МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Кафедра комп’ютерних систем та мереж

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри комп’ютернихсистем та мереж

_________________ Жуков І.А.

«___» _______________202_ р.

ДИПЛОМНА РОБОТА
(**ПОЯСНЮВАЛЬНА ЗАПИСКА**)

ВИПУСКНИКА ОСВІТНЬО–КВАЛІФІКАЦІЙНОГО РІВНЯ
"МАГІСТР"
напрям підготовки - 123 “Комп’ютерна інженерія”

Тема: Методологія оптимізації веб-додатків за критеріями пошукової системи Google.

Виконавець:____ Вовк М.О.

Керівник:____ Гузій М.М.

Нормоконтролер:____ Андрєєв В.І.

Засвідчую, що у магістерській роботі немає запозичень праць інших авторів без відповідних посилань

Студент _____________ Вовк М.О

Київ 2020

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет кібербезпеки, комп’ютерної та програмної інженерії
Кафедра комп’ютерних систем та мереж
Освітнього ступеня магістр
Напрям 123 Комп’ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри

_________________Жуков І.А.

«___» _______________2020 р.

ЗАВДАННЯ
на виконання дипломного проекту

Вовку Максиму Олеговичу
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. Тема проекту: Методологія оптимізації веб-додатків за критеріями пошукової системи Google.
затверджена наказом ректора від "25" вересня 2020 року № 1793/ст.

2. Термін виконання проекту (роботи): з 25.09.2020 до 21.12.2020

3. Вихідні дані до проекту (роботи): Сучасні дані про методи оптимізації веб-додатків. Інформація про інструментальні засоби для аналізу продуктивності веб-додатку.

4. Зміст пояснювальної записки (перелік питань, що підлягають розробці):
Методологія для оптимізації веб-додатку. Програмна реалізація алгоритму для видалення надлишкового CSS-коду. Програмна реалізація алгоритму для генерування корисного JavaScript коду.

5. Перелік обов’язкового графічного матеріалу:
1) Презентація *PowerPoint*;

6. Календарний план

№ п/п	Етапи виконання дипломного проекту	Термін виконання етапів	Примітка
1	Узгодження технічного завдання з керівником проекту	25.09.20	
2	Підбір та вивчення науково-технічної літератури за темою дипломного проекту	26.09.20 – 01.10.20	
3	Аналіз варіантів створення методології оптимізації веб-додатків	02.10.20 – 10.10.20	
4	Аналіз впроваджених методів оптимізації веб-додатків	11.10.20 – 15.10.20	
5	Розробка практичної частини	16.10.20 – 08.11.20	
6	Оформлення пояснювальної записки	09.11.20 – 07.12.20	
7	Оформлення графічних матеріалів проекту та представлення роботи на кафедру	08.12.20 – 10.12.20	

7. Дата видачі завдання «26» вересня 2020 р.

Керівник дипломного проекту ____________________ Гузій М.М.
(підпис)

Завдання прийняв до виконання ____________________ Вовк М.О.
(підпис випускника) (П.І.Б.)

Дата ___.___.________

РЕФЕРАТ

Дипломна робота присвячена розробленню методології та інструментальних засобів для оптимізації веб-додатків.

Пояснювальна записка до дипломного проекту «Методологія оптимізації веб-додатків за критеріями пошукової системи Google». містить 75 с., 54 рис., 3 табл., 40 літературних джерел.

ПОШУКОВІ СИСТЕМИ, ПОШУКОВА ОПТИМІЗАЦІЯ, *HTML*, *CSS*, *JAVASCRIPT*, *TYPESCRIPT*.

Мета дипломного проекту – розробити методологію та інструментальні засоби для оптимізації веб-додатків.

Об’єкт дослідження – технології інформаційного пошуку в мережі Інтернет.

Предмет дослідження – методологія оптимізації веб-додатків за критеріями пошукової системи *Google*.

Метод дослідження – аналіз сучасних варіантів оптимізації веб-додатків, розробка алгоритмів видалення надлишкового *CSS*-коду та генерування корисного *JavaScript* коду.

Новизна – створено методологію оптимізації швидкості веб-додатків, яка безпосередньо впливає на показники швидкості завантаження та розроблено інструментальні засоби для ефективної роботи при оптимізації швидкості веб-додатків.

Актуальність – розвиток нових технологій веб-розробки та проблема ефективності пошукової оптимізації, у зв’язку з постійним оновленням алгоритмів пошукових систем.

Матеріали дипломного проекту рекомендуються використовувати в сучасній веб-розробці.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ........................6
ВСТУП……………………………………………………………………..7
РОЗДІЛ 1 ТЕХНОЛОГІЇ ІНФОРМАЦІЙНОГО ПОШУКУ В МЕРЕЖІ ІНТЕРНЕТ..9
1.1. Принципи роботи пошукових систем...9
1.2. Загальна архітектура пошукових систем...13
1.3. Пошукова оптимізація веб-додатків...17
1.4. Висновки до розділу...25
РОЗДІЛ 2 АНАЛІЗ МЕТОДОЛОГІЇ ОПТИМІЗАЦІЇ ВЕБ-ДОДАТКІВ...............28
2.1. Основні методи внутрішньої оптимізації веб-додатків.......................28
2.2. Інструменти для аналізу веб-додатків..33
2.3. Висновки до розділу...40
РОЗДІЛ 3 РОЗРОБКА МЕТОДОЛОГІЇ ТА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ОПТИМІЗАЦІЇ ВЕБ-ДОДАТКІВ...42
3.1. Розробка методології оптимізації веб додатків за критеріями пошукової системи Google..42
3.2. Принципи роботи та тестування інструменту для виявлення надлишкового CSS коду...53
3.3. Розробка та практична цінність інструменту для генерування корисного JavaScript коду..60
3.4. Результати використання розробленої методології та інструментальних засобів для оптимізації веб-додатків......................69
3.5. Висновки до розділу...74
ВИСНОВКИ..76
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Кафедра КСМ				НАУ 20 01 43 – 000 ПЗ			
Виконав	Вовк М.О.			МЕТОДОЛОГІЯ ОПТИМІЗАЦІЇ ВЕБ-ДОДАТКІВ ЗА КРИТЕРІЯМИ ПОШУКОВОЇ СИСТЕМИ *GOOGLE*	Літера	Аркуш	Аркушів
Керівник	*Гузій М.М.*					13	80
Консульт.					123 КС-201Мз		
Норм. контр.	*Андрєєв В.І.*						
Зав. Каф.	*Жуков І.А..*						

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

WEB – World Wide Web (всесвітня мережа)

HTML – HyperText Markup Language (мова гіпертекстової розмітки)

CSS – Cascading Style Sheets (каскадні таблиці стилів)

JS – JavaScript (мова програмування)

SEO – Search Engine Optimization (пошукова оптимізація)

FCP – First Contentful Paint (час до першого відображення контенту)

SI – Speed Index (індекс швидкості)

LCP – Largest Contentful Paint (відображення найбільшого контенту)

TTI – Time to Interactive (час до взаємодії)

TBT – Total Blocking Time (загальний час блокування)

CLS – Cumulative Layout Shift (загальний зсув макету)

ВСТУП

Тема оптимізації швидкості веб-додатків є актуальною, адже рішення цієї проблеми на етапах розробки, безпосередньо впливає на якість кінцевого продукту. В сучасній веб-розробці, велику роль відіграє не тільки правильна архітектура та якість коду веб-додатку, а і швидкість та стабільність його роботи. Відомі пошукові системи мають свої інструменти та критерії оптимізації швидкості веб-додатків, але більшість з них мають загальний характер. В результаті стрімкого росту кількості нових технологій в розробці, виникає необхідність пошуку нових методів оптимізації. Тому це питання залишається актуальним та потребує детального дослідження.

За статистикою, повільні веб-додатки, швидкість завантаження яких складає понад три секунди, втрачає майже 57% користувачів. Результатом порівняння двох веб-додатків, швидкість завантаження яких відрізняється на одну секунду, є те, що на швидшому веб-додатку кількість переглядів, в середньому, більша на 22%, а кількість користувачів, що відмовились від використання цього додатку менша на 50%. Ці данні впливають на репутацію та доходи компанії, яка втрачає кількість користувачів.

Об’єктом дослідження виступають технології інформаційного пошуку в мережі Інтернет. Предметом дослідження є методологія оптимізації веб-додатків за критеріями пошукової системи *Google*.

Мета дипломної роботи – розробити методологію та інструментальні засоби для оптимізації веб-додатків. Для досягнення визначеної мети, ставляться такі завдання:

- дослідити принципи роботи пошукових систем
- проаналізувати загальну архітектуру пошукових систем
- визначити задачу пошукової оптимізації

- проаналізувати методи пошукової оптимізації
- обрати інструмент для аналізу продуктивності веб-додатків
- створити методологію оптимізації веб-додатків
- розробити інструментальні засоби для оптимізації веб додатків
- протестувати розроблені інструменти

Новизна отриманих результатів – створено методологію оптимізації швидкості веб-додатків, яка безпосередньо впливає на показники швидкості завантаження та розроблено інструментальні засоби для ефективної роботи при оптимізації швидкості веб-додатків.

Розроблено особисто. Алгоритм пошуку та видалення *CSS* стилів, які не використовуються на веб-сторінці. Створено інтерфейс та логіку додатку для генерування корисного *JavaScript* коду, при розробці конкретного проекту.

При виборі теми дипломного проекту, я зупинився на цій, оскільки виділені критерії та розроблені інструменти можуть використовуватись для оптимізації швидкості веб-додатків в сучасній веб-розробці.

РОЗДІЛ 1
ТЕХНОЛОГІЇ ІНФОРМАЦІЙНОГО ПОШУКУ В МЕРЕЖІ ІНТЕРНЕТ

1.1. Принципи роботи пошукових систем

Пошукова система – це сукупність програмного та апаратного забезпечення, призначеного для пошуку в Інтернеті. Це дозволяє користувачам швидко знаходити потрібну інформацію, відповідаючи на їх запитання та надаючи список посилань на ресурси. Всі пошукові системи використовують власні алгоритми побудови списку сайтів, що містять, на думку користувача, відповідь на його запит. Крім алгоритмів пошукова система використовує в роботі роботів, які індексують сайти, зображення, перевіряють доступність сайтів та інше. Пошук може здійснюватися не тільки по текстовому запиту, що вводиться в рядок пошуку, але, наприклад і по картинці або голосовому повідомленню. Більшість пошукових систем враховують регіональність сайту користувача і його запиту, видаючи йому найбільш коректну, на думку пошукової системи, відповідь у вигляді списку сайтів. Пошукові системи поділяються на такі види:

- національні пошукові системи – розробляються спочатку для пошуку сайтів всередині конкретної країни, тобто для внутрішнього ринку. Більшість з них поступово вийшли за рамки своєї держави, але при цьому не перейшли в розряд транснаціональних.
- транснаціональні пошукові системи – здійснюють пошук відповіді на запит користувача на сайтах всіх країн, незалежно від їх доменної зони та країни перебування.

Алгоритми пошукової системи для підготовки результатів на запити досить складні. Коли користувач вводить будь-який пошуковий запит, система дає відповідь майже миттєво завдяки алгоритму генерації результату пошуку по базі даних проіндексованих сайтів. Даний процес може бути представлений таким чином:

- отримання запиту;
- лінгвістичний аналіз, інтерпретація, морфологія, видалення омонімів, додавання синонімів та визначення тематики;
- пошук проіндексованих сторінок з відповідним вмістом, пов’язаним з темою та ключовими словами пошукового запиту;
- створення послідовності видачі з урахуванням ряду факторів;
- передача результатів пошуку користувачам.

Отримавши запит від користувача, пошукова машина виконує детальний аналіз введеного тексту. Для зручності користувачів пошукові системи визначають мову за алфавітом та поєднанню символів у запиті. Це дозволяє отримувати точні результати, навіть якщо користувач не змінив розкладку клавіатури. Пошукові фрази розширюються на основі введених користувачем ключових шаблонів морфології та доповнюються синонімами для максимального охоплення матеріалів, які будуть обрані для видачі результату. При створенні послідовності видачі переважно враховується відповідність ключовим словам, крім омонімів, тобто слів які пишуться однаково, але мають різне значення. Визначення омонімів виконуються на основі статистики слів, що використовуються одночасно. У процесі розширення пошукового запиту відбувається пошук синонімів, абревіатур та правопису на різних мовах. У процесі визначення назв об'єктів, як правило, визначаються назви компанії, місця та назви об’єктів, до яких не потрібно підбирати синонім та розширювати діапазон пошуку. Запити також обробляються на наявність граматичних та

орфографічних помилок. Завдяки високій продуктивності обладнання та використанні розподілених обчислень, ці операції проводяться за частки секунди, після чого формується пошуковий запит для подальшої обробки пошуковою системою. Природно, що він зазнає значних змін для повного охоплення тематики та максимального задоволення інтересу користувача.

Мова запиту до пошукової машини називається інформаційно-пошуковою. Вона містить логічні оператори, морфологію мови, регістри, префікси, можливість визначати інтервал між словами та розширений пошук. Таке представлення запиту створене для забезпечення швидкої вибірки з великих баз даних.

Очевидно, що користувачеві не завжди потрібно використовувати розширення пошуку, і якщо він справді пам’ятає фразу з потрібного йому документа, йому навряд чи сподобається той факт, що на першій сторінці видачі результату пошуку буде показана інформація зібрана по синонімах. Тому для пошукових систем існують загальні правила створення запитів, за допомогою яких можна визначити, що цікавить користувача. Якщо потрібно звузити пошук або він повинен відповідати правилам пошукових термінів, потрібно правильно формувати запит. В разі, коли потрібно знайти документ за ключовою фразою, необхідно вкладати її в лапки. У цьому випадку буде видано лише точну відповідність без розширення пошуку. Якщо потрібно знайти документи з декількома словами на будь-якій частині сторінки, слід написати їх після основного запиту зі знаком плюс. Уточнення значно звужують коло пошуку. Коли потрібно навпаки, не включати в пошук документи, що містять ключові слова, після основної фрази вказується виняток за допомогою знака мінус. Це можна використовувати для того, щоб відділити комерційні тексти від інформаційних, вказавши після мінуса: "ціна", "придбати", "не дорого" тощо. Використання логічного "АБО" у запиті, дозволяє шукати документи, що містять

хоча б одне ключове слово з переліку. Наприклад, в пошуковій системі Google, логічне "АБО" вказується службовим словом "OR".

Після подання запиту на пошук за індексом він базується на основі цього створюється вибірка з бази даних, що містить посилання на сторінки, які відповідають усім зазначеним умовам. Як правило, ця вибірка досить велика навіть для продуманих пошукових фраз і може містити сотні тисяч сторінок. Надати її користувачеві в такій необробленій формі, означає змусити його шукати вручну та аналізувати найбільш бажаний результат. Отже, після її формування видача ранжується за досить складною технологією. Ранжування – це сортування посилань на веб-сторінки в Інтернеті за критерієм корисності для користувача. Завдання ранжування полягає в тому щоб надати користувачу найбільш корисну інформацію, яка повністю відповідає критеріям його запиту. Саме ранжування має найбільший вплив на інтернет-маркетинг, безпосередньо визначаючи позицію сайту в результатах пошуку. Кожне оновлення може призвести до значних змін стосовно позиції сайту в пошуковій видачі, особливо якщо попередня позиція була досягнута не шляхом наповнення якісним вмістом, а шляхом використання технічних методів *SEO*-просування. Сучасні алгоритми ранжирування можуть враховувати тисячі параметрів, починаючи від статистичної характеристики тексту і закінчуючи змістом сайту. Весь процес сортування сторінок за пошуковою фразою розділений на два етапи:

- вибираються сторінки веб-сайту, найбільш релевантні, інформація з яких, повинна з’явитись у видачі;
- вибрана сторінка упорядковується за релевантністю запиту повторно.

Таким чином, в кінцевий результат видачі, як правило, потрапить одна сторінка окремого сайту, що полегшує користувачам вибір найцікавіших сайтів із декількох. Однак у деяких ситуаціях користувачу можуть бути видані кілька сторінок одного сайту. Кінцеві формули та алгоритми, які використовуються при

сортуванні пошуковою системою, відомі лише компаніям – власникам пошукових систем.

Після попереднього сортування вибрана сторінка проходить через кілька суворіших фільтрів, включаючи систему штрафів, яка зменшує важливість сторінки у пошуковій видачі. Штрафи або песивізація можуть бути застосовані як до всього ресурсу, так і до окремих сторінок за порушення правил публікації вмісту, плагіат, використання заборонених технологій та багатьох інших причин. Після завершення кожного кроку алгоритму визначення релевантності, знайдені сторінки сортуються від більш релевантних до менш релевантних і надсилаються користувачеві на екран у вигляді пошукової видачі.

Варто зазначити, що пошукові системи не розкривають усіх критеріїв ранжирування сайтів та технології визначення релевантності. Існують лише загальні поради, суть яких полягає в поліпшенні якості вмісту та його корисності для кінцевого користувача. Основними факторами є:

- внутрішні – текст, дизайн, графіка, нові посилання на сайті.
- зовнішні – посилання на сторінки веб-сайтів з інших джерел, діяльність у соціальних мережах;
- поведінкові – коефіцієнт відмов, час перебування на місці, глибина огляду тощо.

Однак часто на першій сторінці видачі можна побачити контент, який не відповідає всім вимогам та займає своє місце завдяки використанню технології "чорного *SEO*". Зазвичай, до таких сайтів застосовуються санкції після оновлення алгоритмів ранжування. Але іноді, оновлення алгоритмів спричиняє зниження позиції якісних ресурсів, тому їх власники мають увагою стежити та своєчасно реагувати на зміну ситуації, щоб запобігти втратам прибутку через зменшення потоку клієнтів з пошукових систем.

1.2 Загальна архітектура пошукових систем

Сьогодні пошукові системи є найпоширенішими доступними ресурсами пошуку інформації в мережі інтернет. В архітектурі пошукової системи можна виділити три базові частини:

- Робот (краулер, спайдер, індексатор) – відповідає за збір інформації, тобто емулює роботу користувача, завантажуючи сторінки та зберігаючи їх в базі даних.

- База даних, в якій зберігається і сортується зібрана роботом інформація.

- Клієнт, де обробляються запити користувача. Насправді клієнт може бути рознесений за кількома фізично непов'язаними комп'ютерами. Однак, варто відзначити, що всі ці комп'ютери повинні мати доступ до бази даних.

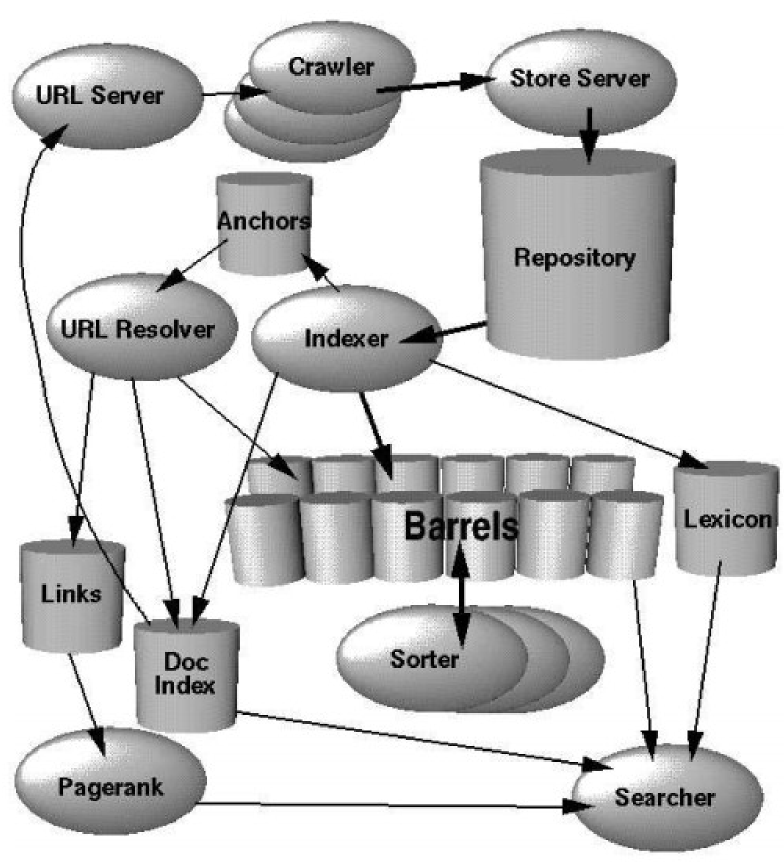


Рис. 1.1 – Загальна архітектура пошукової системи

На рис. 1.1 зображена загальна архітектура пошукової системи, яка складається з таких компонентів:

- *URL Server* – сервер, що містить список всіх *URL*-адрес.
- *Crawler* – це робот, який завантажує сторінки зі списку *URL*-адрес і передає в *Store Server*.
- *Store Server* – сервер, що зберігає сторінки в *Repository*, найчастіше у вигляді *HTML*-документа. Вся додаткова інформація, така як зображення, медіафайли, анімація та інше, не зберігається.
- *Indexer* – індексатор, який розбирає збережені в *Repository HTML*-документи в послідовності слів і зберігає їх в *Barrles* (база даних).
- *Lexicon* – містить список всіх слів. Найчастіше слова зберігаються в таблиці з двома полями "номер" і "слово". Таким чином досягається економія місця в базі даних, оскільки довгі слова замінюються досить коротким номером.
- *Anchors* – містить посилання, виділені індексатором (*Indexer*).
- *URL Resolver* - обробник *URL*-адрес. При знаходженні нових посилань, передає їх в *URL Server*.
- *Links* – визначає які сайти на які посилаються і передає результат в *PageRank*.
- *PageRank* – визначає рейтинг сайту, основним критерієм є кількість посилань на цей сайт.
- *Searcher* – клієнт, що користується статичною базою даних, яка оновлюється приблизно раз на добу.

Якщо розглядати варіанти зберігання записів в базі даних, тобто параметр, за яким вони відсортовані, і яка додаткова інформація зберігається для кожного запису, важливо розуміти поняття прямого і зворотного індексу. У випадку

прямого індексу, записи відсортовані за номером документа. Для кожного запису зберігається, відсортований за номером, список слів. Для кожного слова зберігаються перші кілька позицій входження слова в документ, кількість входжень і формат входження. Під форматом входження мається на увазі входження слова в тексті посилання, в описі до картинки, в заголовку і т.д., такі слова матимуть пріоритет при пошуку. Прямий індекс оновлюється постійно при роботі робота.

Для кожної сторінки в базі даних зберігається частота оновлення, яка рахується таким чином: при черговому заході робота на цю сторінку, в разі відсутності оновлень, частота збільшується у два рази, а якщо сторінка за цей період часу змінювалася, то зменшується. Також варто відзначити, що часто, робот індексує не всі слова з документа і не всі документи з одного сайту. Зворотний індекс використовується клієнтом при пошуку. В цьому випадку записи відсортовані за словами. Для кожного запису зберігається номер слова, список документів, в які входить це слово, і повна інформація про входження. Зворотний індекс оновлюється не так часто, як прямий, а приблизно раз на добу. Релевантність або показник наскільки слово відповідає даним документу, має певні характеристики, які можуть впливати на позицію документа в списку відповідей на пошуковий запит:

- Наявність слів в документі. Очевидно, що якщо слова в документі не зустрічаються, то даний документ не підходить під умови пошуку.

- Частота входження слів. Чим частіше слова зустрічаються на сторінці, тим вище документ опиниться в списку пошуку.

- Форматування слів. Тобто якщо в документі зустрічаються слова, що виділяються тегами заголовків, абзаців, описів картинок, то такий документ буде мати вищий пріоритет.

- У разі набору слів, має значення відстань між цими словами в документі і їх порядок.

- В деяких пошукових системах (наприклад, Яндекс) значення має морфологічне входження слів, тобто відмінок (рід, особа), в якому слово входить в документ.

- Кількість та цінність посилань.

- Реєстрація в каталозі пошукової системи. Це дуже важлива характеристика, оскільки каталоги складаються вручну і в них уже задані розділи та тематика сторінки.

Робота клієнта полягає в тому, що спочатку запит розбивається на слова. Далі видаляються так звані "стоп" слова – це слова, які зустрічаються майже у всіх документах (прийменники, сполучники). На наступному кроці кожному слову зіставляється його номер з "лексикону". Для кожного слова із запиту знаходиться у зворотному індексі список документів, які містять це слово. З цих списків створюється новий список, який містить тільки ті документи, які входили в списки для всіх слів. Потім, на основі характеристик релевантності, для кожного документа обчислюється ступінь релевантності та список сортується за цією ознакою. На цьому кроці для всіх документів створюються анотації. Інструкцією може бути зміст тегу *description*, контекст входження слів із запиту, перше речення або заголовок документа.

1.3 Пошукова оптимізація веб-додатків

Пошукова оптимізація веб-додатків - це набір заходів, спрямованих на підвищення позиції веб-додатку в результатах видачі пошукової системи. Пошукову оптимізацію веб-додатків також називають «*search engine optimization*» (з англ. «пошукова оптимізація сайту»). Дане визначення прийшло одночасно з появою пошукових систем, таких як *Google* і Яндекс. Після їх виходу на всесвітній ринок з'явилася величезна потреба оптимізувати сайти. Суть оптимізації полягає в тому, щоб повною мірою відповідати всім вимогам пошукових машин. Якщо сайт максимально оптимізований, то і в результатах

пошуку він буде завжди показуватися вище, ніж неоптимізований ресурс. Пошукова оптимізація сайту складається з зовнішніх і внутрішніх факторів. До внутрішніх, можна віднести роботу над кодом сайту і його вмістом. Наприклад, роботу над заголовками сторінок сайту *title*, а також над унікальністю тексту. До зовнішньої оптимізації відносяться обсяг посилань на сайт. Чим більше посилань, тим більше довіри та "поваги" від пошукових систем. Звичайно, не мається на увазі куплені або орендовані посилання, які відносяться до "чорного" *SEO*. Мова йде про природні та чесно заслужені посилання на сайт.

Оптимізація веб-додатку для пошукових систем – найважливіший етап розробки. Це пов’язано з тим, що просування веб-додатку шляхом купівлі зовнішніх посилань, без оптимізації не дасть ніякого ефекту, а в деяких випадках навпаки може зашкодити веб-додатку та зменшити трафік. Внутрішня *SEO*-оптимізація веб-додатку включає:

- Усунення дублів сторінок
- Робота з тегами
- Створення карти сайту
- Налаштування файлу *robots.txt*
- Написання *SEO* текстів
- Забезпечення унікальності змісту.
- Перелінковка
- Створення мікроформатів
- Юзабіліті та інтерфейс

Це неповний перелік для внутрішньої оптимізації. Але це те з чого починається оптимізація веб-додатків.

Іноді деякі *CMS* (конструктори сайтів) створюють кілька версій однієї сторінки сайту, які містяться в декількох *URL*-адресах. Наприклад, головна сторінка веб-додатку можна бути доступна за адресами:

- *www.my_site.com/index.php*

- *www.my_site.com/index.html*
- *www.my_site.com/index.htm*
- *www.my_site.com/main*
- *my_site.com/index.php*
- *my_site.com/index.html*
- *my_site.com/index.htm*
- *my_site.com/main*

Можна перевірити це, додавши *index.php* або *index.html* до адресного рядка браузера. Якщо сторінка відкривається за цією адресою і немає помилки 404, то на сайті присутні дублікати сторінок, які потрібно усунути. Щоб позбутися дублікатів сторінок, потрібне перенаправлення сервера 301. Ця команда дає зрозуміти пошуковій системі, що сторінки, з яких було здійснене перенаправлення, були переміщені на іншу адресу. Так однакові сторінки сайту об'єднуються, що відповідає вимогам пошукових систем, зокрема *Google* та Яндекса.

Мета-теги – *HTML*-теги сторінки, які необхідні для надання структурованих мета-даних електронного документа – сторінки веб-додатку. Зазвичай, мета-теги вказуються в заголовку *title*. Вони використовуються пошуковими системами для визначення інформації на сторінці та подальшого обліку цих даних при ранжируванні в пошуку. Мета-тег *title* – це заголовок сторінки в браузері, який показується у видачі пошукових систем (рис. 1.2). Мета-тег *title* враховується пошуковою системою при ранжуванні сайтів, вважається основним показником визначення релевантності сторінки до пошукових запитів.

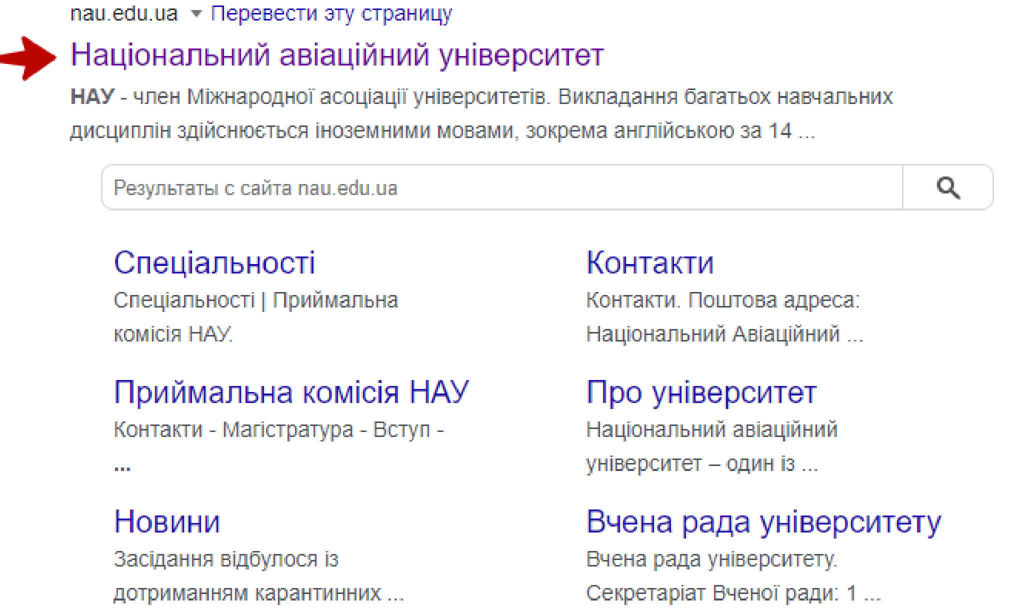


Рис. 1.2 – Мета-тег *title* в пошуковій видачі Google

Є певні вимоги до написання мета-тегів *title*, яких необхідно дотримуватись:

- *Title* на сторінці в коді повинен бути зазначений в обов'язковому порядку;
- Оптимальний розмір заголовка *title* повинен бути від 70 до 80 знаків;
- Писати назву сайту в *title* не обов'язково, тому, що пошукові системи враховують назву домену;
- У *title* повинні міститися пошукові запити релевантні стосовно змісту сторінки;
- Високочастотні запити необхідно прописувати спочатку заголовка;
- У коді сторінки *title* повинен бути розміщений в розділі *head*;
- Неприпустимо повторення одного і того ж мета-тегу *title* на двох різних сторінках;
- Не рекомендується використовувати більше одного разу одне і теж слово.

Мета-тег *description* - короткий опис сторінки сайту в браузері, який знаходиться під мета-тегом *title* у видачі пошукових систем, але на сторінці сайту його не видно відвідувачам (рис. 1.3). Він враховується пошуковими машинами для визначення вмісту сторінки з метою використання алгоритмом в розподілі сайтів в результатах пошуку.

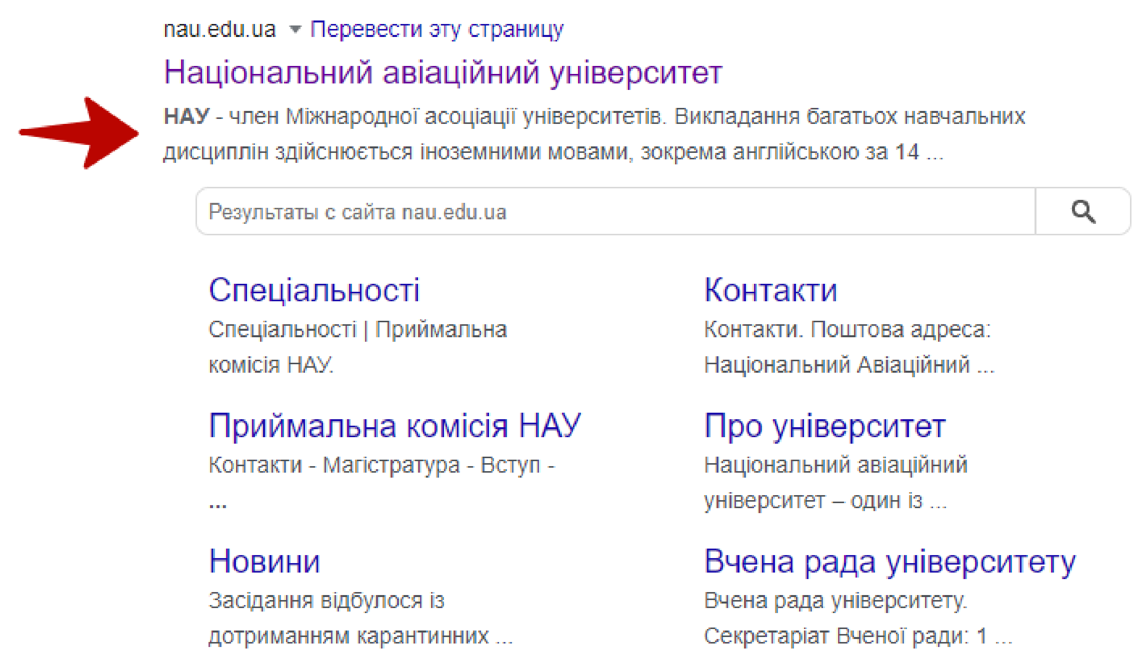


Рис. 1.3 – Мета-тег *description* в пошуковій видачі *Google*

Існують такі вимоги до написання мета-тега *description*:

- Довжина мета-тегу *description* не повинна перевищувати 300 знаків, оптимальний розмір 170-290 символів;
- Правильний мета-тег *description* повинен містити ключові слова, які належать цій сторінці;
- Опис *description* має бути складено у вигляді короткого оголошення, яке передасть користувачеві смислове навантаження сторінки;
- На кожній сторінці сайту мета-тег *description* повинен бути унікальним, не допускається його повторення на інших сторінках сайту;
- Мета теги *title* і *description* повинні відрізнятися один від одного;

Теги заголовків *h1-h6* - це шість заголовків, які показують відносну важливість блоку. Тег *h1* – це заголовок першого рівня, тег *h2* – заголовок другого рівня і так далі. Рівень заголовка показує його важливість. Заголовок *h1* є другим за значимістю елементом на сторінці для сайтів *SEO*, поступаючись лише тегу *title* і є найважливішим серед всіх тегів заголовку. Тег заголовку *h1* має використовуватись на сторінці не більше одного разу. За замовчуванням вміст тегу *h1* зображається найбільшим жирним шрифтом. Інші теги заголовки мають менший розмір шрифту залежно від рівня.

Оскільки пошукові системи ще не здатні визначати тип зображень, фотографій чи діаграм, кожне зображення має мати назву – альтернативний текст, який вказується за допомогою атрибуту *alt*. Це необхідно для залучення додаткового трафіку на сайт під час пошуку користувачів за зображеннями. Можуть бути ситуації, коли люди, які відвідують сайт під час пошуку зображень, не є потенційними клієнтами. Але такі користувачі приносять додатковий трафік, на який звертають увагу пошукові системи.

Також, при пошуковій оптимізації, потрібно створювати карту сайту. Карта сайту – це *html* сторінка сайту або спеціальний *xml* файл, в якому містяться посилання на всі важливі сторінки сайту. Щоб зрозуміти, що таке карта сайту, досить уявити книжковий зміст і відразу ж стане зрозуміло, навіщо потрібна карта сайту. Карта сайту допомагає відвідувачеві, будь це людина або пошуковий робот, швидко знайти будь-яку сторінку сайту, здійснюючи мінімальне число переходів. На простих, невеликих сайтах всі сторінки можуть бути доступні через 1-2 кліка від головної сторінки, але, якщо сайт великий і структура його непроста, він буде незручним не тільки відвідувачам, але і неочевидним для пошукових систем. *XML* карта сайту - це файл формату *xml*, виду *sitemap.xml*, який зазвичай знаходиться в корені сайту. У карти сайту в форматі *xml* є безліч переваг перед *html* картою сайту. *Sitemap xml* – це спеціальний формат карти сайту, який визначається всіма популярними пошуковими системами, наприклад *Google* і Яндекс (рис. 1.4). В *xml sitemap* можна вказати до 50000 посилань. Більш того, в *sitemap xml* можна вказати відносний пріоритет і частоту оновлення сторінок. Варто сказати, що вміст карти сайту є лише рекомендацією для

пошукового робота. Наприклад, якщо встановити для сторінки сайту щорічну частоту оновлення, пошукові роботи все одно ходитимуть частіше. А якщо встановити частоту оновлення сторінки щогодини, це не означає, що робот буде індексувати сторінку щогодини.

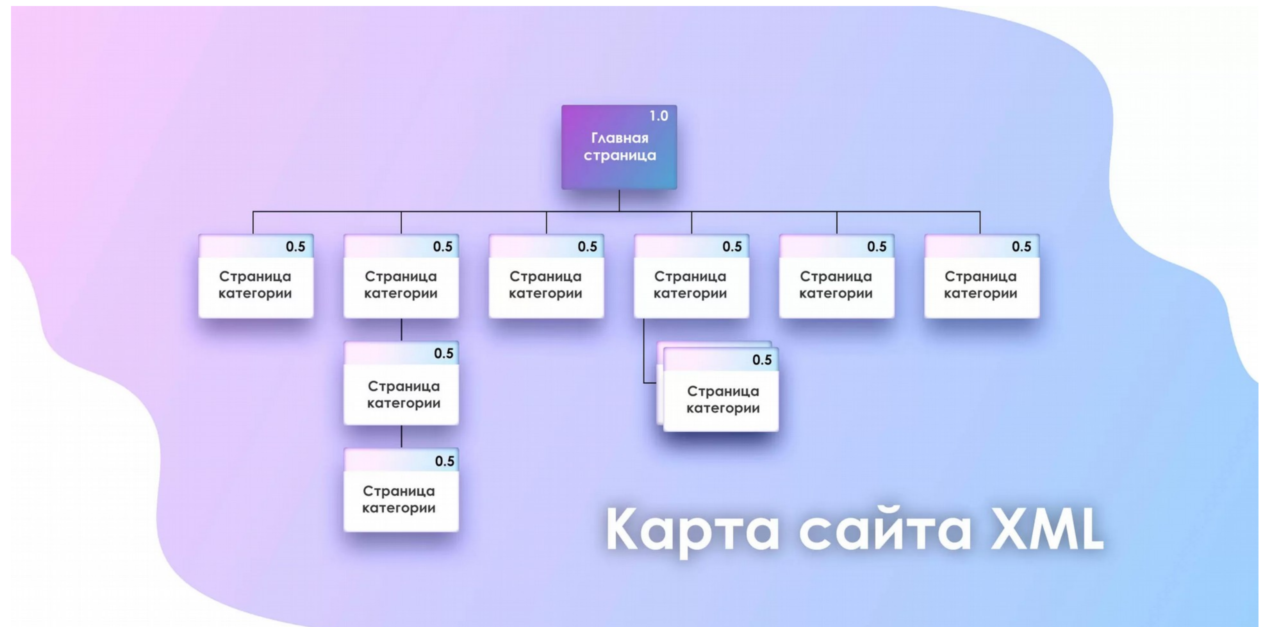


Рис. 1.4 – Схематичне зображення карти сайту

Файл *robots.txt* – це звичайний текстовий документ у кодуванні *UTF-8*, що діє для протоколів *http*, *https* та *FTP*. Файл вказує пошуковим системам, які сторінки чи файли слід сканувати. Якщо файл не містить символів в *UTF-8*, але використовує інше кодування, пошукова машина може не обробити його належним чином. Правила, що містить файл *robots.txt*, дійсні лише для хосту, протоколу та номера порту, на якому знаходиться файл. Файл повинен знаходитися в кореневому каталозі у вигляді простого текстового документа, який можна знайти за відповідною адресою, наприклад: *https: //my_site/robots.txt*. Коли правила обробляються у файлі *robots.txt*, пошукові системи отримують одну з трьох інструкцій:

- Частковий доступ: скануються окремі елементи сайту.
- Повний доступ: можна сканувати все.
- Повністю заборонено: робот не може нічого сканувати.

Під час сканування файлу *robots.txt* робот отримує таку відповідь:

- 2*xx* – сканування успішне.
- 3*xx* – пошуковий робот, слідує по переадресації, доки не буде надана інша відповідь.
- 4*xx* – пошуковий робот вважає, що можна сканувати весь вміст сайту.
- 5*xx* – тимчасова помилка сервера, сканування повністю заборонено. Робот звертатиметься до файлу, поки не отримає іншу відповідь.

Файл *robots.txt* потрібен, коли необхідно запобігти відвідуванню пошуковими роботами певної сторінки. Наприклад, іноді робот не повинен відвідувати:

- Сторінки, що містять особисту інформацію користувачів на сайті.
- Сторінки з різними типами передачі даних.
- Дзеркала.
- Сторінки результатів пошуку.

Без файлу *robots.txt*, сторінки які необхідно приховувати від сторонніх поглядів, потрапляють в пошукову видачу, але за допомогою правильно налаштованого файлу *robots.txt* пошукові роботи ігнорують існування сторінки (рис. 1.5).

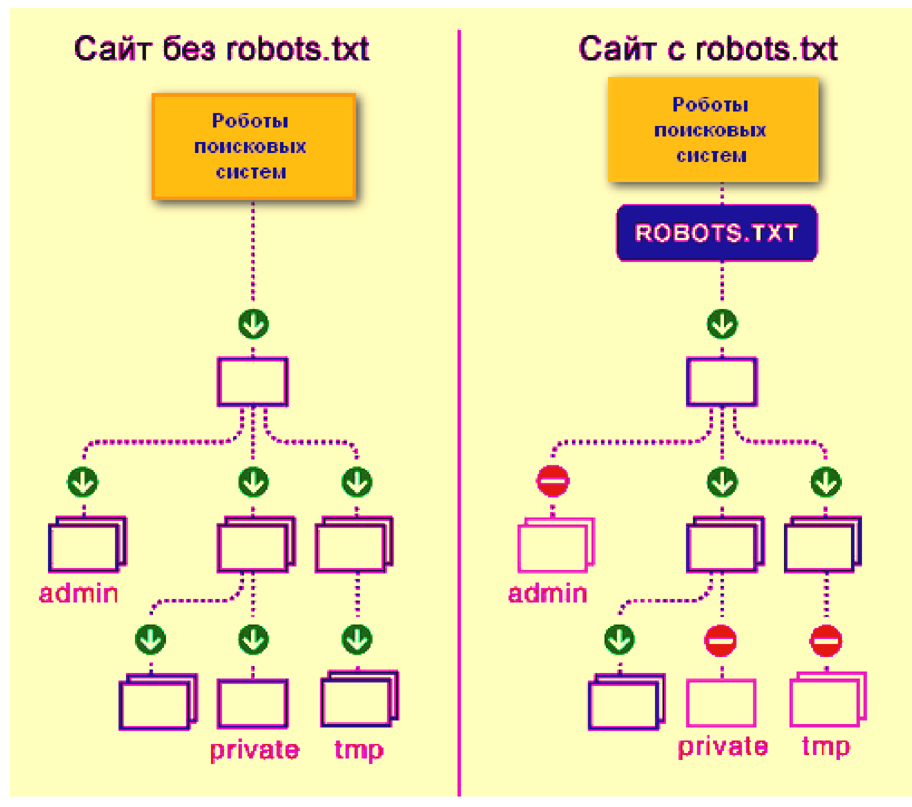


Рис. 1.5 – Сайт без *robots.txt* та з правильно налаштованим *robots.txt*

Для того, щоб оптимізувати веб-додаток, необхідно використовувати *SEO*-тексти, щоб надати можливість знаходити веб-додаток за ключовими словами. Але тексту, зазвичай, дуже багато, і статті повинні бути написані окремо для кожного пошукового запиту правильно. Це займає багато часу та дорого коштує. Тож в *SEO*-тексті часто пишуть кілька десятків пошукових запитів, але такий підхід не забезпечує присутність всіх запитів у видачі пошукової системи. При правильному методі написання *SEO*-тексту, зазвичай, працює більша частина пошукових фраз.

Для того, щоб підвищити внутрішню продуктивність веб-додатку, перед його просуванням необхідно переглянути та зробити весь контент унікальним. *Google* та Яндекс, в результатах пошуку будуть знаходитись вище ті веб-додатки, де розміщений унікальний текст. Якщо унікальність тексту нижче 80%, потрібно його переписати, а якщо він не є унікальним, потрібно написати новий, корисний і зрозумілий *SEO*-текст.

Перелінковка – це зв’язка сторінок веб-додатку за допомогою посилань. Цей набір посилань формує позитивне ставлення пошукових систем до веб-додатку. Посилання на сторінки поділяються на зовнішні та внутрішні. Зовнішні посилання – це посилання на інші веб-ресурси, а внутрішні - це посилання, що ведуть з однієї сторінки веб-додатку на іншу. Обидва типи перелінковки дуже важливі для пошукової оптимізації веб-додатку.

Мікроформати - це спеціальні теги в коді сторінки у форматі *HTML*. Мікроформати розмічають веб-сторінки та допомагають пошуковим системам визначити інформацію, яку несе ця сторінка. Наприклад, це може бути сторінка, що містить інформацію про особу, замовлення чи товар.

Юзабіліті є важливим елементом, який збільшує конверсію веб-додатку. Коротко кажучи, без правильної та простої структури веб-додатку, немає сенсу в

просуванні. Сайт повинен бути простим і зрозумілим. Продажі, реєстрації та інші конверсії базуються на простоті використання.

Якщо веб-додаток гарний, зручний та оптимізований, але при цьому повільно завантажується, то він буде негативно сприйматись пошуковими системами. *Google* закликає створювати швидкі веб-додатки, тому що швидкість пошуку інформації залежить від швидкості завантаження веб-додатку. Тому при пошуковій оптимізації, збільшення швидкості завантаження веб-додатку є головним завданням.

Тому, для внутрішньої оптимізації, перед просуванням сайту потрібно виконати багато роботи. Кожна дрібниця при оптимізації сайту може вплинути на його позицію в пошуковій видачі. Для досягнення найкращого результату, використовують різні сервіси та інструменти.

1.4. Висновки до розділу

Було досліджено принципи роботи пошукових систем. При виконанні пошукового запиту вони працюють за наступним алгоритмом:

- отримання запиту;
- лінгвістичний аналіз, інтерпретація, морфологія, видалення омонімів, додавання синонімів та визначення тематики;
- пошук проіндексованих сторінок з відповідним вмістом, пов’язаним з темою та ключовими словами пошукового запиту;
- створення послідовності видачі з урахуванням ряду факторів;
- передача результатів пошуку користувачам.

Проаналізовано процес сортування сторінок за пошуковою фразою, який розділений на два основні етапи:

- вибір сторінок веб-сайту, найбільш релевантних, інформація з яких, повинна з’явитись у видачі;

- вибрана сторінка упорядковується за релевантністю запиту повторно.

Досліджено загальні поради, суть яких полягає в поліпшенні якості вмісту та його корисності для кінцевого користувача. Основними факторами є:

- внутрішні – текст, дизайн, графіка, нові посилання на сайті.
- зовнішні – посилання на сторінки веб-сайтів з інших джерел, діяльність у соціальних мережах;
- поведінкові – коефіцієнт відмов, час перебування на місці, глибина огляду тощо.

Проаналізовано загальну архітектуру пошукових систем. В архітектурі пошукової системи виділено три базові частини:

- Робот (краулер, спайдер, індексатор) – відповідає за збір інформації.
- База даних, в якій зберігається і сортується зібрана роботом інформація.
- Клієнт, де обробляються запити користувача.

Визначена головна задача внутрішньої пошукової оптимізації веб-додатків, а саме, збільшення швидкості завантаження та продуктивності веб-додатку. Проаналізовано основні етапи *SEO*-оптимізації, до складу яких відносять:

- Усунення дублів сторінок
- Робота з тегами
- Створення карти сайту
- Налаштування файлу robots.txt
- Написання SEO текстів
- Забезпечення унікальності змісту.
- Перелінковка
- Створення мікроформатів
- Юзабіліті та інтерфейс

РОЗДІЛ 2
АНАЛІЗ МЕТОДОЛОГІЇ ВНУТРІШНЬОЇ ОПТИМІЗАЦІЇ ВЕБ-ДОДАТКІВ

2.1. Основні методи внутрішньої оптимізації веб-додатків

Як користувачі, так і пошукові системи не люблять довго чекати поки завантажиться сайт, тому, виникає потреба скоротити час його завантаження до мінімуму. Крім цього, у пошукових систем є обмеження по часу індексації сайту. Тому *Google* додав параметр швидкості завантаження сайту як один з найважливіших факторів ранжування. Іншими словами, якщо веб-сторінки не завантажуються досить швидко, то веб-додаток може втратити свою позицію в результатах видачі. Оптимізація коду веб-додатку належить до внутрішньої оптимізації та є одним з важливих факторів, що впливають на його ранжування. Існує перелік методів внутрішньої оптимізації веб-додатків, який використовується в сучасній розробці та потребує детального аналізу.

Пошуковий робот проводить збір інформації з мільйонів сайтів щодня. І навіть тут оптимізатор може знайти важелі управління. Пошукові системи висловлюють деякі рекомендації по оптимізації, які можуть спростити завдання робота. Серед цих заходів провідне значення належить оптимізації *html*-коду. Заходи, які спрямовані на адаптацію коду сторінки під вимоги пошукових систем, іменуються оптимізацією *html*-коду. Для початку варто усвідомити основні завдання та цілі, які переслідує цей вид оптимізації. В першу чергу це поліпшення внутрішніх характеристик сайту, які можуть вплинути на індексацію. Якщо робот

буде регулярно заходити на сайт, аналізуючи його вміст і при цьому не буде зустрічати перешкод, то сайт буде знаходитися на пріоритетних позиціях в пошуковій видачі. Інший момент, коли код сайту складний, містить помилки, то робот буде відвідувати такі проекти в останню чергу. Також крім погіршення індексації, сайт може гірше ранжуватися. Тому для досягнення максимального результату, внутрішня оптимізація передбачає комплексний підхід до роботи над *html*-кодом, який має п'ять основних етапів:

1) Стилізація.
2) Видалення зайвих тегів.
3) Усунення помилок.
4) Структура коду.
5) Закриття зовнішніх посилань.

В процесі створення веб-додатку, розробники часто стикаються з проблемою переповнення таблиць стилів. В цьому випадку для очищення коду, використовується метод перенесення стилів з *HTML* в каскадну таблицю стилів – *CSS*. Суть методу полягає в тому, що стилі, які знаходяться в різних тегах сайту, конвертуються в класи. Таким чином, можна істотно скоротити код, передаючи різним елементам необхідні класи.

У процесі розробки або редагування коду, часто допускаються помилки, наприклад наявність незакритих тегів або використання тегів, які частково чи повністю не підтримуються браузерами. Такі помилки мають бути усунені при оптимізації веб-додатку.

Дотримання структури коду є також дієвим методом, який допомагає швидко редагувати різні відділи сайту одночасно. Цей метод передбачає створення структури по типу глобальних блоків. При цьому код веб-додатку розділяється на розділи: верхній, тіло, бічні елементи, низ. При такому підході, кожен розділ може редагуватися незалежно від інших.

Наявність зовнішніх посилань, що потрапляють на веб-сторінку разом з коментарями користувачів, внаслідок незаконних дій або встановлення сторонніх скриптів, можуть нести загрозу для веб-додатку. Усунення цієї неполадки відбувається за допомогою видалення зовнішніх посилань або вкладанням посилання в тег "*noindex*".

Для використання всіх можливостей сучасного *CSS* коду, важливо правильно налаштувати потрібні інструменти для розробки. Для прискорення завантаження веб-додатку, бажано максимально зменшити розмір *CSS*-файлів. У наші дні досить часто використовуються інструментальні засоби для зміни *CSS* під час збирання проекту (або постпроцесор, або *PostCSS*), щоб забезпечити резерв для старих браузерів або деякі інші поліпшення. Це може здатися тривіальною проблемою з дуже невеликим ефектом, але в результаті економія, яка складає навіть 3 Кб для невеликої таблиці стилів, позитивно впливає на кінцевий результат. Це безумовно велике поліпшення при дуже невеликих зусиллях. А для великого *CSS* це може мати ще більший вплив.

Після операцій по зменшенню *CSS* коду, часто розробники вбудовують стилі прямо в *HTML*, щоб уникнути необхідності завантажувати будь-які зовнішні таблиці стилів і тим самим заощадити час. Звичайно, включення таблиці стилів розміром 9 Кб (або більшою для більших проектів) на кожну сторінку не дуже ефективно. Тому варто включати тільки ту критичну частину стилів, які необхідні для рендерингу сторінки до першої взаємодії, а завантаження інших стилів відкладати (рис. 2.1). Таким чином, можна використовувати кешування браузера для інших сторінок і прискорити завантаження веб-сторінки.

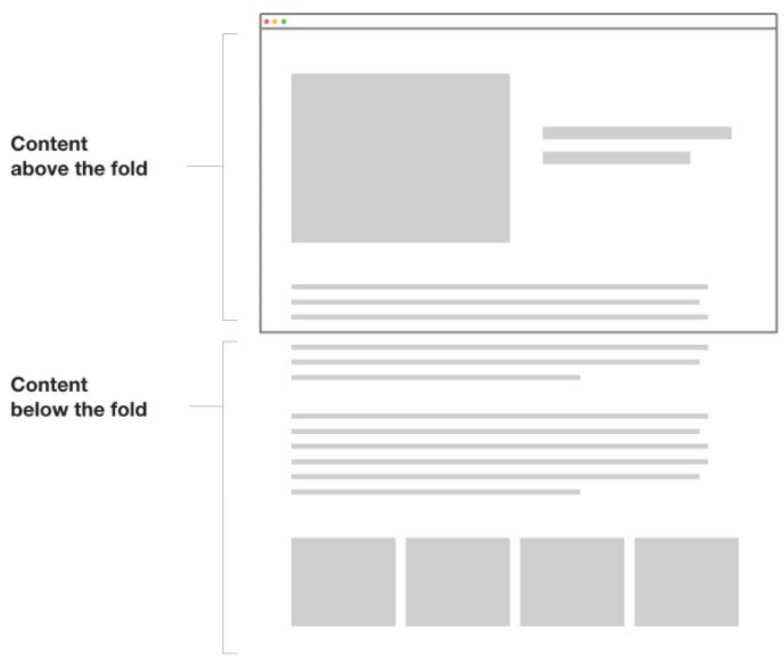


Рис. 2.1 – Рендеринг критичної частини стилів

Ще одним методом оптимізації є розділення *CSS* коду на основі медіа-запитів, сторінок або компонентів. При підході розділення на основі медіа-запитів створюється окремий файл, який містить медіа-запити сторінок та завантажується, коли ширина сторінки відповідає умові медіа-запиту. Інший підхід – використання окремого *CSS* для кожної сторінки для завантаження тільки тих стилів, які використовуються на відкритій сторінці. Більш актуальним методом є розділення *CSS* на компоненти для поступового завантаження *CSS* тільки для тих компонентів, які використовуються на сторінці (футер, хедер, контент і т. д.). Після завершення оптимізації стилів, варто звернути увагу на те, що всі *CSS*-файли мають бути підключені в тегу *head*. Це потрібно для того, щоб файли стилів завантажувались одразу та не заважали рендерингу сторінки, що впливає на швидкість завантаження до першої взаємодії з користувачем.

Оскільки кожний веб-додаток має власний набір скриптів, розробники яких реалізують їх по різному, є дуже велика кількість методів оптимізації JavaScript коду спрямованих на досягнення вищої продуктивності веб-додатку. В основному найбільший вплив на швидкість завантаження веб-сторінок мають скриптові бібліотеки. В процесі розробки часто підключаються ті скрипти, які

потім не використовуються. Наприклад, скрипти для тестування коду або скрипти, які збиралися використовувати, але в процесі розробки відмовились від даного рішення. Такі скрипти зазвичай видаляються при оптимізації веб-додатку. Якщо без деяких скриптів сайт функціональний, то завантаження цих *JS*-скриптів здійснюють після завантаження сторінки. Завантаження скриптів має відбуватись з піддоменів, з інших доменів або з використанням *CDN* – навіть у найбільш просунутих браузерів файли з одного домену завантажуються в обмежене число потоків. Якщо зображень, стилів і скриптів на сторінці досить багато, то виникає черга на завантаження даних. Кількість використовуваних потоків лімітується тільки для домену, тому якщо *JS*-файли будуть завантажуватися з іншого домену (або піддомену), то їх завантаження відбудеться швидше внаслідок паралельності. Спільно використовувані скрипти повинні бути об'єднані в один файл. Завантаження одного файлу в 50 Кб здійснюється швидше завантаження 10 файлів по 5 Кб, віддача таких файлів менше навантажує сервер і стиснення ефективніше працює на великих файлах.

Велике значення має використання *GZIP* для стиснення даних. Сучасні браузери підтримують обробку стислих даних. Оптимальний спосіб – попереднє стиснення використовуваних скриптів на максимальному рівні компресії та віддача веб-сервером попередньо стиснутих файлів. Стиснення «на льоту» навантажує сервер, тому його на навантажених проектах краще не використовувати. Потрібно кешувати скрипти на стороні клієнта, щоб не змушувати користувачів декілька раз завантажувати один скрипт. Якщо великі бібліотеки використовуються без потреби – наприклад, заради якогось елементарного ефекту на головній сторінці сайту, така бібліотека повинна бути видалена. Для реалізації окремих елементів інтерактивності веб-додатку може вистачити 30 рядків простого *JS*-коду, тому використання об'ємної бібліотеки просто нераціонально. *HTML*, *CSS* та *JS* треба стиснути, мінімізувати і оптимізувати. Видалення пробілів, перенесення рядків, скорочення назв змінних

та інші оптимізації значно зменшують розмір файлів скриптів і прискорюють завантаження. При використанні сторонніх бібліотек, потрібно обов'язково підключати саме мінімізовану версію в *production*-оточенні.

Варто приділити особливу увагу коректній оптимізації зображень для пошукових систем. Оптимізація зображень для сайту – це приведення зображень сайту до вимог пошукових систем для їх кращого ранжування в пошуку по зображеннях і, як наслідок, збільшення пошукового трафіку. Зображення з великою вагою завантажуються повільніше, тим самим знижуючи швидкість завантаження сторінки. Для оптимізації зображення, використовують метод стискання їх за допомогою спеціальних сервісів і компресорів. При великій кількості зображень на сторінці, використовується метод *lazy loading* (ліниве завантаження), який відкладає завантаження контенту під час завантаження сторінки. Таким чином завантажуються лише файли, які видно на екрані сторінки, а всі інші файли завантажуються при прокрутці сторінки, коли потрапляють в поле зору користувача.

Проаналізовані методи та підходи до оптимізації безумовно сприяють приросту продуктивності веб-додатку. Такі методи можна використовувати при виникненні тих чи інших поточних проблем з оптимізацією. Оскільки, алгоритми пошукових систем постійно змінюються, такий підхід до оптимізації, може давати різні результати, тому виникає потреба створення методології оптимізації веб-додатків, яка може бути застосована незалежно від вмісту та типу веб-додатків.

2.2. Інструментальні засоби для аналізу продуктивності веб-додатків

В процесі оптимізації веб-додатку, важливу роль відіграють інструментальні засоби для аналізу. Зараз різноманітні сервіси застосовуються розробниками як для спрощення аналізу веб-сторінок, так і для досягнення

кращих показників продуктивності. В мережі існує чимало спеціалізованих програм, які роблять просування веб-ресурсів найбільш динамічним і успішним, незалежно від тематики та специфіки проекту. Безумовно, найважливішим фактором, який впливає на продуктивність веб-додатку, є швидкість завантаження. Тому необхідно проаналізувати сервіси, які можуть надати розробнику звіти та рекомендації щодо поліпшення продуктивності веб-ресурсу. На сьогодні, найвідомішими сервісами для аналізу веб-сторінок є:

- *Pingdom;*
- *WebPagetest;*
- *PageSpeed Insights;*

Pingdom є потужною і багатофункціональною службою моніторингу веб-сайтів. Як і більшість інших конкурентів, *Pingdom* включає різні функції моніторингу доступності сайту, моніторингу реальних користувачів (*RUM*), моніторингу транзакцій, оповіщення та багато іншого. Швидке оповіщення, показники часу безвідмовної роботи і ретельний аналіз першопричин дають розробникам можливість швидко визначити, які проблеми з продуктивністю або транзакціями виникають у їх веб-додатку і чому. Інші переваги системи *Pingdom* – це перегляди сторінок і кількість текстових попереджень. Моніторинг сайту *Pingom* заснований на *Uptime* – доступності *HTML* на сайті, який також включає час очікування і час відгуку. Панель інструментів *Pingdom* орієнтована на лазерне зображення метрик, що показують поточний стан компонентів і транзакцій, що відстежуються на веб-сайті: останні відключення перевірки протягом останніх 24 годин, будь-які поточні інциденти та час завантаження сторінки *RUM* (рис. 2.2).

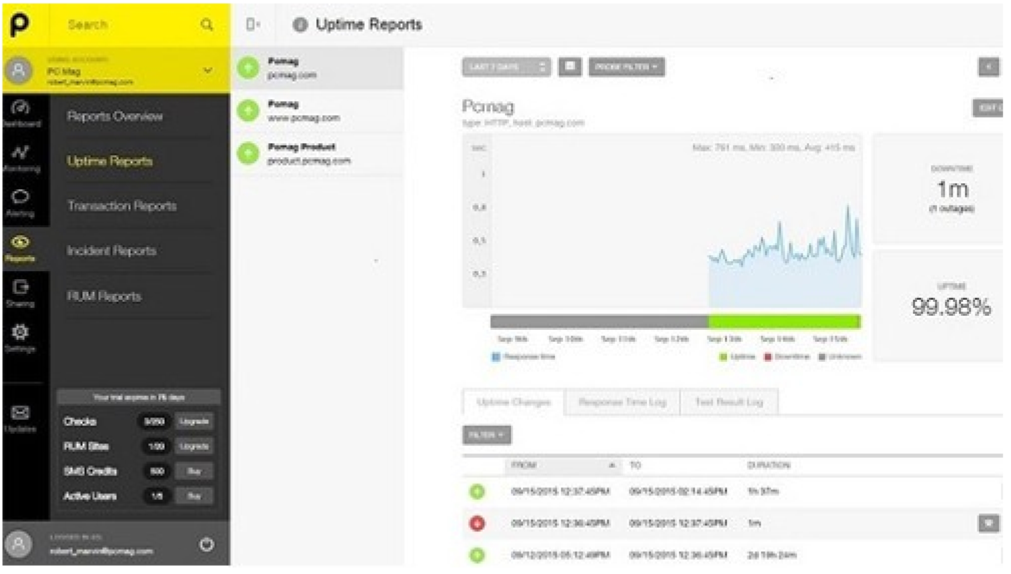


Рис. 2.2 – Панель інструментів веб-додатка *Pingdom*

Зелена стрілка вказує, що веб-додаток працює гладко, а червона стрілка означає, що виникають неполадки. *Pingdom* забезпечує рівномірний розподіл між синтетичним моніторингом продуктивності та *RUM*. Моніторинг часу безвідмовної роботи та транзакцій заснований на синтетичному трафіку, коли сервери *Pingdom* перевіряють веб-додаток або конкретний шлях, скажімо, клієнта електронної комерції на веб-додатку. Нині *Pingdom* пропонує тестування з Північної Америки або Європи. У *Pingdom* також є безкоштовний інструмент під назвою «Тест швидкості веб-сайту *Pingdom*», який використовується для тестування веб-додатку та вимірювання запитів на завантаження сервера в секунду, часу завантаження веб-додатку і розміру сторінки, даючи оцінку продуктивності за шкалою 100 (рис. 2.3).

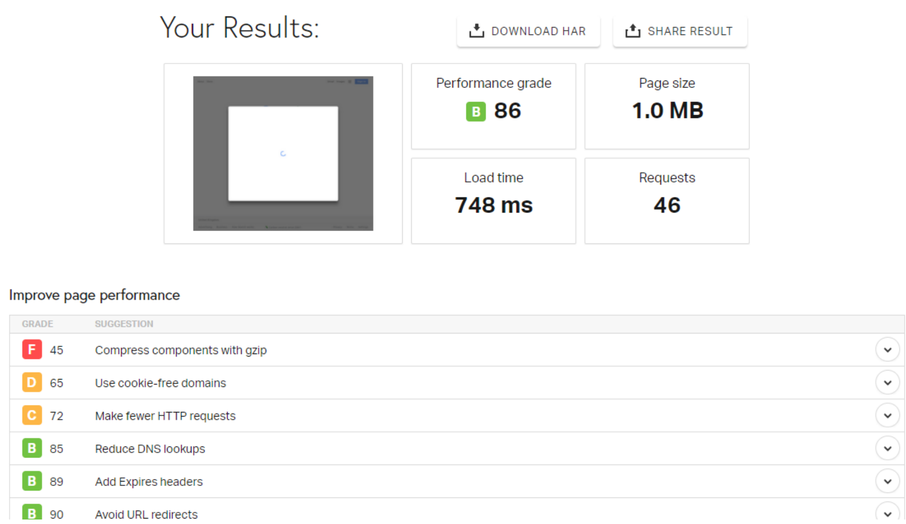


Рис. 2.3 – Безкоштовний інструмент «Тест швидкості веб-сайту *Pingdom*»

WebPageTest – безкоштовний інструмент, який можна використовувати для вимірювання і поліпшення ефективності сайту. По суті *WebPageTest* оцінює те, як завантажується конкретна веб-сторінка. У міру її завантаження записується ряд корисних показників, вони заносяться в каталоги та потім зображаються на різних діаграмах і таблицях, зручних для виявлення прогалин в ефективності. Ці метрики та графіки допомагають ефективніше розв'язувати важливі питання щодо покращення продуктивності веб-додатку. В *WebPageTest* можна контролювати багато аспектів аналізу, такі як, яку платформу використовувати (ПК / мобільний пристрій), який вибрати браузер (*Chrome*, *Firefox*, *IE* та ін.) та присутня можливість визначити географічне положення. Для тестування, потрібно вказати *URL*-адресу веб-додатку та обрати потрібні опції. Опція *Test Location* містить список більш ніж 40 різних регіонів світу. При введенні *URL*-адреси веб-додатку, який тестується, сервер в даному місці буде завантажувати *URL*-адресу локально через браузер, який був обраний розробником. Звичайно, швидкість і надійність тестів може сильно варіюватися від місця до місця. Крім того, як виявилось, не всі регіони підтримують однакові параметри тестування. Опція *Browser* містить велику кількість різних

конфігурацій браузера для ПК і мобільних пристроїв, доступних для тестування в даному місці (рис. 2.4).

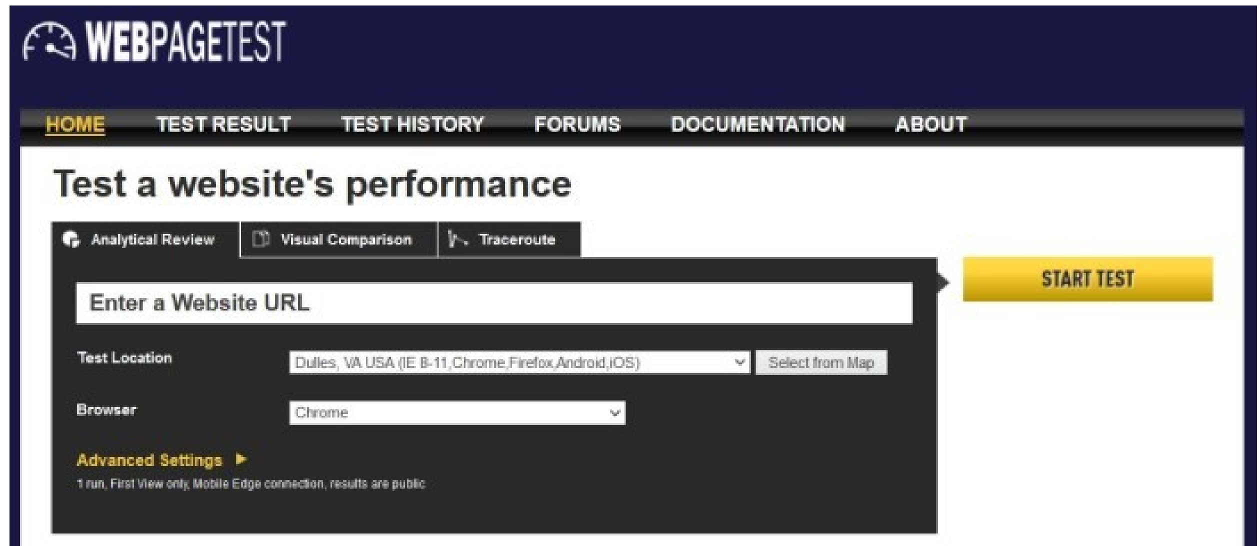
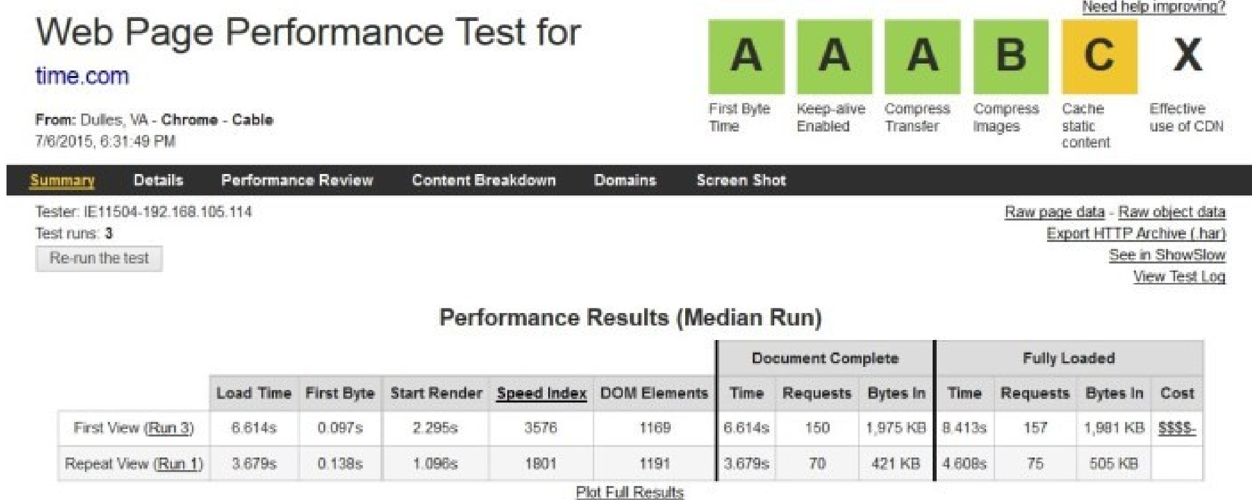


Рис. 2.4 – Інтерфейс вибору опцій для тестування в *WebPageTest*

Потрібно звернути увагу на те, що якщо не вказано інше, мобільні тести в конкретному браузері насправді запускаються на реальних мобільних пристроях, а не на емуляторах. Отримання результату зазвичай займає 30-60 секунд, в залежності від обраних параметрів та розміру веб-додатку. Після завершення тесту, зображається великий обсяг даних (рис. 2.5).



	Load Time	First Byte	Start Render	Speed Index	DOM Elements	Document Complete: Time	Document Complete: Requests	Document Complete: Bytes In	Fully Loaded: Time	Fully Loaded: Requests	Fully Loaded: Bytes In	Fully Loaded: Cost
First View (Run 3)	6.614s	0.097s	2.295s	3576	1169	6.614s	150	1,975 KB	8.413s	157	1,981 KB	$$$$-
Repeat View (Run 1)	3.679s	0.138s	1.096s	1801	1191	3.679s	70	421 KB	4.608s	75	505 KB	

Рис. 2.5 – Результати тестування веб-додатку в *WebPageTest*

Серед них можна знайти такі ключові показники як:

- *First Byte* – показник часу до завантаження першого байту *(Time to First Byte*);
- *Start Render* – точка, коли користувач починає бачити щось крім білого екрана.
- *Document Complete* – точка, де веб-додаток завантажив всі вихідні компоненти *HTML DOM*.
- *Fully Loaded* – точка, в якій веб-додаток завантажився повністю.
- *Speed Index* – середній час завантаження візуальних елементів.

WebPageTest - це незамінний інструмент для пошуку і налагодження проблем з ефективністю сайту, адже за замовчуванням надає ряд ключових показників, важливих для оптимізації веб-додатку.

PageSpeed Insights - це інструмент технічного аналізу сайту від компанії *Google*, який вказує на можливі помилки, які пропустили розробники. *PageSpeed* оцінює роботу сайту за кількома напрямками – від розміру картинок до ступеня стиснення коду. Тобто, по суті, дає більш-менш повну картину проблем, на які варто звернути увагу. Серцем цього інструменту є технологія *Lighthouse*. *Lighthouse* - це автоматизований інструмент з відкритим кодом для підвищення якості веб-сторінок. Він перевіряє ефективність, доступність, прогресивні веб-додатки, *SEO* та багато іншого. Для аналізу веб-додатку, потрібно ввести тільки *URL*-адресу. Після запуску аналізу, *Lighthouse* виконує ряд перевірок щодо веб-додатку, а потім створює звіт. В першу чергу варто сказати, як сервіс в принципі ранжує підсумкові показники. Все оцінюється в балах, від 0 до 100 (рис. 2.6). Чим ближче до сотні, тим краще. Є три градації:

- 0 - 49 (повільно): червона зона;
- 50 - 89 (середнє): помаранчева зона;
- 90 - 100 (швидко): зелена зона.

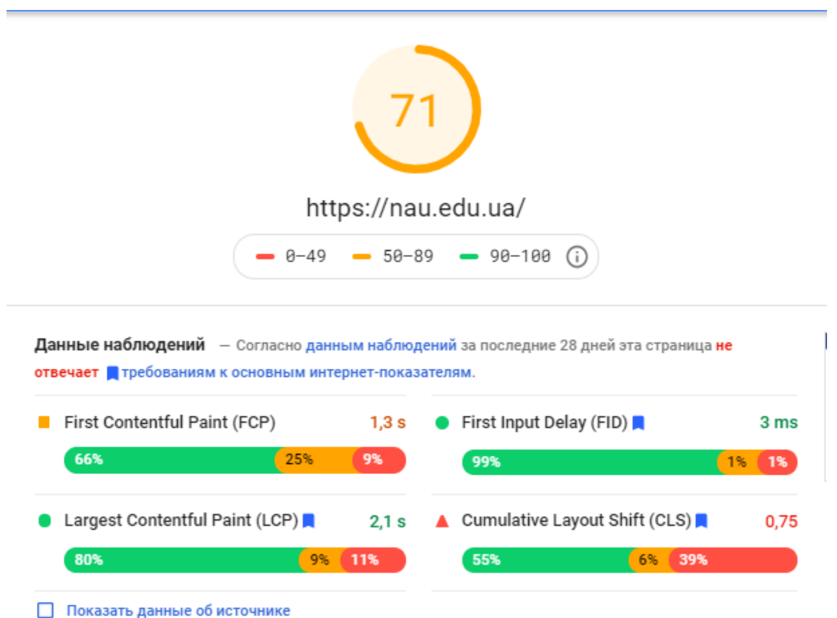


Рис. 2.6 – Оцінка продуктивності веб сторінки в *PageSpeed*

Відповідно, після аналізу інструмент виводить оцінку швидкості роботи веб-додатку і рекомендації щодо поліпшення продуктивності. Причому окремо оцінюється робота сайту для мобільних пристроїв і для комп'ютерів. І рекомендації також видаються окремими вкладками. У звіті зображаються аудити, які пройшли перевірку успішно та аудити які потребують уваги зі сторони веб-розробника (рис. 2.7).

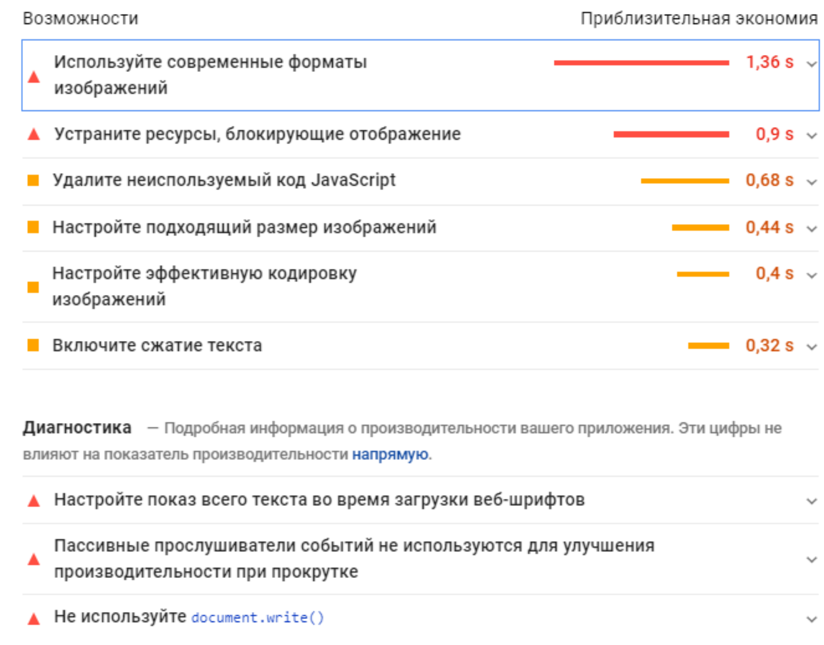


Рис. 2.7 – Аудити які потребують уваги в *PageSpeed*

Кожен аудит має довідковий документ, який пояснює, чому він важливий, а також як це виправити. Замість того, щоб викладати лише загальні рекомендації, *Lighthouse* надає більш відповідні та ефективні поради, залежно від використовуваних інструментів. Пакети стеків дозволяють *Lighthouse* визначати, на якій платформі побудований веб-додаток, та показати конкретні рекомендації на основі стеку. Отже, можна сказати, що даний інструмент несе найбільшу користь для веб-розробника.

Проаналізувавши найпопулярніші інструменти для аналізу продуктивності веб-додатку, можна зробити висновок, що кожен з них може використовуватись при оптимізації веб-додатку. Оскільки, *PageSpeed Insights* показує реальне відношення пошукової системи *Google* до веб-додатку, який аналізується, даний інструмент найбільше підходить для виявлення проблем пошукової оптимізації. Таким чином, можна оперативно виявляти та усувати помилки під час оптимізації веб-додатку.

2.3. Висновок до розділу

Проаналізовано сучасні методи пошукової оптимізації веб-додатків, які включають:

- Комплексний підхід до роботи над *HTML*-кодом.
- Завантаження критичної частини стилів.
- Розділення *CSS*-коду.
- Завантаження скриптів з піддоменів.
- Використання *GZIP* для стиснення даних.
- Метод *lazy loading* для відкладеного завантаження зображень.

Зроблено висновок, що дані методи не повністю розв'язують питання оптимізації. Виявлена потреба в створенні комплексного рішення, тобто методології для оптимізації веб-додатків.

Обрано інструмент для аналізу продуктивності веб-додатків *PageSpeed Insights*, оскільки він найкраще покахує відношення пошукової системи до веб-додатку, який аналізується. Серед критеріїв пошукових систем виділено основні:

- First Byte – показник часу до завантаження першого байту.
- Start Render – точка, коли користувач починає бачити щось крім білого екрана.
- Document Complete – точка, де веб-додаток завантажив всі вихідні компоненти DOM.
- Fully Loaded – точка, в якій веб-додаток завантажився повністю.
- Speed Index – середній час завантаження візуальних елементів.

Досліджено довідкові документи аудитів та рекомендації, що надаються в вигляді звіту після аналізу продуктивності веб-додатку технологією *Lighthouse*. Визначено найбільш ефективні поради, що надає *PageSpeed Insights* та практичну цінність використання даного інструменту при оптимізації веб-додатку

Прийнято рішення створити методологію оптимізації веб-додатків за критеріями пошукової системи *Google*.

РОЗДІЛ 3
РОЗРОБКА МЕТОДОЛОГІЇ ТА ІНСТРУМЕНТАЛЬНИХ ЗАСОБІВ ДЛЯ ОПТИМІЗАЦІЇ ВЕБ-ДОДАТКІВ

3.1. Розробка методології оптимізації веб додатків за критеріями пошукової системи *Google*

Оскільки пошукова система *Google* займає перше місце у світі серед пошукових систем за кількістю користувачів, кожна компанія, яка має власний веб-додаток, знаходиться в постійному пошуку методології, яка дозволять швидко провести пошукову оптимізацію. Це також пов’язано з тим, що технології створення веб-додатків не стоять на місці та кожного року з’являються нові рішення, які допомагають реалізувати проекти різного типу.

Такий стрімкий розвиток напрямку веб-розробки, вимагає нових підходів до оптимізації веб-додатків. Саме тому, для досягнення поставленої мети, а саме, розробки актуальної методології пошукової оптимізації, були взяті за основу критерії компанії *Google*. Для перевірки того, чи відповідає веб-додаток обраним критеріям, було прийнято рішення використовувати інструментальні засоби для аналізу веб-сторінок *PageSpeed Insights*. Щоб зрозуміти за якими критеріями *Google* оцінює веб сторінку, достатньо проаналізувати її в *PageSpeed Insights* і ознайомитися зі звітом. За період пошуку актуальних рішень, був обраний перелік критеріїв, на які потрібно опиратись для досягнення кращої продуктивності веб-додатку.

Важливо розуміти на що *Google* звертає увагу в першу чергу. В цьому питанні допоможе калькулятор оцінки технологією *Lighthouse* (рис 3.1).

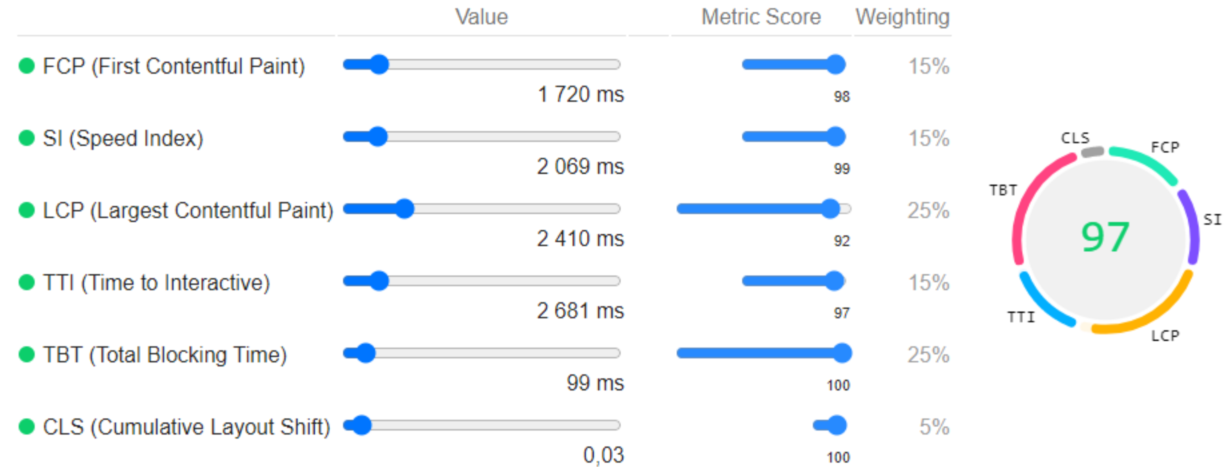
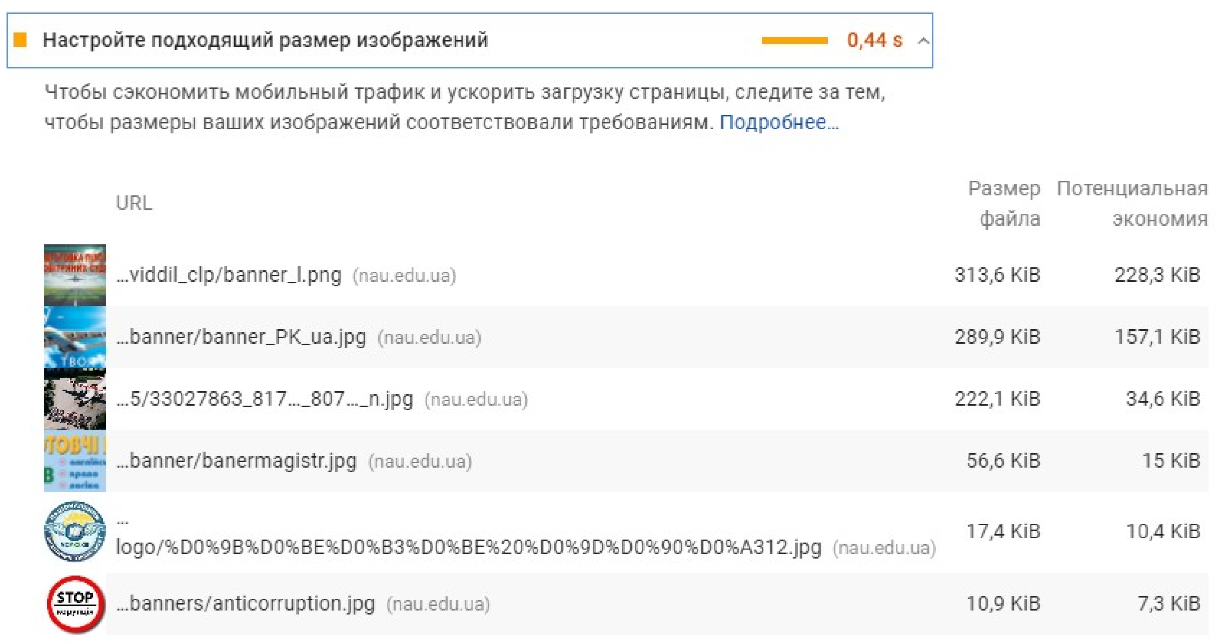


Рис 3.1 – Калькулятор оцінки веб-додатку технологією *Lighthouse*

Отже, можна зробити висновок, що *Google* приділяє увагу таким показникам як:

1) *FCP (First Contentful Paint)* – час до першого показу контенту після білого екрана.
2) *SI (Speed Index)* – індекс швидкості завантаження веб-додатку
3) *LCP (Largest Contentful Paint)* – час до показу найбільших компонентів веб-додатку
4) *TTI (Time to Interactive)* – час до першої взаємодії з користувачем, тобто скільки часу пройде до моменту, коли користувач взаємодіяти з веб-додатком.
5) *TBT (Total Blocking Time)* – загальний час блокування веб-додатку, тобто час, за який були завантажені певні файли, що використовуються на сторінці, котрі блокували взаємодію веб-додатку з користувачем.
6) *CLS (Cumulative Layout Shift)* – загальний зсув макету веб-сторінки.

Дані критерії можна назвати основними при оптимізації веб-додатку, тож вони потребують першочергового втручання веб-розробника. Окрім основних критеріїв, також існують рекомендації для поліпшення показників *PageSpeed Insights* (рис. 3.2), але розглядати їх слід в останню чергу, після основного етапу оптимізації.



Настройте подходящий размер изображений 0,44 s

Чтобы сэкономить мобильный трафик и ускорить загрузку страницы, следите за тем, чтобы размеры ваших изображений соответствовали требованиям. Подробнее...

URL	Размер файла	Потенциальная экономия
...viddil_clp/banner_l.png (nau.edu.ua)	313,6 KiB	228,3 KiB
...banner/banner_PK_ua.jpg (nau.edu.ua)	289,9 KiB	157,1 KiB
...5/33027863_817..._807..._n.jpg (nau.edu.ua)	222,1 KiB	34,6 KiB
...banner/banermagistr.jpg (nau.edu.ua)	56,6 KiB	15 KiB
... logo/%D0%9B%D0%BE%D0%B3%D0%BE%20%D0%9D%D0%90%D0%A312.jpg (nau.edu.ua)	17,4 KiB	10,4 KiB
...banners/anticorruption.jpg (nau.edu.ua)	10,9 KiB	7,3 KiB

Рис 3.2 – Рекомендація *PageSpeed Insights* щодо оптимізації зображень

Більшість веб-розробників, починають оптимізувати веб-додаток після його реалізації. Такий підхід неприпустимий при використанні розробленої методології. Для досягнення найкращого результату, потрібно продумувати процес оптимізації до створення веб-додатку.

Тож початок оптимізації веб-додатку має відбуватись на етапі закладення його архітектури. При такому підході, як правило, розробник витратить вдвічі менше часу на пошук проблем, які можуть виникнути за період тестування веб-додатку. Якщо говорити про сучасні архітектури створення веб-додатків, для пошукової оптимізації найкраще використовувати принцип *Mobile First*. *Mobile First* - це принцип, згідно з яким спочатку розробляється *UI/UX* інтерфейс веб-додатку для розширень, які встановлюються на смартфони. В подальшому дизайн-макету масштабуються під планшетні та *desktop*-пристрої (рис. 3.3).

Даний підхід дозволяє якісно опрацювати зручність мобільної версії сайту, а не за залишковим принципом, як при типовому підході, коли робота над макетами починається з *desktop*-версії.

Рис 3.3 – Схематичне зображення принципу *Mobile First*

Оскільки, за останні 20 років, тенденція росту мобільного трафіку зросла близько до 52,2% від загального трафіку, даний підхід є вірним шляхом до успішного сприймання веб-додатку пошуковими системами. Але головною перевагою даного підходу є відсутність потреби завантажувати одразу всі *CSS* стилі сторінки. При традиційному підході до розробки веб-додатку, що починається з *desktop*-версії, процес адаптації сторінки для мобільних версій відбувається після того, коли *desktop*-версія готова. Тобто якщо користувач буде використовувати веб-додаток на мобільному пристрої, при завантаженні в браузері, стилі для мобільної версії будуть завантажуватись після стилів для *desktop*-версії. В результаті, втрачається швидкість завантаження сторінки. Тому виникає потреба в завантаженні тільки необхідних стилів для певного пристрою. Тож варто використовувати *Mobile First* для полегшення процесу оптимізації в майбутньому, адже при такій архітектурі, на мобільних пристроях

завантажуються лише стилі, які потрібні для зображення контенту на ньому. Стилі для інших пристроїв повинні бути написанні в медіа-запитах і завантажуватись тільки в тому разі, коли умова медіа-запиту відповідає ширині екрану користувача. Таким методом можна не тільки задовольнити критерії пошукових систем, а і підвищити продуктивність веб-додатку на всіх пристроях.

Створення структури веб-додатку, за допомогою *HTML*-коду (чи інших сучасних технологій) та написання стилів *CSS* є найважливішим етапом розробки веб-додатку. Єдиним правильним принципом побудови структури веб-сторінок, будем вважати компонентний підхід. В кожному веб додатку існують елементи, які мають спільне семантичне значення та однакові стилі. До таких елементів можна віднести кнопку замовлення продукту, меню навігації по веб-додатку та інші елементи які можуть дублюватись в різних місцях сторінки, як окремі незалежні компоненти. Головна ідея цього методу, поділити веб-додаток на окремі незалежні компоненти, які можуть повторно використовуватись як на етапі розробки, так і на етапі підтримки та розширювання веб-додатку. Кожен елемент на сторінці повинен мати клас, за яким до нього можна звернутись з файлу *CSS*.

При компонентному підході, можна використовувати методологію БЕМ (блок, елемент, модифікатор), яка була створена компанією Яндекс (рис. 3.4).

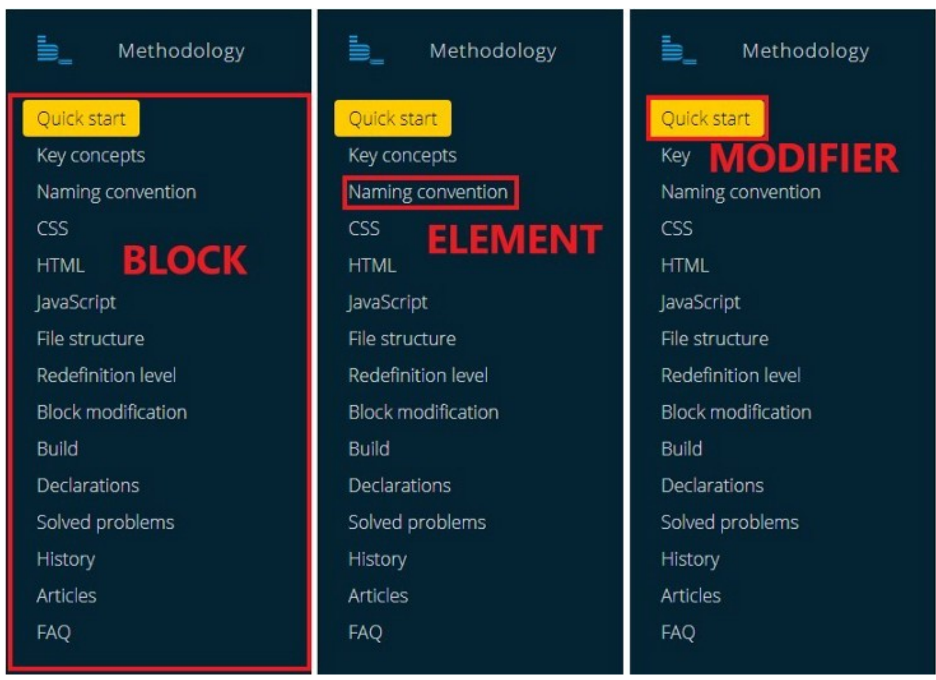


Рис 3.4 – Схематичне зображення методології БЕМ

Суть цієї методології полягає в тому, що кожен компонент (блок) повинен мати унікальну назву класу, а всі його елементи повинні складатись з назви компонента, в якому вони знаходяться, і назви самого елементу. Таким чином ми семантично вірно можемо вказати, що компонент є самостійним та незалежним і має свій набір елементів, які виконують певну функцію та мають сенс тільки в контексті свого блоку. Модифікатор - це те, як ми уявляємо варіанти блоку. На рис. 3.5 наведений приклад компонентного підходу з використанням методології БЕМ.

```
14  <header class="header">        Блок
15      <h1 class="header__title">Едемент1</h1>        Елемент
16      <h3 class="header__subtitle">Едемент2</h3>        Елемент
17      <h3 class="header__subtitle--red">Модифікатор</h3>        Модифікатор
18  </header>
```

Рис 3.5 – Приклад компоненту з використанням методології БЕМ

Найкращим прикладом буде розміри кнопок. Розміри кнопок просто варіації розміру самої кнопки, що робить його модифікатором. При такому підході, об’єм *CSS*-файлу суттєво зменшиться за рахунок відсутності проблеми стилізації подібних компонентів повторно, що зводить до нуля вірогідність дублювання *CSS*-коду.

Для зменшення часу блокування веб-додатку, потрібно відповідально підійти до розробки *JavaScript* коду. Більша кількість розробників, користується готовими рішеннями (бібліотеками) для пришвидшення реалізації певних елементів сторінки. Проблема такого підходу в тому, що *JavaScript* бібліотеки містять в собі, як правило, набір модулів, що надають розробнику можливість більш гнучкого налаштування. Яскравим прикладом є реалізація слайдера на веб-сторінці. Щоб не витрачати час на розробку потрібного слайдера та його функціоналу, часто використовуються бібліотеки, які потребують лише налаштування під потреби розробника, але містять в собі велику кількість коду. В результаті ми отримуємо швидке рішення, яке потребує завантаження великого *JS*-файлу, що приводить до наявності великої кількості коду, який не використовується. Це безпосередньо впливає на індекс завантаження сторінки, час до першого показу контенту і першої взаємодії з користувачем, а також на час блокування контенту. Тобто ми отримуємо негативний вплив відразу на чотири основних критерії оцінки веб-додатку технологією *PageSpeed Insights*. Запропонованим методом, який розв'язує цю проблему є написання потрібного функціоналу без використання стороннього коду. Якщо подібний підхід неможливий, бажано відкладати завантаження стороннього коду до завершення завантаження основного контенту сторінки та використовувати мінімізований код бібліотеки. В разі можливості написання *JS*-коду самостійно, варто враховувати декілька факторів, які дозволять задовольнити критерії *Google*. Насамперед, потрібно слідкувати за тим, чи всі частини коду використовуються веб-додатком і видаляти надлишковий код та створювати більш універсальні конструкції для функцій, які використовуються. Це дозволить зменшити загальний розмір файлу скриптів та позбавить розробника від проблеми надлишкового коду. На рис. 3.6 зображено надлишковий *JS*-код, який виділено червоними відрізками.

```
46          : function(a) {
47              return a && typeof Symbol === "function" && a.constructor === Symbol && a !== (typeof Symbol === "function" ? Sym
48          }
49            , h = function() {
50              function a(a, b) {
51                  for (var c = 0; c < b.length; c++) {
52                      var d = b[c];
53                      d.enumerable = d.enumerable || !1;
54                      d.configurable = !0;
55                      "value"in d && (d.writable = !0);
56                      Object.defineProperty(a, d.key, d)
57                  }
58              }
59              return function(b, c, d) {
60                  c && a(b.prototype, c);
61                  d && a(b, d);
62                  return b
63              }
64          }()
65            , i = function() {
66              function a(a, b) {
67                  var c = []
68                    , d = !0
69                    , e = !1
70                    , f = void 0;
71                  try {
72                      for (var a = a[typeof Symbol === "function" ? Symbol.iterator : "@@iterator"](), g; !(d = (g = a.next()).
73                          c.push(g.value);
74                          if (b && c.length === b)
75                              break
76                      }
77                  } catch (a) {
78                      e = !0,
```

Рис 3.6 – Надлишковий *JS*-код на веб-сторінці

Також потрібно попіклуватись про кешування, адже воно збільшує швидкість і продуктивність веб-додатку шляхом зменшення затримки та зниження мережевого трафіку. Тобто, завдяки кешуванню на показ ресурсу на екрані потрібно менше часу. Кешувати необхідно стилі, скрипти та зображення (рис. 3.7).

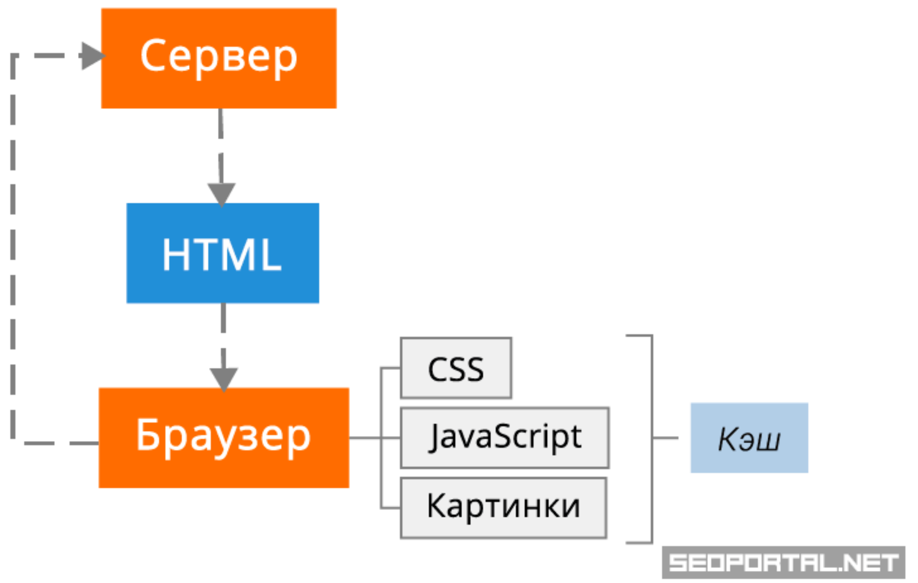


Рис 3.7 – Схематичне зображення правильного кешування

Оскільки *JavaScript* є високорівневою мовою, він піклується про деякі низькорівневі речі, такі як управління пам'яттю. Прибирання сміття - процес, загальний для більшості мов програмування. Грубо кажучи, прибирання сміття - це просто збір та звільнення пам'яті, яка була виділена об'єктам, але в цей час не використовується ні в одній частині веб-додатку. Попри те, що в *JavaScript* прибирання сміття здійснюється автоматично, можуть бути випадки, коли це не ідеальне рішення. В *JavaScript ES6* слід використовувати *Map* і *Set* - *WeakMap* і *WeakSet* (англ. *Weaker* - «слабший»). Ці «слабші» аналоги містять «слабкі» посилання на об'єкти. Вони дозволяють збирати непотрібні значення і запобігати витоку пам'яті.

При оптимізації *JavaScript* коду важливо слідкувати за роботою циклів. Проходження великих циклів займає багато часу. Ось чому завжди слід прагнути виходити з циклу якомога раніше. У цьому допомагають ключові слова *break* і *continue*.

Наступним методом підвищення продуктивності є зниження кількості обчислень змінних. Це реалізується за допомогою замикання. Якщо говорити по-простому, замикання в *JavaScript* надають доступ до області видимості зовнішньої функції з внутрішньої функції. Замикання створюються при створенні функції, а не при її виклику. Внутрішні функції будуть мати доступ до змінних зовнішньої області видимості навіть після того, як зовнішня функція повернеться.

Не завадить також використовувати *throttle* і *debounce*. *Throttling* - це визначення максимального числа запусків функції в певний проміжок часу. Це означає, що навіть якщо користувач викликає 20 раз в секунду одну і ту ж функцію, подія буде спрацьовувати тільки один раз за вказаний розробником проміжок часу. А це зменшить навантаження на код. *Debouncing* - вказівка мінімального проміжку часу після виклику функції, через який функція може бути виконана знову. Це означає, що якщо з останнього виклику цієї функції ще

не пройшов вказаний розробником проміжок часу, то вона буде викликатися повторно. Ці заходи допоможуть знизити час блокування сторінки. До цього також можна віднести потребу в використанні асинхронного коду для запобігання блокування потоків. Для асинхронності використовуються *Promises* (з англ. «обіцянки»). Задля підвищення результатів оцінки часу до першого показу контенту, варто надавати користувачеві на етапі завантажування тільки найнеобхідніші модулі. Це суттєво пришвидшить завантаження сторінки. Якщо після виконання перерахованих методів оптимізації *JS*-коду або використання *JS*-бібліотек, виникає проблема блокування контенту, найефективнішим методом буде використання атрибутів *async* і *defer* при підключенні скриптів. Атрибут *async* вказує браузеру, що завантаження скриптів повинне відбуватись без впливу на рендеринг, а *defer* вказує на необхідність завантаження скрипту після рендерингу. Використовуються ці атрибути в залежності від специфіки проекту.

Важливо приділяти більше уваги зображенням. Маже всі методи оптимізації зображень пов’язані зі стисканням та використанням фактичних розмірів, що відповідають заданим розмірам у файлі *CSS*. При цьому, найчастіше використовуються формати *jpg*, *png* та *svg*. Але при такому підході до оптимізації, ми неодмінно отримаємо негативно оцінку в *PageSpeed Insights* (рис. 3.8).

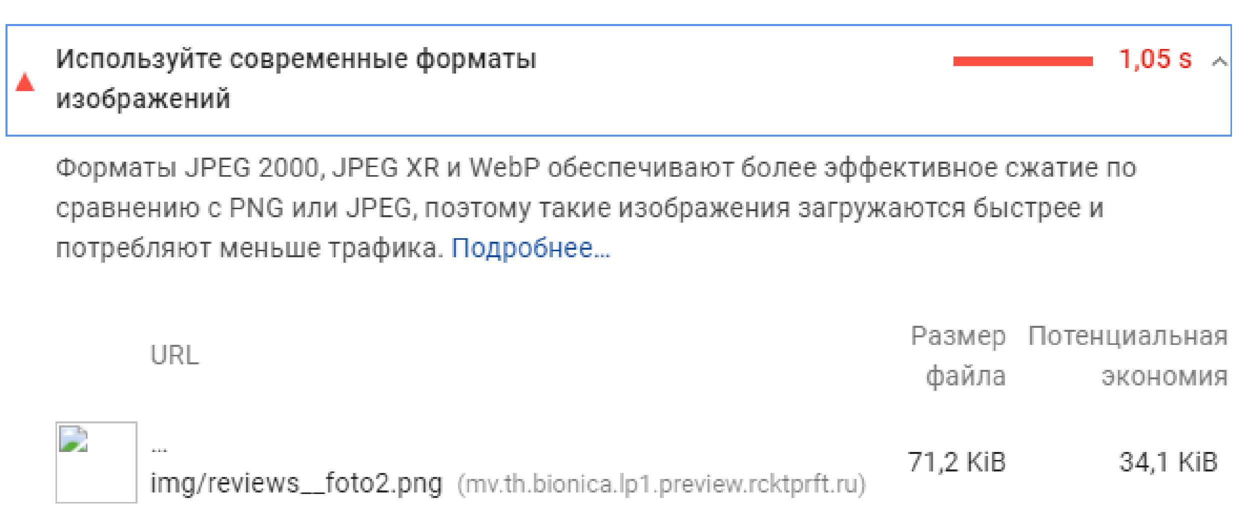


Рис 3.8 – Негативна оцінка при використанні старих форматів для зображень

Зазвичай, цей критерій від *Google* розробники активно ігнорують. Річ у тому, що інструмент аналізу пропонує використовувати сучасні формати зображень, але ці формати підтримують не всі браузери. Отже, виникає проблема кросбраузерності при використанні рекомендованих форматів, а саме: *JPEG 2000*, *JPEG XR*, *WebP*. Звісно, найбільшу перевагу *Google* віддає *WebP*, оскільки це їх власна розробка, яка була присвячена розв'язанню проблеми оптимізації зображень. Методом, який розв'язує проблему використання сучасних форматів зображень, є конструкція *picture* в *HTML*. Конструкція *picture* виступає в ролі контейнеру для набору елементів *source* і одного елементу *img*. В кожному *source* ми можемо вказувати шляхи до потрібних картинок, а якщо браузер не зможе їх зобразити, він перейде до шляху, що вказано за замовчуванням в елементі *img* (рис 3.9).

```
19
20  <picture>
21      <source srcset="img/example-1.webp" type='image/webp'>
22      <source srcset="img/example-1.png" type='image/png'>
23      <img src="img/example-1.jpg" alt="example">
24  </picture>
25
```

Рис 3.9 – Приклад створення конструкції *picture*

Цей метод допомагає вирішити проблему, якщо шлях до зображень вказаний в *HTML*-коді. Якщо браузер не підтримує сучасні формати зображень, він пропустить source, який містить шлях до нього, та обере той формат, який може зобразити. В разі, коли зображення використовуються в *CSS*-коді, знадобиться наступний метод. Перше що потрібно зробити, це додати клас *webp* до тегу *body*. Даний клас будемо використовувати для вказування стилям, підтримує браузер формат *webp* чи ні. Далі потрібно реалізувати функцію, яка буде створювати новий елемент *img* зі шляхом до зображення, розміром в один піксель, формату *webp*, що може зберігатись на сервері. Після цього реалізуємо просту перевірку, яка буде перевіряти висоту створеного зображення. Якщо висота не буде дорівнювати одиниці, ми видаляємо клас *webp* з тегу *body* (рис. 3.10).

```
function checkWebp() {
    const WebP = new Image();
    WebP.src = 'data:image/webp;base64,UklGRjoAAABXRUJQVlA4IC4AAACyAgCdASoCAAIALmk0mk0iIiIiIgBoSygABc6WWgAA/veff/0PP8bA//LwYAAA';
    WebP.onload = WebP.onerror = function() {
        let isWebp = (WebP.height === 1);
        if (!isWebp) {
            document.querySelector('body').classList.remove('webp');
        }
    };
}
checkWebp();
```

Рис 3.10 – Функція для перевірки підтримки формату *WebP*

Отже, при завантаженні сторінки, присутній клас *webp* буде тільки в браузерах котрі підтримують даний формат. Це дозволяє нам побудувати два правила *CSS*, які будуть містити шляхи до зображення *WebP* та до зображення, яке підтримується іншими браузерами (рис. 3.11).

```
.section1 {
    background: url(../images/bg1.jpg) no-repeat center/cover;
}

.webp .section1 {
    background: url(../images/bg1.webp) no-repeat center/cover
}
```

Рис 3.11 – *CSS* правила для кросбраузерного зображення картинки

3.2. Розробка та тестування інструменту для виявлення надлишкового *CSS* коду

Оскільки надлишковий код *CSS* суттєво впливає на *показники PageSpeed Insights* виникає потреба в видаленні коду, який не використовується. Звичайно, на маленьких проектах, де файл стилів не складає десятки тисяч стрічок, доволі легко контролювати появу надлишкового коду та видаляти його вручну. Річ в тому, що кількість таких маленьких додатків знижується майже кожного дня, адже кожен веб-додаток підтримується та поступово розширюється розробниками. Дуже часто виникають випадки, коли розширенням займається зовсім інша команда розробників, яким складніше орієнтуватись в стилях проекту, тому видалення стилів вручну може призводити до появлення нових

помилок чи некоректного зображення елементів. Виходячи з цього, було прийнято рішення розробити додаток, який аналізує сторінку на предмет надлишкових *CSS* правил, та видаляє їх. Було прийнято рішення реалізувати додаток мовою програмування *TypeScript*, яка створена на основі мови *JavaScript*, для можливості використання його в виді веб-сервісу або інтегруванні в збірку проекту. Вибір цієї мови програмування був обраний саме тому, що *TypeScript* є мовою програмування з суворою типізацією. Це надає нам можливість вказувати тип даних для змін, що зберігають відповідні дані та перевіряти їх коректність на кожному етапі обробки. Таким чином, можна легко звузити коло можливих помилок, які можуть з'явитись, як в період розробки, так і при підтримці додатку в майбутньому. Суть роботи додатку полягає в тому, що він аналізує всі файли проекту крім *CSS*-файлів. Це можуть бути файли *HTML* або *JavaScript*. На рис. 3.12 зображено вміст файлу *HTML* та *CSS*, де одразу можна помітити, що в файлі *CSS* знаходяться стилі для селекторів, які не існують в *HTML*-коді.

```html
<!DOCTYPE html>
<html lang="en">
  <body>
    <section class="bg-white">
      <p class="text-black">
        Lorem ipsum dolor sit amet, consectetur...
      </p>
      <button class="btn btn-blue">
        Button
      </button>
    </section>
  </body>
</html>
```

```css
p { font-size: 1.125rem; }
.btn { height: 80px; }
.btn-blue {
  color: white;
  background-color: blue;
}

.text-transparent { color: transparent; }
.text-black { color: #22292f; }
.text-grey-darkest { color: #3d4852; }
.text-grey-darker { color: #606f7b; }
.text-grey-dark { color: #8795a1; }
.text-grey { color: #b8c2cc; }
.text-grey-light { color: #dae1e7; }

.bg-white { background-color: #ffffff; }
.bg-black { background-color: #22292f; }
.bg-grey-darkest { background-color: #3d4852; }
.bg-grey-darker { background-color: #606f7b; }
.bg-grey-dark { background-color: #8795a1; }
```

Рис 3.12 – Вміст файлу *HTML* та *CSS*, що має неіснуючі селектори в *HTML*

Перш за все, був побудований алгоритм для аналізу файлів з подальшим утворенням списку селекторів, які можуть використовуватись на сторінці. Для процесу аналізу файлів був створений модуль, який приймає в якості аргументу об’єкт конфігурації. Конфігурація відбувається наступним чином. В об’єкт конфігурації потрібно передати всі розширення файлів, крім *CSS*, які потребують аналізу. В залежності від розширення файлу вміст файлів з потрібними розширеннями аналізуються за відповідними алгоритмами та утворюють масив всіх *CSS*-селекторів, які можуть використовуватись для стилізації. Цей масив потрапляє в інтерфейс опцій та доступний за ключем content. Ключ *CSS* містить масив всіх селекторів *CSS*-файлу, які будуть проаналізовані за окремим алгоритмом, який буде створено далі. Ключ *output* буде містити в собі результат роботи додатку в строковому вигляді, тобто очищений *CSS* -файл. Також потрібен інтерфейс результату для подання його розробнику в комфортному для нього вигляді (рис. 3.13).

```
60  export interface Options {
61      content: Array<string | RawContent>;
62      css: Array<string | RawCSS>;
63      defaultExtractor: ExtractorFunction;
64      output?: string;
65  }
66
67  export interface ResultFiles {
68      css: string;
69      file?: string;
70  }
```

Рис 3.13 – Інтерфейси опцій та результату

В результаті аналізу формується масив селекторів в строковому вигляді (рис 3.14).

```
[
  'DOCTYPE',
  'html',
  'lang',
  'en',
  'body',
  'section',
  'bg-white',
  'p',
  'class',
  'text-black',
  'Lorem',
  'ipsum',
  'dolor',
  ...
]
```

```
p { font-size: 1.125rem; }
.btn { height: 80px; }
.btn-blue {
  color: white;
  background-color: blue;
}

.text-black { color: #22292f; }

.bg-white { background-color: #ffffff; }
```

Рис 3.14 – Сформований результат аналізу можливих селекторів

Обробка конфігурації перед потраплянням в інтерфейс обробляється сервером на базі *Node.js* та відповідає за обробку команд введених в командну стрічку (рис. 3.15).

```
const program = require("commander");
const fs = require("fs");
const {
    default: DDStyles,
    defaultOptions,
    setOptions,
} = require("../lib/ddStyles");

async function writeCSSToFile(filePath, css) {
    try {
        await fs.promises.writeFile(filePath, css);
    } catch (err) {
        console.error(err.message);
    }
}

program
    .usage("--css <css...> --content <content...> [options]")
    .option("-con, --content <files...>", "glob of content files")
    .option("-css, --css <files...>", "glob of css files")
    .option("-c, --config <path>", "path to the configuration file")
    .option(
        "-o, --output <path>",
        "file path directory to write purged css files to"
    )

program.parse(process.argv);
```

Рис 3.15 – Обробка команд на сервері

Після цього був реалізований модуль парсингу *CSS*-коду, який працює з отриманим масивом селекторів. Інтерфейси аналізують по черзі кожен селектор в *CSS*-коді на предмет наявності його в масиві можливих селекторів. В разі успішного знаходження, переходить до наступного селектора, а в разі помилки видаляє селектор з *CSS*-коду разом з правилами, які до нього відносяться. Результатом виконання модулю персингу є новий *CSS*-файл та *CSS*-код очищений від стилів, які не використовуються веб-додатком. Результати

потрапляють до відповідного інтерфейсу. Після реалізації основних алгоритмів додатку, була додана можливість використання його, як *npm* модулю, та можливість інтегрування його в інструмент для автоматизації *front-end* розробки під назвою *Gulp*. Для створення задачі очищення *CSS*-файлу, достатньо завантажити в проект *npm* модуль та використовуючи синтаксис *Gulp.js* запустити роботу модулю (рис 3.16).

```
const ddStyles = require('ddstyles'),
    ddStylesCleaner = require('ddstyles-cleaner');

gulp.task('ddstyles-cleaner', () => {
    return gulp
        .src('src/**/*.css')
        .pipe(
            ddStylesCleaner({
                content: ['src/**/*.html']
            })
        )
        .pipe(gulp.dest('build/'))
})
```

Рис 3.16 – Інтегрування інструменту в *Gulp.js*

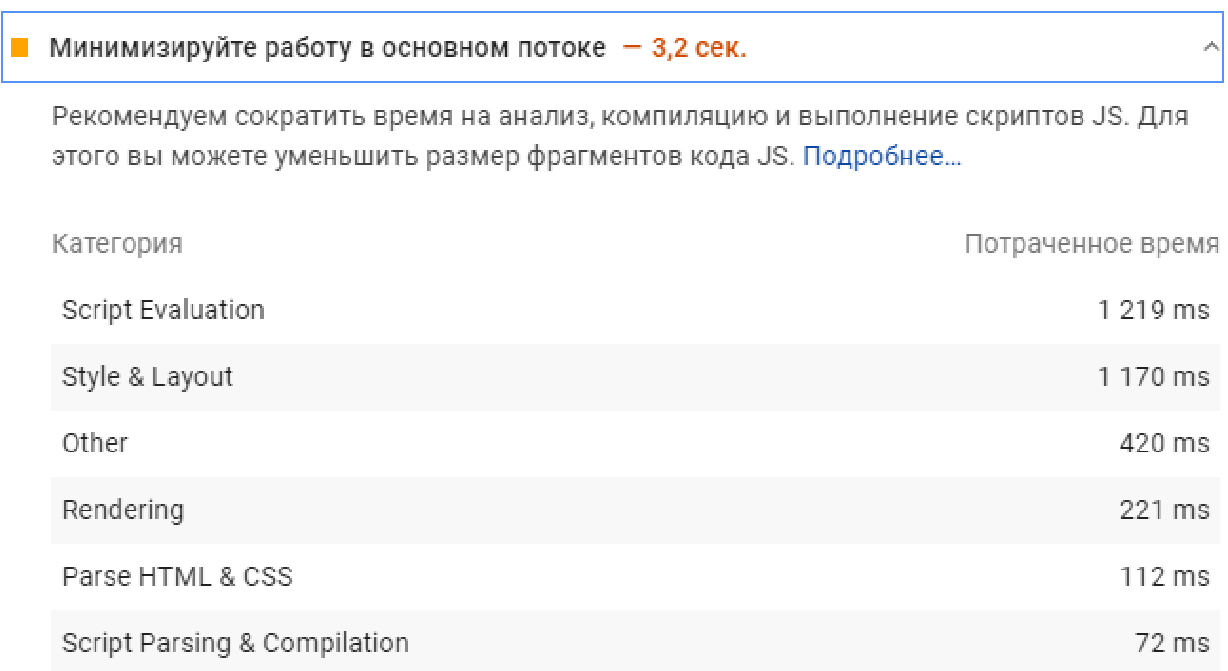
Для тестування розробленого інструменту, був взятий готовий код веб-ресурсу, який було оптимізовано з використанням існуючих методів пошукової оптимізації та підготовлено до завантаження на хостинг. Перш за все, було прийнято рішення проаналізувати веб-ресурс за допомогою інструменту аналізу *PageSpeed Insights*, для чіткого розуміння його стану. На рис. 3.17 зображений результат аналізу сторінки, який виявився незадовільним.

Имитация загрузки страницы

First Contentful Paint	3,4 сек.	Time to Interactive	4,3 сек.
Speed Index	3,4 сек.	Total Blocking Time	340 мс
Largest Contentful Paint	4,5 сек.	Cumulative Layout Shift	0,005

Рис 3.17 – Результати загального аналізу сторінки

Для того, щоб переконатись в тому, що однією із причин негативної оцінки є надлишковий *CSS*-код, було переглянуто звіт в якому показано проблеми з роботою в основному потоці. На рис. 3.18 можна помітити значне витрачення часу на завантаження *CSS*-файлу, що може бути пов'язано саме з наявністю надлишкового коду.


Минимизируйте работу в основном потоке — 3,2 сек.

Рекомендуем сократить время на анализ, компиляцию и выполнение скриптов JS. Для этого вы можете уменьшить размер фрагментов кода JS. Подробнее...

Категория	Потраченное время
Script Evaluation	1 219 ms
Style & Layout	1 170 ms
Other	420 ms
Rendering	221 ms
Parse HTML & CSS	112 ms
Script Parsing & Compilation	72 ms

Рис 3.18 – Звіт про вплив на основний потік з боку *CSS*-файлу до очищення

Тому, починаючи роботу з файлом стилів, в першу чергу було оцінено розмір *CSS*-файлу, який складав 60 Кб та містив в собі 3063 стрічки коду (рис. 3.19).

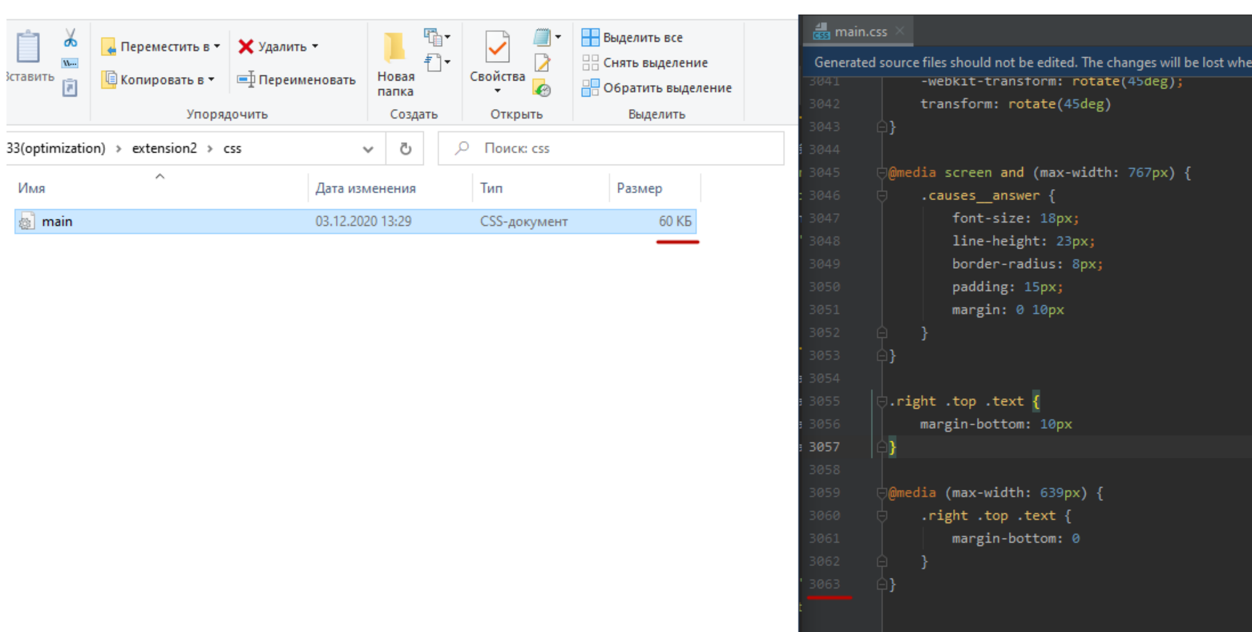


Рис 3.19 – Об’єм та вміст *CSS*-файлу до очищення

Оцінивши розмір файлу стилів, було застосовано розроблений інструмент с цілю видалення надлишкового *CSS*-коду. Оскільки веб-ресурс складався з *html*, *css* та *js* файлів, потрібно передати цю інформацію в виді файлу конфігурації розробленому інструменту. Файл конфігурації в такому випадку буде складатися з контенту, який потрібно аналізувати для виявлення можливих селекторів, які можуть використовуватись в файлі стилів. Контентом мають виступати шляхи до всіх файлів *html* та *js*, які присутні в теці проекту. Після цього, вказуємо шлях до *css*-файлу та шлях за яким буде збережено очищений файл стилів (рис. 3.20).

```
module.exports = {
    content: ['**/*.html', '**/*.js'],
    css: ['css/*.css'],
    output: 'css/new_css.css',
};
```

Рис 3.20 – Вміст файл конфігурації інструмента

Після запуску інструменту за допомогою командної стрічки, через деякий час з’являється новий файл стилів, який ми можемо порівняти зі старим файлом. На рис 3.21 зображено результат роботи інструменту, за допомогу якого, вдалося

видалити надлишкові селектори та правила, що привело до скорочення коду на 32% та до зменшення розміру файлу стилів на 30%.

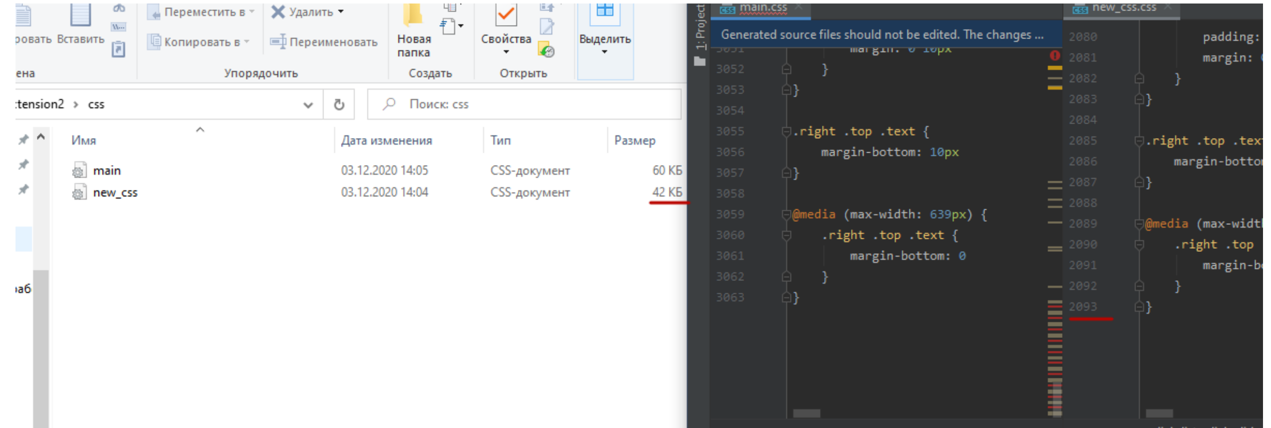


Рис 3.21 – Об’єм та вміст *CSS*-файлу після очищення.

Також був проведений повторний аналіз сторінки, який демонструє скорочення часу на аналіз та виконання *CSS*-коду, в цьому випадку, на 30%.

Категория	Потраченное время
Style & Layout	810 ms
Rendering	616 ms
Other	493 ms

Рис 3.22 – Результати навантаження на основний потік після очищення *CSS*-файлу.

Отже, розроблений додаток працює та може використовуватись при оптимізації файлу стилів будь-якого розміру. Це може бути особливо корисним для розробників, які використовують великі *CSS*-бібліотеки, але не використовують всі їх функції. Також корисно включати даний інструмент в збірку проекту для постійного аналізу та видалення надлишкових стилів, які можуть з’являтись за період розробки чи підтримки веб-додатку.

3.3 Розробка та практична цінність інструменту для генерування корисного JavaScript коду

Наступною глобальною проблемою при оптимізації веб-додатків, слід вважати надлишковий *JS*-код. Головною причиною виникнення *JS*-коду який не використовується є використання розробниками *JS*-бібліотек. Тому виникає потреба розробки інструменту, який може використовуватись авторами великих *JS*-бібліотек та надавати розробнику вибір, яку частину коду обрати для використання в розробці. Розроблений інструмент, має складатись з інтерфейсу для вибору конфігурації коду та логіки, яка залежно від обраної конфігурації, буде генерувати лише корисний *JS*-код. Отже, для розробки інтерфейсу було обрано мову *HTML* та *CSS*. Для реалізації потрібної логіки інструменту, обрано мову програмування *JavaScript*. Для тестування інструменту, використовувалась власна бібліотека для виводу дати в різних форматах та на різних мовах. Насамперед, потрібно оцінити код *JS*-бібліотеки, та умовно поділити функціонал на різні модулі. На рис 3.23 зображено структуру бібліотеки яка буде інтегрована в розроблений інструмент. Вона складається з низки функцій, які надають користувачеві певний функціонал (рис. 3.23).

```
function postDate(daysName, daysMinName, monthsName, monthsMinName, seasonsName) {
    const _counterLength = 60;
    for (let counter = 0; counter < _counterLength; counter++) {...}

    // Функция добавления даты и перемотки через класс "d...
    function innerDate(counter, dateType) {...}
    // Функция изменения формата, принимающая на вход дат...
    function changeFormat(_day, _month, _year, format, counter) {...}
    //Функция выводит название дня, который соответствует дате, указаной в классе date
    function getDaysName(_day, _month, _year, daysName, bigFirstLetter) {...}
    //Функция выводит название месяца, который соответствует дате, указаной в классе date
    function getMonthName(_month, monthsName, bigFirstLetter, counter) {...}
    //Функция выводит название сезона, который соответствует дате, указаной в классе date
    function getSeasonsName(array, month, bigFirstLetter) {...}
    //Функция мотает год (при помощи класса date), который выводиться отдельно при значении data-format="year"
    function getYearWithCounter(year, counter) {...}

    // Фукция вставляет рандомное время в формате 00:00 п...
    function innerCommentTime() {...}
    innerCommentTime();
    //Функция генерации рандомного времени
    function getRandomTime(nodeList, hourLimit) {...}
    //Функция сортировки в порядке возрастания, принимающая на вход массив
    function sortArr(arr) {...}
    //Функция добавляющая "0" в начало строки, если число не является двузначным
    function addZero(numb) {...}
    //Функция меняет первую букву на заглавную, если в объекте dateFormat, при вызове функций, установлен последний аргумент "true"
    function changeFirstLetter(isBig, str) {...}
}
postDate()
```

Рис 3.23 – *JS*-бібліотека, розділена на окремі модулі

Тобто для інтеграції в розроблений додаток, необхідно щоб бібліотека складалась з окремих модулів, які можуть бути необхідні розробнику окремо. Опираючись на функціонал бібліотеки, реалізуємо інтерфейс майбутнього

інструменту. Візьмемо за основу компонентний підхід до розробки та створимо шапку інтерфейсу, яка буде містити назву інструменту (рис. 3.24).

```
<header class="header block">
    <div class="header__wrapper container">
        <h1 class="header__title">PostDate</h1>
        <h3 class="header__subtitle">Constructor</h3>
    </div>
</header>
```

Рис 3.24 – Створення шапки інтерфейсу

Далі нам потрібно реалізувати блок налаштування бібліотеки. Блок налаштування має містити в собі блоки опцій, які доступні для вибору розробника. Оскільки блок опцій може повторюватись на сторінці більше одного разу, будемо вважати що це окремий незалежний компонент сторінки, який може використовуватись повторно (3.25).

```
<section class="settings block">
    <div class="settings__wrapper container">
        <h2 class="settings__title title">Настройки:</h2>

        <div class="option"...>

        <div class="option"...>

        <div class="option"...>

        <div class="option option-type option--off"...>

        <div class="option option-type option--off"...>

        <div class="option"...>

        <button class="settings__btn btn">Сгенерировать код</button>
    </div>
</section>
```

Рис 3.25 – Створення блоків опцій

В блоці опцій потрібно надати розробнику можливість конфігурації. Для прикладу, створимо декілька варіантів вибору певних налаштувань. Перший варіант реалізований за допомогою тегу *select*, який містить список країн, мову яких буде використовувати бібліотека. Такий підхід допоможе відразу генерувати потрібні масиви та не потребує наявності інших масивів.

```
<div class="option">
    <p class="option__desc">Выбор локализации:</p>
    <label>
        <select class="option__select">
            <option value="ar" class="option__item">Арабский язык: Ливан, Саудовская Аравия, ОАЭ, Оман, Катар, Бахрейн, Кувейт</option>
            <option value="bd" class="option__item">Бангладеш</option>
            <option value="bg" class="option__item">Болгария</option>
            <option value="cy" class="option__item">Кипр</option>
            <option value="cz" class="option__item">Чехия</option>
            <option value="de" class="option__item">Германия, Австрия</option>
            <option value="el" class="option__item">Греция</option>
            <option value="en" class="option__item">Англоязычные страны (США, Великобритания etc.)</option>
```

Рис 3.26 – Варіант реалізації вибору опцій за допомогою тегу *select*

Наступним варіантом є реалізація поля input для вводу потрібного значення, яке запускає цикл обходу *DOM* дерева. Це тонка конфігурація, направлена на оптимізацію вихідного коду, що дозволяє зменшити час роботи циклу та виходити з нього раніше при потребі. За замовчуванням ставимо значення 60 (рис. 3.27).

```
<div class="option">
    <p class="option__desc">Максимальное количество дней, которые нужно отнять или прибавить (по умолчанию 60 дней):</p>
    <label>
        <input class="option__input option__input--counter option__select" name="counter" value="60">
    </label>
</div>
```

Рис 3.27 – Варіант реалізації поля для введення потрібного значення

Наступний варіант реалізований для випадків, коли від розробника очікується вибір однієї опції зі списку запропонованих. Для цього використовуємо поля *input*, вказуючи тип *radio*. Оскільки цей варіант очікує від розробника обов’язкового вибору опції, варто залишити одну з них вибраною за замовчуванням, вказавши атрибут checked (рис. 3.28).

```
<div class="option">
    <p class="option__desc">Выбор вывода даты:</p>
    <form action="/" class="form form-type">
        <label class="form__label">
            <input class="form__input form__input--date-type" type="radio" name="dateType" value="defaultMode" checked>
            <span class="form__text">Стандартный (01.12.2020)</span>
        </label>
        <label class="form__label">
            <input class="form__input form__input--date-type" type="radio" name="dateType" value="changeMode">
            <span class="form__text">С изменением формата</span>
        </label>
    </form>
</div>
```

Рис 3.28 – Варіант реалізації вибору однієї з опцій

Останнім варіантом має бути можливість вибору декількох опцій, які не обов’язкові для роботи бібліотеки та можуть бути обрані в залежності від потреб користувача. Для цього створюємо поля input, вказуючи тип *checkbox* (рис. 3.29).

```
<div class="option option-type option--off">
    <p class="option__desc">Выбор доступных форматов даты:</p>
    <form action="/" class="form">
        <label class="form__label">
            <input class="form__input form__input--format" type="checkbox" value="dayFull">
            <span class="form__text">Полное название дня</span>
        </label>
        <label class="form__label">
            <input class="form__input form__input--format" type="checkbox" value="monthFull">
            <span class="form__text">Полное название месяца</span>
        </label>
        <label class="form__label">
            <input class="form__input form__input--format" type="checkbox" value="dayMin">
            <span class="form__text">Сокращенное название дня</span>
        </label>
        <label class="form__label">
            <input class="form__input form__input--format" type="checkbox" value="monthMin">
            <span class="form__text">Сокращенное название месяца</span>
        </label>
        <label class="form__label">
            <input class="form__input form__input--format" type="checkbox" value="monthOnly">
            <span class="form__text">Динамический вывод названия месяца</span>
        </label>
        <label class="form__label"...>
        <label class="form__label"...>
    </form>
</div>
```

Рис 3.29 – Варіант реалізації вибору одразу декількох опцій

Після додавання всіх варіантів опцій, додаємо кнопку підтвердження створеної конфігурації. Кнопка підтвердження має запускати процес генерування коду та зображення його в інтерфейсі. Для зображення коду, створимо відповідний блок, який в майбутньому буде відображати код (рис. 3.30).

```
<section class="code block">
    <div class="code__wrapper container">
        <h3 class="code__title title">Код:</h3>
        <div class="code__output">
            <a id="code" href="" class="code__text">Выберите тип вывода даты и нажмите кнопку "сгенерировать"</a>
        </div>
    </div>
</section>
```

Рис 3.30 – Створення блоку для виводу згенерованого коду

Після завершення реалізації структури інтерфейсу, переходимо до стилізації (рис. 3.31).

```css
.header__wrapper {
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center
}

.header__title {
    display: inline-flex;
    align-items: flex-start;
    font-size: 40px;
    font-weight: 700;
    color: #e9e9e9;
}

.header__subtitle {
    color: #cd2933;
}

.option {
    border: 1px solid #f4b5b8;
    border-radius: 10px;
    padding: 15px;
    margin-bottom: 15px
}
```

Рис 3.31 – Стилізація інтерфейсу

Після завершення етапу стилізації, як результат, ми маємо інтерфейс, що створений для взаємодії з веб-розробником, що адаптований в процесі стилізації для різних пристроїв (рис. 3.32).

Рис 3.32 – Реалізований інтерфейс інструменту

Наступним етапом є розробка логіки інструменту. Для реалізації подібної логіки найкраще підходить створення класу, який буде мати власний конструктор. В конструктор передається конфігурація, що відповідає набору опцій обраних розробником в інтерфейсі інструменту (рис. 3.33).

```
73  class App {
74      //Конструктор класса, принимающий на вход параметры, выбраные пользователем
75      constructor(lang, counter, dateType, format, firstLetter, time) {
76          this.lang = lang; //Язык
77          this.counter = counter; //Счетчик
78          this.dateType = this.radioChecked(dateType); //Активная опция типа даты
79          this.formatArr = this.createFormatArr(format); //Массив выбраных форматов даты
80          this.bigLetter = this.isBigLetter(firstLetter); //Массив выбраных форматов с большой буквы
81          this.time = this.radioChecked(time) // Активная опция наличия таймера
82      }
```

Рис 3.33 – Конструктор конфігурації інструменту

Для обробки деяких значень конфігурації, можна використовувати набір службових методів. На базі цих методів, можна додатково побудувати нові, якщо того потребує інтегрування бібліотеки в інструмент (рис. 3.34).

```
84      //***Служебные методы***
85      //Метод возвращает value активной опции
86      radioChecked(nodeList) {
87          let dateType = '';
88          nodeList.forEach(el => el.checked ? dateType = el.value : false);
89          return dateType
90      }
91
92      //Метод возвращает дополненый массив форматов
93      createFormatArr(formatList) {
94          let formatArr = ['dd', 'mm', 'yyyy'];
95          formatList.forEach(el => el.checked ? formatArr.push(el.value) : false);
96          return formatArr
97      }
98
99      //Метод возвращает массив выбраных форматов с большой буквы
100     isBigLetter(nodeList) {
101         let bigLetterArr = [];
102         nodeList.forEach(el => el.checked ? bigLetterArr.push(el.value) : false);
103         return bigLetterArr
104     }
```

Рис 3.34 – Службові методи інструменту

Отже, маючи набір конфігурації, потрібно створити для кожного окремого функціоналу бібліотеки свій метод, який буде генерувати функцію в залежності від значень, які містяться в конструкторі. Для прикладу візьмемо вибір масиву в залежності від обраної локалізації. Метод вибору мови, містить в собі об’єкт в якому знаходяться масиви на різних мовах. Задача методу перевірити, чи була обрана локалізація та в разі успіху повернути масиви з обраною мовою (рис. 3.35).

```
133                 ro: `const months=['ianuarie','februarie','martie','aprilie','mai','iunie','iulie','august
134                 ru: `const months=["января","февраля","марта","апреля","мая","июня","июля","августа","сент
135                 sk: `const months=['január','február','marec','apríl','máj','jún','júl','august','septembr
136                 sl: `const months=['januar','februar','marec','april','maj','junij','julij','avgust','sept
137                 th: `const months=['มกราคม','กุมภาพันธ์','มีนาคม','เมษายน','พฤษภาคม','มิถุนายน','กรกฎาคม','สิงหาคม
138                 tw: `const months=['一月','二月','三月','四月','五月','六月','七月','八月','九月','十月','十一
139                 ua: `const months=['січень','лютий','березень','квітень','травень','червень','липень','сер
140                 vi: `const months=['tháng một','tháng hai','diễu hành','tháng tư','có thể','tháng sáu','th
141             };
142             return this.lang ? langObg[this.lang.toLowerCase()] : ''
143         }
```

Рис 3.35 – Реалізація вибору локалізації

Таким чином, реалізуємо кожен метод класу, використовуючи залежність від значень змінних в конструкторі. В результаті отримуємо набір методів, які відповідають за різні частини коду бібліотеки та повертають різний код, в залежності від конфігурації, яку обрав розробник. Але для того, щоб зібрати всі ці функції в одне ціле, потрібен метод, який перевіряє значення в конструкторі та створює функцію, що відповідає обраним налаштуванням (рис. 3.36).

```
280     createPostDate() {
281         const postDateFunc = `function postDate(daysName, daysMinName, monthsName, monthsMinName, seasonsName)
282             ${this.setInnerDate()}
283             ${this.dateType !== 'defaultMode' ? this.setChangeFormat() : ''}
284             ${this.setDaysName()}
285             ${this.setMonthName()}
286             ${this.setSeasonsName()}
287             ${this.setYearWithCounter()}
288             ${this.time === 'withTime' ? this.setCommentTime() : ''}
289             ${this.addZero()}
290             ${this.dateType !== 'defaultMode' && this.formatArr.length > 3 ? this.setBigLetter() : ''}
291             }`;
292
293         return this.setLang() + postDateFunc + this.callPostDate()
294     }
```

Рис 3.36 – Метод створення сконфігурованої бібліотеки

Для більш комфортного використання інструменту, реалізуємо функцію поза класом, яка буде відповідати за копіювання в буфер обміну по натисканню на згенерований код (рис. 3.37).

```
39  copyCodeLink.addEventListener( type: 'click',  listener: e => {
40      e.preventDefault();
41      copyToClipboard(copyCodeLink.innerText);
42      modalWindow.classList.add('modal--active');
43      setTimeout( handler: () => {...},  timeout: 1000)
46  });
47
48  // ***Функции интерфейса***...
50  const copyToClipboard = code => {
51      const newTextNode = document.createElement( tagName: 'input');
52      document.body.appendChild(newTextNode);
53      newTextNode.setAttribute( qualifiedName: 'value', code);
54      newTextNode.select();
55      document.execCommand( commandId: 'copy');
56      document.body.removeChild(newTextNode)
57  };
```

Рис 3.37 – Реалізація копіювання коду за натисканням

Останнім етапом розробки є тестування інструменту. Для цього переходимо до інтерфейсу. Спочатку оберемо всі опції, для того, щоб перевірити чи коректний код бібліотеки генерується інструментом та натиснемо кнопку, що відповідає за генерацію коду. В результаті отримуємо повну бібліотеку, яка відповідає оригіналу. При виборі мінімальної кількості опцій, отримуємо зменшену бібліотеку в якій присутні тільки ті функції, які були вказані в інтерфейсі (рис. 3.38).


```
const months=['يناير','فبراير','مارس','أبريل','مايو','يونيو','يوليو','أغسطس','سبتمبر','أكتوبر','نوفمبر','ديسمبر'],monthMin = ['','','','','','','','','','','',''],days = ['الأحد','الاثنين','الثلاثاء','الأربعاء','الخميس','الجمعة','السبت'],seasons,['شتاء','ربيع','صيف','خريف'] = daysMin = ['','','','','','',''],function postDate(daysName, daysMinName,
monthsName, monthsMinName, seasonsName) {const _counterLength = 60;for (let counter = 0; counter < _counterLength; counter++)
{innerDate(counter, 'date-');innerDate(counter, 'date')} function innerDate(counter, dateType) {let newCounter;dateType === 'date-' ?
newCounter = -counter : newCounter = counter; const _msInDay = 86400000, _localDate = new Date(Date.now() + (newCounter *
_msInDay)), _day = _localDate.getDate(), _month = _localDate.getMonth() + 1, _year = _localDate.getFullYear(); const dayDefault =
addZero(_day), monthDefault = addZero(_month), defaultDate = dayDefault + '.' + monthDefault + '.' + _year; const dateClass = dateType +
counter, nodeList = document.querySelectorAll('.' + dateClass); for (let i = 0; i < nodeList.length; i++) {const dateFormat =
nodeList[i].dataset.format;dateFormat !== undefined && dateFormat !== ''? nodeList[i].innerHTML = String(changeFormat(dayDefault,
monthDefault, _year, dateFormat, newCounter)): nodeList[i].innerHTML = defaultDate} } function changeFormat(_day, _month, _year,
```

Рис 3.38 – Результат роботи інструменту

Для того, щоб побачити ефективність цього інструменту для оптимізації, потрібно порівняти два варіанти використання бібліотеки. В традиційному варіанті, розробники мають можливість підключати файл бібліотеки тільки в повному обсязі, використовуючи об'єкт налаштувань згідно з документацією.

Припустимо, що розробнику потрібний мінімальний набір опцій, який він отримує завдяки розробленому інструменту, та порівняємо, як змінився розмір файлу, який він використовує. Отже, в повному обсязі, бібліотека має розмір 9 Кб та містить 116 стрічок коду (рис. 3.39).

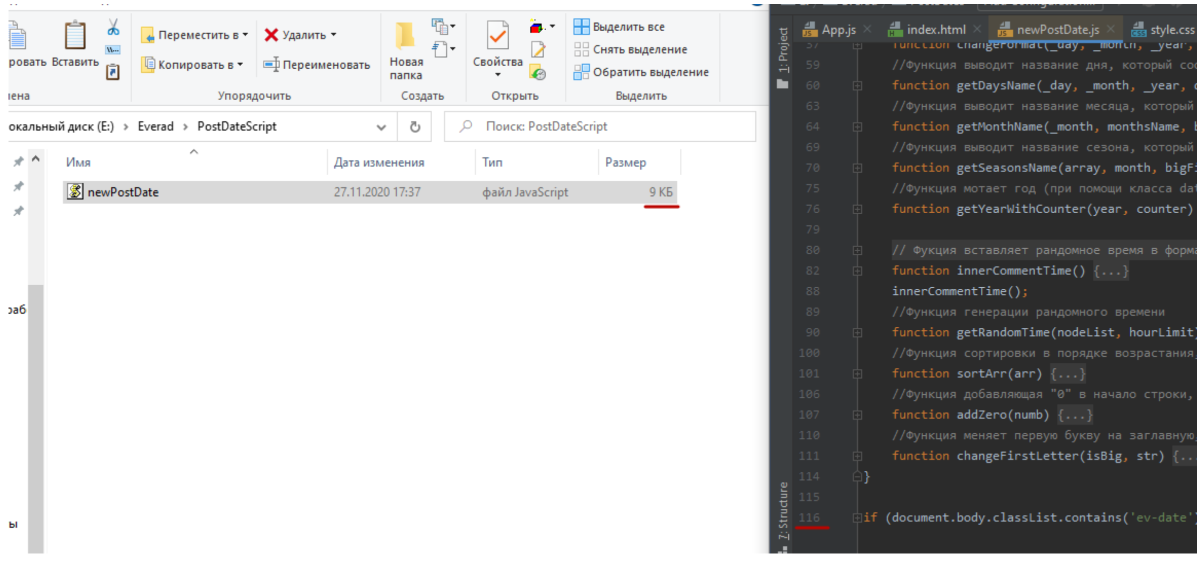


Рис 3.39 – Об'єм бібліотеки в повному обсязі

При використанні мінімального функціонала, який потрібен розробнику для реалізації веб-додатку, розмір бібліотеки буде складати 2 Кб та містити всього 29 стрічок (рис. 3.40).

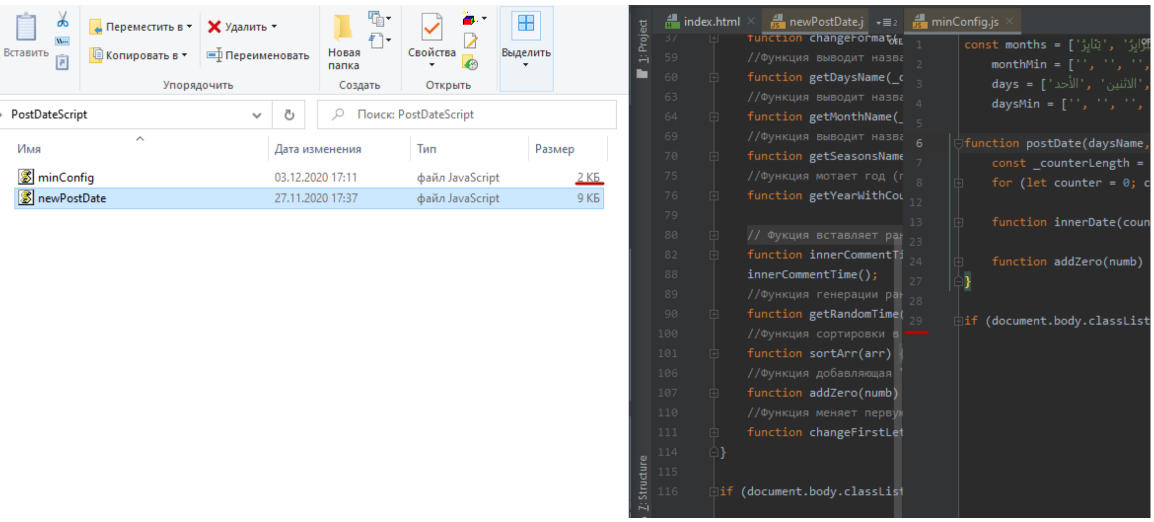


Рис 3.40 – Об'єм бібліотеки в повному обсязі після застосування інструменту

Враховуючи, що при використанні потрібного функціоналу бібліотеки, в цьому випадку, було зменшено розмір *JS*-файлу на 78%, а кількість стрічок коду

зменшено на 75%. Таким чином, браузер буде витрачати на 70% менше часу на аналіз та компіляцію скрипту. Отже, даний інструмент необхідно використовувати розробникам великих *JS*-бібліотек, які містять низку рішень та можуть конфігуруватись для потреб користувача. Такий підхід суттєво впливає на результати аналізу веб-ресурсу. Також, слід зауважити, що розроблений інструмент спростить використання та налаштування бібліотек розробниками, адже має інтуїтивно зрозумілий інтерфейс та не потребує пошуку необхідної інформації в документації.

3.4 . Результати використання розробленої методології та інструментальних засобів для оптимізації веб-додатків

Після розробки методології для оптимізації веб-додатку, доцільно перевірити її реальний вплив на різні веб-додатки. Для цього було обрано три веб-додатки, які відрізняються структурою коду та підходом до розробки. Враховувались прості додатки, що розроблялись малою командою розробників, середні додатки, в розробці яких брало участь декілька команд розробників та великі додатки, що постійно підтримуються різними командами розробників. Такий підхід дозволяє зібрати більш точну статистику, яка дозволить показати вплив розробленої методології на різні веб-додатки з різним підходом до їх розробки.

Для тестування інструментального засобу, який дозволяє генерувати лише корисний код *JavaScript* бібліотек, в дані додатки було інтегровано власну бібліотеку. Оскільки, жоден розробник не використовує весь функціонал певної бібліотеки, за допомогою розробленого інструменту, було згенеровано три її версії, які містили 25%, 50% та 75% від загального вмісту та інтегровано в маленький, середній та великий додаток відповідно. В таблиці 3.1 наведені

результати оптимізації малого веб-додатку, де бал *PageSpeed* є загальним показником оптимізації за критеріями *Google*.

Таблиця 3.1

Веб-додаток	Файли веб-додатку			
	HTML	*CSS*	*JS*	Бал *PageSpeed* (макс. 100)
До оптимізації	49 Кб	60 Кб	50 Кб	84
Після оптимізації	45 Кб	51 Кб	39 Кб	100
Результат оптимізації, %	9%	15%	22%	16%

На основі отриманих даних, побудуємо графік, який показує на скільки збільшилась продуктивність маленького веб-додатку (рис. 3.41).

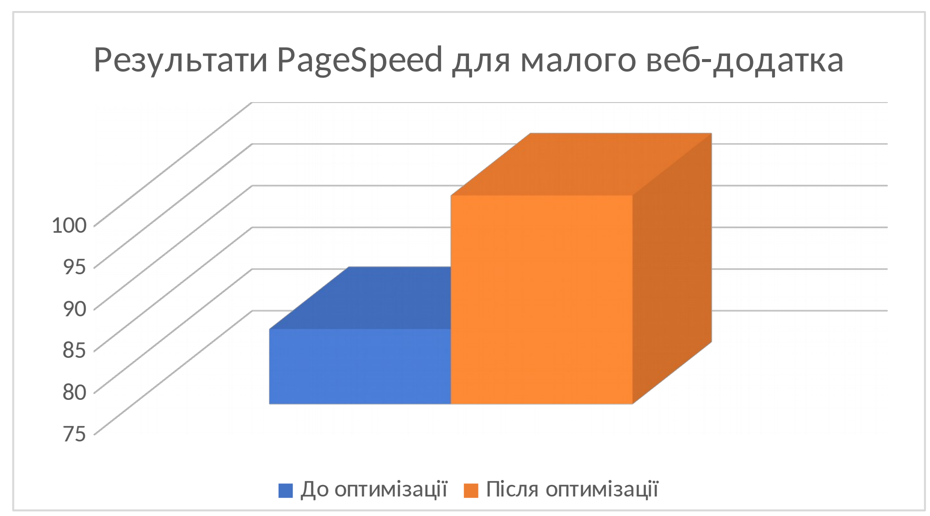


Рис 3.41 – Результат оптимізації малого веб-додатку

Роблячи висновок на основі отриманих даних, можна стверджувати, що застосування розробленої методології в поєднанні з розробленими інструментальними засобами сприяють росту оптимізації малих веб-додатків додатків та задовольняють критерії пошукової оптимізації компанії *Google*.

Наступним веб-додатком, який потрібно протестувати є середній додаток. Оскільки в розробці подібних веб-додатків, зазвичай, залучено декілька команд розробників, цілком можливий ріст надлишкового коду, що використовується на сторінці. Отже, в результаті оптимізації, було отримано такі показники (табл. 3.2).

Таблиця 3.2

Веб-додаток	Файли веб-додатку			
	HTML	*CSS*	*JS*	Бал *PageSpeed* (макс. 100)
До оптимізації	74Кб	107Кб	96Кб	78
Після оптимізації	70Кб	84Кб	74Кб	94
Результат оптимізації, %	6%	22%	23%	18%

Для результату оптимізації середнього веб-додатка, побудуємо графік (рис. 3.42).

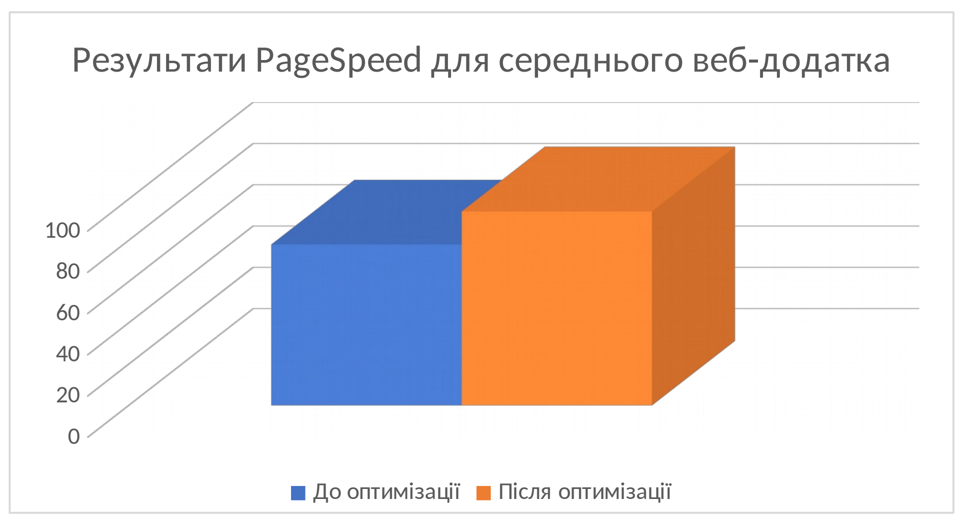


Рис 3.42 – Результат оптимізації середнього веб-додатка

Отримані результати оптимізації середнього додатка також показують приріст продуктивності. При застосуванні інструменту для видалення надлишкового *CSS*-коду, було отримано кращий результат, ніж у випадку середнього веб-додатка. Це пов’язано з більшою кількістю розробників, які працювали над проектом. В результаті такої роботи, може рости відсоток надлишкового *CSS*-коду, що використовується додатком, який впливає на загальну продуктивність.

Останній веб-додаток, який було оптимізовано є найбільшим. Це простий веб-сервіс, який використовується користувачами щодня та потребує постійного втручання веб-розробників. Такі веб-додатки, зазвичай підтримуються різними командами розробників, що спричиняє виникнення

великої кількості надлишкового коду. Зазвичай, в кожній подібній команді є розробники, які слідкують за ростом надлишкового коду та вчасно усувають проблему. При такому підході, багато часу витрачається на пошук проблемних місць в коді веб-додатку. Даний додаток було оптимізовано до втручання спеціалістів в код сторінки та отримано позитивні результати (табл. 3.3).

Таблиця 3.1

Веб-додаток	Файли веб-додатку			
	HTML	*CSS*	*JS*	Бал *PageSpeed* (макс. 100)
До оптимізації	163Кб	257Кб	284Кб	66
Після оптимізації	155Кб	184Кб	242Кб	91
Результат оптимізації, %	5%	29%	15%	28%

На рис. 3.43 показаний графік, де зображений результат оптимізації великого веб-додатка.

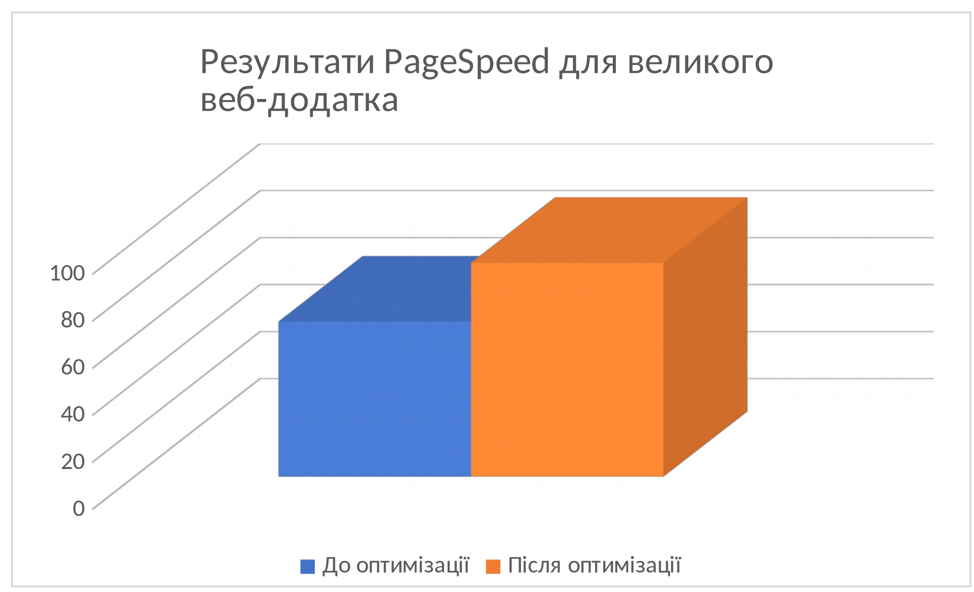


Рис 3.43 – Результат оптимізації великого веб-додатку

Результат оптимізації великого веб-додатка показав задовільний результат, але він не покахує в повному обсязі результат роботи інструментального засобу для генерування корисного *JavaScript* коду. Причиною цього є використання великої кількості *JS*-бібліотек від сторонніх розробників, код яких було мінімізовано в цілях безпеки та оптимізації. Такий код є незрозумілим для розробника додатку, тому він використовує повну версію такої бібліотеки,

конфігуруючи її за інструкцією в документації. Отже, розроблений інструментальний засіб для генерування корисного *JavaScript* коду, може використовуватись розробниками великих *JS*-бібліотек, які мають різноманітний набір функцій. Такі бібліотеки можуть бути інтегровані в розроблений інструмент, що надаватиме веб-розробникам доступ до інтерфейсу конфігурації бібліотеки. Такий підхід суттєво вплине на оптимізацію сучасних веб-додатків. Щоб зрозуміти практичну цінність розробленого інструменту, достатньо припустити, що веб-розробник може використовувати від 25% до 75% функціоналу бібліотеки. Отже, при використанні максимального набору функціоналу бібліотеки, можна зменшити об’єм коду, який використовується на 25%, а при використанні мінімального набору функціоналу, до 75%. Зобразимо на графіку, результат роботи інструменту для генерування корисного *JavaScript* коду (рис. 3.44).

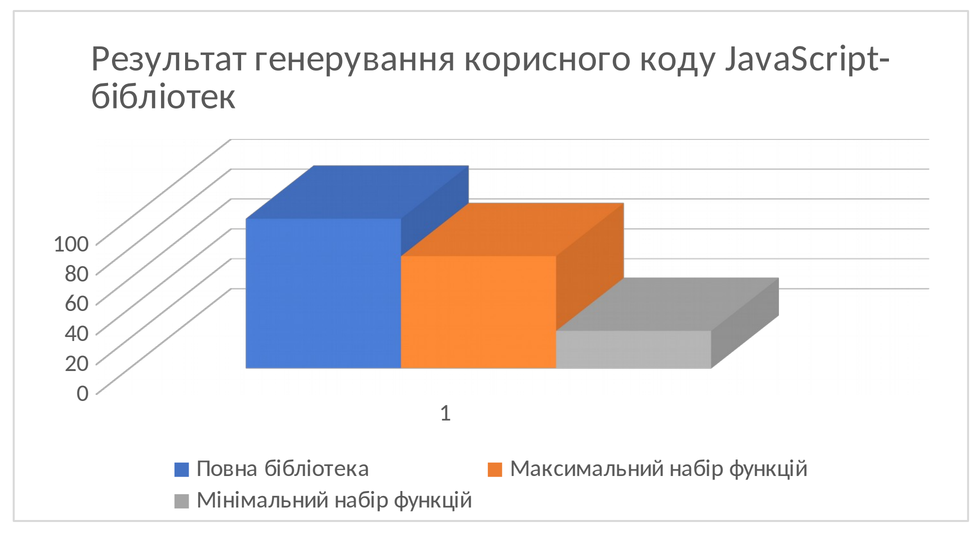


Рис 3.43 – Результат генерування корисного коду *JS*-бібліотек

Отже, можна зробити висновок, що розроблена методологія має позитивний вплив на продуктивність різних типів веб-додатків та допомагає оптимізувати код для пошукових систем. Розроблений інструмент для очищення надлишкового *CSS*-коду, найкращий результат показав в роботі з проектами, де важко слідкувати за виникненням коду, що не використовується веб-додатком. Але не зважаючи на це, він може використовуватись на будь-якому проекті з

цілю автоматизації відстежування та видалення надлишкових стилів. Щодо інструменту для генерування корисного *JS*-коду, варто зазначити, що при інтеграції з різними *JS* бібліотеками, можна надавати можливість веб-розробнику конфігурувати вміст бібліотеки для власних потреб. Це скорочує витрачений час на оптимізацію веб-додатку та розв'язує проблему наявності надлишкового *JS*-коду.

3.5. Висновки до розділу

Розроблено методологію оптимізації веб-додатків за критеріями пошукової системи *Google* на основі отриманих звітів та рекомендацій *PageSpeed*.

Реалізовано інструмент для аналізу та видалення надлишкового *CSS*-коду. Розроблений інструмент може використовуватись при оптимізації *CSS*-файлів, як в ручному режимі запуску, так і в автоматичному при інтеграції в *Gulp.js*. Це має велику практичну цінність для розробників, які використовують великі *CSS*-бібліотеки частково. Також корисно включати даний інструмент в збірку проекту для постійного аналізу та видалення надлишкових стилів, які можуть з’являтись за період розробки чи підтримки веб-додатка.

Розроблено інструментальний засіб для генерування корисного *JavaScript* коду. Інструмент необхідно використовувати розробникам великих *JS*-бібліотек, які містять низку рішень та можуть конфігуруватись для потреб користувача. Такий підхід суттєво впливає на результати аналізу веб-ресурсу. Також, слід зауважити, що розроблений інструмент спростить використання та налаштування бібліотек розробниками.

За допомогою розробленої методології та інструментальних засобів вдалося підвищити продуктивність веб-додатків різного типу та оптимізувати їх код для пошукових систем. Середній приріст продуктивності склав близько 20%.

ВИСНОВКИ

Дипломна робота присвячений актуальній тематиці пошукової оптимізації веб-додатків. В ході виконання дипломної роботи, проаналізовано алгоритм роботи пошукових систем:

- отримання запиту;

- лінгвістичний аналіз, інтерпретація, морфологія, видалення омонімів, додавання синонімів та визначення тематики;
- пошук проіндексованих сторінок з відповідним вмістом, пов'язаним з темою та ключовими словами пошукового запиту;
- створення послідовності видачі з урахуванням ряду факторів;
- передача результатів пошуку користувачам.

Проаналізовано загальну архітектуру пошукових систем. В архітектурі пошукової системи виділено три базові частини:

- Робот – відповідає за збір інформації.
- База даних, в якій зберігається і сортується зібрана роботом інформація.
- Клієнт, де обробляються запити користувача.

Визначена головна задача внутрішньої пошукової оптимізації веб-додатків, а саме, збільшення швидкості завантаження та продуктивності веб-додатка.

Проаналізовано сучасні методи пошукової оптимізації веб-додатків, які включають:

- Комплексний підхід до роботи над *HTML*-кодом.
- Завантаження критичної частини стилів.
- Розділення *CSS*-коду.
- Завантаження скриптів з піддоменів.
- Використання *GZIP* для стиснення даних.
- Метод *lazy loading* для відкладеного завантаження зображень.

Зроблено висновок, що дані методи не повністю розв'язують питання оптимізації. Виявлена потреба в створенні комплексного рішення, тобто методології для оптимізації веб-додатків.

Обрано інструмент для аналізу продуктивності веб-додатків *PageSpeed Insights*, оскільки він найкраще показує відношення пошукової системи до веб-додатку, який аналізується.

Досліджено довідкові документи аудитів та рекомендації, що надаються в вигляді звіту після аналізу продуктивності веб-додатку технологією *Lighthouse*. Визначено найбільш ефективні поради, що надає *PageSpeed Insights* та практичну цінність використання даного інструменту при оптимізації веб-додатку

Розроблено методологію оптимізації веб-додатків за критеріями *Google*, а саме:

- Використання підходу до архітектури *Mobile First.*
- Компонентний підхід до структури веб-додатку.
- Використання методології БЕМ.
- Методи оптимізації *JavaScript* коду.
- Кешування *CSS, JS* та зображень.
- Метод використання сучасного формату *WebP*.

Розроблено алгоритм роботи інструментального засобу для аналізу та видалення *CSS*-коду. Проведено архітектурне проектування програмної системи, яка складається з інтерфейсу та алгоритму генерації корисного *JavaScript* коду.

Протестовано розроблені інструментальні засоби. Отримано позитивні результати використання інструментів в поєднанні з розробленою методологією, а саме:

- Приріст продуктивності тестованих веб-додатків, який склав близько 20%
- Зменшення розміру *CSS*-файлів, шляхом аналізу та видалення надлишкового коду, при умові його присутності в коді веб-додатку.
- Зменшення розміру *JS*-файлу інтегрованої в розроблений інструмент *JS*-бібліотеки внаслідок її конфігурування розробником.

Матеріали дипломного проекту рекомендується використовувати в сучасній веб-розробці для пошукової оптимізації веб-додатків.

Розроблено особисто. Алгоритм роботи інструменту для аналізу та видалення надлишкового CSS-коду, алгоритм генерування корисного JavaScript

коду. Створено інтерфейс та логіку програмного засобу за допомогою TypeScript і JavaScript.

Практичне значення результатів дає змогу користувачам розробленої методології проводити якісну пошукову оптимізацію та підвищувати показники продуктивності веб-додатку.

При виборі теми дипломного проекту, я зупинився саме на цій. Оскільки розроблені інструментальні засоби можуть не тільки показати всі навички, набуті за період навчання, а й бути корисними в майбутньому.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1) *Dover D., Dafforn E. Search Engine Optimization Secrets. Indianapolis: Wiley Publishing, Inc*., 2011. 456 p.

2) Дмитрий Ландэ, Андрей Снарский, Игорь Безсуднов. Интернетика. Навигация в сложных сетях. Модели и алгоритмы. – Либроком, 2009. – 264 с.

3) *Kevin Forsythe, HTML for the Business Developer: with JavaSec.rver Pages, PHP, ASP.NET, CGI, and JavaScript (Business Developers series) / Kevin Forsythe, Laura Ubelhor*. - Москва: Высшая школа, 2008. - 350 с.

4) *Axel, Rauschmayer Speaking JavaScript / Axel Rauschmayer. - M.: O'Reilly Media*, 2014. - 460 с.

5) *David, Herman Effective JavaScript: 68 Specific Ways to Harness the Power of JavaScript (Effective Software Development Series) / David Herman*. - Москва: СИНТЕГ, 2012. - 240 с.

6) *Garann, Means Node for Front-End Developers / Garann Means. - M.: O'Reilly*, 2012. - 642 с.

7) Глушаков, С. В. Программирование *Web*-страниц. *JavaScript*. *VBScript* / С.В. Глушаков, И.А. Жакин, Т.С. Хачиров. - М.: Фолио, 2005. - 390 с.

8) *Eric Enge, Stephan Spencer, Jessie Stricchiola. The Art of SEO: Mastering Search Engine Optimization. - M.: O'Reilly*, 2017. - 994 с.

9) Сергеев. С.Ф. Методы тестирования и оптимизации интерфейсов инфомационных систем. 2013. – 115 с.

10) Введение в информационный бизнес: Учеб. пособие / под ред. Тихомирова В.П. – М.: Финансы и статистика, 1996.

11) Крохина, О. И. Первая книга *SEO*-копирайтера: как написать текст для поисковых машин и пользователей [Текст] / О. И. Крохина, М. Н. Полосина, А. В. Рубель, О. И. Сахно, Е. В. Селин, М. С. Ханина. — М.: «Инфра-Инженерия», 2012. — 216 с.

12) Ашманов, И. С. Оптимизация и продвижение сайтов в поисковых системах [Текст] / И. С. Ашманов. — СПб: Питер Пресс, 2014. — 463 с.

13) Ландэ, Д. В. Интернетика: навигация в сложных сетях: модели и алгоритмы [Текст] / Д. В. Ландэ, А. А. Снарский., И. В. Безсуднов. — М.: Либроком (*Editorial URSS*), 2009. — 264 с.

14) Маннинг, К. Введение в информационный поиск [Текст] / К. Маннинг, П. Рагхаван, Х. Шютце. — М.: Вильямс, 2011.

15) Дударь, З. В. Метаконтекстный поиск в Internet [Текст] / З. В. Дударь, В. С. Хапров, А. В. Мусинов // Восточно-Европейский журнал передовых технологий. — 2005. — С. 104–107.

16) Барсенгян, А.А., Анализ данных и процессов [Текст] / А. А. Барсенгян. — 3-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. – 512 с.

17) Гасанов, Э. Э., Кудрявцев, В. Б. Теория хранения и поиска информации [Текст] / Э. Э. Гасанов, В. Б. Кудрявцев. — М.: Физматлит, 2002. — 288 с.

18) Чапайкина, Н. Е. Семантический анализ текстов: Основные положения [Текст] / Н. Е. Чапайкина // Молодой ученый, Вып. 5. — Казань, 2012. — С. 112–115.

19) Некрестьянов, И.С. Тематико-ориентированные методы информационного поиска [Текст] / И. С. Некрестьянов // Диссертация на соискание степени к. ф-м.н. — СПб.: СПбГУ, 2000.

20) Кент, П. Поисковая оптимизация для чайников [Текст] / Питер Кент. — 4-е изд. — М: Вильямс, 2011. — 421 с.

21) Яковлев, А. А. Раскрутка и продвижение сайтов: основы, секреты, трюки [Текст] / А. А. Яковлев. — СПб: БХВ-Петербург, 2007. — 336 с.

22) Севостьянов, И. О. Посковая оптимизация. Практическое руководство по продвижению сайта в Интернете [Текст] / И. О. Севостьянов. — СПб.: Питер, 2010. — 240 с.

23) Бабаев, А. Раскрутка: секреты эффективного продвижения сайтов [Текст] / А. Бабаев, Н. Евдокимов, М. Боде, Е. Костин, А. Штарев. — СПб.: Питер, 2013. — С. 272.

24) Энж, Э. Искусство раскрутки сайтов [Текст] / Э. Энж. — СПб: БХВ-Петербург, 2011. — 591 с.

25) Введение в СЕО // Статьи для начинающих веб-мастеров [Електронний ресурс]. — 2014. — Режим доступу: http://bigfozzy.com.

26) *The Anatomy of a Large-Scale Hypertextual Web Search Engine* [Електронний ресурс]. – *Sergey Brin*, 2000. – Режим доступу: http://www-db.stanford.edu/pub/papers/google.pdf.

27) Успешный сайт для *Google* за 12 месяцев [Електронний ресурс]. – *Brett Tabke*, 2011. Режим доступу: http://www.searchengines.ru/articles/004523.html.

28) *High Accessibility Is Effective Search Engine Optimization* [Електронний ресурс]. – Andy Hagans, 2005. – Режим доступу: http://alistapart.com/articles/accessibilityseo.

29) Семантическое ядро [Електронний ресурс]. – *GoGetLinks*, 2015. – Режим доступу: http://blog.gogetlinks.net/archives/619.

30) Как составить семантическое ядро сайта [Електронний ресурс]. – Станислав Врублевский, 2016. – Режим доступу: http://great-world.ru/kak-sostavit-semanticheskoe-yadro-sajta/#zag-2.

31) Демина А.В., Ситалиев Д.С., Абдрахманов И.И. Методы продвижения, влияющие на посещаемость молодого сайта [Електронний ресурс] / Современная техника и технологии, 2015. – Режим доступу: http://technology.snauka.ru/2015/04/6450.

32) Юзабилити сайта – анализ, оценка и тестирование [Електронний ресурс]. – Режим доступу: http://www.sembook.ru/book/povyshenie-konversii-sayta/yuzabiliti-sayta.

33) Конверсия сайта [Електронний ресурс]. – Режим доступу: http://1ps.ru/blog/seo/site-conversion/.

34) *Eye-tracking* для оценки пользовательского опыта - Блог *GetGoodRank* [Електронний ресурс]. – Режим доступу: http://blog.getgoodrank.ru/eye-tracking-dlya-ocenki-polzovatelskogo-opyta.

35) *SEO*-словарь *SeoPult* [Електронний ресурс]. — 2015. — Режим доступа: http://www.3ta.

36) *PageRank* [Електронний ресурс]. — 2016. — Режим доступу: https://ru.wikipedia.org.

37) *Google PageRank - A Survey* [Електронний ресурс]. – *Markus Sobek*, 2003. – Режим доступу: http://pr.efactory.de/.

38) *PageSpeed Insights DOCS* [Електронний ресурс]. – *Google*, 2020. – Режим доступу: https://developers.google.com/speed/docs/insights/v5/about.

39) *Pingdom API* [Електронний ресурс]. - *SolarWinds Worldwide*, 2020. . – Режим доступу: https://www.pingdom.com/api/2.1/.

40) *WebPagetest DOCS* [Електронний ресурс]. – Режим доступу: https://github.com/WPO-Foundation/webpagetest-docs/