Міністерство освіти і науки України
Університет митної справи та фінансів

Факультет інноваційних технологій
Кафедра комп'ютерних наук та інженерії програмного забезпечення

Кваліфікаційна робота бакалавра

на тему : «Оптимізація відображення веб-додатку з використанням технологій кешування»

Виконав: студент групи ІПЗ20-1
Спеціальність 121 «Інженерія програмного забезпечення»
Мацапура Дмитро Сергійович
(прізвище та ініціали)
Керівник к.е.н., Яковенко Т.Ю.
(науковий ступінь, вчене звання, прізвище та ініціали)
Рецензент Університет митної справи та фінансів
(місце роботи)
Доцент кафедри кібербезпеки та інформаційних технологій
(посада)
к.т.н., доцент Прокопович- Ткаченко Д.І.
(науковий ступінь, вчене звання, прізвище та ініціали)

Дніпро – 2024

АНОТАЦІЯ

Мацапура Д. С. Оптимізація відображення веб-додатку з використанням технологій кешування.

Кваліфікаційна робота на здобуття освітнього ступеня бакалавр за спеціальністю 121 «Інженерія програмного забезпечення» – Університет митної справи та фінансів, Дніпро, 2024.

Дана кваліфікаційна робота присвячена оптимізації відображення веб-додатку шляхом використання технологій кешування. Збільшення обсягу даних та складність інтерактивних функцій веб-додатків підвищують вимоги до їх продуктивності, що обумовлює необхідність постійного вдосконалення методів забезпечення високої швидкості завантаження та обробки інформації.

В процесі дослідження були проаналізовані різні підходи до рендерингу інтерфейсу користувача, зокрема серверний рендеринг, статичний рендеринг, клієнтський рендеринг та використання WebSockets. Особливу увагу приділено технології кешування, яка дозволяє зменшити час завантаження сторінок, знизити навантаження на сервер та підвищити загальну продуктивність системи.

Розробка додатку проводилась з використанням сучасних технологій програмування, таких як Node.js та JavaScript. Створений додаток забезпечує оптимізацію швидкодії веб-додатку, надаючи користувачам швидке завантаження сторінок та покращений користувацький досвід. В додатку реалізовані методи веб-кешування та Lazy Loading. Це дозволяє користувачам скоротити час завантаження сторінок, зменшити навантаження на сервер та підвищити загальну продуктивність веб-додатків, що сприяє покращенню користувацького досвіду та підвищенню ефективності онлайн-платформ.

Ключові слова: кешування, рендеринг, Lazy Loading, Node.js, JavaScript.

ABSTRACT

Matsapura D. S. Optimization of Web Application Rendering Using Caching Technologies.

Bachelor's thesis for obtaining a degree in Software Engineering, specialty 121. – University of Customs and Finance, Dnipro, 2024.

This bachelor's qualification work is dedicated to optimizing the rendering of a web application using caching technologies. The increase in data volume and the complexity of interactive web application functions heighten performance requirements, necessitating continuous improvement of methods to ensure high loading speed and information processing.

During the research, various approaches to user interface rendering were analyzed, including server-side rendering, static rendering, client-side rendering, and the use of WebSockets. Special attention is given to caching technology, which reduces page load time, decreases server load, and enhances overall system performance.

The application was developed using modern programming technologies, such as Node.js and JavaScript. The created application ensures the optimization of web application performance, providing users with fast page loading and an improved user experience. The application implements web caching methods and Lazy Loading. This allows users to reduce page load times, decrease server load, and increase the overall performance of web applications, contributing to an improved user experience and enhancing the efficiency of online platforms.

Keywords: caching, rendering, Lazy Loading, Node.js, JavaScript.

ЗМІСТ

ВСТУП

У сучасному світі інформаційних технологій ефективність веб-додатків стає критичним фактором успіху будь-якої онлайн-платформи. Збільшення обсягу даних, а також складність інтерактивних функцій веб-додатків значно підвищує вимоги до їхньої продуктивності. Це обумовлює необхідність постійного вдосконалення технологій, методів і підходів, що забезпечують високу швидкість завантаження та обробки інформації. Тому питання оптимізації швидкодії веб-додатків набуває особливої актуальності.

Веб-розробка є складним, тривалим та багатоетапним процесом. Сьогодні існує багато різних методик розробки веб-додатків. Часто як новачки, так і досвідчені розробники фокусуються на сучасних і популярних технологіях, забуваючи про основи роботи Інтернету. Це може призвести до вибору непідходящих технологічних рішень для їхніх проєктів, що негативно позначиться на якості кінцевого продукту.

Веб-додаток складається з взаємопов'язаних веб-сторінок, доступ до яких відбувається через Інтернет за допомогою гіпертекстових посилань. Це електронна система зберігання даних, що може містити текст, зображення, аудіо та відео.

Ефективність рендерингу веб-додатків має вирішальне значення для забезпечення швидкого завантаження, покращення користувацького досвіду та оптимізації для пошукових систем. Різні методи рендерингу мають свої унікальні переваги та компроміси щодо продуктивності, доступності та динамічності контенту.

Алгоритми завантаження веб-сторінок визначають порядок і спосіб завантаження та відображення ресурсів у браузері користувача. Оптимізація цих процесів може значно покращити час завантаження та взаємодію з сайтом.

Серед різноманітних методів оптимізації, особливе місце займає кешування даних. Цей метод дозволяє значно зменшити час завантаження сторінок, знизити навантаження на сервер і підвищити загальну продуктивність

системи. Ефективне використання кешування стає особливо важливим при розробці складних веб-додатків, що мають високий трафік користувачів і великий обсяг динамічного контенту. Головною метою використання кешування є покращення продуктивності системи через мінімізацію звернень до первинного, значно повільнішого джерела зберігання даних. На відміну від традиційних баз даних, які зберігають інформацію на довготривалий термін і в повному обсязі, кеш зберігає тільки частину даних і, як правило, робить це на тимчасовій основі Недостатня увага до цієї технології може призвести до зниження користувацького досвіду, збільшення часу відповіді сервера і, як наслідок, до втрати потенційних користувачів і доходів.

Окрім цього, сучасні веб-додатки мають інтегрувати різноманітні мультимедійні та анімаційні ефекти, що також підвищує вимоги до швидкодії системи. Технології, такі як паралакс, анімаційні списки, а також оптимізація обробки зображень і відео, стають невід'ємною частиною сучасного веб-дизайну. Вони не лише підвищують естетичну привабливість веб-сайтів, але й можуть значно впливати на їхню продуктивність.

Аналіз методів та технологій оптимізації швидкодії веб-сайту має велике значення. Існує чотири основні методи рендерингу інтерфейсу користувача у веб-додатках: серверний рендеринг, статичний рендеринг, клієнтський рендеринг та використання WebSockets.

Таким чином можемо зробити висновок, що тема кваліфікаційної роботи «Оптимізація відображення веб-додатку з використанням технологій хешування» є актуальною.

Метою роботи є оптимізації швидкодії веб-блогу на основі дослідження методів оптимізації завантаження веб-додатків.

Завданнями кваліфікаційної роботи є:

1. Аналіз сучасних методів оптимізації завантаження контенту;
2. Розробка алгоритму оптимізації завантаження веб-додатку;
3. Імплементація алгоритму у веб-додаток;
4. Тестування отриманого рішення;

5. Аналіз отриманого рішення переваги та недоліки;

6. Порівняння отриманих результатів з іншими підходами до вирішення цієї задачі.

Методи та технології дослідження – методи аналізу, синтезу, узагальнення, технології розробки веб-додатків, проектування інтерфейсів.

Об'єкт дослідження – розробка програмного забезпечення для зберігання, завантаження та керування даними на клієнтській частині веб-застосунку.

Предметом дослідження є технології керування локальним станом даних веб-застосунку.

Структура роботи - робота складається з вступу, трьох розділів, списку використаних джерел з 20 найменувань, 21 рисунків

РОЗДІЛ 1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Аналіз літературних джерел за темою кваліфікаційної роботи.

Веб-розробка є складним та багатоетапним процесом, що вимагає глибоких знань і практичних навичок. Існує безліч методик для створення веб-додатків, але часто як новачки, так і досвідчені розробники зосереджуються на сучасних і популярних технологіях, забуваючи про основи функціонування Інтернету. Це може призвести до вибору невдалих технологічних рішень для їхніх проєктів.

Веб-додаток складається з інтегрованих веб-сторінок, доступ до яких здійснюється через Інтернет за допомогою гіпертекстових посилань. Це електронна система зберігання даних, що може включати текст, зображення, аудіо та відео.

Веб-додатки задовольняють різноманітні потреби користувачів у цифровому світі. Ось деякі з основних типів веб-додатків:

1. Інформаційні ресурси – веб-додатки, які надають користувачам різну інформацію від рецептів для приготування їжі до науково-популярних статей.
2. Електронна комерція – інтернет-магазини та платформи для торгівлі й аукціонів, що забезпечують зручність онлайн-покупок.
3. Комунікація та соціальні мережі – веб-додатки, що сприяють соціальній взаємодії такі як Facebook або Twitter.
4. Освітні платформи – сайти, що надають доступ до навчальних матеріалів, курсів та ресурсів, як-от Coursera чи Khan Academy.
5. Розваги та медіа – веб-додатки, що забезпечують доступ до мультимедійного контенту: музика, ігри, фільми. Наприклад Netflix або Spotify.
6. Портфоліо та особисті сторінки – веб-сайти, створені для демонстрації робіт та досягнень.

Розробка структури веб-додатку вимагає ретельного планування, оскільки це впливає на навігацію та загальний досвід користувача. Нижче наведено кілька поширених моделей організації контенту:

1) Стандартна структура

Модель організації контенту має центральний вузол – головна сторінка, від якого розходяться всі інші сторінки. Такий підхід гарантує зручну та інтуїтивно зрозумілу навігацію, дозволяючи користувачам легко знаходити потрібну інформацію та швидко переміщуватися між різними сторінками веб-додатку (рис. 1.1).

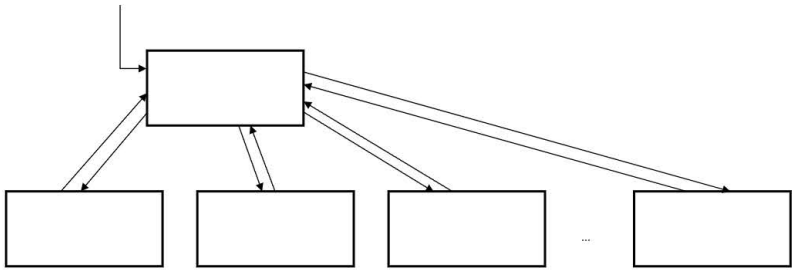

Рисунок 1.1 – Стандартна структура веб-додатку

2) Каскадна структура

Ця модель передбачає послідовне перегортання сторінок, що дає користувачеві можливість переміщатися лише вперед або назад (рис. 1.2). Такий підхід чудово підходить для інструкцій, електронних книг або презентацій, де необхідно дотримуватися логічної та послідовної подачі інформації, забезпечуючи користувачам цілісне розуміння матеріалу. [1].

Рисунок 1.2 – Каскадна структура веб-додатку

3) Структура «Хмарочос»

Модель застосовує ієрархічний підхід до доступу інформації: кожен наступний рівень стає доступним лише після перегляду попереднього (рис. 1.3). Це можна порівняти з подорожжю по поверхах хмарочоса, де кожен поверх розкриває новий набір можливостей та ресурсів.

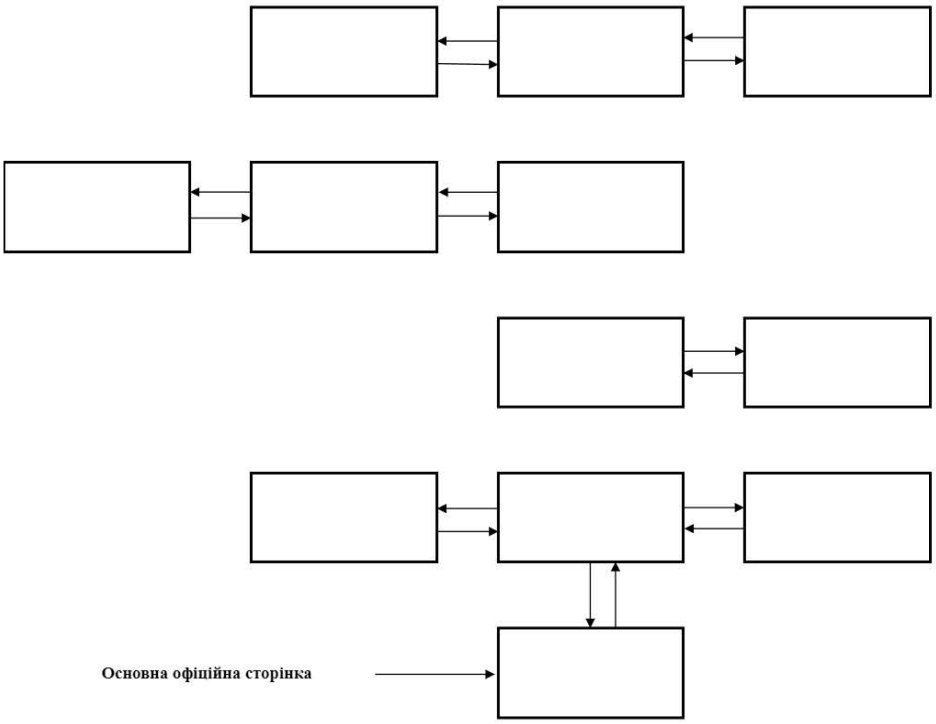

Рисунок 1.3 – Структура сайту типу «Хмарочос»

Кожна структура веб-додатку має свої унікальні переваги та недоліки Вибір відповідної структури є важливим рішенням, оскільки вона має гармонійно відповідати змісту додатку, забезпечувати зручність та ефективність навігації.

Одним із критично важливих рішень для веб-розробників є вибір способу реалізації логіки та відображення графіки в додатку. Це складне завдання через наявність кількох можливих підходів. Щоб прийняти обґрунтоване рішення на етапі розробки, необхідно детально розуміти кожен підхід і користуватися узгодженою та зрозумілою термінологією, що сприятиме успіху проекту.

1.2. Аналіз методів та технологій оптимізації швидкодії веб-сайту

Розглянемо чотири основні методи рендерингу інтерфейсу користувача у веб-додатках: серверний рендеринг, статичний рендеринг, клієнтський рендеринг та використання WebSockets [1].

Серверний рендеринг (SSR) полягає у генерації та відправленні готового HTML-коду клієнту з боку сервера, що значно зменшує обсяги передаваного JavaScript. SSR можна розділити на два підтипи: з гідратацією та без неї. Основна мета SSR – надати користувачу сторінку з вже вбудованими даними, що зменшує потребу в додаткових викликах API для отримання даних, на відміну від клієнтського рендерингу (CSR), де сервер попередньо вставляє всі необхідні дані в HTML-шаблон. Цей підхід дозволяє забезпечити швидке завантаження сторінок і покращити користувацький досвід, особливо на пристроях з обмеженими ресурсами (рис. 1.4).

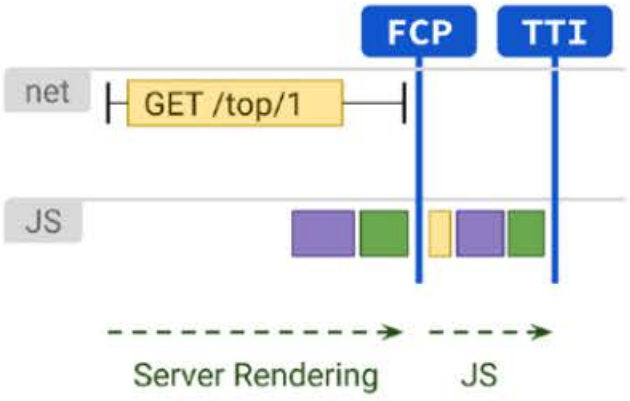

Рисунок 1.4 – Схема SSR рендерингу

Статичний рендеринг – це один із найпростіших і найефективніших методів рендерингу для веб-додатків. При цьому підході сторінки генеруються на етапі компіляції додатку, що забезпечує миттєве завантаження контенту та швидкий перехід до інтерактивності, мінімізуючи необхідність JavaScript на стороні клієнта. Найчастіше цей метод передбачає завчасне створення окремого HTML-файлу для кожної URL-адреси. Завдяки такому підходу, статичні

сторінки можуть ефективно розподілятися через різні CDN, використовуючи переваги кешування. Це дозволяє значно підвищити продуктивність, доступність та надійність веб-додатку, забезпечуючи користувачам бездоганний досвід перегляду (рис. 1.5).

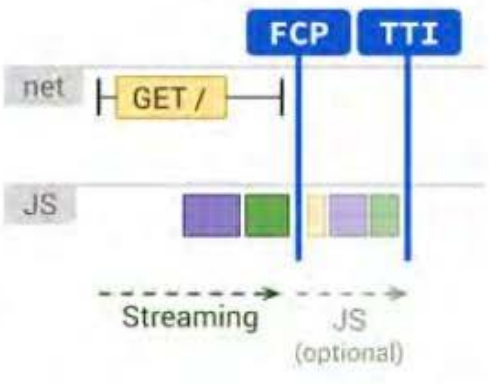

Рисунок 1.5 – Схема статичного рендерингу

Клієнтський рендеринг (CSR) включає створення сторінок безпосередньо в браузері за допомогою JavaScript (рис. 1.6). У цьому підході вся логіка: вибірка даних, створення шаблонів і маршрутизація відбуваються на стороні клієнта. Це дозволяє динамічно оновлювати вміст сторінки без повного перезавантаження забезпечуючи плавний та інтерактивний користувацький досвід [1].

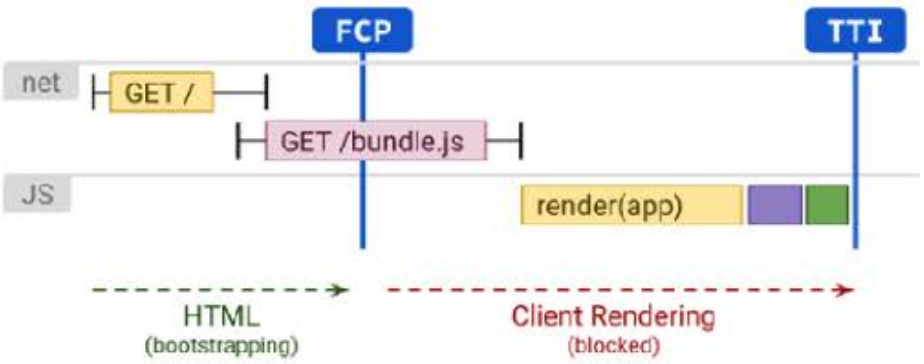

Рисунок 1.6 – Схема CSR

WebSockets – це механізм двостороннього зв'язку, заснований на подіях, який дозволяє встановити постійне з'єднання між клієнтом і сервером. У типовій архітектурі WebSocket клієнтський додаток підключається до WebSocket API, подійної шини або бази даних. Більшість сучасних архітектур використовують його як альтернативу REST, де часте опитування сервера стає неефективним. Завдяки постійному двосторонньому з'єднанню, WebSockets дозволяють миттєво отримувати оновлення з сервера без необхідності створення нових запитів, що забезпечує плавний та інтерактивний користувацький досвід. (рис. 1.7).

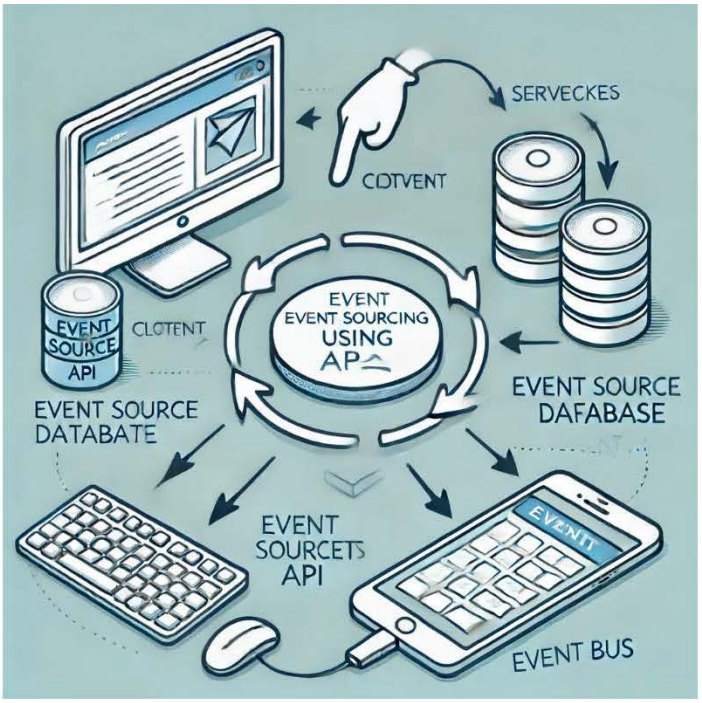

Рисунок 1.7 – Схема WebSocket

Ефективність рендерингу веб-додатків відіграє велику роль у забезпеченні швидкого завантаження, покращенні користувацького досвіду та оптимізації для пошукових систем. Різні методи рендерингу мають свої унікальні переваги та недоліки впливаючи на продуктивність, доступність та динамічність контенту. Зважений вибір підходу до рендерингу може значно підвищити загальну ефективність та успішність веб-додатку [3].

1.3. Висновок до першого розділу

Оптимізація продуктивності веб-сайту є критично важливою для забезпечення швидкого завантаження, покращення користувацького досвіду та досягнення високої видимості в пошукових системах. Вибір методу рендерингу має вирішальне значення і залежить від специфічних потреб та пріоритетів вашого веб-додатку.

Статичний рендеринг та рендеринг на стороні сервера (SSR) забезпечують швидке первісне завантаження сторінок, що особливо корисно для веб-сайтів, де пріоритетом є миттєве відображення контенту та висока пошукова оптимізація.

Клієнтський рендеринг (CSR) – забезпечують динамічний та інтерактивний користувацький досвід, що підходить для додатків, які вимагають багато взаємодії на стороні клієнта.

WebSocket є оптимальним вибором для сценаріїв, що потребують негайного обміну даними, таких як онлайн-ігри, торгові платформи або чати, оскільки він мінімізує затримки та навантаження на сервер.

Проте, важливо враховувати не лише поточні потреби, але й перспективи масштабування проєкту, вимоги до інфраструктури та доступні ресурси для підтримки та розвитку додатку. Баланс між швидкодією, функціональністю та витратами на утримання є ключовим фактором для успішної реалізації оптимізованого веб-додатку.

РОЗДІЛ 2. АНАЛІЗ ТА ВИБІР МЕТОДІВ ВИРІШЕННЯ ПРОБЛЕМИ

2.1. Методи оптимізації швидкодії веб-сайту

Алгоритми завантаження веб-сторінок визначають порядок і спосіб завантаження та відображення ресурсів у браузері користувача. Оптимізація цих процесів може значно покращити час завантаження та взаємодію з сайтом. Розглянемо основні методи оптимізації:

1. Веб-кешування

Веб-кешування, або HTTP кешування, дозволяє тимчасово зберігати копії веб-документів та медіафайлів для мінімізації затримок у відповідях сервера. Це досягається шляхом зберігання запитаних документів, що дозволяє обслуговувати майбутні запити безпосередньо з кешу. Це значно підвищує швидкість завантаження сторінок.

Основна перевага кешування полягає у прискоренні завантаження веб-додатків, що підвищує рівень задоволеності користувачів. Користувачі оцінюють додаток позитивніше, коли сторінки завантажуються швидко.

Динамічний вміст веб-додатків варіюється та адаптується відповідно до конкретного користувача або змінних обставин. Він може залежати від часу відвідування сайту, географічного розташування, типу пристрою, віку, статі та історії переглядів. Такий контент відомий як персоналізований. Інший тип динамічного вмісту змінюється з часом, наприклад, оновлення новин, блогів або списків продукції в інтернет-магазинах.

Традиційно динамічний вміст не кешувався через його змінну природу. Проте, завдяки прогресу у веб-технологіях, тепер можливе кешування динамічного вмісту, що знижує затримки передачі даних та зберігає інтерактивність з користувачем. Винятками є персональні дані користувача, фінансові рахунки та дані про взаємодію з додатком, наприклад, особистий кабінет або історія замовлень [2].

2. Critical Rendering Path

Critical Rendering Path – це послідовність кроків, які браузер виконує для перетворення HTML, CSS та JavaScript у пікселі на екрані. Оптимізація цього процесу значно покращує швидкість завантаження сторінки та час до першого відображення контенту (First Contentful Paint). Для ефективної роботи цього алгоритму потрібно визначити найважливіші ресурси, які користувач повинен побачити в першу чергу, а ресурси з нижчим пріоритетом завантажувати пізніше [8].

Для визначення порядку завантаження сторінки використовується "дерево рендерингу" – комбінація дерев DOM і CSSOM. Браузер аналізує кожен елемент, починаючи з кореня DOM, і визначає, до яких елементів застосовуються CSS-правила. Дерево рендерингу відтворює тільки видимі на сторінці елементи. Наприклад, елементи зі стилем display: none; не включаються до дерева рендерингу.

3. Lazy Loading

Lazy Loading – це техніка, що дозволяє динамічно завантажувати та відображати вміст лише тоді, коли користувач прокручує сторінку до нього. Цей метод покращує користувацький досвід, оскільки зображення та інший вміст завантажуються тільки тоді, коли вони потрібні. Впровадження Lazy Loading у веб-додатках може бути викликом через відсутність вбудованої підтримки та необхідність врахування технічних аспектів, таких як багатопоточність, HTTP-запити, управління пам'яттю та кешування. Незважаючи на переваги, такі як підвищення продуктивності та покращення користувацького досвіду, реалізація лінивого завантаження може ускладнити деякі оптимізації.

Один зі способів реалізації Lazy Loading – поділ коду. Це дозволяє розбити JavaScript, CSS і HTML на менші фрагменти (чанки). Цей підхід дозволяє при початковому завантаженні веб-додатку відправляти лише ті частини коду, які необхідні для відображення стартової сторінки, а додаткові дані завантажувати пізніше за допомогою AJAX-запитів [4].

Ще один метод – використання Intersection Observer API, який асинхронно відстежує, коли цільовий елемент перетинається з видимою частиною сторінки. Це API відкриває нові можливості для оптимізації веб-досвіду, такі як:

- Lazy Loading зображень та іншого контенту.
- Створення сайтів з "нескінченною прокруткою".
- Відстеження видимості рекламних блоків.
- Активація анімацій або дій тільки тоді, коли користувач може їх бачити.

2.2. Вибір методу реалізації

Кеш – це шар пам'яті, спеціально призначений для тимчасового зберігання обмеженого набору даних. Цей технічний засіб дозволяє забезпечувати швидшу відповідь на запити користувачів або систем, скорочуючи час доступу до даних, які розташовані на первинних, більш повільних носіях. Підвищена швидкість обробки запитів досягається завдяки використанню швидкодіючих компонентів, таких як оперативна пам'ять, що дозволяє ефективно використовувати інформацію, яка вже була раніше отримана або обчислена [5].

Головною метою використання кешування є покращення продуктивності системи через мінімізацію звернень до первинного, значно повільнішого джерела зберігання даних. На відміну від традиційних баз даних, які зберігають інформацію на довготривалий термін і в повному обсязі, кеш зберігає тільки частину даних і, як правило, робить це на тимчасовій основі (рис. 2.1).

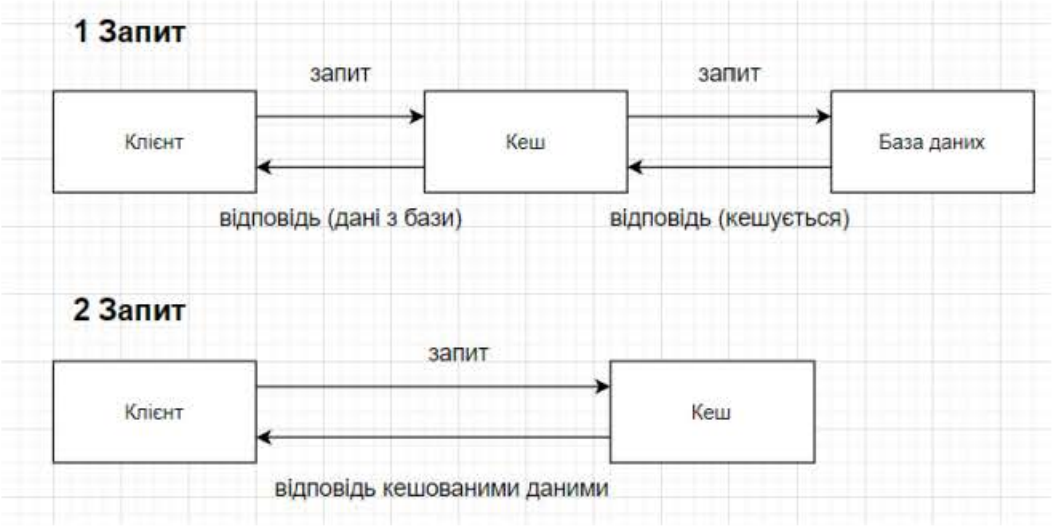

Рисунок 2.1 – Зображено процес звернення до одних і тих же даних користувача до кешу

Такий підхід дозволяє оптимізувати використання ресурсів системи та забезпечити швидший доступ до найчастіше запитуваної інформації. Використання кешування пропонує ряд важливих переваг для покращення роботи програмних додатків та зниження вартості їх експлуатації:

- Швидкість

Доступ до даних з кешу значно швидший ніж з дисків, що скорочує час реакції додатку та покращує загальну продуктивність.

- Економія

Кешування знижує навантаження на бази даних, зменшуючи потребу в потужних та дорогих рішеннях для зберігання даних. Це знижує загальні витрати на інфраструктуру.

- Стабільність

Кешування вирівнює пікові навантаження роблячи роботу програми передбачуваною навіть підчас пікового навантаження, що гарантує високу доступність та швидкість обробки даних [6].

2.3. Висновок до другого розділу

Оптимізація завантаження веб-сторінок є критично важливою для покращення користувацького досвіду та загальної ефективності веб-додатків. Використання технік, таких як веб-кешування, оптимізація Critical Rendering Path та Lazy Loading, може значно прискорити завантаження контенту, знизити навантаження на сервери та покращити взаємодію користувачів зі сторінкою.

З усіх доступних методів було обрано веб-кешування як основний підхід для оптимізації блогу. Це рішення було прийнято через його значні переваги:

1. Підвищення швидкості завантаження
2. Економія ресурсів
3. Стабільність та надійність
4. Покращення користувацького досвіду

Головною метою використання кешування є покращення продуктивності системи через мінімізацію звернень до первинного, значно повільнішого джерела зберігання даних. На відміну від традиційних баз даних, які зберігають інформацію на довготривалий термін і в повному обсязі, кеш зберігає тільки частину даних і, як правило, робить це на тимчасовій основі.

РОЗДІЛ 3. РОЗРОБКА ТА ТЕСТУВАННЯ ВЕБ-БЛОГУ

3.1. Концепція проекту

Розробка алгоритму кешування важливим завданням в сучасну інформаційну епоху, де швидкість доступу до даних є одним з вагомих факторів для задоволення потреби користувачів. Кешування дозволяє суттєво зменшити час завантаження веб-сторінок та відповіді на запити користувачів. Це особливо актуально для сайтів з великим трафіком, де кожна секунда на рахунку для забезпечення користувацького задоволення. Для розробки оптимізованого алгоритму кешування сторінок сайту було створено таку структуру проекту (рис. 3.1):

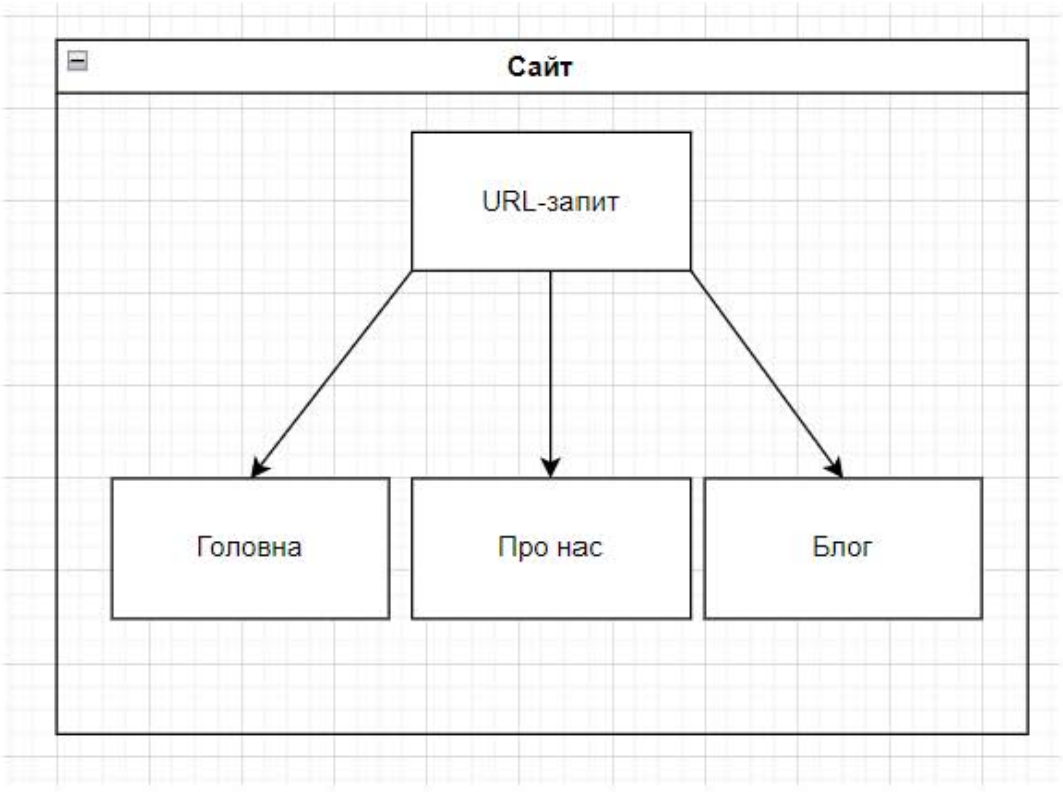

Рисунок 3.1 – Блок-схема проекту

Відповідно до представленої структури – кешування буде здійснюватися для оптимізації завантаження та тестування однієї з трьох сторінок. Якщо під час

завантаження сторінки кеш вже знаходиться в локальному сховищі, алгоритм повинен витягувати дані безпосередньо з цього сховища.

У контексті проекту, слід розробити наступну структуру алгоритму для ефективного кешування даних. (рис. 3.2):

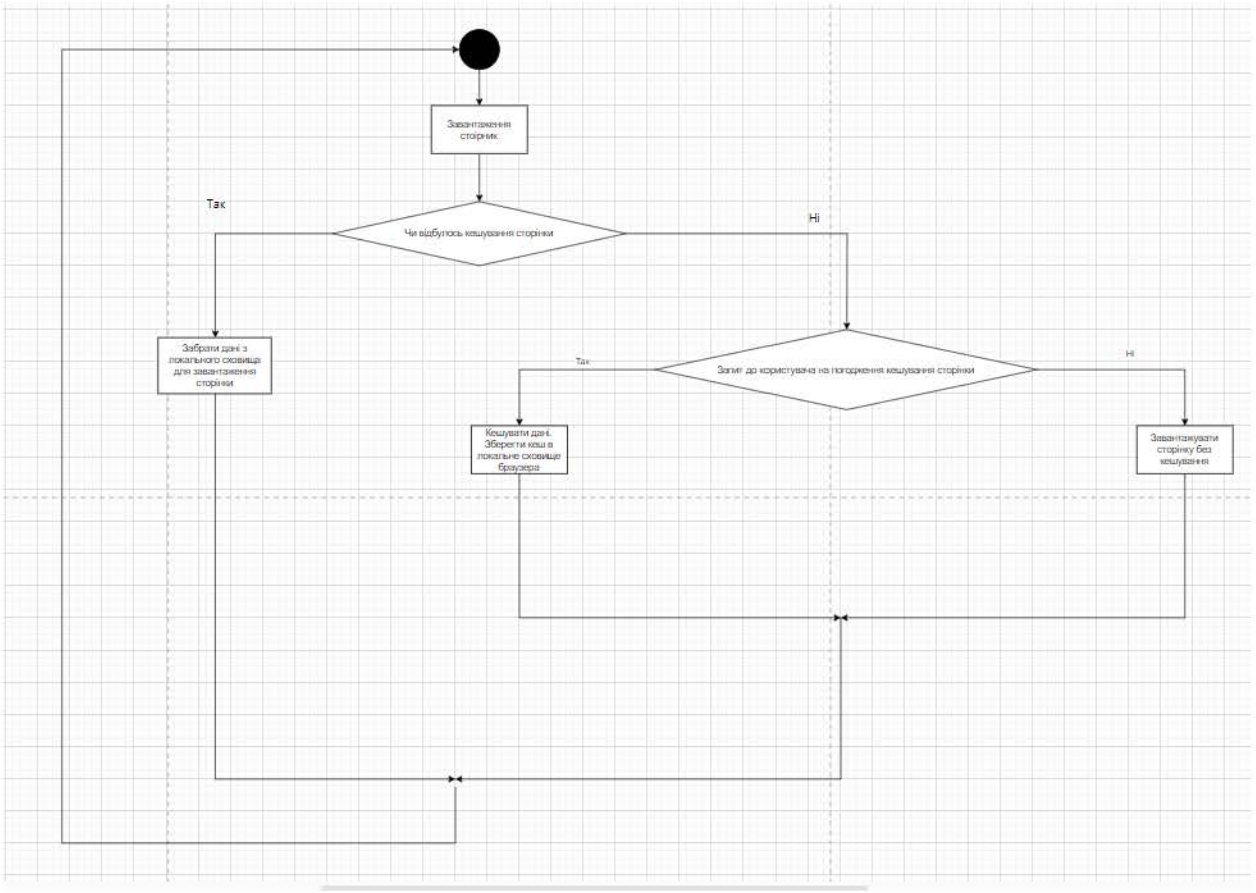

Рисунок 3.2 – Структура проекту веб-сторінки

Для створення даної структури розробнику потрібно виконати наступні етапи реалізації:

1. Визначення вимог до веб-сторінки
2. Розробка веб-сторінки
3. Створення веб-серверу
4. Розробка кешування в локальному середовищі [7].

3.2. Інструменти розробки

Для реалізації цього проекту пропонується використовувати платформу Node.js –середовище для виконання JavaScript, побудоване на основі високопродуктивного рушію V8 JavaScript від Google Chrome. Node.js дозволяє розробникам виходити за межі веб-браузерів використовуючи JavaScript для написання серверного програмного забезпечення. Однією з найвагоміших переваг Node.js є його асинхронна архітектура, яка дозволяє ефективно обробляти паралельні з'єднання без блокування, що робить його ідеальним для розробки масштабованих мережевих додатків, таких як веб-сервери, API-сервіси та додатки в реальному часі. Наприклад чати або онлайн-ігри.

Екосистема Node.js, підкріплена величезним репозиторієм модулів npm, пропонує розробникам широкий набір інструментів і бібліотек для швидкої та ефективної розробки додатків. Використання Node.js варіюється від створення простих веб-сайтів до складних розподілених систем, демонструючи його універсальність та масштабованість. Завдяки цим особливостям Node.js стає незамінним інструментом для сучасних веб-розробників, що прагнуть створювати високоефективні та масштабовані рішення[10a].

Для створення алгоритму оптимізації завантаження сторінок сайту, функціоналу та анімацій використовується мова програмування JavaScript – динамічна мова програмування, відіграє ключову роль у розробці інтерактивних веб-сайтів.

З часом JavaScript еволюціонував із простої клієнтської мови у потужну платформу для розробки, що включає серверні технології (Node.js) і численні бібліотеки та фреймворки (React.js, Angular.js і Vue.js). Ці інструменти відкривають можливості для створення масштабованих та високоефективних додатків.

JavaScript також займає важливе місце у сфері мобільної розробки завдяки фреймворкам на кшталт React Native, які дозволяють створювати нативні додатки для Android та iOS за допомогою JavaScript. Крім того інновації, такі як

Progressive Web Apps (PWA), підкреслюють центральну роль JavaScript у наданні веб-додаткам можливостей, схожих на нативні, забезпечуючи кращу продуктивність та зручність використання.

З огляду на постійний розвиток мови і екосистеми, JavaScript залишається в центрі уваги сучасних веб-технологій, адаптуючись до вимог часу та забезпечуючи розробникам потужні інструменти для створення інтерактивних і високопродуктивних веб-додатків [11].

Серед інструментів для розробки важливу роль відіграють вбудовані функції браузера, зокрема «Інструмент розробника». Цей потужний інструмент, доступний у сучасних веб-браузерах, таких як Google Chrome, Mozilla Firefox, Safari та Microsoft Edge, дозволяє розробникам і тестувальникам детально переглядати, аналізувати і змінювати HTML, CSS та JavaScript код веб-сторінки. Завдяки ним можна миттєво виявляти та виправляти помилки, оптимізувати продуктивність та вдосконалювати користувацький досвід, роблячи процес розробки більш гнучким та ефективним. (рис. 3.5) [12].

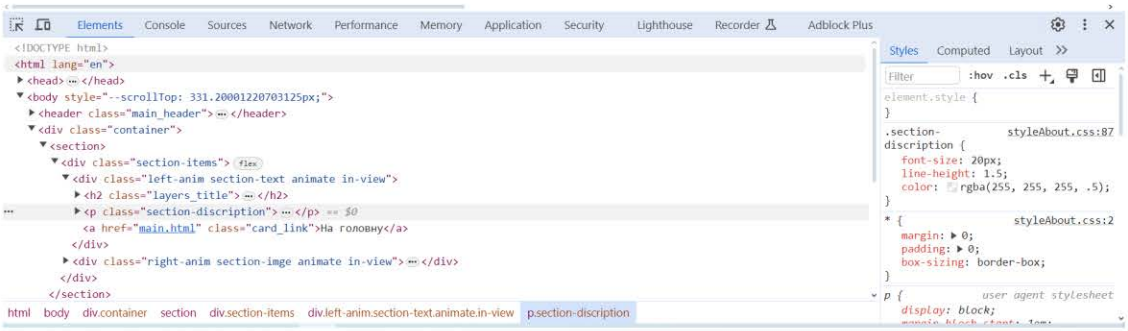

Рисунок 3.5 – «Інструмент розробника»

Завдяки «Інструментам розробника», можна миттєво редагувати код для експериментів із дизайном та функціональністю сторінки, а також тестувати скрипти через консоль JavaScript. Вони дозволяють аналізувати мережеві запити, відповіді сервера та час завантаження, що є ключовим для оптимізації продуктивності веб-сторінок. Інструменти також забезпечують тестування адаптивності сторінки на різних екранах і профілювання виконання скриптів для

виявлення проблем з продуктивністю. Надають доступ до локального та сесійного сховища, cookies (рис. 3.6) [13].

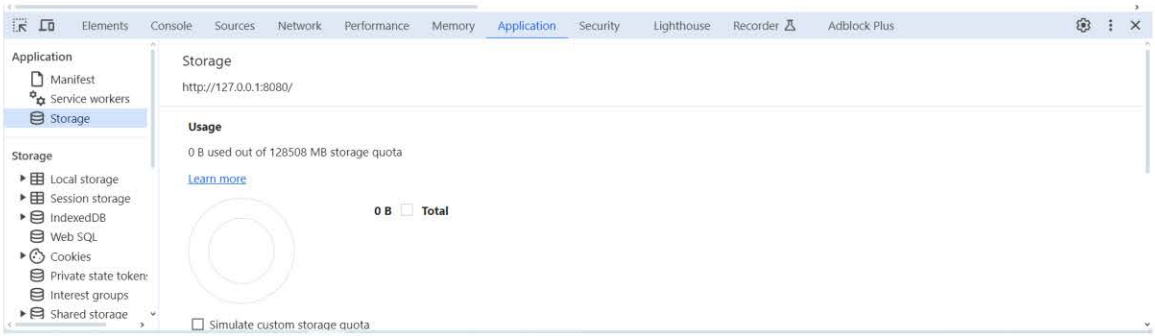

Рисунок 3.6 – Локальне сховище сторінки

У підсумку, цей інструмент допоможе аналізувати продуктивність системи та ефективність алгоритмів завантаження сторінок, які будуть розроблені в процесі.

Для створення алгоритмів завантаження та веб-сторінок використовувався Visual Studio Code. Visual Studio Code (VS Code) – це безкоштовний, потужний, легкий і відкритий редактор коду для Windows, macOS та Linux, що підтримує широкий спектр мов програмування від JavaScript і Python до C++ та Rust. Розроблений компанією Microsoft. Популярний серед розробників у всьому світі.

Оособливість VS Code – його модульність. Редактор можна розширити за допомогою плагінів з Visual Studio Marketplace, які додають підтримку нових мов, інструменти для розробки, теми інтерфейсу та утиліти для роботи з базами даних.

VS Code має вбудовану інтеграцію з системами контролю версій – Git. Редактор пропонує візуальні інструменти для комітів, роботи з гілками, злиття та вирішення конфліктів прямо з інтерфейсу, що значно спрощує управління кодом. Завдяки цим особливостям VS Code є незамінним інструментом для розробників, що прагнуть до високої продуктивності та ефективності.

3.3 Розробка веб-блогу

Спершу необхідно визначити структуру для розробки веб-сайту та тестування алгоритмів. Основних сторінок буде три: «Головна», «Про нас» та «Блог». Ці сторінки будуть заповнені багатим візуальним контентом для ретельної перевірки алгоритмів завантаження.

Далі слід розробити код для створення веб-сервера за допомогою Node.js та налаштувати веб-адресу сайту та його дочірніх сторінок. Для цього завдання було створено такий код у файлі index.js (рис. 3.8):

index.js > server > http.createServer() callback

```
const http = require('http');
const fs = require('fs');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html; charset=utf-8');
  switch(req.url)
  {
    case '/':
      fs.createReadStream('./sites/main.html').pipe(res);
      break;
    case '/about':
      fs.createReadStream('./sites/about.html').pipe(res);
      break;
    case '/blog':
      fs.createReadStream('./sites/blogUnity.html').pipe(res);

}

});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});

```

Рисунок 3.8 – Код для роботи веб-серверу сайту

Цей код створює простий веб-сервер за допомогою Node.js, який відповідає на HTTP запити і повертає HTML сторінки відповідно до запитаної URL адреси. Спочатку завантажуються модулі const http = require('http') та const fs = require('fs'). Модуль http дозволяє Node.js передавати та приймати HTTP запити та відповіді. Fs модуль, який надає функціонал для роботи з файлами.

Далі за допомогою наступних рядків: const hostname = '127.0.0.1'; і const port = 3000 визначають IP-адресу та порт, на яких сервер буде слухати запити.

Далі створюється http-сервер: const server = http.createServer((req, res) => {...}). Функція, передана у createServer, викликається кожного разу, коли сервер отримує новий запит. req (запит) і res (відповідь) - це об'єкти, що представляють вхідний запит від клієнта та вихідну відповідь від сервера, відповідно. Наступним кроком йде маршрутизація URL-запитів за допомогою конструкції switch/case:

- case '/':: Якщо URL є кореневим шляхом (/), сервер читає і відправляє файл main.html.
- case '/about':: Для шляху /about, сервер відправляє файл about.html.
- case '/blog':: Для шляху /blog, сервер відправляє файл blogGodot.html.

Далі запускається сам сервер: server.listen(port, hostname, () => { console.log(Server running at http://${hostname}:${port}/); });

Наступним етапом є розробка трьох веб-сторінок сайту. Почнемо з створення першої сторінки – «Головної». Ця сторінка буде насичена візуальним контентом та анімаціями, щоб забезпечити користувачам цікавий та естетично приємний інтерфейс (рис. 3.9).

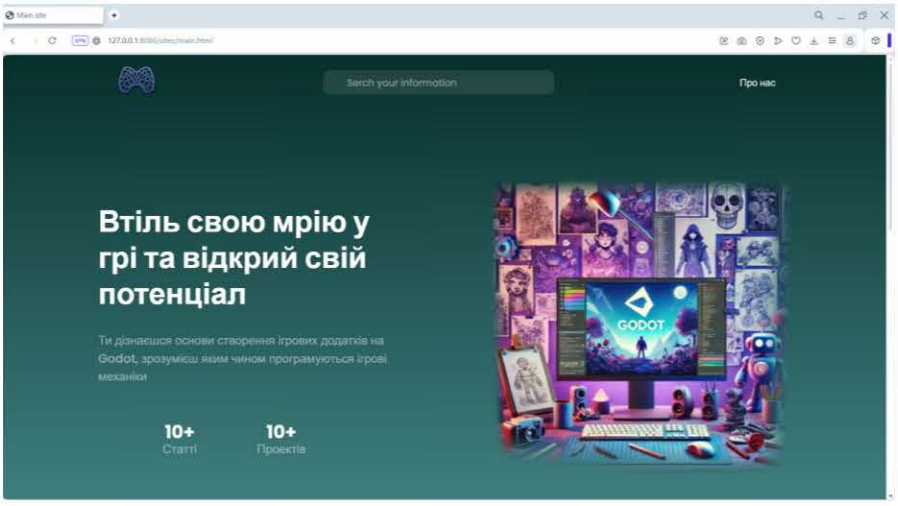

Рисунок 3.9 – Головна сторінка

Для побудови такої сторінку потрібно створити html-файл та підключити css стилі для розташування елементів сайту та встановлювати їм колір та шрифт. Для роботи стилів та скриптів, всередині тегу <head> потрібно підключити style.css (рис. 3.10).

```
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0"
    <title>Main site</title>
    <link rel="stylesheet" href="stiles/normalize.min.css">
    <link rel="stylesheet" href="stiles/style.css">
</head>
```

Рисунок 3.10 – Підключення стилів до сторінки

CSS (Cascading Style Sheets) – це мова стилів, яка використовується для визначення візуального оформлення веб-сторінок, написаних на HTML або XML. CSS дозволяє веб-розробникам і дизайнерам контролювати стиль, розташування, шрифти, кольори та інші візуальні аспекти веб-сторінки, створюючи естетично приємні і функціональні інтерфейси. Стилі будуть застосовуватися в основному через клас «.classname». Для зручності адаптації основного вмісту сайту створено тег з класом container (рис. 3.11).

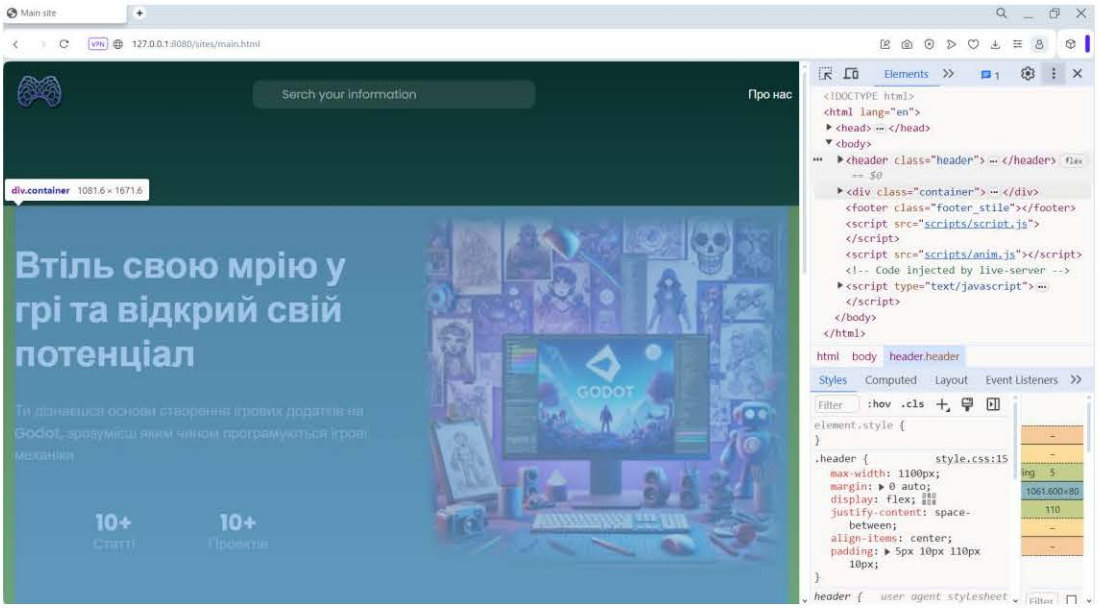

Рисунок 3.11 – Структура container

Клас container має наступні властивості (рис. 3.12):

- max-width: 1150px: властивість встановлює максимальну ширину елемента 1150 пікселів. Це допомагає забезпечити, що контент усередині контейнера не стане занадто широким на великих екранах, підтримуючи краще візуальне представлення та зручність читання.
- margin: 0 auto: відповідає за вирівнювання елемента по горизонталі. 0 вказує на відсутність вертикального відступу (margin)
- padding-left: 15px; і padding-right: 15px: задають відступи всередині елемента з лівого та правого боків відповідно.

```
.container{
    max-width: 1150px;
    margin: 0 auto;
    padding-left: 15px;
    padding-right: 15px;
}
```

Рисунок 3.12 – Властивості стилів класу «container»

Наступний блок з 3 візуальними елементами створюється за допомогою наступної розмітки html (рис 3.13):

```
<section>
<h2 class="title">Вибери статтю для навчання</h2>
<p class="desc">Для поновлення навичок є можливість розглянути різні статті</p>
<div class="card animate">
    <div class="card_item">
        <div class="card_cover">
            <img class="card_img" src="../sites/stiles/images/card_foto1.png" alt="">
        </div>
        <div class="card_info">
            <p class="card_main_inf">Введення в Unity</p>
            <p class="card_disc">@Unity</p>
        </div>
        <a href="blogUnity.html" class="card_link">Перейти</a>
    </div>
    <div class="card_item">
        <div class="card_cover">
            <img class="card_img" src="../sites/stiles/images/card_foto2.png" alt="">
        </div>
        <div class="card_info">
            <p class="card_main_inf">Базові функції в Unity</p>
            <p class="card_disc">@Unity</p>
        </div>
        <a href="blogUnity.html" class="card_link">Перейти</a>
```

Рисунок 3.13 – Блок контенту з вибором статті

Даний блок з трьох карток, які оформлюються тегом . Для кожної картки є своє фото, що додається тегом <img> та оформленим за допомогою стилів посиланням на окрему статтю. Загальний тег в свою чергу має важливі властивості для позиціонування (рис. 3.14):

- display: flex; встановлює контейнер на використання Flexbox, по одній осі.
- justify-content: space-between: визначає, як елементи всередині контейнера розподілятимуться вздовж головної осі (в цьому випадку горизонтальної, оскільки display: flex за замовчуванням встановлює горизонтальну ось як головну).
-

```
.card{
    display: flex;
    justify-content: space-between;
}
```

Рисунок 3.14 – Властивості класу card

Завдяки цим властивостям елементи розташовані горизонтально та рівномірно та відносно рівною відстанню між собою (рис. 3.15).

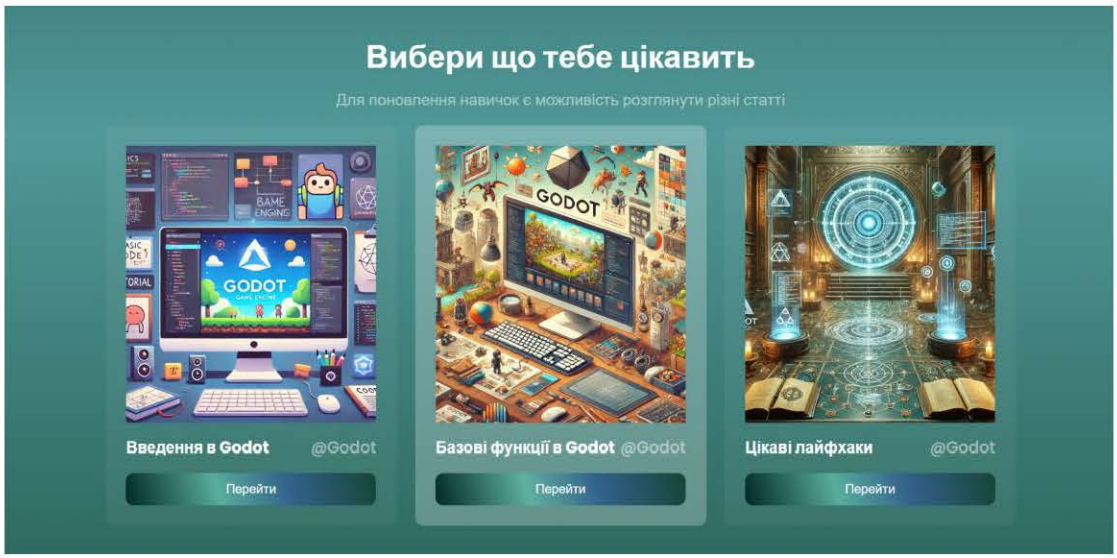

Рисунок 3.15 – Блок контенту, що відповідає за вибір статті

Всередині цих карток розміщені кнопки, які перенаправляють користувачів на сторінку блогу при натисканні. Цей елемент реалізовано за допомогою тега Перейти. Хоча технічно це просто клікабельне посилання, воно стилізоване під елегантну кнопку. Для надання цьому елементу привабливого кольору та форми, використовуються наступні CSS-стилі (рис. 3.16):

```
.card_link{
    display: block;
    text-decoration: none;
    color: #fff;
    border-radius: 10px;
    background: linear-gradient(90deg, rgba(73,83,134,1) 0%, rgba(252,70,107,1) 100%);
    padding: 11px 20px;
    text-align: center;
}
```

Рисунок 3.16 – CSS властивості класу card_link

Картки також мають анімаційні ефекти: при наведенні курсору їхній колір плавно змінюється на сірий, а при відведенні повертається до початкового. Ці

анімації реалізовані за допомогою CSS-стилів. Анімація активується при зміні положення курсору миші над об'єктом завдяки використанню властивості:hover (рис. 3.17).

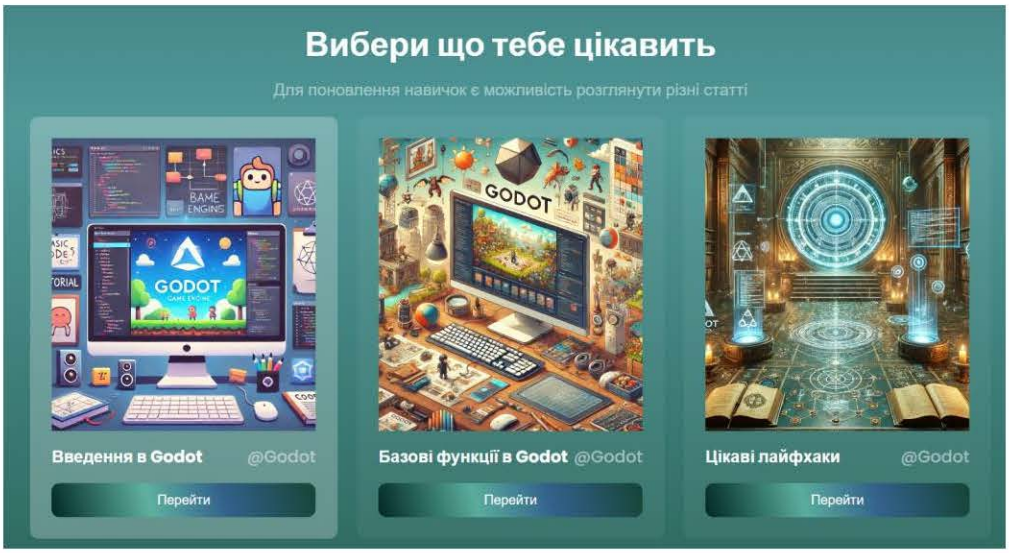

Рисунок 3.17 – Робота анімації при наведенні

Останній контентний блок включає анімовані випадаючі списки, призначені для розміщення корисних посилань для користувачів. Особливо варто відзначити, що ці елементи вже анімовані, використовуючи як CSS, так і JavaScript. Скрипт запускає плавну анімацію відкривання та закривання вмісту всередині випадаючого списку (рис. 3.18):

```
sites > scripts > JS script.js > ...
1   var coll = document.getElementsByClassName("collapsible");
2   var i;
3
4   for (i = 0; i < coll.length; i++) {
5     coll[i].addEventListener("click", function() {
6       this.classList.toggle("active");
7       var content = this.nextElementSibling;
8       if (content.style.maxHeight){
9         content.style.maxHeight = null;
10      } else {
11        content.style.maxHeight = content.scrollHeight + "px";
12      }
13    });
14  }
```

Рисунок 3.18 – Код для роботи анімації випадаючого списку

Звичайно, що для роботи скрипта потрібно додати відповідну розмітку обов'язково з тегом класу collapsible. Для оформлення використовуються звичайні тег та <button> (рис. 3.19):

```
<div class="link_list animate">
    <div class="link_item">
        <button type="button" class="collapsible">Корисні програми</button>
        <div class="content">
        <p>Lorem ipsum...</p>
        </div>
        <button type="button" class="collapsible">Статті та форуми</button>
        <div class="content">
        <p>Lorem ipsum...</p>
        </div>
    </div>
    <div class="link_item">
        <button type="button" class="collapsible">Корисні посилання та канали</button>
        <div class="content">
        <p>Lorem ipsum...</p>
        </div>
        <button type="button" class="collapsible">Документація</button>
        <div class="content">
        <p>Lorem ipsum...</p>
        </div>
    </div>
</div>
```

Рисунок 3.19 – Розмітка блоку «Корисні посилання»

Наступна сторінка – «Блог», створена для надання навчальної інформації та коротких відомостей. Ця сторінка має мінімальною кількістю анімацій, які обмежуються лише розгортаючимся меню, забезпечуючи зосередженість на змісті та легкість у користуванні. (рис. 3.20).

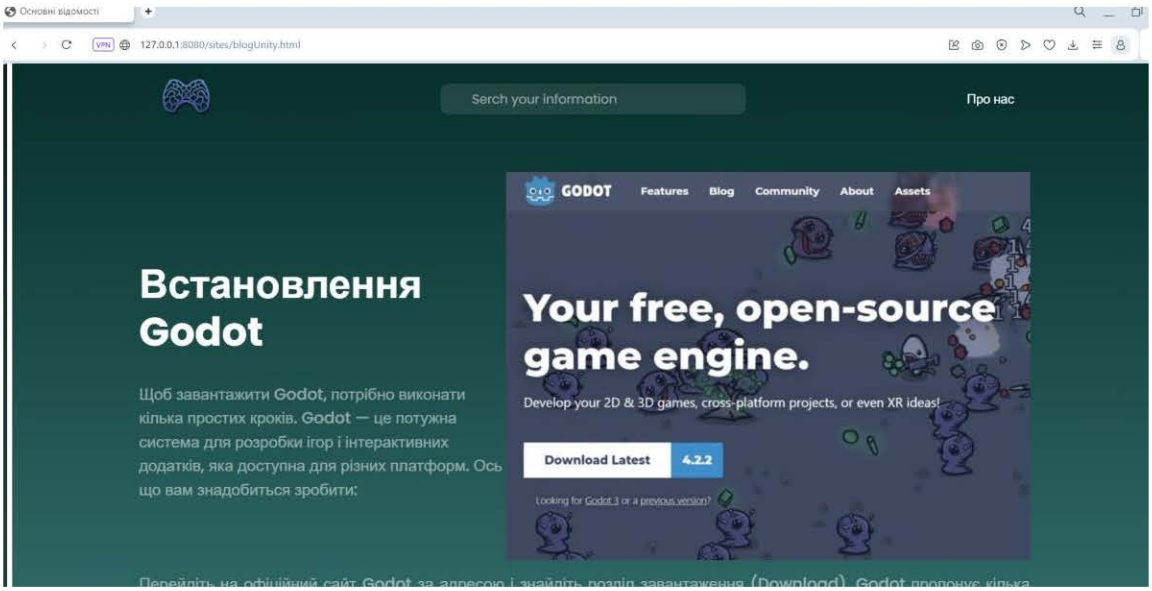

Рисунок 3.20 – Зовнішній вигляд сторінки «Блог»

HTML-розмітка цієї сторінки зберігає загальну структуру, подібну до головної сторінки, проте відрізняється за контентом і відсутністю анімацій. Для спрощення розробки було використано частину стилів з головної сторінки, що дозволило зберегти узгодженість дизайну та скоротити час на налаштування (рис. 3.21):

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Основні відомості</title>
    <link rel="stylesheet" href="stiles/normalize.min.css">
    <link rel="stylesheet" href="stiles/style.css">
    <link rel="stylesheet" href="stiles/blogStile.css">
    <script src="scripts/cacheScript.js"></script>
</head>
```

Рисунок 3.21 – Підключені стилі CSS до сторінки

Як і на попередній сторінці, основний контент обгорнутий в клас container для зручного та впорядкованого розміщення блоків. Однак, для цієї сторінки додатково передбачено анімоване меню навігації. Щоб створити цей компонент, спершу потрібно розробити відповідну HTML-розмітку. (рис. 3.22):

```
<div class="menu">
    <div class="blogtitle">
        <h1>Навігація</h1>
    </div>
    <div class="list_menu">
        <ul>
            <li><a href="#Installation">Встановлення Unity</a></li>
            <li><a href="#Registration">Реєстрація</a></li>
            <li><a href="#Download Version">Завантаження версії</a></li>
            <li><a href="#views">Корисні посилання</a></li>
        </ul>
    </div>
</div>
```

Рисунок 3.22 – HTML-розмітка меню навігації

Наступним кроком є налаштування меню за допомогою класу menu, щоб завжди залишалося фіксованим і доступним у будь-якій частині сторінки. Для досягнення цього ефекту необхідно додати наступні CSS-властивості (рис. 3.23):

```
.menu{
    padding: 10vh 50px;
    position: fixed;
    background: rgba(20, 20, 20, 0.5);
    height: 100vh;
    backdrop-filter: blur(16px);
    border-right: 7px solid #fff;
    left: -345px;
    transition: all 0.2s ease;
}
```

Рисунок 3.23 – Стилі для класу menu

Однією з ключових властивостей є position: fixed. Цей параметр фіксує положення об'єкта на сторінці, забезпечуючи, щоб елемент завжди залишався на одному місці, незалежно від того, яку частину сторінки переглядає користувач. Таким чином, меню залишається постійно доступним. За анімацію відкривання меню теж відповідає властивість .menu:hover{left: 0; transition: all 0.2s ease;} (рис. 3.24).

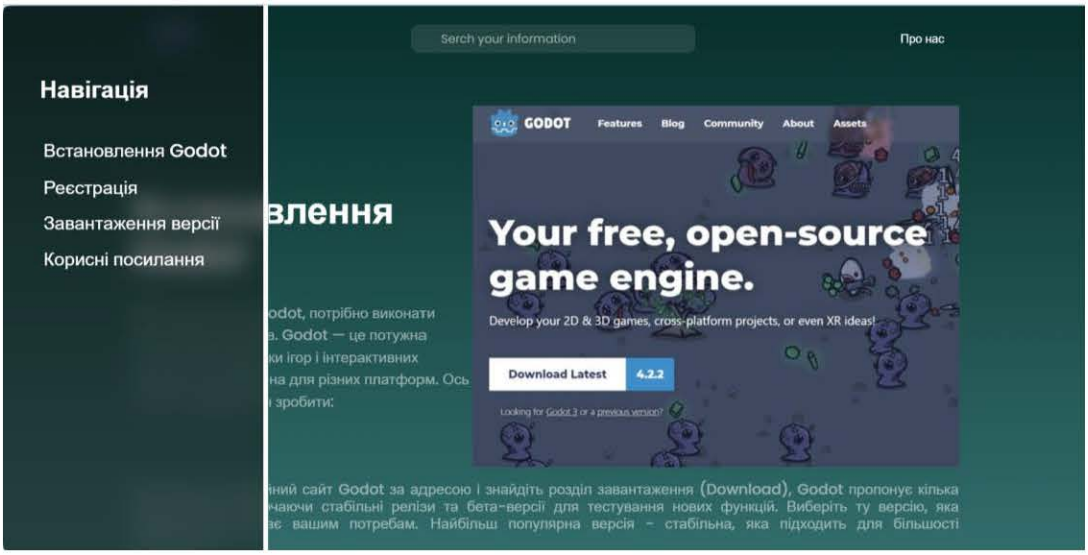

Рисунок 3.24 – Відкриття меню навігації

Загалом, меню виконує роль змісту сторінки сайту, полегшуючи навігацію та дозволяючи користувачам миттєво переходити до потрібних розділів.

Після реалізації сторінки «Блог», наступним кроком є розробка сторінки «Про нас». Ця сторінка, хоча й містить менше контенту, відрізняється унікальними принципами побудови анімацій та використанням різних компонентів, що надає їй особливого стилю (рис. 3.25).

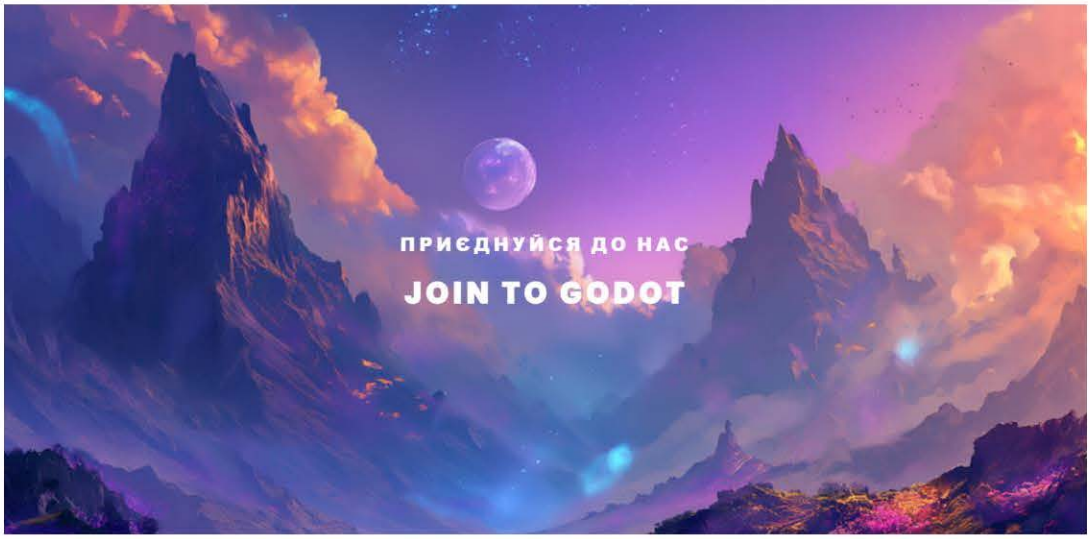

Рисунок 3.25 – Початок сторінки «Про нас»

З самого початку сторінки активується анімаційний ефект «Паралакс». Цей візуальний прийом створює відчуття глибини та об'єму під час прокручування сторінки. Ефект досягається завдяки трьом шарам (передньому, середньому і задньому), які рухаються з різною швидкістю. Шар, розташований ближче до користувача, рухається швидше, ніж ті що розташовані далі, створюючи враження просторової глибини. Анімація цього ефекту реалізована за допомогою коду в файлі anim.js. (рис. 3.26):

```
sites > scripts > JS anim.js > ...
1   document.addEventListener("DOMContentLoaded", () => {
2
3       // Use Intersection Observer to determine if objects are within the viewport
4       const observer = new IntersectionObserver(entries => {
5         entries.forEach(entry => {
6           if (entry.isIntersecting) {
7             entry.target.classList.add('in-view');
8             return;
9           }
10          entry.target.classList.remove('in-view');
11        });
12      });
13
14      // Get all the elements with the .animate class applied
15      const allAnimatedElements = document.querySelectorAll('.animate');
16
17      // Add the observer to each of those elements
18      allAnimatedElements.forEach((element) => observer.observe(element));
19
20  });
```

Рисунок 3.26 – Скрипт anim.js

Під час скролу в низ користувач може побачити як різні шари зображення рухаються по різному, і створюється ефект «Паралакс» (рис. 3.27):

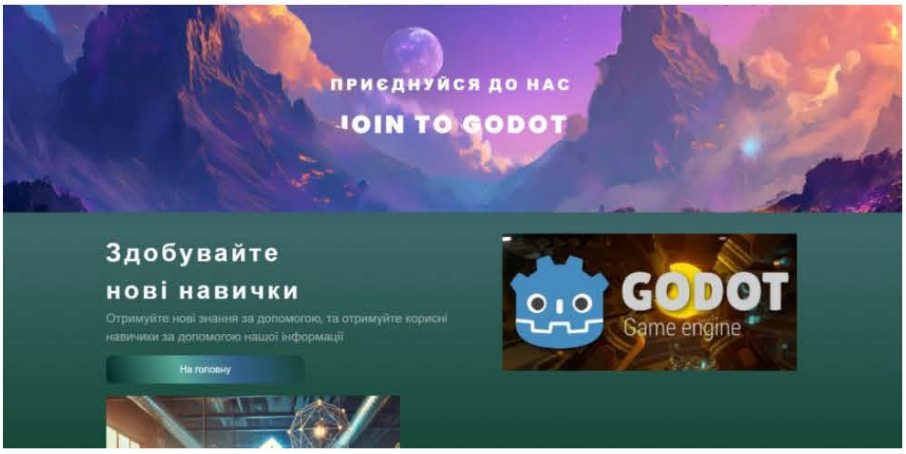

Рисунок 3.27 – Робота анімації «Паралакс»

Наступний вміст крім останнього компоненту має ідентичну структуру як і в попередніх сторінок це використання класу container (рис. 3.28).

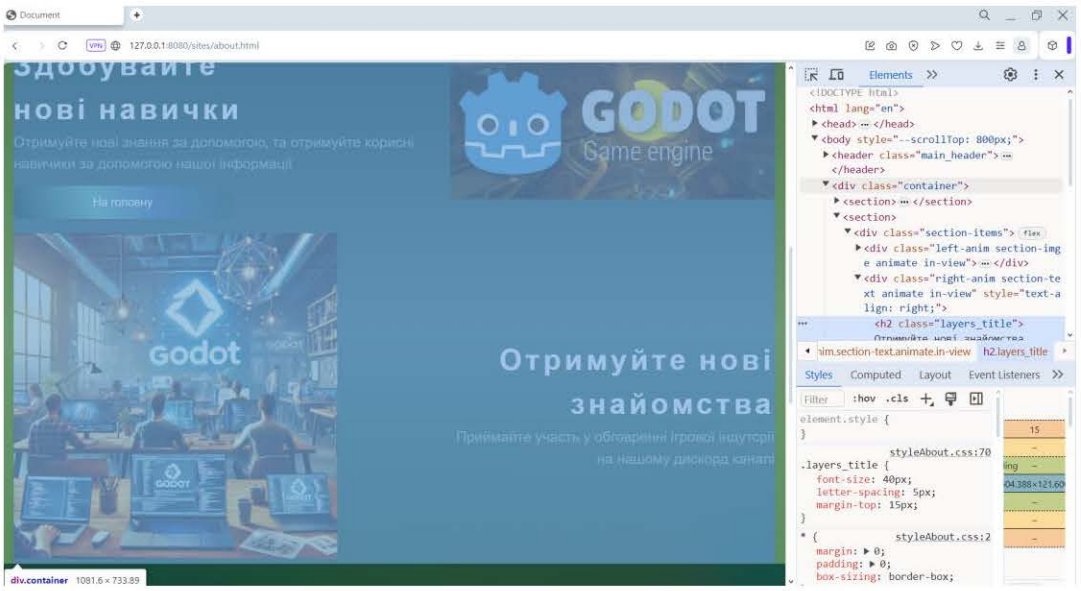

Рисунок 3.28 – Елементи, що належать класу container

Останнім компонентом є галерея відомих ігор створених на Godot. Цей інтерактивний елемент складається з декількох анімованих фото-карток, які демонструють популярні інді-ігри. Для розробки цього компонента використовувалися сторонні бібліотеки та спеціальні стилі, що додали йому унікального вигляду і функціональності в результаті отримано наступну розмітку (рис. 3.29):

```
<div class="swiper slider slider main">
    <div class="swiper-wrapper slier  wrapper">
        <div class="swiper-slide slider  item">
            <div class="slider__img" data-swiper-parallax="20%" style="background-image: url(stiles/imagesGalary/journey.png);"></div>
        </div>
        <div class="swiper-slide slider__item">
            <div class="slider__img" data-swiper-parallax="30%" style="background-image: url(stiles/imagesGalary/2.png);"></div>
        </div>
        <div class="swiper-slide slider__item">
            <div class="slider__img" data-swiper-parallax="20%" style="background-image: url(stiles/imagesGalary/3.png);"></div>
        </div>
        <div class="swiper-slide slider__item">
            <div class="slider__img" data-swiper-parallax="30%" style="background-image: url(stiles/imagesGalary/4.png);"></div>
        </div>
        <div class="swiper-slide slider__item">
            <div class="slider__img" data-swiper-parallax="20%" style="background-image: url(stiles/imagesGalary/5.png);"></div>
        </div>
        <div class="swiper-slide slider__item">
            <div class="slider__img" data-swiper-parallax="30%" style="background-image: url(stiles/imagesGalary/6.png);"></div>
        </div>
        <div class="swiper-slide slider__item">
            <div class="slider  img" data-swiper-parallax="20%" style="background-image: url(stiles/imagesGalary/7.png);"></div>
        </div>
    </div>
```

Рисунок 3.29 – Розмітка компоненту «Галерея»

Також для сторінок були розроблені спеціальні анімації появи, реалізовані за допомогою CSS-стилів. Ці анімації додають динамічності та візуальної привабливості інтерфейсу використовуючи наступні властивості (рис. 3.30):

```
@keyframes animText_see{
    from{
        color: transparent;
        transform: translateX(-500px);
    }
    to{
        transform: translateX(0px);
    }
}
@keyframes rightImage{
    from{
        opacity: 0;
        transform: translateX(+600px);
    }
    to{
        transform: translateX(0px);
    }
}
@keyframes cardAnim{
    from{
        transform: translateX(-1000px);
    }
    to{
        transform: translateX(0px);
    }
}
```

Рисунок 3.30 – CSS анімації для сайту

За допомогою @keyframes можна створювати плавні анімації, змінюючи певні параметри від початкового до кінцевого стану.

Після створення трьох сторінок сайту, наступним кроком є розробка алгоритмів для їх оптимального завантаження. Перший алгоритм, який ми розглянемо, це цілеспрямоване кешування візуальних об'єктів. Цей алгоритм працює за принципом: під час першого завантаження сторінки він створює кеш і зберігає його в пам'яті браузера. При повторному відвідуванні сторінки, дані завантажуються вже з кешу, що значно пришвидшує процес завантаження. Для реалізації цього алгоритму використовується скрипт cacheScript.js (рис. 3.31):

```
let db;
const request = window.indexedDB.open("imagesCacheDB", 1);

request.onerror = function(event) {
    console.error("Database error: ", event.target.error);
};

request.onupgradeneeded = function(event) {
    const db = event.target.result;
    db.createObjectStore("images", { keyPath: "url" });
};

request.onsuccess = function(event) {
    db = event.target.result;
    checkAndLoadImages();
};
```

Рисунок 3.31 – Початок коду алгоритму кешування

Цей код демонструє процес створення та використання IndexedDB в браузері для кешування зображень. IndexedDB – це потужне низькорівневе API для клієнтського зберігання великих обсягів структурованих даних з можливістю виконання запитів до цих даних.

Спершу створюється змінна db для збереження посилання на базу даних. Далі викликається метод window.indexedDB.open("imagesCacheDB"), де "imagesCacheDB" є назвою бази даних. Цей метод встановлює з'єднання з базою даних або створює нову базу, якщо вона ще не існує.

Функція request.onerror = function(event) {...} визначена для обробки помилок, які можуть виникнути під час відкриття бази даних. У разі невдачі IndexedDB ця функція викликається і виводить повідомлення про помилку в консоль, що допомагає відстежувати і виправляти проблеми.

Метод db.createObjectStore("images", { keyPath: "url" }) створює об'єктне сховище (таблицю) з назвою "images", де ключем є url. Це дозволяє унікально ідентифікувати кожен запис в сховищі, забезпечуючи ефективне кешування і швидкий доступ до зображень.

Наступним кроком є виведення вікна для увімкнення користувачем алгоритму (рис. 3.32):

```
function checkAndLoadImages() {
    const transaction = db.transaction("images", "readonly");
    const store = transaction.objectStore("images");
    const countRequest = store.count();

    countRequest.onsuccess = function() {
        if (countRequest.result === 0) {
            // Кеш пустий, питаємо про кешування
            var cacheEnabled = confirm("Включити кешування? Натисніть OK для включення або Скасувати для вимкнення.");
            if (cacheEnabled) {
                cacheImages();
            }
        } else {
            // Кеш не пустий, завантажуємо зображення з кешу
            loadImagesFromCache();
        }
    };
}
```

Рисунок 3.32 – Виведення випливаючого вікна

Функція checkAndLoadImages перевіряє наявність зображень у кеші бази даних IndexedDB і вирішує, чи потрібно завантажувати зображення з кешу або додавати нові. Вона починається зі створення транзакції для доступу до об'єктного сховища images у режимі "readonly", що дозволяє лише зчитувати дані без їх зміни.

У межах цієї транзакції створюється змінна store, яка забезпечує доступ до сховища зображень. Далі функція виконує запит на підрахунок кількості записів у цьому сховищі за допомогою методу count(). Результат цього запиту обробляється через подію onsuccess, яка визначає, чи є зображення в кеші. Якщо результат підрахунку дорівнює нулю, це означає, що кеш порожній. У такому випадку користувачеві пропонується активувати кешування через діалогове вікно confirm, яке дозволяє вибрати, чи включати кешування зображень. Якщо користувач погоджується, викликається функція cacheImages(), яка відповідає за кешування нових зображень.

Якщо в кеші вже є зображення (результат підрахунку більший за нуль), виконується функція loadImagesFromCache(), яка завантажує зображення безпосередньо з кешу. Це значно покращує швидкість завантаження сторінки, оскільки зображення не потрібно завантажувати з інтернету, забезпечуючи швидший і плавніший користувацький досвід.

Наступна функція – це функція кешування (рис. 3.33).

```
function cacheImages() {
    document.querySelectorAll('img').forEach(img => {
        const src = img.getAttribute('src');
        fetch(src).then(response => {
            if (response.ok) {
                return response.blob();
            }
            throw new Error('Network response was not ok.');
        })
        .then(blob => {
            const tx = db.transaction("images", "readwrite");
            const store = tx.objectStore("images");
            const item = {
                url: src,
                blob: blob
            };
            store.add(item);
        })
        .catch(error => console.error('Could not cache image:', src, error));
    });
}
```

Рисунок 3.33 – Функція кешування даних

Функція cacheImages виконує кешування зображень, знайдених на веб-сторінці, у локальну базу даних IndexedDB. Процес кешування починається з вибору всіх елементів <img> на сторінці за допомогою document.querySelectorAll('img'). Для кожного зображення з атрибутом src, який містить URL-адресу зображення, виконується HTTP-запит методом fetch щоб отримати зображення з мережі. Якщо запит завершується успішно, відповідь сервера перетворюється у BLOB-об'єкт, який представляє двійкові дані зображення. У випадку невдачі запиту генерується виняток з відповідним повідомленням про помилку.

Після отримання BLOB-об'єкта відкривається транзакція з базою даних db в режимі "readwrite", що дозволяє змінювати дані у базі. Створюється новий запис у сховищі images цієї бази даних, куди додається об'єкт з полями url та blob. Такий об'єкт містить URL-адресу зображення та саме зображення у форматі BLOB. Запис додається до бази даних методом store.add(item).

В разі виникнення помилки під час кешування зображення (наприклад, якщо зображення не може бути завантажене або збережене у базі даних), виконується блок catch, який реєструє помилку в консолі. Це допомагає в

ідентифікації проблем, пов'язаних із мережевими запитами або збереженням даних.

Ця функція важлива для покращення продуктивності веб-сторінок, оскільки зменшує залежність від мережевих запитів при повторному доступі до сторінок, швидше відображаючи зображення з локального кешу.

Наступна функція – це завантаження даних з кешу та їх форматування в готові зображення (рис. 3.34).

```
    function loadImagesFromCache() {
        const transaction = db.transaction("images", "readonly");
        const store = transaction.objectStore("images");
        document.querySelectorAll('img').forEach(img => {
            const src = img.getAttribute('src');
            const request = store.get(src);

            request.onsuccess = function() {
                if (request.result) {
                    const url = URL.createObjectURL(request.result.blob);
                    img.src = url;
                }
            };
        });
    }
});
```

Рисунок 3.34 – Функція loadImagesFromCache()

Функція loadImagesFromCache використовується для завантаження зображень з локального кешу, збереженого у базі даних IndexedDB. Функція ініціює транзакцію з базою даних db, використовуючи режим "readonly". Далі за допомогою методу document.querySelectorAll('img') знаходяться всі елементи <img> на сторінці. Для кожного зображення виконується функція, яка спочатку зчитує його атрибут src для визначення URL-адреси зображення. Для кожного URL зображення створюється запит до бази даних за допомогою методу get(src), де src є ключем для пошуку в об'єктному сховищі images. Коли запит завершується успішно, обробник request.onsuccess перевіряється чи існує результат запиту. Якщо зображення знайдено (request.result), для BLOB об'єкта

зображення створюється URL за допомогою URL.createObjectURL(request.result.blob). Цей URL встановлюється як новий атрибут src для тега <img>, ефективно замінюючи посилання на зовнішнє зображення на локальне збережене зображення.

Для дослідження ефективності цього алгоритму необхідно розробити відомий метод оптимізації Lazy Loading. Цей алгоритм широко використовується для підвищення продуктивності веб-сторінок та мобільних додатків. Основна ідея полягає в тому, щоб завантажувати контент лише тоді, коли користувач його потребує, а не завантажувати весь вміст одразу при завантаженні сторінки. Для реалізації цієї техніки слід написати відповідний код:

```
ites > scripts > JS lazyLoading.js > ...
1   document.addEventListener('DOMContentLoaded', () => {
2       const images = document.querySelectorAll('img');
3
4       if ('IntersectionObserver' in window) {
5           // Встановлення data-src та очищення src для усіх зображень
6           images.forEach(img => {
7               img.dataset.src = img.src;
8               img.src = ''; // Очистити src або встановити запасне зображення
9           });
10
11          let lazyImageObserver = new IntersectionObserver((entries, observer) => {
12              entries.forEach((entry) => {
13                  if (entry.isIntersecting) {
14                      let lazyImage = entry.target;
15                      lazyImage.src = lazyImage.dataset.src; // Відновити src з data-src
16                      lazyImageObserver.unobserve(lazyImage); // Припинити спостереження
17                  }
18              });
19          });
```

Рисунок 3.35 – Початок скрипта «Lazy Loading»

Звичайно, можна зазначити що для того щоб додати завантаження за цим алгоритмом потрібно просто до тегу додати параметр loading="lazy", але складність в тому, що деякі браузери можуть цілеспрямовано встановлювати пріоритет на завантаження тегів з цим параметром. В даному скрипті розглядається вид алгоритму «примусовий lazy loading». Примусовим алгоритм є в результаті дій певний функцій які потрібно розглянути більш детально.

Спочатку код встановлює обробник події DOMContentLoaded, що гарантує запуск скрипта після повного завантаження початкової HTML-сторінки. Це важливо для забезпечення доступності всіх зображень для маніпуляцій у DOM. Код використовує document.querySelectorAll('img') для отримання всіх зображень на сторінці і зберігає їх у змінну images, аналогічно до попереднього алгоритму. Далі відбувається інтеграція з компонентом IntersectionObserver, який дозволяє ефективно визначати, коли об'єкт входить в область видимості або виходить з неї не використовуючи значні ресурси необхідні для постійного відстеження позиції об'єкта за допомогою подій прокрутки. Всі зображення налаштовуються так, що їх початковий атрибут src зберігається у data-src, а сам src очищується. Це запобігає небажаному завантаженню зображень під час ініціалізації сторінки.

```
        images.forEach((lazyImage) => {
            lazyImageObserver.observe(lazyImage); // Почати спостереження
        });
    } else {
        // Для браузерів, що не підтримують Intersection Observer
        images.forEach(img => {
            img.src = img.dataset.src;
        });
    }
});
```

Рисунок 3.36 – Кінець код алгоритму «Lazy Loading»

Якщо IntersectionObserver не підтримується браузером, встановлюється src для кожного зображення безпосередньо з data-src. Це гарантує, що зображення завантажуються звичайним способом, хоча і без оптимізації lazy loading.

Наступним кроком є його перевірка та тестування та порівняння роботи алгоритмів.

3.4 Тестування

Для перевірки функціональності додатку користувачу спершу необхідно створити локальний веб-сервер та завантажити додаток. Якщо ж додаток розміщений на веб-хостингу, слід перейти за відповідною URL-адресою.

У даному випадку, додаток завантажується з локального серверу, тому необхідно спочатку його запустити, а потім перейти за вказаною URL-адресою: http://127.0.0.1:8080/sites/main.html.

Після завантаження сторінки користувач потрапляє на головну сторінку сайту (рис. 3.37).

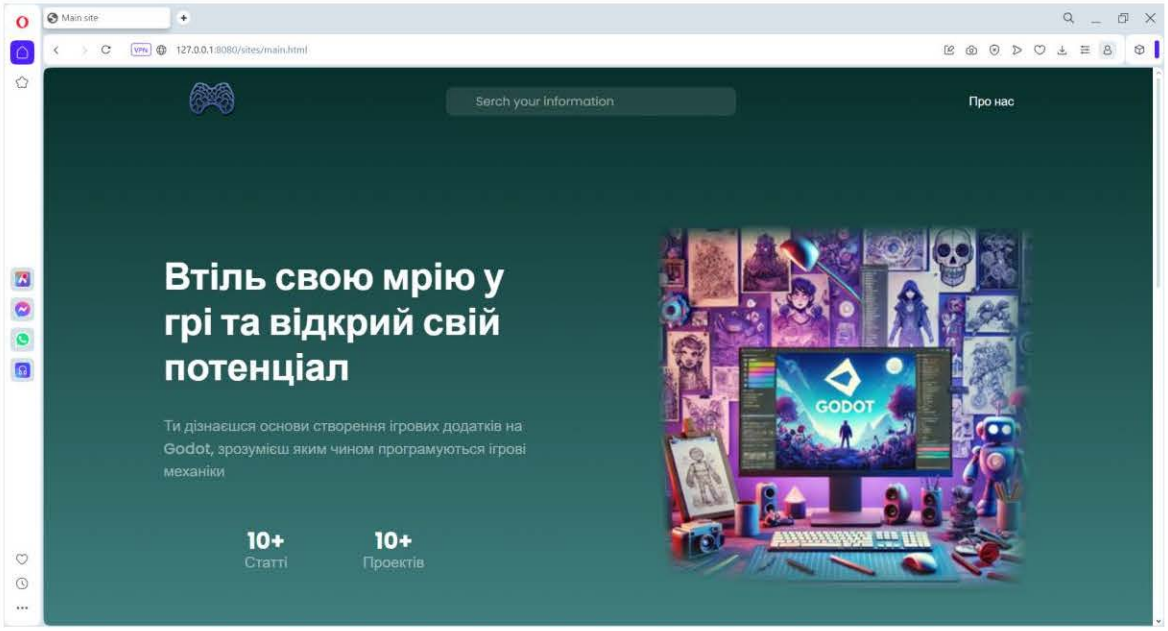

Рисунок 3.37 – Початок головної сторінки

Ця сторінка вражає плавною анімацією появи елементів під час поступового скролінгу вниз. Вона структурована у кілька контентних блоків. Перший блок унікально розділений на дві частини, де текст і фото гармонійно доповнюють один одного, створюючи привабливий візуальний і інформаційний досвід (рис. 3.37). Наступним анімованим блоком є вибір статті, три з яких пропонуються авторами сайту (рис. 3.38).

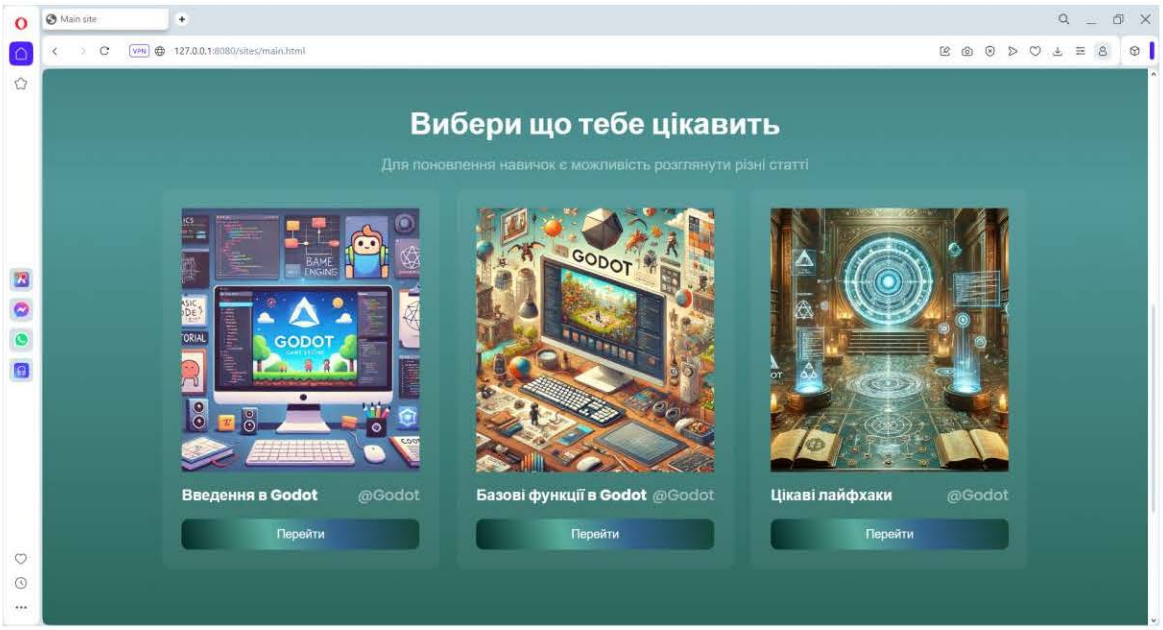

Рисунок 3.38 – Друга частина сторінки

Даний блок з’являється за допомогою анімації коли користувач тільки перейде в поле зору цього блоку. Даний блок складається з 3 анімованих карток при наведені курсору колір змінюється на сірий.

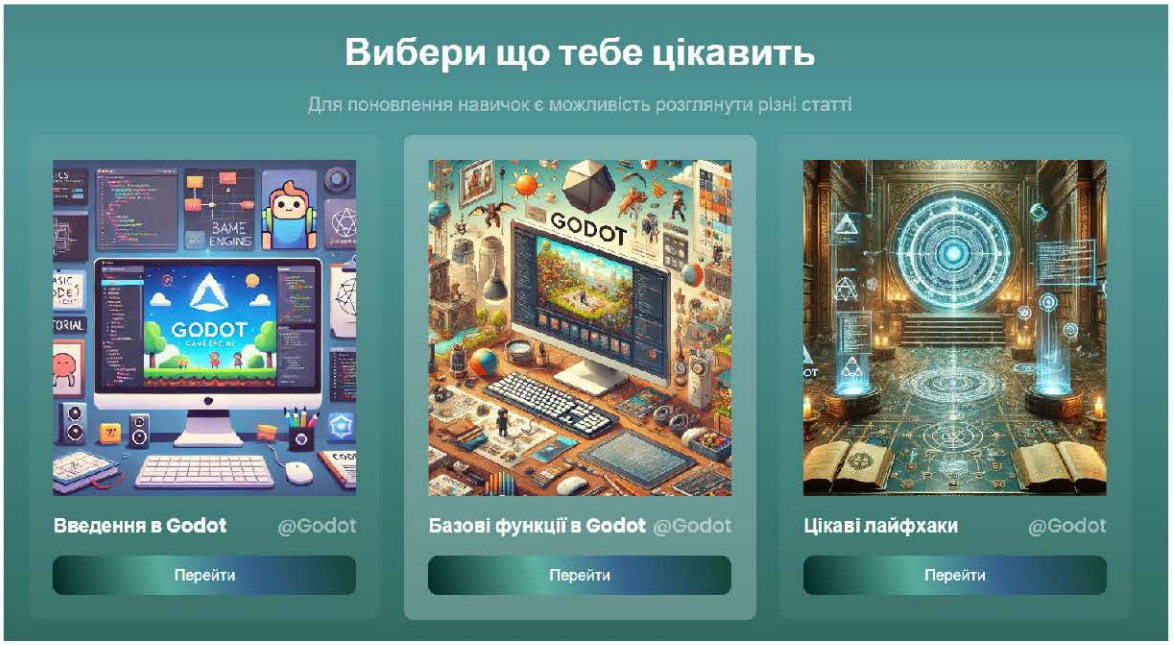

Рисунок 3.39 – Робота анімації

У картках є кнопки натискаючи на які можна перейти до статті. Наступний блок містить випадаючі списки, які в майбутньому можуть бути заповнені корисним контентом (рис. 3.40).

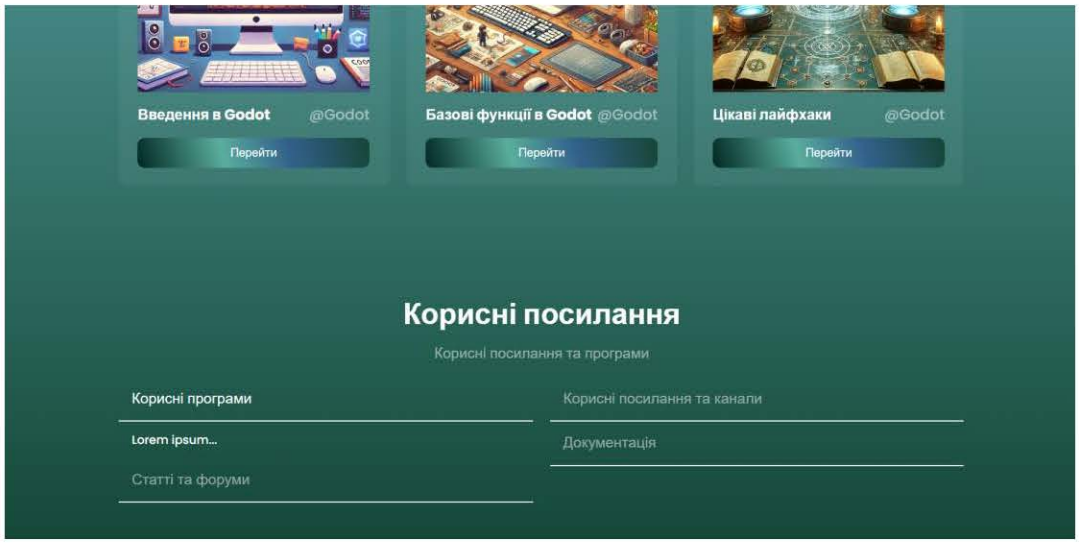

Рисунок 3.40 – Кінець головної сторінки

Наступна сторінка – «Про нас». Для переходу користувачу потрібно повернутися на початок сторінки, де його зустріне кнопка «Про нас», що дозволяє швидко і легко дізнатися більше про команду розробників (рис. 3.41).

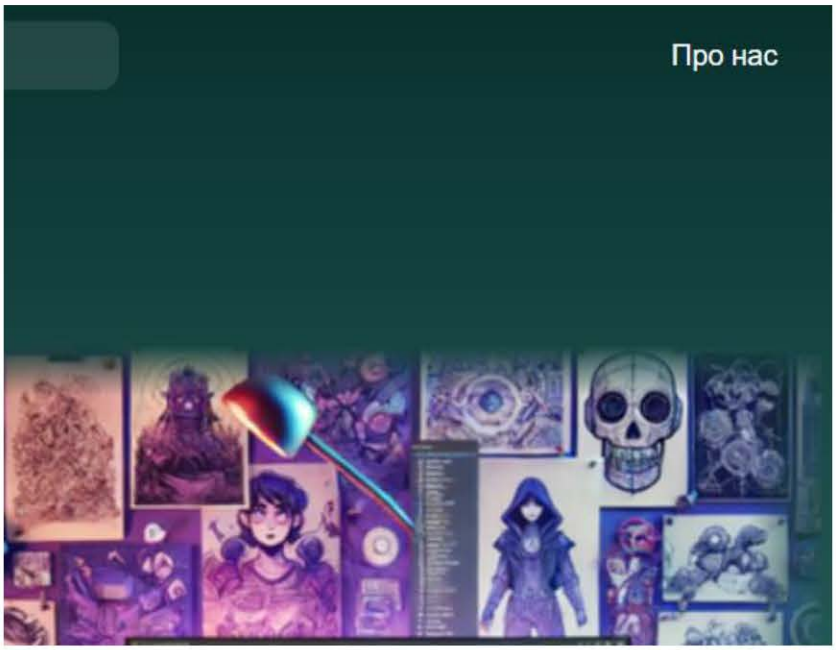

Рисунок 3.41 – Посилання на сторінку «Про нас»

Або користувач може ввести відповідний URL-адрес: http://127.0.0.1:8080/sites/about.html. Після переходу користувач спостерігає фото пейзажу та заголовок (рис. 3.42).

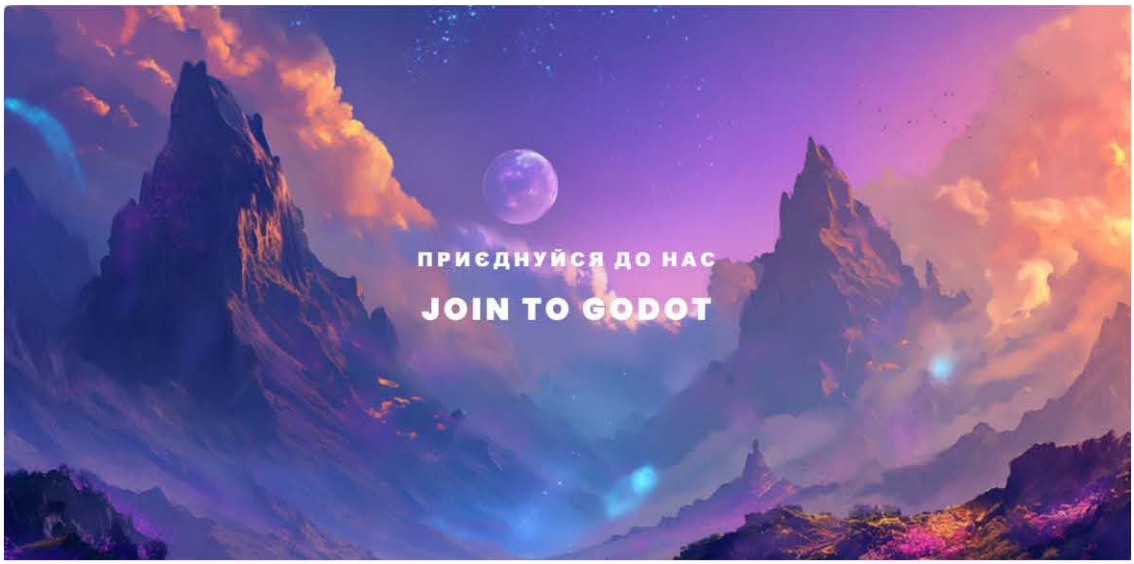

Рисунок 3.42 – Початок сторінки «Про нас»

Далі під час прокручування користувач може спостерігати так званий ефект «Паралакс», що надає ілюзії об'єму зображення.

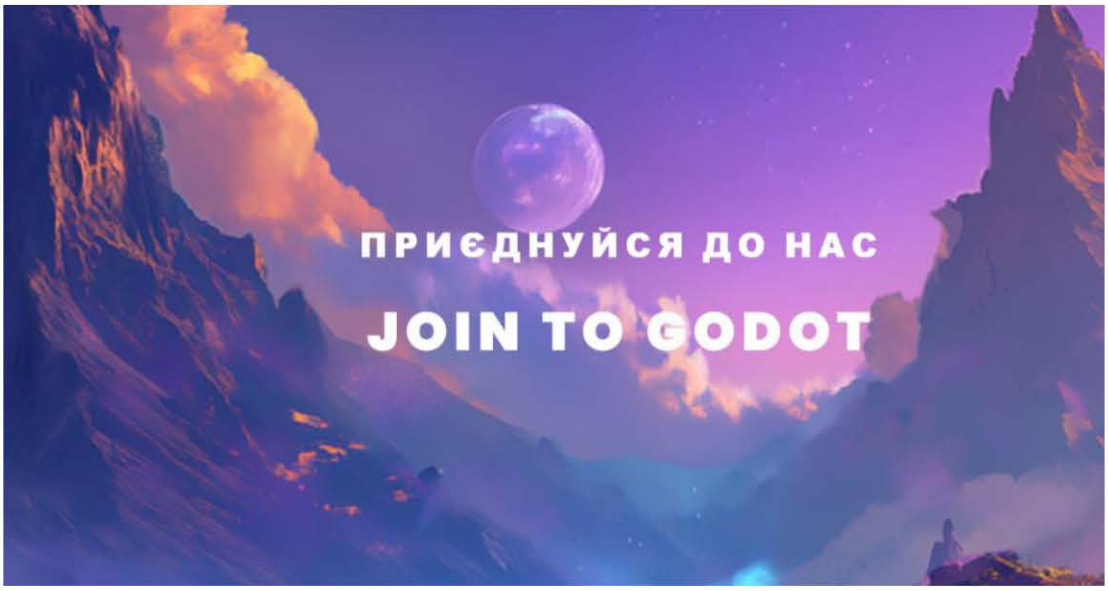

Рисунок 3.43 – Ефект «Паралакс»

Цей ефект створюється завдяки різним швидкостям переміщення трьох шарів зображень, що надає глибини та об'ємності. Після цього користувач побачить плавну анімацію появи контентних блоків, схожу на ту що була на попередній сторінці (рис. 3.44).

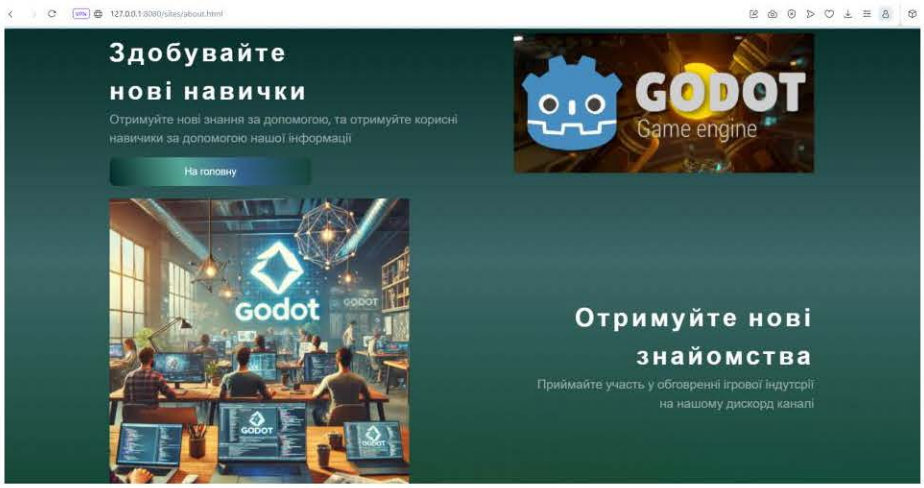

Рисунок 3.44 – Друга частина сторінки «Про нас»

Кінець цієї сторінки включає складну анімацію скролінгу та перегляду фотографій. Тут представлена фотогалерея з численними зображеннями популярних ігор, створених на Godot, які можна переглядати, просто прокручуючи сторінку. Інтерактивна галерея дозволяє насолодитися візуальним досвідом та оцінити різноманітність ігор (рис. 3.45).

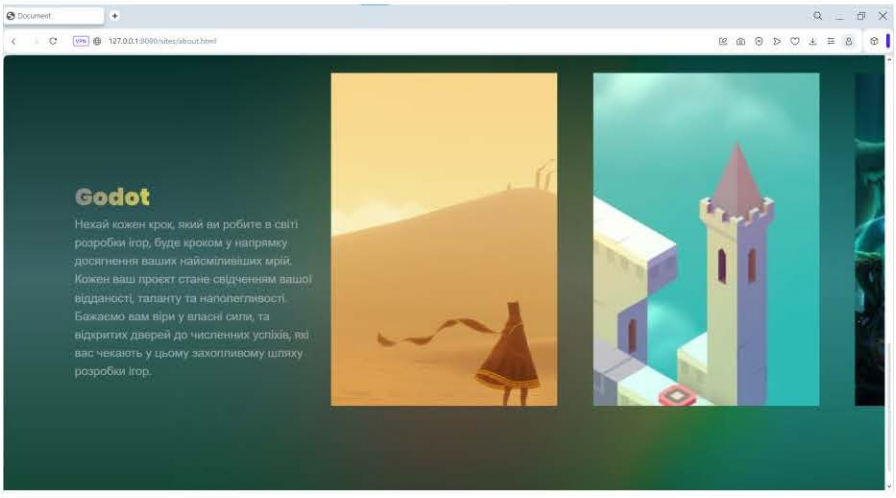

Рисунок 3.45 – Фото-галерея відео-ігор

В процесі перегляду текст плавно зникає та з’являються нові фото (рис. 3.46).

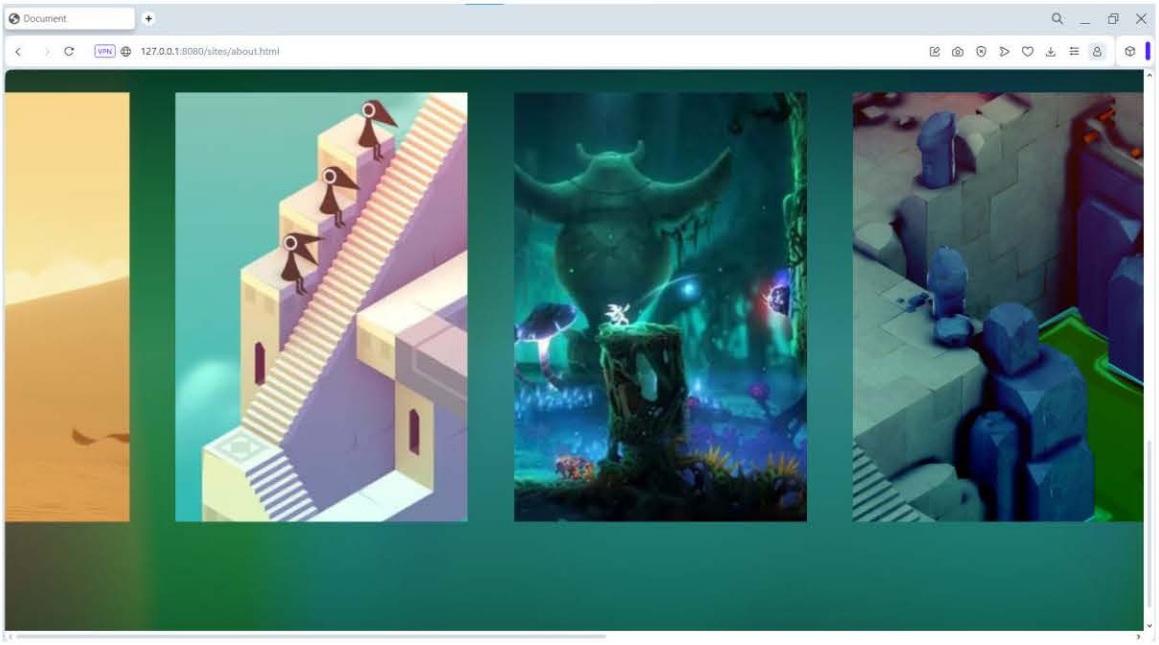

Рисунок 3.46 – Анімація перегляду фото-галереї

Щоб перейти на головну сторінку, користувач може повернутися в гору до фото пейзажу (рис. 3.47):

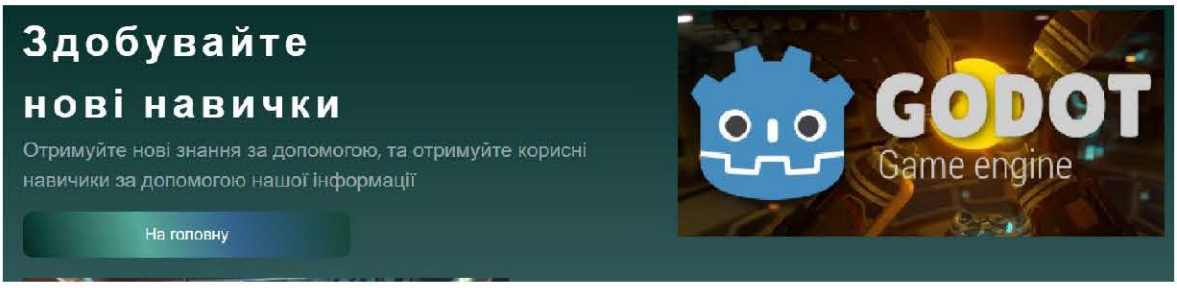

Рисунок 3.47 – Кнопка для повернення на головну сторінку

Наступна сторінка – «Блог» зі статтями, що детально розкривають принципи роботи програми або пояснюють окремі функції. Як вже згадувалося, щоб перейти до статті, виберіть одну з трьох запропонованих і натисніть кнопку «Перейти».

Рисунок 3.48 – Перехід на сторінку «Блог»

Користувач перейде на сторінку, що відповідає за надання документаційної інформації.

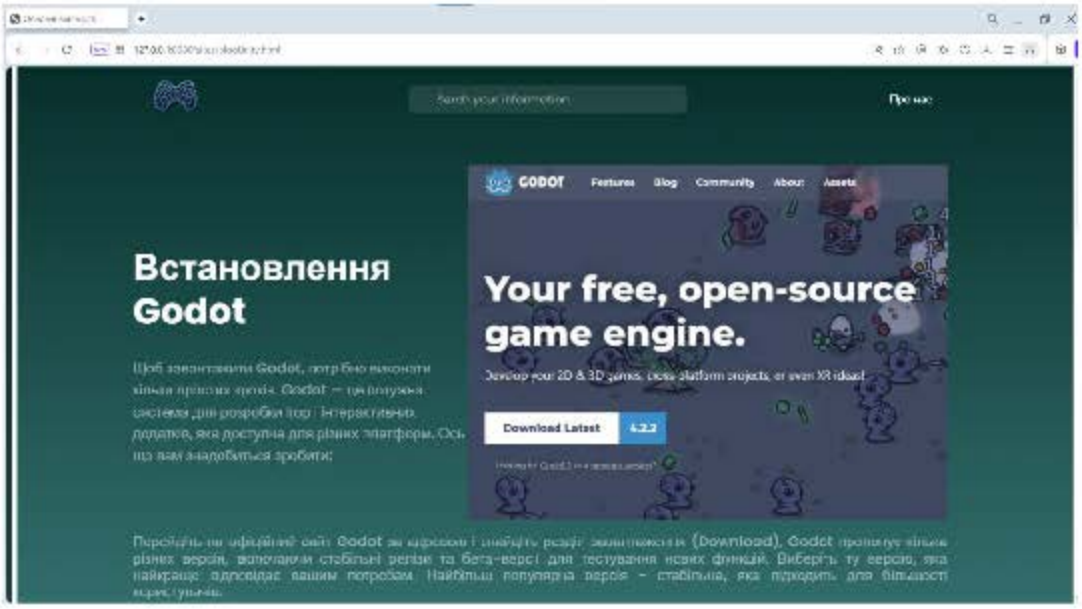

Рисунок 3.49 – Завантаження сторінки «Блог»

Для максимальної зручності перегляду інформації, на цій сторінці всі анімації блоків, за винятком навігаційного меню, були усунені. Це дозволяє користувачам миттєво знаходити необхідні дані без затримок. Мета сторінки – надавати чіткі відомості, пояснювати терміни та надавати документацію, як це робиться в Godot.

Кожен контентний блок супроводжується текстом, зображенням та кнопкою, яка веде на відповідну сторінку пов'язану з Godot (рис. 3.50).

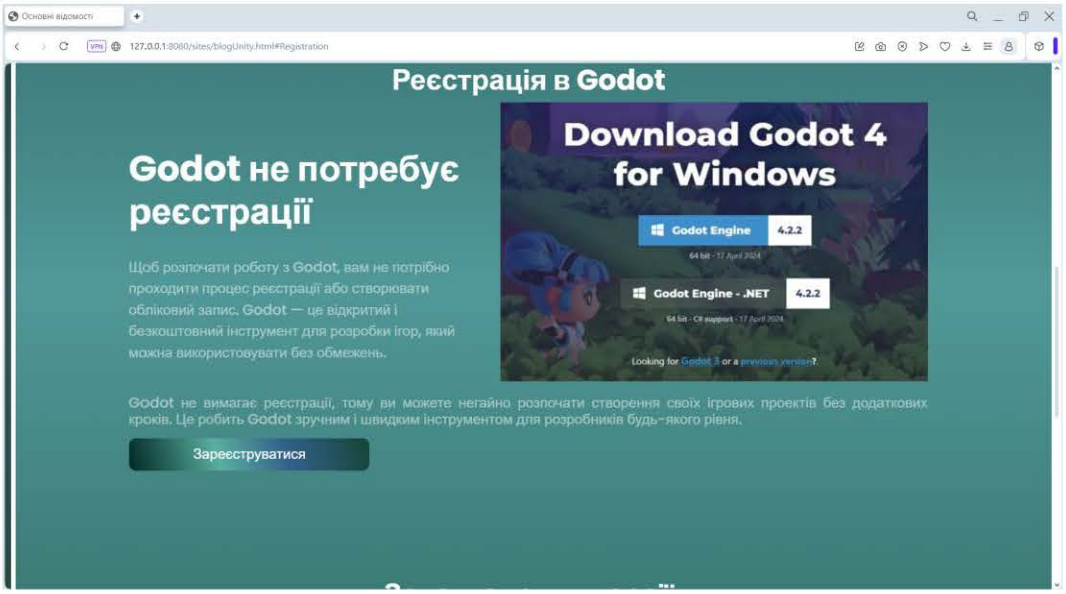

Рисунок 3.50 – Перегляд сторінки «Блог»

Щоб відкрити навігаційне меню, користувачеві необхідно перемістити курсор до лівого краю сторінки. Це викличе анімацію, яка плавно відкриє меню:

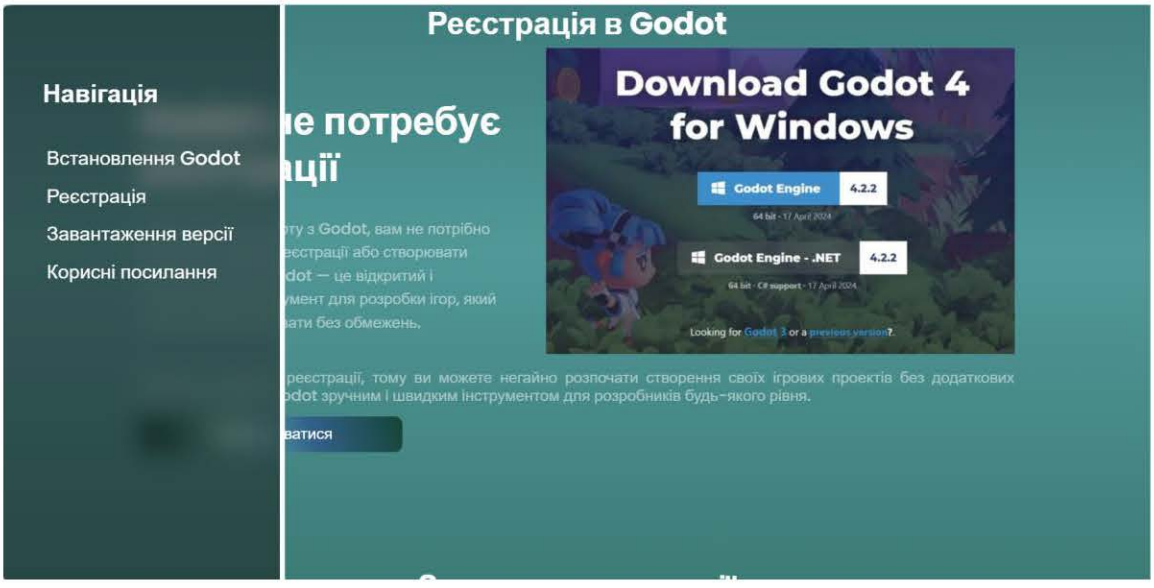

Рисунок 3.51 – Меню навігації

Це меню служить для навігації по сторінці. Для переходу до конкретної глави натисніть на відповідний пункт у меню (рис. 3.52):

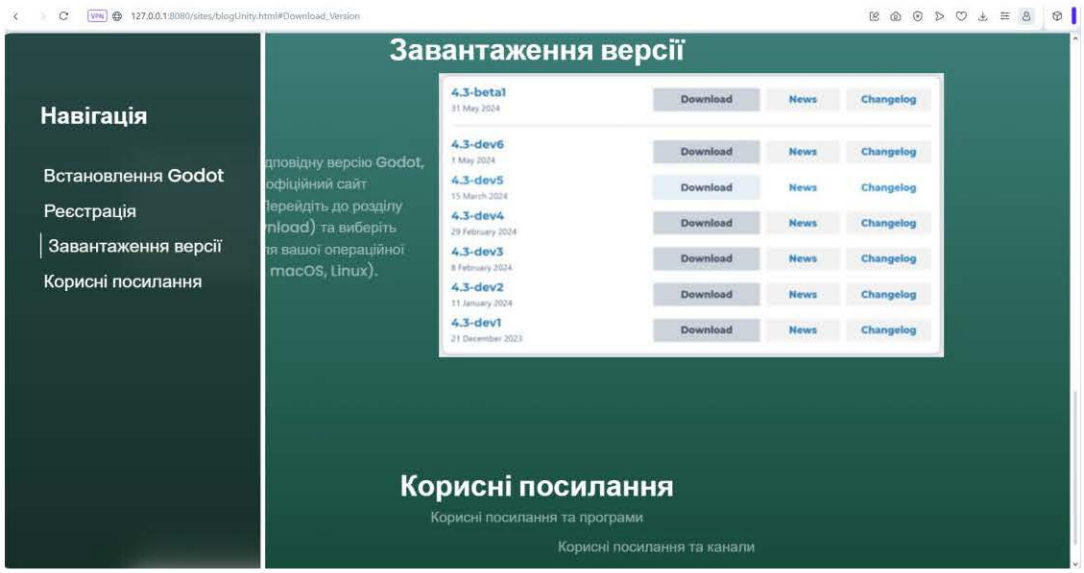

Рисунок 3.52 – Переміщення по сторінці через меню

Далі необхідно провести аналіз завантаження сторінки та зафіксувати результати. Для цього слід виконати кілька підготовчих кроків. Спочатку відкрийте інструменти розробника в браузері, щоб отримати доступ до детальних даних про завантаження. (рис. 3.53):

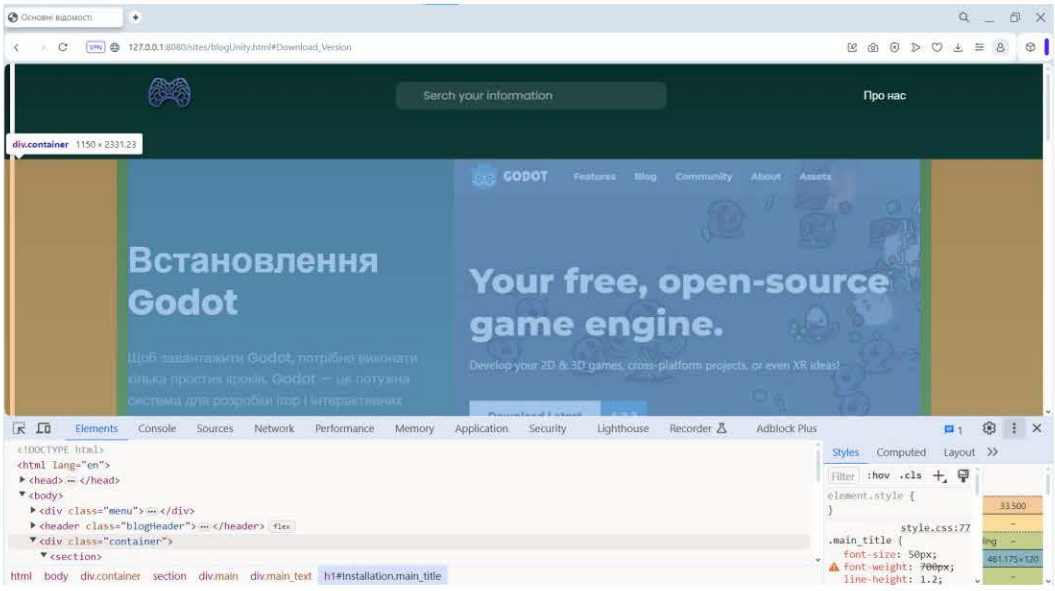

Рисунок 3.53 – Інструмент розробника

Далі слід перейти у вкладку «Application», щоб перевірити наявність кешованих даних. Якщо такі дані виявляться, необхідно їх очистити скориставшись кнопкою «Clear site data», таким чином забезпечимо чистий тестовий запуск. (рис. 3.54).

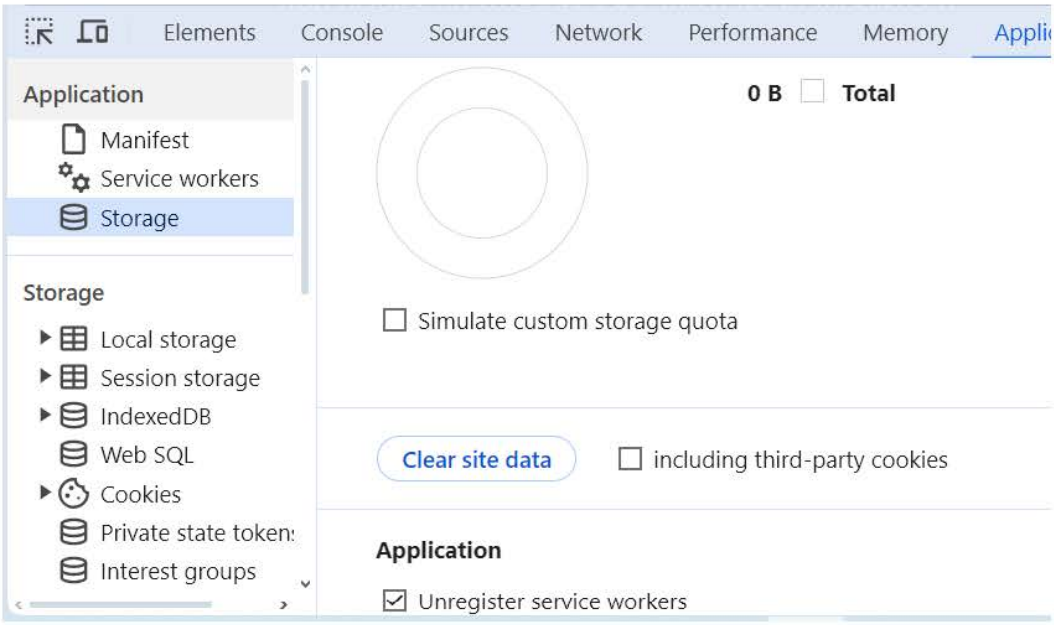

Рисунок 3.54 – Видалення кешу

Далі переходимо у вкладку «Network». Ця вкладка дозволяє нам детально відстежити процес завантаження кожного файлу та загальний час завантаження сторінки. На завершення потрібно встановити налаштування мережі на «Fast 3G» для моделювання умов повільної мережі та аналізу продуктивності (рис. 3.55).

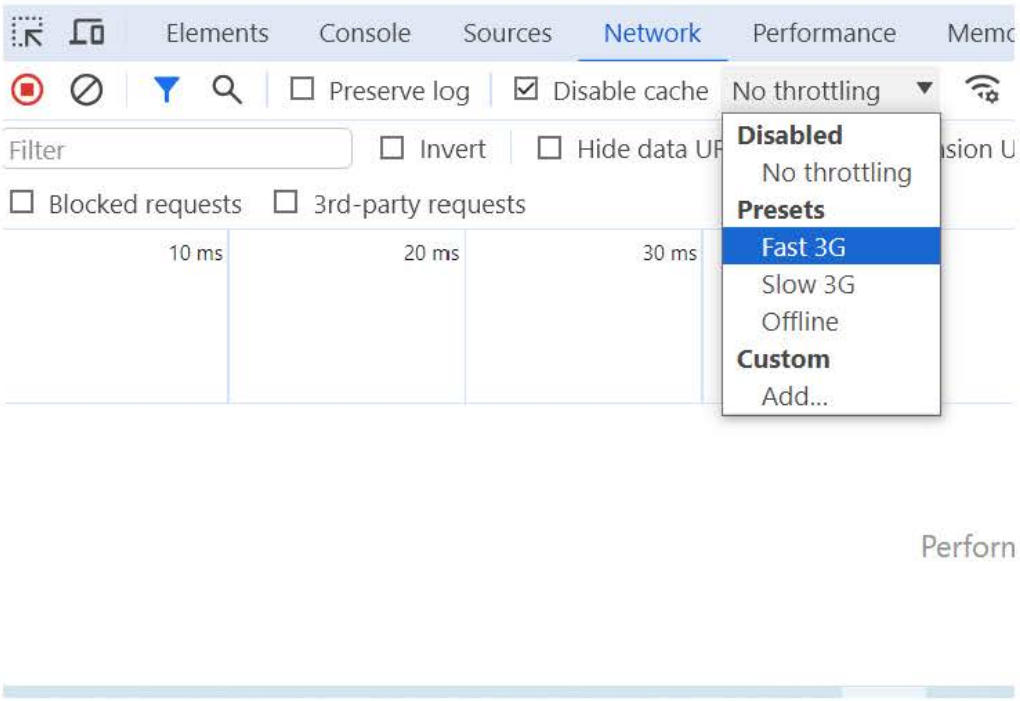

Рисунок 3.55 – Вкладка «Network»

Далі наступним кроком потрібно підключити сам скрипт алгоритму до сторінки вписавши 1 тег:

```
sites > blogUnity.html > html > head > script
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Основні відомості</title>
    <script src="scripts/cacheScript.js"></script>
    <link rel="stylesheet" href="stiles/normalize.min.css">
    <link rel="stylesheet" href="stiles/style.css">
    <link rel="stylesheet" href="stiles/blogStile.css">
</head>
<body>
    <div class="menu">
        <div class="blogtitle">
            <h1>Навігація</h1>
```

Рисунок 3.56 – додавання алгоритму на сторінку

Після перезавантаження сторінки користувач отримає випливаюче вікно в якому йому потрібно погодитись на кешування даних (рис. 3.57):

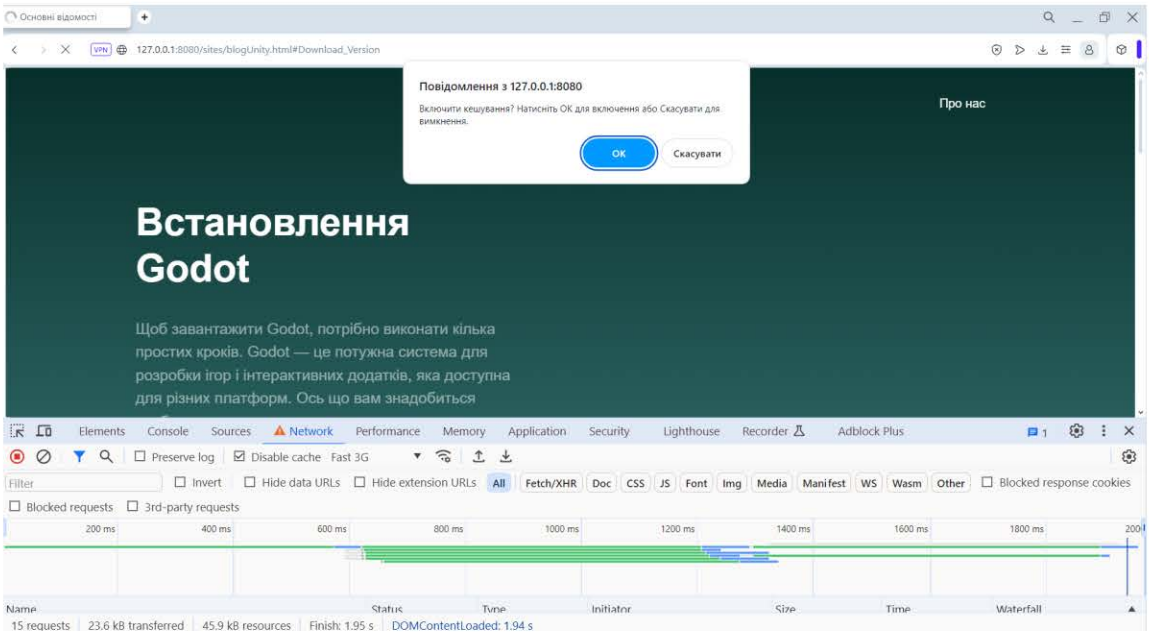

Рисунок 3.57 – Поява випливаючого вікна

Після перезавантаження знову повернемося у вкладку «Application» та проаналізуємо, скільки місця зайняло кешування даних, щоб оцінити ефективність використання ресурсів (рис. 3.59):

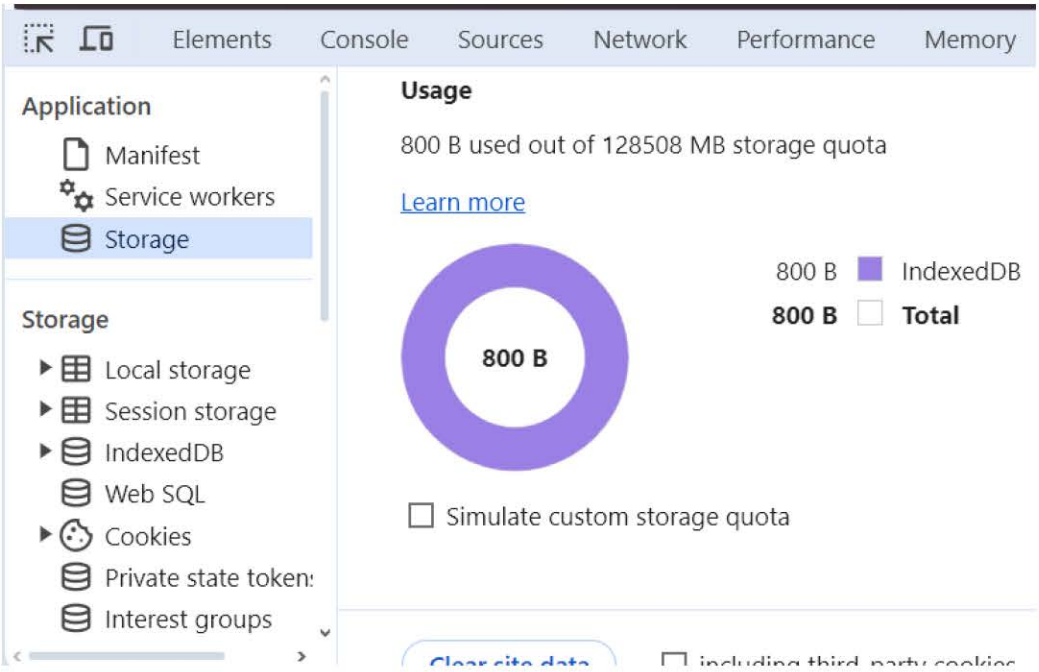

Рисунок 3.58 – Розгляд зарезервованої пам’яті

Після того як переконалися, що алгоритм працює і кешування відбулося успішно, потрібно перейти до вкладки «Network» та виконати перезавантаження сторінки для подальшого аналізу.

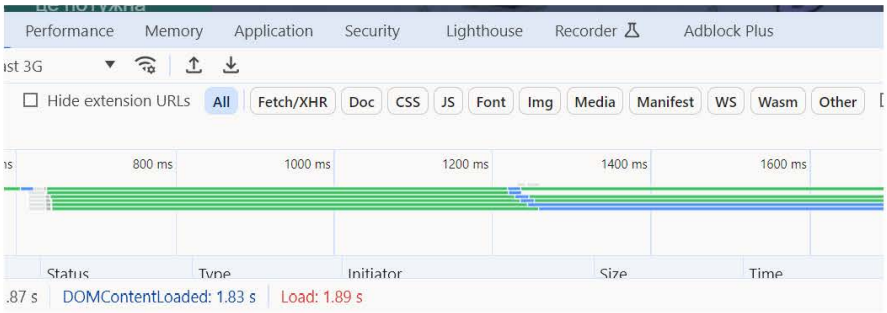

Рисунок 3.59 – Результати завантаження сторінок

Отже, використовуючи власний алгоритм кешування, час завантаження сторінки становить 1.86 секунд.

Тепер необхідно провести порівняння з алгоритмом «Lazy Loading». Для цього потрібно відключити кешування, натиснувши «Disable cache» (рис. 3.60), щоб забезпечити точні результати без впливу кешу.

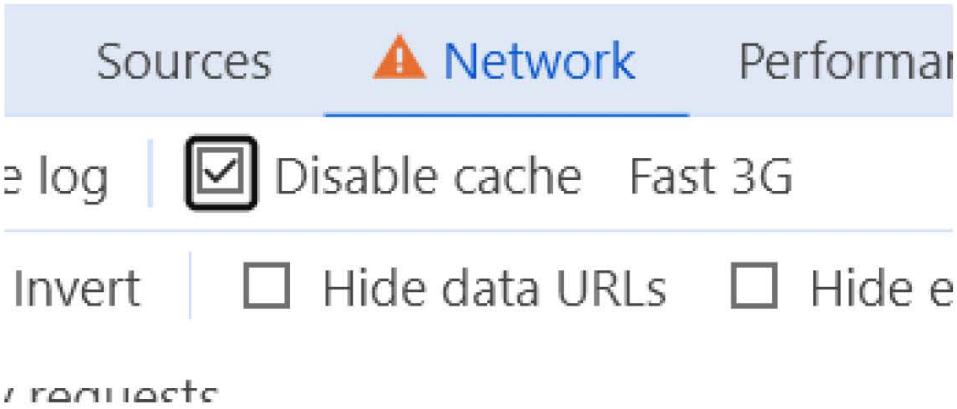

Рисунок 3.60 – Заборона на завантаження з кешу

Після перезавантаження сторінки час завантаження візуального контенту, збільшився до 3.59 секунд. Окрім повільного завантаження основної сторінки, відбувається також поступове дозавантаження додаткових візуальних елементів, таких як фотографії, що ще більше погіршує загальний користувацький досвід.

Name	Status	Type	Initiator	Size	Time
pxiByp8kv8JHgFVrLGT9Z1xlFQ.woff2	200	font	css2	(memory cache)	0 ms
pxiByp8kv8JHgFVrLCz7Z1xlFQ.woff2	200	font	css2	(memory cache)	0 ms
pxiByp8kv8JHgFVrLBT5Z1xlFQ.woff2	200	font	css2	(memory cache)	0 ms
logo.png	304	png	style.css	244 B	593 ms
ws	101	websocket	main.html:128	0 B	Pending
card_foto3.png	304	png	lazyLoading.js:15	245 B	593 ms
card_foto2.png	304	png	lazyLoading.js:15	245 B	605 ms
card_foto1.png	304	png	lazyLoading.js:15	245 B	599 ms

Рисунок 3.61 – Результати завантаження з алгоритмом «Lazy Loading»

Отже, кешування даних є більш ефективним для завантаження сторінки ніж з алгоритмом «Lazy Loading».

Впровадження автоматизованого алгоритму кешування дозволило успішно вирішити основні завдання оптимізації продуктивності веб-додатку, забезпечуючи швидкий доступ до необхідної інформації та покращуючи загальний користувацький досвід.

3.5 Висновок до третього розділу

У третьому розділі було розглянуто процес розробки, тестування та впровадження веб-додатку з використанням алгоритмів кешування для оптимізації швидкодії. Здійснено детальний аналіз і порівняння двох основних методів оптимізації: власного алгоритму кешування та методу "Lazy Loading".

Розробка алгоритму кешування даних довела свою ефективність, значно зменшуючи час завантаження сторінки порівняно з алгоритмом "Lazy Loading". Під час тестування було виявлено, що використання кешування дозволяє завантажувати сторінку за 1.86 секунд, тоді як "Lazy Loading" збільшує цей час до 3.59 секунд. Це підтверджує, що власний алгоритм кешування є більш продуктивним для даного типу веб-додатку.

Важливим аспектом стало використання сучасних технологій і платформ для розробки, таких як Node.js та JavaScript. Це забезпечило високу ефективність і гнучкість розробки, дозволило інтегрувати складні анімації та ефекти, такі як паралакс і анімовані випадаючі списки, що покращує користувацький досвід.

Отже, запропонована оптимізація швидкодії веб-додатку довела свою ефективність та може бути використана для вдосконалення інших веб-додатків.

ВИСНОВОК

Кваліфікаційна робота присвячена актуальному питанню оптимізації швидкодії веб-додатків для забезпечення ефективної роботи сучасних онлайн-платформ. Збільшення обсягу даних та складність інтерактивних функцій значно підвищують вимоги до продуктивності веб-додатків, що обумовлює необхідність постійного вдосконалення технологій, методів та підходів до забезпечення високої швидкості завантаження та обробки інформації. Метою роботи було дослідження методів оптимізації завантаження веб-додатків та розробка алгоритму оптимізації швидкодії веб-блогу.

У першому розділі кваліфікаційної роботи було проведено дослідження предметної області, яке включало аналіз сучасних методів оптимізації завантаження контенту. Було встановлено, що питання оптимізації швидкодії веб-додатків набуває особливої актуальності через необхідність забезпечення швидкого завантаження сторінок, зниження навантаження на сервери та підвищення загальної продуктивності системи. Особливу увагу було приділено кешуванню даних, що дозволяє значно зменшити час завантаження сторінок, знизити навантаження на сервер та підвищити загальну продуктивність системи. Ефективне використання кешування стає особливо важливим при розробці складних веб-додатків з високим трафіком користувачів та великим обсягом динамічного контенту.

Крім цього, сучасні веб-додатки мають інтегрувати різноманітні мультимедійні та анімаційні ефекти, що також підвищує вимоги до швидкодії системи. Технології, такі як паралакс, анімаційні списки, а також оптимізація обробки зображень і відео, стають невід'ємною частиною сучасного веб-дизайну. Вони не лише підвищують естетичну привабливість веб-сайтів, але й можуть значно впливати на їхню продуктивність.

У другому розділі кваліфікаційної роботи було проаналізовано основні методи та технології оптимізації швидкодії веб-додатків. Розглянуто чотири основні методи рендерингу інтерфейсу користувача у веб-додатках: серверний

рендеринг (SSR), статичний рендеринг, клієнтський рендеринг (CSR) та використання WebSockets.

Серверний рендеринг (SSR) полягає в тому, що сервер генерує та надсилає клієнту готовий HTML, мінімізуючи обсяги передаваного JavaScript. Цей підхід забезпечує швидке первісне завантаження сторінок, що особливо корисно для веб-сайтів з високим трафіком та великим обсягом динамічного контенту. Статичний рендеринг, у свою чергу, забезпечує стабільно швидкий час до першого байта (TTFB), оскільки HTML сторінок не потрібно генерувати в реальному часі. Клієнтський рендеринг (CSR) забезпечує динамічний та інтерактивний користувацький досвід, що підходить для додатків, які вимагають багато взаємодії на стороні клієнта. Використання WebSockets є оптимальним вибором для сценаріїв, що потребують негайного обміну даними, таких як онлайн-ігри, торгові платформи або чати.

Було обрано веб-кешування як основний метод оптимізації для розробки алгоритму завантаження веб-додатку. Веб-кешування дозволяє тимчасово зберігати копії веб-документів та медіафайлів для мінімізації затримок у відповідях сервера. Це значно підвищує швидкість завантаження сторінок та покращує користувацький досвід.

У третьому розділі кваліфікаційної роботи було розроблено та протестовано веб-додаток з використанням алгоритмів кешування для оптимізації швидкодії. Основна структура веб-додатку включала три сторінки: "Головна", "Про нас" та "Блог". Для розробки використовувались сучасні технології, такі як Node.js та JavaScript, що забезпечило високу ефективність та гнучкість розробки. Було реалізовано алгоритм кешування даних, який дозволяє значно зменшити час завантаження сторінок та підвищити швидкодію веб-додатку.

Результати тестування показали, що використання власного алгоритму кешування забезпечує більш ефективне завантаження сторінок порівняно з методом "Lazy Loading". Під час тестування було виявлено, що використання кешування дозволяє завантажувати сторінку за 1.86 секунд, тоді як "Lazy

Loading" збільшує цей час до 3.59 секунд. Це підтверджує, що власний алгоритм кешування є більш продуктивним для даного типу веб-додатку.

Важливим аспектом стало використання сучасних технологій і платформ для розробки, таких як Node.js і JavaScript. Це забезпечило високу ефективність і гнучкість розробки, дозволило інтегрувати складні анімації та ефекти, такі як паралакс і анімовані випадаючі списки, що покращує користувацький досвід.

Розроблений алгоритм кешування та впроваджені методи оптимізації довели свою ефективність та можуть бути використані для вдосконалення інших веб-додатків. Результати дослідження можуть служити основою для подальших робіт у галузі оптимізації веб-додатків, сприяючи покращенню продуктивності та користувацького досвіду.

Для подальшого вдосконалення розробленого веб-додатку рекомендовано розширити функціональність алгоритму кешування, враховуючи динамічний контент та персоналізовані дані користувачів. Також варто звернути увагу на інтеграцію додаткових методів оптимізації, таких як використання сучасних фреймворків та бібліотек для розробки веб-додатків, що забезпечують ще більш високу продуктивність та гнучкість.

Загалом, проведене дослідження та розробка алгоритму кешування показали значний потенціал для покращення швидкодії веб-додатків, що є важливим аспектом для забезпечення конкурентоспроможності та успішності сучасних онлайн-платформ. Розроблені підходи та алгоритми можуть бути успішно застосовані для вдосконалення інших веб-додатків, забезпечуючи високу швидкість завантаження, стабільність та привабливий користувацький досвід.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Iskandar, Taufan Fadhilah, et al. Comparison between client-side and server-side rendering in the web development. In: IOP Conference Series: Materials Science and Engineering. IOP Publishing, 2020. p. 21-36.

2. NAKANO, Yuusuke, et al. Web performance acceleration by caching rendering results. In: 2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS). IEEE, 2015. p. 244-249.

3. Iskandar, T. F., Lubis, M., Kusumasari, T. F., & Lubis, A. R. (2020, May). Comparison between client-side and server-side rendering in the web development. In IOP Conference Series: Materials Science and Engineering (Vol. 801, No. 1, p. 012136). IOP Publishing.

4. Nakano, Y., Kamiyama, N., Shiomoto, K., Hasegawa, G., Murata, M., & Miyahara, H. (2015, August). Web performance acceleration by caching rendering results. In 2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS) (pp. 244-249). IEEE.

5. Barker, R. (2014). High Performance Responsive Design: Building Faster Sites Across Devices. Apress.

6. Osmani, A. (2018). Practical Performance Profiling: Improving the Efficiency of JavaScript Applications. O'Reilly Media.

7. Mohan C. Caching Technologies for Web Applications – [Електронний ресурс] – Режим доступу: http://infolab.stanford.edu/infoseminar/archive/SpringY2002/speakers/mohan/mohan.pdf

8. J. Mertz, I. Nunes. Understanding Application-level Caching in Web Applications: a Comprehensive Introduction and Survey of State-of-the-art Approaches [Електронний ресурс] – Режим доступу: https://arxiv.org/pdf/2011.00477.pdf

9. ЩО таке UML-діаграми? – [Електронний ресурс] – Режим доступу: https://evergreens.com.ua/ua/articles/uml-diagrams.html

10. Node.js. [Електронний ресурс] – Режим доступу: https://nodejs.org/en

11. JavaScript Guide - JavaScript | MDN. [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

12. Learn to style HTML using CSS - Learn web development | MDN. [Електронний ресурс] – Режим доступу: https://developer.mozilla.org/en-US/docs/Learn/CSS

13. Code.org CS Workshop — Changing Expectations. [Електронний ресурс] – Режим доступу: https://www.changeexpectations.org/code-org

14. HTML Tutorial. [Електронний ресурс] – Режим доступу: https://www.w3schools.com/html/

15. CSS Tutorial. [Електронний ресурс] – Режим доступу: https://www.w3schools.com/css/