценаріях, де система виконує однотипні запити або обробляє дані, що рідко змінюються. Використання TTL (time-to-live), контроль узгодженості та інвалідації дозволяє управляти актуальністю інформації, що усуває ризик застарілих відповідей. У складних архітектурах серверний кеш часто поєднується з багаторівневими політиками кешування, що забезпечує адаптивність до змінних навантажень. Застосування цього механізму суттєво підвищує пропускну здатність системи, скорочує час відповіді та сприяє стабільній роботі навіть під дією пікових навантажень, що робить серверний кеш одним із фундаментальних інструментів масштабованих серверних архітектур.

Reverse-proxy кешування. Reverse-proxy кешування є високоефективною технологією оптимізації веб-сервісів, яка передбачає зберігання кешованих копій відповідей на рівні проміжного сервера – наприклад, Nginx, Varnish [14], HAProxy або Cloudflare [16]. Такий підхід дозволяє перехоплювати запити користувачів і повертати вже готову відповідь без необхідності звернення до бекенд-застосунку або бази даних. Це значно скорочує час відгуку, зменшує навантаження на сервери обробки та забезпечує можливість обслуговувати на порядок більше одночасних клієнтів.

Ефективність reverse-proxy кешування особливо помітна у випадках статичного або умовно статичного контенту: HTML-сторінок, API-результатів, медіа-файлів, даних із високим співвідношенням читання до запису. Крім того, багато reverse-proxy систем підтримують складні механізми інвалідації, варіацію контенту за заголовками (Vary), а також edge-кешування, що дозволяє перенести дані географічно ближче до кінцевого користувача.

Застосування reverse-proxy кешу дозволяє суттєво покращити стійкість системи до пікових навантажень, атак типу DDoS та збоїв бекенда. У масштабованих мікросервісних архітектурах цей механізм є одним з основних для забезпечення високої доступності та прогнозованого часу

відповіді, що робить його критично важливим елементом оптимізації сучасних веб-інфраструктур.

Application-level кешування. Application-level кешування – це механізм керування даними на рівні бізнес-логіки застосунку, який дозволяє зберігати попередньо обчислені результати, складні структури даних або частини DOM/шаблонів для повторного використання. На відміну від серверного чи мережевого кешу, цей тип кешування тісно інтегрований із внутрішніми процесами застосунку, завдяки чому він може враховувати доменні правила, контекст виконання та специфіку запитів. Як правило, application-level кеш реалізується через локальні in-memory сховища, Redis [15], APCu, або вбудовані засоби фреймворків (наприклад, Laravel Cache [17; 18], Symfony Cache Component).

Головною перевагою цього підходу є можливість стратегічного кешування саме тих елементів, які є найбільш ресурсозатратними: результатів складних SQL-запитів, зовнішніх API-викликів, підготовлених об’єктів, шаблонів, агрегованих структур чи проміжних обчислень. Завдяки цьому застосунок значно зменшує кількість повторних операцій, знижує затрати на CPU та мережеву взаємодію і тим самим забезпечує підвищення швидкодії.

Крім того, application-level кешування дозволяє будувати адаптивні стратегії інвалідації, враховуючи оновлення даних, ролі користувачів, різні контексти запитів або складні правила TTL. У результаті цей механізм забезпечує високу ефективність, гнучкість та контрольований баланс між актуальністю даних і продуктивністю системи.

Кеш на рівні бази даних. База даних є критично важливим компонентом продуктивності програми. Різниця між добре продуктивною та погано продуктивною базою даних може бути найбільш впливовим фактором на загальну продуктивність програми. Проблеми з базою даних, такі як швидкість обробки запитів, вартість масштабування та легкість доступу до

даних, ускладнюють пошук оптимізованого балансу. Важко врахувати всі три фактори, серед багатьох інших.

Кешування бази даних – це метод буферизації, який зберігає часто запитувані дані у тимчасовій пам'яті. Кеш сховища призводить до того, що майбутні запити на ці дані обробляються швидше, ніж це можливо за допомогою доступу до основної бази даних [19].

Стратегія кешування бази даних допомагає вашій основній базі даних, полегшуючи навантаження, яке вона може нести. Найчастіше це проявляється в перенаправленні запитів часто зчитуваних даних до самого кешу, а не до основної бази даних. Сам кеш знаходиться або в базі даних, або в застосунку, або навіть як окремий сервіс (Redis). Наприклад, якщо ваша програма вперше запитує інформацію про користувача з бази даних, цей запит переходить від сервера програм до сервера бази даних і повертає запитувану інформацію. Завдяки кешу дані зберігається ближче до ресурсу запитувача після початкового зчитування, і відбувається значне скорочення часу обробки запитів та робочого навантаження бази даних для всіх наступних запитів на зчитування цих даних. Ось перелік п’яти найпопулярніших стратегій кешування бази даних [20]:

– cache-aside;

– read-through;

– write-through;

– write-back;

– write-around.

Особливості DataBase Flow Tables та DataBase Indexes при роботі з кешуванням. Flow tables і індекси не є кешем, оскільки вони не дублюють дані, а структурують їх або оптимізують спосіб доступу. Втім, з погляду прискорення запитів вони виконують функціонально схожу роль: скорочують час отримання результатів і знижують навантаження на обчислювальні ресурси. Тому в аналітичній або архітектурній документації

можна описати їх як «механізми внутрішньої оптимізації, що частково виконують роль кешування».

Flow tables у контексті систем управління даними передбачають формалізоване подання потоків обробки інформації, коли проміжні результати або маршрутизовані вибірки зберігаються у структурованому вигляді для подальшого повторного використання.

Попри те, що ці таблиці не виконують повноцінного кешування, вони забезпечують подібний ефект: зменшують кількість повторних операцій доступу або перетворень даних. Завдяки цьому знижується навантаження на базу даних, особливо у випадках, коли складні аналітичні або агрегаційні операції виконуються багаторазово. Формалізація потоку через flow tables дозволяє СУБД ефективно оптимізувати план виконання запитів, оскільки структура таких таблиць заздалегідь адаптована до очікуваних патернів роботи.

У результаті забезпечується істотний приріст продуктивності, що наближений до використання механізмів кешування.

Індекси баз даних являють собою спеціалізовані структури даних (найчастіше B-tree або hash-based), що забезпечують прискорений доступ до записів шляхом мінімізації кількості операцій читання.

Хоча індекс не є кешем у прямому сенсі, він виконує аналогічну функцію скорочення часу отримання інформації. За рахунок оптимізованої організації даних індекси істотно знижують обсяг сканування таблиць та навантаження на дискову підсистему, що особливо важливо в системах з високою інтенсивністю читання. Їх застосування можна розглядати як форму структурного кешування, оскільки СУБД фактично підтримує додатковий шар даних, який дає змогу швидко знаходити результат без повного проходження основної таблиці. Використання індексів є одним із ключових методів підвищення продуктивності реляційних СУБД.

Memoization. Також програмуванні існують загальні практики, підходи чи ідіоми, які допомагають зберегти тяжко обчислювальні операції прямо під час виконання методу, функції чи сутності.

Мемоізація – це простий, але потужний трюк, який може допомогти пришвидшити наш код, особливо при роботі з повторюваними та важкими обчислювальними функціями [21].

Реалізації на різних мовах програмування можуть відрізняться через архітектурні особливості кожної мови програмування, але концепція завжди залишається одинаковою.

Наприклад для роботи, ми використаємо класичний приклад послідовності Фібоначчі. Послідовність Фібоначчі – це набір чисел, який починається з одиниці або нуля, за якими йде одиниця, і продовжується на основі правила, що кожне число (яке називається числом Фібоначчі) дорівнює сумі двох попередніх чисел.

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...
```

Припустимо, нам потрібно написати функцію, яка повертає n-й елемент послідовності Фібоначчі. Знаючи, що кожен елемент є сумою двох попередніх, рекурсивне рішення може бути таким:

```
function fibonacci(num) {
  if (num <= 1) {
    return 1;
  }
  return fibonacci(num - 1) + fibonacci(num - 2);
}
```

Рекурсія – це підхід, за якого функція викликає саму себе, доки не буде досягнуто граничної умови, що запобігає нескінченному циклу. У наведеному прикладі такою умовою є перевірка $n \leq 1$.

Якщо викликати функцію, наприклад `fib(5)`, фактичний процес її виконання включатиме послідовне породження численних рекурсивних викликів. У цьому випадку одні й ті самі проміжні значення, зокрема `fib(0)`, `fib(1)`, `fib(2)` та `fib(3)`, обчислюються багаторазово. Така надмірність є типовою проблемою рекурсивної реалізації.

Мемоізація усуває необхідність повторного обчислення однакових проміжних результатів. Сутність методу полягає в збереженні вже отриманих значень та подальшому поверненні їх з кешу під час наступних звернень. Після застосування мемоізації функція набуває оптимізованого вигляду, у якому кожне значення рекурсії обчислюється лише один раз.

Реалізуючи мемоізацію, наша функція виглядатиме так:

```
function fibonacci(n, stored = {}) {
  if (stored[n]) {
    return stored[n];
  }

  if (n <= 1) {
    return 1;
  }

  const result = fibonacci(n - 1, stored) + fibonacci(n - 2, stored);
  stored[n] = result;
  return result;
}
```

На початковому етапі функція перевіряє, чи передано їй попередньо створений об’єкт для зберігання проміжних результатів. Якщо такого об’єкта немає, він ініціалізується як порожній: `stored = stored || {}`

Далі виконується перевірка: якщо у структурі memo уже міститься значення для переданого параметра n, воно негайно повертається. На цьому етапі мемоізація реалізовує своє основне призначення – повторний рекурсивний виклик стає непотрібним, оскільки результат уже доступний: `if (stored [n]) { return stored [n]; }`. Якщо ж значення для n ще не

збережене, функція продовжує рекурсивне обчислення, передаючи той самий об’єкт memo у подальші виклики. Завдяки цьому всі рекурсивні виклики спільно використовують єдине сховище обчислених результатів. Після отримання підсумкового значення воно записується у кеш і повертається: `return stored[n] = fib(n - 1, stored) + fib(n-2, stored).`

Таким чином, усього кілька додаткових рядків коду дозволяють інтегрувати мемоізацію в рекурсивний алгоритм і суттєво підвищити його ефективність.

1.4 Використання cookies у контексті кешування та безпеки даних вебзастосунків

Cookies-файли – це невеликі фрагменти даних, що зберігаються у браузері користувача та передаються на сервер із кожним HTTP-запитом до відповідного домену. Їх основною метою є підтримка стану, збереження налаштувань користувача, ідентифікація сесії та персоналізація інтерфейсу. На відміну від серверного або мережевого кешу, cookies не призначені для оптимізації продуктивності, хоча можуть сприяти зменшенню кількості повторних налаштувань чи конфігурацій з боку користувача. Заміна кешу cookies-файлами можлива лише у вузьких, строго визначених випадках – наприклад, коли необхідно зберігати невеликі дані конфігурації, індивідуальні UI-параметри або попередньо обрані користувачем опції (мова, тема, фільтри). Це не має прямого впливу на продуктивність застосунку, але скорочує кількість зайвих звернень щодо налаштувань і підвищує комфорт взаємодії. Для «справжнього» кешування – збереження SQL-результатів, HTML-фрагментів, API-відповідей – cookies технічно не придатні через жорсткі обмеження за обсягом, продуктивністю та безпекою.

Cookies схильні до таких загроз, як перехоплення (якщо не використовується HTTPS), маніпуляція з боку користувача, XSS-атаки або витік даних при неправильному налаштуванні доступу. Саме тому в cookies

категорично не рекомендується зберігати конфіденційні або критичні дані: токени доступу без обмежень, персональні дані, внутрішні ідентифікатори або службову інформацію. Сучасні атрибути (HttpOnly, Secure, SameSite) значно підвищують безпеку, але повністю не усувають ризики.

Кеш на серверній стороні значно безпечніший, оскільки його розміщено у контрольованому середовищі, доступ до якого регламентується політиками безпеки.

1.5 Особливості використання практик кешування

Перелічимо переваги, які надає кешування для кінцевого користувача вебзастосунку:

Пришвидшення завантажень. Сторінки завантажуються миттєво під час повторних відвідувань з одного девайсу. Більшість користувачів не помічають, що перше завантаження триваліше понад 30-80% (кінцеві значення дуже варіюються від вмісту кінцевого сторінки, швидкості інтернету, апаратних можливостей серверу, тому загальної константи для всіх можливих випадків не існує).

Для кращого усвідомлення результатів кешування розглянемо вебсторінку Тернопільський національний педагогічний університета імені Володимира Гнатюка (https://tnpu.edu.ua 21.05.2025) при загальній швидкості мережі інтернет: завантаження (Download) 93.18 Mbps, вивантаження (Upload) 94.13 Mbps; та використанні браузера Google Chrome (Version 135.0.7049.52 (Official Build) (64-bit)).

Для об'єктивності тесту потрібно очистити всі існуючі кеші, або ж використати «Incognito tab», яка перед створенням вкладки повністю ігнорує вже існуючі кеші та володіє тимчасовим сховищем для власних майбутніх кешів. Заміри швидкості будуть використовуватися вмонтованими інструментами розробника, зокрема, інструменти для роботи з мережею (Network tab).

Проведемо перший замір (рис. 1.3).

Рис 1.3. Результати першого заміру без кешованих даних (джерело: отримано автором)

За першим розміром без використання будь яких кеш файлів кінцеве завантаження відбулося за 1.93 с., DOM (загальна структура HTML сторінки) - 0.921 с.

Проведемо наступний замір, який вже використовує кешовані дані (рис. 1.4).

Рис 1.4. Результати другого заміру із використанням кешу (джерело: отримано автором)

За наступним заміром, з використання кешів вийшли наступні результати: кінцеве завантаження відбулося за 0.49 с., DOM (загальна структура HTML сторінки) – 0.483 с.

У порівнянні результатів стало відомо, що швидкість завантаження сторінки підвищилась на: кінцеве завантаження сторінки ~ 75%, DOM (загальна структура HTML сторінки) ~ 50%.

Варто зазначити, що для більш комплексних та більших веб застосунків тривалість завантаження може розтягуватися на декілька секунд, а гіршому випадку й на десятки секунд. Також потрібно розуміти, що «прості» чи «декілька» секунд грають критичну роль в кількості відвідувачів вебсторінки та позиції відображені в пошукових системах.

Швидкість відображення, та подальше кешування ресурсів береться до уваги спеціальними інструментами, які визначають певний ряд «оцінок», «метрик» для вебсторінки.

Згідно результатів діагностики пошукові системи визначають наступні критерії:

– позиція відображення посилання вебсторінки під час пошуку за ключовими словами, описом;

– чи варто відображати діагностовано сторінку (трапляється у випадках повного нехтування правилами безпеки користувача, жахливою швидкістю завантаження сторінки, сумнівним вмістом вебсторінки).

Зрозуміло, що ці критерії є критичними для будь-якої бізнесу та нехтування подібними правилами зі сторони розробки є повноцінною «професійної недбалістю» у разі розробки комерційного веб застосунку (рис. 1.5).

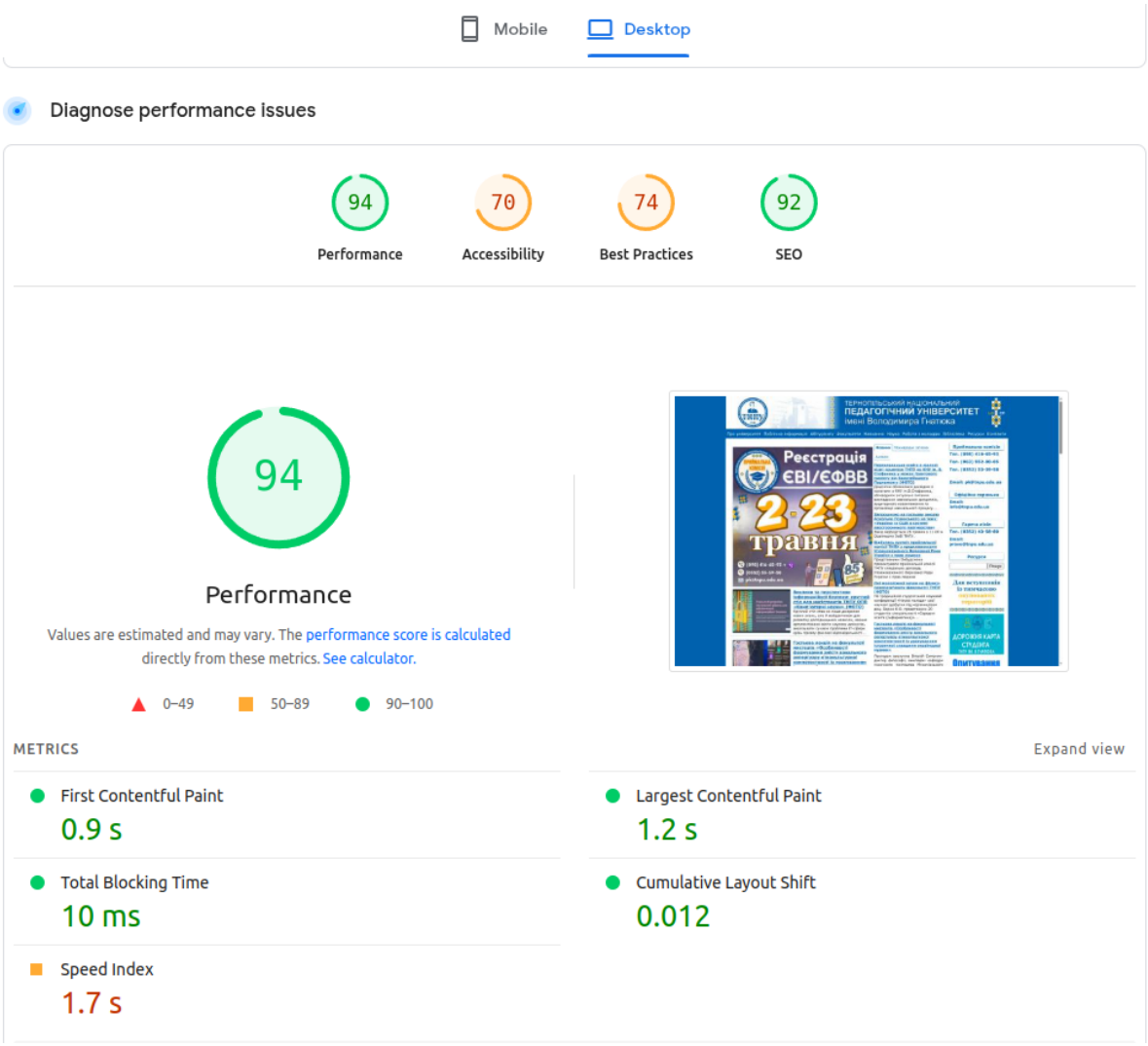

Рис 1.5. Результати оцінювання вебсторінки за допомогою сервісу «PageSpeed Insights»

Розглянемо ще практики оцінювання вебсторінки. Для прикладу було обрано оцінювання вебсторінки за допомогою веб сервісу «PageSpeed Insights» (https://pagespeed.web.dev/) – сервіс від «Google», який створений, щоб перевіряти та адаптувати власні вебсторінки до сучасних потреб пошукового сервісу «Google». Весь напрямок розробки та адаптації називається «SEO», «Search Engine Optimization». Кожний пошуковий сервіс має власні алгоритми для перевірки, аналізу та діагностування, тому результати відображення можуть сторінки у пошуковій системі можуть кардинально відрізнятися.

Варто зазначити, що існує інші схожі сервіси та інструменти. Одним з них є інструменти розробника від «Google» під назвою «Lighthouse», який прямо вбудований в браузер. Однією з причин обрання використання сервісу «PageSpeed Insights» є: налаштоване по замовчуванню обмеження по трафіку, приблизно завантаження (Download) 10 Mbps та вивантаження (Upload) 10.00 Mbps; фокусування сервісу на детальній діагностиці метрики «performance», яка є провідною для інших метрик та для швидкості вебсторінки (також вона зачіпає тему кешування).

Як було зазначено, нас цікавить оцінювання вебсторінки за допомогою сервісу «PageSpeed Insights». На рис. 1.3 показано отримані заміри.

Згідно діагностики вебсервісу ми отримали наступні результати:

– «Performance» (Продуктивність) – загальна швидкість вебсторінки, перше та найдовше відображення вмісту сторінки, загальний час відтворення HTML структури;

– «Accessibility» (Доступність) – дотримання правильної семантики сторінки, використання спеціальних описуючих атрибутів (aria-*, title, description, label і так далі), загальних практик для кращого досвіду користувача (використання контрастної кольорової палітри, доступні для розміру шрифти та інтерактивні елементи);

– «Best Practices» – дотримання загальних практик при роботі з сервером, скріптами, передаванням даних під час розорбки та кінцевих реалізацій;

– «SEO», «Search Engine Optimization» (Пошукова оптимізація) – дотримання правил пошукових систем, використання спеціальних семантичних та мета тегів для опису сторінки, продукту, розташування тощо.

Вебсервіс поставив нам відмінний результат (у межах від 90-100) для широкоформатних девайсів (також існує відокремлена діагностика для мобільних девайсів).

Для нашого розслідування розбір та вивчення всіх метрик не є доцільним, так як питання кешування піднімається лиш в метриці «Performance» (яка у свою чергу впливає на інші).

Під час аналізу вебсторінки в розділі «Performance», були знайдені зауваження та місця для покращення нашої вебсторінки (рис. 1.6).

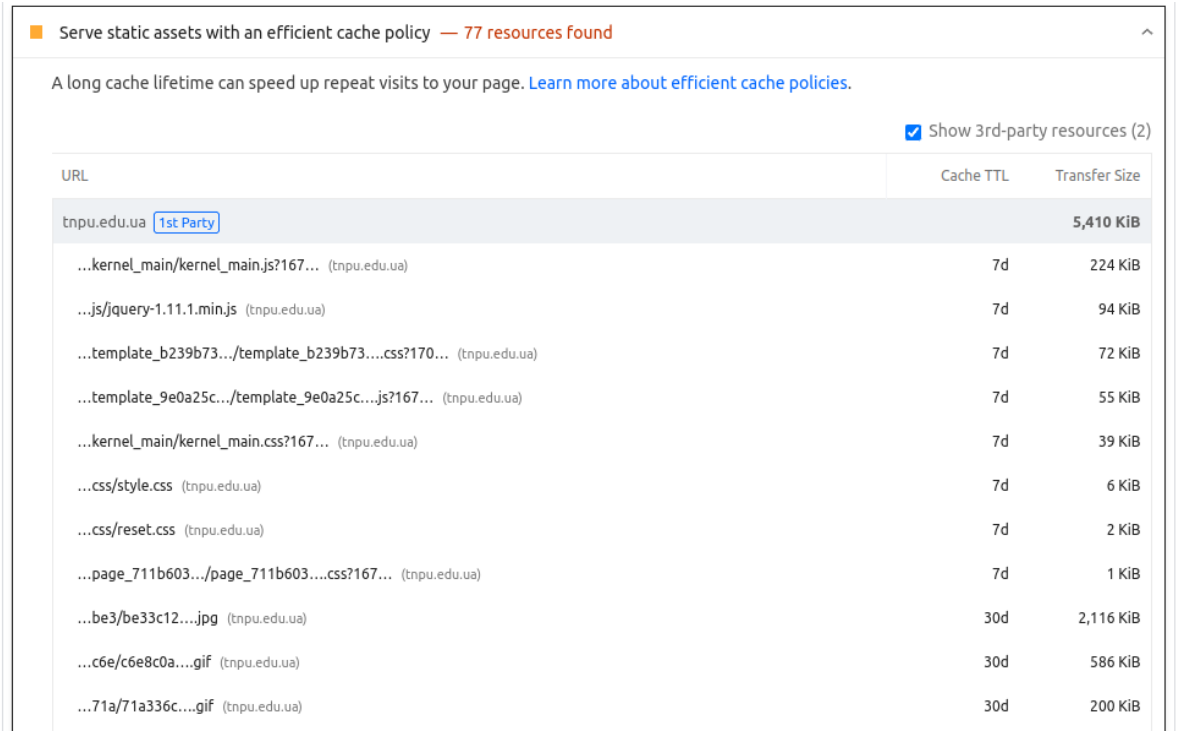

Рис 1.6. Розділ «Serve static assets with an efficient cache policy»

Розділ «Serve static assets with an efficient cache policy» (Обслуговуйте статичні ресурси з ефективною політикою кешування) (підказує, до яких

файлів ми б могли застосувати більш ефективніші правила. Згідно рекомендацій від документації для розробників від «Google», нам пропонують збільшити час тривалості життя кешованих на значно триваліший час. Наприклад, всі відмічені файли мають тривалість життя кеш файлів 7 днів та 30 днів. «Вам слід кешувати незмінні статичні ресурси протягом тривалого часу, наприклад, року або довше» (https://developer.chrome.com/docs/lighthouse/performance/uses-long-cache-ttl/?utm_source=lighthouse&utm_medium=lr#how_to_cache_static_resources_using_http_caching/).

Також вони застерігають: «Одним із ризиків тривалого кешування є те, що ваші користувачі не бачитимуть оновлень статичних файлів. Ви можете уникнути цієї проблеми, налаштувавши інструмент збірки на вбудовування хешу в імена файлів статичних ресурсів, щоб кожна версія була унікальною, що спонукатиме браузер отримувати нову версію із сервера».

Практика використання спеціального хешу в імені файлу для статичних є одним і найпопулярніших методів, для звільнення від старого кешу та отримання нових ресурсів прямо від сервера.

Також нам пропонують відмічати файли, які не повинні кешуватися за допомогою «no-cache» з заголовці HTTP запиту. Відповідна конфігурація повинна відбутися на стороні веб сервера.

При дотриманні всіх рекомендаціх вебсторінка повинна здобути вищу «оцінку», що приведе до частішого відображення в пошуковій системі та підняттям позиції ближче до верху.

При розгляді питання з точки зору бізнесу використання кешування є критично важливим аспектом з точки зору ефективного бізнесу. На сучасному ринку використання системи кешування відбиваються на наступних аспектах:

– вартості апаратного запезбечення;

– кількості потенційних користувачів;

– конкурентоспроможності.

Вартість апаратного забезпечення є важливим аспектом в роботі веб застосунках. Створення власного сервера, або оренда у спеціальних сервісах (AWS, Google Cloud, Digital Ocean) послуги на наданні VPS не є безкоштовними. При великих обсягах вебсторінки витрати на апаратну частину можуть сягати значних витрати. Мова може йти про десятки тисяч доларів, тому будь яка нагода зменшити статтю витрат вітається будь-де.

Попри те, що система кешування створювалися для пришвидшення роботи системи, кешування також показало себе як хороший інструмент для розгруження завантаження. Будь який запит, який обраховується не сервером, а кеш файлами - зменшує навантаження на сервер. Це в свою чергу надає змогу зменшити вимоги до апаратного забезпечення, що і призведе до зменшення витрат коштів.

Кількість потенційних користувачів прямо залежить від швидкості вебсторінки. Відсутність системи кешування лиш погіршить ситуацію та призведе до меншої кількості можливих відвідувачів на вебсторінку. Варто пам'ятати з точки зору бізнесу та маркетингу – втрачений відвідувач вебсторінки еквівалентно до втрати заробітку. Ситуацію також погіршують вимоги пошукових систем та їх «оцінки» відносно, яких визначається позиція відображення вебсторінки у пошуку. Хто буде користуватися сервісом, якого не можна знайти в перших рядках (сторінки) видачі результатів пошукового сервісу?

Згідно статті [11] форуму «web.dev» від «Google» 0.1 секунди може покращити кількість продаж понад на 8.5%, а загальну відвідуваність понад 10 % (рис. 1.7).

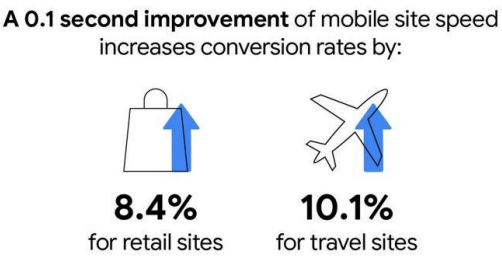

Рис. 1.7. Результати дослідження Milliseconds make millions [11]

«Нове дослідження Milliseconds Make Millions виявляє значний вплив швидкості завантаження мобільних сайтів на готовність споживачів витрачати гроші та взаємодіяти з брендами онлайн. Результати показують, що навіть покращення часу завантаження на 0,1 секунди може покращити темпи просування по всій воронці покупки. Також спостерігався позитивний вплив на кількість переглядів сторінок, коефіцієнти конверсії та середню вартість замовлення».

Тобто будь яке пришвидшення роботи вебсторінки позитивно показується в продажах. Ігнорування системи кешування загрожує життю бізнесу.

Конкурентоздатність. Виходячи з попередніх двох пунктів стає зрозуміло, що ігнорування кеш системи в сучасному бізнесі просто критично впливає на спроможність бізнесу. Чи можна назвати ефективним бізнесом, який ігнорує ефективні практики, які вже десятки років на озброєнні інших бізнесах? Чи можна прогнозувати, що бізнес, який уникає вже класичних інструментів для зменшення витрат апаратних статей успішним? Відповідь, вважаю, що очевидна.

Чи завжди потрібне кешування? Для комерційної розробки – так. Не комерційної розробки – так. Чи існують винятки – так. Можливі винятки:

– надмірно динамічні дані;

– невеликий розмір проєкту;

– робота з персональними даними користувача.

Невеликий розмір проєкту. Якщо ваш кінцевий продукт складається з декількох або однієї вебсторінки візитки (landing page), то доцільність часу витраченого на створення власної системи кешування на рівні застосунку – не дуже вигідно з точки зору комерційної розробки.

Також важливо розуміти, що в сучасних реаліях розробки вже існують десятки готових рішень. Більшість фреймворків завчасно мають інтегровані

системи кешування. Вже при релізованій системі для розробника залишається витрати час лиш для налаштування системи кешування.

Також існують завчасно створенні правила кешування зі сторони веб серверів (Apache, Nginx), у відповідних існують значні конфігураційні налаштування для цієї задачі. Використання веб сервера вже гарантує кешування.

Веббраузер теж не стоїть в стороні питання кешування. Якщо зі сторони веб сервера не прийшло жодних вказівок, щодо кешування, то він застосовує власні правила кешування до отриманих даних.

В кінці-кінців уникнути кешування – майже не можливо. Рівнів кешування десятки. Базові налаштування браузер та вебсервер – гарантовано створять кешовані файли, як зі сторони користувачів так і зі сторони вебсервера.

Висновок до першого розділу

У першому розділі досліджено теоретичні засади кешування як одного з базових механізмів підвищення продуктивності сучасних вебзастосунків, зокрема прогресивних вебзастосунків (PWA). У ході аналізу встановлено, що кешування є універсальною практикою комп’ютерних обчислень, яка передбачає тимчасове збереження попередньо обчислених або отриманих даних з метою мінімізації повторних операцій доступу, обчислень і мережевих звернень.

Розглянуто сутність поняття кешу, його часову природу та ключову роль у зменшенні затримок, оптимізації використання апаратних ресурсів і покращенні користувацького досвіду. Показано, що ефективність кешування безпосередньо залежить від коректного вибору даних для кешування, обсягу збережених ресурсів та механізмів інвалідації, які забезпечують актуальність інформації. Обґрунтовано, що кешування доцільно застосовувати насамперед

до статичних і умовно статичних ресурсів, тоді як динамічні та критичні дані потребують обмеженого або контрольованого часу життя кешу.

У межах розділу проаналізовано принципи розміщення кешу на різних рівнях вебархітектури – від клієнтського браузерного кешу до серверного, мережевого та application-level кешування. Визначено, що сучасні вебзастосунки, зокрема PWA, функціонують у середовищі багаторівневого кешування, де кожен рівень виконує власну оптимізаційну функцію та доповнює інші. Особливу увагу приділено HTTP-кешуванню, керованим кешам (CDN, reverse-proxy, Service Worker + Cache API), серверним in-memory сховищам, а також кешуванню на рівні бази даних.

Окремо розглянуто допоміжні механізми оптимізації, зокрема мемоізацію, індекси баз даних і flow tables, які, хоча й не є кешем у класичному розумінні, виконують схожу функцію скорочення часу обробки запитів. Проаналізовано використання cookies у контексті кешування та безпеки, що дозволило зробити висновок про їх обмежену придатність для оптимізації продуктивності та доцільність застосування лише для збереження невеликих некритичних даних конфігурації.

Практичний аналіз швидкодії вебсторінки підтвердив, що використання кешування забезпечує суттєве скорочення часу завантаження сторінок, позитивно впливає на ключові метрики продуктивності та безпосередньо корелює з вимогами пошукових систем. Доведено, що кешування є не лише технічним, а й економічно значущим інструментом, який сприяє зниженню витрат на інфраструктуру, підвищенню конкурентоспроможності вебпродукту та збільшенню залученості користувачів.

Узагальнюючи результати першого розділу, можна стверджувати, що кешування є невід’ємним компонентом сучасної веброзробки, уникнути якого практично неможливо. Його правильне проєктування та використання є необхідною умовою створення ефективних, масштабованих і конкурентоспроможних PWA-застосунків. Отримані теоретичні положення

слугують основою для подальшого аналізу конкретних стратегій і технологій кешування, що буде здійснено у наступних розділах роботи.

РОЗДІЛ 2
СТРАТЕГІЇ ТА ТЕХНОЛОГІЇ КЕШУВАННЯ ПРИ РОЗРОБЦІ PWA-БЛОГІВ

2.1 Особливості кешування в PWA: загальні принципи та роль Service Worker

Використання прогресивних вебзастосунків (Progressive Web Applications, PWA) є сучасним підходом до розробки вебсистем, що поєднує переваги традиційних вебтехнологій та нативних мобільних застосунків. Однією з ключових характеристик PWA є можливість стабільної роботи в умовах нестабільного або повністю відсутнього мережевого з'єднання. Досягнення цієї властивості безпосередньо пов'язане з ефективною організацією кешування ресурсів, яке ґрунтується на використанні Service Worker та відповідних механізмів керування мережевими запитами.

Загальні принципи кешування в PWA полягають у збереженні критично важливих ресурсів застосунку на стороні клієнта з метою скорочення часу завантаження, зменшення мережевого трафіку та підвищення надійності взаємодії користувача із системою. До таких ресурсів зазвичай належать HTML-документи, таблиці стилів, JavaScript-файли, шрифти, зображення, а також результати запитів до API. На відміну від традиційного браузерного кешу, який працює автоматично і з обмеженими можливостями контролю, кешування в PWA є програмно керованим процесом, що дозволяє розробнику визначати правила збереження, оновлення та видалення даних.

Центральним елементом архітектури кешування в PWA є Service Worker – спеціальний скрипт, що виконується браузером у фоновому режимі та працює незалежно від основного потоку виконання вебсторінки. Service Worker функціонує як проксі між мережею та клієнтським застосунком,

перехоплюючи HTTP-запити та приймаючи рішення щодо їх обробки. Завдяки цьому він може повертати відповіді з кешу, звертатися до мережі або комбінувати обидва підходи залежно від заданої логіки.

Процес життєвого циклу Service Worker складається з кількох етапів, кожен із яких має важливе значення для реалізації кешування. Під час фази встановлення (install) зазвичай здійснюється попереднє кешування основних ресурсів застосунку. Це забезпечує миттєву доступність інтерфейсу після першого відвідування сайту. На етапі активації (activate) відбувається очищення застарілих кешів та підготовка Service Worker до обробки запитів. Після активації Service Worker починає перехоплювати події fetch, що дозволяє реалізувати різноманітні стратегії кешування.

Приклад базової реалізації кешування під час встановлення Service Worker може виглядати наступним чином:

```
const CACHE_NAME = 'pwa-cache-v1';
const ASSETS_TO_CACHE = [
  '/',
  '/index.html',
  '/styles/main.css',
  '/scripts/app.js'
];

self.addEventListener('install', event => {
  event.waitUntil(
    caches.open(CACHE_NAME).then(cache => {
      return cache.addAll(ASSETS_TO_CACHE);
    })
  );
});
```

У наведеному фрагменті коду визначається ім’я кешу та перелік ресурсів, які необхідно зберегти локально. Під час встановлення Service Worker ці файли завантажуються та поміщаються у Cache Storage, що є спеціалізованим сховищем браузера для HTTP-відповідей. Такий підхід

забезпечує доступність інтерфейсу навіть за відсутності мережевого з’єднання.

Обробка мережевих запитів здійснюється через подію fetch, яка дозволяє визначити логіку взаємодії між кешем та мережею. Найпростішим прикладом є стратегія «cache-first», за якої Service Worker спочатку намагається знайти відповідь у кеші, а лише за її відсутності звертається до мережі

```
self.addEventListener('fetch', event => {
  event.respondWith(
    caches.match(event.request).then(response => {
      return response || fetch(event.request);
    })
  );
});
```

Застосування такої стратегії є доцільним для статичних ресурсів, які рідко змінюються та не потребують постійного оновлення. Водночас для динамічних даних, наприклад відповідей API, можуть використовуватися інші підходи, зокрема «network-first» або комбіновані стратегії з фоновим оновленням.

Важливою особливістю кешування в PWA є необхідність балансування між актуальністю даних та продуктивністю. Надмірне використання кешу може призводити до відображення застарілої інформації, тоді як часті звернення до мережі зменшують переваги PWA. Service Worker дозволяє гнучко керувати цим балансом, реалізовуючи логіку інвалідації кешу, версіонування ресурсів та умовного оновлення.

Окрім кешування статичних файлів, Service Worker може використовуватися для збереження результатів мережевих запитів до серверних API. Це особливо актуально для застосунків, орієнтованих на мобільні пристрої, де стабільність з’єднання не завжди гарантована. У таких випадках кешування відповідей дозволяє забезпечити часткову

функціональність застосунку в офлайн-режимі та покращити користувацький досвід.

Таким чином, загальні принципи кешування в PWA базуються на програмному контролі над мережевими запитами, використанні локального сховища для збереження ресурсів та впровадженні продуманих стратегій доступу до даних. Service Worker виступає ключовим компонентом цієї архітектури, забезпечуючи гнучкість, масштабованість і високу продуктивність прогресивних вебзастосунків. Саме завдяки його використанню PWA можуть наближатися за функціональністю та зручністю до нативних рішень, зберігаючи при цьому універсальність вебплатформи.

2.2 Workbox та додаткові механізми кешування в PWA

У процесі розробки прогресивних вебзастосунків реалізація ефективного кешування за допомогою чистого API Service Worker може швидко ускладнюватися. Необхідність обробляти різні типи ресурсів, застосовувати декілька стратегій кешування, керувати версіями кешу та враховувати крайові випадки призводить до зростання обсягу коду і підвищення ризику помилок. Для спрощення цих завдань було створено бібліотеку Workbox, яка надає високорівневі абстракції для роботи з кешуванням у PWA.

Workbox є набором модулів, розроблених компанією Google, які значно полегшують створення, підтримку та масштабування Service Worker. Основна ідея Workbox полягає в тому, щоб замінити ручну обробку подій install і fetch декларативним описом правил кешування. Завдяки цьому розробник може зосередитися на логіці застосунку, а не на деталях низькорівневої взаємодії з Cache Storage та Fetch API.

Однією з ключових переваг Workbox є чітке розмежування стратегій кешування. Бібліотека пропонує готові реалізації найбільш поширених підходів, таких як cache-first, network-first, stale-while-revalidate, network-only

та cache-only. Кожна стратегія інкапсулює оптимальну логіку обробки запитів і враховує типові сценарії використання. Наприклад, стратегія stale-while-revalidate дозволяє миттєво повертати дані з кешу, паралельно виконуючи фонове оновлення з мережі, що є ефективним компромісом між швидкодією та актуальністю інформації.

Приклад базової конфігурації Service Worker із використанням Workbox може виглядати наступним чином:

```
import { registerRoute } from 'workbox-routing';
import { CacheFirst } from 'workbox-strategies';
import { ExpirationPlugin } from 'workbox-expiration';

registerRoute(
  ({ request }) => request.destination === 'style' || request.destination
=== 'script',
  new CacheFirst({
    cacheName: 'static-resources',
    plugins: [
      new ExpirationPlugin({
        maxEntries: 50,
        maxAgeSeconds: 30 * 24 * 60 * 60
      })
    ]
  })
);
```

У цьому прикладі всі запити до стилів і скриптів обробляються за стратегією cache-first. Workbox автоматично створює відповідний кеш, зберігає відповіді та застосовує плагін ExpirationPlugin для обмеження кількості записів і часу їх зберігання. Такий підхід дозволяє уникнути накопичення застарілих даних без необхідності ручного керування життєвим циклом кешу.

Важливою особливістю Workbox є підтримка попереднього кешування (pre-caching), яке використовується для збереження ключових ресурсів під час встановлення Service Worker. На відміну від ручного додавання файлів у

кеш, Workbox може автоматично генерувати список ресурсів на основі збірки проєкту, включаючи хешовані імена файлів. Це значно зменшує ризик помилок і спрощує процес оновлення застосунку. При зміні вмісту файлу змінюється його хеш, що автоматично призводить до оновлення кешу.

Приклад використання попереднього кешування в Workbox:

```
import { precacheAndRoute } from 'workbox-precaching';
precacheAndRoute(self.__WB_MANIFEST);
```

Приклад використання Змінна ‘__WB_MANIFEST’ генерується на етапі збірки за допомогою відповідних плагінів і містить перелік ресурсів із контрольними сумами. Workbox самостійно керує процесом встановлення, активації та видалення застарілих кешів, що значно знижує складність коду Service Worker:

Окрім роботи зі статичними ресурсами, Workbox ефективно застосовується для кешування запитів до API. Для цього можуть використовуватися стратегії network-first або stale-while-revalidate залежно від вимог до актуальності даних. Наприклад, для новинних стрічок або списків товарів доцільно застосовувати підхід network-first, за якого система намагається отримати актуальні дані з мережі, але у разі недоступності з’єднання повертає збережену відповідь із кешу.

Застосування Workbox також сприяє підвищенню надійності та передбачуваності поведінки PWA. Бібліотека враховує особливості різних браузерів, обробляє типові помилки та забезпечує узгоджену роботу кешування в різних середовищах. Це особливо важливо в умовах реальних проєктів, де необхідно підтримувати широкий спектр пристроїв і версій браузерів.

З науково-практичної точки зору Workbox можна розглядати як рівень абстракції, що формалізує процес кешування в PWA та переводить його з імперативного стилю програмування в декларативний. Це сприяє кращій підтримуваності коду, зменшенню когнітивного навантаження на розробника

та підвищенню якості кінцевого продукту. У великих застосунках, де кількість правил кешування може бути значною, використання Workbox стає не просто зручністю, а необхідністю.

Таким чином, Workbox відіграє важливу роль у сучасній екосистемі PWA, надаючи стандартизовані інструменти для реалізації кешування та роботи з Service Worker. Його використання дозволяє ефективно поєднувати продуктивність, надійність і актуальність даних, що є ключовими вимогами до прогресивних вебзастосунків у реальних умовах експлуатації.

2.3 Використання Content Delivery Network при створенні PWA застосунку

Мережі доставки контенту (Content Delivery Network, CDN) є важливим компонентом сучасної вебінфраструктури, орієнтованої на високу продуктивність, масштабованість і надійність. У контексті прогресивних вебзастосунків (PWA) CDN відіграє особливу роль, оскільки доповнює клієнтські механізми кешування, зменшує затримки під час завантаження ресурсів і сприяє стабільній роботі застосунку незалежно від географічного розташування користувача [12]. Поєднання CDN з серверним рендерингом (SSR), Service Worker та серверною частиною на базі Laravel формує багаторівневу архітектуру доставки контенту, де кожен рівень виконує чітко визначену функцію.

Принцип роботи CDN полягає у розподіленні копій статичних і, в окремих випадках, динамічних ресурсів вебзастосунку між географічно рознесеними вузлами, так званими edge-серверами. Коли користувач ініціює запит до вебресурсу, цей запит обробляється не центральним сервером застосунку, а найближчим до користувача вузлом CDN. У результаті значно скорочується час відповіді, зменшується навантаження на основний сервер і підвищується загальна відмовостійкість системи.

У типовому стеку з використанням SSR-фреймворку Astro JS CDN зазвичай виступає першим рівнем взаємодії між користувачем і системою. Astro JS генерує HTML-сторінки на сервері, що дозволяє віддавати повністю сформований контент вже на першому запиті. Ці HTML-документи, разом зі статичними ресурсами (CSS, JavaScript, зображення), можуть кешуватися на рівні CDN. Таким чином, повторні запити до одних і тих самих сторінок обслуговуються без звернення до серверної частини на Laravel, що суттєво знижує час завантаження.

Використання CDN у зв’язці з SSR має важливе значення для початкової продуктивності PWA. Швидке отримання серверно згенерованого HTML сприяє скороченню показників First Contentful Paint та Largest Contentful Paint, що безпосередньо впливає на сприйняття швидкодії користувачем. Після початкового завантаження управління переходить до клієнтської логіки, де важливу роль починає відігравати Service Worker.

Service Worker у цій архітектурі функціонує як другий рівень кешування після CDN. Якщо CDN орієнтований на оптимізацію доставки контенту з мережі, то Service Worker забезпечує локальне кешування ресурсів безпосередньо в браузері користувача. Важливим моментом є те, що ці механізми не конкурують між собою, а взаємодіють комплементарно. CDN обслуговує перший запит і забезпечує швидку доставку даних, тоді як Service Worker дозволяє повторно використовувати ресурси без будь-яких мережевих звернень.

Наприклад, статичні файли, згенеровані Astro JS, можуть кешуватися на CDN з довгим часом життя за рахунок хешованих імен файлів. Паралельно Service Worker може використовувати стратегію cache-first для цих самих ресурсів, гарантуючи миттєвий доступ навіть у разі повної відсутності мережевого з’єднання. Таким чином, CDN оптимізує доставку при онлайн-доступі, а PWA-механізми забезпечують офлайн-функціональність.

Laravel у цьому стеку зазвичай відповідає за бізнес-логіку, роботу з базою даних та надання API. Запити до API можуть частково проходити через CDN, зокрема якщо йдеться про публічні GET-запити, які допускають кешування. При цьому важливим аспектом є коректна конфігурація HTTP-заголовків, таких як Cache-Control та ETag, що дозволяє CDN приймати рішення щодо збереження або перевалідації відповідей. У поєднанні з Service Worker це дає змогу реалізувати багаторівневе кешування динамічних даних.

Важливим моментом інтеграції CDN з PWA є узгодження стратегій оновлення контенту. Надмірно агресивне кешування на рівні CDN може призводити до ситуацій, коли користувач отримує застарілу версію застосунку, навіть якщо Service Worker вже оновив локальні ресурси. Тому в практиці широко застосовується підхід версіонування ресурсів і чітке розмежування типів контенту. Критично важливі HTML-документи зазвичай кешуються з коротким часом життя або з обов’язковою перевіркою актуальності, тоді як статичні ресурси можуть зберігатися тривалий час.

Ще одним аспектом є безпека та контроль доступу. CDN часто використовується як перший захисний бар’єр, виконуючи функції фільтрації трафіку, захисту від DDoS-атак та обмеження доступу до певних ресурсів. У PWA це має особливе значення, оскільки Service Worker працює лише в безпечному контексті HTTPS. Використання CDN із підтримкою TLS дозволяє спростити налаштування безпечного з’єднання та забезпечити стабільну роботу застосунку.

З архітектурної точки зору поєднання CDN, SSR Astro JS, Service Worker та Laravel формує багаторівневу модель доставки контенту. CDN мінімізує затримки на глобальному рівні, SSR забезпечує швидкий старт і доступність контенту для пошукових систем, Service Worker відповідає за офлайн-режим і локальну продуктивність, а Laravel реалізує серверну бізнес-логіку та управління даними. Така модель відповідає сучасним вимогам до високонавантажених вебзастосунків і добре масштабується зі зростанням

CDN є невід’ємним елементом ефективної PWA-архітектури, особливо в умовах використання серверного рендерингу та складної серверної логіки. Його правильна інтеграція з клієнтськими механізмами кешування дозволяє досягти оптимального балансу між швидкодією, актуальністю даних і надійністю. У стеку Astro JS, Service Worker та Laravel CDN виступає не просто інструментом оптимізації, а фундаментальним рівнем інфраструктури, що забезпечує стабільну та прогнозовану поведінку прогресивного вебзастосунку в реальних умовах експлуатації.кількості користувачів.

HTTP-акселератор Varnish: принцип роботи. Varnish Cache є високопродуктивним HTTP-акселератором, призначеним для зменшення навантаження на серверні застосунки та суттєвого підвищення швидкодії вебсистем. Його основне завдання полягає у кешуванні HTTP-відповідей і подальшому обслуговуванні клієнтських запитів без необхідності звернення до бекенд-сервера. У сучасних умовах, коли вебзастосунки обслуговують значну кількість користувачів і виконують складні серверні обчислення, використання Varnish стає важливим архітектурним рішенням.

З точки зору мережевої архітектури Varnish розташовується між клієнтом і серверною частиною застосунку, виконуючи роль зворотного проксі-сервера. Усі HTTP-запити спочатку надходять до Varnish, який приймає рішення щодо їх подальшої обробки. Якщо відповідь на запит уже присутня в кеші та вважається актуальною, Varnish негайно повертає її клієнту. У протилежному випадку запит передається до бекенд-сервера, відповідь якого зберігається у кеші для подальшого використання.

Принцип роботи Varnish ґрунтується на зберіганні кешованих об’єктів у оперативній пам’яті, а не на диску. Такий підхід дозволяє досягати надзвичайно низьких затримок і високої пропускної здатності. На відміну від класичних проксі-рішень, орієнтованих на файлові кеші, Varnish оптимізований саме для обробки великої кількості HTTP-запитів з мінімальними накладними витратами.

Важливою особливістю Varnish є використання власної мови конфігурації – Varnish Configuration Language (VCL). За допомогою VCL розробник або системний адміністратор може детально описати правила обробки запитів і відповідей, включаючи умови кешування, модифікацію HTTP-заголовків, логіку маршрутизації та механізми інвалідації кешу. Це дозволяє адаптувати поведінку Varnish до специфіки конкретного застосунку.

Типовий сценарій роботи Varnish можна описати наступним чином. Клієнт надсилає HTTP-запит до вебресурсу, який потрапляє до Varnish. Система аналізує метод запиту, URL, заголовки та інші параметри. Якщо запит відповідає правилам кешування, Varnish перевіряє наявність відповідного об’єкта у кеші. У разі позитивного результату відповідь повертається клієнту майже миттєво. Якщо ж кешований об’єкт відсутній або втратив актуальність, запит передається до бекенд-сервера, а отримана відповідь зберігається для подальших звернень.

Для прикладу, кешування HTTP-відповідей у Varnish може бути налаштоване за допомогою VCL таким чином:

```
sub vcl_backend_response {
    if (bereq.url ~ "^/posts/") {
        set beresp.ttl = 10m;
    }
}
```

У наведеному фрагменті визначається правило, за яким відповіді для URL, що починаються з /posts/, зберігаються в кеші протягом десяти хвилин. Після завершення цього часу Varnish вважатиме кешований об’єкт застарілим і повторно звертатиметься до бекенд-сервера.

Однією з ключових переваг Varnish є можливість значного зменшення навантаження на серверні застосунки. У типових сценаріях більшість запитів користувачів спрямовані на однакові ресурси, наприклад сторінки блогу,

списки товарів або публічні API-ендпоїнти. Кешування таких відповідей дозволяє скоротити кількість звернень до бази даних, зменшити використання процесорних ресурсів і підвищити стабільність системи в пікові моменти навантаження.

Varnish також широко використовується для згладжування сплесків трафіку. У ситуаціях, коли кількість запитів різко зростає, кеш дозволяє обслуговувати більшість клієнтів без деградації продуктивності бекенд-сервера. Це є особливо актуальним для інформаційних порталів, новинних сайтів і блогів, де значна частина контенту є публічною і не потребує персоналізації.

Важливим аспектом роботи Varnish є керування кешем і його інвалідація. Оскільки кешування може призводити до відображення застарілих даних, необхідно забезпечити механізми своєчасного оновлення контенту. Varnish підтримує як пасивну інвалідацію на основі часу життя (TTL), так і активну – шляхом явного очищення кешу за URL або шаблоном. Це дозволяє інтегрувати Varnish у процеси розгортання та оновлення застосунку.

З точки зору безпеки та контролю доступу Varnish може виконувати роль першого рівня фільтрації HTTP-запитів. За допомогою VCL можна обмежувати доступ до певних ресурсів, блокувати підозрілий трафік або змінювати заголовки відповідей. Хоча Varnish не є повноцінним засобом захисту, він може доповнювати інші компоненти безпекової інфраструктури.

Varnish Cache є потужним інструментом оптимізації вебзастосунків, який забезпечує високопродуктивне кешування HTTP-відповідей на рівні оперативної пам’яті. Його використання дозволяє суттєво скоротити час відповіді, зменшити навантаження на бекенд і підвищити масштабованість системи. Завдяки гнучкій конфігурації та підтримці складних сценаріїв кешування Varnish залишається одним із ключових рішень для побудови ефективних і надійних вебархітектур.

2.4 Інтеграція сховища Redis в REST API блог

Redis є високопродуктивним сховищем даних типу key-value, що працює переважно в оперативній пам’яті та широко використовується для оптимізації серверних застосунків. У контексті REST API блогу Redis зазвичай застосовується як допоміжний компонент бекенд-архітектури з метою зменшення кількості звернень до бази даних, прискорення відповіді API та підвищення стабільності системи під навантаженням. На відміну від класичних реляційних баз даних, Redis не призначений для довготривалого зберігання основних даних, а виконує роль швидкого кешу або тимчасового сховища.

Принцип роботи Redis ґрунтується на зберіганні даних у пам’яті у вигляді простих або складних структур, таких як рядки, хеші, списки, множини та впорядковані множини. Доступ до цих даних здійснюється за ключем, що забезпечує надзвичайно низьку затримку операцій читання і запису. У REST API блогу це дозволяє швидко повертати популярний контент, наприклад списки статей, окремі публікації або метадані, без повторного виконання дорогих SQL-запитів.

Типовий сценарій інтеграції Redis у бекенд блогу полягає у кешуванні результатів GET-запитів. Наприклад, під час звернення клієнта до ендпоїнту, який повертає список останніх публікацій, сервер спочатку перевіряє наявність відповідних даних у Redis. Якщо кешований результат присутній і не втратив актуальності, він негайно повертається клієнту. У разі відсутності кешу сервер виконує запит до основної бази даних, формує відповідь і зберігає її в Redis з визначеним часом життя.

У застосунках на базі Laravel інтеграція Redis є відносно простою, оскільки фреймворк надає вбудовану підтримку цього сховища. Redis може використовуватися як драйвер кешу, черг або сесій, що дозволяє централізувати логіку роботи з тимчасовими даними. Для REST API блогу

найчастіше застосовується саме кешування відповідей або проміжних результатів обчислень.

Приклад кешування результату API-запиту в Laravel із використанням Redis може виглядати наступним чином:

```
use Illuminate\Support\Facades\Cache;

public function index()
{
    return Cache::remember('posts_list', 600, function () {
        return Post::latest()->take(10)->get();
    });
}
```

У цьому прикладі результат запиту до бази даних зберігається в Redis на десять хвилин. Протягом цього часу всі повторні запити до ендпоїнту будуть обслуговуватися з кешу, що значно зменшує навантаження на базу даних і скорочує час відповіді.

Важливим аспектом інтеграції Redis є стратегія інвалідації кешу. У блогових системах дані змінюються не надто часто, однак оновлення або публікація нового матеріалу повинні відображатися користувачам без значних затримок. Тому після створення, оновлення або видалення публікації доцільно очищати або оновлювати відповідні ключі в Redis. У Laravel це може бути реалізовано через події моделі або спостерігачі, що дозволяє автоматизувати процес підтримання актуальності кешу.

Окрім кешування списків і окремих ресурсів, Redis часто використовується для зберігання лічильників переглядів, рейтингів або інших агрегованих даних. Такі операції є типовими для блогів, але можуть створювати значне навантаження на реляційну базу даних. Redis дозволяє виконувати інкрементальні операції з мінімальними витратами, а періодична синхронізація з основною базою забезпечує узгодженість даних.

Ще одним важливим напрямом використання Redis у REST API є обмеження частоти запитів, або rate limiting. Захист API від надмірної кількості запитів з одного джерела є критично важливим для стабільної роботи системи. Redis добре підходить для цієї задачі завдяки швидким операціям і підтримці атомарних лічильників. У результаті система може ефективно контролювати доступ до API без значного впливу на продуктивність.

З архітектурної точки зору Redis у бекенд-системі блогу виконує роль проміжного шару між застосунком і базою даних. Він дозволяє ізолювати основну базу від пікових навантажень і забезпечити прогнозовану швидкодію API. Водночас важливо враховувати, що Redis є інструментом оптимізації, а не заміною постійного сховища. Дані в Redis можуть бути втрачені у разі перезапуску або збою, тому критично важлива інформація повинна зберігатися в основній базі даних.

У підсумку інтеграція Redis у REST API блогу є ефективним способом підвищення продуктивності та масштабованості серверної частини застосунку. Використання Redis для кешування, лічильників і контролю доступу дозволяє значно зменшити навантаження на базу даних, скоротити час відповіді API та покращити загальну якість обслуговування клієнтів. За умови коректної організації інвалідації кешу та узгодження з основною базою даних Redis стає важливим елементом сучасної бекенд-архітектури блогових платформ.

2.5 Laravel Cache. Кешування для REST API

Laravel є сучасним PHP-фреймворком, орієнтованим на розробку масштабованих і підтримуваних вебзастосунків. Його архітектура базується на принципах MVC, інверсії керування та чіткого розділення відповідальностей. У контексті headless-підходу Laravel найчастіше використовується не як класичний сервер рендерингу HTML, а як бекенд, що

надає REST API або GraphQL-інтерфейси для клієнтських застосунків. У такій моделі фронтенд, наприклад побудований за допомогою SSR-фреймворку Astro JS, повністю відокремлений від серверної логіки і взаємодіє з Laravel виключно через HTTP-запити.

У headless-архітектурі Laravel виконує роль центрального сервісу, відповідального за бізнес-логіку, роботу з базою даних, автентифікацію, авторизацію та підготовку даних у форматі JSON. Саме тому питання продуктивності REST API є критично важливим. Кожен SSR-запит з боку Astro JS може ініціювати кілька HTTP-звернень до API, що за відсутності оптимізації створює значне навантаження на базу даних. Laravel Cache у цьому контексті виступає ключовим інструментом підвищення ефективності.

Механізм кешування в Laravel є абстрактним шаром, який дозволяє працювати з різними сховищами даних через єдиний API. Фреймворк підтримує декілька драйверів кешу, зокрема файлову систему, базу даних, Redis та Memcached [28]. Для REST API у production-середовищі зазвичай використовується Redis, оскільки він забезпечує низьку затримку доступу та добре масштабується під навантаженням. При цьому логіка кешування залишається однаковою незалежно від конкретного драйвера.

У типовому сценарії headless-блогу більшість запитів припадає на операції читання, зокрема отримання списку публікацій та перегляд окремих матеріалів. Саме ці запити є основними кандидатами для кешування. Наприклад, SSR-фреймворк Astro JS під час генерації сторінки блогу може звертатися до API для отримання списку постів. Якщо кожен такий запит призводить до виконання SQL-запиту SELECT * FROM posts, це створює зайве навантаження на сервер. Кешування дозволяє один раз отримати дані з бази і надалі повертати їх з оперативної пам’яті.

Розглянемо наведений контролер PostController, який реалізує базові ендпоїнти REST API. Метод index повертає всі пости без будь-якої оптимізації:

```
public function index(): JsonResponse
{
    return response()->json(["ok" => true, "posts" => Post::all()]);
}
```

У такому вигляді кожен виклик цього ендпоїнту призводить до прямого звернення до бази даних. Для інтеграції кешу достатньо обгорнути логіку отримання даних у механізм Cache::remember. Це дозволяє зберігати результат виконання запиту протягом визначеного часу:

```
use Illuminate\Support\Facades\Cache;

public function index(): JsonResponse
{
    $posts = Cache::remember('posts.index', 600, function () {
        return Post::all();
    });

    return response()->json(["ok" => true, "posts" => $posts]);
}
```

У цьому прикладі список постів кешується на 600 секунд. Якщо Astro JS або будь-який інший клієнт повторно звертається до цього ендпоїнту протягом зазначеного часу, Laravel повертає дані з кешу, минаючи базу даних. Для SSR це має особливе значення, оскільки скорочує час генерації сторінок і зменшує загальну затримку відповіді.

Аналогічний підхід застосовується до ендпоїнту show, який повертає конкретний пост за ідентифікатором. У базовій реалізації кожен запит виконує пошук у базі даних:

```
public function show(string $identifier): JsonResponse
{
    $post = Post::where("identifier", $identifier)->first();

    if (!isset($post)) {
```

```
        return response()->json(["ok" => false, "post" => null], 404);
    }

    return response()->json(["ok" => true, "post" => $post]);
}
```

Для кешування цього запиту доцільно використовувати унікальний ключ, що залежить від ідентифікатора поста:

```
public function show(string $identifier): JsonResponse
{
    $post = Cache::remember("posts.show.{$identifier}", 600, function ()
use ($identifier) {
        return Post::where("identifier", $identifier)->first();
    });

    if (!isset($post)) {
        return response()->json(["ok" => false, "post" => null], 404);
    }

    return response()->json(["ok" => true, "post" => $post]);
}
```

Такий підхід дозволяє SSR-фреймворку Astro JS багаторазово отримувати дані одного й того ж поста без повторних звернень до бази даних. Особливо це актуально для сторінок, що часто відвідуються або активно індексуються пошуковими системами.

З архітектурної точки зору Laravel Cache у headless-сценарії виступає внутрішнім рівнем оптимізації між REST API і базою даних. Він не замінює кешування на рівні CDN або Service Worker, але ефективно доповнює їх. CDN може кешувати HTTP-відповіді, а Laravel Cache зменшує вартість генерації цих відповідей. У поєднанні ці механізми дозволяють досягти стабільної та передбачуваної продуктивності системи.

Важливо зазначити, що кешування на рівні Laravel є повністю прозорим для клієнта. Astro JS не потребує жодних змін у своїй логіці,

оскільки API зберігає той самий контракт. Це відповідає принципам чистої headless-архітектури, де оптимізація бекенду не впливає на структуру фронтенд-застосунку.

Ефективність кешування в REST API визначається не лише швидкістю доступу до кешу, але й коректною стратегією його інвалідації. У блогових системах основною проблемою є узгодженість даних: після створення або оновлення публікації кешовані відповіді мають бути оновлені або видалені, щоб клієнтські застосунки не отримували застарілу інформацію. У headless-архітектурі, де SSR-фреймворк Astro JS покладається на API як єдине джерело істини, ця задача набуває особливої важливості. Розглянемо метод store у наведеному контролері PostController. Він відповідає за створення нового поста і наразі не містить логіки роботи з кешем:

```
public function show(string $identifier): JsonResponse
{
    $post = Post::where("identifier", $identifier)->first();

    if (!isset($post)) {
        return response()->json(["ok" => false, "post" => null], 404);
    }

    return response()->json(["ok" => true, "post" => $post]);
}
```

З точки зору кешування цей метод є критичним, оскільки після створення нового поста стає неактуальним кеш списку публікацій (posts.index). Найпростішим і водночас ефективним рішенням є явне очищення відповідного ключа кешу після успішного запису в базу даних:

```
use Illuminate\Support\Facades\Cache;

public function store(Request $request): JsonResponse
{
    $validated = $request->validate([
```

```
        "title" => "required|string|max:255",
        "author" => "required|string|max:255",
        "content" => "required|min:150",
        "time_to_read" => "required"
    ]);

    $post = Post::create($validated);

    Cache::forget('posts.index');

    return response()->json(["ok" => true, "post" => $post]);
}
```

Такий підхід гарантує, що наступний запит до ендпоїнту index призведе до повторного звернення до бази даних і оновлення кешу. Для SSR Astro JS це означає, що нова публікація буде врахована вже під час наступної серверної генерації сторінки.

Однак у більш складних системах ручне управління окремими ключами кешу може стати джерелом помилок. Саме тому Laravel підтримує механізм cache tags, який дозволяє логічно групувати кешовані дані. Наприклад, усі кеші, пов’язані з постами блогу, можуть бути об’єднані одним тегом:

```
$posts = Cache::tags(['posts'])->remember('posts.index', 600, function () {
    return Post::all();
});
```

Аналогічно для перегляду окремого поста:

```
$post = Cache::tags(['posts'])->remember("posts.show.{$identifier}", 600,
function () use ($identifier) {
    return Post::where("identifier", $identifier)->first();
});
```

Після створення або оновлення публікації достатньо очистити всі кеші, пов’язані з тегом posts:

```
Cache::tags(['posts'])->flush();
```

Цей підхід значно спрощує підтримку кешу і зменшує ризик залишення застарілих даних. Важливо зазначити, що cache tags підтримуються не всіма драйверами кешу. Для production-середовища з Redis це не є проблемою, оскільки Redis повністю підтримує тегований кеш у Laravel.

Redis у headless-REST API виступає оптимальним драйвером кешування з кількох причин. По-перше, він працює в оперативній пам’яті, що забезпечує мінімальну затримку доступу. По-друге, Redis добре масштабується і може використовуватися одночасно для кешу, черг та rate limiting. По-третє, Laravel надає глибоку інтеграцію з Redis, що дозволяє використовувати його без додаткових абстракцій. У конфігурації Laravel достатньо вказати Redis як основний драйвер кешу: CACHE_DRIVER=redis.

Після цього всі виклики Cache::remember, Cache::forget та Cache::tags працюватимуть через Redis. З точки зору REST API це означає, що кешовані відповіді можуть обслуговувати тисячі запитів без залучення бази даних.

Особливу роль у headless-архітектурі відіграє час життя кешу (TTL). Занадто великий TTL може призводити до застарілих даних, тоді як занадто малий – зменшує ефективність кешування. У блогових системах часто застосовується компромісний підхід: списки постів кешуються на коротший час, наприклад 5-10 хвилин, тоді як окремі публікації можуть зберігатися довше. При цьому інвалідація кешу при створенні або редагуванні постів гарантує актуальність даних незалежно від TTL.

Взаємодія Laravel Cache з SSR Astro JS має непрямий, але критично важливий характер. Astro JS під час серверного рендерингу виконує HTTP-запити до REST API, очікуючи швидких і стабільних відповідей. Кешування на рівні Laravel зменшує час обробки цих запитів і забезпечує передбачувану

продуктивність незалежно від кількості одночасних SSR-запитів. У результаті SSR перестає бути дорогим з точки зору серверних ресурсів.

У багаторівневій архітектурі кешування Laravel Cache займає внутрішній рівень, розташований між базою даних і зовнішніми механізмами, такими як CDN. CDN може кешувати вже готові HTTP-відповіді, тоді як Laravel Cache оптимізує сам процес їх генерації. У випадку промаху CDN або при персоналізованих запитах саме кеш Laravel стає основним фактором швидкодії.

Laravel Cache у headless-REST API є фундаментальним інструментом оптимізації серверної частини застосунку. У поєднанні з Redis він дозволяє ефективно кешувати запити читання, контролювати інвалідацію даних і підтримувати високу продуктивність під навантаженням. Для SSR-фреймворків, таких як Astro JS, це означає швидку генерацію сторінок, стабільну роботу і чітке розмежування відповідальностей між фронтендом і бекендом. Такий підхід відповідає сучасним архітектурним практикам і є оптимальним для масштабованих headless-систем.

Висновок до другого розділу

У другому розділі було комплексно проаналізовано стратегії та технології кешування, що застосовуються при розробці прогресивних вебзастосунків блогового типу. Основну увагу приділено практичним аспектам побудови багаторівневої архітектури кешування, яка поєднує клієнтські, мережеві та серверні механізми з метою підвищення продуктивності, надійності та масштабованості PWA.

У ході аналізу встановлено, що ключовим елементом клієнтського кешування в PWA є Service Worker, який забезпечує програмний контроль над мережевими запитами та дозволяє реалізовувати різні стратегії доступу до ресурсів. Розглянуто життєвий цикл Service Worker та показано, що саме етапи встановлення, активації та обробки подій fetch створюють основу для

реалізації офлайн-режиму, швидкого старту застосунку та зменшення залежності від стабільності мережевого з’єднання. Обґрунтовано доцільність використання різних стратегій кешування залежно від типу ресурсів, зокрема cache-first для статичних файлів і network-first або комбінованих підходів для динамічних даних.

Окрему увагу приділено бібліотеці Workbox як інструменту високого рівня абстракції для роботи з Service Worker. Показано, що Workbox суттєво спрощує реалізацію кешування, переводячи її з імперативного стилю програмування у декларативний. Використання готових стратегій, механізмів попереднього кешування та автоматичного керування версіями ресурсів зменшує складність коду, підвищує його підтримуваність і знижує ризик помилок. У контексті реальних PWA-проєктів Workbox розглянуто не лише як допоміжну бібліотеку, а як важливий інфраструктурний компонент.

У межах розділу проаналізовано роль мереж доставки контенту (CDN) у PWA-архітектурі. Встановлено, що CDN є ефективним засобом оптимізації доставки статичних і частково динамічних ресурсів, особливо у поєднанні з серверним рендерингом. Показано, що інтеграція CDN з SSR-фреймворком Astro JS, Service Worker та серверною частиною на Laravel формує багаторівневу модель кешування, де кожен рівень виконує окрему функцію. CDN мінімізує мережеві затримки на глобальному рівні, тоді як Service Worker забезпечує локальну продуктивність і офлайн-доступність.

Розглянуто принципи роботи HTTP-акселератора Varnish як серверного проксі-кешу, орієнтованого на високопродуктивне обслуговування HTTP-запитів. Доведено, що використання Varnish дозволяє суттєво зменшити навантаження на бекенд-застосунок, скоротити час відповіді та підвищити стабільність системи під піковими навантаженнями. Завдяки гнучкій конфігурації на основі VCL Varnish може бути адаптований до специфіки блогових платформ і ефективно інтегрований у сучасну вебархітектуру.

Значну увагу приділено серверному кешуванню на рівні REST API з використанням Redis та вбудованих механізмів Laravel Cache. Встановлено,

що Redis є оптимальним рішенням для кешування результатів запитів читання, лічильників і допоміжних даних завдяки роботі в оперативній пам’яті та низькій затримці доступу. Проаналізовано практичні сценарії кешування в Laravel, зокрема використання Cache::remember, інвалідацію кешу та механізм тегів. Показано, що коректна інтеграція Redis і Laravel Cache дозволяє ізолювати базу даних від надмірного навантаження та забезпечити передбачувану продуктивність REST API.

Узагальнюючи результати другого розділу, можна зробити висновок, що ефективна реалізація кешування в PWA-блогах можлива лише за умови комплексного підходу. Поєднання клієнтського кешування (Service Worker, Workbox), мережевих рішень (CDN, Varnish) та серверних механізмів (Redis, Laravel Cache) формує багаторівневу архітектуру. Такий підхід дозволяє досягти оптимального балансу між швидкодією, актуальністю даних і масштабованістю системи, що є критично важливим для сучасних прогресивних вебзастосунків.

Отримані у другому розділі результати створюють практичну й теоретичну основу для подальшої реалізації та експериментальної перевірки обраних стратегій кешування, що здійснено в наступному розділі кваліфікаційної роботи.

РОЗДІЛ 3
АНАЛІЗ ШВИДКОДІЇ ЗАСТОСУНКУ ТА КЕШУ

3.1 Оцінка продуктивності вебзастосунку за допомогою інструменту PageSpeed Insights

PageSpeed Insights (PSI) є одним із базових інструментів оцінки продуктивності вебзастосунків [22], розробленим компанією Google та побудованим на основі Lighthouse. На відміну від сервісів, орієнтованих переважно на синтетичні тести завантаження, PageSpeed Insights фокусується на ключових показниках користувацького досвіду, зокрема Core Web Vitals, які безпосередньо впливають на SEO та загальну сприйнятність вебресурсу. У цьому розділі розглядаються результати тестування сторінки блогу, реалізованого з використанням SSR Astro JS, серверного кешування (Varnish, Redis) та клієнтських механізмів оптимізації.

Згідно з результатами PageSpeed Insights, сторінка демонструє Performance score 100, що є максимальним можливим значенням. Додатково зафіксовано високі оцінки за іншими напрямами: Accessibility – 94, Best Practices – 63, SEO –100 [23]. Такий розподіл балів свідчить про те, що основний фокус оптимізації був зосереджений саме на продуктивності та пошуковій доступності, що є типовим для headless-архітектур із серверним рендерингом.

У секції «Discover what your real users are experiencing» зазначено No Data, тобто відсутні дані CrUX (Chrome User Experience Report). Це означає, що сторінка або домен ще не мають достатньої кількості реальних користувацьких сесій для формування статистично значущих польових метрик. У контексті наукового аналізу це не є недоліком, оскільки оцінювання проводиться на основі вимірювань Lighthouse, які дозволяють об’єктивно порівнювати архітектурні рішення та результати оптимізації.

Основні метрики продуктивності, зафіксовані PageSpeed Insights, мають надзвичайно низькі значення, що свідчить про ефективну оптимізацію всього ланцюга завантаження:

– First Contentful Paint (FCP) – 0.3 с;

– Largest Contentful Paint (LCP) – 0.3 с;

– Speed Index (SI) – 0.3 с;

– Total Blocking Time (TBT) – 0 мс;

– Cumulative Layout Shift (CLS) – 0.002.

Такі показники практично недосяжні для традиційних SPA або клієнтсько-орієнтованих рішень без SSR. Значення FCP та LCP у 300 мс означають, що основний контент сторінки відображається майже миттєво після отримання HTML-документа. Це є прямим наслідком використання SSR у Astro JS, де HTML генерується на сервері й передається браузеру без необхідності очікування виконання JavaScript.

Особливо показовим є Total Blocking Time = 0 мс, що свідчить про повну відсутність довгих задач у головному потоці браузера. Такий результат досягається завдяки мінімальному використанню клієнтського JavaScript, що є однією з ключових філософій Astro JS. Уся складна логіка обробки даних винесена на сервер (Laravel + Redis), а клієнт отримує вже готовий до відображення результат.

Показник Cumulative Layout Shift дорівнює 0.002, що є практично ідеальним значенням. Це означає, що візуальна стабільність сторінки збережена, а будь-які зсуви елементів є мінімальними і непомітними для користувача. Такий результат досягається за рахунок попереднього резервування місця під контент, стабільного шрифтового рендерингу та відсутності динамічних DOM-змін після початкового завантаження.

У даному стеку це забезпечується поєднанням Varnish і Redis. Varnish кешує HTTP-відповіді та зменшує кількість запитів, що доходять до Laravel-застосунку, тоді як Redis забезпечує миттєвий доступ до кешованих

результатів REST API без виконання повторних SQL-запитів. У сукупності ці механізми формують надзвичайно короткий шлях обробки запиту на сервері.

PageSpeed Insights ідентифікує потенційну проблему у вигляді render-blocking requests з орієнтовною економією 130 мс. Йдеться про два CSS-файли Astro (index.css, about.css) загальним розміром менше 4 КБ. З практичної точки зору ці ресурси мають мінімальний вплив на продуктивність, що підтверджується ідеальними значеннями FCP та LCP.

Також у звіті наведено інформацію про layout shift culprits, де загальний CLS формується елементом списку постів. Проте сумарне значення зсуву залишається на рівні 0.002, що вважається відмінним показником і не потребує додаткових коригувань.

Результати PageSpeed Insights демонструють, що вебзастосунок досяг максимальної продуктивності в лабораторних умовах Lighthouse. Це є наслідком не окремих точкових оптимізацій, а системного підходу до архітектури: використання SSR Astro JS для швидкого первинного рендерингу, серверного кешування через Varnish і Redis для мінімізації часу відповіді, а також мінімального клієнтського JavaScript для усунення блокування головного потоку браузера.

Отримані показники формують надійну основу для подальшого аналізу в GTmetrix, який доповнює Lighthouse-метрики детальнішою інформацією про мережеві таймінги, waterfall-діаграми та повне завантаження сторінки. Таким чином, PageSpeed Insights у цьому дослідженні виступає як первинний інструмент оцінки ефективності оптимізації, що підтверджує коректність обраного стеку технологій та архітектурних рішень.

3.2 Дослідження метрик завантаження вебзастосунку за допомогою інструменту GTmetrix

Для цього дослідження було використано сервіс GTmetrix, який поєднує результати Lighthouse Performance та власні метрики завантаження сторінки [24].

У блоці LCP breakdown зазначено, що Time to First Byte (TTFB) фактично становить 0 мс у рамках лабораторного тесту. Це не означає фізичну відсутність серверної затримки, а вказує на те, що відповідь сервера надходить швидше, ніж мінімальний поріг фіксації Lighthouse [26]. Такий результат можливий лише за умови ефективного серверного кешування.

Загальна оцінка GTmetrix Grade становить A, що свідчить про високий рівень оптимізації вебзастосунку. Показник Performance дорівнює 98%, а Structure – 96%, що вказує не лише на швидке завантаження сторінки, але й на коректну організацію ресурсів, мережевих запитів і внутрішньої структури застосунку. Такі результати є нетиповими для стандартних вебпроєктів без оптимізації і підтверджують ефективність використаних кешуючих механізмів.

Одним із найбільш показових результатів є значення Largest Contentful Paint (LCP), яке становить 936 мс. Це означає, що найбільший візуально значущий елемент сторінки відображається менш ніж за одну секунду після початку завантаження. Такий результат безпосередньо пов’язаний із використанням серверного рендерингу (SSR) в Astro JS. Завдяки SSR браузер отримує вже сформований HTML-документ, що дозволяє швидко відобразити основний контент без очікування виконання JavaScript.

Додатково слід відзначити First Contentful Paint (FCP), який також становить 937 мс. Практично ідентичні значення FCP та LCP свідчать про те, що сторінка не має важких або відкладених елементів, які суттєво затримують первинне відображення. Це є характерною перевагою

архітектури Astro JS, де JavaScript підключається вибірково, а більшість контенту рендериться на сервері.

У блоці LCP breakdown зазначено, що Time to First Byte (TTFB) фактично становить 0 мс у рамках тесту. Це не означає фізичну відсутність серверної затримки, а вказує на те, що відповідь сервера надходить швидше, ніж мінімальний поріг фіксації Lighthouse. Такий результат можливий лише за умови ефективного серверного кешування.

Показник Total Blocking Time (TBT) дорівнює 0 мс, що є ідеальним значенням. Це означає, що головний потік браузера не блокується тривалими JavaScript-операціями. Такий результат є прямим наслідком мінімального використання клієнтського JavaScript та відсутності важких синхронних скриптів. У даному випадку логіка застосунку зосереджена або на сервері (Laravel + Redis), або виконується асинхронно, що повністю відповідає сучасним рекомендаціям щодо продуктивності.

Не менш важливим є показник Cumulative Layout Shift (CLS), який дорівнює 0. Це свідчить про відсутність несподіваних зсувів макету під час завантаження сторінки. Такий результат досягається завдяки чітко визначеним розмірам елементів, попередньому резервуванню місця під шрифти та зображення, а також відсутності динамічного DOM-маніпулювання після первинного рендерингу.

Окремої уваги заслуговує аналіз Time to First Byte (TTFB), який становить 524 мс. Для динамічного вебзастосунку з серверною логікою це є хорошим показником. Згідно з деталізацією браузерних таймінгів, безпосередній час обробки запиту бекендом становить лише 157 мс, тоді як решта часу припадає на встановлення з’єднання та мережеві накладні витрати.

Зменшення серверного часу відповіді стало можливим завдяки використанню Varnish як HTTP-акселератора та Redis як кешу для REST API. Varnish кешує HTML-відповіді та зменшує кількість запитів, що доходять до Laravel-застосунку. Redis, у свою чергу, дозволяє швидко повертати дані API

без виконання повторних SQL-запитів. У комплексі ці рішення значно знижують навантаження на сервер і стабілізують TTFB навіть під потенційним зростанням трафіку.

Час Fully Loaded Time становить 3.5 с, а Onload Time – 2.8 с. Ці показники свідчать про те, що навіть другорядні ресурси, такі як шрифти та іконки, завантажуються без критичного впливу на основний користувацький досвід. Важливо зазначити, що загальний розмір сторінки становить лише 930 КБ, а кількість HTTP-запитів – 9, що є надзвичайно низьким показником для сучасного вебсайту.

Найбільшу частку трафіку займають шрифти, зокрема файл змінного шрифту розміром 855 КБ. Незважаючи на це, відсутність негативного впливу на LCP і FCP свідчить про правильну стратегію їх завантаження та кешування. У цьому контексті важливу роль відіграє Service Worker, який після першого відвідування зберігає шрифти та статичні ресурси у браузерному кеші, забезпечуючи миттєвий доступ при повторних візитах.

GTmetrix ідентифікує декілька потенційних зауважень, зокрема рекомендацію Use a Content Delivery Network (CDN) та Eliminate render-blocking resources [25]. Важливо підкреслити, що ці зауваження мають низький пріоритет і не чинять суттєвого впливу на загальну продуктивність. У даному випадку роль CDN частково компенсується використанням Varnish, який виконує схожу функцію кешування на edge-рівні. Крім того, мінімальний обсяг CSS та відсутність блокуючих JavaScript-ресурсів зменшують актуальність цих рекомендацій.

Також зазначається Reduce initial server response time, однак фактичний час обробки бекендом становить лише 156–157 мс, що є результатом застосування Redis-кешу та оптимізованої серверної логіки. Таким чином, ці зауваження слід розглядати не як недоліки, а як потенційні напрями подальшого мікрооптимізування.

Отримані результати GTmetrix демонструють, що висока швидкодія вебзастосунку є прямим наслідком комплексної оптимізації на всіх рівнях

архітектури. SSR у Astro JS забезпечує швидке первинне відображення контенту, Varnish і Redis мінімізують час обробки запитів на сервері, а Service Worker відповідає за ефективне клієнтське кешування. Такий підхід відповідає сучасним вимогам до продуктивних вебсистем і дозволяє досягати показників, близьких до нативних застосунків.

У підсумку можна стверджувати, що представлений стек технологій і реалізовані механізми кешування є ефективним рішенням для побудови швидких, масштабованих і стабільних вебзастосунків. Результати GTmetrix підтверджують, що оптимізація була виконана цілеспрямовано та системно, а не випадково, що є важливим аргументом у науковому або практичному обґрунтуванні обраної архітектури.

Висновок до третього розділу

У третьому розділі було проведено експериментальний аналіз швидкодії вебзастосунку блогу з використанням сучасних інструментів оцінки продуктивності – PageSpeed Insights та GTmetrix. Основною метою дослідження було емпірично підтвердити ефективність реалізованих механізмів кешування та обґрунтувати вплив обраної архітектури на ключові показники користувацького досвіду.

Результати тестування за допомогою PageSpeed Insights продемонстрували максимально можливий рівень продуктивності вебзастосунку в лабораторних умовах Lighthouse. Отримані значення Core Web Vitals, зокрема First Contentful Paint, Largest Contentful Paint, Speed Index та Total Blocking Time, свідчать про практично миттєве відображення основного контенту та відсутність блокування головного потоку браузера. Це підтверджує доцільність використання серверного рендерингу в Astro JS у поєднанні з мінімальним клієнтським JavaScript, що дозволяє усунути типові проблеми продуктивності, характерні для традиційних SPA.

Особливо важливим результатом є нульове або близьке до нуля значення Total Blocking Time та мінімальний Cumulative Layout Shift, що вказує на високу візуальну стабільність сторінки та передбачувану поведінку інтерфейсу під час завантаження. Це свідчить про коректну організацію макету, відсутність динамічних зсувів елементів і правильну стратегію завантаження стилів, шрифтів та контенту.

Аналіз результатів GTmetrix доповнив дані Lighthouse більш детальною інформацією про мережеві таймінги та серверну частину обробки запитів. Отримана оцінка A та високі значення показників Performance і Structure підтверджують, що оптимізація охоплює не лише візуальний рендеринг, а й внутрішню організацію ресурсів та мережевих запитів. Значення Largest Contentful Paint менше однієї секунди та відсутність затримок, пов’язаних з виконанням JavaScript, свідчать про ефективність обраного підходу до побудови фронтенду.

Окрему увагу в межах дослідження було приділено показнику Time to First Byte. Аналіз GTmetrix засвідчив, що безпосередній час обробки запиту серверною частиною є мінімальним, а загальне значення TTFB формується переважно мережевими накладними витратами. Це дозволяє зробити висновок, що використання Varnish як HTTP-акселератора та Redis як кешу для REST API суттєво скорочує серверну складову затримки і забезпечує стабільну швидкодію навіть для динамічного контенту.

Додаткові зауваження інструментів тестування, зокрема рекомендації щодо використання CDN або усунення рендер-блокуючих ресурсів, мають низький пріоритет і не чинять суттєвого впливу на загальну продуктивність. У даному випадку їх наявність не суперечить отриманим високим показникам і може розглядатися лише як потенційний напрям подальшого мікрооптимізування, а не як критичний недолік реалізації.

Загалом результати третього розділу переконливо підтверджують, що висока швидкодія вебзастосунку є наслідком системного підходу до оптимізації та кешування на всіх рівнях архітектури. Серверний рендеринг у

Astro JS забезпечує швидке первинне відображення контенту, Varnish і Redis мінімізують час обробки HTTP-запитів на сервері, а Service Worker сприяє ефективному повторному використанню ресурсів на клієнтській стороні. У сукупності ці механізми дозволяють досягти показників продуктивності, близьких до нативних застосунків, що повністю відповідає сучасним вимогам до прогресивних вебзастосунків.

Отримані експериментальні результати підтверджують коректність обраних технологічних рішень та архітектури застосунку і слугують практичним доказом ефективності різних стратегій кешування при розробці PWA-блогів. Це створює надійну основу для формування загальних висновків кваліфікаційної роботи та обґрунтування практичної цінності отриманих результатів.

ЗАГАЛЬНІ ВИСНОВКИ

У ході виконання даної кваліфікаційної роботи було комплексно досліджено роль кешування як одного з ключових механізмів підвищення продуктивності сучасних вебзастосунків, зокрема блогових платформ, реалізованих на основі технології Progressive Web Apps (PWA). Досягнута мета дослідження підтверджує доцільність і ефективність використання багаторівневих стратегій кешування для оптимізації швидкодії, покращення користувацького досвіду та підвищення стабільності вебсистем.

У межах виконання першого завдання було теоретично обґрунтовано та проаналізовано технологічні аспекти використання PWA. Досліджено архітектурні особливості Progressive Web Applications, роль Service Worker, Cache API та типових патернів кешування. Встановлено, що застосування клієнтського кешування є фундаментальною складовою сучасної вебархітектури, яка безпосередньо впливає на швидкість завантаження ресурсів, зменшення мережевих затримок і забезпечення офлайн-доступності.

У процесі виконання другого завдання здійснено порівняльний аналіз використання кешування та сценаріїв його ігнорування, а також досліджено різні інструменти інтеграції кешування, зокрема Redis та Apollo Cache. Результати аналізу показали, що відсутність кешування призводить до значного збільшення часу відповіді сервера та навантаження на інфраструктуру, тоді як використання in-memory та HTTP-кешування забезпечує істотний приріст продуктивності. Визначено переваги та обмеження кожного підходу, що дозволяє обґрунтовано обирати оптимальну стратегію залежно від типу даних і сценарію використання.

Виконання третього завдання дало змогу дослідити практичні аспекти розробки PWA, зокрема використання сучасних інструментів, фреймворків і кращих практик. Особливу увагу приділено аналізу впливу кешування на

користувацький досвід. На основі вимірювань швидкості завантаження, часу відгуку та поведінкових метрик встановлено, що кешування істотно покращує сприйняття швидкодії вебзастосунку кінцевим користувачем, зменшує кількість блокуючих операцій і сприяє підвищенню показників Core Web Vitals.

У межах четвертого завдання було розроблено прикладний вебзастосунок типу блогу на основі технології PWA, який використовується як демонстраційна платформа для практичного тестування реалізованих стратегій кешування. Архітектура застосунку базується на headless-підході з використанням REST API на Laravel та серверного рендерингу (SSR) за допомогою Astro JS. Кешування в цій моделі виконує роль ключового елемента взаємодії між бекендом і фронтендом, забезпечуючи швидке отримання даних і стабільне відображення контенту.

Ефективність обраних технічних рішень підтверджена результатами аналізу за допомогою інструментів PageSpeed Insights та GTmetrix. Отримані показники, зокрема високі значення Performance Score, мінімальні значення First Contentful Paint і Largest Contentful Paint, відсутність Total Blocking Time та майже нульовий Cumulative Layout Shift, свідчать про реальний практичний ефект застосування кешування та його безпосередній вплив на якість користувацького досвіду.

Окремо встановлено, що максимальна ефективність досягається не за рахунок використання окремої технології, а завдяки комплексному поєднанню серверного та клієнтського кешування. Саме узгоджене застосування Redis, Varnish, HTTP-кешування, Service Worker і Workbox дозволяє оптимізувати всі етапи обробки запиту – від звернення до бази даних до відображення сторінки в браузері.

У підсумку можна зробити висновок, що кешування є невід’ємною складовою сучасної веброзробки та одним із найефективніших інструментів підвищення продуктивності PWA-застосунків. Результати, отримані в межах даної роботи, підтверджують доцільність застосування багаторівневих

стратегій кешування, їх позитивний вплив на користувацький досвід, масштабованість системи та зниження витрат на інфраструктуру, а також демонструють практичну цінність реалізованих рішень у реальному вебпроєкті.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Caching. URL: https://web.dev/learn/pwa/caching (дата звернення: 16.10.2025).

2. Progressive Web Apps: Caching. Mozilla Developer Network (MDN).URL: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps/Guides/Caching (дата звернення: 16.10.2025).

3. Service Worker API. Mozilla Developer Network URL: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API (дата звернення: 11.11.2025).

4. Cache API. Mozilla Developer Network (MDN). URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache (дата звернення: 10.09.2025).

5. Workbox Documentation. Google Developers. URL: https://developer.chrome.com/docs/workbox (дата звернення: 10.09.2025).

6. Client-side caching. Adobe Commerce PWA Studio Documentation URL: https://developer.adobe.com/commerce/ pwa-studio/guides/general-concepts/client-side-caching (дата звернення: 10.09.2025).

7. Redis. IBM Think. URL: https://www.ibm.com/think/topics/redis (дата звернення: 15.09.2025).

8. Varnish Cache – Introduction. Varnish Software. URL: https://varnish-cache.org/intro (дата звернення: 15.09.2025).

9. Caching in Progressive Web Apps web.dev. URL: https://web.dev/learn/pwa/caching/ (дата звернення: 15.09.2025).

10. Using the Cache interface. Mozilla Developer Network (MDN). URL: https://developer.mozilla.org/en-US/docs/Web/API/Cache (дата звернення: 15.09.2025).

11. Demidova O. Milliseconds make millions. URL: https://web.dev/case-studies/milliseconds-make-millions (дата звернення: 15.09.2025).

12. DigitalOcean. Enable CDN for Spaces. URL: https://docs.digitalocean.com/products/spaces/how-to/enable-cdn/ (дата звернення: 25.11.2025).

13. MDN Contributors. HTTP caching. MDN Web Docs. U RL: https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Caching (дата звернення: 15.10.2025).

14. Varnish Software. Varnish Cache Documentation. URL: https://vinyl-cache.org/docs/ (дата звернення: 24.11.2025).

15. Redis Ltd. Redis Documentation. URL: https://redis.io/docs/latest/ (дата звернення: 24.11.2025).

16. Cloudflare. Caching Overview. URL: https://developers.cloudflare.com/cache/ (дата звернення: 25.11.2025).

17. Laravel. Cache – Laravel 12.x Documentation. URL: https://laravel.com/docs/12.x/cache (дата звернення: 24.11.2025).

18. Laravel. Laravel 12.x Documentation. URL: https://laravel.com/docs/12.x (дата звернення: 24.11.2025).

19. Amazon Web Services. Database Caching // Amazon Web Services, Inc. URL: https://aws.amazon.com/caching/ (дата звернення: 15.10.2025).

20. Emerich, Alex. Introduction to database caching. Prisma Data Guide. URL: https://www.prisma.io/dataguide/managing-databases/introduction-database-cachingCocca (дата звернення: 15.10.2025).

21. What is Memoization? How and When to Memoize in JavaScript and React. FreeCodeCamp. FreeCodeCamp News, 26 Apr 2022. URL: https://www.freecodecamp.org/news/memoization-in-javascript-and-react/ (дата звернення: 24.10.2025).

22. Google Developers. About PageSpeed Insights. URL: https://developers.google.com/speed/docs/insights/v5/about (дата звернення: 22.11.2025).

23. Google Developers. Run PageSpeed Insights API. URL: https://developers.google.com/speed/docs/insights/rest/v5/pagespeedapi/runpagespeed (дата звернення: 23.11.2025).

24. GTmetrix. How to View CrUX Data on GTmetrix. URL: https://gtmetrix.com/blog/how-to-view-crux-data-on-gtmetrix/ (дата звернення: 23.11.2025).

25. GTmetrix. Using the GTmetrix Waterfall Chart. URL: https://gtmetrix.com/blog/using-the-gtmetrix-waterfall-chart/ (дата звернення: 23.11.2025).

26. Google Chrome Developers. Serve static assets with an efficient cache policy (Lighthouse). Chrome for Developers. URL: https://developer.chrome.com/docs/lighthouse/performance/uses-long-cache-ttl/ (дата звернення: 30.08.2025).

27. Fastly. Working with Object Storage. URL: https://www.fastly.com/documentation/guides/platform/object-storage/working-with-object-storage/ (дата звернення: 25.11.2025).

28. Memcached. Memcached Documentation. URL: https://docs.memcached.org/ (дата звернення: 25.11.2025).

29. Google Chrome Developers. Performance Scoring in Lighthouse. URL: https://developer.chrome.com/docs/lighthouse/performance/performance-scoring (дата звернення: 25.11.2025).

30. Google Chrome Developers. Lighthouse Total Blocking Time. URL: https://developer.chrome.com/docs/lighthouse/performance/lighthouse-total-blocking-time (дата звернення: 25.11.2025).

31. Базиволяк М., Шмигер Г. Порівняльний аналіз механізмів кешування в прогресивних вебзастосунках. *Сучасні цифрові технології та інноваційні методики навчання: досвід, тенденції, перспективи*: матеріали XV Міжнародної науково-практичної інтернет-конференції (м. Тернопіль, 10 квітня 2025 р.). Тернопіль: ТНПУ ім. Володимира Гнатюка, 2025. С. 131–135.

32. Базиволяк М., Шмигер Г. Особливості використання різних стратегій кешування при розробці блогів на основі технології PWA. *Сучасні цифрові технології та інноваційні методики навчання: досвід, тенденції, перспективи*: матеріали XVI Міжнародної науково-практичної інтернет-конференції (м. Тернопіль, 6–7 листопада 2025 р.). Тернопіль: ТНПУ ім. Володимира Гнатюка, 2025. С. 223–226.

ДОДАТКИ

ДОДАТОК А

Рис. 1. Домашня сторінка PWA застосунку

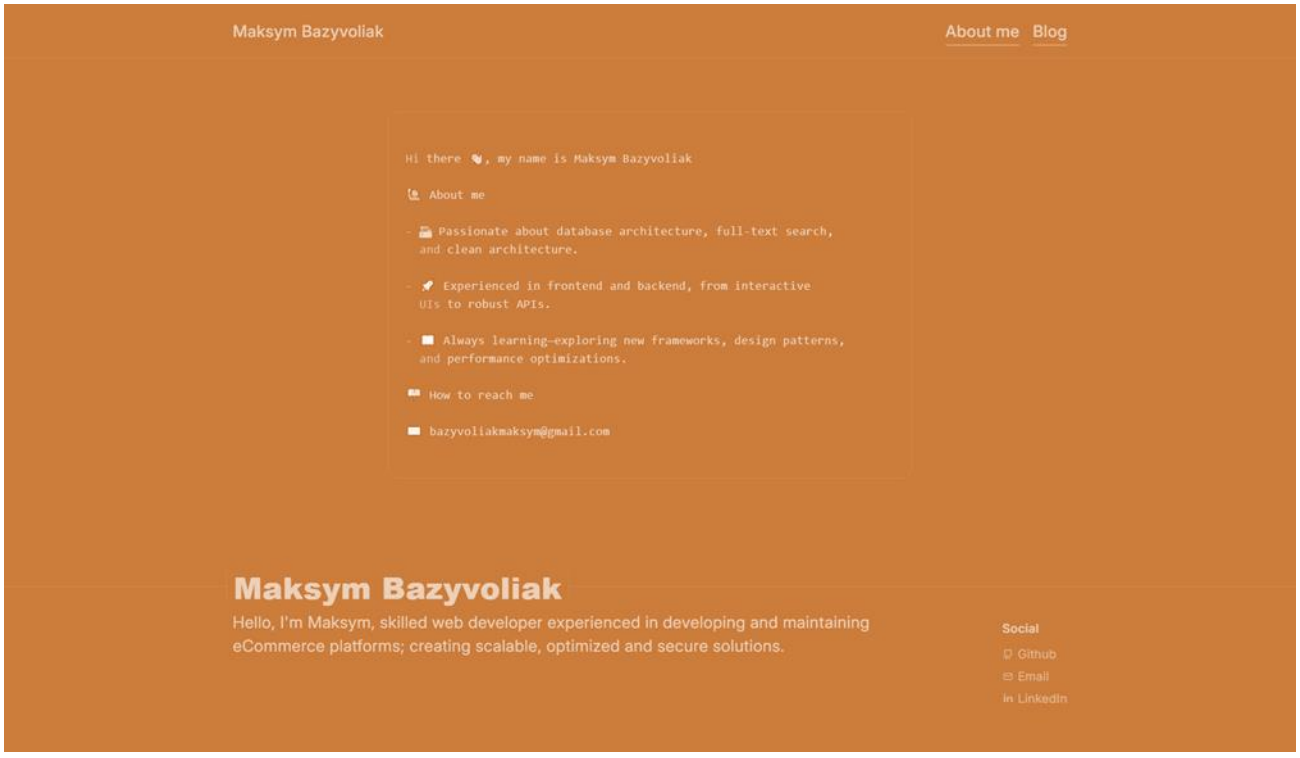

Рис. 2. «About me» сторінка PWA застосунку

Рис. 3. Стаття «SEO optimization. Best practices» PWA застосунку

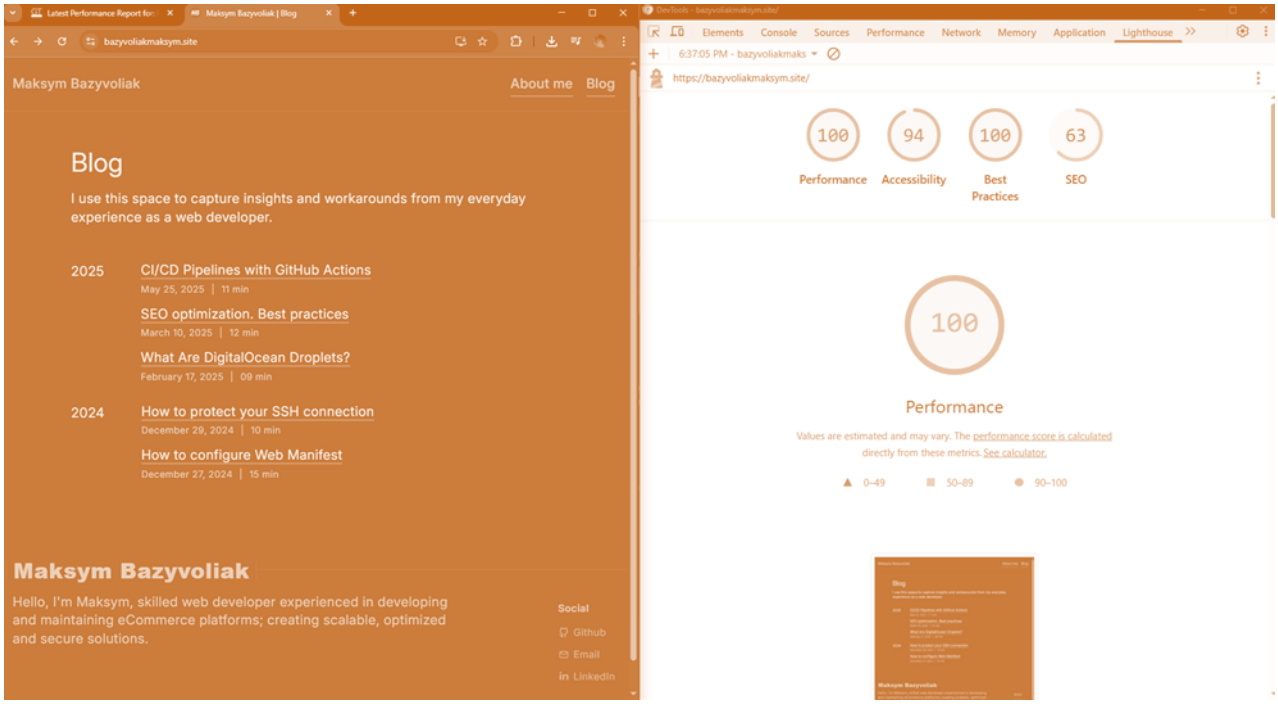

Рис. 4. Репорт швидкодії «Lighthouse» PWA застосунку

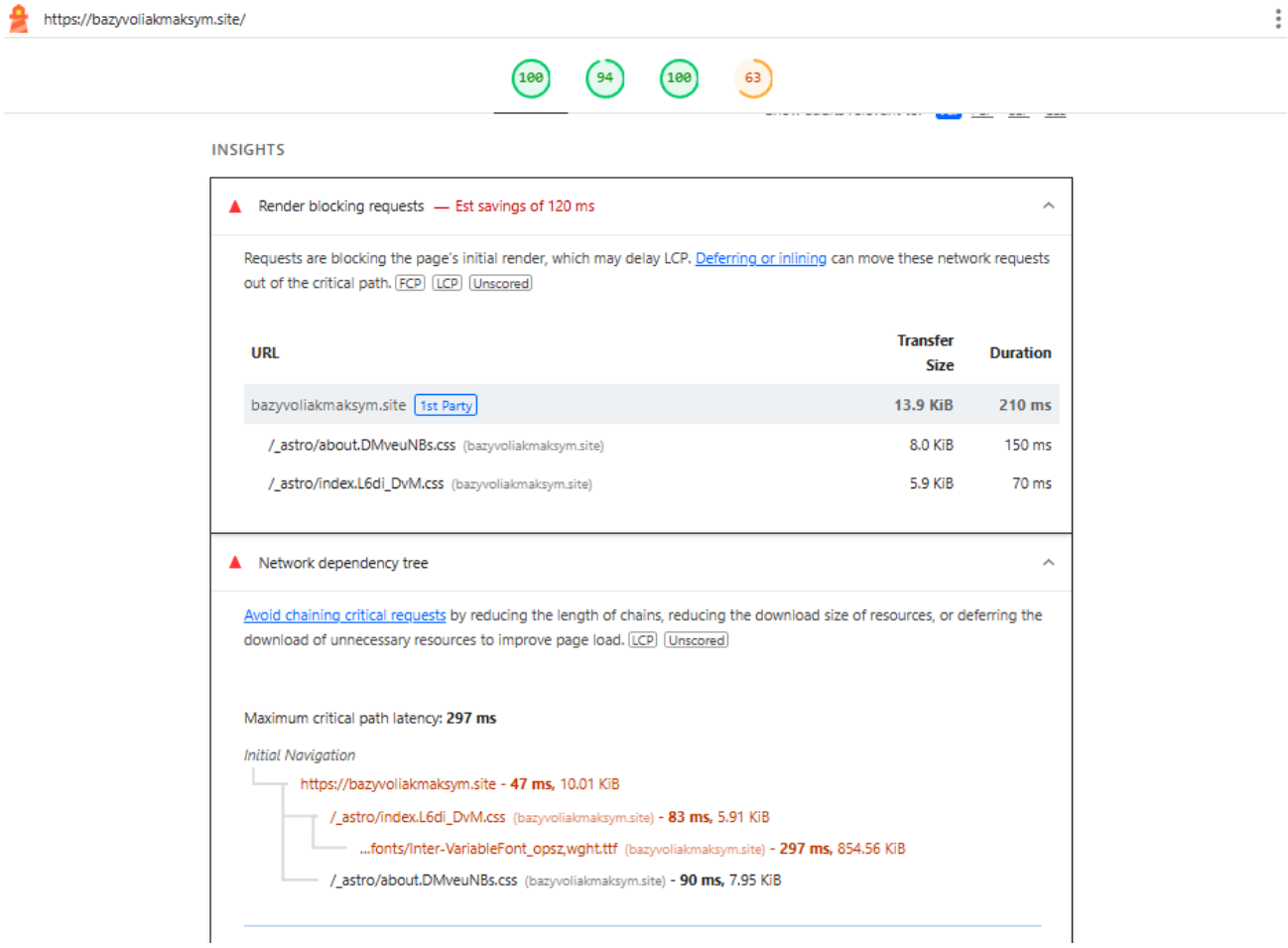

Рис. 5. Зауваження репорту швидкодії «Lighthouse» PWA застосунку

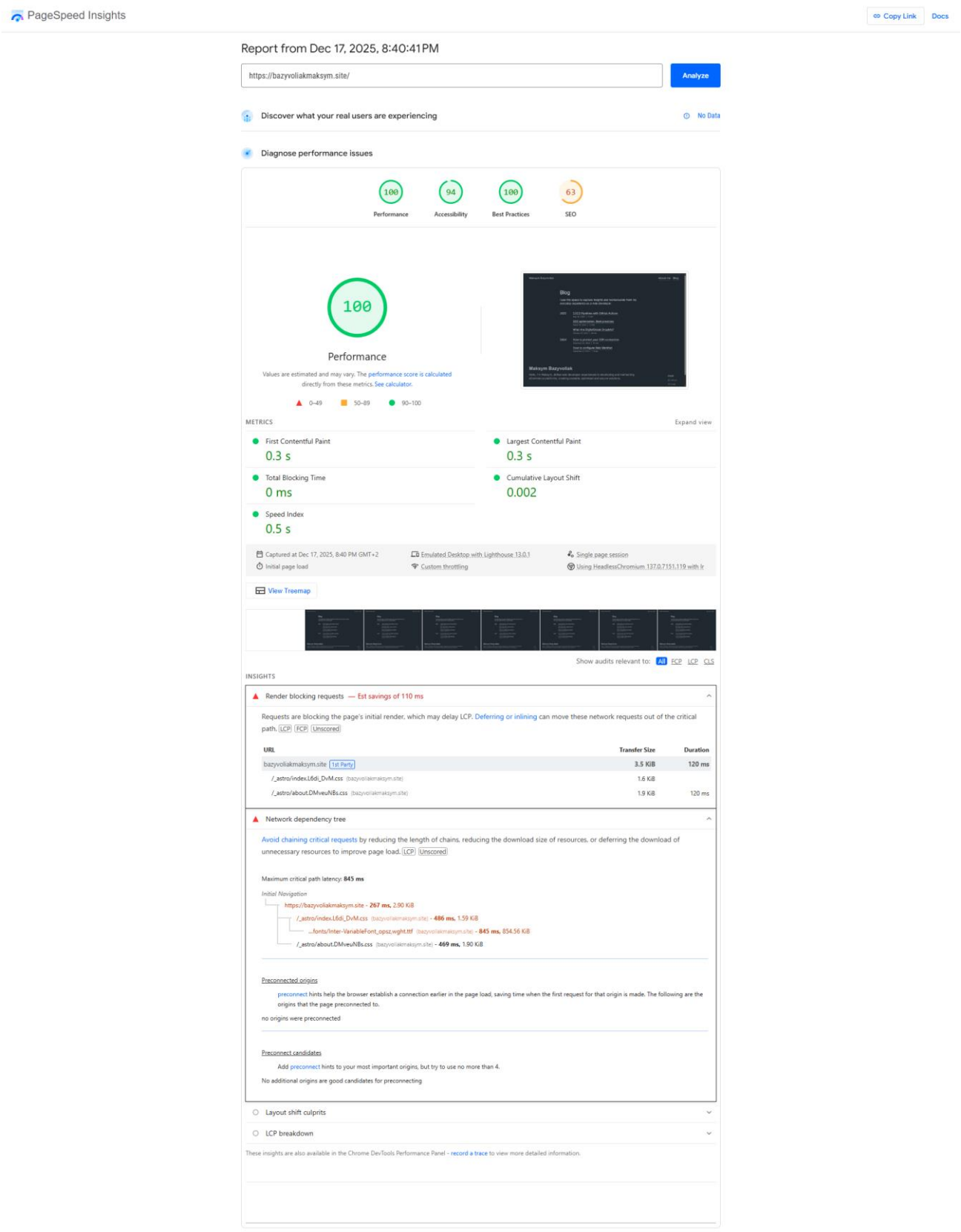

Рис. 6. Репорт швидкодії «Pagespeed Insights» PWA застосунку

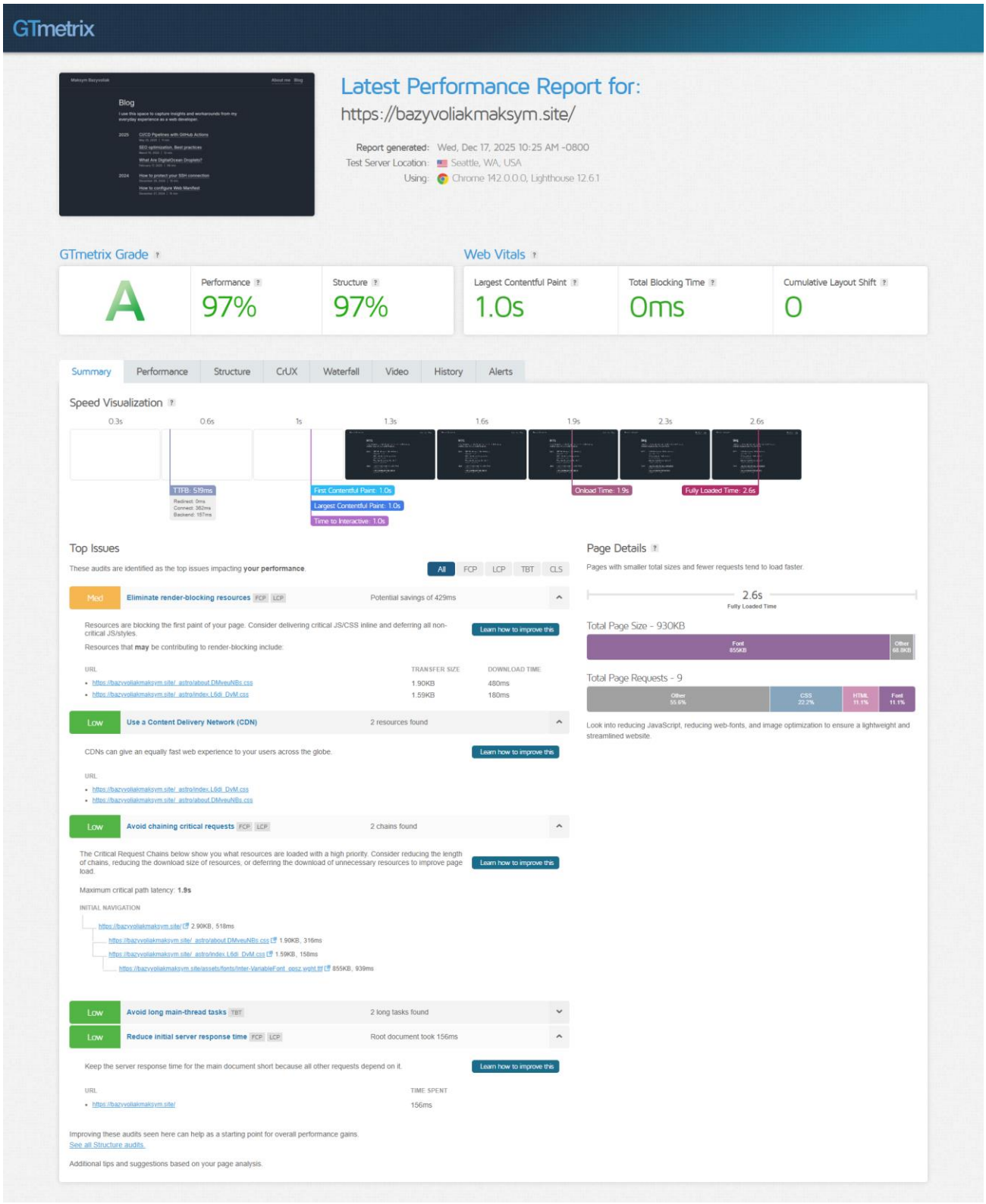

Рис. 7. Підсумки репорту швидкодії «GTmetrix» PWA застосунку

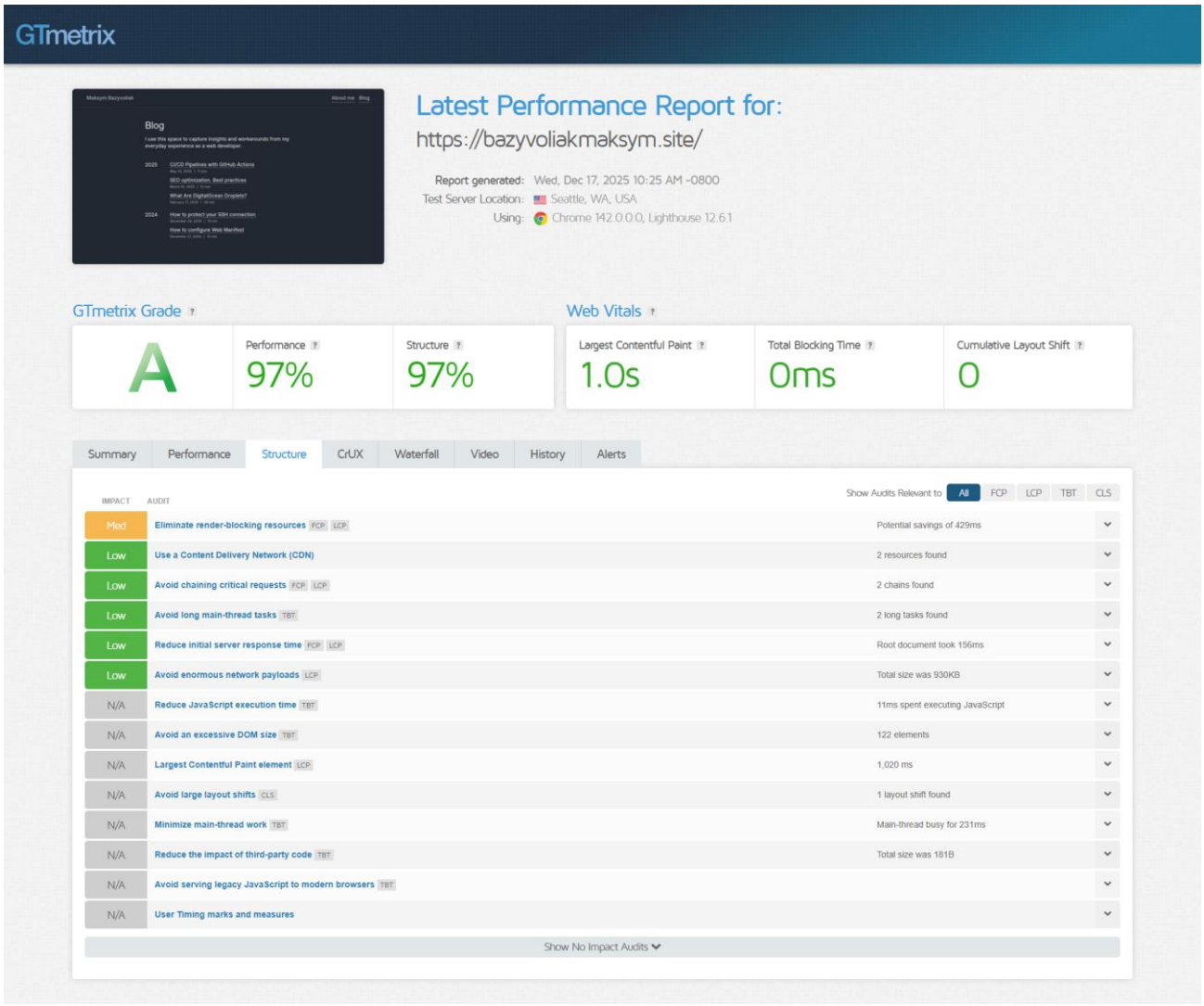

Рис. 8. Репорт структури «GTmetrix» PWA застосунку

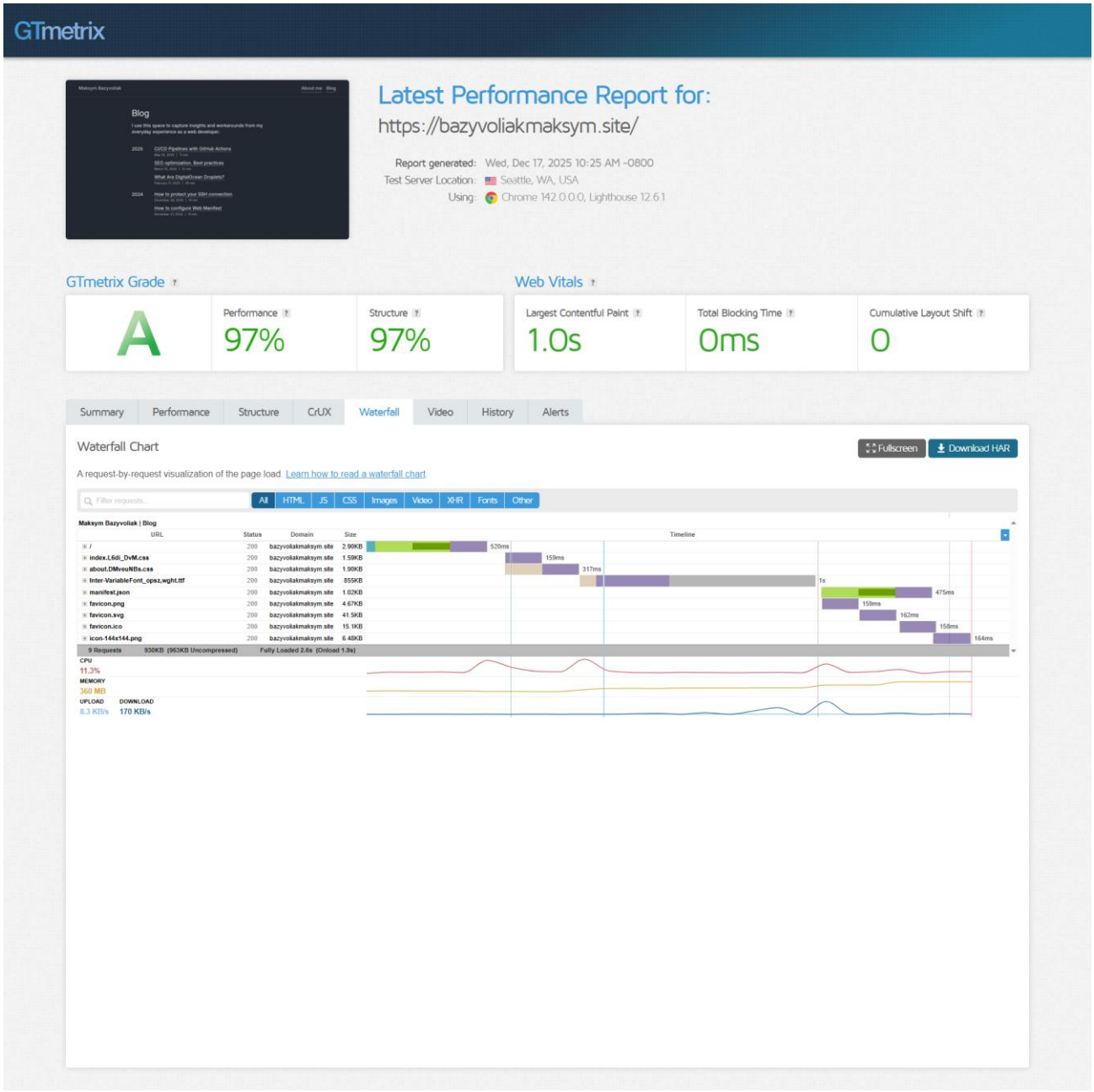

Рис. 9. Репорт «waterfall» «GTmetrix» PWA застосунку

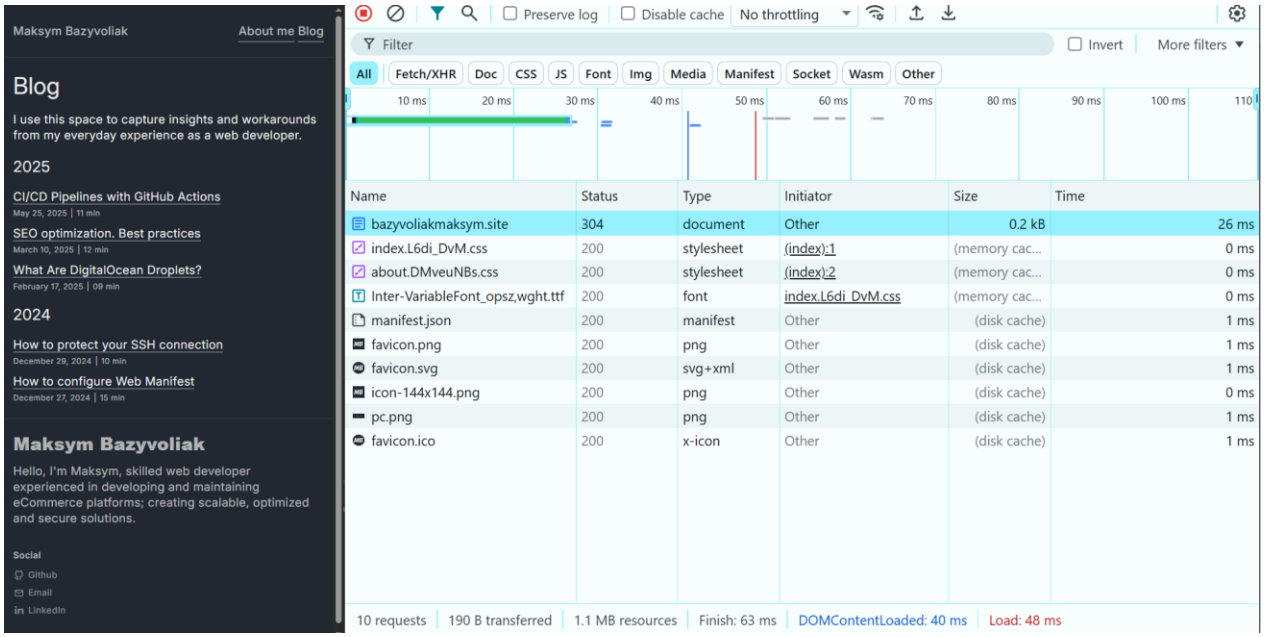

Рис. 10. Перевірка кешованих ресурсів та їх час завантаження в Google Chrome Dev Tools «Network» tab PWA застосунку

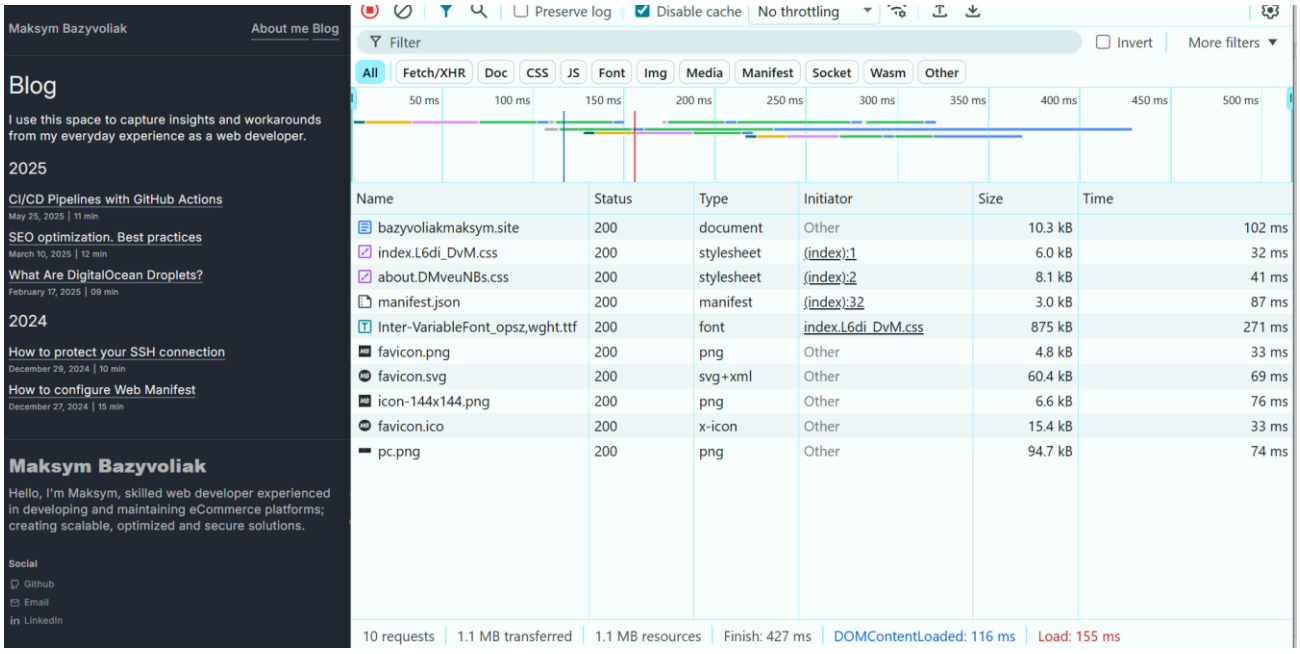

Рис. 11. Перевірка часу завантаження PWA застосунку в Google Chrome Dev Tools «Network» tab без використання браузерного кешування