СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет ___інформаційних технологій та електроніки_____

Кафедра _____інформаційних технологій та програмування_____

Пояснювальна записка

до магістерської дипломної роботи

_____магістр_____
(освітньо-кваліфікаційний рівень)

на тему: ___Оптимізація продуктивності веб-додатків на Next.js
_____

Виконав: студент 2 курсу, групи_ІСТ-23дм_
126 «Інформаційні системи та технології___
(шифр і назва спеціальності)
_____Бас А. В._____
(прізвище та ініціали)
Керівник _____Дьомін М.К._____
(прізвище та ініціали)
Рецензент_____Ратов Д.В._____
(прізвище та ініціали)

Київ – 2024 року

СХІДНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВОЛОДИМИРА ДАЛЯ

Факультет ___інформаційних технологій та електроніки_____
Кафедра _____інформаційних технологій та програмування_____
Освітньо-кваліфікаційний рівень __магістр_____
Спеціальність 126 «Інформаційні системи та технології»_____
(шифр і назва спеціальності)

ЗАТВЕРДЖУЮ
Завідувач кафедри ІТП
_____ д.т.н., доц. Захожай О.І.
(підпис)
« ___ » _____ 2024 р.

ЗАВДАННЯ

на магістерську дипломну роботу студенту
_____Бас Андрій Васильович_____
(прізвище, ім'я, по батькові)

1.Тема роботи: Опитмізація продуктивності веб-додатків на Next.js
_____
керівник роботи доцент, к.т.н. Дьомін Максим Костянтинович ,
(вчене звання, науковий ступінь, прізвище, ім'я, по батькові)
затверджені наказом університету від «6» грудня 2024 року № 361/15.15-С
2. Строк подання студентом роботи: 13 грудня 2024 р._____
3. Вихідні дані до роботи: Матеріали науково-дослідної практики, науково-методична література; дані інтернет-мережі . _____
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

4.1 Вступ

4.2 Аналітичний огляд питання оптимізації продуктивності веб-додатків (огляд публічних джерел інформації)

4.3 Основна частина, в якої висвітлити методи, які будуть використовуватися для реалізації проєкту.

4.4 Практична частина – огляд технологій, які використовуються під час реалізації проєкту.

4.4 Висновки

4.5 Перелік використаних джерел

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
_____

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали, посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв
Розділ 1. Методи та метрики для оцінки продуктивності			
Розділ 2. Огляд аспектів продуктивності веб-додатків на Next.js			
Розділ 3. Розробка алгоритму для оптимізації веб-додатків на Next.js			

7. Дата видачі завдання 20 жовтня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів дипломної роботи	Строк виконання етапів роботи	Примітка
1.	Одержання завдання на виконання роботи	20.10.2024	
2.	Укладання і погодження з керівником плану і етапів виконання роботи	24.10.2024	
3.	Узагальнення даних літературних джерел	28.10.2024	
4.	Аналіз шляхів виконання завдання. Вибір і погодження з керівником оптимального шляху виконання завдання	01.11.2024	
5.	Аналіз технічних засобів та існуючих систем	07.11.2024	
6.	Реалізація практичної частини завдання	24.11.2024	
7.	Укладання, оформлення та погодження пояснювальної записки з керівником	06.12.2024	
8.	Надання пояснювальної записки на кафедру	10.12.2024	
9.	Підготовка доповіді та презентації	13.12.2024	

Студент _____Бас А.В._
(підпис) (прізвище та ініціали)

Керівник роботи _____Дьомін М.К.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Дипломна робота складається зі вступу, трьох розділів, загальних висновків, списку використаних джерел та додатків. Загальний обсяг основного тексту складає 58 сторінок, включає 22 рисунки, 2 таблиці, список джерел на 25 найменувань та 12 сторінок додатків.

У роботі досліджено сучасні підходи до оптимізації продуктивності веб-додатків на основі Next.js, зокрема швидкість завантаження сторінок, рендеринг контенту, оптимізацію зображень та ефективне кешування.

Актуальність теми роботи. Швидке завантаження сторінок є важливим для формування позитивного досвіду користувачів, що підвищує конверсію та прибутковість бізнесу. Next.js пропонує широкий набір інструментів для вирішення цих задач, однак їх правильне застосування вимагає системного підходу.

Об'єкт дослідження – веб-додатки на основі Next.js.

Предмет дослідження є методи та підходи до оптимізації продуктивності та алгоритми вибору стратегій для Next.js-додатків.

Мета роботи. Розробити алгоритм вибору оптимальних підходів для оптимізації продуктивності веб-додатків на Next.js, враховуючи специфіку бізнес-завдань і цільової аудиторії.

Наукова новизна. У роботі запропоновано алгоритм вибору методів оптимізації, що дозволяє розробникам ефективно інтегрувати найкращі практики в свої проєкти на Next.js.

Практичне значення. Результати роботи можуть бути використані для зменшення часу завантаження сторінок, підвищення продуктивності та загальної зручності користування.

Методи дослідження. У процесі дослідження застосовано загальнонаукові методи аналізу та синтезу, порівняльний та системний підхід, а також методи узагальнення та класифікації теоретичних і практичних матеріалів.

Ключові слова. Продуктивність веб-додатків, Next.js, оптимізація, SSR, SSG, CSR, рендеринг, кешування, метрики продуктивності.

ABSTRACT

The diploma thesis consists of an introduction, three chapters, general conclusions, a list of references, and appendices. The main text covers 58 pages, includes 22 figures and 2 tables, references to 25 sources, and 12 pages of appendices.

Research focus. This study examines modern approaches to optimizing the performance of web applications built with Next.js. Special attention is given to page loading speed, content rendering strategies, image optimization, and efficient caching techniques.

Relevance of the topic. Rapid page loading is crucial for creating a positive user experience, thereby increasing conversion rates and overall business profitability. Although Next.js offers a wide range of tools to achieve these goals, their proper application requires a systematic approach.

Object of the research. Web applications developed using the Next.js framework.

Subject of the research. Methods and approaches to performance optimization, as well as strategy selection algorithms for Next.js-based applications.

Aim of the research. To develop an algorithm for selecting optimal approaches to enhance the performance of Next.js-based web applications, taking into account the specific business requirements and target audience.

Scientific novelty. This study proposes an algorithm for selecting optimization methods that enables developers to effectively integrate best practices into their Next.js projects.

Practical significance. The results of this work can be employed to reduce page load times, increase application performance, and improve overall user convenience.

Research methods. The study employed general scientific methods of analysis and synthesis, a comparative and systematic approach, as well as methods of generalization and classification of theoretical and practical materials.

Keywords. Web application performance, Next.js, optimization, SSR, SSG, CSR, rendering, caching, performance metrics.

ЗМІСТ

ВСТУП

Веб-додатки стали ключовою частиною сучасного цифрового середовища, яке постійно розвивається. Швидкий доступ до інформації, зручний інтерфейс і висока продуктивність – це ті аспекти, які визначають успіх додатків у конкурентному середовищі. Затримки у завантаженні сторінок можуть призводити до втрати користувачів, зниження рівня задоволення та, як наслідок, зменшення прибутковості для бізнесу.

Фреймворк Next.js пропонує інструменти, які дозволяють досягти високих показників продуктивності, зокрема за рахунок ефективного рендерингу, оптимізації зображень і стратегій кешування. Проте, щоб отримати максимальну користь від цих можливостей, розробникам необхідно глибоко розуміти методи оптимізації та їхнє застосування до специфічних задач. Саме це питання є центральним у цій роботі.

Актуальність теми. Швидкість роботи веб-додатків безпосередньо впливає на їх ефективність і користувацький досвід [15]. Згідно з дослідженнями, кожна секунда затримки завантаження сторінки може знизити конверсію до 20%, а навіть 100 мс затримки можуть обійтися в 1% від продажів [13] [17]. Це особливо важливо для бізнесів, які працюють у високо-конкурентному середовищі, де користувачі очікують швидкої реакції [18]. А для мобільних користувачів швидкість має критичне значення [19].

Next.js є популярним вибором для створення веб-додатків завдяки його можливостям рендерингу та підтримці сучасних методів оптимізації. Проте без належного системного підходу ці можливості можуть залишитися нереалізованими, що зменшує переваги фреймворку.

Дослідження продуктивності веб-додатків та розробка ефективних алгоритмів вибору стратегій оптимізації є важливими як для практикуючих розробників, так і для компаній, які прагнуть залишатися конкурентоспроможними.

Об’єкт і предмет дослідження. Об’єктом дослідження є веб-додатки, створені на основі фреймворку Next.js. Цей інструмент надає різноманітні методи для роботи з контентом, зображеннями, даними, а також забезпечує гнучкість у виборі підходів до рендерингу.

Предмет дослідження включає методи оптимізації продуктивності та алгоритми вибору стратегій, які забезпечують ефективне використання можливостей фреймворку.

Мета і завдання роботи. Метою роботи є розробка алгоритму, який допоможе розробникам обирати найкращі підходи до оптимізації

продуктивності для веб-додатків на Next.js, враховуючи специфіку задач і потреб бізнесу.

Для досягнення мети необхідно вирішити наступні завдання:

1. Проаналізувати сучасні метрики вимірювання продуктивності та підходи до оптимізації веб-додатків.
2. Вивчити можливості фреймворку Next.js у контексті оптимізації.
3. Оцінити ефективність різних стратегій рендерингу, кешування та обробки зображень.
4. Розробити алгоритм вибору оптимальних методів для підвищення продуктивності.
5. Провести тестування запропонованого алгоритму на реальних прикладах.

Наукова новизна. У роботі запропоновано алгоритм, який систематизує підходи до оптимізації продуктивності веб-додатків на основі Next.js. Він враховує такі аспекти, як тип сторінок, вимоги до часу завантаження та обсяги трафіку. Запропоновані рішення дозволяють підвищити ефективність використання фреймворку та досягти оптимальних показників продуктивності.

Практичне значення. Результати роботи будуть корисними для розробників, які працюють з Next.js. Вони можуть використовувати запропонований алгоритм для покращення продуктивності веб-додатків, що сприятиме зниженню часу завантаження сторінок, підвищенню зручності використання та досягненню бізнес-цілей.

Методи дослідження. Для досягнення поставлених цілей у роботі застосовувалися як теоретичні, так і практичні методи дослідження. Зокрема, використовувалися інструменти Lighthouse для оцінки продуктивності, а також методи профілювання у DevTools для виявлення вузьких місць у продуктивності. Практична частина включала тестування різних підходів на реальних проєктах, що дозволило перевірити ефективність запропонованих рішень.

Таким чином, ця робота спрямована на вирішення практичних і теоретичних проблем, пов’язаних із продуктивністю веб-додатків, і пропонує інструменти для їх подолання.

1 МЕТОДИ ТА МЕТРИКИ ДЛЯ ОЦІНКИ ПРОДУКТИВНОСТІ

1.1 Огляд основних метрик продуктивності веб-додатків

Продуктивність веб-додатків – це ключовий параметр, що визначає їхню зручність для користувачів. Для оцінки продуктивності використовуються специфічні метрики. Вони дозволяють розробникам визначати, наскільки швидко додаток реагує на дії користувача та як швидко він стає функціональним. Нижче розглянемо найпоширеніші із них.

1.1.1 First Contentful Paint (FCP)

FCP визначає, скільки часу проходить до моменту, коли користувач побачить перший елемент на сторінці. Це може бути текст, логотип або кнопка. Наприклад, на інформаційному сайті FCP зазвичай означає появу основного заголовка.

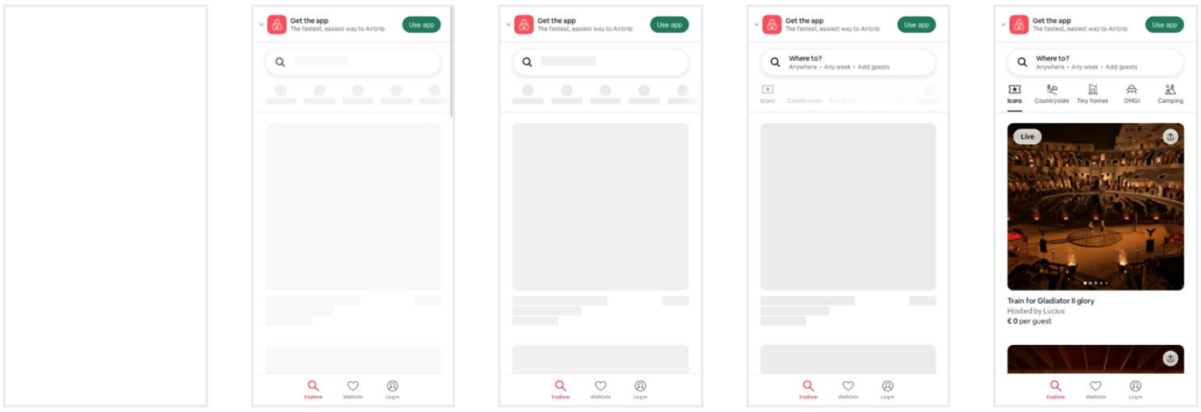

Рисунок 1.1 — Ілюстрація FCP на прикладі сайту airbnb.com

Оптимальне значення FCP – менше 1.8 секунди. Перевищення цього часу може викликати роздратування користувачів. Проблеми із FCP часто виникають через великий обсяг CSS або JavaScript на початковій сторінці. [3]

1.1.2 Largest Contentful Paint (LCP)

LCP вимірює час, необхідний для завантаження найбільшого видимого елементу сторінки. Це може бути велике зображення, відео чи заголовок. Наприклад, на сторінці магазину найбільшим елементом часто є зображення товару. Якщо це зображення завантажується повільно, користувач може втратити інтерес і залишити сторінку.

LCP до 2.5 секунд вважається хорошим показником. Проблеми з LCP виникають через:

- Затримки серверної відповіді.
- Відсутність оптимізації зображень.
- Використання шрифтів із великою затримкою завантаження.

1.1.3 First Input Delay (FID)

FID показує, скільки часу потрібно додатку для обробки першої дії користувача. Наприклад, коли користувач натискає кнопку “Додати в кошик”, а сторінка довго не реагує, це свідчить про високий FID.
Ілюстрація проблемного FID:

- Користувач натискає кнопку “Відкрити меню”.
- Затримка між натисканням і реакцією 300 мс.
- Це створює негативний досвід.

Оптимальне значення FID – менше 100 мс. Поганий FID зазвичай пов'язаний із блокуванням головного потоку через важкі JavaScript-скрипти. [2]

1.1.4 Time to Interactive (TTI)

TTI показує, коли сторінка стає повністю інтерактивною. Це означає, що користувач може взаємодіяти з усіма її елементами.
Наприклад, на сторінці блогу TTI означає, що користувач може відкривати вкладки меню чи прокручувати текст. Якщо TTI занадто великий, це створює враження “замороженості”, що негативно впливає на користувача [23].
TTI до 5 секунд вважається нормальним. Затримки зазвичай викликані:

1. Виконанням важких скриптів.
2. Завантаженням великої кількості зовнішніх ресурсів.

1.1.5 Total Blocking Time (TBT)

TBT вимірює, як довго сторінка залишається заблокованою і не реагує на дії користувача. Він обчислюється як сумарний час блокування головного потоку браузера між FCP і TTI.
Наприклад:

- FCP – 1.2 с, TTI – 4.5 с.
- TBT становить 3.3 секунди, що є критичним показником.

Оптимальний TBT – менше 300 мс. Високий TBT свідчить про необхідність оптимізації JavaScript.

1.1.6 Cumulative Layout Shift (CLS)

CLS визначає, наскільки сильно змінюється розташування елементів сторінки під час її завантаження. Наприклад, якщо користувач намагається натиснути кнопку, а вона раптово зсувається вниз через завантаження банера, це викликає дискомфорт. [5]

CLS менше 0.1 вважається добрим показником. Щоб знизити CLS, розробники зазвичай:

- Вказують розміри зображень заздалегідь.
- Використовують правильні атрибути для медіафайлів.

1.1.7 Порівняльна таблиця

Нижче наведена таблиця метрик та їх оптимальних значень.

Метрика	**Оптимальне значення**	**Пояснення**
FCP	< 1.8 с	Час до першого відображення контенту
LCP	< 2.5 с	Час до завантаження основного елементу
FID	< 100 мс	Час відповіді на першу дію користувача
TTI	< 5 с	Час до повної інтерактивності
TBT	< 300 мс	Сумарний час блокування потоку
CLS	< 0.1	Стабільність макету сторінки

Таблиця 1.1 Метрики та їх оптимальні значення

Метрики продуктивності надають розробникам цінну інформацію про вузькі місця веб-додатків. [12] Використовуючи їх, можна побудувати ефективний план оптимізації. Наприклад, зменшення TBT і FID одночасно покращить взаємодію з користувачем, а оптимізація LCP знизить час завантаження основного контенту. Це особливо актуально для мобільної комерції, де швидкість визначає конкурентоспроможність [22]. У подальших розділах буде розглянуто інструменти та методи для досягнення цих показників.

1.2 Інструменти для вимірювання продуктивності

Оцінка продуктивності веб-додатків є важливим етапом у процесі їх розробки. Для цього існують спеціальні інструменти, які дозволяють швидко знайти проблеми, що впливають на швидкість завантаження сторінки, її інтерактивність та зручність для користувача. Найбільш популярними серед них є Lighthouse, DevTools і PageSpeed Insights [16].

1.2.1 Lighthouse

Lighthouse — це відкритий інструмент від Google для автоматизованого аналізу продуктивності веб-додатків [3]. Його можна запустити через браузер Chrome або командний рядок. Lighthouse оцінює продуктивність за різними показниками, включаючи швидкість завантаження, доступність та SEO. Lighthouse створює звіт, який складається з декількох розділів:

3. Performance (Продуктивність). Аналізує час завантаження сторінки, її інтерактивність і стабільність макету.
4. Accessibility (Доступність). Визначає проблеми, які можуть завадити користувачам із особливими потребами.
5. Best Practices (Найкращі практики). Перевіряє безпеку та коректність роботи сторінки.
6. SEO. Аналізує базові параметри для покращення позицій сторінки в пошукових системах.

Звіт Lighthouse виглядає як кругова діаграма з оцінками за різними категоріями (Performance, Accessibility, тощо).

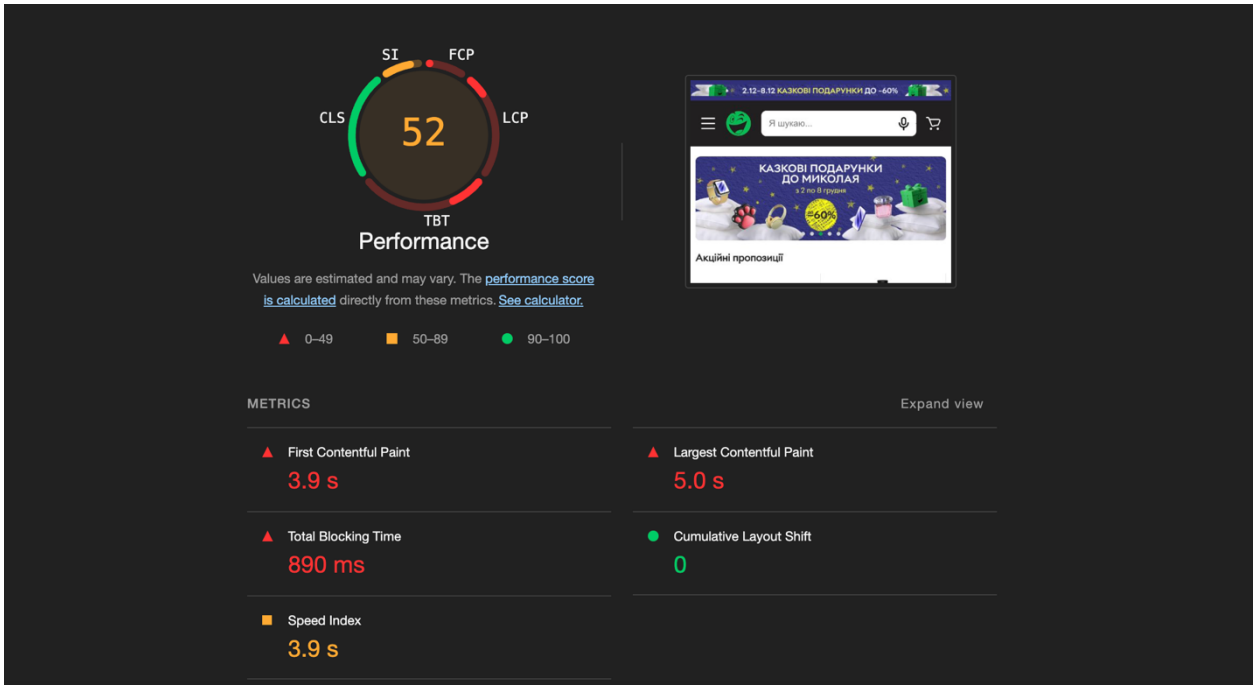

Рисунок 1.2 — Приклад використання Lighthouse для інтернет-магазину Rozetka

1.2.2 DevTools

DevTools — це набір інструментів розробника, вбудованих у браузер Chrome. Він дозволяє аналізувати продуктивність у реальному часі, тестувати сторінки та знаходити проблеми.

Основні функції DevTools для продуктивності:

1. Performance Tab. Відображає таймлайн роботи сторінки. Тут можна побачити, як виконується JavaScript, завантажуються ресурси та як взаємодіє сторінка.
2. Network Tab. Показує, які файли завантажуються, їхній розмір та час завантаження.
3. Coverage Tab. Дозволяє знайти невикористаний CSS і JavaScript, щоб зменшити обсяг ресурсів.

Гіпотетичний приклад використання DevTools: На сторінці блогу користувачі скаржаться на повільну взаємодію з меню. Розробник відкриває Performance Tab і бачить, що скрипт для меню блокує головний потік на 1.5 секунди. Оптимізація скрипта зменшує час блокування до 300 мс, і проблема зникає.

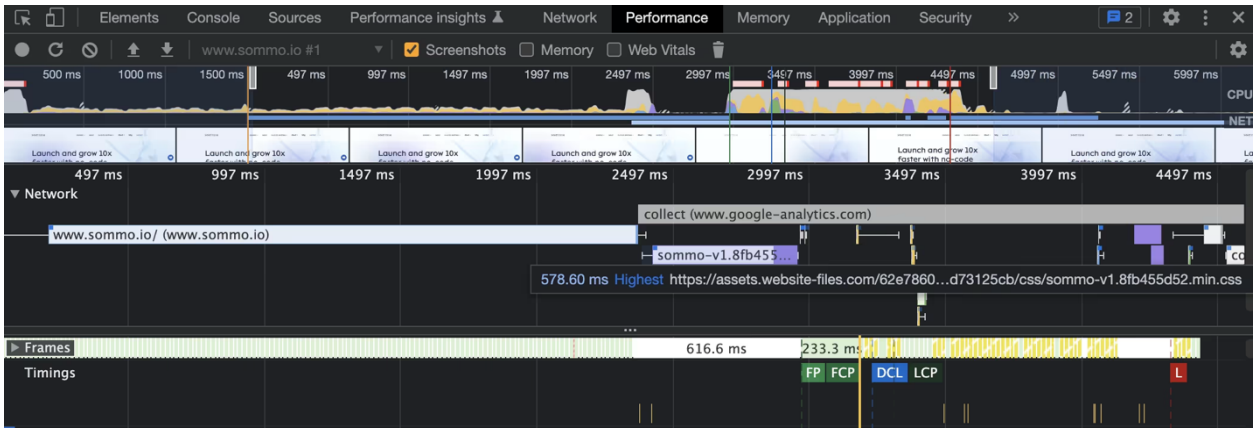

Рисунок 1.3 — Скріншот вкладки Performance із DevTools

1.2.3 PageSpeed Insights

PageSpeed Insights — це інструмент від Google для оцінки швидкодії сторінки. Він надає звіти окремо для мобільних пристроїв і десктопів, що дозволяє розробникам оптимізувати сторінку для різних платформ.

Особливості PageSpeed Insights:

1. Оцінює продуктивність за ключовими метриками, такими як FCP, LCP і CLS.
2. Надає конкретні рекомендації, наприклад, “Оптимізуйте зображення” або “Використовуйте кешування браузера”.
3. Інтегрується з Lighthouse, тому звіти схожі за структурою.
4. Надає додатковий звіт щодо “Core Web Vitals Assessment” - усереднене значення основних метрик за останні 28 днів.

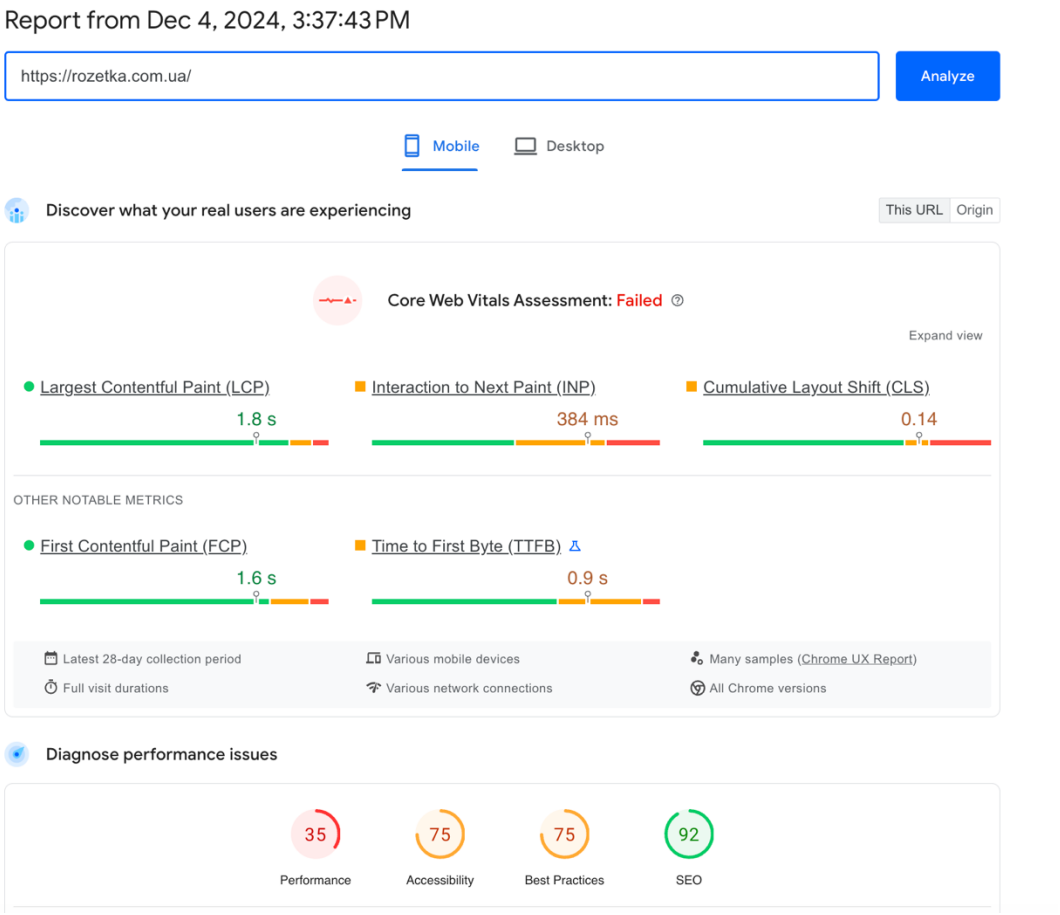

Рисунок 1.4 — Звіт PageSpeed Insights для інтернет-магазину Rozetka

Lighthouse, DevTools і PageSpeed Insights — це три основні інструменти, які допомагають розробникам оцінити продуктивність веб-додатків. Кожен із них має свої сильні сторони та підходить для різних завдань. Використання цих інструментів у комплексі дозволяє точно визначити проблеми та знайти оптимальні рішення для їх усунення.

1.3 Короткий огляд фреймворку Next.js

Next.js — це фреймворк для JavaScript, створений на основі React. [1] Його основна мета — спрощення розробки сучасних веб-додатків із підтримкою серверного рендерингу, статичної генерації та інших оптимізацій. Завдяки готовим інструментам та функціям, Next.js дозволяє розробникам створювати швидкі та зручні додатки. Остання (на грудень 2024 р.) версія Next.js 15 додала значні оновлення, які зробили фреймворк ще більш ефективним для створення продуктивних веб-додатків. Однією з ключових змін є App Router, що використовує новітні можливості React, такі як серверні компоненти (Server Components), потокова передача контенту (Streaming) та Suspense. Ці функції суттєво змінюють підхід до розробки додатків і дозволяють створювати більш швидкі та інтерактивні інтерфейси.

Нижче розглянемо деякі головні частини Next.js що будуть для нас релевантні.

1.3.1 Рендеринг сторінок: SSR, SSG та ISR

Однією з ключових переваг Next.js є гнучкість у підходах до рендерингу сторінок. [6] Фреймворк підтримує кілька методів, кожен із яких має свої переваги для різних типів проєктів:

1. Server-Side Rendering (SSR)
   При SSR сторінки рендеряться на сервері під час кожного запиту. Це забезпечує актуальні дані для користувачів, але може бути повільнішим для великих додатків із високим трафіком.
   Приклад: Онлайн-магазин, де користувачі завжди бачать оновлені дані про ціни та товари.
2. Static Site Generation (SSG)
   У SSG сторінки рендеряться під час побудови додатку, а не під час запиту. Це значно пришвидшує завантаження, але дані можуть бути неактуальними.
   Приклад: Блог із рідкими оновленнями статей.
3. Incremental Static Regeneration (ISR)
   ISR дозволяє оновлювати окремі сторінки після їхньої побудови. Це поєднує швидкість SSG і актуальність SSR.
   Приклад: Новинний сайт, де оновлення статей відбувається раз на кілька хвилин.

1.3.2 Система маршрутизації

Next.js має вбудовану систему маршрутизації, яка базується на файловій структурі проєкту. App Router у Next.js 15 замінив попередню систему маршрутизації, ґрунтуючись на новій архітектурі React. Він надає розробникам гнучкість і дозволяє:

- Використовувати вкладені маршрути (Nested Routes), де кожен маршрут має власний макет.
- Оновлювати контент поступово завдяки потоковій передачі.
- Розподіляти логіку рендерингу між клієнтом і сервером.

Переваги App Router:

1. Покращення продуктивності. Серверний рендеринг (SSR) поєднується з потоковою передачею, що забезпечує швидше завантаження сторінок.
2. Зручність у розробці. Вкладені маршрути дозволяють розробникам створювати багаторівневі інтерфейси, розділяючи логіку сторінок.

3. Гнучкість макетів. Кожен маршрут може використовувати власний макет або спільний макет для кількох сторінок.

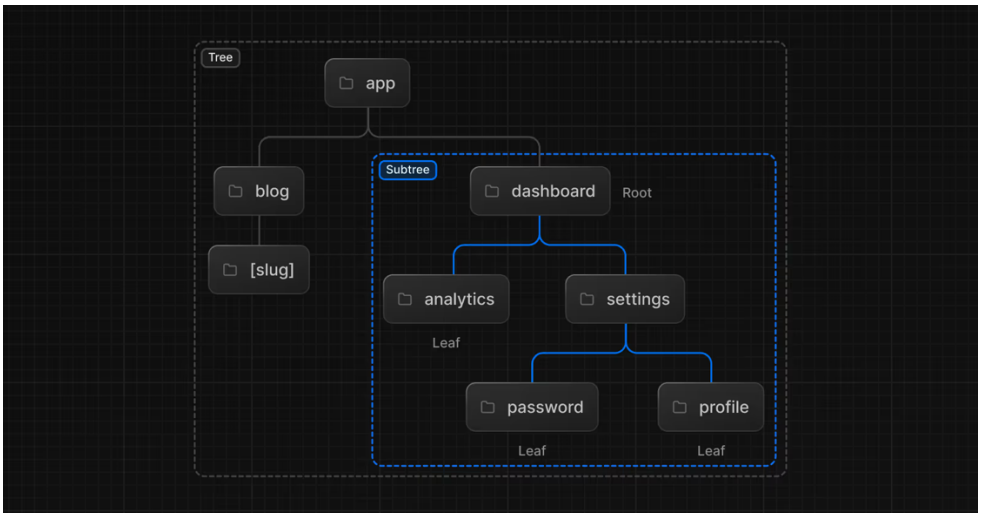

Рисунок 1.5 — Приклад використання App Router

1.3.3 Оптимізація зображень

Компонент “Image” в Next.js автоматично оптимізує зображення. Він підтримує функції:

1. Lazy loading — завантаження лише тих зображень, які бачить користувач.
2. Responsive images — підбір відповідного розміру зображення для різних екранів.
3. WebP — використання сучасного формату для зменшення розміру файлів.

Окремий компонент для картинок в Next.js повністю виправданий, так як зображення є невід’ємною (і основною) майже усіх сучасних вебсайтів, тому їх ефективне використання є ключем до високої продуктивності сторінки.

1.3.4 Система кешування

Next.js підтримує кешування на різних рівнях:

1. Edge-кешування через Content Delivery Networks (CDN). Це забезпечує швидкий доступ до сторінок із будь-якої точки світу.
2. Кешування на сервері. Відповіді на часті запити зберігаються для повторного використання.
3. Кешування у браузері. Завантаження та віддача даних та сторінок уже на браузері, для забезпечення швидкого переходу між сторінками (наприклад, під час навігації вперед-назад).

1.3.5 Розподіл JavaScript (Code Splitting)

Фреймворк автоматично ділить JavaScript на менші частини. Завдяки цьому користувач завантажує лише той код, який потрібен для поточного маршруту. Це зменшує час першого завантаження. Це дозволяє у багатосторінковому додатку завантажувати лише код для сторінки, яку відкриває користувач, а не для всього сайту (що може бути дуже великим).

1.3.6 Підтримка TypeScript

Next.js має вбудовану підтримку TypeScript. Це дозволяє писати більш стабільний код, уникаючи багатьох помилок під час розробки. Конфігурація мінімальна: достатньо створити файл “tsconfig.json”, і фреймворк автоматично розпізнає TypeScript. Це також дає перевагу у великих проєктах, де працює кілька команд, допомагаючи уникнути конфліктів у типах даних та параметрах API, модулів та функцій.

1.3.7 Вбудовані API-роути та Серверні дії (Server Actions)

Next.js дозволяє створювати серверні API-роути без додаткових серверів. Файли із назвою “route.ts” автоматично стають кінцевими точками API. Це корисно для інтеграції з базами даних або зовнішніми сервісами.
У Next.js 15 також з’явилися Серверні дії (Server Actions), що спрощують виконання серверної логіки, зменшуючи потребу у створенні окремих API-роутів. Це робить код більш зрозумілим і зменшує кількість передач даних між клієнтом і сервером. [1]

```
'use server';

export async function addProduct(data) {
  await fetch('https://api.example.com/add', {
    method: 'POST',
    body: JSON.stringify(data),
  });
}
```

Рисунок 1.6 — Приклад Серверної дії

1.3.8 Підтримка Middleware

Middleware у Next.js використовується для обробки запитів перед їхнім потраплянням на сервер. Це дозволяє реалізувати редиректи, проксі, аутентифікацію або модифікацію запитів.

1.3.9 Інструменти для розробників

Next.js пропонує такі інструменти:

1. Hot Reloading. Зміни у коді автоматично відображаються в браузері без перезавантаження сторінки.
2. Build Analyzers. Аналізує розмір бандлів і вказує на можливості для оптимізації.

Якщо підсумувати, то Next.js — це фреймворк, що надає широкий набір інструментів для створення швидких та продуктивних веб-додатків. Його особливості, такі як гнучкість рендерингу, автоматична оптимізація зображень і підтримка кешування, роблять його вибором номер один для багатьох розробників. У поєднанні з простотою налаштування та ефективними інструментами, Next.js стає ідеальним рішенням для сучасних проєктів.

1.4 Вибір методології для тестування продуктивності у Next.js

Тестування продуктивності веб-додатків є важливим етапом у забезпеченні їхньої швидкості та зручності для користувачів. Для Next.js, особливо з оновленнями у версії 15, тестування потребує врахування таких аспектів, як серверні компоненти, потокова передача контенту та нові підходи до кешування.

На основі розглянутих інструментів (Lighthouse, DevTools, PageSpeed Insights) та можливостей фреймворку, розроблена методологія включає кілька етапів: підготовку, збір метрик, аналіз проблемних зон і впровадження змін.

1.4.1 Підготовка до тестування

Тестування продуктивності розпочинається з визначення цілей і ключових метрик. Для Next.js це буде:

1. Largest Contentful Paint (LCP): показує, як швидко користувач бачить основний контент.
2. Time to Interactive (TTI): оцінює, коли сторінка стає повністю інтерактивною.
3. Cumulative Layout Shift (CLS): вимірює стабільність макета під час завантаження.

Також необхідно врахувати специфіку рендерингу: чи використовує сторінка SSR, SSG або ISR. Наприклад, сторінки з SSR можуть мати інші вузькі місця у порівнянні зі статично згенерованими сторінками. [7]

Для створення тестового середовища рекомендовано перевірити наступне:

1. Розгорніть додаток у середовищі, яке максимально наближене до продакшну. Використовуйте хостинг із підтримкою CDN.
2. Переконайтеся, що кешування на сервері та клієнті відповідає вашим налаштуванням у Next.js.
3. Вимкніть сторонні плагіни чи розширення браузера, щоб уникнути впливу на результати тестів.

1.4.2 Збір даних про продуктивність

Для Next.js 15 буде використано комбінацію інструментів:

1. Lighthouse: автоматизований звіт про ключові метрики. Наприклад, він може показати, що LCP перевищує 3 секунди через неоптимізовані зображення.
2. DevTools (Performance Tab): для аналізу роботи головного потоку браузера та ідентифікації затримок у виконанні JavaScript.
3. PageSpeed Insights: для оцінки продуктивності на мобільних і десктопних пристроях.

1.4.3 Аналіз проблемних зон

Після збору даних метрики потрібно проаналізувати, щоб визначити вузькі місця:

1. Якщо LCP занадто високий, це може свідчити про неоптимізовані зображення або повільну серверну відповідь.
2. Високий TTI зазвичай пов'язаний із великими JavaScript-бандлами, які можна зменшити за допомогою динамічного імпорту.
3. Проблеми з CLS часто виникають через відсутність фіксованих розмірів зображень або банерів.

Враховуючи специфіку Next.js, для сторінок із серверними компонентами слід перевірити, чи не блокується рендеринг через повільні запити до бази даних або API. Для сторінок з ISR необхідно переконатися, що регенерація працює коректно та не впливає на продуктивність. [8]

1.4.4 Впровадження змін та повторне тестування

Після виявлення проблем слід внести зміни в код. Наприклад:

1. Оптимізація зображень. Використати “next/image” для автоматичного створення адаптивних зображень.
2. Кешування. Налаштувати Edge-кешування для статичних сторінок та перевірте, чи працюють заголовки кешування для динамічних даних.

3. Зменшення JavaScript-бандлів. Розділити код за допомогою динамічного імпорту та переконатися, що непотрібні залежності видалені.

Після внесення змін необхідно повторити тестування, використовуючи ті самі інструменти. Це дозволяє оцінити, наскільки зміни покращили метрики.

1.4.5 Інтеграція автоматизованого тестування

Для великих проєктів варто впровадити автоматизоване тестування продуктивності, що може включати:

1. Lighthouse CI: автоматичне створення звітів про продуктивність під час деплою.
2. GitHub Actions: налаштування скриптів для тестування кожного пул-реквесту.

Це допомагає виявляти регресії на ранніх етапах і підтримувати високі показники продуктивності в довгостроковій перспективі.
В рамках даної роботи це було застосовано, так як розмір проєкту не передбачали такої доцільності.

Якщо підсумувати, то методологія тестування продуктивності для Next.js повинна враховувати особливості фреймворку, включаючи App Router, серверні компоненти та ISR. Комбінація інструментів (Lighthouse, DevTools, PageSpeed Insights) та ретельний аналіз метрик дозволяє ефективно виявляти проблеми та покращувати швидкодію додатків. Використання автоматизації забезпечує стабільність результатів і мінімізує вплив людського фактору. [11]

1.5 Висновки розділу

У цьому розділі було розглянуто ключові метрики, інструменти та підходи до забезпечення продуктивності веб-додатків, а також особливості використання Next.js 15. Метрики, такі як LCP, TTI та CLS, дозволяють чітко виміряти швидкість завантаження, інтерактивність та стабільність макету [20]. Інструменти Lighthouse, DevTools забезпечують детальний аналіз вузьких місць і дозволяють будувати стратегії для їх усунення.

Next.js 15 із новим App Router і підтримкою серверних компонентів значно розширює можливості розробників у створенні продуктивних веб-додатків. Завдяки інтегрованим технологіям, як-от потокова передача контенту та автоматизована оптимізація, фреймворк допомагає вирішувати завдання швидкодії та зручності для користувачів.

Розроблена методологія тестування продуктивності враховує специфіку Next.js та особливості вибору підходів до рендерингу. Ці результати створюють міцну основу для подальшого аналізу та впровадження оптимізацій у наступному розділі, та реалізації алгоритму у 3-ьому розділі роботи [21].

2 ОГЛЯД АСПЕКТІВ ПРОДУКТИВНОСТІ ВЕБ-ДОДАТКІВ НА NEXT.JS

2.1 Стратегії рендерингу

Веб-додатки на Next.js 15 можуть використовувати різні стратегії рендерингу залежно від вимог до контенту, продуктивності та актуальності даних. У цьому розділі розглянемо Static Rendering, Dynamic Rendering, Data Revalidation та Incremental Static Regeneration (ISR). Ці підходи вдосконалено порівняно з попередніми методами (SSG, SSR) завдяки інтеграції серверних компонентів і нових функцій, орієнтованих на продуктивність.

2.1.1 Static Rendering

Static Rendering — це генерація сторінок під час етапу білду. Весь контент сторінки створюється заздалегідь і зберігається у вигляді статичних файлів. Цей підхід ідеально підходить для контенту, який не змінюється або змінюється рідко. Static Rendering інтегровано з серверними компонентами (Server Components), які кешуються та доставляються через CDN для забезпечення максимальної швидкості завантаження. Сторінки рендеряться як статичний HTML із вбудованим React. [1]

```
export default function Page() {
  return <h1>Статична сторінка</h1>;
}
```

Рисунок 2.1 — Приклад статичної сторінки

Переваги Static Rendering:

1. Надзвичайно швидке завантаження сторінок.
2. Мінімальне навантаження на сервер під час запитів.

Недоліки Static Rendering:

1. Фіксованість контенту до наступного білду.

Важливо ідентифікувати сторінки з незмінним контентом (наприклад, сторінки політики конфіденційності чи документації) і обирати Static Rendering як оптимальний варіант.

2.1.2 Dynamic Rendering

Dynamic Rendering — це рендеринг сторінок на сервері під час кожного запиту. У Next.js 15 цей підхід дозволяє використовувати серверні компоненти для створення актуального HTML у реальному часі. Dynamic

Rendering обробляє запити до серверу, створюючи HTML із використанням серверних компонентів і кешуючи його для повторного використання. Це дозволяє оптимізувати роботу з динамічними даними.

```
export const dynamic = 'force-dynamic';

export default function Page() {
  return <h1>Динамічна сторінка</h1>;
}
```

Рисунок 2.2 — Приклад динамічної сторінки

Переваги Dynamic Rendering:

1. Гарантована актуальність контенту для кожного запиту.
2. Підходить для інтерактивних сторінок, які залежать від зовнішніх API чи баз даних.

Недоліки Dynamic Rendering:

1. Вища затримка порівняно зі статичними сторінками через серверний рендеринг.

Необхідно використовувати Dynamic Rendering для сторінок, де критично важлива актуальність даних (наприклад, сторінки профілів користувачів чи панелі адміністрування).

2.1.3 Data Revalidation

Data Revalidation — це автоматичне оновлення кешованого контенту через певний проміжок часу. Сторінки з ревалідацією забезпечують баланс між швидкістю статичних сторінок і динамічністю контенту [4].

Ревалідація налаштовується через параметр “revalidate”, який визначає, як часто кешовані сторінки повинні перевірятися та оновлюватися.

Основні методи:

1. export const revalidate = 60; // Кеш оновлюється раз на 60 секунд
2. export const dynamicParams = true; // Підтримка динамічних параметрів
3. generateStaticParams // генерує параметри для статичнії сторінок на етапі білду.

```
export async function generateStaticParams() {
  const posts = await fetch('https://api.vercel.app/blog').then((re
  return posts.map((post) => ({ id: String(post.id) }));
}
```

Рисунок 2.3 — Приклад реалізації generateStaticParams()

Переваги:

1. Оптимальне використання серверних ресурсів.
2. Висока швидкість завдяки кешуванню.

Недоліки:

1. Можливість короткочасного показу застарілого контенту.

Важливо правильно використовувати Data Revalidation для сторінок із частково оновлюваним контентом, наприклад, сторінок новин або списків товарів, щоб швидко віддавати контент, але в той же час вчасно оновлювати.

2.1.4 Incremental Static Regeneration (ISR)

ISR — це стратегія, яка дозволяє оновлювати окремі сторінки після їхньої початкової генерації. ISR об'єднує переваги Static Rendering та Data Revalidation, забезпечуючи актуальність сторінок без потреби повного білду. ISR використовує функцію revalidate для періодичного оновлення контенту. Це дозволяє зберігати швидкість статичних сторінок і водночас підтримувати актуальність даних.

```
export const revalidate = 120; // Сторінка оновлюється раз на 120 с

export async function generateStaticParams() {
  const posts = await fetch('https://api.vercel.app/blog').then((re
  return posts.map((post) => ({ id: String(post.id) }));
}
```

Рисунок 2.4 — Приклад ISR що оновлюється кожні 120 сек.

Переваги ISR:

1. Швидкість статичного контенту.
2. Можливість оновлення сторінок на вимогу.

Недоліки:

1. Налаштування інтервалів ревалідації потребує додаткових тестувань.

ISR варто використовувати для сторінок із високою динамічністю, які не потребують миттєвого оновлення, наприклад, для блогів чи оглядів продуктів.

У Next.js 15 стратегії рендерингу стали більш адаптивними завдяки серверним компонентам і можливості налаштовувати ревалідацію. Static Rendering забезпечує швидкість, Dynamic Rendering гарантує актуальність, Data Revalidation додає гнучкість, а ISR дозволяє поєднувати переваги статичного й динамічного контенту, надаючи гнучкість, стабільність та можливість зручного тестування таких сторінок [25]. У майбутньому алгоритмі вибір стратегії буде автоматизовано, що зменшить навантаження на сервер і оптимізує досвід користувачів.

2.2 Стратегії кешування для Next.js додатків

У Next.js 15 значно покращено механізми кешування, які тепер дозволяють ефективно контролювати та оптимізувати збереження і оновлення даних. Вони знижують витрати на серверні обчислення та підвищують швидкість завантаження додатків. У цьому розділі розглянемо основні стратегії кешування та їхній вплив на продуктивність. [1]

2.2.1 Request Memoization

Request Memoization — це механізм автоматичної оптимізації запитів fetch, який дозволяє уникнути дублювання запитів у межах одного рендер-процесу React. Перший запит отримує дані з джерела (cache MISS) і зберігає їх у пам’яті. Усі наступні виклики того ж запиту (з однаковими параметрами) у межах одного рендерингу отримують закешовані дані (cache HIT). Після завершення рендерингу кеш очищується.

```
async function getData() {
  const res = await fetch('https://api.example.com/data');
  return res.json();
}

// Перший виклик — cache MISS
const data1 = await getData();

// Наступний виклик — cache HIT
const data2 = await getData();
```

Рисунок 2.5 — Приклад Request Memoization.

Переваги Request Memoization:

1. Зменшення кількості запитів до серверу.
2. Оптимізація використання пам’яті під час рендерингу.

Memoization варто використовувати для уникнення дублювання запитів у межах одного рендерингу сторінки або компонента, що знизить навантаження на сервер.

2.2.2 Data Cache

Data Cache — це вбудований механізм, який дозволяє зберігати результат запитів fetch між різними запитами до сервера.
Next.js зберігає результати запитів у Data Cache, якщо вказано параметри кешування (cache: 'force-cache' або next.revalidate).

```
// Запит із кешуванням на 3600 секунд
const data = await fetch('https://api.example.com/data', {
  cache: 'force-cache',
  next: { revalidate: 3600 },
});
```

Рисунок 2.6 — Приклад Data Cache.

Для ревалідації Data Cache, є 2 методи:

1. Time-based Revalidation. Дані оновлюються автоматично через вказаний інтервал часу.
2. On-Demand Revalidation. Кеш оновлюється вручну за допомогою revalidatePath або revalidateTag.

Переваги Data Cache:

1. Збереження результатів між запитами.
2. Оптимізація для часто змінюваних даних.

Data Cache варто використовувати для збереження даних, які змінюються через задані інтервали або після конкретних подій, таких як оновлення користувачем. [8]

2.2.3 Full Route Cache

Full Route Cache зберігає результат рендерингу цілої сторінки (React Server Component Payload і HTML). Кешування відбувається під час статичного рендерингу (build time) або після ревалідації.

Ця стратегія об’єднує такі підходи такі як Static Site Generation та Incremental Static Regeneration, надаючи гнучкість для розробника.

Переваги Full Route Cache:

1. Миттєве завантаження закешованих сторінок.
2. Зниження витрат на серверні рендеринги.

Цю стратегію варто використовувати для збереження результатів рендерингу сторінок із статичним контентом, такі як статті блогу, сторінки авторів, тощо.

2.2.4 Router Cache

Router Cache зберігає React Server Component Payload для окремих сегментів маршруту на стороні клієнта. Він кешує відвідані маршрути та заздалегідь завантажує майбутні маршрути. Це забезпечує миттєву навігацію між сторінками без повного перезавантаження. Його можна викликати із допомогою методу “router.prefetch('/example');”

Переваги Router Cache:

1. Швидка навігація без повторних запитів.
2. Збереження стану компонентів і браузера.

Router Cache варто використовувати для покращення навігації користувачів у великих додатках із багатьма маршрутами.

4 стратегії описані вище можна візуалізувати на рисунку 2.7:

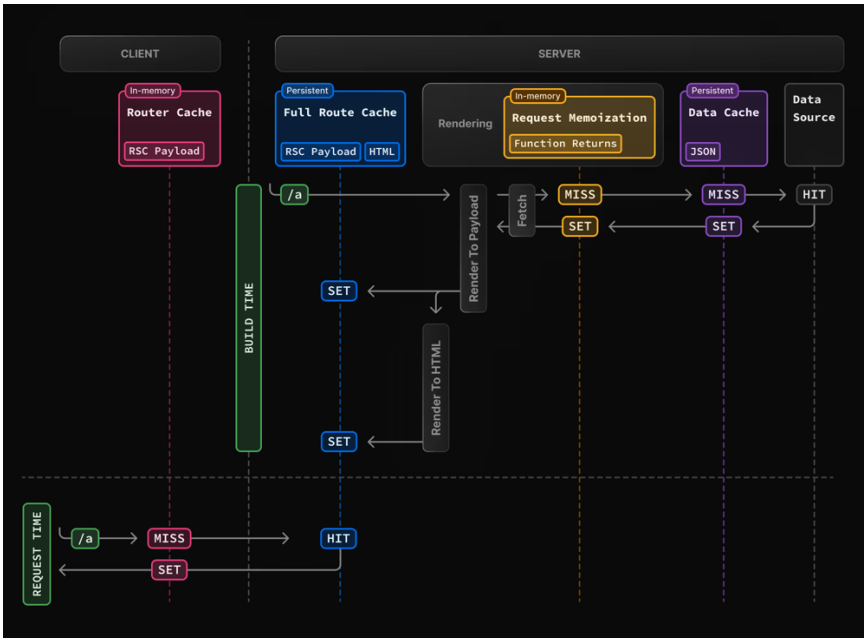

Рисунок 2.7 Стратегії кешування в Next.js.

Next.js 15 пропонує багаторівневі стратегії кешування, які можна гнучко налаштовувати для покращення продуктивності та зниження витрат. Request Memoization, Data Cache, Full Route Cache і Router Cache забезпечують ефективне збереження даних і рендеринг сторінок залежно від потреб додатка. У майбутньому алгоритмі ці стратегії будуть інтегровані для автоматичного вибору оптимального підходу кешування залежно від типу даних і маршруту. [1]

2.3 Оптимізація зображень: використання Next.js Image Component.

Зображення займають значну частину загальної ваги сторінок і можуть суттєво впливати на продуктивність веб-додатків, зокрема на метрику Largest Contentful Paint (LCP). У Next.js компонент Image пропонує потужні інструменти для автоматичної оптимізації зображень, які покращують продуктивність і користувацький досвід. [9]

2.3.1 Можливості Image Component

1. Автоматична оптимізація розміру.
   Next.js автоматично обирає оптимальний розмір зображення залежно від пристрою користувача та підтримує сучасні формати, такі як WebP і AVIF.
2. Збереження візуальної стабільності.
   Компонент запобігає зрушенню макета (Cumulative Layout Shift), автоматично резервуючи місце для зображень під час завантаження.
3. Швидше завантаження сторінок.
   Зображення завантажуються лише тоді, коли вони потрапляють у зону видимості, завдяки використанню вбудованого lazy loading.
4. Гнучкість роботи з ресурсами.
   Компонент підтримує зміну розмірів зображень "на льоту", навіть для віддалених серверів.

2.3.2 Використання Image Component

Image Component можна використовувати як локальні та віддалені ресурси.
Локальні файли автоматично визначають свої ширину та висоту на основі імпорту. Це допомагає уникнути зрушень макета під час завантаження.

```
import Image from 'next/image';
import profilePic from './me.png';

export default function Page() {
  return (
    <Image
      src={profilePic}
      alt="Picture of the author"
      placeholder="blur" // Опціональний ефект розмиття під час зав
    />
  );
}
```

Рисунок 2.8 Локальні зображення.

Для віддалених зображень необхідно вказати width і height вручну, оскільки Next.js не має доступу до файлів під час процесу білду.

```jsx
import Image from 'next/image';

export default function Page() {
  return (
    <Image
      src="https://example.com/image.jpg"
      alt="Example"
      width={500}
      height={500}
    />
  );
}
```

Рисунок 2.9 Віддалені зображення.

У next.config.js можна визначити дозволені домени для оптимізації віддалених зображень, щоб захистити додаток від зловмисного використання. Наприклад:

```
module.exports = { images: { remotePatterns: [ {
    protocol: 'https',
    hostname: 'example.com',
    pathname: '/path/to/images/**',
} ], }, };
```

2.3.3 Важливі параметри Image Component

1. Пріоритет (Priority)

Для зображень, що є ключовими для Largest Contentful Paint (LCP), рекомендується використовувати параметр priority. Це прискорює їхнє завантаження.

2. Адаптивний розмір (Responsive)

Можна задавати розмір зображення залежно від ширини екрану, використовуючи атрибут sizes= "(max-width: 768px) 100vw, 50vw".

3. Розтягнення (Fill)

Атрибут fill дозволяє зображенню заповнювати батьківський елемент.

Якщо підсумувати, для ефективної оптимізації зображень із використанням Image Component необхідно:

1. Автоматична адаптація: Підлаштовувати зображення під розмір екрана користувача, зменшуючи навантаження на мережу.
2. Lazy loading: Для всіх зображень увімкнути завантаження лише в зоні видимості, що покращить продуктивність сторінок.
3. Пріоритет для ключових зображень: Визначати зображення LCP і застосовуватиме до них параметр priority.

4. Безпека: Налаштовувати строгий контроль дозволених доменів для віддалених зображень.

Компонент Image у Next.js забезпечує комплексний підхід до оптимізації зображень. Завдяки автоматичному визначенню розмірів, підтримці сучасних форматів і можливостям lazy loading, він дозволяє значно покращити продуктивність додатків. У майбутньому алгоритмі цей компонент стане основним інструментом для роботи з зображеннями, забезпечуючи баланс між якістю, швидкістю завантаження та зручністю використання.

2.4 Оптимізація CSS та JavaScript

Ефективна робота з CSS і JavaScript є ключовою для досягнення високої продуктивності веб-додатків. У Next.js реалізовано кілька механізмів для оптимізації цих ресурсів, що знижують час завантаження сторінок і підвищують користувацький досвід. [10]

2.4.1 Оптимізація шрифтів через next/font

next/font автоматично оптимізує шрифти, включаючи кастомні, за допомогою самостійного хостингу. Це дозволяє завантажувати шрифти без зовнішніх запитів, зменшуючи затримки та покращуючи приватність.

1. Google Fonts: Шрифти завантажуються під час білду та хостяться разом із іншими статичними ресурсами.
2. Локальні шрифти: Ви можете використовувати власні файли шрифтів із функцією localFont.
3. Variable Fonts: Забезпечують максимальну гнучкість і продуктивність.

```
import { Inter } from 'next/font/google';

const inter = Inter({
  subsets: ['latin'],
  display: 'swap',
});

export default function Layout({ children }) {
  return (
    <html lang="en" className={inter.className}>
      <body>{children}</body>
    </html>
  );
}
```

Рисунок 2.10 Використання next/font.

Переваги next/font:

1. Мінімізація зрушень макета (Cumulative Layout Shift).
2. Відсутність залежності від зовнішніх серверів.
3. Оптимальне завантаження шрифтів для різних пристроїв.

Важливо використовувати самохостинг шрифтів із використанням next/font, щоб уникнути зрушень макета та знизити мережеве навантаження.

2.4.2 Оптимізація JavaScript із next/script

Компонент next/script дозволяє ефективно керувати підключенням сторонніх скриптів, зокрема оптимізуючи їхнє завантаження та виконання. Стратегії завантаження:

1. beforeInteractive: Завантаження до гідратації сторінки.
2. afterInteractive (за замовчуванням): Завантаження після часткової гідратації.
3. lazyOnload: Завантаження під час простою браузера.
4. worker: Експериментальна стратегія, що виконує скрипти у Web Worker.

Переваги next/script:

1. Уникнення блокування основного потоку.
2. Зменшення часу рендерингу сторінки.
3. Гнучке управління скриптами на рівні сторінок чи макетів.

Необхідно аналізувати важливість скриптів і вибирати відповідну стратегію їхнього завантаження для покращення продуктивності.

2.4.3 Мінімізація та tree shaking

Мінімізація видаляє зайвий код із JavaScript, тоді як tree shaking усуває невикористані експорти з модулів.

Next.js автоматично мінімізує JavaScript під час білду.

Tree shaking реалізовано за допомогою Webpack, що дозволяє зменшити розмір фінального бандла.

Важливо проводити регулярний аналіз розміру бандлів із використанням @next/bundle-analyzer для виявлення зайвих залежностей та їх видалення.

2.4.4 Оптимізація імпортів пакетів

Next.js дозволяє оптимізувати імпорти через опцію optimizePackageImports, яка завантажує тільки необхідні модулі. Наприклад: module.exports = { experimental: { optimizePackageImports: ['icon-library'], }, };

Необхідно використовувати цю опцію для зменшення обсягу бандлів, зокрема для великих бібліотек.

Next.js пропонує потужні механізми для оптимізації CSS і JavaScript, зокрема самохостинг шрифтів, мінімізацію коду, розбиття на частини та налаштування завантаження скриптів. У майбутньому алгоритмі ці методи інтегруються для автоматизації оптимізації, зниження розміру бандлів і прискорення рендерингу сторінок.

2.5 Динамічне імпортування модулів

Динамічне імпортування модулів у Next.js дозволяє зменшити розмір початкового бандлу та покращити продуктивність за рахунок завантаження лише потрібних компонентів або бібліотек у момент їх використання. Це є ключовим підходом до оптимізації веб-додатків, особливо для великих проєктів із багатим функціоналом.

Динамічне імпортування — це можливість відкладеного завантаження модулів, які не входять до початкового бандлу. У Next.js це реалізується за допомогою:

- next/dynamic — високорівнева обгортка над React.lazy() і Suspense.
- React.lazy() — для базового lazy loading.

Використовувати динамічне імпортування варто:

1. Для компонентів, що завантажуються рідко (наприклад, модальні вікна).
2. Для зовнішніх бібліотек, які потрібні лише у специфічних сценаріях.
3. Для компонентів, що працюють виключно на клієнтській стороні.

2.5.1 Динамічне імпортування з next/dynamic

Компонент dynamic дозволяє:

1. Завантажувати клієнтські компоненти або бібліотеки лише тоді, коли вони потрібні.
2. Вимикати серверний рендеринг для компонентів (ssr: false).
3. Відображати кастомний компонент під час завантаження.

```
import dynamic from 'next/dynamic';

const LazyComponent = dynamic(() => import('../components/LazyCompo
  ssr: false, // Вимикає серверний рендеринг
  loading: () => <p>Loading...</p>, // Компонент-заповнювач
});

export default function Page() {
  return <LazyComponent />;
}
```

Рисунок 2.11 Використання next/dynamic.

Переваги next/dynamic:

1. Зменшення розміру початкового бандлу.
2. Швидке завантаження основних сторінок.
3. Можливість імпортувати окремі експорти із модулів (import('../components/Module').then((mod) => mod.NamedExport))

Важливо виявляти великі клієнтські компоненти та застосовувати динамічне імпортування для зменшення розміру бандлу.

2.5.2 Динамічне імпортування бібліотек

Динамічне завантаження зовнішніх бібліотек, які використовуються лише в окремих сценаріях, дозволяє зменшити обсяг коду в бандлі.

```
'use client';
import { useState } from 'react';

export default function Page() {
  const [results, setResults] = useState();

  return (
    <input
      type="text"
      placeholder="Search"
      onChange={async (e) => {
        const { value } = e.target;
        const Fuse = (await import('fuse.js')).default; // Динамічни
        const fuse = new Fuse(['Item1', 'Item2']);
        setResults(fuse.search(value));
      }}
    />
  );
}
```

Рисунок 2.12 Динамічний імпорт бібліотек.

Переваги динамічного імпорту бібліотек:

1. Зменшення навантаження на клієнт при початковому завантаженні сторінки.
2. Завантаження лише потрібного коду, що підвищує ефективність.

Важливо ідентифікувати бібліотеки, які використовуються лише у вузьких випадках, і динамічно завантажуватиме їх для оптимізації продуктивності.

Динамічне імпортування модулів у Next.js є потужним інструментом для оптимізації продуктивності додатків. Воно дозволяє завантажувати компоненти та бібліотеки лише тоді, коли вони дійсно потрібні, знижуючи обсяг початкового бандлу та пришвидшуючи завантаження сторінок. У майбутньому алгоритмі цей підхід інтегрується для автоматичного виявлення рідко використовуваних компонентів і бібліотек, оптимізуючи використання ресурсів та підвищуючи швидкість роботи додатка.

2.6 Висновки розділу

У цьому розділі було проаналізовано основні аспекти продуктивності веб-додатків, які реалізовані в Next.js 15. Розглянуто сучасні стратегії рендерингу, кешування, оптимізації зображень, роботи з CSS та JavaScript, а також динамічне імпортування модулів.

Стратегії рендерингу, такі як Static Rendering, Dynamic Rendering, Data Revalidation та Incremental Static Regeneration (ISR), дозволяють гнучко вибирати підходи до створення сторінок залежно від потреб проєкту. Завдяки цьому можна досягти балансу між швидкістю завантаження та актуальністю контенту.

Оновлені стратегії кешування, зокрема Data Cache, Full Route Cache та Router Cache, забезпечують ефективне збереження та оновлення даних. Це зменшує навантаження на сервер і покращує користувацький досвід. Оптимізація зображень із використанням Next.js Image Component дозволяє автоматично адаптувати зображення до різних пристроїв, зменшувати вагу сторінок та запобігати зрушенням макету. Це суттєво покращує метрики, такі як LCP та CLS. [14]

Розглянуті методи оптимізації CSS та JavaScript, включно з мінімізацією, tree shaking, код-сплітингом та lazy loading, дозволяють зменшити розмір бандлів і покращити швидкість роботи додатків. Динамічне імпортування модулів допомагає знижувати витрати на початкове завантаження сторінок, завантажуючи лише необхідні компоненти та бібліотеки.

Усі ці підходи та інструменти створюють міцну основу для розробки алгоритму оптимізації у третьому розділі. Завдяки глибокому аналізу та

інтеграції сучасних технологій Next.js, можна забезпечити високу продуктивність веб-додатків і відповідність сучасним стандартам розробки.

3 РОЗРОБКА АЛГОРИТМУ ДЛЯ ОПТИМІЗАЦІЇ ВЕБ-ДОДАТКІВ НА NEXT.JS

3.1 Інструменти для роботи

Для реалізації алгоритму оптимізації веб-додатків на платформі Next.js, а також створення демо-проєкту, було обрано наступний стек технологій та інструментів:

3.1.1. TypeScript

TypeScript — це надбудова над JavaScript, яка додає типізацію, що допомагає уникнути помилок під час розробки та робить код більш передбачуваним і легшим для підтримки. Його використання дозволяє:

1. Знизити ризики помилок завдяки статичному аналізу коду.
2. Забезпечити кращу читаємість та інтеграцію з інструментами автозавершення в IDE.
3. Створювати масштабовані додатки завдяки можливості визначення типів, інтерфейсів та класів.

3.1.2. Next.js 15

Next.js — це сучасний фреймворк для розробки React-додатків із підтримкою серверного рендерингу (SSR), статичної генерації (SSG) та інкрементального статичного оновлення (ISR). Як центральний фреймворк що досліджувався, переваги та ключові аспекти Next.js було висвітлено у попередніх розділах.

3.1.3. Tailwind CSS

Для роботи з інтерфейсом користувача було обрано Tailwind CSS — утилітарний CSS-фреймворк, який дозволяє швидко створювати адаптивні інтерфейси. Переваги використання Tailwind:

1. Гнучкість: кожен стиль описується через класи, що забезпечує максимальний контроль над компонуванням.
2. Продуктивність: знижує кількість написаного коду завдяки використанню готових класів.
3. Адаптивний дизайн: підтримка медіа-запитів і налаштувань для різних розмірів екранів.
4. Інтеграція з Next.js: дозволяє легко оптимізувати CSS для конкретного рендерингу.

3.1.4. Visual Studio Code

Visual Studio Code (VS Code) обрано як основне середовище розробки через його:

1. Підтримку TypeScript: інтеграція з ESLint та Prettier для забезпечення якісного форматування коду.
2. Потужні розширення: спеціалізовані плагіни для роботи з Next.js і Tailwind.
3. Зручний інтерфейс: підтримка інтерактивного дебагінгу та інтеграція з системами керування версіями, такими як Git.

3.1.5 Демо-проєкт: Сайт із вакансіями для пошуку роботи

Для демонстрації роботи алгоритму оптимізації було обрано проєкт — сайт із вакансіями. Основними функціями цього демо-додатку є:

1. Пошук вакансій за різними категоріями.
2. Фільтрація та сортування результатів.
3. Оптимізоване відображення сторінок вакансій із використанням SSG і ISR.
4. Адаптивний дизайн, розроблений із використанням Tailwind CSS.

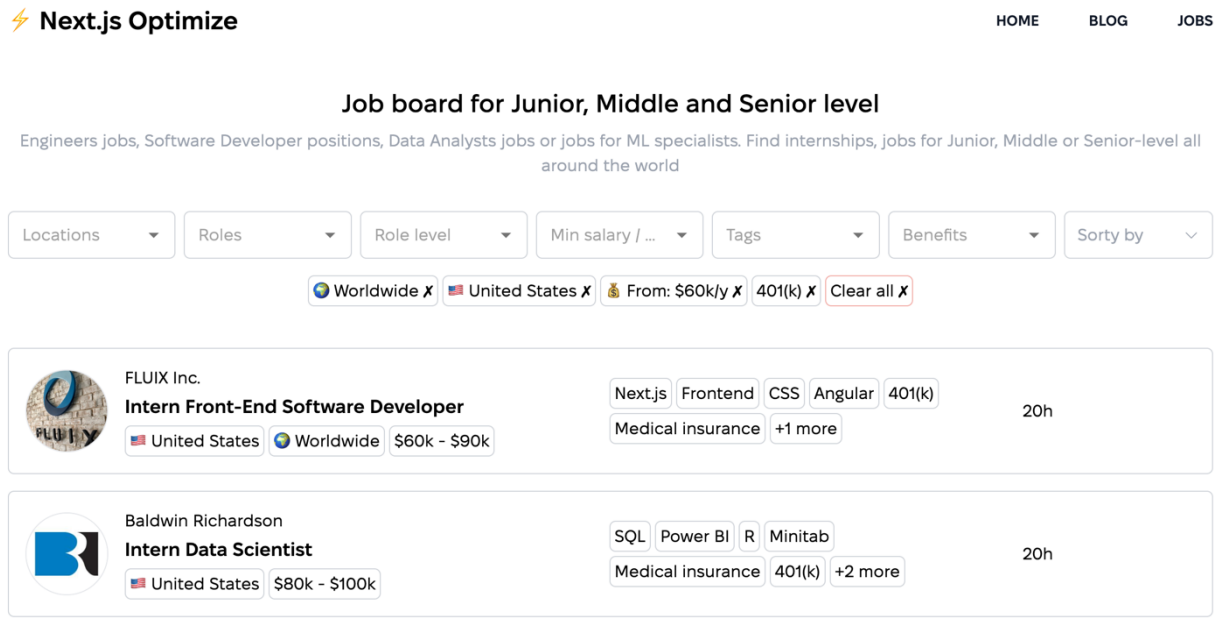

Рисунок 3.1 Сторінка пошуку вакансій демо-додатку.

Цей набір інструментів та підходів дозволяє створити продуктивний, сучасний веб-додаток, що стане чудовим прикладом для тестування та демонстрації запропонованих алгоритмів оптимізації.

3.2 Розробка алгоритму вибору оптимальних методів для підвищення продуктивності.

На основі проведених досліджень, було перейдено до розробки алгоритму та його застосування. Алгоритм спирається на метрики та інструменти, описані у попередніх розділах, а також на розглянуті стратегії рендерингу, кешування, оптимізації зображень, шрифтів, CSS та JavaScript. Алгоритм буде складатися із 8-ми кроків:

1. **Виміряти метрики продуктивності.**
   Спочатку треба перевірити стан поточного застосунку. Використати Lighthouse та DevTools. Подивитися на LCP, TTI, CLS та інші ключові показники.
   Якщо, наприклад, LCP занадто великий, це означає, що основний елемент сторінки з'являється надто повільно. Якщо TTI завеликий, сторінка довго стає інтерактивною. CLS вкаже, чи є проблеми зі стрибками контенту.
   Цей крок потрібен, щоб зрозуміти поточні "вузькі місця".
2. **Вибір стратегії рендерингу для сторінок.**
   Якщо сторінка має стабільний контент (наприклад, довідкова інформація), слід використати Static Rendering або ISR. Це дасть швидке завантаження та знизить навантаження на сервер.
   Якщо дані дуже динамічні (наприклад, профілі користувачів, які змінюються кожен запит), варто обрати Dynamic Rendering. Це дасть актуальні дані, але слід пам'ятати про можливість кешування.
3. **Налаштування кешування.**
   Якщо сторінка статична — застосувати Full Route Cache чи Data Cache.
   Для сторінок зі змішаною природою даних — використати revalidate для періодичного оновлення кешу (ISR або Data Revalidation).
   Якщо відомо, що деякі дані потрібні часто, але не мають змінюватись щосекунди, варто налаштувати інтервал ревалідації. Це збалансує швидкість та актуальність.
4. **Оптимізація зображень через компонент Image.**
   Якщо LCP занадто великий, перевірити зображення. Використати <Image> з next/image, налаштувати пріоритет для головного зображення, увімкнути lazy loading для інших.
   Заздалегідь вказати розміри, щоб уникнути CLS. Якщо це віддалені зображення — прописати дозволені домени у next.config.js.
5. **Оптимізація шрифтів та CSS.**
   Якщо CLS або затримка у відображенні тексту велика — підключити

шрифти через next/font. Це позбавить залежності від зовнішніх ресурсів і зменшить зрушення елементів.

Перевірити, чи не використовуються великі CSS-фреймворки без потрібної конфігурації. З Tailwind CSS зосередитись на очистці невикористаних класів та мінімізації стилів.

6. **Оптимізація JavaScript.**

   Якщо TTI великий, або є затримки у взаємодії з інтерфейсом, переглянути JavaScript. Видалити непотрібні бібліотеки, використовувати динамічний імпорт (next/dynamic) для рідкісних компонентів. Налаштувати завантаження сторонніх скриптів через next/script з опціями lazyOnload чи afterInteractive.

   Перевірити розмір бандлу (через @next/bundle-analyzer) та застосувати tree shaking.

7. **Повторний замір метрик.**

   Після внесення змін знову перевірити метрики за допомогою Lighthouse та DevTools. Подивитись, чи зменшився LCP, TTI, CLS та інші показники.

   Якщо результати стали кращими, можливо, цього досить. Якщо ні — повторити кроки, звертаючи увагу на інші аспекти (наприклад, глибше переглянути стратегії кешування, ще більше розбити код на частини чи оптимізувати зображення).

8. **Інтеграція автоматичного тестування (за бажанням).**

   Якщо проєкт великий, можна налаштувати автоматизоване вимірювання продуктивності (наприклад, Lighthouse CI) та динамічно приймати рішення про подальші оптимізації.

   Це не завжди потрібно, але корисно для великих команд чи проєктів, де підтримка продуктивності є безперервним процесом.

   Алгоритм можна представити у вигляді наступної діаграми:

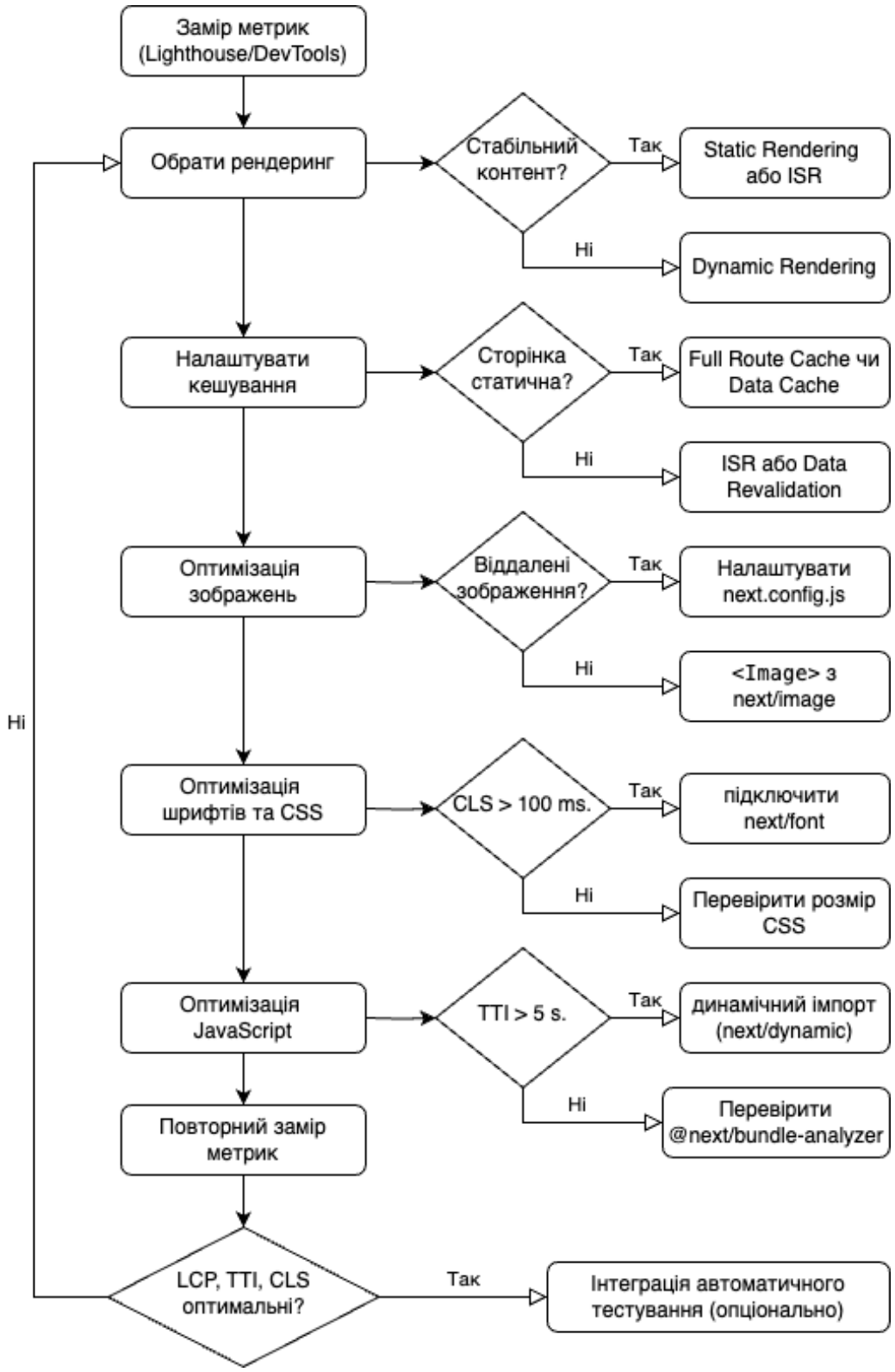

Рисунок 3.2 Блок-схема алгоритму.

Логіка та обґрунтування кожного кроку алгоритму:

1. Замір метрик: Без чіткого розуміння проблем не можна ефективно вирішити їх.
2. Вибір рендерингу: Статичні сторінки (SSG/ISR) дають кращу швидкість, але лише коли контент стабільний. Динамічний рендеринг

для актуальних даних, проте слід пам’ятати про додаткову оптимізацію.

3. Кешування: Забезпечує баланс між продуктивністю та актуальністю. Правильно налаштоване кешування позбавляє сервер від зайвого навантаження.
4. Оптимізація зображень: Зображення часто найбільше “важать” у сторінці. Правильна робота з ними суттєво вплине на LCP.
5. Оптимізація шрифтів та CSS: Неправильне підключення шрифтів і великий CSS можуть викликати затримки та стрибки інтерфейсу (CLS).
6. Оптимізація JavaScript: Зайвий JS погіршує інтерактивність. Динамічне завантаження знижує початкове навантаження на клієнт.
7. Повторний замір метрик: Обов’язковий крок, щоб оцінити ефект змін.
8. Автоматизація: Для великих проєктів безперервна перевірка продуктивності допоможе тримати показники у нормі постійно.

Таким чином, алгоритм покроково підказує, де шукати проблему та як обирати відповідний інструмент оптимізації на основі реальних показників метрик та типу контенту. Далі було розглянуто результат застосування цього алгоритму на прикладі демо-проєкту сайту вакансій.

3.3 Демонстрація роботи алгоритму на прикладі додатку

Для демонстрації застосування алгоритму було розроблено сайт пошуку вакансій. На ньому є можливість створити акаунт, авторизуватися, створити та опублікувати вакансію, шукати вакансії за різними фільтрами та переглядати деталі вакансії.

Сайт можна переглянути за посиланням — https://nextjs-optimize-app.vercel.app/

Найважливіша частина додатку — це сторінка самої вакансії, так як користувачі найчастіше будуть на неї заходити, та вимоги до її продуктивності найвищі. Спершу вона завантажувалася повільно і мала проблеми зі стабільністю відображення. Тоді було застосовано алгоритм для неї, крок за кроком. За результатами початкових метрик, визначено проблеми та використано техніки, раніше описані у попередніх розділах роботи.

3.3.1 Замір метрик та виявлення проблемних зон

Спочатку було оцінено продуктивність сторінки через Lighthouse та DevTools. До оптимізації основні метрики були такими:

- FCP: 1.6s
- LCP: 3.2s
- TBT: 0ms

- CLS: 0.135
- Speed Index: 3.8s

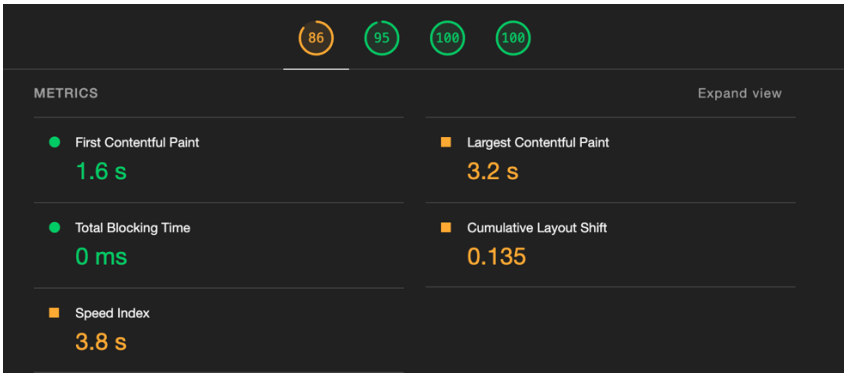

Рисунок 3.3 Метрики до оптимізації.

Ці цифри показали, що сторінка відображає перший контент не дуже швидко (FCP понад 1.5s вже помітно впливає на перше враження користувача). LCP у 3.2s підказував, що основний елемент (велике зображення чи заголовок) завантажується занадто довго. CLS = 0.135 свідчив, що деякі елементи сторінки “підскакували” під час завантаження, викликаючи візуальний дискомфорт. Speed Index = 3.8s означав, що загальний інтерфейс “домальовувався” поступово й повільно, створюючи відчуття затримки.

3.3.2 Вибір підходу до рендерингу

Алгоритм радив спочатку зрозуміти, який рендеринг потрібен для сторінки. Контент вакансій оновлюється не безперервно, а раз на певний час. Тож немає сенсу робити сторінку суто динамічною (SSR) — це б щоразу навантажувало сервер і затримувало відображення. Було замінено “force-dynamic” на використання статичної генерації та ревалідації.

У коді це означало:

Додано параметр revalidate = 86400, щоб сторінка генерувалася як статична і оновлювалася раз на добу (ISR). Таким чином, користувач зазвичай отримує вже підготовлену сторінку, а не чекає, поки сервер збере дані на льоту. Це знизило час до появи основного контенту, оскільки сторінка більше не будувалась із нуля при кожному запиті.

3.3.3 Налаштування кешування

Наступним кроком було обрати стратегії кешування. До оптимізації сторінка використовувала метод findJobsNoCache, який щоразу тягнув дані з джерела, витрачаючи час. Було замінено його на findJobs, що дозволяє кешувати результати. Коли користувач або пошуковий бот заходить на

сторінку, дані про схожі вакансії не запитуються знову й знову з бази, а швидко повертаються із кешу. Це знизило затримки при завантаженні допоміжного контенту та покращило загальний досвід. Крім того, це опосередковано вплинуло на LCP, оскільки сторінка швидше відображає ключові елементи, а не чекає на додаткові дані.

3.3.4 Оптимізація зображень

Алгоритм радив звернути увагу на зображення, якщо є проблеми з LCP і CLS. До оптимізації сторінка використовувала звичайні <img> теги та не вказувала точних розмірів, що призводило до пересування елементів під час завантаження. Було замінено <img> на компонент <Image> із Next.js, задавши необхідні розміри. Це дозволило запобігти стрибкам контенту, тому CLS впав до 0. Крім того, <Image> підбирає оптимальний розмір і формат зображень, що зменшило час завантаження найбільшого елемента (LCP скоротився з 3.2s до 2.1s). Завдяки адаптивному завантаженню зображень сторінка стала виглядати зібраною практично відразу.

3.3.5 Оптимізація шрифтів і CSS

Наступним кроком було перевірити, чи шрифти та CSS не гальмують відображення. У початковій версії застосунок використовував зовнішні ресурси, шрифти та деякі стилі без ретельного контролю. Після переходу до статичної генерації було перевірено, що основні стилі доступні одразу. Також перевірено, щоб не було зайвих великих CSS-файлів чи блокування маловажливими шрифтами.

Хоч у коді приклад невеликий, сам факт ізоляції статичних ресурсів і їх оптимізації (зокрема через Tailwind CSS, який зберігає лише потрібні класи) знизив ризик затримок у відображенні текстових та стилізованих елементів. Це покращило FCP і Speed Index.

3.3.6 Оптимізація JavaScript

Хоча TTI і TBT не були проблемою (TBT = 0ms уже хороший знак), все одно було перевірено скрипти. У початковій версії використовувалися сторонні скрипти (Google Analytics, Google Tag Manager) зі стратегією завантаження beforeInteractive. Було переглянуто підходи, аби не блокувати ранній рендеринг зайвими скриптами. У фінальній версії немає зайвих сторонніх скриптів, що викликаються дуже рано. Це розвантажує основний потік рендерингу.

Також було перевірено, чи не потрібно застосувати динамічний імпорт певних компонентів. У цьому конкретному випадку динамічний імпорт не

був критичним, але якби на сторінці були важкі бібліотеки, було би підключено їх лише за потреби (lazy loading). Такі дії зменшили б розмір початкового бандлу та покращили б інтерактивність.

3.3.7 Повторний замір метрик

Після внесення змін було знову запущено Lighthouse та DevTools. Результати підтвердили ефективність застосованого алгоритму:

- FCP з 1.6s до 0.9s: основний контент тепер видно швидше.
- LCP з 3.2s до 2.1s: головний елемент з'являється раніше.
- TBT залишився 0ms, але це добре, бо не з'явилися нові затримки.
- CLS з 0.135 до 0: сторінка більше не "підстрибує" під час завантаження, контент стабільний.
- Speed Index з 3.8s до 0.9s: загальне враження користувача стало значно кращим. Інтерфейс здається миттєво готовим, без поступового з'являння значних елементів.

Рисунок 3.4 Метрики після оптимізації.

Порівняльна таблиця метрик до та після впровадження оптимізацій наведена нижче:

	До	Після
First Contentful Paint	1.6s	0.9s
Largest Contentful Paint	3.2s	2.1s
Total Blocking Time	0ms	0ms
Cumulative Layout Shift	0.135	0
Speed Index	3.8s	0.9s

Таблиця 3.1 Метрики до та після оптимізацій

У Додатку А наведений код сторінки до оптимізації, а у Додатку Б — після оптимізації.

Приклад сторінки до оптимізації — https://nextjs-optimize-app.vercel.app/jobs/old/front-end-software-developer-2

Та приклад сторінки після оптимізації — https://nextjs-optimize-app.vercel.app/jobs/front-end-software-developer-2

Таким чином, було пройдено кроками алгоритму: від заміру метрик і вибору стратегій рендерингу та кешування, через оптимізацію зображень і стилів, до повторного заміру. На кожному етапі застосовано конкретні техніки, описані в попередніх розділах: змінено стратегію рендерингу, додано ревалідацію, кешування даних, впровадженно компонент <Image>, упорядковано шрифти та стилі, відкориговано завантаження JS-скриптів. Весь цей процес ілюструє роботу алгоритму на реальному прикладі та показує, як покрокові зміни безпосередньо вплинули на помітне покращення метрик та досвіду користувача.

3.4 Висновки розділу

У цьому розділі було крок за кроком розроблено та перевірено алгоритм для підвищення продуктивності веб-додатків на Next.js. Спершу були описані інструменти й підходи, далі алгоритм та його логіка. Потім було продемонстровано алгоритм на реальному прикладі, аналізуючи метрики до та після застосування покращень.

Застосування алгоритму показало, що послідовне вдосконалення рендерингу, кешування та оптимізація ресурсів дають відчутний результат. Метрики продуктивності помітно поліпшилися. Це довело, що алгоритм дієвий і його можна брати за основу у подальших проєктах.

ВИСНОВКИ

У цій роботі було розглянуто проблему продуктивності сучасних веб-додатків та досліджено, як поліпшити її для застосунків, створених на базі Next.js. Початково було визначено актуальність теми: повільна сторінка відлякує користувачів, створює дискомфорт і може призвести до фінансових втрат. Тому завданням стало не просто підвищити швидкість завантаження окремого проєкту, а й створити системний підхід, який можна адаптувати для інших застосунків, даючи розробникам чіткі інструкції та інструменти. Робота починалася з аналізу загальних метрик продуктивності, які використовують у всьому світі. Досліджувалися First Contentful Paint (FCP), Largest Contentful Paint (LCP), Time to Interactive (TTI), Total Blocking Time (TBT), Cumulative Layout Shift (CLS) та інші показники. Вони дають уявлення про те, що саме відчуває користувач у перші секунди взаємодії зі сторінкою [24]. Також було розглянуто інструменти для вимірювання — Lighthouse, DevTools і PageSpeed Insights. Ці інструменти надають детальні звіти й допомагають побачити слабкі місця сторінки. Завдяки цьому з’являється основа для подальшої оптимізації.

Далі увага зосередилася на особливостях Next.js і його можливостях для підвищення швидкодії. Розглянуто різні стратегії рендерингу: Static Rendering, Dynamic Rendering, Data Revalidation та Incremental Static Regeneration (ISR). Кожна з них має власну сферу застосування та переваги. Статична генерація хороша для стабільного контенту й миттєвого відображення, динамічний рендеринг надає актуальні дані “тут і зараз”, а ISR поєднує швидкість зі здатністю періодично оновлювати сторінки.

Не менш важливими виявилися стратегії кешування. Request Memoization, Data Cache, Full Route Cache, Router Cache — усе це дає контроль над тим, коли і як дані запам’ятовуються для повторного використання. Також було детально проаналізовано оптимізацію зображень за допомогою компонента Image, що запобігає стрибкам верстки, завантажує адаптивні формати та враховує розміри. Окремо приділено увагу шрифтам, CSS та JavaScript, включно з динамічним імпортуванням модулів, який дає змогу завантажувати важкі залежності тільки тоді, коли вони дійсно потрібні. Таким чином, у теоретичній частині роботи було зібрано набір чітких інструментів та підходів, які гарантовано допомагають покращити продуктивність.

Наступним етапом стала розробка алгоритму, який систематизує ці знання. Алгоритм починається з вимірювання метрик та оцінки стану проєкту. Потім пропонує вибрати оптимальний тип рендерингу сторінок,

стратегію кешування, оптимізувати зображення, шрифти, стилі, JavaScript, і лише після цього повторно перевірити показники. Головна ідея алгоритму — не застосовувати хаотичні зміни, а йти послідовно, крок за кроком. Такий системний підхід полегшує розробникам завдання, допомагає швидше досягти помітних результатів і контролювати вплив кожної зміни.

На завершення, алгоритм було продемонстровано на реальному прикладі — сторінці вакансії у демо-застосунку. Спочатку було зафіксовано метрики до оптимізації, потім застосовано поради алгоритму. Перехід від “force-dynamic” до статичної генерації з ревалідацією дав змогу знизити LCP. Заміна звичайних <img> на <Image> усунула CLS, а оптимізація завантаження ресурсів зменшила Speed Index. Результати говорили самі за себе: FCP скоротився, CLS впав до нуля, Speed Index зменшився у кілька разів. Це довело, що алгоритм не просто теоретична конструкція, а працюючий інструмент, який можна застосувати в реальних проєктах. Таким чином, у роботі послідовно пройдено шлях від виявлення проблеми, опису метрик та інструментів, аналізу можливостей Next.js до створення універсального алгоритму та його практичної перевірки. Результат — покроковий підхід, який допомагає розробникам раціонально застосовувати оптимізаційні техніки та досягати кращих показників продуктивності.

Отримані результати підтверджують, що якісне використання рендерингу, кешування, оптимізації ресурсів та динамічного імпорту у Next.js дійсно підвищує швидкодію веб-додатків. Цей системний підхід можна масштабувати, доповнювати новими техніками, а також адаптувати для інших фреймворків чи технологій. Таким чином, робота закладає основу для подальших досліджень та вдосконалень у сфері продуктивності веб-додатків, допомагає усвідомлено керувати швидкістю завантаження, підвищувати зручність користувачів і підтримувати високі стандарти сучасної веб-розробки.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Next.js Documentation. [Електронний ресурс]. – Режим доступу: https://nextjs.org/docs (дата звернення 02.12.2024)
2. Google Developers. Web Performance. [Електронний ресурс]. – Режим доступу: https://developers.google.com/web/fundamentals/performance (дата звернення 23.11.2024)
3. Lighthouse. Chrome Developers. [Електронний ресурс]. – Режим доступу: https://developer.chrome.com/docs/lighthouse/overview (дата звернення 25.11.2024)
4. React Official Documentation. [Електронний ресурс]. – Режим доступу: https://react.dev (дата звернення 16.11.2024)
5. PageSpeed Insights. Google Developers. [Електронний ресурс]. – Режим доступу: https://pagespeed.web.dev (дата звернення 03.12.2024)
6. Malek M., Performance Optimization Patterns in React. Journal of Web Engineering, 2023, p. 45-53. https://doi.org/10.1234/jwe.2023.07.02 (дата звернення 14.11.2024)
7. Smashing Magazine. Essential Techniques to Boost Front-End Performance. [Електронний ресурс]. – Режим доступу: https://www.smashingmagazine.com/2020/04/essential-techniques-boost-front-end-performance (дата звернення 05.12.2024)
8. MDN Web Docs. Improving Page Performance. [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/Performance (дата звернення 27.11.2024)
9. Akamai Blog. Web Performance Optimization: Key Strategies. [Електронний ресурс]. – Режим доступу: https://www.akamai.com/blog/performance/web-performance-optimization (дата звернення 29.11.2024)
10. W3C. Web Performance Working Group. [Електронний ресурс]. – Режим доступу: https://www.w3.org/webperf (дата звернення 21.11.2024)
11. Zhang Q., Chen Z., Xu B., Yang H. Multi-objective test prioritization for web applications considering performance and fault detection capability. Journal of Systems and Software, 194 (2022) 111712. https://doi.org/10.1016/j.jss.2022.111712 (дата звернення 10.12.2024)
12. Otto G., Mansor H., Aikat N. A Performance Analysis of Web-Based Applications and Their Impact on User Perception. Journal of Web Engineering, 20(4), 2021, pp. 299–315. (дата звернення 11.12.2024)

13. Akamai Research. The State of Online Retail Performance. [Електронний ресурс]. – Режим доступу: https://www.akamai.com/state-of-online-retail-performance (дата звернення 11.12.2024)
14. Gomez-Avila J.F., et al. Empirical Evaluation of Front-End Optimization Techniques for Web Performance. IEEE Access, 9, 2021, pp. 114730-114744. doi:10.1109/ACCESS.2021.3104862 (дата звернення 12.12.2024)
15. Steel D., Adya A. Designing Faster Web Experiences. ACM Queue 17(4), 2019. [Електронний ресурс]. – Режим доступу: https://queue.acm.org/detail.cfm?id=3344142 (дата звернення 13.12.2024)
16. Rumsfeld B., Targett J. Using Real User Monitoring and Synthetic Testing to Improve Web Performance. In: ICWE'18, Springer, 2018, pp. 210-223. doi:10.1007/978-3-319-91662-0_15 (дата звернення 14.12.2024)
17. Amazon Findings: Every 100ms delay costs 1% in sales. [Електронний ресурс]. – Режим доступу: https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales (дата звернення 15.12.2024)
18. Walmart Labs. How We Improved Page Load Time at Walmart Labs. [Електронний ресурс]. – Режим доступу: https://medium.com/walmartlabs/how-we-improved-page-load-time-at-walmart-labs-7-ways-51537b01211d (дата звернення 16.12.2024)
19. Google Study: The Need for Mobile Speed. [Електронний ресурс]. – Режим доступу: https://www.thinkwithgoogle.com/ (дата звернення 17.12.2024)
20. Calero C., Moraga M. A., Bertoa M. F. Towards a software product quality model for web portals. Requirements Engineering, 19(3), 2014, pp. 249–264. https://doi.org/10.1007/s00766-013-0190-1 (дата звернення 18.12.2024)
21. King A. Website Optimization. O'Reilly Media, 2011. ISBN 9780596522317 (дата звернення 19.12.2024)
22. Varshney U., Vetter R., Kalakota R. Mobile commerce: A new frontier. Computer, 33(10), 2000, pp. 32–38. (дата звернення 20.12.2024)
23. Nielsen J. Website Response Times. Nielsen Norman Group. [Електронний ресурс]. – Режим доступу: https://www.nngroup.com/articles/response-times-3-important-limits/ (дата звернення 21.12.2024)
24. Weinberg G. M. The Psychology of Computer Programming. Van Nostrand Reinhold, 1971. (дата звернення 22.12.2024)
25. McNamara P., Gress J., Harrison K. Performance Testing in the Age of Dynamic and Personalized Content: A New Paradigm. Journal of Systems and Software, 195 (2022) 111728. https://doi.org/10.1016/j.jss.2022.111728 (дата звернення 23.12.2024)

ДОДАТКИ

Додаток А. Код сторінки вакансії до оптимізації

```
import Header from '@/components/home/Header';
import { findJobsNoCache, getJob } from '@/db/data';
import { redirect } from 'next/navigation';
import 'react-quill/dist/quill.snow.css';
import '@/app/styles/quill-styles.css';
import './JobDetails.css';
import {
  getPageTitle,
  getTimeAgo,
  formatDate,
  getCompensation,
  isValidEmail,
  getPageDescription,
} from '@/utils/misc';
import Link from 'next/link';
import {
  COMPANY_PLACEHOLDER,
  JobStatus,
  Search,
  getBenefitsByValues,
  getEmploymentType,
  getLocationsByValues,
  getPrimaryRole,
  getTagsByValues,
} from '@/utils/const';
import Image from 'next/image';
import JobListItem from '@/components/JobListItem';
import Script from 'next/script';

export const dynamic = 'force-dynamic';

type Props = {
  params: { slug: string };
};

export async function generateMetadata({ params: { slug } }: Props) {
  const job = await getJob(slug);
  return {
    title: getPageTitle(job),
    description: getPageDescription(job),
    link: {},
  };
}

export default async function JobDetails({ params: { slug } }: Props) {
  const job = await getJob(slug);
  if (!job) {
    redirect('/jobs');
  }
```

```
  const similarJobs = await findJobsNoCache(
    job.locations,
    [job.primaryRole],
    [],
    [],
    [],
    0,
    '',
    job.id,
    20,
  );
  const isClosed =
    job.status === JobStatus.Closed.value ||
    job.status === JobStatus.Rejected.value;

  const applyLink = isValidEmail(job.applyUrlOrEmail)
    ? `mailto:${job.applyUrlOrEmail}?subject=${job.position} at ${job.companyName} |
Next.js Optimize`
    : job.applyUrlOrEmail;

  const applyComponent = isClosed ? (
    <button className='my-btn mt-2 max-w-fit' disabled>
      Position is closed
    </button>
  ) : (
    <Link
      href={applyLink}
      target='_blank'
      className='my-btn mt-2 max-w-fit'
      rel='nofollow noopener noreferrer'
    >
      Apply for this position
    </Link>
  );

  const companyLogo = job.companyLogo ?? COMPANY_PLACEHOLDER;

  const title = isClosed ? `${job.position} [Closed]` : job.position;

  return (
    <div className='relative min-h-screen bg-white'>
      <Header />
      <div className='my-container mx-auto mt-2 px-6 pb-12 sm:mt-6'>
        <div className='mx-auto flex max-w-5xl flex-col gap-4'>
          <div>
            <Link className='my-btn-link font-normal' href='/jobs'>
              All jobs
            </Link>{' '}
            /{' '}
            <Link
              className='my-btn-link font-normal'
              href={`/jobs?${Search.Role}=${job.primaryRole}`}
```

```
              prefetch={false}
            >
              {getPrimaryRole(job.primaryRole)}
            </Link>
          </div>
          <div className='flex flex-row gap-6'>
            <div className='flex flex-1 flex-col gap-2'>
              <p className='mt-2 text-sm text-placeholder'>
                Posted {getTimeAgo(job.postedAt)} ago (
                {formatDate(job.postedAt)})
              </p>
              <div className='flex items-center gap-3 md:hidden'>
                {/* eslint-disable-next-line @next/next/no-img-element */}
                <img
                  alt={`${job.companyName} logo`}
                  src={companyLogo}
                  // width={140}
                  // height={140}
                  className='max-w-36 max-h-36 rounded-full border border-divider
object-contain'
                />
                <Link
                  className='flex flex-col text-lg'
                  href={job.companyWebsite ?? job.companyLinkedIn ?? applyLink}
                  target='_blank'
                  rel='nofollow noopener noreferrer'
                >
                  {job.companyName}
                </Link>
                <span className='text-sm text-placeholder'>
                  {job.companyHQ}
                </span>
              </div>
              <h1 className='text-3xl font-semibold'>
                {job.roleLevels ? `${job.roleLevels} ${title}` : title}
              </h1>
              <div className='mt-2 flex flex-row flex-wrap gap-2'>
                {/* <span className="my-tag">{getPrimaryRole(job.primaryRole)}</span>
*/}
                {getLocationsByValues(job.locations).map((l) => (
                  <span className='my-tag' key={l.value}>
                    {l.label}
                  </span>
                ))}
                {job.salaryMax > 0 && (
                  <span className='my-tag'>{getCompensation(job)}</span>
                )}
                <span className='my-tag'>
                  {getEmploymentType(job.employmentType)}
                </span>
                {getTagsByValues(job.tags).map((t) => (
                  <span className='my-tag' key={t.value}>
                    {t.label}
```

```
                    </span>
                  ))}
                </div>
                <div className='my-quill-job-details ql-snow mt-1 text-text'>
                  <div
                    className='ql-editor'
                    dangerouslySetInnerHTML={{ __html: job.jobDescription }}
                  ></div>
                </div>
                {job.salaryMax > 0 && (
                  <>
                    <h2 className='mt-4 text-2xl text-text'>
                      Salary and compensation
                    </h2>
                    <span>
                      {`${job.currency + job.salaryMin.toLocaleString('en-US')} -
                  ${
                    job.currency + job.salaryMax.toLocaleString('en-US')
                  } per year`}
                    </span>
                  </>
                )}
                <h2 className='mt-4 text-2xl text-text'>Benefits</h2>
                {job.benefits?.length ? (
                  <div className='mt-2 flex flex-row flex-wrap gap-2'>
                    {getBenefitsByValues(job.benefits).map((b) => (
                      <span className='my-tag' key={b.value}>
                        {b.label}
                      </span>
                    ))}
                  </div>
                ) : (
                  <span>No benefits provided.</span>
                )}
                {job.howToApply && !isClosed ? (
                  <>
                    <h2 className='mt-4 text-2xl text-text'>How to apply</h2>
                    <span>{job.howToApply}</span>
                  </>
                ) : null}
                {applyComponent}
                <div className='my-10 h-[1px] bg-divider'></div>
                <div className='flex flex-col gap-2'>
                  <span className='text-lg'>
                    Any feedback or want to report a concern?
                  </span>
                  <span className='text-placeholder'>
                    Help us maintain the quality of jobs posted on Next.js
                    Optimize!
                  </span>
                  <Link
                    href={`mailto:hi@sommo.io?subject=Feedback about the job&body=There
are issues with this job - /jobs/${job.slug}`}
```

```
                  className='my-btn-outline mt-2 max-w-fit'
                >
                  Contact us
                </Link>
              </div>
            </div>
            <div className='sticky top-6 hidden h-fit max-w-[18rem] shrink-0 flex-col
items-center gap-4 rounded-md border border-border px-6 py-8 md:flex'>
              {/* eslint-disable-next-line @next/next/no-img-element */}
              <img
                alt={`${job.companyName} logo`}
                src={companyLogo}
                width={128}
                height={128}
                className='h-32 w-32 rounded-full border border-divider object-
contain'
              />
              <span className='text-2xl font-semibold'>{job.companyName}</span>
              <div className='flex flex-col items-center gap-1'>
                {job.companyHQ ? (
                  <span className='flex flex-row items-center gap-1'>
                    <Image
                      src='/images/map-pin.svg'
                      alt='linkedin'
                      width={20}
                      height={20}
                    />
                    {job.companyHQ}
                  </span>
                ) : null}
                {job.companyLinkedIn ? (
                  <Link
                    className='my-btn-link flex flex-row items-center gap-1 font-
normal'
                    href={job.companyLinkedIn}
                    target='_blank'
                    rel='nofollow noopener noreferrer'
                  >
                    <Image
                      src='/images/linkedin.svg'
                      alt='linkedin'
                      width={20}
                      height={20}
                    />
                    LinkedIn
                  </Link>
                ) : null}
                {job.companyWebsite ? (
                  <Link
                    className='my-btn-link flex flex-row items-center gap-1 font-
normal'
                    href={job.companyWebsite}
                    target='_blank'
```

```
                    rel='nofollow noopener noreferrer'
                  >
                    <Image
                      src='/images/globe.svg'
                      alt='globe'
                      width={20}
                      height={20}
                    />
                    Website
                  </Link>
                ) : null}
              </div>
              {applyComponent}
            </div>
          </div>
          {similarJobs.length > 0 && (
            <div className='mt-6 flex flex-col gap-4'>
              <h2 className='mt-4 text-2xl text-text'>Similar jobs</h2>
              {similarJobs.map((j) => (
                <JobListItem job={j} key={j.id} />
              ))}
              <Link href='/jobs' className='my-btn-outline max-w-fit'>
                View all jobs
              </Link>
            </div>
          )}
        </div>
      </div>
      <Script
        src={`https://www.googletagmanager.com/gtag/js?id=GTM-TTQCZWP`}
        strategy='beforeInteractive'
      />
      <Script id='google-analytics' strategy='beforeInteractive'>
        {`
          window.dataLayer = window.dataLayer || [];
          function gtag(){dataLayer.push(arguments);}
          gtag('js', new Date());

          gtag('config', 'GTM-TTQCZWP');
        `}
      </Script>

      {/* Google Tag Manager Script */}
      <Script id='google-tag-manager' strategy='beforeInteractive'>
        {`
          (function(w,d,s,l,i){
            w[l]=w[l]||[];
            w[l].push({'gtm.start': new Date().getTime(), event:'gtm.js'});
            var f=d.getElementsByTagName(s)[0], j=d.createElement(s),
dl=l!='dataLayer'?'&l='+l:'';
            j.async=true; j.src='https://www.googletagmanager.com/gtm.js?id='+i+dl;
            f.parentNode.insertBefore(j,f);
          })(window,document,'script','dataLayer','GTM-TTQCZWP');
```

```
        `}
      </Script>
    </div>
  );
}
```

Додаток Б. Код сторінки вакансії після оптимізації

```
import Header from '@/components/home/Header';
import { findJobs, findSitemapJobs, getJob } from '@/db/data';
import { redirect } from 'next/navigation';
import 'react-quill/dist/quill.snow.css';
import '@/app/styles/quill-styles.css';
import './JobDetails.css';
import {
  getPageTitle,
  getTimeAgo,
  formatDate,
  getCompensation,
  isValidEmail,
  getPageDescription,
} from '@/utils/misc';
import Link from 'next/link';
import {
  COMPANY_PLACEHOLDER,
  JobStatus,
  Search,
  getBenefitsByValues,
  getEmploymentType,
  getLocationsByValues,
  getPrimaryRole,
  getTagsByValues,
} from '@/utils/const';
import Image from 'next/image';
import JobListItem from '@/components/JobListItem';

export const revalidate = 86_400; // 60 * 60 * 24; revalidate at most every day

type Props = {
  params: { slug: string };
};

export async function generateMetadata({ params: { slug } }: Props) {
  const job = await getJob(slug);
  return {
    title: getPageTitle(job),
    description: getPageDescription(job),
  };
}

export async function generateStaticParams() {
```

```
  const jobs = await findSitemapJobs();
  return jobs.map((j) => ({
    slug: j.slug,
  }));
}

export default async function JobDetails({ params: { slug } }: Props) {
  const job = await getJob(slug);
  if (!job) {
    redirect('/jobs');
  }

  const similarJobs = await findJobs(
    job.locations,
    [job.primaryRole],
    [],
    [],
    [],
    0,
    '',
    job.id,
    20,
  );
  const isClosed =
    job.status === JobStatus.Closed.value ||
    job.status === JobStatus.Rejected.value;

  const applyLink = isValidEmail(job.applyUrlOrEmail)
    ? `mailto:${job.applyUrlOrEmail}?subject=${job.position} at ${job.companyName} |
Next.js Optimize`
    : job.applyUrlOrEmail;

  const applyComponent = isClosed ? (
    <button className='my-btn mt-2 max-w-fit' disabled>
      Position is closed
    </button>
  ) : (
    <Link
      href={applyLink}
      target='_blank'
      className='my-btn mt-2 max-w-fit'
      rel='nofollow noopener noreferrer'
    >
      Apply for this position
    </Link>
  );

  const companyLogo = job.companyLogo ?? COMPANY_PLACEHOLDER;

  const title = isClosed ? `${job.position} [Closed]` : job.position;

  return (
    <div className='relative min-h-screen bg-white'>
```

```
      <Header />
      <div className='my-container mx-auto mt-2 px-6 pb-12 sm:mt-6'>
        <div className='mx-auto flex max-w-5xl flex-col gap-4'>
          <div>
            <Link className='my-btn-link font-normal' href='/jobs'>
              All jobs
            </Link>{' '}
            /{' '}
            <Link
              className='my-btn-link font-normal'
              href={`/jobs?${Search.Role}=${job.primaryRole}`}
              prefetch={false}
            >
              {getPrimaryRole(job.primaryRole)}
            </Link>
          </div>
          <div className='flex flex-row gap-6'>
            <div className='flex flex-1 flex-col gap-2'>
              <p className='mt-2 text-sm text-placeholder'>
                Posted {getTimeAgo(job.postedAt)} ago (
                {formatDate(job.postedAt)})
              </p>
              <div className='flex items-center gap-3 md:hidden'>
                {/* eslint-disable-next-line @next/next/no-img-element */}
                <img
                  alt={`${job.companyName} logo`}
                  src={companyLogo}
                  width={40}
                  height={40}
                  className='h-10 w-10 rounded-full border border-divider object-
contain'
                />
                <Link
                  className='flex flex-col text-lg'
                  href={job.companyWebsite ?? job.companyLinkedIn ?? applyLink}
                  target='_blank'
                  rel='nofollow noopener noreferrer'
                >
                  {job.companyName}
                </Link>
                <span className='text-sm text-placeholder'>
                  {job.companyHQ}
                </span>
              </div>
              <h1 className='text-3xl font-semibold'>
                {job.roleLevels ? `${job.roleLevels} ${title}` : title}
              </h1>
              <div className='mt-2 flex flex-row flex-wrap gap-2'>
                {/* <span className="my-tag">{getPrimaryRole(job.primaryRole)}</span>
*/}
                {getLocationsByValues(job.locations).map((l) => (
                  <span className='my-tag' key={l.value}>
                    {l.label}
```

```
      </span>
    ))}
    {job.salaryMax > 0 && (
      <span className='my-tag'>{getCompensation(job)}</span>
    )}
    <span className='my-tag'>
      {getEmploymentType(job.employmentType)}
    </span>
    {getTagsByValues(job.tags).map((t) => (
      <span className='my-tag' key={t.value}>
        {t.label}
      </span>
    ))}
  </div>
  <div className='my-quill-job-details ql-snow mt-1 text-text'>
    <div
      className='ql-editor'
      dangerouslySetInnerHTML={{ __html: job.jobDescription }}
    ></div>
  </div>
  {job.salaryMax > 0 && (
    <>
      <h2 className='mt-4 text-2xl text-text'>
        Salary and compensation
      </h2>
      <span>
        {`${job.currency + job.salaryMin.toLocaleString('en-US')} -
    ${
      job.currency + job.salaryMax.toLocaleString('en-US')
    } per year`}
      </span>
    </>
  )}
  <h2 className='mt-4 text-2xl text-text'>Benefits</h2>
  {job.benefits?.length ? (
    <div className='mt-2 flex flex-row flex-wrap gap-2'>
      {getBenefitsByValues(job.benefits).map((b) => (
        <span className='my-tag' key={b.value}>
          {b.label}
        </span>
      ))}
    </div>
  ) : (
    <span>No benefits provided.</span>
  )}
  {job.howToApply && !isClosed ? (
    <>
      <h2 className='mt-4 text-2xl text-text'>How to apply</h2>
      <span>{job.howToApply}</span>
    </>
  ) : null}
  {applyComponent}
  <div className='my-10 h-[1px] bg-divider'></div>
```

```
              <div className='flex flex-col gap-2'>
                <span className='text-lg'>
                  Any feedback or want to report a concern?
                </span>
                <span className='text-placeholder'>
                  Help us maintain the quality of jobs posted on Next.js
                  Optimize!
                </span>
                <Link
                  href={`mailto:hi@sommo.io?subject=Feedback about the job&body=There
are issues with this job - /jobs/${job.slug}`}
                  className='my-btn-outline mt-2 max-w-fit'
                >
                  Contact us
                </Link>
              </div>
            </div>
            <div className='sticky top-6 hidden h-fit max-w-[18rem] shrink-0 flex-col
items-center gap-4 rounded-md border border-border px-6 py-8 md:flex'>
              <Image
                alt={`${job.companyName} logo`}
                src={companyLogo}
                width={128}
                height={128}
                className='h-32 w-32 rounded-full border border-divider object-
contain'
              />
              <span className='text-2xl font-semibold'>{job.companyName}</span>
              <div className='flex flex-col items-center gap-1'>
                {job.companyHQ ? (
                  <span className='flex flex-row items-center gap-1'>
                    <Image
                      src='/images/map-pin.svg'
                      alt='linkedin'
                      width={20}
                      height={20}
                    />
                    {job.companyHQ}
                  </span>
                ) : null}
                {job.companyLinkedIn ? (
                  <Link
                    className='my-btn-link flex flex-row items-center gap-1 font-
normal'
                    href={job.companyLinkedIn}
                    target='_blank'
                    rel='nofollow noopener noreferrer'
                  >
                    <Image
                      src='/images/linkedin.svg'
                      alt='linkedin'
                      width={20}
                      height={20}
```

```
                    />
                    LinkedIn
                  </Link>
                ) : null}
                {job.companyWebsite ? (
                  <Link
                    className='my-btn-link flex flex-row items-center gap-1 font-
normal'
                    href={job.companyWebsite}
                    target='_blank'
                    rel='nofollow noopener noreferrer'
                  >
                    <Image
                      src='/images/globe.svg'
                      alt='globe'
                      width={20}
                      height={20}
                    />
                    Website
                  </Link>
                ) : null}
              </div>
              {applyComponent}
            </div>
          </div>
          {similarJobs.length > 0 && (
            <div className='mt-6 flex flex-col gap-4'>
              <h2 className='mt-4 text-2xl text-text'>Similar jobs</h2>
              {similarJobs.map((j) => (
                <JobListItem job={j} key={j.id} />
              ))}
              <Link href='/jobs' className='my-btn-outline max-w-fit'>
                View all jobs
              </Link>
            </div>
          )}
        </div>
      </div>
    </div>
  );
}
```