**Міністерство освіти і науки України**
**Національний технічний університет**
**«Дніпровська політехніка»**

Інститут електроенергетики
(інститут)
Факультет інформаційних технологій
(факультет)
Кафедра інформаційних технологій та комп'ютерної інженерії
(повна назва)

# ПОЯСНЮВАЛЬНА ЗАПИСКА
**кваліфікаційної роботи ступеня** ___магістра___
(бакалавра, спеціаліста, магістра)

**студента** _____Комаристого Миколи Миколайовича_____
(ПІБ)
**академічної групи** _____123м-20-1_____
(шифр)
**спеціальності** _____123 «Комп'ютерна інженерія»_____
(код і назва спеціальності)
**за освітньо-професійною програмою** ___«Комп'ютерна інженерія»___
(офіційна назва)
**на тему** «Обґрунтування вибору параметрів клієнт-серверної архітектури комп'ютерної системи з протоколом Web Socket та технологією SSR»
(назва за наказом ректора)

| **Керівники** | **Прізвище, ініціали** | **Оцінка за шкалою** | | **Підпис** |
|---|---|---|---|---|
| | | **рейтинговою** | **інституційною** | |
| кваліфікаційної роботи | проф. Цвіркун Л.І. | | | |
| розділів: | | | | |
| теоретичний розділ | проф. Цвіркун Л.І. | | | |
| синтез системи | доц. Ткаченко С.М. | | | |
| розроблення програмного забезпечення | ас. Бешта Л.В. | | | |
| **Рецензент** | | | | |
| **Нормоконтролер** | проф. Цвіркун Л.І. | | | |

**Дніпро**
**2022**

**ЗАТВЕРДЖЕНО:**

завідувач кафедри
інформаційних технологій
та комп'ютерної інженерії
(повна назва)

_____ __Гнатушенко В.В._
(підпис) (прізвище, ініціали)
«___»_____ 202__ року

**ЗАВДАННЯ**
**на кваліфікаційну роботу**
**ступеня _____магістр _____**
(бакалавра, спеціаліста, магістра)

**студенту** _____Комаристому М.М._____ **академічної групи** ____123м-20-1____
(прізвище та ініціали) (шифр)

**спеціальності** _____123 «Комп'ютерна інженерія»_____

**за освітньою-професійною програмою** _123 «Комп'ютерна інженерія» _
(офіційна назва)

**на тему** «Обґрунтування вибору параметрів клієнт-серверної архітектури комп'ютерної системи з протоколом Web Socket та технологією SSR»,

затверджену наказом ректора НТУ «Дніпровська політехніка» від 10.12.2021 р. №1036

| Розділ | Зміст | Термін виконання |
|---|---|---|
| Стан питання та постановка завдання | На основі матеріалів виробничих практик, інших науково-технічних джерел сформулювати наукове завдання, конкретизувати предмет та мету досліджень | 20.09.2021 |
| Теоретичний | Обґрунтувати теоретичну базу розв'язання наукового завдання, якому присвячено роботу | 25.10.2021 |
| Синтез системи | Розробка комп'ютерної системи | 15.11.2021 |
| Розроблення програмного забезпечення | Розробка програмного забезпечення | 29.11.2021 |
| Експериментальний розділ | Проведення і обробка результатів експериментів | 15.12.2021 |
| Графічна частина | Графічні результати роботи подати у вигляді рисунків схем таблиць на 10 арк. формату А4. | 05.01.2022 |

**Завдання видано** _____ проф. Цвіркун Л. І.
(підпис керівника) (прізвище, ініціали)

**Дата видачі** 06 вересня 2021 р.

**Дата подання до екзаменаційної комісії** 10.01.2022 р.

**Прийнято до виконання** _____ Комаристий М. М.
(підпис студента) (прізвище, ініціали)

# РЕФЕРАТ

Пояснювальна записка: 110 с., 31 рис., 9 табл., 13 джерел, 1 додаток.

Об’єкт розробки - процес розробки та впровадження веб-застосунку шляхом реалізації клієнт-серверної архітектури та за допомогою використання технології серверного рендерингу та передачею даних через WebSocket протокол.

Мета дослідження полягає у підвищенні ефективності розробки, збільшення швидкодії клієнтської та серверної частини застосунку, та покращення SEO оптимізації клієнтської частини.

У пояснювальні записці було проведено аналіз сучасних тенденцій та підходів в організації сучасних клієнт-серверних систем та веб-додатків. На основі результатів аналізу було сформульовано задачі на розробку системи.

Наступним кроком в роботі, було рішення поставлених наукових завдань. В результаті цього кроку обрано технології та методи розробки системи, розроблено алгоритм роботи та сформульовано методи експериментальних досліджень.

Після вирішення наукового завдання, було визначено сферу застосування та призначення системи, технічні вимоги до неї та вимоги до видів забезпечення, для того, щоб далі створити систему контролю.

Далі, було розроблено робочий програмний прототип на основі опису технічних характеристик програми, опису вибраних технічних та програмних вимог та алгоритму роботи програми.

Зрештою, на основі обраної методики експерименту було проведено експеримент, та зроблено аналіз отриманих результатів.

КОМП’ЮТЕРНА СИСТЕМА, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, WEBSOCKET, SSR.

## ABSTRACT

Explanatory note: 110 pages, 31 figures, 9 tables, 13 sources, 1 appendix.

The object of development is the process of developing and implementing a web application by implementing a client-server architecture and by using server-based rendering technology and data transfer via the WebSocket protocol.

The purpose of the study is to increase the efficiency of development, increase the speed of the client and server part of the application, and improve SEO optimization of the client part.

The explanatory note analyzed current trends and approaches in the organization of modern client-server systems and web applications. Based on the results of the analysis, tasks for system development were formulated.

The next step in the work was to solve scientific problems. As a result of this step, the technologies and methods of system development were selected, the algorithm of work was developed and the methods of experimental research were formulated.

After solving the scientific problem, the scope and purpose of the system, technical requirements for it and requirements for the types of software were determined in order to further create a control system.

Next, a working software prototype was developed based on the description of the technical characteristics of the program, the description of the selected technical and software requirements and the algorithm of the program.

Finally, based on the chosen method of experiment, an experiment was conducted and the results were analyzed.

COMPUTER SYSTEM, CLIENT-SERVER ARCHITECTURE, WEBSOCKET, SSR.

**ЗМІСТ**

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, СКОРОЧЕНЬ І ТЕРМІНІВ

WebSocket – це протокол, що призначений для обміну інформацією між браузером та веб-сервером в режимі реального часу.

HTTP – HyperText Transfer Protocol, протокол передачі гіпертекстових документів. Тест – ап

SSR – Server-Side Rendering, рендеринг даних на стороні серверу.

SPA – Single-Page Application, односторінковий застосунок.

API – Application Programming Interface, прикладний програмний інтерфейс.

ОС – операційна система.

# ВСТУП

В останнє десятиріччя істотно збільшилася доля веб-розробки в загальному обсязі IT індустрії. На сьогоднішній день на це також дуже вплинула пандемія (COVID-19). Обумовлено це тим, що такий тяжкий час доступ до продукту можливо отримати, якщо в тебе є майже любий пристрій та підключення до мережі інтернет. Бізнес зрозумів, що потрібно переводити сервіси в Інтернет. Дуже багато продуктів які розробляються, є насамперед веб-додатками або веб-аналогами різних продуктів. Прикладом може стати ріст популярності продажу продуктів як сервісу, наприклад: інфраструктура як послуга(IaaS), платформа як послуга(PaaS) та програмне забезпечення як послуга(Saas). Також належну популярність отримала концепція інтернету речей (IOT), інтерфейси для якої здебільшого розробляються як веб-додатки.

Якість та підходи розробки прямо пропорційно впливають на подальший успіх продукту. Перед тим як почати розробку важливо визначитися з типом відображення даних. В залежності від вимог до фінального результату виділяють три основних типів: відображення даних на стороні серверу(SSR), на стороні клієнта(CSR) та статична генерація сайтів(SSG).

Один із основних показників, по якому можливо припустити чи буде додаток успішний це його швидкість роботи. Якщо додаток повільний, то скоріш за все користувач просто покине його, навіть не дочекавшись завантаження контенту. Оптимізація продукту допомагає більш ефективно бізнесу досягати своїх цілей, наприклад: отримати клієнтів, збільшити продажі чи зменшити витрати на утримання.

Важливою частиною ефективності додатку також є досконала оптимізація для пошукових систем(SEO). Після проведення такої оптимізації збільшиться кількість пошукових видач, і як наслідок продукт буде більш успішний.

Одною з основних проблем в сучасній розробці є розмір файлів які користувач буде завантажувати. Вона закладаються в тому, що перед тим, як користувач

побачить хочаб якийсь контент, необхідно завантажити більшість файлів, які передбачає веб-додаток. Розмір цих файлів обчислюється мегабайтами, і користувачам із повільним інтернетом або слабкими пристроєм може не дочекатися їх завантаження, або після відображення контенту пристрій буде гальмувати.

Однак не менш помітний вплив на швидкість завантаження сторінки справляє і той факт, що всі запити на отримання файлів виконує браузер. Затрачується час на пошук DNS сервера, підключення, встановлення SSL сертифікатів, відправку запитів, очікування формування відповіді сервером та отримання відповіді. Не зовсім радісним є той факт, що браузер не в змозі обробляти всі запити, які хоче отримати сторінка. Він має обмеження на кількість відкритих з'єднань, і зазвичай це обмеження становить від 4 до 6 запитів. І якщо сторінка перевищить максимальну кількість дозволених з'єднань, то браузеру доведеться оброблювати їх по черзі групами, що займе значно більший час, аніж очікувалось.

**Об'єкт дослідження:**

- процес розробки веб додатків за допомогою використання різних типів рендерингу даних;
- процес розробки та впровадження веб-застосунку шляхом реалізації клієнт-серверної архітектури та за допомогою використання технології серверного рендерингу та передачею даних через WebSocket протокол.

**Предмет дослідження** – типи рендерингу і передачі даних в клієнт серверних веб-застосунках та методи їх реалізації.

**Завдання дослідження:** Підвищення якості розробки веб застосунків, шляхом зменшення навантаження на клієнтські пристрої, та його оптимізації для пошукових систем.

**Мета дослідження:** Полягає у підвищенні ефективності розробки, збільшення швидкодії клієнтської та серверної частини застосунку, та покращення SEO оптимізації клієнтської частини. Для досягнення поставленої мети дослідження необхідно вирішити такі **завдання:**

- дослідити недоліки та переваги типів рендерингу даних які використовуються в сучасних веб-застосунках;

- порівняти протоколи за допомогою яких здійснюється передача даних між клієнтською та серверною частиною;
- розробити алгоритм, за домогою якого буде здійснюватися серверний рендеринг даних, та їх подальша передача до клієнта через WebSocket протокол;
- програмно реалізувати алгоритм, і дослідити побудований прототип;
- за результатами досліджень зробити висновки щодо ефективності передачі відрендериних даних через протокол WebSocket.

**Ідея роботи:** полягає в тому, що через протокол WebSocket можливо організувати серверний рендеринг частини даних та швидко оновлювати їх у клієнта.

**Галузь застосування** – проектування та впровадження веб-додатків різної складності.

**Наукові положення:**

1. Встановлено, що застосування протоколу WebSocket при використанні серверного рендерингу на основі клієнт-серверної архітектури дозволить покращити пошукову видачу додатку та зменшити час на його завантаження і розмір файлів.

**Наукові результати:**

1. Отримані нові теоретичні знання щодо побудови клієнт-серверних застосунків, протоколу WebSocket та типів рендерингу;
2. Запропонований новий метод організації часткового оновлення інтерфейсу, шляхом відправки запиту до сервера через WebSocket, та отримання відображених даних на сервері;
3. Обґрунтовано ефективності застосування даної програмної системи для застосування у сфері надання інформаційних послуг за допомогою мережі інтернет.

**Обґрунтованість і достовірність наукових положень, висновків і рекомендацій** підтверджуються тим, що в роботі використані сучасні технології і

підходи до розробки клієнт-серверних додатків результативність яких підтверджена рядом експериментальних досліджень.

**Практичне значення отриманих результатів полягає** в розробці нового підходу до розробки додатків, які дозволяють отримати позитивний економічний ефект для сфери діяльності, в якій система буде використовуватися.

# 1 СТАН ПИТАННЯ ТА ПОСТАНОВКА ЗАДАЧ

## 1.1 Ключові технології в розробці веб застосунків

Сучасні веб технології надають розробникам необмежені можливості для реалізації своїх ідей. Для того, щоб використовувати весь їх потенціал, необхідно знати ці технології, і те, як їх використовувати. Веб-розробка одна з найпростіших і популярних напрямків серед інженерів програмного забезпечення. Низький поріг входу обумовлений тим, що для успішного старту роботи не потрібно вивчати алгоритми на високому рівні, принцип написання програм достатньо очевидний. Достатньо лише вивчити конструкції мови та базовий синтаксис, і вже можливо досягти результатів.

Для базового початку роботи необхідно вивчити невеликий стек технологій: HTML, CSS, JS. Цього буде достатньо, щоб охопити більшість можливість варіантів задач, які можуть бути поставлені перед розробником.

### 1.1.1 Мова гіпертекстової розмітки – HTML

HTML (HyperText Markup Language, мова гіпертекстової розмітки) – це система для будування розмітки яка визначає, як і які елементи повинні розміщуватися на веб-сторінці. Під гіпертекстом(«hypertext») натякають, на посилання, які з’єднують веб-сторінки один з одним або в межах одного веб-сайту [1]. Посилання є фундаментальним аспектом Інтернету. Завантажуючи контент в інтернет, та пов'язуючи його зі сторінками, створеними іншими людьми, ви стаєте активним учасником Всесвітньої паутини. HTML використовує розмітку (‘markup’) для відображення інформації різного плану, наприклад: тексту, зображень, відео і так далі. Ця розмітка включає в себе спеціальні елементи, такі як <head>, <title>, <body>, <header>, <footer>, <article>, <section>, <p>, , , <img>, <aside>, <audio >, <canvas>, <datalist>, <details>, <embed>, <nav>, <output>, <progress>, <video> та багато інших. Контент, який знаходиться всередині цих елементів і відображається для користувачів. Ці елементи виділяються за допомогою тегів, які складаються з

імені елемента оточеного «<» та «>» символами, та «/» для закриваючого тега. Приклад побудови стандартного тега показано на рисунку 1.1.

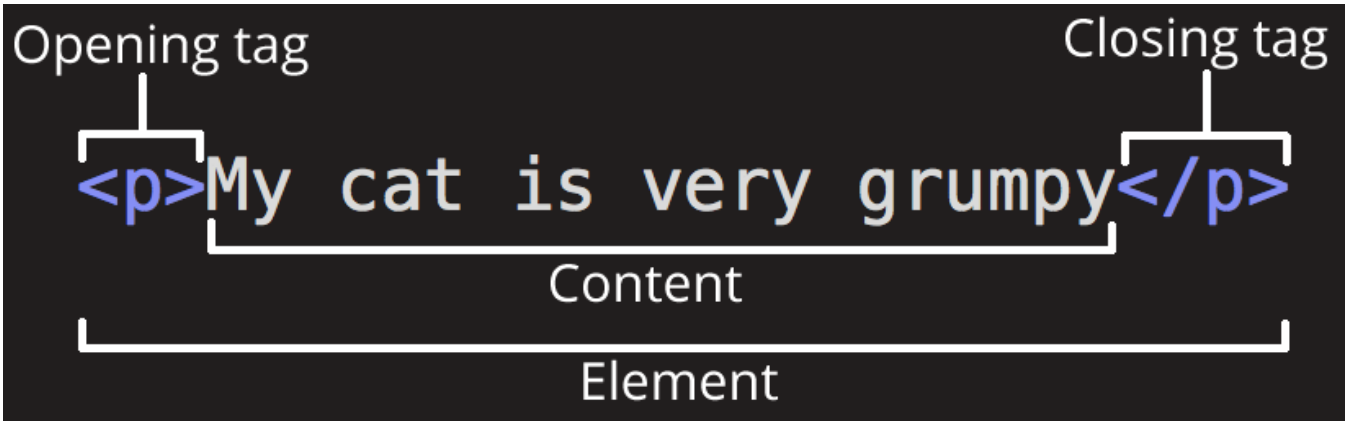


Рисунок 1.1 – Структура тега

Головними частинами нашого елемента є:

– Відкриваючи тег. Складається з імені елемента (в даному випадку, "p"), укладеного в кутові дужки, що відкривають і закривають. Відкриваючий тег вказує, де елемент починається чи починає діяти, у нашому випадку — де починається абзац;
– Закриваючий тег. Це те саме, що й відкриваючий тег, за винятком того, що він включає в себе косу риску перед ім'ям елемента. Закриваючий елемент показує, де елемент закінчується, у нашому випадку — де закінчується абзац;
– Контент. Це зміст елемента, який у разі є просто текстом;
– Елемент. Відкриваючий тег, закриваючий тег і контент разом складають елемент;

Елементи також можуть мати атрибути, які містять додаткову інформацію про елемент, який ви не бажаєте показувати і фактичному контенті. Вони можуть бути вкладені один в інший, або взагалі не мати контенту, а тільки атрибути.

Деякі елементи не мають особливого сенсу для користувача, але за їх допомогою розробник може сформувати анатомію документа. На рисунку 1.2 показано базове дерево HTML розмітки для сторінки.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Моя тестовая страница</title>
  </head>
  <body>
    <img src="images/firefox-icon.png" alt="Моё тестовое изображение">
  </body>
</html>
```

Рисунок 1.2 – Структура HTML документа

На цьому рисунку ми бачимо деякі складові частини документа:

– <!DOCTYPE html> – тип документу. У минулому, коли HTML був «молодим», тип документу виступав як посилання на набір правил, яким HTML сторінка мала слідувати, щоб вважатися хорошим HTML, що могло означати автоматичну перевірку помилок та інші корисні речі. Однак у наші дні ніхто не дбає про це, і вони насправді просто історичний артефакт, який має бути включений для того, щоб усе працювало правильно;

– <html></html> – елемент <html>. Цей елемент обертає весь контент на всій сторінці, та іноді його називають кореневим елементом;

– <head></head> – елемент <head>. Цей елемент виступає як контейнер для всього, що ви бажаєте включити на HTML сторінку, але це не стане контентом, який ви показуєте користувачам вашої сторінки. До них відносяться такі речі, як ключові слова та опис сторінки, які будуть з'являтися в результатах пошуку, CSS стилі нашого контенту, кодування та багато іншого;

– <body></body> – елемент <body>. В ньому міститься весь контент, який ви хочете показувати користувачам, коли вони відвідують вашу сторінку, будь то текст, зображення, відео, гри, звукозапису , що програються, або щось інше.

### 1.1.2 Каскадні таблиці стилей – CSS

CSS (Каскадні таблиці стилей) – використовується для стилізації та компонування веб сторінок – наприклад, для зміни шрифту, кольору, розміру та інтервалу вмісту [2]. Браузер має свої стилі за замовченням, це найголовніші стилі, які браузер застосовує до HTML, щоб гарантувати, що він буде читабельним, навіть якщо розробник не вказав стилі явно. Щоб стилізувати документ так, щоб він наприклад сподобався вашій аудиторії, і з'явився CSS.

CSS – це мова на основі правил: ви визначаєте правила, які визначають групи стилів та в підсумку повинні бути застосовані до певних елементів або їх груп на сторінці.

```
h1 {
    color: red;
    font-size: 5em;
}
```

Рисунок 1.3 – Приклад CSS правила

Це просте правило, яке описую що текст елемента h1 повинен бути заданого розміру та червоного кольору.

Правило відкривається за допомогою селектора. Цей селектор вибирає HTML – елемент, який ми збираємося стилізувати. У цьому випадку ми використовуємо заголовки першого рівня (<h1>). Потім у нас є набір фігурних дужок {}. Усередині них буде один або кілька оголошень, які набувають форми пари: властивості та її значення. Кожна пара вказує властивість елемента, яку ми вибираємо, а потім значення, яке хотіли б привласнити властивості.

Перед двокрапкою у нас є властивість, а після двокрапки — значення. CSS-властивості мають різні допустимі значення в залежності від того, яка властивість вказується. У даному прикладі ми маємо властивість color, яка може набувати різних колірних значень. Ми також маємо властивість font-size. Ця властивість може набувати різних значень розміру, як і властивості.

Таблиця стилів CSS містить багато таких правил, написаних одне за одним.

```
h1 {
    color: red;
    font-size: 5em;
}

p {
    color: black;
}
```

Рисунок 1.4 – Демонстрація групування стилей

### 1.1.3 Мова програмування – JavaScript

JavaScript – це мова програмування, яка дозволяє створювати динамічно оновлюваний контент на сторінці [3]. На приклад це може бути управління мультимедією, анімація зображень і так далі. Знання JavaScript вважається ключовою навичкою в контексті веб-розробки. Програми, що написані за допомогою цієї мови – називаються скриптами. Вони можуть вбудовуватись у HTML та виконуватися автоматично під час завантаження веб-сторінки. Скрипти поширюються та виконуються як простий текст. Їм не потрібна спеціальна підготовка чи компіляція для запуску.

JavaScript – це "безпечна" мова програмування. Він не надає низькорівневий доступ до пам'яті або процесора, тому що спочатку був створений для браузерів, які цього не вимагають.

У браузері для JavaScript є все, що пов'язано з маніпулюванням веб-сторінками, взаємодією з користувачем і веб-сервером.

Наприклад, у браузері JavaScript може:

- додавати новий HTML-код на сторінку, змінювати існуючий вміст, модифікувати стилі;
- реагувати на дії користувача, натискання миші, переміщення вказівника, натискання клавіш;
- надсилати мережеві запити на віддалені сервери, завантажувати та завантажувати файли;

– отримувати та встановлювати cookie, ставити запитання відвідувачу, показувати повідомлення;
– запам'ятовувати дані за клієнта («local storage»).

Перевагою JS є те, що це єдина браузерна технологія, яка має три основні речі:

– повна інтеграція з HTML/CSS;
– прості речі робляться просто;
– підтримується всіма основними браузерами та включений за замовчуванням.

Ще більш цікавою є функціональність, створена поверх основної мови JavaScript. Так звані інтерфейси прикладного програмування (API) надають вам додаткові функції для використання у вашому коді JavaScript.

API – це готові набори блоків коду, які дозволяють розробнику реалізовувати програми, які інакше було б важко чи неможливо реалізувати.

Вони зазвичай поділяються на дві категорії:

– API-інтерфейси браузера;
– сторонні API-інтерфейси.

API-інтерфейси браузера вбудовані у ваш веб-браузер і можуть відображати дані з навколишнього комп'ютерного оточення або робити корисні складні речі.

Вікно браузера, яке бачить користувач, відповідає об’єкту вікна(window), а HTML документ – об’єкту документа(document). Відповідно до об'єктної моделі документа («Document Object Model», коротко DOM), кожен HTML-тег є об'єктом. Вкладені теги є дітьми батьківського елемента. Текст, який знаходиться всередині тега, є об'єктом.

Всі ці об'єкти доступні за допомогою JavaScript, ми можемо використовувати їх, щоб змінити сторінку.

Наприклад, document.body об'єкт для тега <body>. DOM – це подання HTML-документу у вигляді дерева тегів.

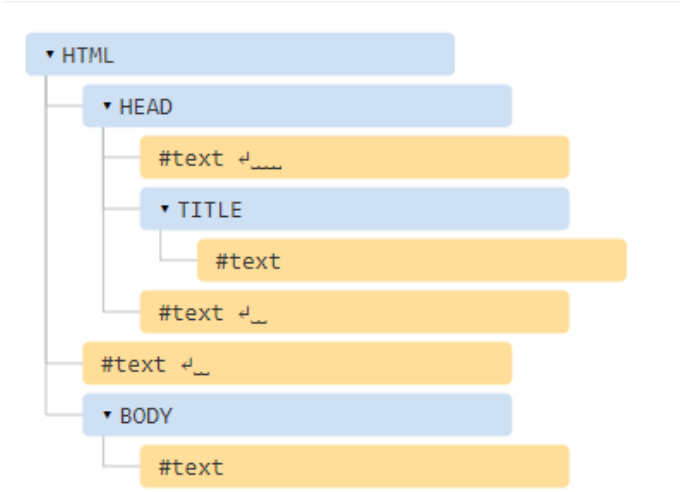


Рисунок 1.5 – Вигляд DOM дерева

Кожен вузол цього дерева – це об'єкт.

Теги є вузлами-елементами (або елементами). Вони утворюють структуру дерева: <html>– це кореневий вузол, <head>та <body>її дочірні вузли тощо.

Текст усередині елементів утворює текстові вузли , позначені як #text. Текстовий вузол містить лише рядок тексту. Він не може мати нащадків, тобто він знаходиться завжди на найнижчому рівні.

Ці об'єкти, як контейнери для збереження інформації. Вони мають свої властивості, методи і події, на які JavaScript може реагувати.

Завдяки DOM дереву JavaScript може редагувати та керувати HTML документом, робити його більш інтерактивним, а через CSSOM – керувати стилями окремих елементів.

### 1.2 Принцип роботи веб браузера

Здебільшого браузери розглядаються як однопотокові додатки. Щоб досягти плавності взаємодії, розробник повинен забезпечувати продуктивність у всьому, починаючи від плавного скролінгу до швидкої реакції на натискання екрана. Час рендеру – це ключове поняття. Розробник повинен забезпечити таку роботу програми, щоб усі його завдання могли бути виконані досить швидко. У такому випадку процесор буде вільний для обробки введення користувача. Щоб вирішити проблему

однопоточності, ви повинні зрозуміти природу браузерів і навчитися розвантажувати основний потік процесу там, де це можливо і допустимо.

### 1.2.1 Навігація

Навігація – це перший етап при завантаженні програми. Він відбувається кожного разу, коли користувач запитує сторінку, вводячи URL-адресу в адресний рядок браузера, натискає на посилання, відправляє заповнені поля форми і виконує деякі інші дії.

#### 1.2.1.1 DNS запит

Перший крок навігації до сторінки – це пошук місця, звідки потрібно запитувати дані. Ваш браузер запитує DNS запис. Як правило, запит містить ім'я сервера, який має бути перетворений на IP-адресу. Відповідь на цей запит буде збережено в кеші пристрою, щоб його можна було швидко отримати при наступному запиті до того ж сервера.

DNS запит зазвичай потрібно зробити лише один раз під час завантаження сторінки. Однак, DNS запити повинні бути виконані для кожного унікального імені хоста, які запитуються сторінкою. Скажімо, якщо ваші шрифти, картинки, скрипти, реклама або лічильники аналітики знаходяться на різних доменах, запит DNS буде здійснений для кожного з них.

Це може бути проблемою з точки зору продуктивності, особливо для мобільних мереж. Коли користувач знаходиться в мобільній мережі, кожен запит DNS повинен пройти від мобільного пристрою до стільникової вежі, а вже звідти дійти до DNS-сервера. Відстань та перешкоди між телефоном, вежею та сервером імен можуть значно збільшити затримку.

#### 1.2.1.2 TCP Рукостискання

У той момент, коли IP адреса стає відома, браузер починає встановлення з'єднання до сервера за допомогою рукостискання. Цей механізм спроектований так, щоб два пристрої, які намагаються встановити зв'язок, могли обмінятися параметрами

з'єднання, перш ніж приступати до передачі даних. Найчастіше - через захищене з'єднання HTTPS.

Трьох етапне рукостискання TCP – це техніка, що дуже часто згадується як "SYN-SYN-ACK" (SYN, SYN-ACK, ACK, якщо бути точніше), так як під час встановлення з'єднання передаються 3 повідомлення. Це означає, що перед встановленням з'єднання браузер повинен обмінятися ще трьома повідомленнями з сервером.

#### 1.2.1.3 TLS Переговори

Для встановлення безпечних з'єднань з використанням HTTPS потрібне ще одне рукостискання. На цей раз це вже TLS переговори. На цьому кроці визначається, який шифр буде використовуватися для шифрування з'єднання, засвідчується надійність сервера та встановлюється безпечне з'єднання. Цей крок також вимагає кілька додаткових повідомлень, якими повинні обмінятися сервер та браузер, перш ніж дані будуть надіслані.

І хоча забезпечення безпеки з'єднання знижує швидкість завантаження програми, безпечне з'єднання коштує своїх затрат, тому що в цьому випадку дані не можуть бути дешифровані третьою особою.

Після обміну вісьмома повідомленнями, браузер, нарешті, досягає всіх умов, щоб зробити запит.

#### 1.2.1.4 Відповідь на запит

Як тільки ми встановили з'єднання з веб-сервером, браузер відправляє запит HTTP GET від імені користувача. Найчастіше запитується HTML файл. У момент, коли сервер отримує запит, він починає відповідь із посилки заголовків відповіді та вмістом HTML-файлу.

Ця відповідь містить перший байт отриманих даних. Час до першого байта (Time to First Byte , TTFB) – це час між моментом, коли користувач відправив запит, скажімо, натиснувши на посилання, і моментом отримання першого пакету даних HTML. Перший пакет зазвичай містить 14КБ даних.

#### 1.2.1.5 TCP повільний старт / правило 14kb

Об'єм першого пакета даних – завжди 14KB. Це частина специфікації TCP slow start – алгоритму, що балансує швидкість з'єднання. Таке правило дозволяє поступово, при необхідності, збільшувати розміри даних, що передаються, поки не буде визначена максимальна ширина каналу.

У алгоритмі TCP slow start кожен наступний відправлений сервером пакет збільшується у розмірі вдвічі. Наприклад, розмір другого пакета буде близько 28КБ. Розмір пакетів буде збільшуватися до тих пір, поки не досягне якогось порогового значення або не впорається в проблему переповнення.

Якщо ви колись чули про правило 14КБ, повинні розуміти, що оптимізація продуктивності завантаження повинна враховувати обмеження цього початкового запиту. Повільний старт TCP дозволяє плавно прискорювати передачу даних так, щоб уникнути проблеми переповнення, коли багато даних очікують відправки, але не відправляються через обмеження ширини каналу.

#### 1.2.1.6 Контроль переповнення

Будь-яке з'єднання має обмеження, пов'язані з апаратною та мережевою системами. Якщо сервер відправить багато пакетів за раз – вони можуть бути відкинуті. Щоб уникнути таких проблем, браузер повинен реагувати на отримання пакетів і підтверджувати, що він отримує їх. Така відповідь-підтвердження називається Aknowledgements (ACK). Якщо через обмеження з'єднання браузер не отримає даних, то він не відправить підтвердження ACK. У цьому випадку, сервер зареєструє, що якісь пакет не дійшли і відправить їх заново, що призведе до зайвої роботи сервера і додаткового навантаження мережі.

#### 1.2.1.7 Парсинг

Як тільки браузер отримує перший шматочок даних, він відразу починає обробляти інформацію, що отримується. Ця обробка називається парсинг. Під час парсингу отримані дані перетворюються на DOM і CSSOM, які безпосередньо беруть участь у відображенні даних.

Навіть якщо відповідь на запит більше 14КБ, браузер все одно починає парсинг даних і намагається відобразити сторінку з даними, які вже доступні. Саме тому при оптимізації продуктивності дуже важливо включати в 14КБ, що ініціює відповідь всі необхідні для рендеру дані – так браузер зможе швидше почати формування сторінки. Однак, перш ніж будь-що з'явиться на екрані, HTML, CSS і JavaScript повинні бути оброблені.

#### 1.2.1.8 Побудова дерева об'єктної моделі документа

Перший крок – це обробка розмітки HTML та побудова дерева DOM. Обробка HTML включає токенізацію і побудову дерева. HTML-токени складаються з тегів старту та фінішу, а також атрибутів. Якщо документ сформований правильно, його обробка прямолінійна та швидка. Парсер перетворює вхідні токени на документ і будує дерево документа.

Об'єктна модель документа (DOM) визначає вміст документа. Елемент <html> – це перший тег та кореневий елемент дерева документа. Дерево відображає зв'язки та ієрархію між різними тегами. Теги, вкладені в інші теги, є дітьми. Чим більше існує вузлів у дереві, тим складніше це дерево збудувати.

Коли парсер знаходить неблокуючі ресурси (наприклад зображення), браузер відправляє запит на завантаження ресурсів, але сам продовжує обробку. Обробка може продовжуватися, коли виявлено посилання на CSS файл, але якщо виявлено <script>, особливо якщо він без параметрів async або defer – такий скрипт вважається блокуючим і зупиняє обробку HTML до завершення завантаження скрипта. Незважаючи на те, що сканер предзавантаження (про нього нижче) браузера може знаходити та вимагати такі скрипти заздалегідь, складні та об'ємні скрипти все ще можуть стати причиною помітних затримок завантаження сторінки.

#### 1.2.1.9 Сканер предзавантаження

Побудова дерева DOM займає весь процес. Так як це явно вузьке місце у продуктивності, було створено особливий сканер предзавантаження. Він обробляє доступний вміст документа та запитує високопріоритетні ресурси (CSS, JavaScript та

шрифти). Завдяки цьому сканеру нам не потрібно чекати, доки парсер дійде до конкретного місця, де викликається ресурс. Він запитує і отримує ці дані заздалегідь, у фоновому режимі, тому коли основний потік HTML-парсера доходить до запиту ресурсу, висока ймовірність, що ресурс вже запитаний або знаходиться в процесі завантаження. Оптимізації, які надає цей сканер, зменшують час блокування рендера.

### 1.2.2 Побудова моделі стилів CSSOM

Другий крок при проходженні критичного шляху рендерингу – це обробка CSS та побудова CSSOM дерева. Вони є незалежними структурами даних. Браузер перетворює CSS файли на карту стилів, яку він може зрозуміти і з якою може працювати. Браузер зчитує кожен набір правил у CSS, створює дерево вузлів з батьками, дітьми та сусідами, ґрунтуючись на CSS селекторах.

Як і в HTML, браузер повинен перетворити отримані правила CSS на щось, з чим він може працювати. Таким чином, весь цей процес – це повторення формування DOM лише для CSS.

CSSOM дерево включає стилі агента користувача – це стилі, які браузер вставляє за замовчуванням. Браузер починає побудову моделі з найбільш загальних правил кожного вузла, поступово застосовуючи більш специфічні правила. Інакше кажучи, він застосовує правила каскадно. Звідси і назва CSS – Cascading Style Sheets.

Побудова CSSOM відбувається дуже швидко і не відображається окремим кольором у засобах розробника. Воно настільки швидке, що найчастіше входить у показник "Повторне обчислення стилів (Recalculate Styles)" у засобах розробника. Цей показник показує загальний час обробки стилів – обробку CSS, побудова CSSOM та рекурсивне обчислення стилів. З погляду оптимізації продуктивності тут нічого робити, оскільки побудова CSSOM, загалом, займає навіть менше часу, ніж DNS запит.

#### 1.2.2.1 Компіляція JavaScript

Як CSS оброблений і CSSOM створено інші ресурси, наприклад, JavaScript-файли, продовжують завантажуватися. JavaScript після закінчення завантаження

повинен бути інтерпретований, скомпільований, оброблений та виконаний. Скрипти перетворюються на абстрактне синтаксичне дерево (AST). Деякі браузери беруть Abstract Syntax Tree і передають його в інтерпретатор, який перетворює дерево на байт-код. Байт-код виконується переважно потоці. Весь цей процес називається компіляцією.

#### 1.2.2.2 Побудова дерева доступності

Браузер також будує дерево доступності, яке використовується пристроями-помічниками для розуміння та інтерпретування контенту. Об'єктна модель доступності (accessibility object model, AOM) – це семантична версія DOM. Браузер оновлює AOM в той же час, коли оновлюється DOM. Водночас дерево доступності не може бути змінено допоміжними технологіями.

#### 1.2.2.3 Рендеринг

Етапи рендерингу включають стилізацію, компонування (layout), відмальовування (paint) і, в деяких випадках, композицію (composition). CSSOM і DOM дерева, створені на попередньому етапі, комбінуються в дерево рендера, яке потім використовується для розрахунку положення кожного видимого елемента. Після цього елементи відобразяться на екрані. У деяких випадках вміст може бути винесений на окремі шари і суміщений (composition) - такий підхід збільшує продуктивність, дозволяючи малювати вміст екрана на графічному процесорі замість ЦПУ. Це звільняє основний потік.

### 1.2.3 Створення дерева рендерингу

Третій крок у критичному шляху рендерингу – це комбінування DOM та CSSOM у дерево рендерингу. Конструювання цього дерева починається з проходу всього DOM-дерева від кореня, з виявленням кожного видимого вузла.

Елементи, які не повинні бути показані, не будуть включені в дерево рендера, так як вони не повинні бути відмальовані.

Кожен видимий вузол має правила з CSSOM. Дерево рендеру містить всі видимі вузли з їх вмістом та обчисленими стилями.

### 1.2.4 Компонування (Layout)

Четвертий крок на критичному шляху рендерингу – це запуск компонування (layout) елементів дерева рендеру. На цьому етапі обчислюється геометрія кожного вузла, тобто ширина, висота, положення елементів. Reflow (перекомпонування) – це будь-який подальший процес визначення розмірів та позиції для будь-якої з частин цілого документа.

Як тільки дерево рендеру побудовано – починається layout. Дерево несе в собі інформацію про те, які вузли мають бути відмальовані (навіть якщо вони невидимі), і які стилі мають бути застосовані, але в дереві немає жодної інформації про розміри та позиції елементів. Щоб визначити ці значення, браузер розпочинає обхід дерева.

На сторінці практично всі елементи прямокутні (box). Різні пристрої та налаштування мають на увазі незліченну кількість різних розмірів видимої області. На початковій фазі браузер, враховуючи розмір видимої області, визначає, які розміри різних елементів повинні бути на екрані. Використовує розмір видимої області як базис, процес починає обчислення з елемента body, потім переходить до його нащадків, обчислює розміри кожного елемента та резервує місце для тих елементів, розміри яких він ще не знає (наприклад, зображення).

Момент, коли позиція та розміри вузлів обчислені, називається layout. Наступні обчислення позицій та розмірів називаються reflow. У прикладі передбачуваний початковий layout відбувається перед тим, як зображення отримано. Оскільки ми не задавали розміру зображення, в момент отримання зображення відбудеться reflow.

### 1.2.5 Відмальовка (Paint)

Останній крок критичного шляху рендерингу – це відображення кожного окремого вузла на екрані. Момент, коли це відбувається вперше, називається first meaningful paint (перше значне відмальовування). Під час фази відтворення або ростеризації, браузер конвертує кожен контейнер box у справжні пікселі на екрані

(нагадаємо, що дані контейнерів формуються на етапі layout). Малювання передбачає малювання кожної візуальної частинки елемента на екрані (текст, кольори, межі, тіні) і малювання елементів, що замінюються (картинки, кнопки). Браузер повинен виконувати це швидко.

Щоб забезпечити плавну прокручування та анімацію, відображення кожного елемента займає весь основний потік. Сюди включається обчислення стилів, повторне обчислення стилів та малювання. Всі ці етапи мають виконуватися не довше 16.67 мс. (1000мс. / 60 кадрів за секунду). Для того, щоб зробити ініціюючий і повторний малювання швидше, можна розбити весь процес на кілька шарів. Коли це трапляється – стає необхідна композиція.

Малювання може розбити елементи в дереві візерунок на шари. Для того, щоб прискорити їхній рендер, браузер може перенести малювання різних шарів на GPU (замість основного потоку CPU). Вузли, створені таким чином, будуть відмальовані на їхньому власному шарі, разом з їх нащадками, якщо тільки нащадки самі по собі не будуть винесені в окремі шари.

Шари покращують продуктивність. Але, з погляду управління пам'яті, вони неефективні. Тому намагайтеся не використовувати їх там, де немає необхідності.

#### 1.2.5.1 Композиція (Compositing)

Коли розділи документа відображені на різних шарах, а один шар знаходиться над іншим або перекриває його, стає потрібна композиція. Цей крок дозволяє браузеру гарантувати, що кожен шар відображений на екрані в правильному порядку, а вміст коректно відображається.

При довантаженні раніше запрошених ресурсів (наприклад, зображень) може знадобитися перерахувати розміри та положення елементів щодо один одного. Цей перерахунок – reflow – запускає перемальовування (repaint) та перекомпозицію (re-composite). Якщо ми заздалегідь визначили розмір зображення, перерахунок не буде необхідний і в цьому випадку лише той шар, який має бути перемальований – буде перемальовано. Але якщо ми не визначили розмір зображення заздалегідь, то браузер

після отримання відповіді від сервера буде змушений відмотати процес рендерингу назад до кроку компонування (layout) і почати процес відтворення ще раз.

#### 1.2.5.2 Інтерактивність

Можна було б подумати, що як тільки основний потік завершує малювання сторінки – "все готово". Це не завжди так. Якщо серед завантажуваних ресурсів є JavaScript, завантаження якого було коректо відкладено, а запуск якого відбувається тільки після події onload, основний потік починає обробку скриптів. Під час цієї обробки браузер не може обробляти події прокручування, натискання та ін.

Time to Interactive (TTI, час до інтерактивності) – це показник того, як багато часу проходить між першим мережевим запитом і моментом, коли сторінка стає інтерактивною. У хронології цей етап слідує відразу за First Contentful Paint. Інтерактивністю називається показник того, що сторінка відреагувала на дію користувача за 50мс. Якщо процесор зайнятий обробкою, компіляцією та виконанням JavaScript, то браузер не може відреагувати досить швидко, а значить сторінка вважається не інтерактивною.

## 1.3 SEO-оптимізація сайту для пошукових роботів та кінцевого споживача

SEO-оптимізація сайту – комплекс заходів, спрямованих на підвищення його позицій у видачі пошукових систем або інших систем за спеціально підібраними запитами [4]. Їхня мета – збільшення відвідувань веб-сторінок і, відповідно, конверсійних дій.

Сайти малого та середнього бізнесу отримують 85% відсотків трафіку з пошукової видачі. Чим вище позиції веб-ресурсу в пошукових системах, тим більше людей на нього заходять. Без SEO-просування навіть найякісніший сайт буде марним: якщо пошукові системи не бачать контенту, його існування не має сенсу.

Навіщо потрібно SEO-просування та чому їм потрібно займатися? Пошукова оптимізація – одна з найефективніших у наш час маркетингових інвестицій, яка працює довго та дає високу віддачу.

Основні її переваги:

- запуск вирви продажів. Для того щоб отримати конверсію, необхідно розповісти про себе потенційному клієнту. У сайта на перших позиціях шансів залучити відвідувача набагато більше, ніж у тих, хто перебуває нижче;
- зацікавлена аудиторія. Якщо людина заходить на сайт із пошукового питання, зрозуміло, що його цікавить певний продукт і він перебуває на стадії вибору;
- довіра аудиторії. Успішні компанії знаходятся в топі по видачі, користувачі думають що це один із найкращих варіантів, та обирають саме його;
- економія реклами. Безумовно, SEO вимагає фінансових вкладень, проте у довгостроковій перспективі він вигідніший, ніж пошукова реклама.

SEO передбачає зовнішню та внутрішню оптимізацію. Зовнішнє просування здійснюється за рахунок нарощування маси посилань, розміщення посилань на сторонніх ресурсах, на форумах, в соцмережах.

Внутрішнє просування вимагає кропіткої роботи з цілого ряду напрямів:

- текстове наповнення та ілюстрації мають бути унікальними. Текст обов'язково оптимізовано з урахуванням ключових фраз. Крім того, необхідно дотримуватись вимог щодо грамотності та читабельності;
- зручний інтерфейс та відсутність повторів сторінок;
- оптимізація html коду сторінок. Обов'язково прописуються мета-теги: заголовки h1-h6, заголовки вкладки, опис сторінки;
- технічні налаштування: sitemap.xml, robots.txt, індексація, формат URL.

Які є фактори ранжирування сторінок у пошукових системах:

- рік вашого домену. Чим старший його вік, тим краще просування;
- історія домену сайту – добре, коли його тематика та власники не змінюються;
- назву домену доцільно вибирати відповідно до ключового запиту, який характеризує тематику вашого ресурсу. Так вас простіше та швидше знайде цільова аудиторія;

- фактори ранжирування, що залежать від контентної частини вашого ресурсу,
- короткі та точні заголовки (заголовок вкладки, h1, h2, h3…);
- пам'ятайте, добре ранжуються сторінки з текстом понад 2000 символів;
- краще більше наповнювати текст ключовими ключами на самому початку матеріалу;
- використовуйте в текстовому контенті марковані чи номерні списки, таблиці – це спростить розуміння роботам;
- розміщуйте унікальний текстовий контент і такі самі зображення;
- частіше оновлюйте свій ресурс – так і сторінки будуть індексуватися частіше;
- оптимізуйте швидкість завантаження вашого ресурсу.

Які переваги несе SEO оптимізація для користувачів? При пошуку товару або послуги, буду видаватися більш релевантні пошукові видачі, тобто користувач буде отримувати саме те, що він шукав. Він може витрачати менше часу на пошук якісного продукту, за кошт того, що вкаже точні ключові слова. Як наслідок виграють всі – користувач який отримав продукт, і бізнес, який його реалізував.

### 1.4 Основні відомості про типи рендерингу даних

Якщо дивитися на початок історії Інтернету, то єдиним способом відобразити ваш сайт на екрані, це був варіант з розташуванням статичних файлів на сервері. Насамперед вам потрібно було просто завантажити файли додатку на сервер, і потім сервер віддавав би ці файли за звичайними маршрутами.

Але на сьогоднішній день веб-розробка стала складнішою, і з’явилось купа методів доставити додаток до користувача. Можливо виділити три основні види – рендеринг на стороні сервера (SSR), рендеринг на стороні клієнта (CSR) і статичну генерацію сайтів (SSG).Варто відзначити більш нові методи рендерингу даних – Universal Rendering і Pre-Rendering.

Виходьте з того, яких параметрів розробник хоче досягти, можливо використовувати будь який з цих типів рендерингу. Але кожний із них має свої сильни і слабкі сторони.

### 1.4.1 Рендеринг на стороні клієнта – CSR

У разі, коли користувач відкриває веб-сторінку, його браузер робить запит на сервер, після чого сервер починає повертати нам дані – зазвичай це звичайний HTML документ невеликого розміру. Далі браузер починає завантажувати весь JavaScript, який був підключейний в цьому документі. Третій крок – браузер починає запускати JavaScript код, який буде відображати дані на нашу сторінку. Поки виконуються ці три пункти, користувач бачитиме білий екран або індикатор завантаження.

Плюси:

- швидке відображення нових даних, після першого завантаження сторінки;
- швидка навігація між сторінками;
- менше навантаження на сервер;
- висока інтерактивність додатку с користувачем.

Основна перевага в тому, що коли у користувача завантажилися і відобразилися всі скрипти, користувачу в основному не потрібно більше нічого завантажувати, наприклад, для того щоб перейти на наступну сторінку. Тобто навігація по додатку буде працювати набагато швидше, аніж при серверному рендерингу.

Що стосується мінусів CSR:

- повільне перше завантаження. "Повільне" – відносний опис, все залежить від потужності пристрою та швидкості мережі користувача, але іноді і 2-3 секунди білого екрана вже для нього критичні;
- рішення з маршрутизацією по додатку на стороні клієнта може відкласти роботу пошукового робота. Парсерам буде складніше зчитувати вашу сторінку, так як їм потрібно буде запускати скрипти, але не всі пошукові системи мають здатність навіть запускати JavaScript, щоб проіндексувати вашу сторінку;

- запит при ініціалізації завантажує одразу всі сторінки, стилі, шаблони і так далі, сильно навантажуючи цим техніку користувача.

### 1.4.2 Рендеринг на стороні серверу – SSR

Процес рендерингу на сервері кардинально відрізняється від рендерингу на клієнті. Перший крок буде аналогічним до попереднього методу — браузер робить запит на сервер, але після цього сервер відправляє нам вже готову до відображення HTML-сторінку. Тобто вже є всі мета-теги, всі блоки, які нам потрібні — все вирушає до браузера, де сторінка рендериться і стає видимою для користувача. Коли користувач побачив якийсь базовий контент, браузер починає завантажувати JavaScript, після чого виконує стрипти, і далі користувач може взаємодіяти зі сторінкою.

Плюси SSR:

- SSR гарантує відмінну індексацію сторінок і читання пошуковими механізмами, це відбуватиметься набагато частіше, тому що. пошуковим механізмам не потрібно буде запускати JS, він просто зчитуватиме вашу сторінку з мета-тегами;
- поліпшення продуктивності для користувача - користувач бачитиме контент на сторінці швидше, так як від сервера прийде відповідь у вигляді HTML і нам доведеться лише відмалювати його у браузері;
- оптимізація під соціальні мережі – коли користувач буде ділитися такою сторінкою в соціальних мережах, наприклад, у Facebook або Twitter, вони відображатимуться як гарні та стилізовані блоки інформації з назвою, картинкою та описом(здійснюється це за допомогою правильно підібраних мета-тегів);
- девайс користувача буде менш навантажений – оскільки нам вже надходить відповідь від сервера і нам потрібно буде лише вивести цю інформацію, нічого не потрібно буде рендерити на стороні клієнта, браузері чи пристрої користувача.

Мінуси SSR:

- TTFB буде повільнішим. Це одна з метрик продуктивності веб-сторінки, яка описує час, який минув з моменту відправлення браузером запиту на сторінку до отримання першого байта інформації від сервера. Замість того, щоб відправити практично порожній HTML документ із посиланнями на JavaScript (як у випадку з відмальовуванням на стороні клієнта), серверу потрібно буде якийсь час для підготовки HTML для вашої сторінки;
- сервер буде більш зайнятий і оброблятиме менше запитів за секунду. Коли ми додаємо відображення контенту на сторону сервера, пропускна здатність сервера зменшиться;
- HTML-документ буде більшим за розміром, так як в ньому вже буде повністю вся структура додатку;
- незважаючи на те, що сторінка завантажується швидше, вона буде непридатна для використання (користуватися нею можна буде тільки після компіляції JavaScript).

### 1.4.3 Статична генерація сайтів – SSG

Статична генерація сайтів – це новий підхід для веб-розробки, який дозволяє створювати веб-сайти, сторінки яких генеруються в статичні файли в момент складання. Тим самим, коли браузер робить запит, сервер не генерує розмітку, як у випадку SSR , а віддає вже готову.

Різниця між SSR та SSG в тому, що замість використання HTML і CSS тепер ми використовуємо сучасні інструменти, такі як React, Vue і Angular . Тобто додаток, написаний на сучасних інструментах, перетворюється на HTML, CSS і JavaScript за допомогою транспіляторів, збирачів пакетів і так далі.

Проблема SSG закладається в тому, що він не передбачає динамічну роботу з сервером, тобто. ви не можете створити, наприклад, кошик інтернет-магазину за допомогою SSG , так як інформація, що відображається в кошику, повинна бути унікальна для кожного користувача. У такому випадку ми можемо застосувати лише SSR або CSR. Статична генерація сайтів зручна коли сторінка ідентична для кожного користувача.

Плюси:

- як і у випадку з SSR , пошуковий робот отримує всю необхідну інформацію і може правильно індексувати сторінку;
- продуктивність. SSG має найбільшу продуктивність проти SSR і CSR , так як всі дані вже згенеровані під час складання;
- використовуючи статичну генерацію сайтів, зменшуються затрати на серверну частину, так як додаток збирається один раз;
- у статичних сайтів немає сервера, а значить у зловмисників немає можливості отримати доступ до вашої бази даних або панелі адміністратора;

Мінуси:

- немає панелі адміністратора. Плюс, як відсутність вразливостей, але мінус, тому що всі зміни на сайті вам доведеться робити у редакторі коду, і оновлювати вручну;
- відображення лише статичних даних. Ви не можете відобразити динамічні дані за допомогою SSG, відповідно у вас немає можливості зробити складний веб-додаток.

Який тип рендерингу обрати? Все залежить від бізнес вимог, які стоять перед проектом. CSR чудово підходить для різного плану адмін панелей, де не важлива SEO оптимізація. SSR чудово підходить для проектів, де наприклад потрібно постійно генерувати трафік і збільшувати пошукові видачі. SSG в свою чергу чудово підходить для проектів, де не часто оновлюється контент, наприклад лендінги, сайти для документації і так далі.

### 1.5 Висновки по розділу

У ході виконання кваліфікаційної роботи потрібно довести і обґрунтувати гіпотезу, що за допомогою протоколу WebSocket можливо організувати якісне та швидке з'єднання для передачі відрисованих даних на сервері через клієнт серверну архітектуру. Такий підхід може зібрати в собі переваги різних типів рендерингу як SSR та CSR, але без особливо навантаження на клієнта.

Для доведення, буде використаний кількісний підхід до дослідження. Кількісне дослідження використовує цифри та вимірювання, щоб зв’язати емпіричні спостереження з потенційною неупередженою відповіддю.

Прогнози та висновки цих досліджень обмежені невеликою кількістю випадків загального користування. Для виконання вимірювань можна уявити нескінченну кількість ситуацій, але не буде доречним представляти всі ці альтернативи, тому використовуються більш-менш реальні ситуації використання частини функціоналу веб додатків

# 2 ТЕОРЕТИЧНИЙ РОЗДІЛ

## 2.1 Інструменти для створення клієнтського середовища

На сьогоднішній день існує 3 головних мови для розробки веб додатків. Це HTML, CSS та JS. Існує ще 4 інструмент, який набирає стрімку популярність серед розробників – це WebAssembly (WASM). WebAssembly — незалежний від браузера універсальний низькорівневий проміжний код для виконання в браузері застосунків, скомпільованих з різних мов програмування. Але ця технологія є занадто нова, тому ми не будемо торкатися її в рамках даної кваліфікаційної роботи.

Розробка будь якого веб-додатку починається з розробки HTML шаблону. На цьому кроці немає якихось особливих інструментів для прискорення або оптимізації розробки. Насамперед необхідно вивчити семантичну верстку, SEO оптимізацію а також інструменти для забезпечення доступності контенту людям з обмеженими можливостями. С цими знаннями можливо створити ідеальну сторінку, яка буде добре індексуватися пошуковими роботами, швидко завантажуватися і гарно виглядати.

### 2.1.1 Розмітка

HTML — це технологія, що не має альтернатив, так як браузери вміють відображати тільки інтерфейси, які засновані на HTML. При цьому є кілька підходів до створення гіпертекстових документів.

У найпростішому випадку сторінки формуються на сервері та відправляються до браузера. Цей спосіб можна назвати класичним і він працює навіть у давніх браузерах. Але не варто чекати високої інтерактивності та швидкодії від такого підходу.

Більш просунуті підходи засновані на тому, що використовувана HTML-розмітка частково або повністю формується на стороні браузера за допомогою JavaScript на базі шаблонів і ґрунтується на даних, отриманих з сервера. І тут можна досягти високої швидкості переходів між сторінками, і зробити інтерфейс дуже інтерактивним. Найпростіший випадок – це коли розмітка повністю формується за

допомогою JS, а спілкування з сервером обмежено лише передачею даних (найчастіше в JSON-форматі).

### 2.1.2 Стилізація

CSS – мова опису зовнішнього вигляду документа, написаного за допомогою мови розмітки. Зазвичай використовується для опису оформлення веб-сторінок, написаних за допомогою HTML-розмітки.

CSS використовується для завдання шрифтів і кольорів, розташування окремих елементів та інших параметрів зовнішнього вигляду сторінок. CSS відокремлює опис зовнішнього вигляду від логічної структури веб-сторінки, що реалізується за допомогою HTML. Цей поділ збільшує доступність документа та надає велику гнучкість управління поданням.

CSS3 – найсвіжіша версія стандарту, яка привнесла багато нововведень у розробку веб-інтерфейсів. Також для CSS існує ряд препроцесорів, які спрощують розробку візуального формування, додаючи в мову опис оформлення можливості мов програмування (змінні, функції, міксини та інші можливості). Часто використовувані препроцесори – SASS (SCSS), LESS, stylus і PostCSS. Розглянемо всі ці інструменти на прикладі SASS (SCSS). Інші розглядати немає сенсу, так як вони дуже схожі між собою.

SASS (SCSS) це препроцесори CSS. Вони дозволяють використовувати у створенні CSS-коду вкладеність правил та можливості мов програмування – змінні, цикли та функції. Це полегшує розробку масштабних проектів та підтримку цілісності правил усередині великого набору стилів. Ці інструменти повністю сумісні з усіма версіями CSS, тому у таких проектах можна використовувати будь-які доступні бібліотеки CSS.

Що є у SASS/SCSS, чого немає у стандартному CSS?

Вкладені правила: можна вкладати CSS властивості, в кілька наборів дужок {}. Це зробить ваш CSS чистішим і зрозумілішим.

Змінні: у стандартному CSS теж є змінні, але змінні SASS значно потужніший інструмент. Наприклад, можна використовувати змінні в циклах і генерувати

значення властивостей динамічно. Також можна впроваджувати змінні в імена властивостей, наприклад: property-name-N { … }.

Найкраща реалізація операторів: ви можете підсумовувати, віднімати, ділити та множити CSS значення. SASS реалізація більш інтуїтивна, ніж стандартний функціонал CSS calc().

Функції: Sass дозволяє багаторазово використовувати CSS стилі як функції.

Тригонометрія: крім базових операцій (+, -, *, /), SCSS дозволяє писати власні функції. Наприклад, функції sin та cos можна написати, використовуючи лише синтаксис SASS/SCSS. Звичайно, вам знадобляться знання тригонометрії. Такі функції можуть знадобитися для створення анімації.

Зручний робочий процес: ви можете писати CSS, використовуючи конструкції, знайомі з інших мов: for-цикли, while-цикли, if-else. Але майте на увазі, це лише препроцесор, а не повноцінна мова, Sass контролює генерацію властивостей та значень, а на виході ви отримуєте стандартний CSS.

Міксини: дозволяють один раз створити набір правил, щоб потім їх багаторазово або змішувати з іншими правилами. Наприклад, міксини використовують для створення окремих тем макету.

### 2.1.3 Скрипти

JavaScript – мультипарадигменна мова програмування, підтримує об'єктно-орієнтований, імперативний та функціональний стилі.

Найбільш широко JS використовується в браузерах як мова сценаріїв для надання інтерактивності веб-сторінок, але крім цього він може використовуватися для бекенд-розробки та інших завдань.

Основні архітектурні риси:

- слабка динамічна типізація;
- автоматичне керування пам'яттю;
- ООП основане на прототипах;
- функції як об'єкти першого класу.

У мові відсутні стандартна бібліотека, стандартні інтерфейси до веб-серверів та баз даних, а також немає вбудованої системи керування пакетами. Але все це вирішується підключенням сторонніх бібліотек та використанням додаткового ПЗ.

Фреймворки JS – це бібліотеки програмування JavaScript, в яких є попередньо написаний код для використання у стандартних функціях та завданнях програмування. Це основа для створення веб-сайтів або веб-застосунків навколо.

Почнемо з того, навіщо нам потрібні фреймворки JavaScript? Кодування цілком можливе без їх використання, але правильно підібране середовище може значно полегшити роботу. Більш того, вони безкоштовні і з відкритим вихідним кодом, тому немає ризику.

Насамперед це підвищить вашу продуктивність. Розглядайте це як свого роду обхідний шлях: вам доведеться писати менше за код вручну, тому що вже є заздалегідь написані і готові до використання функції та шаблони. Деякі компоненти веб-сайту не повинні бути виготовлені за індивідуальним замовленням, тому ви можете створювати та розширювати заздалегідь створені компоненти. Фреймворки більш адаптовані для дизайну веб-сайтів, і більшість розробників сайтів віддають перевагу їм.

#### 2.1.3.1 React

В даний час лідером в галузі інфраструктури JavaScript UI є React [5]. Спочатку розробники Facebook почали працювати над цим, щоби спростити свою роботу. Програма під назвою Facebook Ads зростала дуже швидко, що означало складне управління та підтримку. В результаті команда почала створювати структуру, яка допоможе їм ефективно. У них був ранній прототип до 2011 року, а через два роки, структура була з відкритим вихідним кодом і доступна для громадськості. В даний час його використовують багато бізнес-гіганти: AirBNB, PayPal, Netflix і т.д.

React базується на компонентному підході, що сприяє до перевмкористання коду. Це частини коду, які можна класифікувати як класи чи функції. Компонент представляє собою якусь частину додатку. Параметри, що використовуються компонентами для відображення, називаються властивостями.

React використовує JSX, синтаксис XML, який поєднує JavaScript і HTML. Це не шаблон JavaScript; це повний JavaScript. Спочатку деякі нові розробники можуть знайти JSX трохи заплутаним. Однак, попрацювавши з ним якийсь час, ви зрозумієте, наскільки це корисно. Це бібліотека, на яку ви повинні звернути увагу, якщо ви займаєтесь розробкою веб-інтерфейсів.

### 2.1.3.2 Angular

Angular – одне з найпотужніших середовищ JavaScript [6]. Google використовує цю платформу для розробки односторінкової програми (SPA). Це середовище розробки відоме насамперед тому, що воно надає розробникам найкращі умови для об'єднання JavaScript з HTML та CSS. Понад півмільйона сайтів, таких як google.com, youtube.com тощо, використовують Angular.

Це також переважне середовище інтерфейсу JavaScript для розробників додатків Google. Angular має компонентну структуру, як і React. Ви можете маніпулювати, вкладати та використовувати їх у міру потреби. Вам потрібно буде використовувати TypeScript, щоб написати програму в Angular. Це розширений набір JavaScript, який використовує той же синтаксис, але також підтримує статичну типізацію та класи. У TypeScript ви отримуєте модифікатори доступу, перерахування, узагальнення, гібридні типи та багато іншого. Простіше кажучи, Angular це фантастична платформа, на яку можна поглянути, якщо ви новий розробник.

### 2.1.2.3 Vue.js

Vue це JavaScript-фреймворк з відкритим вихідним кодом для створення креативного інтерфейсу [7]. Інтеграція з Vue у проектах, які використовують інші бібліотеки JavaScript, спрощена, оскільки вона розроблена для адаптації. Більше 36 000 веб-сайтів зараз використовують Vue. Такі компанії, як Stackoverflow, PlayStation і т.д., покладаються на Vue для своїх сайтів інтерфейсу користувача.

Vue.js також досить простий у освоєнні: все, що потрібно, це JavaScript і HTML. Іншою сильною стороною Vue.js є його інтерфейс командного рядка (CLI). Це базовий інструмент, який прискорює розробку, пропонуючи безліч плагінів, пресетів,

миттєвого прототипування та інтерактивного інструменту розробки проектів. Деякі з його функцій включають компоненти, шаблони, переходи та двостороннє зв'язування даних, а також фокус реактивності. Реактивність виникає при зміні або оновленні будь-якого з об'єктів JavaScript у Vue. Vue.js використовує те, що називається Shadow DOM, що робить рендеринг сторінки швидким.

Це основні фреймворки, які використовують для розробки клієнтської частини. Майже для кожного з них, є альтернатива яка використовує підхід SSR або SSG. Наприклад для React це Next.js і Gatsby.

В результаті, для створення додатку були обрані стандартні технології у вигляді – HTML, CSS, JS. Такий вибір був обумовлений тим, що додаток не буде мати непотрібних залежностей, і стане можливим досягти максимальної швидкості роботи додатку.

### 2.2 Інструменти для створення серверного середовища

У наші дні платформа Node.js є однією з найпопулярніших платформ для побудови ефективних та масштабованих REST API's [8]. Вона також підходить для побудови гібридних мобільних додатків, десктопних програм і навіть для IoT.
Node.js це платформа для виконання коду на стороні серверу. Node.js працює на основі движка V8, який вміє компілювати JavaScript код в машинний код.

Node.js використовує подійно-орієнтовану модель та неблокуючу архітектуру введення/виведення, що робить його легковажним та ефективним.

Node.js використовує операції, що не блокують введення/виведення, що ж це означає:

- головний потік не блокуватиметься операціями введення/виводу;
- сервер продовжуватиме обслуговувати запити;

Коли користувач відправляє запит на сервер, найчастіше той в свою чергу може відповісти або даними, або відправивши файл. Щоб надіслати HTML сторінку спочатку її потрібно прочитати з файлу. Для цього використовується нативний модуль fs для читання файлу.

Функції, які входять до http.createServer і fs.readFile як аргументи, є зворотними викликами . Ці функції будуть виконані в одному з моментів у майбутньому (Перша, як тільки сервер отримає запит, а друга — коли файл буде прочитаний з диска і поміщений у буфер).

Поки файл зчитується з диска, Node.js може обробляти інші запити і навіть зчитувати файл знову і все це в одному потоці. Це все можливо завдяки циклу подій.

Цикл подій

Цикл подій це «функція», яка виконується всередині Node.js. Це буквально нескінченний цикл, і насправді він працює в одному потоці. Libuv – С бібліотека, яка реалізує цей патерн і є частиною ядра Node.js. Цикл подій має 6 фаз, кожне виконання всіх 6 фаз називають tick-ом.

Фази цикла подій:

- timers – у цій фазі виконуються коллбеки, заплановані методами setTimeout()та setInterval();
- pending callbacks – виконуються майже всі коллбеки, за винятком подій close, таймерів та setImmediate();
- idle, prepare – використовується лише для внутрішніх цілей;
- poll – відповідальний отримання нових подій вводу/вывода. Node.js може блокуватись на цьому етапі;
- check – коллбеки, викликані методом setImmediate(), виконуються на цьому етапі;
- close callbacks – наприклад, socket.on('close', ...).

Є тільки один потік, і цей потік є циклом подій, але тоді хто виконує всі операції введення/виводу?

Коли циклу подій потрібно виконати операцію введення/виводу він використовує потік ОС з тредпулу (thread pool), а коли завдання виконано, коллбек ставиться в чергу під час фази pending callbacks .

### 2.1.1 Express.js

Express.js – це безкоштовна платформа веб-додатків з відкритим вихідним кодом написаним на Node.js. Він використовується для швидкого та легкого проектування та створення веб-додатків на стороні серверу.

Оскільки для Express.js потрібен лише javascript, програмістам і розробникам стає легше створювати веб-додатки та API без будь-яких зусиль. Express.js – це фреймворк Node.js, що означає, що більшість коду вже написана для роботи програмістами. Ви можете створювати веб-додадки різного плану за допомогою Express.js. Express.js є легким і допомагає організувати веб-додатки на стороні сервера в більш організовану архітектуру MVC.

Важливо вивчити JavaScript і HTML, щоб мати можливість використовувати Express.js. Express.js полегшує керування веб-додатками. Частіше за все Express.js є внутрішньою частиною MEAN стеку і керує маршрутизацією, сеансами, HTTP-запитами, обробкою помилок тощо. Цей фрейворк покращує функціональність node.js. Насправді, якщо ви його не використовуєте, вам доведеться написати дуже багато різноманітного коду, щоб створити ефективний API. С приходом Express.js програмування на node.js і надало багато додаткових функцій, які розробник може використовувати при написанні коду.

Express.js підтримує JavaScript, який є широко використовуваною мовою, її дуже легко вивчити і вона виконується майже всюди. Тому, якщо ви вже її знаєте, то вам буде дуже легко програмувати за допомогою Express.js. Використання цього фреймворку значно пришвидшує роботу розробника, і дає змогу в значно коротший час виконувати бізнес задачі. Він влаштовую просту маршрутизацію для запитів, зроблених клієнтами, а також забезпечує проміжне програмне забезпечення, яке відповідає за прийняття рішень щодо надання правильних відповідей на запити, зроблені клієнтом. Без Express.js вам доведеться написати свій варіант маршрутизації на сервері, що є достатньо тяжким завданням майже для будь-якого розробника.

## 2.3 Протоколи передачі даних для клієнт-серверного веб застосунку

### 2.3.1 HTTP протокол

Протокол HTTP є основою обміну даними в Інтернеті. HTTP є протоколом клієнт-серверної взаємодії, що означає ініціювання запитів до сервера самим одержувачем, як правило, веб-браузером.

В результаті ми отримуємо документ, який може складатися з різних вкладених документів, наприклад: зображень, відео-файлів, скриптів, стилів та багато іншого.

Клієнти та сервери взаємодіють, обмінюючись поодиночними повідомленнями. Повідомлення, надіслані клієнтом, називаються запитами, а повідомлення, надіслані сервером, називаються відповідями.

HTTP є протоколом прикладного рівня, який найчастіше використовує можливості іншого протоколу – TCP (або TLS – захищений TCP) – для пересилання своїх повідомлень, проте будь-який інший надійний транспортний протокол теоретично може бути використаний для доставки таких повідомлень. Завдяки своїй розповсюдженості він використовується не тільки для отримання клієнтом гіпертекстових документів, зображень та відео, але й для передачі вмісту серверам, наприклад, за допомогою HTML-форм. HTTP також може бути використаний для отримання лише частин документа для оновлення веб-сторінки на запит (наприклад, за допомогою AJAX запиту).

Найчастіше в якості учасника виступає веб-браузер, але ним може бути будь-хто, наприклад, робот, що подорожує по мережі для поповнення та оновлення даних індексації веб-сторінок для пошукових систем.

Кожен запит відправляється серверу, який обробляє його і повертає відповідь. Між цими запитами і відповідями зазвичай існують чисельні посередники, які називаються проксі серверами. Вони виконують різноманітні операції, наприклад, можуть працювати як шлюз або кеш.

Завдяки тому, що мережа побудована на основі системи рівнів взаємодії, ці посередники знаходятся на мережному та транспортному рівнях. У цій системі рівнів HTTP займає найвищий рівень, який називається "прикладним".

Учасник обміну – це будь-який інструмент чи пристрій, що діють від імені користувача. Це завдання переважно виконує веб-браузер; в деяких випадках учасниками виступають програми, які використовуються інженерами та веб-розробниками для налагодження своїх програм.

Браузер завжди є тією сутністю, що створює запит. Сервер зазвичай цього не робить, хоча за багато років існування мережі були придумані методи, які можуть дозволити запити з боку сервера.

Щоб відобразити веб-сторінку, браузер надсилає вихідний запит для отримання HTML-документа цієї сторінки. Після цього браузер вивчає цей документ і запитує додаткові файли, необхідні для відображення змісту веб-сторінки (скрипти, інформацію про макет сторінки – CSS таблиці стилів, додаткові ресурси у вигляді зображень та відео-файлів), які є частиною вихідного документа, але розташовані у інших місцях мережі. Далі браузер з'єднує всі ці ресурси для відображення їх користувача як єдиного документа – веб-сторінки. Скрипти, що виконуються самим браузером, можуть отримати додаткові ресурси по мережі на наступних етапах обробки веб-сторінки, і браузер відповідним чином оновлює відображення цієї сторінки для користувача.

З іншого боку комунікаційного каналу розташований сервер, який обслуговує користувача, надаючи йому документи на запит. З точки зору кінцевого користувача, сервер завжди є якоюсь однією віртуальною машиною, що повністю або частично генерує документ, хоча фактично він може бути групою серверів, між якими балансується навантаження, тобто перерозподіляються запити різних користувачів, або складним програмним забезпеченням, що опитує інші комп'ютери (такі як сервери кешу, сервери баз даних та інші).

#### 2.3.1.1 Версії протоколу HTTP

Існує кілька версій протоколу HTTP. Перша експериментальна версія HTTP 0.9 була розроблена в ЦЕРН в 1991 році. Перша офіційна версія HTTP 1.0 була прийнята як стандарт у 1996 році і майже відразу ж після цього у 1997 році була прийнята розширена версія протоколу HTTP 1.1. Саме ця версія використовується досі. У 2015

році з'явилася нова версія протоколу HTTP 2, зараз ця версія тільки вводиться в експлуатацію, вона підтримується ще не всіма браузерами і не всіма веб-серверами.

#### 2.3.1.2 Структура протоколу

Структура протоколу визначає, що кожне HTTP-повідомлення складається з трьох частини, які передаються у такому порядку:

- стартовий рядок – визначає тип повідомлення;
- заголовки – характеризують тіло повідомлення, параметри передачі та інші відомості;
- тіло повідомлення – безпосередньо дані повідомлення.

#### 2.3.1.3 Стартовий рядок HTTP

Стартовий рядок є обов'язковим елементом, оскільки вказує на тип запиту/відповіді, заголовки та тіло повідомлення можуть бути відсутніми.

Стартові рядки розрізняються для запиту та відповіді. Рядок запиту виглядає так: метод URI HTTP/Версія протоколу.

Стартовий рядок відповіді сервера має такий формат:

- HTTP/Версія Код Стану;
- методи протоколу;
- метод HTTP (англ. HTTP Method) – послідовність з будь-яких символів, крім керуючих та роздільників, що вказує на основну операцію над ресурсом. В таблиці 2.1 вказані основні методи які використовуються.

Таблиця 2.1 - Методи протоколу HTTP

| Метод | Короткий опис |
|---|---|
| OPTIONS | Використовується для визначення можливостей веб-сервера або настройок з'єднання для конкретного ресурсу. |
| GET | Використовується для запиту вмісту вказаного ресурсу. |

| | |
|---|---|
| HEAD | Аналогічний метод GET, за винятком того, що у відповіді сервера відсутнє тіло. |
| POST | Застосовується передачі даних даних заданому ресурсу. |
| PUT | Використовується для завантаження вмісту запиту на вказаний у запиті URI. |
| PATCH | Аналогічно PUT, але застосовується лише фрагменту ресурсу. |
| DELETE | Видаляє цей ресурс. |
| TRACE | Повертає отриманий запит так, що клієнт може побачити, що проміжні сервери додають або змінюють запит. |
| LINK | Встановлює зв'язок зазначеного ресурсу коїться з іншими. |
| UNLINK | Забирає зв'язок зазначеного ресурсу з іншими. |

Кожен сервер повинен підтримувати як мінімум способи GET і HEAD. Якщо сервер не розпізнав вказаний клієнтом метод, він повинен повернути статус 501 (Not Implemented).

#### 2.3.1.4 Коди стану

Код стану інформує клієнта про результати виконання запиту та визначає його подальшу поведінку. Набір кодів стану є стандартом і всі вони описані у відповідних документах RFC.

Кожен код є цілим тризначним числом. Перша цифра свідчить про клас стану, наступні – порядковий номер стану (рисунок 2.1). За кодом відповіді зазвичай слідує короткий опис англійською мовою.


Рисунок 2.1 – Структура коду стану HTTP

Введення нових кодів має здійснюватися лише після погодження з IETF. Клієнт може знати всі коди стану, але він зобов'язаний відреагувати відповідно до класу коду.

Класи кодів стану, що застосовуються в даний час, наведені в таблиці 2.2.

Таблиця 2.2 – Коди стану протоколу HTTP

| Клас кодів | Короткий опис |
|---|---|
| 1xx Informational (Інформаційний) | У цей клас виділено коди, що інформують про процес передачі. |
| 2xx Success (Успішно) | Повідомлення даного класу інформують про випадки успішного прийняття та обробки запиту клієнта. |
| 3xx Redirection (Перенаправлення) | Коди статусу класу 3xx повідомляють клієнту, що для успішного виконання операції необхідно зробити наступний запит до іншого URI. |
| 4xx Client Error (Помилка клієнта) | Клас кодів 4xx призначений для вказівки помилок клієнта. |
| 5xx Server Error (Помилка сервера) | Коди 5xx виділені у разі невдалого виконання операції з вини сервера. |

#### 2.3.1.5 Заголовки HTTP

Заголовок HTTP – це рядок у HTTP-повідомленні, що містить розділену двокрапкою пару виду «параметр-значення». Формат заголовка відповідає загальному формату заголовків текстових повідомлень. Як правило, браузер і веб-сервер включають повідомлення більше ніж по одному заголовку. Заголовки повинні надсилатися раніше тіла повідомлення і відокремлюватися від нього хоча б одним порожнім рядком.

Назва параметра повинна складатися мінімум з одного друкованого символу. Після назви відразу повинен слідувати символ двокрапки. Значення може містити будь-які символи, крім перекладу рядка та повернення каретки.

Пробільні символи на початку та в кінці значення обрізаються. Послідовність кількох пробілів усередині значення може сприйматися як один пробіл. Регістр

символів у назві та значення не має значення (якщо інше не передбачено форматом поля).

Усі HTTP-заголовки поділяються на чотири основні групи:

- General Headers (Основні заголовки) – головні заголовки, які повинні бути прикріплені до кожного запиту;
- Request Headers (Заголовки запиту) – використовуються лише при відправки запиту зі сторони клієнта;
- Response Headers (Заголовки відповіді) – використовуються лише при відправки відповіді зі сторони сервера;
- Entity Headers (Заголовки сутності) – супроводжують кожну сутність повідомлення.

Сутності – це корисна інформація, що передається в запиті або відповіді. Сутність складається з метаінформації (заголовки) та безпосередньо змісту (тіло повідомлення).

В окремий клас заголовки сутності виділені, щоб не плутати їх із заголовками запиту або заголовками відповіді під час передачі множинного вмісту (multipart/*). Заголовки запиту і відповіді, як і основні заголовки, описують все повідомлення загалом і розміщуються лише у початковому блоці заголовків, тоді як заголовки сутності характеризують вміст кожної частини окремо, розташовуючись безпосередньо її тілом.

#### 2.3.1.6 Тіло повідомлення

Тіло HTTP повідомлення, використовується для передачі сутності, пов'язаної із запитом або відповіддю. Тіло повідомлення відрізняється від тіла сутності тільки в тому випадку, коли під час передачі застосовується кодування, зазначене в заголовку Transfer-Encoding. В інших випадках тіло повідомлення ідентичне тілу сутності.

Заголовок Transfer-Encoding має надсилатися для вказівки будь-якого кодування передачі, застосованого додатком з метою гарантування безпечної та правильної передачі повідомлення. Transfer-Encoding – це властивість повідомлення,

а не сутності, і воно може бути додане або видалено будь-яким додатком у ланцюжку запитів/відповідей.

Присутність тіла повідомлення у запиті відзначається додаванням до заголовків запиту поля заголовка Content-Length або Transfer-Encoding. Тіло повідомлення (message-body) може бути додано до запиту лише коли метод запиту допускає тіло об'єкта (entity-body).

Всі відповіді містять тіло повідомлення, можливо нульової довжини, крім відповідей на запит методом HEAD та відповідей з кодами статусу 1xx (Інформаційні), 204 (Немає вмісту, No Content), та 304 (Не модифікований, Not Modified).

### 2.3.2 WebSocket протокол

WebSocket – це протокол прикладного рівня стеку протоколів TCP/IP та моделі взаємодії відкритих систем OSI. Він знаходиться там же, де і протокол http. Для передачі в пакети на транспортному рівні використовують протокол TCP.

На відміну від сокетів транспортного рівня, які є інтерфейсом, якими протоколи прикладного рівня звертаються до сервісів транспортного рівня, web сокет це протокол, який визначає правила взаємодії між різними хостами на прикладному рівні.

У web сокетах на відміну від http, між клієнтом та сервером встановлюється постійне двонаправлене з'єднання. По цьому з'єднанню клієнт може будь-коли відправляти дані серверу і сервер також у будь-який момент може надсилати дані клієнту з власної ініціативи.

Також будь-яка сторона отримавши запит, може відповісти на нього відразу або через деякий час так, як це здається правильним і ефективним. Таким чином web сокети набагато краще підходять для розробки програм реального часу, ніж протокол http.

Web сокети як і http працюють на портах 80 і 443 якщо використовуються шифрування. Для веб-сокетів є спеціальний префікс в URL ws/ скорочення від веб-сокетів або wss/ якщо сокети використовуються спільно з SSL TLS для шифрування.

Робота протоколів web сокетів складається з двох частин, перша частина це встановлення з'єднання з англійської (Opening Handshake), друга частина – передача даних. Ці частини логічні розділені між собою. Для того щоб веб-сокети могли працювати в сучасній інфраструктурі розрахованої на протокол http, етап установки з'єднання дуже схожий на роботу протоколу HTTP.

У веб-сокетах запит клієнта на встановлення з'єднання виглядає як get запит http. При цьому вказується опція Upgrade та протокол, на який потрібно перейти – websocket. Також у заголовку вказується ключ web сокетів – це 16 байт згенерованих випадковим чином та представлених у кодуванні base 64.

Ключ використовується для захисту від фальшивих запитів на встановлення з'єднання з веб-сокетами.

Відповідь сервера на встановлення з'єднання веб-сокетів, також виглядає як відповідь за протоколом http. Код відповіді 101, перемикання протоколів.

У заголовку Upgrade, також вказується протокол, на який відбувається перемикання web сокети і вказується ключ безпеки, який отриманий на основі ключа відправленого клієнтам за спеціальним алгоритмом описаним у документі RFC 6455. Після цього між клієнтом і сервером встановлюється пряме tcp з'єднання, по якому дані передаються вже без використання протоколу http.

#### 2.3.2.1 Передача даних

У web сокетах дані передаються кадрами. Розробники протоколу web сокету намагалися знизити накладні витрати на передачу даних, тому заголовок web сокетів має бінарний вигляд на відміну від http, де заголовки у текстовому вигляді дуже великі. В результаті накладні витрати на передачу даних через web-сокети набагато нижче ніж через http.

#### 2.3.2.2 Фрагментація

Дані, які ви хочете передати через веб-сокети не обов'язково повинні поміщатися в один кадр, веб-сокети підтримують фрагментацію. Велике повідомлення може бути розділене на кілька частин і передається в декількох кадрах.

#### 2.3.2.3 Типи кадрів

У web сокетах є кадри трьох типів:

- кадри, які передають текстову інформацію, представлену в кодуванні utf-8;
- кадри, які передають дані у двійковому вигляді;
- керуючі кадри, які використовуються для підтримки або для закриття з'єднання.

#### 2.3.2.4 Формат заголовка кадру Web сокетів

Заголовок кадру в веб-сокетах зображено на рисунку 2.2.

Основні поля тут це код операції ( opcode ), який свідчить, що з тип кадру. Це може бути кадр, який передає текстові дані, бінарні дані, кадр ping, pong, close та інші типи кадрів.

У розділі Payload Data є корисні дані, які потрібно передати. У веб сокетах для того щоб знизити накладні витрати використовується хитра схема для вказівки обсягу даних, що передаються. Якщо обсяг даних невеликий, то для поля довжина даних використовується лише 7 біт, а якщо обсяг даних більше, то можуть використовуватися додаткові біти заголовка і розмір поля довжина даних може бути 16 або 64 біта.

У першій частині кадру є прапори. Перший вид прапор FIN використовується для вказівки фрагментації, якщо цей прапор дорівнює одиниці. тобто поточний кадр є останнім. при цьому, якщо дані помістилися в один кадр цілком і фрагментації немає. то прапор FIN встановлено на одиницю.

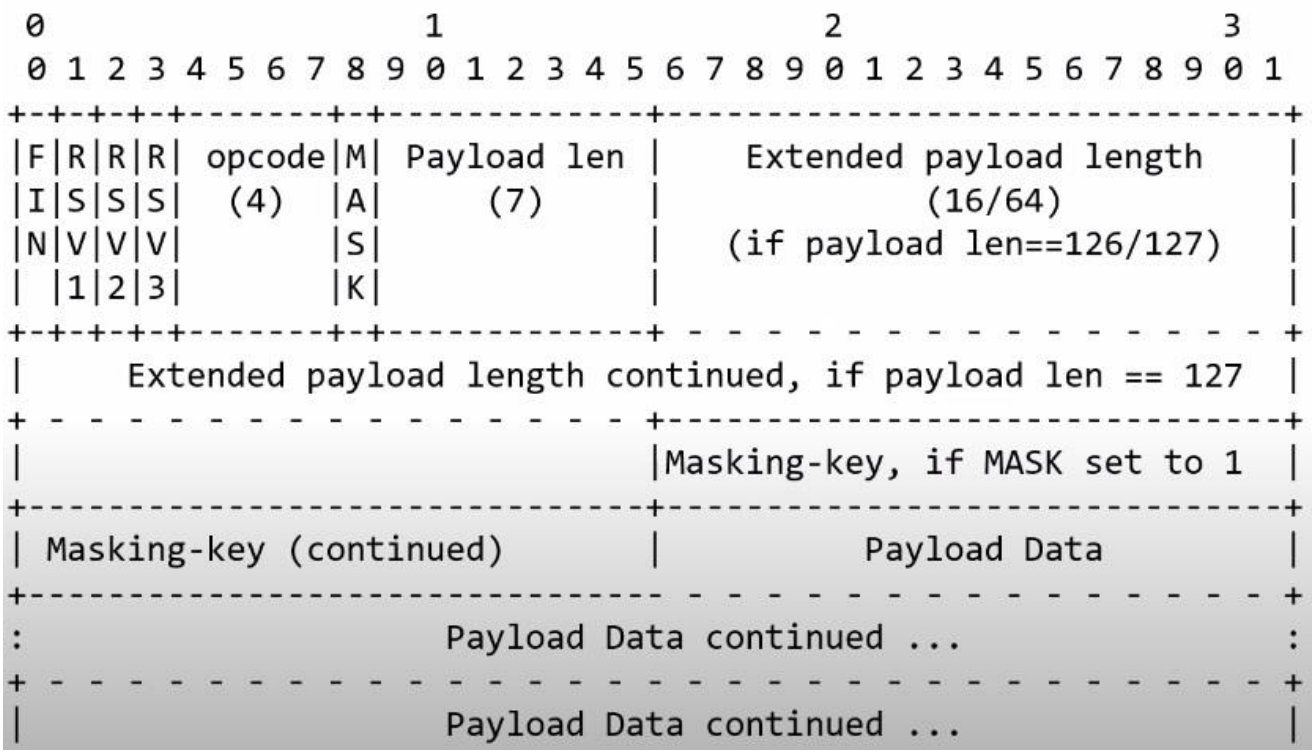


Рисунок 2.2 – Схема заголовку кадру

А якщо дані були фрагментовані і передаються в декількох кадрах, то у всіх кадрах крім останнього фланг FIN встановлено в 0. Наступні три прапори зарезервовані і використовуються для розширення протоколу веб порад. Чи будуть використовуватися розширення і які саме клієнту та серверу необхідно домовитися в процесі встановлення з'єднання.

У веб-сокетах дані при передачі можуть маскуватися і практично це робиться майже завжди, якщо дані маскуються то прапор MASK встановлений в одиницю, інакше в 0. Якщо використовується маскування даних, то кадр повинен включати ключ маскування, Masking-key.

Ключ маскування генерується відправником випадковим чином і цей ключ повинен бути різним для кожного кадру, якщо використовуються маскування, то в частині Payload Data знаходяться дані, отримані в результаті операції xor ключа маскування та оригінальних даних.

### 2.4 Обґрунтування і вибір методів експериментальних дослідження

В якості методу для експериментальних досліджень додатку, найкраще підійде порівняльний. В якості критеріїв було вирішено дослідити SEO оптимізацію додатку, швидкість завантаження та швидкість відпрацювання запитів від користувача.

Результати SEO тестування наглядним чином покажуть оптимізацію додатку для пошукових роботів. Результат даного тестування продемонструє, чи є сенс використовувати даний підхід для вирішення задач бізнесу.

Швидкість завантаження додатку продемонструє якість оптимізації додатку для завантаження, і час який користувач витратить для того, щоб почати користуватися програмою.

А швидкість відпрацювання запитів зможе відобразити якість та швидкість змін в користувальницькому інтерфейсі внаслідок дій користувача.

### 2.5 Висновки по розділу

У теоретичному розділі було розглянуто і обрано технології для розробки клієнтського та серверного середовища додатку. Розібрано основні протоколи які використовуються в сучасній веб розробці та розглянуто нюанси роботи з ними. В результаті було обрано методи експериментальних досліджень, за допомогою яких стане можливо оцінити ефективність додатку.

# 3 СИНТЕЗ СИСТЕМИ

## 3.1 Формулювання технічних вимог до системи

Так як в рамках даної дипломної роботи, розробляється система с серверним рендерингом даних на основі протоколу WebSocket, то працювати вона повинна лише на основі клієнт-серверної архітектури. Клієнт-серверна архітектура – найпоширеніший спосіб організації високорівневої мережевої взаємодії.

Мета впровадження системи – зменшити навантаження на клієнтську частину, шляхом перенесення обчислень, пов’язаних з відображенням даних на серверну частину. Як результат, ми можемо зменшити час завантаження додатку. Файли додатку також стануть меншим розміром і будуть оброблятися браузером швидше. Внаслідок, збільшиться аудиторія додатку, через більш лояльні системі вимоги до клієнтського пристрою.

Дана система повинна бути застосована у сфері надання інформаційних послуг за допомогою мережі інтернет.

Головна ціль будь-якого клієнт-серверного додатку – це забезпечити зв’язок та передачу даних між клієнтом та сервером. В більшості випадків, для веб –додатків зв’язок забезпечуються шляхом використання протоколу HTTP.

В свою чергу, призначення розроблюваної системи – показати, що ефективний способом передачі даних у веб-застосунках може виступати протокол WebSocket. За допомогою нього з’являється можливість встановлення постійного та швидкого з’єднання, яке значно прискорить швидкість роботу інтерактивних частин веб-додатків. Звичайно, на даний момент не має можливості повність відмовитися або замінити протокол HTTP, але є можливість використовувати його тільки там, де без нього не можна або де він має перевагу.

### 3.1.1 Вимоги до реалізації системи

Щоб забезпечити працездатність системи, необхідно мати веб сервер, сервер бази даних, персональні комп’ютери для користувачів, маршрутизатори та комутатори.

Для того, щоб веб-сервер та сервер бази даних міг працювати, потрібно встановити операційну систему. Об’єктивним вибором на сьогоднішній день буде обрати операційну систему Linux та дистрибутив Ubuntu версією 21.04. Її легко встановити, проста у використанні та налаштуванні, а також має комерційну підтримку світового рівня від компанії Canonical. Зараз Ubuntu найчастіше використовують для забезпечення роботи веб-серверів серед усіх інших дистрибутивів та операційних систем. Також, через популярність Ubuntu, немає необхідності перенавчати персонал для роботи з нею, тому що будь-який системний адміністратор вміє з нею працювати.

Для побудови мережі між компонентами системи, було обрано маршрутизатори та комутатори від компанії Cisco. Мережеве обладнання від Cisco забезпечує високий рівень адаптивності до сфери діяльності за рахунок автоматизації мережі, автоматичне відновлення даних та їх резервування після можливих збоїв, показує відмінні показники параметрів надійності та відмовостійкості та мінімізує вразливості у будь-яких точках мережі.

Для персональних комп’ютерів в якості операційної системи було обрано Windows 10, через те, що це найбільш знайома для користувачів система, і у користувачів не виникне проблем з її використанням.

Для забезпечення постійного мережевого з’єднання між компонентами системи повинен бути найманий персонал у вигляді системного адміністратора.

### 3.1.2 Вимоги до видів забезпечення

Вимоги до серверного середовища

На більшості платформ при встановленні Node.JS можливо використовувати декілька підході. Все залежить від операційної системи та кваліфікації розробника. Node.js можливо встановити, якщо ви маєте копію вихідного коду, і скомпілювавши його на цій ОС. Таким чином ви можете встановити цю платформу майже скрізь, де виможете скомпілювати програмний код на мові С. Це означає що програмні вимоги ми будемо обирати такі, як і мінімальні вимоги для Ubuntu.

Мінімальні вимоги до апаратного забезпечення наступні:

- 64-розрядний, двох ядерний процесор з частотою 2 ГГц, та 8 МБ кеш-пам’яті;
- 4 ГБ оперативної пам’яті;
- мінімальний розмір накопичувача – 32 ГБ;
- адаптер Ethernet із пропускною здатністю 100 Мбіт/с.

Дані характеристики забезпечують достатній об’єм оперативної пам’яті та потужності процесора для обробки великої кількості підключень по WebSocket протоколу, дають змогу підтримувати підключення до бази даних та відпрацювати велику кількість вхідних запитів, надають достатній об’єм кеш пам’яті процесора для забезпечення гарної швидкодії при запуску процесів та прискорення читання та запису даних та дозволяють здійснити підключення до мережі інтернет.

Вимоги до клієнтського середовища:

Якихось особливих вимог до клієнтського середовища немає. Пристрій користувача повинен лише мати змогу встановити операційну систему та веб-браузер для користування додатком. Тому при визначенні системних вимог до клієнтського середовища потрібно відштовхуватися перш за все від системних вимог операційної системи.

Мінімальні вимоги до апаратного забезпечення наступні:

- 64-розрядний, двох ядерний процесор з частотою 1 ГГц;
- 2 ГБ оперативної пам’яті;
- мінімальний розмір накопичувача – 32 ГБ;
- графічна карта с підтримкою DirectX 9 з WDDM 1.0 драйвером;
- адаптер Ethernet із пропускною здатністю 100 Мбіт/с.

Дані характеристики забезпечують достатній об’єм оперативної пам’яті, для запуску операційної системи та веб-браузера, дають здатність виводити зображення на екран, мають достатній об’єм жорсткого диску для збереження та відтворення інформації та роблять можливим підключитися до мережі інтернет.

Вимоги до серверу бази даних

На офіційному сайті MongoDB вказано, що розраховувати системні вимоги потрібно відповідно до розрахункового навантаження. Наприклад, розмір ОЗУ розраховується таким чином, що на кожні сто тисяч записів необхідно приблизно 1ГБ оперативної пам’яті. Також необхідно врахувати мінімальні системні вимоги операційної системи Ubuntu.

Мінімальні вимоги до апаратного забезпечення наступні:

- Двох ядерний ЦП, або одно ядерний з підтримкою мультипоточності;
- 8 ГБ ОЗУ;
- мінімальний розмір накопичувача – 512 ГБ (Бажано, щоб накопичувач був на швидких та твердотілих накопичувачах);
- адаптер Ethernet із пропускною здатністю 1 ГБіт/с.

## 3.2 Функції системи та інформаційні зв'язки між компонентами

### 3.2.1 Вимоги до функцій підсистем

Функції, які повинні виконуватися додатком на стороні сервера:

- зберігання, захист та доступ до даних;
- оброблення клієнтських запитів;
- встановлення з’єднання с користувачем на протоколі WebSocket;
- виконання прикладних функцій, характерних для предметної області додатку;
- надсилання відповіді на запит клієнтові;
- отримання запитів на оновлення інтерфейсу, підготовка макету з даними, та відправка відповіді.

Функції, які повинні виконуватися додатком на стороні клієнта:

- надання графічного інтерфейсу, для доступу до додатку;
- отримання вхідних даних від користувача;
- відправлення запиту на встановлення з’єднання з сервером на протоколі WebSocket;
- формулювання запитів до серверу, та подальша їх відправка;

- отримання результатів та відображення їх користувачеві;
- відправка запиту на оновлення інтерфейсу.

### 3.2.2 Інформаційні зв’язки між компонентами системи

В якості інформаційних зв’язків було взято потоки даних між компонентами системи. До пристрою користувача поступають сигнали с пристрої вводу, а в свою чергу він відправляє інформацію пристрої виводу та запити на сервер. Сервер отримує запити, формує на них відповіді, і зберігає або читає дані с серверу бази даниї.

### 3.2.3 Вимоги до якості реалізації функцій

Комп’ютерна система повинна забезпечувати функції, які наведені в пункті 3.2.1 с якістю, яка забезпечує виконання часових регламентів.

Часові регламенти такі:

- час, на підключення до серверу не повинен перевищувати 80 мс;
- час, до отримання першого байту інформації не повинен перевищувати 100мс;
- перше відображення вмісту сторінки повинне відбутися не пізніше чим через 1.8с;
- час, доки сторінка стане повністю робоча, не повинен перевищувати 3.8с.

Канали зв’язку повинні працювати стійко і забезпечувати високу швидкість передачу даних між компонентами системи, а саме 100 Мбіт/с.

## 3.3 Розробка схеми функціональної структури

Виходячи з переліку функцій, які повинні виконувати елементи системи, схема функціональної структури системи матиме вигляд, як показано на рисунку 3.1.

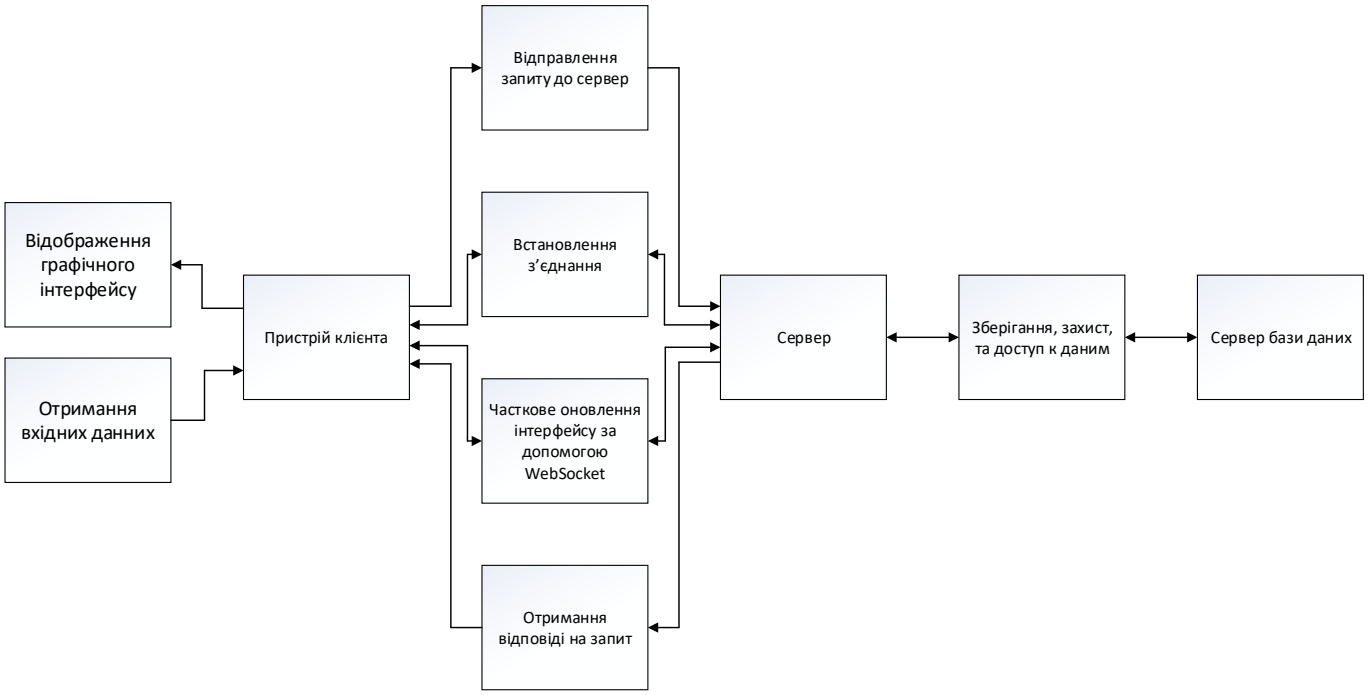


Рисунок 3.1 – Схема функціональної структури системи

### 3.4 Вибір та обґрунтування застосування апаратних засобів

Після визначення вимог до видів забезпечення, було обрано наступні прилади.

В якості веб-серверу обрано – ARTLINE Business T13 v12. Даний сервер має збалансовану продуктивність і невелике споживання енергії, що дає змогу використовувати його з мінімальними затратами.

Сервер ARTLINE Business T13v12 побудований на базі материнської плати Asus Prime H570-PLUS [9]. Плата підтримує процесори Intel 11-го та 10-го поколінь. ASUS PRIME H570-PLUS містить: модуль регулятора напруги VRM (Digi+) для точного настроювання стабільної та ефективної роботи процесора, технологій підвищення продуктивності (ASUS OptiMem) та керування ресурсами енергоспоживання (ASUS EPU).

Продуктивний процесорний чип десятого покоління Intel Pentium Gold G5400 має два ядра (4 потоки обробки) з частотою роботи 4.0 ГГц. Такого процесору буде достатньо для надійного функціонування нашої системи.

Також встановлено швидкий SSD накопичувач на 250 ГБ забезпечить дуже швидке завантаження системних файлів ОС та інших програм.

На рисунку 3.2 зображено фото серверу.

Таблиця 3.1 – Характеристики ARTLINE Business T13 v12

| Призначення | Веб-сервер |
|---|---|
| Тип пристрою | Сервер |
| Процесор | Двоядерний Intel Pentium Gold G6400 (4.0 ГГц) |
| Оперативна пам'ять | 8 ГБ DDR4-2666 МГц |
| Материнська плата | PRIME H570M-PLUS |
| Накопичувачі | SSD: 250 ГБ HDD: 1 ТБ |
| Ширина каналу інтернет | 1 Гбіт/с |

Рисунок 3.2 – Фото ARTLINE Business T13 v12

В якості серверу бази даних було обрано – HPE DL20G9 G4560 [10]. Це компактний сервер на базі процесору Intel. Сервер вирізняється низькою ціною і високою гнучкістю конфігурації, що підходить для вимог даної системи. Основні характеристики збігаються з попереднім варіантом, але даний сервер має декілька цікавих технологій.

Перша, це технологія коригування помилок (ECC) пам'яті HPE DDR4 дає змогу запобігти простою і втраті даних, а також підвищити продуктивність робочих навантажень і оптимізувати споживання енергії.

Друга – технологія HPE Smart Array забезпечує зеркалювання масивів RAID і захист даних завдяки їхньому розподілу між різними накопичувачами, а функція кешування у флеш-пам'ять місткістю до 4 ГБ (FBWC) захоплює дані та зберігає їх на випадок вимкнення електрики, збоїв в обладнанні, вірусної атаки або помилки персоналу.

Ці технології роблять даний сервер найкращим варіантом.

На рисунку 3.3 зображено фото серверу.

Рисунок 3.3 – Фото HPE DL20G9 G4560

Таблиця 3.2 – Характеристики HPE DL20G9 G4560

| | |
|---|---|
| Призначення | Сервер бази-даних |
| Тип пристрою | Сервер |
| Процесор | Двоядерний Intel Pentium G4560 (3.5 ГГц) |
| Оперативна пам'ять | 8 ГБ DDR4-2133 МГц |
| Материнська плата | PRIME H530M |
| Накопичувачі | SSD: 1 ТБ |
| Ширина каналу інтернет | 1 Гбіт/с |

В якості клієнтського пристрою було обрано ноутбук – Lenovo V15-ADA [11]. Так як немає особливо важких задача, які повинен виконувати клієнтський пристрій, то було обрано бюджетний варіант. Він в основному створений для виконання повсякденних офісних задач і використання веб-браузеру.

На рисунку 3.4 зображено фото клієнтського пристрою.

Рисунок 3.4 – Фото Lenovo V15-ADA

Таблиця 3.3 – Характеристики Lenovo V15-ADA

| | |
|---|---|
| Призначення | Клієнтський пристрій |
| Тип пристрою | Ноутбук |
| Процесор | Двоядерний AMD 3020e (1.2 — 2.6 ГГц) |
| Оперативна пам'ять | 4 ГБ DDR4-2400 МГц |
| Екран | 15.6" (1366x768) WXGA HD |
| Накопичувачі | HDD: 500 ГБ |
| Ширина каналу інтернет | 100 Мбіт/с |

В якості маршрутизатора було обрано – Cisco 2911-SEC [12]. Він продемонстрований на рисунку 3.5, а його характеристики відображені в таблиці 3.4.


Рисунок 3.5 – Фото Cisco 2911-SEC

Таблиця 3.4 – Характреристики маршрутизатора Cisco 2911-SEC

| Призначення | Маршрутизатор |
|---|---|
| Підтримка протоколів | L2TP; IPsec; PPPoE; DHCP |
| Функції безпеки | Шифрування DES, 3DES, AES-128 |
| Інші функції | Продуктивність у постачанні 50Mbps з сервісами<br>Розширення продуктивності за допомогою ліцензії Performance до 100Mbps<br>Підтримка ACL, SSHv2 та інших механізмів безпеки |
| Додаткові характеристики | 4 ГБ Flash; 4 ГБ DDR3 DRAM |

В якості комутатора було взято – Cisco Catalyst 2960-X 24 [13]. Він продемонстрований на рисунку 3.6, а його характеристики відображені в таблиці 3.5.

Рисунок 3.6 – Фото Cisco Catalyst 2960-X 24

Таблиця 3.5 – Характеристики комутатора Cisco Catalyst 2960-X 24

| Призначення | Комутатор |
|---|---|
| Швидкість передачи даних | 100 Мбіт/с |
| Повнодуплексний режим | Підтримує |
| Кількість портів | 24 |
| Оперативна пам'ять | 64 МБ |
| Флеш-пам'ять | 32 МБ |

### 3.5 Вибір та обґрунтування системного програмного забезпечення

Завантажити і розгорнути наш додаток можливо майже на будь які операційній системі. Можливі варіанти операційних систем:

- Windows 7 або пізніші версії;
- OS X 10.9 або пізніші версії;
- Ubuntu 12.04 або пізніші версії;
- Fedora 21 або пізніші версії;
- Debian 8 або пізніші версії.

Для серверного обладнання має бути встановлений Node.js версії 16.3.1 та npm версії 8.3.0. Node.js потрібен для запуску вихідного коду, а npm для встановлення всіх необхідних залежностей проекту.

Для сервера бази даних повинно бути встановлено MongoDB версії 5.0.5.

На клієнтському пристрою повинен бути встановлений веб-браузер. Оптимальним варіантом є браузер Google Chrome версії 96.

### 3.6 Структурна схема обладнання системи

В основі системи повинна лежати стандартна клієнт-серверна архітектура, в основі якої будуть лежати наступні компонента: клієнит, веб-сервер і сервер бази даних.

Клієнт – комп’ютер на стороні користувача, який буде відправляти запити до сервера для надання або отримання якоїсь інформації.

Сервер – призначений для вирішення певних завдань з виконанням програмних кодів, виконання сервісних функцій за запитом клієнтів, надання користувачу доступу до певних ресурсів, зберігання інформації в базі даних.

Сервер бази даних – призначений для обслуговування та управління базою даних, а також відповідаю за обробку, збереження та цілісність даних користувача.

Модель такої системи полягає в тому, що клієнт відправляє запит на сервер, де він обробляється, і готовий результат відправляється клієнтові. Сервер може обслуговувати кілька клієнтів одночасно. Якщо одночасно приходить більше одного запиту, то вони встановлюються в чергу і виконуються сервером послідовно. Іноді

запити можуть мати пріоритети. Запити з більш високими пріоритетами повинні виконуватися раніше.

Всередині нашого фізичного сервера, ми розгорнемо HTTP та WebSocket сервер, які саме і будуть обробляти запити і повідомлення від користувача.

Також ми будемо використовувати сервер бази даних, на якому будемо зберігати інформацію.

Структурна схема системи зображена на рисунку 3.5.

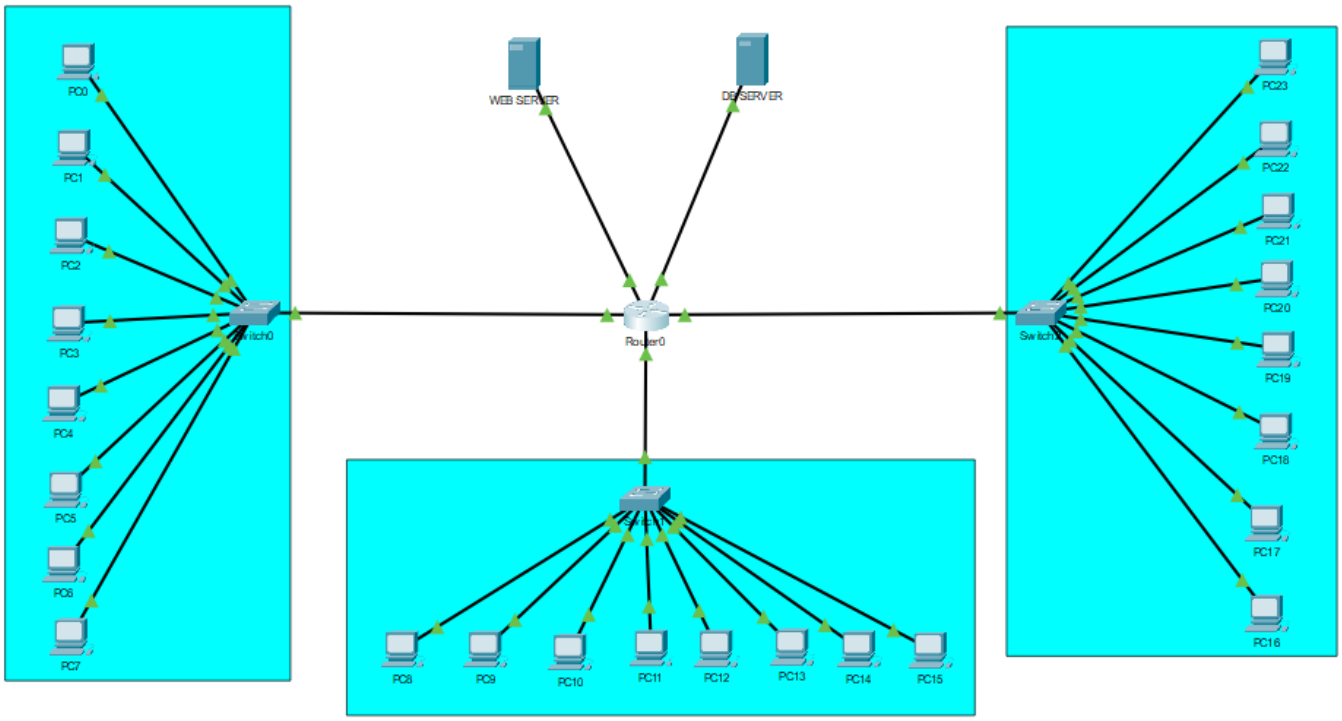


Рисунок 3.7 – Структурна схема системи

### 3.7 Висновки по розділу

В даному розділі було визначено сфери застосування і головне призначення системи. На основі цих даних стало можливо визначити функції, які повинні виконувати елементи системи і створено схему функціональної структури. Далі для кожного з елементів системи були висунуті вимоги до апаратного та програмного забезпечення і обране необхідне обладнання для функціонування системи. Фінальним етапом стало створення структурної схеми додатку.

# 4 РОЗРОБЛЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

## 4.1 Призначення та сфера застосування

Програма призначена для демонстрації нового підходу в сучасній веб-розробці, а саме організація SSR за допомогою протоколу WebSocket. Перевагою такого підходу є покращення досвіду користувача додатка, збільшення швидкості завантаження і покращена SEO оптимізація.

Використовувати такий підхід можна при розробці майже будь-якого веб-додатку та будь-якої складності. Перш за все, алгоритмом роботи цієї програми можуть зацікавитися розробники інтернет магазинів. Їм важливо щоб додаток був інтерактивним і мав гарну пошукову видачу. Все це можливо отримати, розробивши додаток за цим алгоритмом.

## 4.2 Обґрунтування технічних характеристик програми

### 4.2.1 Постановка завдання на розробку програми

Додаток повинен передавати на запит користувача відображені дані, які матимуть вигляд частини HTML розмітки. Передача даних організована за допомогою протоколу WebSocket. Перш за все потрібно завантажити статичні файли користувачеві та встановити з’єднання. Далі при використанні додатку користувач відправлятиме запити на WebSocket сервер і такий же сервер на стандартний HTTP сервер. В результаті, користувач повинен отримати результати виконання запитів, а також інформація про швидкість виконання цих запитів (у вигляді графіку та на віджеті).

### 4.2.2 Опис алгоритму функціонування програми

Для розробки цього додатку ми будемо використовувати клієнт-серверну архітектуру, з використанням підходу «тонкий» клієнт та «товстий» сервер. Суть такого підходу закладається в тому, що завдання з обробки даних переносяться на сервер, не використовуючи свої обчислювальні можливості для їх реалізації. Обчислювальні ресурси такого клієнта дуже обмежені, їх повинно бути досить лише

для запуску необхідного мережевого додатку, використовуючи, наприклад, веб-інтерфейс.

Після того, як користувач введе URL адресу пошуковий рядок, і натисне Enter, користувач повинен отримати всі статичні файли, які необхідні для функціонування додатку на стороні клієнта. До таких файлів відноситься HTML, CSS та JS файли, а також всі допоміжні ресурси (шрифти, картинки та файли, які допомагають краще індексувати сторінку в пошукових видачах).

На цьому кроці буде використатися CDN кешування для веб-сторінок, а також стандартне браузерне кешування для HTTP запитів на отримання відповідей на запити. Важливо щоб таких файлів було не дуже багато, тому що HTTP протокол має обмеження на кількість одночасних запитів. Передавати необхідно тільки ті файли, які необхідні для відображення користувальницького інтерфейсу, файли зі стриптами потрібно відкладати в черзі для завантаження. Також, якщо ви впевнені, на яке сторінку далі перейде користувач, її можна перезавантажити за допомогою тега link та атрибута rel зі значенням prerender.

Всі ці маніпуляції призведуть до найкращого користувацького досвіду, забезпечать найкращу значення, для більшості метрик (час до початку відображення, час до інтерактивності і так далі).

Після того як всі файли завантажаться, і відобразяться – починається завантаження та запуск скриптів. В одному із таких скриптів ми ініціалізуємо підключення до WebSocket серверу. Якщо все проходить вдало, то ми отримуємо постійний та повнодуплексній канал зв’язку, через який ми легко можемо передавати дані в обидві сторони.

Далі, для будь якого даних в інтерфейсі користувача, ми будемо відправляти запит на сервер через WebSocket з’єднання, і сервер нам буде генерувати готовий шаблон з даними і відправляти на клієнт. Саме на цьому етапі ми будемо використовувати технологію SSR, тобто ми будемо генерувати готову розмітку на стороні сервера, і клієнт не буде витрачати час та ресурси на обробку даних.

Перевагами в такому підході є те, що нам не потрібно постійно встановлювати нове з’єднання для кожного запиту, як це ми робимо при роботі з HTTP протоколом.

Також ми виграємо багато часу на тому, що браузер не буде робити preflight запит, щоб дізнатися чи має клієнт доступ до цього серверу. Немало важливим є те, що ми також уникаємо можливих черг в запитах, які можуть створитися при занадто частих зверненнях до серверу.

Слід згадати, шо заголовки в WebSocket кадрах передаються в бінарній формі, а тому набагато менший розмір аніж заголовки HTTP запиту. На цьому ми економимо час та трафік користувача на отримання даних. І в цілому повідомлення передаються набагато швидше аніж запити.

Загальна структура роботи мережевих запитів буде виглядати приблизно як на рисунку 4.1.

Далі таке з’єднання залишається відкритим до того моменту, поки користувач не покине сторінку, або не перейде на якусь іншу, в цьому ж додатку. І весь алгоритм повторюється знову.

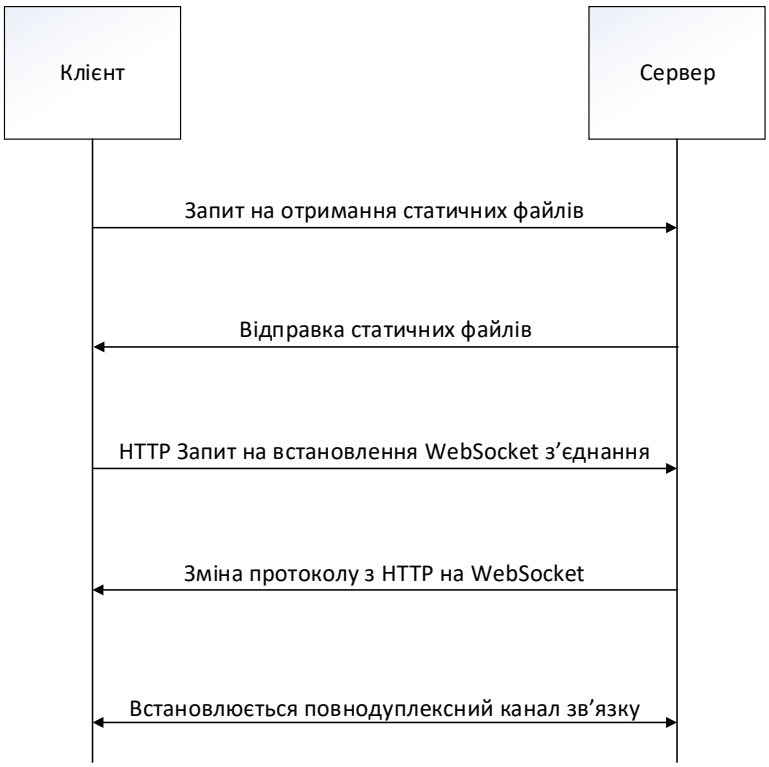


Рисунок 4.1 – Структура роботи мережевих запитів у додатку

## 4.3 Опис і обґрунтування вибору складу технічних і програмних засобів

### 4.3.1 Проектування клієнтського середовища

Клієнтське середовище у даному додатку написане за допомогою стандартного стеку технологій. А саме HTML, CSS та JS. Було підключено тільки бібліотеку для роботу з графіками, так як написати свою було б дуже складно, і не ефективно. Більше ми не використовуємо сторонніх бібліотек та фреймворків через те, щоб нічого не впливало на результати тестування швидкості завантаження сторінки та відпрацювання запитів. Найкращих результатів можна досягти лише в тому разі, якщо в додатку буде використано лише вбудовані в браузер технології.

Вся клієнтська розмітка написана на HTML. В розмітці для цього проекту немає нічого особливо, на чому сильно хотілось би загострити увагу. Із незвичайного тільки те, що в секції head знаходиться посилання, яке завчасно завантажить і відмалює наступну сторінку для користувача, таким чином користувачу не потрібно чекати при переході на неї. Воно має такий вигляд:

```
<link rel="preload" href="./autocomplete/index.html" as="document">.
```

Стилі також написані без будь-яких фреймворків та препроцесорів. Стилі написані за допомогою методології БЕМ (Блок Елемент Модифікатор). Такий підхід спрощує код стилів та подальший рефакторинг, а також допомагає повторно використовувати код та уникнути взаємного впливу компонентів один на одного (колізії стилів).

На початку файлу імпортується користувальницький шрифт. Це зроблено з метою введення більшої кросплатформенності, так як стандартний набір шрифтів у кожного браузера та ОС різний, і додаток може візуально відрізнятися у різних користувачів.

```
@import
url('https://fonts.googleapis.com/css2?family=Roboto:ital,wght@0,300;0,
400;0,700;1,300;1,400;1,700&display=swap');
```

Також використовуються CSS змінні, які допомагають більш гнучко писати стилі та легко змінювати їх.

Приклад оголошення змінних:

```
:root {
  --spacing: 0.5rem;
  --spacing-2: calc(var(--spacing) * 2);
  --spacing-3: calc(var(--spacing) * 3);
  --spacing-4: calc(var(--spacing) * 4);
  --backgroud-color: #E0FFFF;
  --link-color: rgb(33, 59, 204);
  --link-hover-effect: 0px 0px 8px rgb(131, 181, 247);
  --link-font-size: 1.5rem;

}
```

Також стилі для головної сторінки та сторінки з прикладом знаходяться в одному файлі. Таким чином браузер може їх закешувати, і при наступних запитах не скачувати їх заново, а просто брати із кеша браузера.

Для скриптів додатку було додано одну бібліотеку. За допомогою неї ми виводимо дані, про швидкість відпрацювання HTTP запитів і WebSocket повідомлень. Ця бібліотека незначно впливає на швидкість завантаження сторінки, але гарно відображає результати роботи додатку.

Щоб зручно працювати з цією бібліотекою, було створено невеликий допоміжний файл, який містить початкове налаштування для неї.

Окрім цього була розроблена обгортка, над стандартним класом для роботи з WebSocket. Мета цієї обгортки – спрощення розробки додатку і імітування його роботи, наче ми працюємо с HTTP запитами. Причиною той факт, що більшість розробників звикло працювати з звичайними запитами, а також синтаксис мови заточений під них.

Обгортка має наступний вигляд:

```
class SocketClient {
  constructor() {
    const socketProtocol = location.protocol === 'https:' ? 'wss:' :
'ws:';
    this.socketPathname = location.pathname === '/' ? '' :
location.pathname;
    const socketUrl =
`${socketProtocol}//${location.host}${this.socketPathname}`;
    this.socket = new WebSocket(socketUrl);

    this.socket.addEventListener('open', () => {
```

```
      this.socket.addEventListener('message', (event) => {
        const data = JSON.parse(event.data);
        EventEmmiter.emit(this._getRouteMask(data.path, data.method,
data.id), data);
      });
    });
  }

  _getRouteMask(path, method, id) {
    return `[${path}]:[${method}]:[${id}]`;
  }

  sendMessage(message) {
    message.id = Date.now();
    const json = JSON.stringify(message);
    this.socket.send(json);

    return new Promise((resolve) => {
      EventEmmiter.once(
        this._getRouteMask(this.socketPathname, message.method,
message.id),
        (event) => {
          resolve(event.detail);
        }
      );
    })
  }

  subscribe(callback) {
    this.socket.addEventListener('open', () => {
      this.socket.addEventListener('message', (event) => {
        const data = JSON.parse(event.data);
        callback(data, event);
      });
    });
  }

  onInit(callback) {
    this.socket.addEventListener('open', () => callback());
  }
}
```

В додатку ми створюємо екземпляр класу SocketClient і маємо можливість надсилати схожий на HTTP запит і дочекатися отримання результату. Коли в стандартній реалізації потрібно підписуватися на подію message і там обробляти дані, які ми отримали.

Результати продемонстровані на рисунках 4.2 та 4.3.

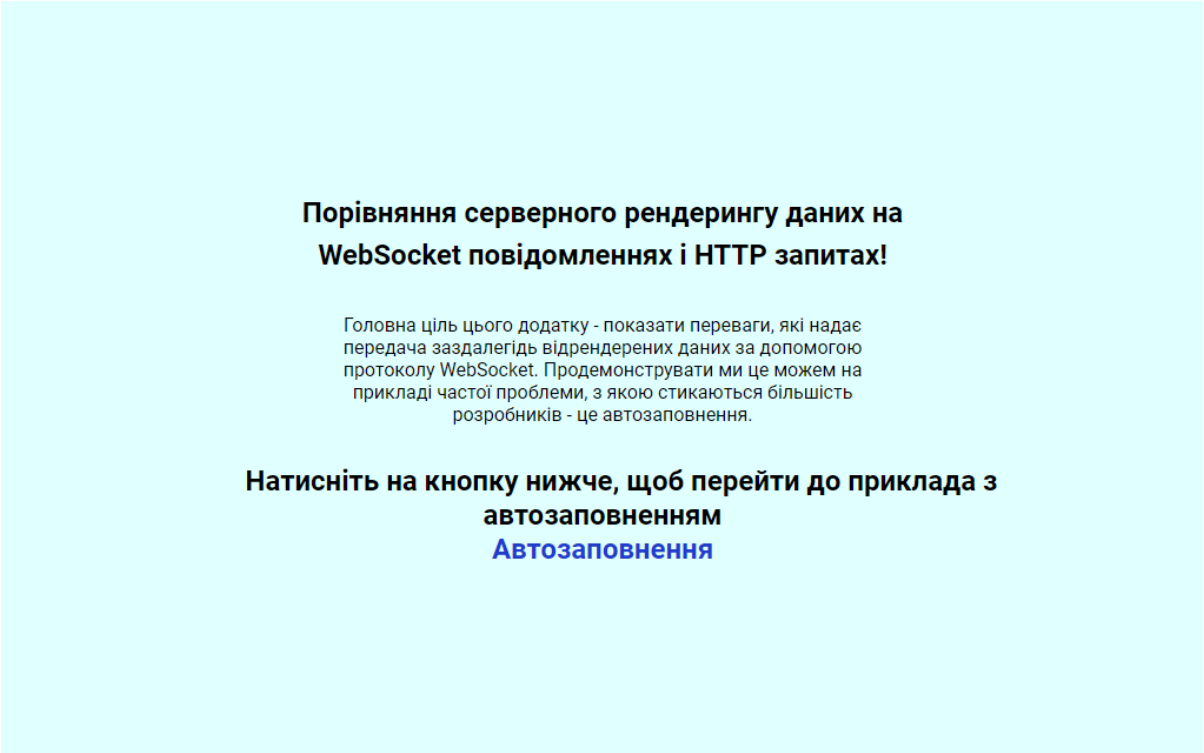


Рисунок 4.2 – Головна сторінка додатку

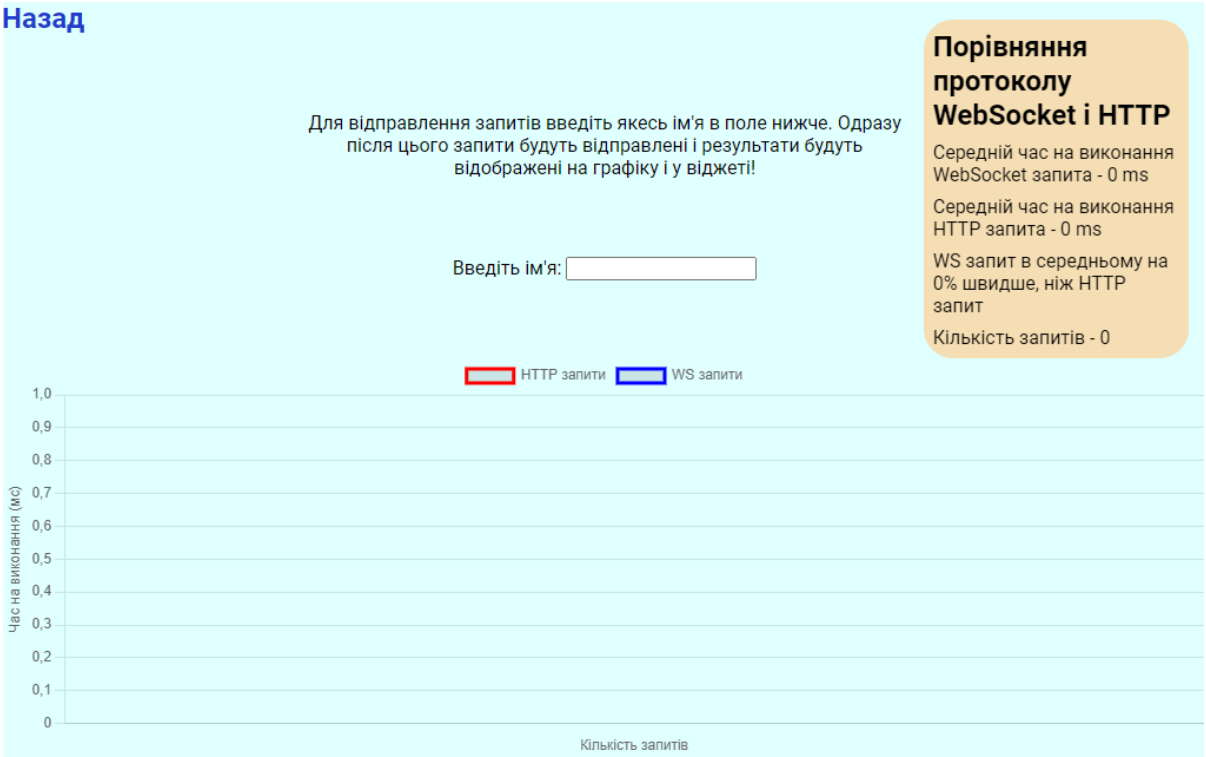


Рисунок 4.3 – Сторінка автозаповнення

### 4.3.2 Проектування серверного середовища

Розробку серверної частини було вирішено вести на основі серверного фреймворку Express.js. Він вирішує більшість проблем, з якими розробник може зіткнутися при розробці додатку такого плану. Він гнучкий, простий і дуже швидкий. При використанні Express.js ми отримаємо невеликий обсяг базового функціоналу, а всі інші необхідні функції ми добираємо за рахунок зовнішніх модулів або можемо написати їх самі.

Архітектура додатку побудована за допомогою шаблону MVC(Model-View-Controller, модель-представлення-контролер). За таким підходом ми ділимо нашу систему на три частини, які взаємопов’язані між собою. Першою частиною є модель даних яка відповідає за дані в нашому додатку, а також за його архітектуру. Другий компонент це представлення, яке відповідає за графічну частину, яку бачить користувач і регулює взаємодію користувача з інтерфейсом. І останній, третій – це компонент, який створює та відповідає за зв’язок між першими двома частинами. Приклад такої архітектури продемонстровано на рисунку 4.4.

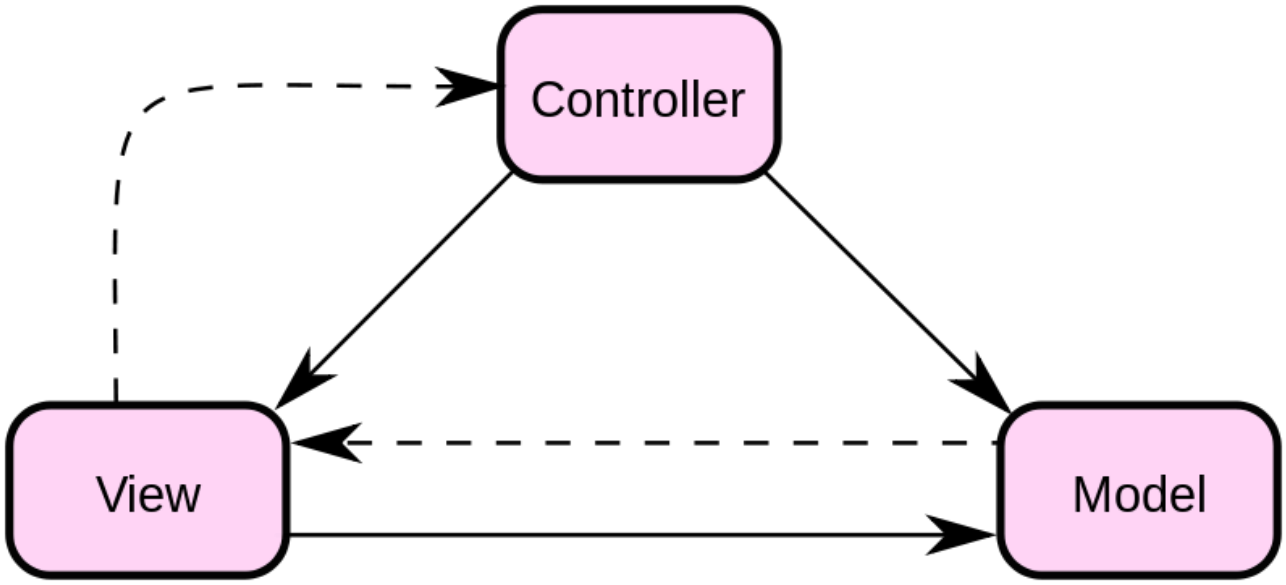


Рисунок 4.4 – Схема зв’язків MVC архітектури

Для розробки WebSocket серверу було встановлено невелику бібліотеку – ws. Вона є проста у використанні та надзвичайно швидка. Працює вона на основі базового http серверу який ми створюємо на основі Express.js.

Задля зручного використання було створено спеціальну обгортку, за допомогою якої стає можливо використовувати WebSocket сервер, як express, з влаштованим маршрутизатором і обробниками проміжних подій.

Ця обгортка виглядає наступним чином:

```
class WebSocketApplication {
    constructor(server) {
        this.webSocketServer = this._createServer(server);
        this.emitter = new EventEmitter();
        this.middlewares = []
    }

    use(middleware) {
        this.middlewares.push(middleware);
    }

    addRouter(router) {
        Object.keys(router.endpoints).forEach(path => {
            const endpoint = router.endpoints[path];
            Object.keys(endpoint).forEach((method) => {
                this.emitter.on(this._getRouteMask(path, method),
(...args) => {
                    const handler = endpoint[method];
                    handler(...args);
                })
            })
        })
    }

    _createServer(server) {
        const wss = new WebSocket.Server({ server: server });

        wss.on('connection', (ws, request) => {
            const path = request.url;

            ws.on('message', (data) => {
                const parsedData = JSON.parse(data);
                const method = parsedData.method;
                this.emitter.emit(this._getRouteMask(path, method),
path, parsedData, ws);
            })
        });

        return wss;
    }

    _getRouteMask(path, method) {
        return `[${path}]:[${method}]`
    }
```

```
}
```

Щоб маршрутизація працювала належним чином, було вирішено додати клас Router, який працює приблизно ж таким чином, як і стандартний маршрутизатор в фреймворці Express.js.

Код маршрутизатора:

```
class Router {
  constructor() {
    this.endpoints = {};
  }

  request(method = 'GET', path, handler) {
    if (!this.endpoints[path]) {
      this.endpoints[path] = {};
    }

    const endpoint = this.endpoints[path];

    if (endpoint[method]) return;

    endpoint[method] = handler;
  }

  get(path, handler) {
    this.request('GET', path, handler);
  }
  post(path, handler) {
    this.request('POST', path, handler);
  }
  put(path, handler) {
    this.request('PUT', path, handler);
  }
  delete(path, handler) {
    this.request('DELETE', path, handler);
  }
};
```

Для відображення даних та серверного відображення даних потрібно використати якийсь шаблонізатор. Було вирішено використовувати Pug, так як він має велику швидкість роботи, простий в написанні шаблонів і повністю підтримує весь можливий функціонал, який би ми могли написати в звичайному HTML документові. Також є можливість передати дані, під час виклику функції render.

Приклад готового шаблону:

```
datalist(id='users')
    each result in results
        option(value=result.name)
```

Приклад генерації шаблону:

```
pug.renderFile(
    path.join(__dirname, '../templates/autocomplete-template.pug'),
    {
      results: users,
    }
  );
```

Після фінальної розробки додатку, він був завантажений на віддалений сервер від Heroku. Heroku – це така платформа, яка дозволяє завантажувати та тестувати додаток безкоштовно. Але, головним недоліком є те, що ресурси, які виділяються для додатку є обмежені, і результати швидкості виконання запитів які ми отримуємо на хостингу є гірші, ніш отримуємо на локальному сервері. Тому результати будуть приведені з локального серверу, а додаток на віддаленому сервері наданий лише в демонстраційних цілях для показання працездатності ідеї.

В якості бази даних використовується документо-орієнтована система керування базами даних MongoDB. Сервер бази даних вирішено не встановлювати локально, а використовувати віддалений сервіс – Atlas Mongo DB. Він повністю задовольняє всі вимоги до серверу БД, і навіть в безкоштовній версії працює надзвичайно швидко.

Для зручності, встановлено драйвер для роботи з Mongo DB – mongoose. Mongoose – просте рішення на основі схем для моделювання даних додатку. Він включає в себе вбудований функціонал для переведення типів, перевірку даних на коректність, побудову запитів та багато іншого.

Приклад підключення додатку до БД:

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost:27017/test');
const User = mongoose.model('User', { name: String });
const user = new User({ name: 'Nikolay' });
user.save().then(() => console.log('Saved Successful'));
```

## 4.4 Опис розробленої програми

### 4.4.1 Загальні відомості

Для розробки програми «Демонстрація технології SSR на протоколі WebSocket» перш за все необхідно обрати мову програмування і додаткові програмні модулі.

Серверна частина написана за допомогою CommonJS – система мови програмування JavaScript, яка дозволяє цьому працювати за межами браузера.

Додатково використовуються наступні програмні модулі: ExpressJS, pug, ws, mongoose.

Клієнтська частина написана за допомогою HTML, CSS та JavaScript.

Додатково використовується програмний модуль Chart.JS.

Код програми написаний за допомогою програми Visual Studio Code.

Системні програмні засоби, що використовуються програмою, повинні бути представлені версією операційної системи Windows 8 або Windows 10. Також для серверної частини –повинен бути встановлений node.js та npm. Для клієнтської частини – будь-який веб-браузер

### 4.4.2 Функціональне призначення

#### 4.4.2.1 Призначення програми

Програма призначена для демонстрації організація серверного рендерингу даних за допомогою протоколу WebSocket. Вона дає можливість наглядно побачити всі переваги та недоліки  даного підходу. Користувач може вводити дані, на які система буде відправляти запит, обробляти його та видавати результат роботи.

Програма вирішує наступні проблеми:

- проблема оптимізації додатку для користувача;
- оптимізація для пошукових роботів;
- передача даних між сервером та клієнтом.

#### 4.4.2.2 Відомості про функціональні обмеження на застосування.

Програма буде коректно тільки в тому випадку, коли у користувача буде стабільне інтернет з’єднання, в іншому випадку він не зможе отримати доступ до неї.

### 4.4.3 Використовувані технічні засоби

До складу технічних засобів входить персональний комп’ютер (ПЕОМ), що включає в себе:

- процесор Intel Pentium 4 або пізніший, який підтримує SSE2;
- 512 МБ ОЗУ;
- накопичувач розміром 128 ГБ;
- ширина каналу інтернет від 10 МБіт/с.

### 4.4.4 Виклик і завантаження

#### 4.4.4.1 Спосіб виклику програми з відповідного носія даних

Програмний виріб передається мережею Internet у вигляді архіву. Перед запуском програми необхідно впевнитися що на комп’ютері встановлено node.js та npm. Далі необхідно завантажити залежності проекту, виконавши команду npm і в командному рядку. Після цього завантаження програми здійснюється набором в командному рядку команди – npm run start з кореневої директорії проекту.

#### 4.4.4.2 Вхідні точки в програму

Точка входу в користувальницький код – файл index.js.

### 4.4.5 Вхідні дані

В якості вхідних даних використовується наступна інформація:

- запит, який користувач вводить з клавіатури. Назва змінної – query. Формат – string.

### 4.4.6 Вихідні дані

В якості вихідних даних використовується наступна інформація:

- документ, який користувач бачить на екрані свого пристрою. Назва змінної – document. Формат – object (екземпляр класу Document).

## 4.5 Висновки по розділу

Метою даного розділу – була розробка робочого прототипу додатку, який би реагував на запити користувача, обробляв дані, генерував фрагмент HTML коду і відправляв його користувачеві через протокол WebSocket. В результаті з’являється можливість швидко реагувати на дії та оновлювати інтерфейс без перезавантаження сторінки та з гарною SEO оптимізацією.

На основі алгоритму функціонування програми розроблено робочий прототип для тестування, який завантажений на віддалений сервер. Для полегшення взаємодії користувача з додатком, було створено інтерфейс, на якому можливо отримати інформацію про запити та швидкість їх виконання. Також представлено графік, за допомогою якого є можливість швидко оцінити результати і зробити висновки.

Результатом розділу стала повністю робоча демонстраційна програма.

# 5 ЕКСПЕРЕМЕНТАЛЬНИЙ РОЗДІЛ

## 5.1 Мета і завдання експерименту

Мета експерименту: перевірити працездатність системи при впровадженні підходу до розробки клієнт-серверної архітектури комп’ютерної системи з протоколом WebSocket та технологією SSR.

Завдання експерименту: дослідним шляхом оцінити отримані показники, які ми отримуємо в результаті впровадження даної системи.

## 5.2 Методика експерименту

Експеримент буде проводитися за допомогою програмного забезпечення, яке було створено в 4-му розділі пояснювальної записки. Запуск та подальше проведення експерименту відбувається в стандартному середовищі персонального комп’ютера.

Для ефективності підходу необхідно отримати результати SEO оптимізації, стандартних метрик які ми можемо отримати з вкладки продуктивність та мережа в інструментах розробника, метрик Web Vitals та порівняти швидкість виконання WebSocket та HTTP запитів безпосередньо у самому додатку.

## 5.3 Оцінка результатів експерименту

### 5.3.1 Оцінка SEO оптимізації

Для перевірки додатку на SEO оптимізацію біло використано інструмент SEO Site Checkup. Навіть в безкоштовній версії інструменту, він надає доступ до багатьох важливих тестів, яких нажаль немає в інших подібних сервісах. Результатом є бали, які сторінка отримує за проходження тестів. Якщо сторінка набрала більше 65 балів – це вважається відмінним результатом.

Так як в додатку знаходяться лише дві сторінки, то ми маємо перевірити їх обоє. Результати продемонстровано на рисунках 5.1 та 5.2.

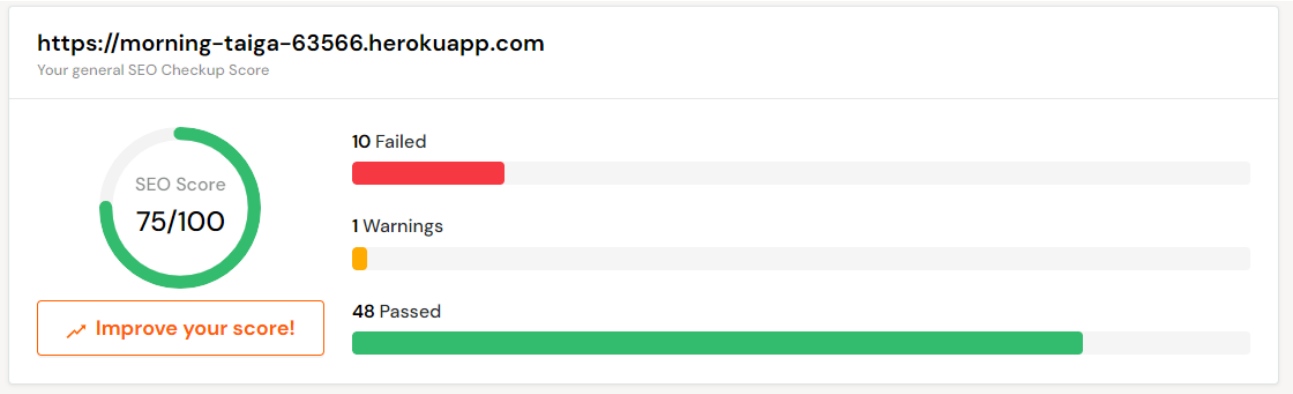


Рисунок 5.1 – Результат SEO тестування головної сторінки

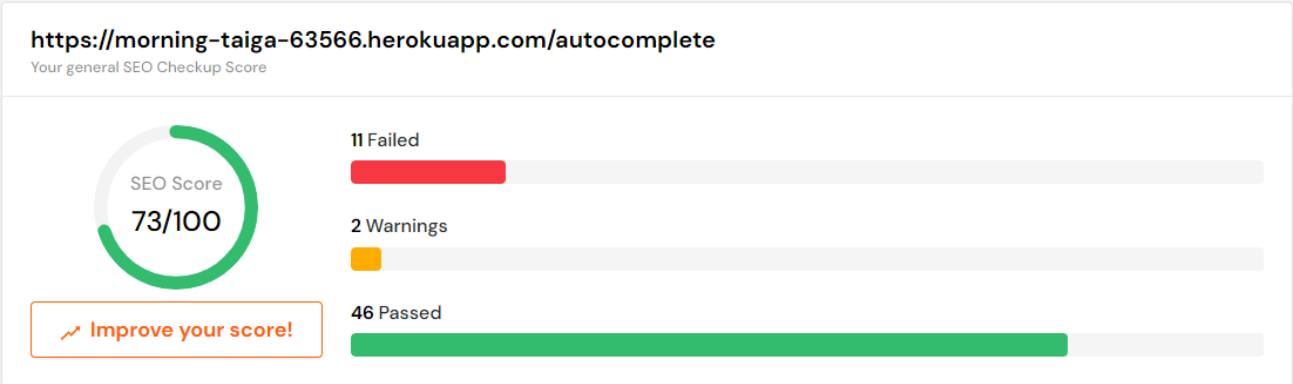


Рисунок 5.2 – Результат SEO тестування сторінки автозаповнення

Для порівняння, протестуємо додаток, який написано на популярному в даний час підході – SPA, а також додаток якому дуже важлива SEO оптимізація (інтернет-магазин «Rozetka»). Результати продемонстровано на рисунках 5.3 та 5.4.

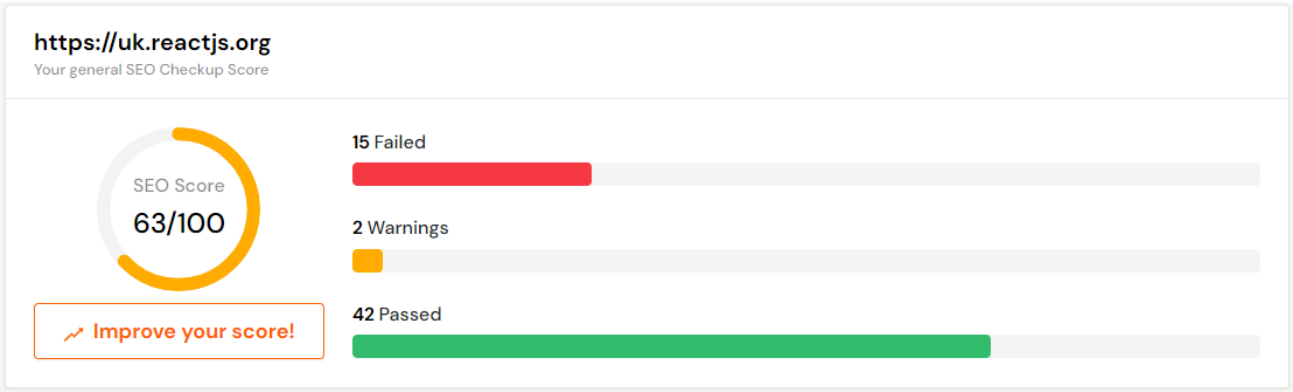


Рисунок 5.3 – Результат SEO тестування сторінки документації React.js

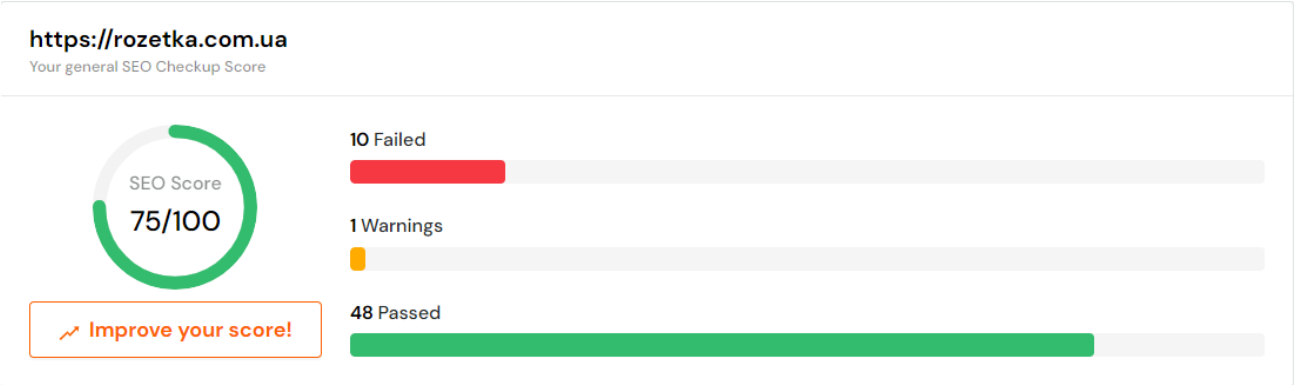


Рисунок 5.4 – Результат SEO тестування сторінки Rozetka

В результаті ми бачимо, що даний підхід гарантує гарну SEO оптимізацію, на рівні з таким сайтом, як https://rozetka.com.ua. Цей інтернет-магазин маж гарну пошукову видачу. При пошуку будь-якого товару в інтернеті, цей додаток точно буде якщо не перший в видачі, то на першій сторінці.

З документацією React.js все не так однозначно. Хоча вона і набрала досить високий бал, але це ще не означає гарну оптимізацію. Є такі пошукові роботи, які не можуть запускати і виконувати код JavaScript. В такому разі пошукова видача зменшиться в рази.

### 5.3.2 Оцінка за допомогою стандартних метрик браузера та Web Vitals

Для оцінки швидкості роботи нашого додатку, потрібно перш за все використовувати інструменти розробника, які надає нам браузер. Тестування буде проводитися в браузері Chrome. Дані отримаємо с вкладки продуктивність та Lighthouse. Також для тестування використаємо розширення для браузер, яке називається Web Vitals.

Метрики, за якими ми будемо оцінювати наш додаток:

– First Contentful Paint (перше відображення контенту) – вимірює час з моменту початку завантаження сторінки, до моменту, коли хоча-б якась частина нашої сторінки відобразиться на екрані користувача;
– Speed Index (індекс швидкості) – вимірює швидкість відображення вмісту під час завантаження сторінки. Тобто, ця метрика вимірює візуальний прогрес, між кадрами додатку під час рендерингу або оновлення;

– Larges Contentful Paint (відображення найбільшого контенту) – вимірює швидкість до завантаження основного контенту;
– Time to Interactive (час до інтерактивності) – вимірює час, який потрібен для того, що сторінка стала повністю інтерактивною;
– Total Blocking Time (загальний час блокування) – вимірює часовий проміжок між FCP і TTI. Тобто, основний потік браузера блокується, і користувач хоч і бачe контент, але не може с ним взаємодіяти;
– First Input Delay (затримка до першої взаємодії) – вимірює час, який потрібен браузеру щоб відреагувати на дію користувача.

Результати тестування продемонстровані на рисунках 5.5 – 5.10

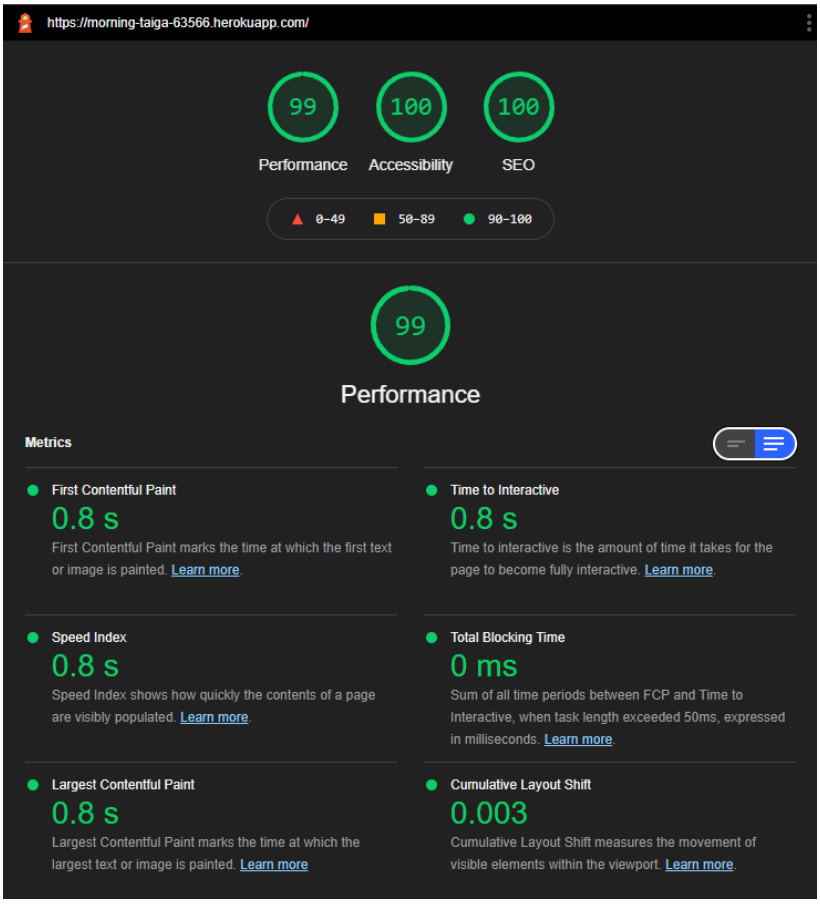


Рисунок 5.5 – Результат тестування головної сторінки інструментом Lighthouse

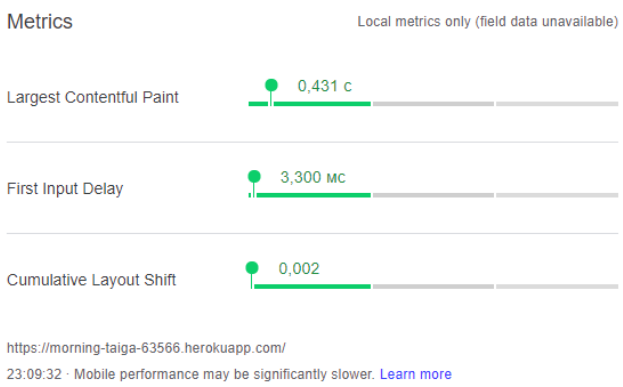


Рисунок 5.6 – Результат тестування головної сторінки інструментом Web Vitals

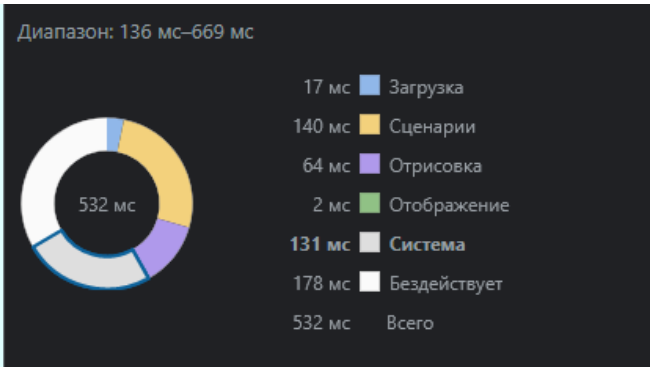


Рисунок 5.7 – Результат тестування головної сторінки з вкладки Продуктивність

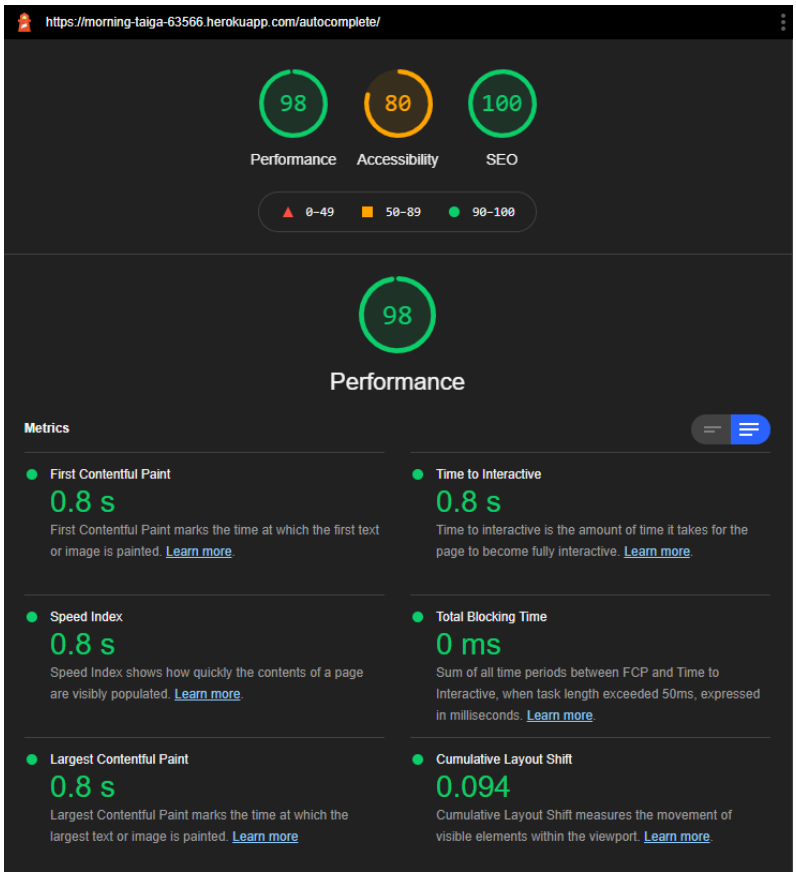


Рисунок 5.8 – Результат тестування сторінки автозаповнення інструментом Lighthouse

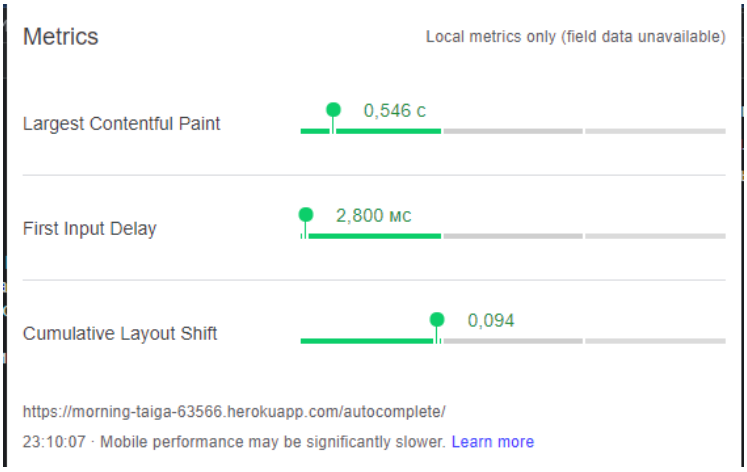


Рисунок 5.9 – Результат тестування сторінки автозаповнення інструментом Web Vitals

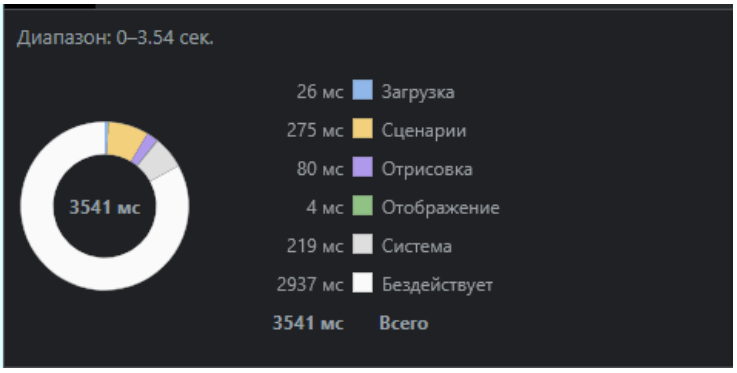


Рисунок 5.10 – Результат тестування сторінки автозаповнення з вкладки Продуктивність

Для зручності виразимо отриману інформацію у вигляді таблиць. Також порівняємо результати с допустимими значеннями. Для наочності, відобразимо таблиці у вигляді графіків. Графіки продемонстровані на рисунку 5.11 та 5.12.

Таблиця 5.1 – Порівняння отриманих значень з допустимими для головної сторінки

| | Отримані значення | Допустимі значення |
|---|---|---|
| FCP | 0.8 с | 1.8 с |
| SI | 0.8 с | 3.4 с |
| LCP | 0.8 с | 2.5 с |
| TTI | 0.8 с | 3.8 с |
| TBT | 0 с | 0.3 с |
| FID | 0.0028 с | 0.1 с |

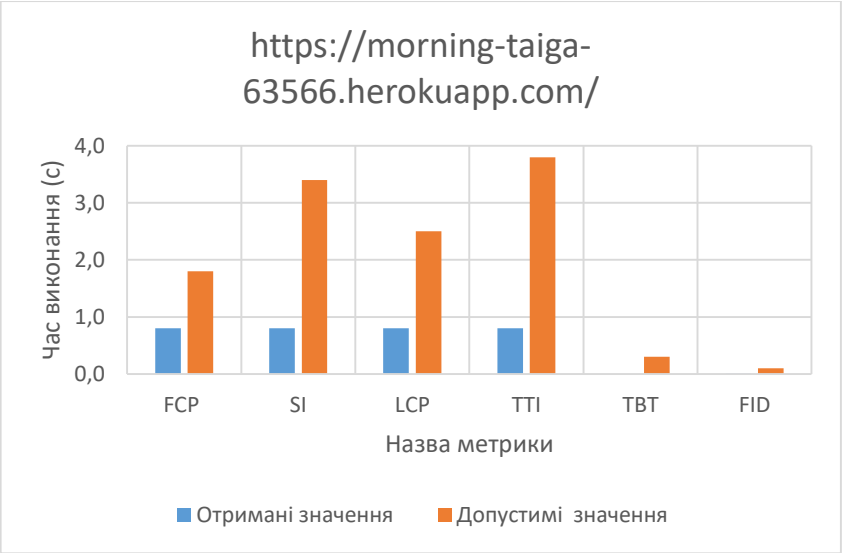


Рисунок 5.11 – Табличні результати у вигляді графіку для головної сторінки

Таблиця 5.2 – Порівняння отриманих значень з допустимими для сторінки автозаповнення

| | Отримані значення | Допустимі значення |
|---|---|---|
| FCP | 0.8 с | 1.8 с |
| SI | 0.8 с | 3.4 с |
| LCP | 0.8 с | 2.5 с |
| TTI | 0.8 с | 3.8 с |
| TBT | 0 с | 0.3 с |
| FID | 0.0028 с | 0.1 с |

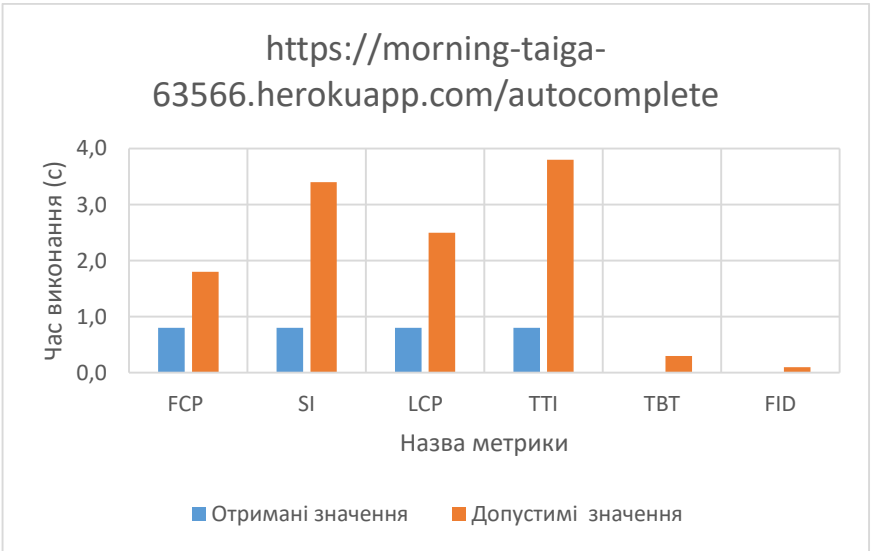


Рисунок 5.12 – Табличні результати у вигляді графіку для сторінки автозаповнення

В результаті було отримано значення, які значно швидші, аніж допустимі. Це свідчить про те, що швидкість завантаження та відображення сайту висока, і користувач для доступу до додатку, не буде довго чекати. Також це збільшить час утримання користувача та пошукову оптимізацію.

### 5.3.3 Оцінка швидкості відпрацювання запитів

Для оцінки швидкості відпрацювання запитів було обрано приклад з автозаповненням поля введення. Цей приклад потребує надзвичайної швидкості відправлення запитів та отримання відповідей, через те, що вони відправляються на кожен введений користувачем символ. Середня швидкість друку – приблизно 220 знаків в хвилину, тобто користувач друкує один символ раз в 270 мс. Це означає, що за одну секунду буде відправлено 4 запити до сервера. Також, вони повинні відпрацьовувати швидше в 2-3 рази, щоб результат був видний перед тим, як користувач введе наступний символ. Тобто очікувана ідеальна швидкість запиту приблизно 110мс. Але повинний бути запас, на той випадок, якщо користувач буде друкувати швидше. Ці всі умови  повинні виконуватися, щоб користувач отримав очікувану швидку реакцію у відповідь на введення символа і здобув максим досвіду при користуванні додатку.

Для тесту було введено ім’я Nikolay, потім стерто, а потім введено Nikita. В результаті відправлено 20 безперервних запитів, результат швидкості відпрацювання яких можна побачити на рисунку 5.13.

Загалом, середній час затрачений на виконання WebSocket запиту складає 80 мс, а HTTP запиту – 135 мс. Тобто, в середньому WebSocket запит відпрацьовує на 40 відсотків швидше, аніж HTTP.  Час, який нам показало WebSocket з’єднання є оптимальним, а також має запас часу, що ніколи не завадить.

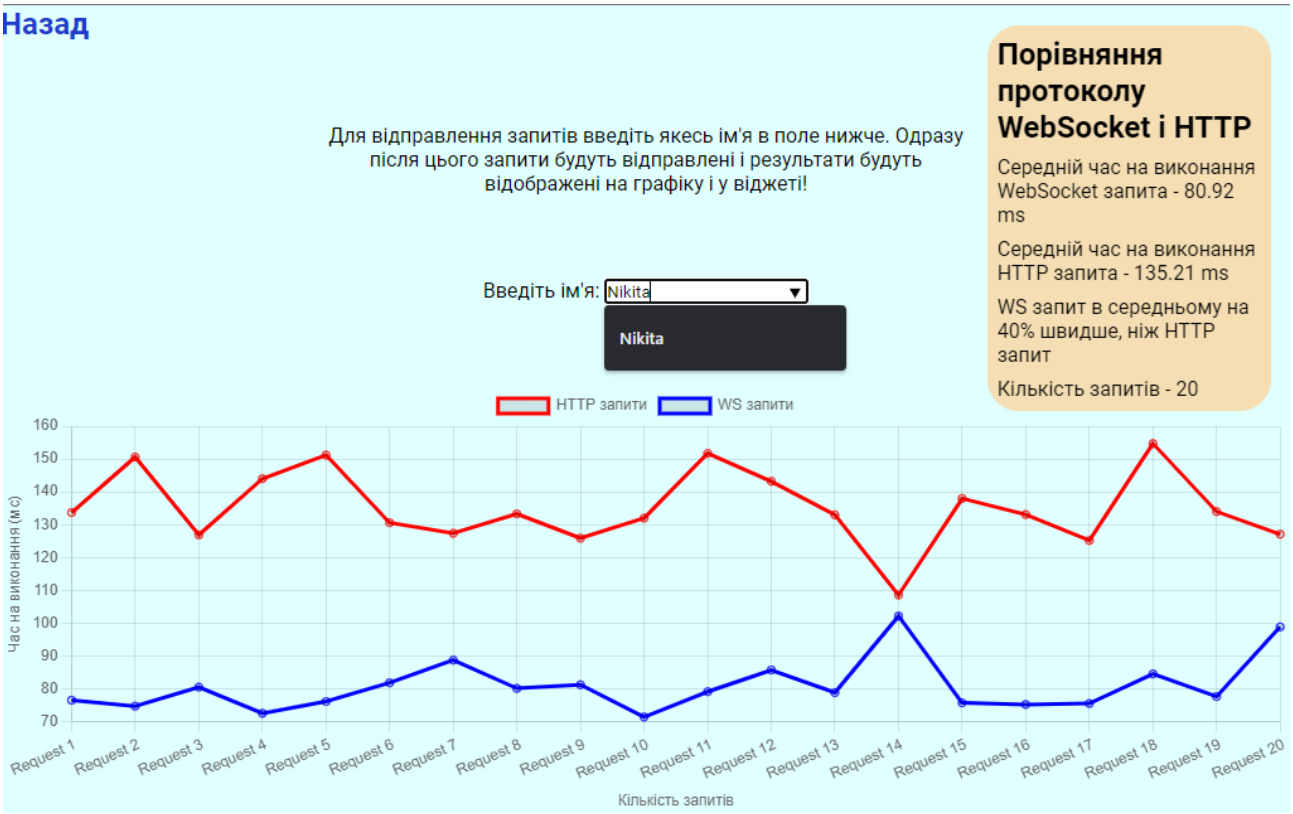


Рисунок 5.13 – Результати відпрацювання запитів

### 5.4 Аналіз отриманих результатів

В підсумку, отримані результати показали, що обраний підхід є достатньо ефективним, а також може мати гарний економічний ефект. Ми отримує гарні SEO показники, що гарантує гарну пошукову видачу, високу швидкість завантаження сторінки, що означає що користувач буде чекати менше часу, і буде більш задоволений додатком, а також він гарантує швидке відпрацювання запитів.

### 5.5 Висновки по розділу

В ході написання експериментального роздуло, було визначено мету та завдання експерименту, а також описано методику його проведення. Експеримент було проведено на системі, яка була розроблена у 4-му розділі. В підсумку було отримано результати, на основі який стало можливим провести аналіз роботи системи.

# ВИСНОВКИ

У ході виконання магістерської роботи було розроблено комп’ютерну систему на основі клієнт-серверної архітектури, яка працює за допомогою протоколу WebSocket та з використанням технології SSR.

Основні висновки і результати роботи полягають у наступному:

1. Проведено аналіз сучасних тенденцій та підходів в організації сучасних клієнт-серверних систем та веб-додатків. На основі результатів аналізу було сформульовано задачі на розробку системи.

2. Розглянуто і обрано технології для розробки клієнтського та серверного середовища додатку. Розібрано основні протоколи які використовуються в сучасній веб розробці та розглянуто нюанси роботи з ними. В результаті було обрано методи експериментальних досліджень, за допомогою яких стане можливо оцінити ефективність додатку.

3. Синтезовано комп’ютерну систему, визначено сфери застосування і головне призначення системи. Визначено функції, які повинні виконувати елементи системи і створено схему функціональної структури. Далі для кожного з елементів системи були висунуті вимоги до апаратного та програмного забезпечення і обране необхідне обладнання для функціонування системи. Далі створили структурну схему системи.

4. На основі алгоритму функціонування програми розроблено робочий прототип для тестування, який завантажений на віддалений сервер. Для полегшення взаємодії користувача з додатком, було створено інтерфейс, на якому можливо отримати інформацію про запити та швидкість їх виконання. Також представлено графік, за допомогою якого є можливість швидко оцінити результати і зробити висновки.

5. Визначено мету та завдання експерименту, а також описано методику його проведення. В підсумку було отримано результати, на основі який стало можливим провести аналіз роботи системи.

# ПЕРЕЛІК ПОСИЛАНЬ


1. Стандарт HTML [Електронний ресурс] URL: https://html.spec.whatwg.org (дата звернення 05.10.2021);
2. Специфікація ECMAScript [Електронний ресурс] URL: https://262.ecma-international.org/ (дата звернення 07.10.2021);
3. Специфікація CSS [Електронний ресурс] URL: https://www.w3.org/Style/CSS/specs.uk.html (дата звернення 08.10.2021);
4. Короткий посібник по пошуковій оптимізації [Електронний ресурс] URL: https://developers.google.com/search/docs/basics/get-started (дата звернення 13.10.2021);
5. Офіційний опис та документація React [Електронний ресурс] URL: https://reactjs.org/ (дата звернення 20.10.2021);
6. Офіційний опис та документація Angular [Електронний ресурс] URL: https://angular.io/docs (дата звернення 24.10.2021);
7. Офіційний опис та документація Vue.js [Електронний ресурс] URL: https://vuejs.org/v2/guide (дата звернення 26.10.2021);
8. Офіційний опис та документація node.js [Електронний ресурс] URL: https://nodejs.org/uk/docs (дата звернення 27.10.2021);
9. Інтернет-магазин Moyo - Сервер ARTLINE Business T13 v12 (T13v12) – купити в Києві | ціни і відгуки [Електронний ресурс] URL: https://www.moyo.ua/ua/server_artline_business_t13_v12_t13v12_/510423.html (дата звернення 23.11.2021);
10. ROZETKA | Сервер HPE DL20G9 G4560/1x8GB/noHDD 2LFF/B140i/290W (819785-271) [Електронний ресурс] URL: https://rozetka.com.ua/ua/hp_proliant_dl20_g9_819785_271/p109224600/?gclid=CjwKCAiAlfqOBhAeEiwAYi43F1w1jfB92H8JvZsaTaV90FfBulFxqqErvqIRbYvW_ssQ34PNMyUgXRoCnx8QAvD_BwE (дата звернення 24.11.2021);
11. Ноутбук Lenovo V15-ADA Iron Grey (82C70010RA) - придбати в інтернет-магазині Lenovo [Електронний ресурс] URL:

https://shop.lenovo.ua/uk/notebooks/noutbuk-lenovo-v15-ada-iron-grey-82c70010ra.html (дата звернення 25.11.2021);

12. Cisco 2911-SEC/K9 (CISCO2911-SEC/K9) - описание, цена и наличие в магазинах Вива-Телеком [Електронний ресурс] URL: https://viva-telecom.org/9310/cisco/2911-sec-k9/ (дата звернення 27.11.2021);
13. Cisco Catalyst 2960-X and 2960-XR Series Switches Data Sheet – Cisco [Електронний ресурс] URL: https://www.cisco.com/c/en/us/products/collateral/switches/catalyst-2960-x-series-switches/datasheet_c78-728232.html (дата звернення 30.11.2021).

# ДОДАТОК А

Текст програми комп’ютерної системи для клієнт-серверного додатку з протоколом Web Socket та технологією SSR.

Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

# ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ КОМП'ЮТЕРНОЇ СИСТЕМИ ДЛЯ КЛІЄНТ-СЕРВЕРНОГО ДОДАТКУ З ПРОТОКОЛОМ WEB SOCKET ТА ТЕХНОЛОГІЄЮ SSR

Текст програми

804.02070743.22010-01 12 01

Листів 17

2022

# АНОТАЦІЯ

Даний документ містить вихідний код програми для клієнт-серверного додатку з протоколом Web Socket та технологією SSR.

Текст програми реалізований на мові JavaScript, CSS і HTML.

Середовище розробки та налагодження –Visual Studio Code.

**ЗМІСТ**

index.js

```
const express = require('express');
const path = require('path');
const cors = require('cors');
const mongoose = require('mongoose');
const pug = require('pug');

const { Autocomplete } = require('./models/autocomplete-model');
const WebSocketApplication = require('./utils/webSocketApplication');
const { autocompleteRouter } = require('./routes/autocomplete-router');

mongoose.connect(
  'mongodb+srv://Nikolay:admin@dw.q39i5.mongodb.net/DW?retryWrites=true&w=majority'
);

const app = express();
const server = require('http').createServer(app);
const port = process.env.PORT || 3000;
const webSocetApplication = new WebSocketApplication(server);

webSocetApplication.addRouter(autocompleteRouter);

app.use(cors('*'));

app.use(express.static(path.resolve(__dirname, '../static')));

app.get('/api/autocomplete/:query', async function (req, res) {
  const regex = new RegExp(`^${req.params.query}`, 'i');
  const users = await Autocomplete.find({ name: regex }).exec();
  const html = pug.renderFile(
      path.join(__dirname, './templates/autocomplete-template.pug'),
      {
        results: users,
      }
    );

    res.json({
      html,
    });
});

app.get('/api/autocomplete', async function (req, res) {
  const data = await Autocomplete.find().exec();
  res.json(data);
});

server.listen(port, () => console.log(`Lisening on port :${port}`));
```

## autocomplete-controller.js

```
const pug = require('pug');
const path = require('path');

const { Autocomplete } = require('../models/autocomplete-model');

const getUsersByName = async (url, data, ws) => {
  let users;
  if (data.query) {
    const regex = new RegExp(`^${data.query}`, 'i');

    users = await Autocomplete.find({ name: regex });
  } else {
    users = await Autocomplete.find();
  }
  const html = pug.renderFile(
    path.join(__dirname, '../templates/autocomplete-template.pug'),
    {
      results: users,
    }
  );
  ws.send(
    JSON.stringify({ html, path: url, method: data.method, id: data.id })
  );
};

module.exports = {
  getUsersByName,
};
```

## autocomplete-model.js

```
const mongoose = require('mongoose');

const autocompleteSchema = new mongoose.Schema({
  name: String,
});

const Autocomplete = mongoose.model(
  'Autocomplete',
  autocompleteSchema,
  'autocomplete'
);

module.exports = {
  Autocomplete,
};
```

## autocomplete-router.js

```
const { Search } = require('../models/autocomplete-model');
const Router = require('../utils/router');
const { getUsersByName } = require('../controllers/autocomplete-controller');

const router = new Router();
const ENDPOINT = '/autocomplete/';

router.get(ENDPOINT, getUsersByName);

module.exports = {
  autocompleteRouter: router,
};
```

## autocomplete-template.pug

```
datalist(id='users')
    each result in results
        option(value=result.name)
```

## router.js

```
module.exports = class Router {
  constructor() {
    this.endpoints = {};
  }

  request(method = 'GET', path, handler) {
    if (!this.endpoints[path]) {
      this.endpoints[path] = {};
    }

    const endpoint = this.endpoints[path];

    if (endpoint[method]) return;

    endpoint[method] = handler;
  }

  get(path, handler) {
    this.request('GET', path, handler);
  }
  post(path, handler) {
    this.request('POST', path, handler);
  }
  put(path, handler) {
    this.request('PUT', path, handler);
  }
  delete(path, handler) {
    this.request('DELETE', path, handler);
```

```
  }
};
```

webSocketApplication.js

```
const http = require('http');
const EventEmitter = require('events');
const WebSocket = require('ws');
const path = require('path');
const console = require('console');

module.exports = class WebSocketApplication {
  constructor(server) {
    this.webSocketServer = this._createServer(server);
    this.emitter = new EventEmitter();
    this.middlewares = [];
  }

  use(middleware) {
    this.middlewares.push(middleware);
  }

  addRouter(router) {
    Object.keys(router.endpoints).forEach((path) => {
      const endpoint = router.endpoints[path];
      Object.keys(endpoint).forEach((method) => {
        this.emitter.on(this._getRouteMask(path, method), (...args) => {
          const handler = endpoint[method];
          handler(...args);
        });
      });
    });
  }

  _createServer(server) {
    const wss = new WebSocket.Server({ server: server });

    wss.on('connection', (ws, request) => {
      const path = request.url;

      ws.on('message', (data) => {
        const parsedData = JSON.parse(data);
        const method = parsedData.method;
        this.emitter.emit(
          this._getRouteMask(path, method),
          path,
          parsedData,
          ws
        );
      });
    });

    return wss;
```

```
  }

  _getRouteMask(path, method) {
    return `[${path}]:[${method}]`;
  }
};
```

index.html

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta
      name="description"
      content="Test website for compare http and websocket technologies"
    />
    <meta name="keywords" content="HTML, CSS, JavaScript, HTTP, WebSocket" />
    <meta name="author" content="Komarystyi Mykola" />
    <link rel="stylesheet" href="./style.css" />
    <link rel="prerender" href="./autocomplete/index.html" as="document" />
    <link rel="icon" type="image/x-icon" href="favicon.ico" />
    <title>Дипломний проект Комаристого М.М.</title>
  </head>
  <body>
    <main class="main">
      <section class="main__container">
        <h1 class="main__heading">
          Порівняння серверного рендерингу даних на WebSocket повідомленнях і
          HTTP запитах!
        </h1>
        <p class="main__description">
          Головна ціль цього додатку - показати переваги, які надає передача
          заздалегідь відрендерених даних за допомогою протоколу WebSocket.
          Продемонструвати ми це можем на прикладі частої проблеми, з якою
          стикаються більшість розробників - це автозаповнення.
        </p>
        <nav class="nav">
          <p class="nav__title">
            Натисніть на кнопку нижче, щоб перейти до приклада з автозаповненням
          </p>
          <ul class="nav__menu">
            <li class="nav__menu-item">
              <a href="/autocomplete" class="link">Автозаповнення</a>
            </li>
          </ul>
        </nav>
      </section>
    </main>
  </body>
```

```
</html>
```

style.css

```
@import
url('https://fonts.googleapis.com/css2?family=Roboto:ital,wght@0,300;0,400;0,700;1,300;1,400;1,700&d
isplay=swap');

:root {
  --spacing: 0.5rem;
  --spacing-2: calc(var(--spacing) * 2);
  --spacing-3: calc(var(--spacing) * 3);
  --spacing-4: calc(var(--spacing) * 4);

  --backgroud-color: #e0ffff;

  --link-color: rgb(33, 59, 204);
  --link-hover-effect: 0px 0px 8px rgb(131, 181, 247);
  --link-font-size: 1.5rem;
}

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
  list-style: none;
  text-decoration: none;
  font-family: Roboto, Arial, Helvetica, sans-serif;
}

html,
body {
  height: 100%;
}

.main {
  height: 100%;
  background-color: var(--backgroud-color);
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
}

.autocomplete__wrapper {
  height: 100%;
  background-color: var(--backgroud-color);
  display: flex;
  flex-direction: column;
  justify-content: space-between;
  align-items: center;
}
```

```
.main__container {
  display: flex;
  flex-direction: column;
  align-items: center;
  gap: 2rem;
}

.main__heading {
  width: 60%;
  text-align: center;
  line-height: 1.5;
}

.main__description {
  width: 50%;
  text-align: center;
}

.nav {
  width: 80%;
  display: flex;
  flex-direction: column;
  align-items: center;
}

.nav__title {
  font-size: 1.5rem;
  font-weight: bold;
  text-align: center;
}

.nav__title::first-letter {
  margin-left: var(--spacing-4);
}

.link {
  position: relative;
  color: var(--link-color);
  font-size: var(--link-font-size);
  font-weight: bold;
}

.link:hover {
  text-shadow: var(--link-hover-effect);
}

.link::before,
.link::after {
  content: '';
  position: absolute;
  bottom: 0;
```

```
  left: 50%;
  right: 50%;
  height: 1px;
  background: var(--link-color);
  transition: all ease-in-out 0.2s;
}

.link:hover::before {
  left: 0;
}

.link:hover::after {
  right: 0;
}

.chart {
  width: 100%;
  max-height: 80%;
  position: relative;
}

.header {
  width: 100%;
}

.info-box {
  position: fixed;
  top: var(--spacing-2);
  right: var(--spacing-2);
  width: 22%;
  border-radius: 25px;
  background-color: wheat;
  padding: var(--spacing);
  display: flex;
  flex-direction: column;
  gap: var(--spacing);
}
```

## robots.txt

```
User-agent: *
Allow: /
```

## sitemap.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>https://morning-taiga-63566.herokuapp.com/autocomplete</loc>
    <lastmod>2022-01-04</lastmod>
  </url>
</urlset>
```

index.html для сторінки автозаповнення

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta http-equiv="X-UA-Compatible" content="IE=edge" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta
      name="description"
      content="Test website for compare http and websocket technologies"
    />
    <meta name="keywords" content="HTML, CSS, JavaScript, HTTP, WebSocket" />
    <meta name="author" content="Komarystyi Mykola" />
    <title>Autocomplete</title>
    <link rel="stylesheet" href="../style.css" />
    <script src="https://cdn.jsdelivr.net/npm/chart.js" defer></script>
    <script defer type="module" src="../scripts/pages/autocomplete.js"></script>
  </head>
  <body>
    <main class="autocomplete__wrapper">
      <header class="header">
        <a href="/" class="link">Назад</a>
      </header>
      <p class="main__description">
        Для відправлення запитів введіть якесь ім'я в поле нижче. Одразу після
        цього запити будуть відправлені і результати будуть відображені на
        графіку і у віджеті!
      </p>
      <div class="autocomplete">
        <label for="autocomplete">Введіть ім'я: </label>
        <input class="autocomplete__input" name="autocomplete" list="users" />
      </div>

      <div class="info-box">
        <h2 class="info-box__title">Порівняння протоколу WebSocket і HTTP</h2>
        <p class="info-box__data">
          Середній час на виконання WebSocket запита -
          <span class="ws">0 ms</span>
        </p>
        <p class="info-box__data">
          Середній час на виконання HTTP запита - <span class="http">0 ms</span>
        </p>
        <p class="info-box__summary">
          WS запит в середньому на <span class="summary">0%</span> швидше, ніж
          HTTP запит
        </p>
        <p class="info-box__request-count">
          Кількість запитів - <span class="count">0</span>
        </p>
      </div>
```

```
    <div class="chart">
      <canvas id="myChart"></canvas>
    </div>
   </main>
  </body>
</html>
```

## autocomplete.js

```
import { SocketClient } from '../websocket.js';
import { chart } from '../chart.js';

const AUTOCOMPLETE_START = 'AUTOCOMPLETE_START';
const AUTOCOMPLETE_END = 'AUTOCOMPLETE_END';

const HTTP_REQUEST_START = 'HTTP_REQUEST_START';
const HTTP_REQUEST_END = 'HTTP_REQUEST_END';

let index = 1;

const socketClient = new SocketClient();

const updateWidget = () => {
  const http = document.querySelector('.http');
  const ws = document.querySelector('.ws');
  const summary = document.querySelector('.summary');
  const count = document.querySelector('.count');

  const httpData = chart.data.datasets[0].data;
  const httpAverage =
    httpData.reduce((acc, curr) => (acc += curr), 0) / httpData.length;

  const wsData = chart.data.datasets[1].data;
  const wsAverage =
    wsData.reduce((acc, curr) => (acc += curr), 0) / wsData.length;
  const summaryDiff = 100 - (wsAverage * 100) / httpAverage;

  http.textContent = `${httpAverage.toFixed(2)} ms`;
  ws.textContent = `${wsAverage.toFixed(2)} ms`;
  summary.textContent = `${summaryDiff.toFixed()}%`;
  count.textContent = httpData.length;
};

const fetchDataFromHTTP = async (query) => {
  performance.mark(HTTP_REQUEST_START + query);
  const resp = await fetch(`${location.origin}/api/autocomplete/${query}`);
  await resp.json();

  performance.mark(HTTP_REQUEST_END + query);
```

```
  performance.measure(
    'time to make request http',
    HTTP_REQUEST_START + query,
    HTTP_REQUEST_END + query
  );
  chart.data.datasets[0].data.push(
    performance.getEntriesByName('time to make request http')[0].duration
  );
  chart.update();
  updateWidget();
  performance.clearMeasures();
};

const fetchDataFromWebSocket = async (query) => {
  performance.mark(AUTOCOMPLETE_START + query);
  const data = await socketClient.sendMessage({
    method: 'GET',
    query,
  });

  const wrapper = document.createElement('div');
  const rootElem = document.querySelector('.autocomplete');
  wrapper.innerHTML = data.html;

  if (rootElem.lastElementChild.tagName !== 'INPUT') {
    rootElem.lastElementChild.remove();
  }

  rootElem.append(wrapper.firstChild);

  performance.mark(AUTOCOMPLETE_END + query);
  performance.measure(
    'time to make request',
    AUTOCOMPLETE_START + query,
    AUTOCOMPLETE_END + query
  );
  chart.data.datasets[1].data.push(
    performance.getEntriesByName('time to make request')[0].duration
  );
  chart.update();
  updateWidget();
  performance.clearMeasures();
};

document
  .querySelector('.autocomplete__input')
  .addEventListener('input', async (e) => {
    chart.data.labels.push(`Request ${index++}`);

    await fetchDataFromWebSocket(e.target.value);

    await fetchDataFromHTTP(e.target.value);
```

```
  });
```

chart.js

```
const config = {
  type: 'line',
  data: {
    labels: [],
    datasets: [
      {
        label: 'HTTP запити',
        data: [],
        fill: false,
        borderColor: 'red',
        tention: 0.1,
      },
      {
        label: 'WS запити',
        data: [],
        fill: false,
        borderColor: 'blue',
        tention: 0.1,
      },
    ],
  },
  options: {
    aspectRatio: 3,
    scales: {
      y: {
        title: {
          display: true,
          text: 'Час на виконання (мс)',
        },
      },
      x: {
        title: {
          display: true,
          text: 'Кількість запитів',
        },
      },
    },
  },
};

const chart = new Chart(document.getElementById('myChart'), config);

export { chart };
```

utils.js

```
export const EventEmmiter = {
  on: (event, listener) => document.addEventListener(event, listener),
  once: (event, listener) =>
```

```
    document.addEventListener(event, listener, { once: true }),
  off: (event, listener) => document.removeEventListener(event, listener),
  emit: (event, data) =>
    document.dispatchEvent(new CustomEvent(event, { detail: data })),
};
```

## websocket.js

```
import { EventEmmiter } from './utils.js';

export class SocketClient {
  constructor() {
    const socketProtocol = location.protocol === 'https:' ? 'wss:' : 'ws:';
    this.socketPathname = location.pathname === '/' ? '' : location.pathname;
    const socketUrl = `${socketProtocol}//${location.host}${this.socketPathname}`;
    this.socket = new WebSocket(socketUrl);

    this.socket.addEventListener('open', () => {
      this.socket.addEventListener('message', (event) => {
        const data = JSON.parse(event.data);
        EventEmmiter.emit(
          this._getRouteMask(data.path, data.method, data.id),
          data
        );
      });
    });
  }

  _getRouteMask(path, method, id) {
    return `[${path}]:[${method}]:[${id}]`;
  }

  sendMessage(message) {
    message.id = Date.now();
    const json = JSON.stringify(message);
    this.socket.send(json);

    return new Promise((resolve) => {
      EventEmmiter.once(
        this._getRouteMask(this.socketPathname, message.method, message.id),
        (event) => {
          resolve(event.detail);
        }
      );
    });
  }

  subscribe(callback) {
    this.socket.addEventListener('open', () => {
      this.socket.addEventListener('message', (event) => {
        const data = JSON.parse(event.data);
        callback(data, event);
```

```
      });
    });
  }

  onInit(callback) {
    this.socket.addEventListener('open', () => callback());
  }
}
```

## package.json

```
{
  "name": "diplomwork2v",
  "version": "1.0.0",
  "description": "",
  "scripts": {
    "start": "nodemon src/index.js"
  },
  "author": "",
  "license": "ISC",
  "dependencies": {
    "cors": "^2.8.5",
    "express": "^4.17.1",
    "mongodb": "^4.1.3",
    "mongoose": "^6.0.13",
    "nodemon": "^2.0.13",
    "pug": "^3.0.2",
    "ws": "^8.2.2"
  },
  "engines": {
    "node": "14.x"
  }
}
```