МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Харківський національний університет радіоелектроніки
Факультет Комп'ютерних наук
Кафедра Програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

другий (магістерський)
(рівень вищої освіти)

Дослідження архітектурних рішень та методів оптимізації для підвищення продуктивності додатків на Node.js і Vue.js

Виконав:
студент 2 курсу, групи ІПЗм-21-3
Стешко В. Ю.
(прізвище, ініціали)
Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми Освітньо-наукова
Керівник: доц. Ревенчук І. А.
(посада, прізвище, ініціали)

Допускається до захисту

Зав. кафедри, проф. _____ Дудар З.В.
2023 р.

Харківський національний університет радіоелектроніки

Факультет Комп'ютерних наук

Кафедра Програмної Інженерії

Рівень вищої освіти другий (магістерський)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва)

Тип програми освітньо-наукова програма

Освітня програма Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ:

Зав. кафедри _____
(підпис)

«_____»_____ 20__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студента Стешко Владислава Юрійовича
(прізвище, ім'я, по батькові)

1. Тема роботи «Дослідження архітектурних рішень та методів оптимізації для підвищення продуктивності додатків на Node.js і Vue.js»

затверджена наказом по університету від «29» березня 2023 р. № 302 Ст

2. Термін подання студентом роботи до екзаменаційної комісії «18» травня 2023 р.
3. Вихідні дані до роботи *середовище розробки WebStorm, інструмент вимірювання продуктивності Lighthouse, пояснювальна записка*.
4. Перелік питань, що потрібно опрацювати в роботі мета роботи, аналіз предметної галузі і постановка задачі, дослідження методів та архітектурних рішень, опис програмних рішень, опис реалізації ПЗ, тестування ПЗ.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	09.02.23 – 13.02.23	виконано
2	Дослідження методів та архітектурних рішень	14.02.23 – 27.02.23	виконано
3	Аналіз програмних рішень і проектування архітектури	28.02.23 – 07.03.23	виконано
4	Створення коду веб-системи	08.03.23 – 16.04.23	виконано
5	Тестування і налагодження додатку	17.04.23 – 21.04.23	виконано
6	Підготовка пояснювальної записки	22.04.23 – 27.04.23	виконано
7	Підготовка презентації та доповіді	28.04.23 - 02.05.23	виконано
8	Перевірка на плагіат та нормоконтроль	03.05.23	виконано
9	Рецензування	08.05.23	виконано
10	Попередній захист	10.05.23	виконано
11	Занесення диплому в електронний архів	10.05.23	виконано
12	Допуск до захисту у зав. кафедри	16.05.23	виконано

Дата видачі завдання 29 березня 2023 р.

Студент _____ Стешко В. Ю.
(підпис) (прізвище, ініціали)

Керівник роботи _____ доц. Ревенчук І. А.
(підпис) (посада, прізвище, ініціали)

РЕФЕРАТ / ABSTRACT

Кваліфікаційна робота магістра містить: 75 с., 38 рис., 1 табл., 14 джер.

АРХІТЕКТУРА, МЕТОДИ ОПТИМІЗАЦІЇ, ПРОДУКТИВНІСТЬ, ВЕБ-СИСТЕМА, NODE JS, VUE.

Об’єкт дослідження – веб-орієнтована система яка буде успадковувати архітектурні рішення та кращі практики оптимізації.

Мета роботи – підвищення продуктивності та масштабованості додатків на Node.js та Vue.js.

Методи розробки базуються на таких технологіях, як середовище розробки WebStorm 2022, мови програмування JavaScript та TypeScript, та фреймворки: Nest.js, Vue.js, СУБД PostgreSQL.

В результаті роботи, було досліджено різні варіації побудови архітектури та методи оптимізації продуктивності, спроектовано архітектуру, як на сервері, так і на клієнті та програмно реалізовано веб-систему, у двох версіях, одна з яких успадковує кращі практики, а друга йде в розріз ними.

ARCHITECTURE, OPTIMIZATION METHODS, PRODUCTIVITY, WEB SYSTEM, NODE JS, VUE.

The object of research is a web-oriented system that will inherit best optimization practices and architectural solutions.

The purpose of the work improving the performance and scalability of applications on Node.js and Vue.js.

Development methods are based on such technologies as WebStorm 2022 development environment, JavaScript and TypeScript programming languages, and frameworks: Nest.js, Vue.js, PostgreSQL database.

As a result, different variations of architecture building and performance optimization methods were explored, the architecture was designed both on the server and on the client, and the web system was programmatically implemented in two versions,

one of which inherits the best practices, and the second goes against them.

Умови публікації пояснювальної записки

Я,

_____
(прізвище, ім’я, по батькові)

студент(ка) групи _____здобувач вищої освіти на другому (магістерському) рівні

кафедра_____програмної інженерії_____,
(повна назва кафедри)

заявляю: моя кваліфікаційна робота на тему

_____,
(назва роботи)

що буде представлена до ЕК для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElArKhNURE. Всі запозичення з друкованих та електронних джерел мають відповідні посилання.

Я ознайомлений (а) з діючим положенням «Про протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів.

ЗМІСТ

ВСТУП

На сучасному етапі розвитку World-wide-web, важливим є завдання підвищення продуктивності веб-додатку, масштабованості та надійності, за рахунок побудови коректної архітектури та використання способів / методів оптимізації.

Коли розробники веб-застосунків розробляють веб-додаток, швидкість, з якою користувачі можуть взаємодіяти з ним, є одним із показників продуктивності, які вони використовують для забезпечення високої якості свого додатку.

Швидкість веб-додатку – надзвичайно важливий фактор успіху продукту. Це вже не розкіш, а потреба. Якщо необхідно додати більше елементів на свій веб-сайт, треба переконатися, що ви не жертвуєте швидкістю свого веб-додатку, інакше ви можете втратити більше, ніж отримати. Незалежно від того, чи хочете ви отримати більше потенційних клієнтів, продажів або скоротити кількість телефонних дзвінків до служби підтримки клієнтів, оптимізація може бути використана для того, щоб зробити ваш додаток більш ефективним для досягнення цих цілей.

Продуктивність веб-додатку – це об'єктивні вимірювання та відчуття користувача, пов'язані із завантаженням і роботою програми. Продуктивність - це про те, як довго веб-додаток завантажується, стає інтерактивним та чуйним, про те, як плавно відбувається взаємодія з контентом.

Відсутність моніторингу продуктивності веб-додатку та приділення уваги, дій спрямованих на її покращення взагалі може призвести безпосередньо до збою, скажемо, у «Чорну п'ятницю», якщо ви не готові впоратися зі збільшенням кількості одночасних користувачів.

Підтримка плавного досвіду. Коли справа доходить до швидкості веб-застосунку, швидкість важлива не тільки для конверсій, але і для створення враження плавної взаємодії. В результаті існує психологічний вимір, який впливає на те, як користувачі сприймають ваш веб-додаток.

Google Research передбачає, що той веб-додаток, який забезпечує завантаження сторінки менш ніж за сто мілісекунд, це те як довго ваша потилична частка, або сховище пам'яті, може зберігати інформацію у вигляді сенсорної пам'яті. Обмежуючи час завантаження сторінки менш ніж сто мілісекундами, створюється ілюзія того, що веб-додаток швидко реагує на дії користувача. Якщо веб-застосунок не відповідає протягом цільового часу в сто мілісекунд, зв'язок між дією та реакцією переривається і користувачі помічають цю затримку. З іншого боку, якщо завантаження сторінки займає 5+ секунд, користувачі будуть розчаровані та нетерплячі, і їм буде складно зосередитися на додатку і вони швидше за все покинуть його. У випадках, коли такої швидкості досягти не вдається, потрібно повідомити користувача з використанням підказок.

Вибір технологій: Node.js та Vue.js, для дослідження, обумовлений особистими уподобаннями. Ці технології, також є досить популярними, як серед компаній, так і розробників.

За будь-яким гарним продуктом, стоїть якісна архітектура та зроблена робота пов'язана з його оптимізацією. Кожен додаток може, якщо не відразу, то через якийсь час зіткнутися з проблемами пов'язаними з продуктивністю. Ці проблеми виникають у наступних ситуаціях:

- додаток розростається і стає більшим (сотні користувальницьких сценаріїв, десятки екранів та ін.);
- додаток починає оперувати великою кількістю даних;
- з'являється велика кількість користувачів.

Все це може призвести до втрати продуктивності, особливо якщо додаток не є оптимізованим і як наслідок, користувачі віддадуть перевагу використанню продукту конкурента. Використання неякісної архітектури застосування негативно впливає на його подальше масштабування і як наслідок втрата розуміння взаємозв'язку компонентів системи. Підтримувати такий додаток стає дуже важко. Тому проектування архітектури та приділення уваги оптимізації є запорукою гарного продукту.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ ТА ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз предметної галузі

Говорячи про архітектуру, можна сказати, що загальноприйнятого терміну немає. Але найточніше, її можна описати так: «Архітектура програмного забезпечення містить у собі низку важливих рішень про організацію програмної системи, серед яких вибір структурних елементів та їх інтерфейсів, що становлять і об'єднують систему в єдине ціле; поведінка, що забезпечується спільною роботою цих елементів; організацію цих структурних та поведінкових елементів у більші підсистеми, а також архітектурний стиль, якого дотримується ця організація». Вибір архітектури ПЗ також стосується функціональності, зручності використання, стійкості, продуктивності, повторного використання, зрозумілості, економічних та технологічних обмежень, естетичного сприйняття та пошуку компромісів [1].

Якщо вдаватися до практики, більшість розробників розуміє, який код є гарним, та який ні. Архітектура яка є гарно спроектованою та реалізованою, це вигідна архітектура, що дозволяє спростити та зробити більш ефективним, процес розробки та підтримки додатку. Додаток з грамотно побудованою архітектурою легше тестувати, змінювати, розширювати та розуміти. Виходячи з цього, можна сформулювати низку критеріїв [2]:

- гнучкість додатку. Будь-який додаток, з часом, доводиться змінювати і як наслідок додаються або змінюються вимоги. Чим швидше та зручніше можна внести зміни до існуючого функціоналу, чим менше проблем та помилок це викличе – тим гнучкіша та конкурентноспроможніша система. Зміна однієї частини системи не повинно впливати на інші її частини;
- ефективність додатку. Незалежно від того, в яких умовах знаходиться, додаток, він повинен виконувати свої функції та поставлені завдання. До цього відносяться наступні характеристики: надійність, безпека, продуктивність, масштабованість;

- розширюваність додатку. Має бути можливість додавати до системи нові сутності та функції, не порушуючи її основної структури. На початковому етапі, має сенс закладати лише основний функціонал, але при цьому архітектура має дозволяти легко нарощувати додатковий функціонал. Вимога, щоб архітектура системи мала гнучкість і розширюваність сформульовано у вигляді окремого принципу «Принцип відкритості/закритості». Програмні сутності (класи, модулі, функції тощо) мають бути відкритими для розширення, але закритими для модифікації;
- тестованість додатку. Залежно від того, як написаний код, залежить наскільки легше він буде тестуватися, міститиме менше помилок і надійніше працюватиме. Це один із найважливіших критеріїв, що дозволяє оцінити його якість;
- супроводжуваність додатку. Додаток не повинен містити дублювання, мати добре оформлений код, структуру та зрозумілу документацію.

При оптимізації веб-додатку, для підвищення його продуктивності, враховується ряд метрик, серед яких: час до отримання першого байта даних від сервера, час до першого відтворення, час до відтворення основної частини контенту, час до початку інтерактивності елементів сторінки. Зазвичай веб-додаток ділиться на дві частини: серверна та клієнтська.

До серверної частини можна віднести:

- кешування та буфіризація запитів до БД;
- оптимізація запитів до БД, використання індексів;
- мінімізація обсягу даних, що передаються (API) та ін.

До клієнтської частини:

- зменшення вкладеності DOM дерева;
- WebWorkers для складних тривалих завдань;
- ліниве завантаження даних / модулів / маршрутів / зображень;
- використання сучасних методів стиснення (webp) та ін.

1.2 Актуальність роботи

Архітектура та кроки, які необхідно зробити для оптимізації системи, завжди були і будуть актуальними. Будь-який додаток, має свою архітектуру, але далеко не кожен додаток, дотримується правил, щодо побудови гарної архітектури, те саме стосується і оптимізації.

Для поганого дизайну можна виділити такі критерії:

- при внесенні змін ламаються інші частини системи;
- код важко використовувати повторно в будь-яких інших додатках, так як виникають складності, з тим, щоб дістати його з поточного додатка, через його заплутаність;
- його важко змінити, оскільки будь-яка зміна впливає на дуже велику кількість інших частин системи.

Через не оптимізований додаток, страждає як мінімум, його продуктивність, що істотно може позначитися, на використанні сервісу користувачами. Продуктивність – це ще один показник того, чому необхідно оптимізувати додаток.

1.3 Постановка задачі

Метою даного дослідження є аналіз архітектурних шаблонів, методів, технологій, які використовуються, аналіз програмних рішень щодо реалізації в предметній галузі та програмна реалізація, яка попередньо тестується та явно відображає архітектуру, продуктивність та масштабованість.

Система має включати великі обсяги даних, відео, аудіо та інші елементи. Для системи будує проведене тестування продуктивності з допомогою інструменту Lighthouse.

2 АНАЛІЗ МЕТОДІВ ТА АРХІТЕКТУРНИХ РІШЕНЬ

2.1 Аналіз архітектурних рішень

Архітектура ПЗ – набір підходів для організації програмно-апаратного комплексу. Опис компонентів системи та взаємозв'язків між ними. Архітектура включає підходи: обмеження, правила і евристики, яким треба слідувати при написанні коду.

Грамотна архітектура допомагає спроектувати та розвивати систему так, щоб її було простіше та зручніше розширювати та змінювати. Тому:

- якщо спілкування між модулями регламентовано, їхню реалізацію простіше замінити іншою;
- якщо спілкування із зовнішнім світом регламентовано, то менше шансів для витоку даних;
- якщо код розділено грамотно, програму простіше тестувати;
- якщо код організований зрозуміло, йде менше часу на додавання нових можливостей та пошук багів у старих;
- якщо архітектура широко відома, занурення у проект проходить швидше [3].

Можна умовно розділити архітектурні підходи щодо їх цілей та зони дії.

Частина підходів розподіляє відповідальність між модулями [4]. Вони визначають, які модулі та за що відповідатимуть. Ці підходи називаються архітектурними патернами, а саме: MVC, MVVM, MVP.

MVC (Model-View-Controller) – схема поділу даних додатку та логіки на три компоненти: модель, уявлення та контролер. Виходячі з цього, модифікація кожного компонента може здійснюватися незалежно.

- Модель (Model) надає дані та реагує на команди контролера, змінюючи свій стан.
- Подання (View) відповідає за відображення даних моделі користувача, реагуючи на зміни моделі.

– Контролер (Controller) інтерпретує дії користувача, сповіщаючи модель про необхідність змін.

Патерн відображено на рисунку 2.1.

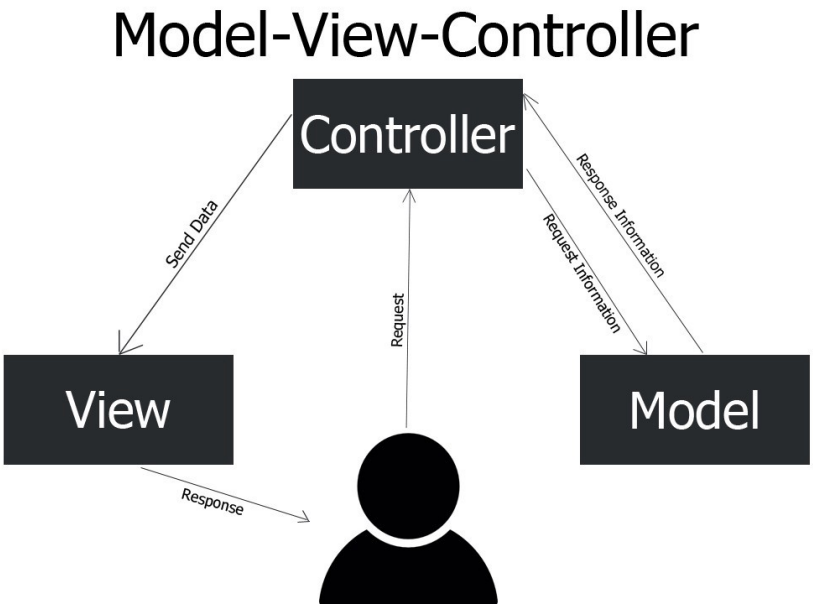

Рисунок 2.1 – MVC

MVVM (Model-View-ViewModel) – дозволяє відокремити візуальну частину від логіки. Цей патерн є архітектурним, тобто він задає загальну архітектуру додатку. MVVM складається з трьох компонентів: моделі (Model), моделі уявлення (ViewModel) та уявлення (View). Патерн можна побачити на рисунку 2.2.

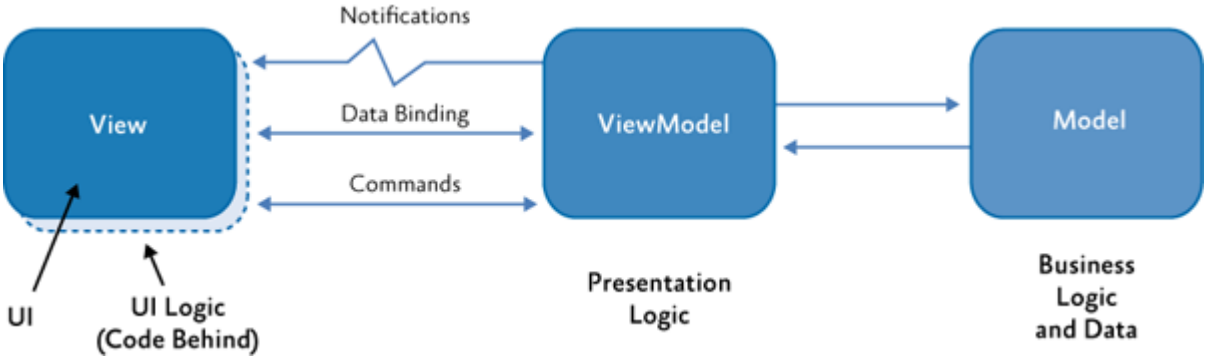

Рисунок 2.2 – MVVM

MVP (Model-View-Presenter) – патерн розробки інтерфейсу користувача. MVP-шаблон є похідним від MVC, але має дещо інший підхід. Основна відмінність

– presenter не сильно пов'язане з моделлю (model). Місце контролера займає презентер. Патерн MVP, можна побачити на рисунку 2.3.

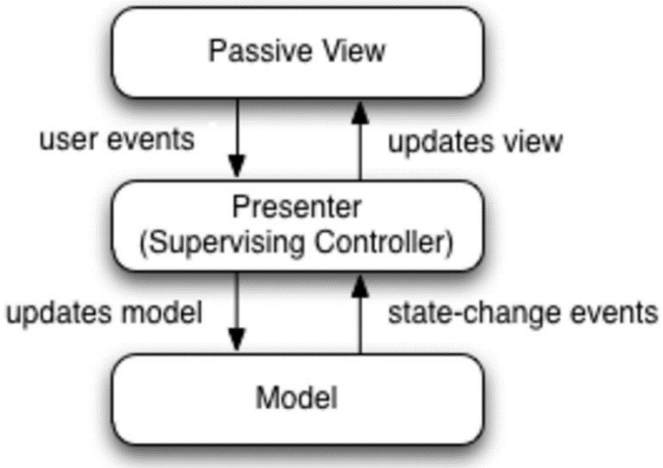

Рисунок 2.3 – MVP

Презентер забирає на себе всю логіку обробки даних, оновлення подання та обробки команд користувача. Подання в цьому випадку пасивне: воно не робить нічого, крім відображення даних так, як йому скаже презентатор. Якщо в MVC уявлення могло брати форматування виведення на себе, то в MVP за це теж відповідатиме презентер.

Інші визначають, наскільки кожен із модулів близький до бізнес-логіки. Таким підходам важливо, яка частина коду безпосередньо займається завданням програми, а яка – інфраструктурними завданнями. Наприклад, у програмі обробки фотографій бізнес-логікою були б функції фільтрів, а інфраструктурними завданнями – звернення до API камери телефону [4].

Треті керують потоками даних у додатку. Вони визначають, як модулі спілкуються одне з одним: безпосередньо, опосередковано чи з допомогою спеціальних сервісів типу шини подій. В цілому, потоки даних можуть бути організовані величезною кількістю способів, але найчастіше на практиці у frontend зустрічаються два: односпрямований та двоспрямований.

В односпрямованому потоці даних кожна частина програми від іншої частини може отримати дані, або передати. Напрямок такого потоку не змінюється.

Дані в двоспрямованому потоці можуть передаватися між частинами програми в обидві сторони. Найчастіше це використовується для зв'язування моделі та подання, щоб оновлення, наприклад, тексту в полі введення одразу оновило дані в моделі – це називається двонаправленим зв'язуванням даних. Фреймворки, які використовують двонаправлене зв'язування, часто реактивні – тобто застосовують зміни миттєво не тільки до UI, а й до даних, що обчислюються. Vue є одним з таких фреймворків.

Наступні підходи визначають компонування програми. Буде це одна велика програма (моноліт) або набір декількох менших програм (мікросервісів). Загалом можна виділити три види архітектури веб-додатків: моноліт, мікросервіси, серверлес.

Моноліт – це архітектурне рішення, в якому всі компоненти та модулі тісно пов'язані між собою і залежать один від одного [5]. Побачити приклад архітектури моноліт, можна на рисунку 2.4.

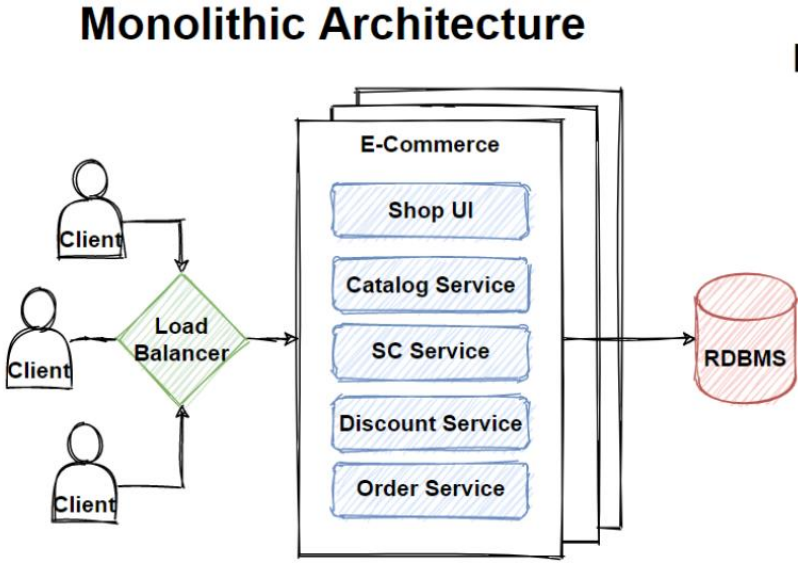

Рисунок 2.4 – Монолітна архітектура

Можна виділити наступні переваги та недоліки.

Переваги архітектури:

– простота розгортання. Моноліти швидко і відносно просто розгортати, тому що моноліт зазвичай має єдину точку входу;

– розробка. Розробка моноліту зазвичай відбувається швидко, тому що всі компоненти та модулі знаходяться в єдиній кодовій базі;
– налагодження. Налагодження моноліту дуже спрощено за рахунок того, що все поряд, і є можливість відстежити всі ланки виконання коду.

Недоліки архітектури:

– масштабування. Моноліти масштабуються тільки повністю, тобто якщо навантаження зростає тільки на один модуль, то не вийде масштабувати тільки цей модуль, потрібно масштабувати весь моноліт;
– надійність. Якщо моноліт виходить з ладу, то виходить увесь цілком;
– недостатня гнучкість.

Мікросервіси – це архітектурне рішення, яке базується на розподілі модулів на окремі системи, які спілкуються між собою за допомогою повідомлень [5]. Побачити приклад архітектури можна на рисунку 2.5.

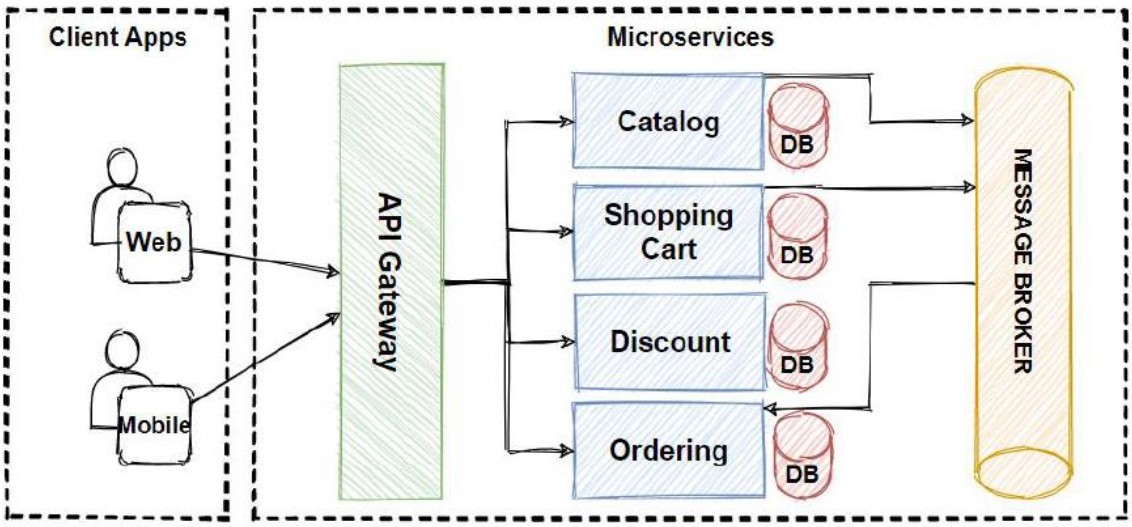

Рисунок 2.5 – Мікросервісна архітектура

Мікросервісна архітектура не зовсім підходить під середні і тим більше маленькі проекти. Серед переваг та недоліків, можна виділити наступні:

Переваги:

– гнучкість. Мікросервіси дуже гнучкі за рахунок того, що кожен сервіс є самостійною системою;

– масштабування. На відміну від моноліту, можна масштабувати лише певну частину системи, оскільки вона самостійна;
– гнучкість технологій. У мікросервісах кожна з підсистем може бути реалізована будь-якою мовою програмування та за допомогою будь-яких технологій;
– надійність. Вихід з ладу однієї з підсистем не ламає всю систему загалом.

Недоліки:

– мікросервіси складно розробляти, тому що потрібно проводити кілька підсистем та налагодити взаємодію між ними;
– налагодження. Мікросервіси складно налагоджувати, тому що потрібно знайти, який сервіс зламався і чому;
– розгортання. Початкове розгортання дуже непросте, і додавання нових сервісів вимагає налаштування ключових частин проекту.

Серверлес – це архітектурне рішення, яке фокусується на розробці замість розгортання та взаємодії між сервісами [5]. Це альтернатива мікросервісам, яка автоматизує все розгортання завдяки хмарним технологіям (рисунок 2.6).

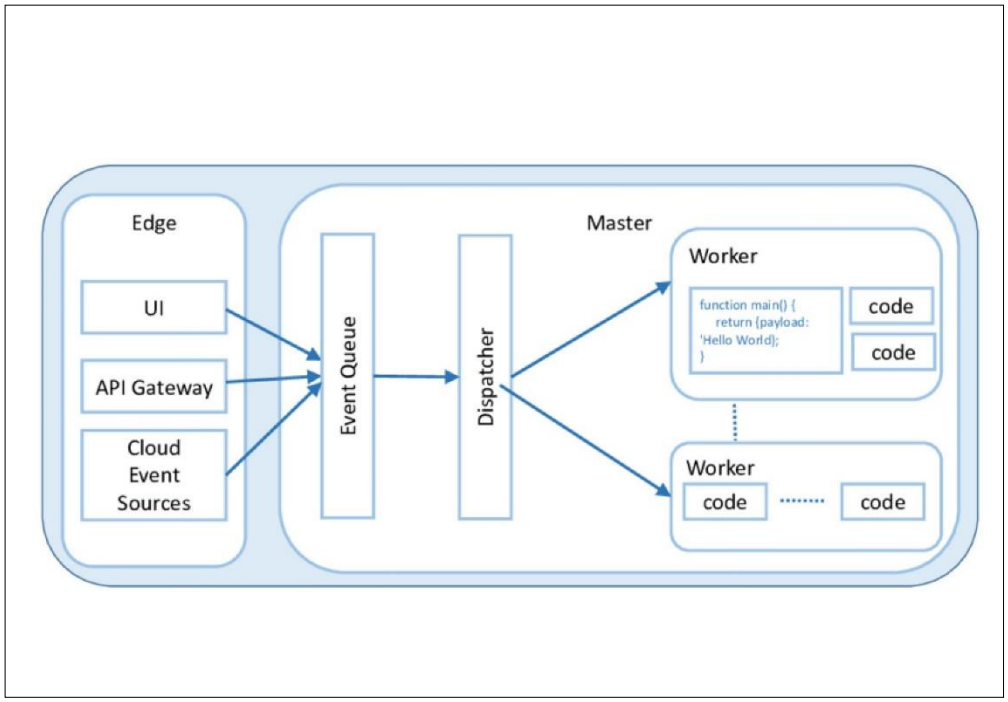

Рисунок 2.6 – Serverless архітектура

Серед переваг та недоліків, можна виділити наступні:

Переваги:

- гнучкість. Серверлес дуже гнучкий за рахунок відокремленості одного модуля від іншого;
- абстракція від операційної системи. Хмара сама вирішить, яка ОС потрібна та як її потрібно налаштувати;
- легкий поріг входу;
- надійність. При правильній побудові, надійність така ж як у мікросервісів.

Недоліки:

- гнучкість. Це як перевага, так і недолік, тому що розробник обмежений вплив на масштабування, все вирішує хмара;
- налагодження ускладнене взаємодією між компонентами;
- Vendor Lock. Кожна хмара працює за власними правилами, тому перехід з однієї хмари на іншу майже неможливий.

Всі ці архітектури є верхньорівневими і кожна гарна у своїй галузі. Для реалізації програми, можна вибрати будь-яку з перерахованих вище архітектур, але це може бути неефективним або в плані розробки, або в плані експлуатації. Тому необхідно спиратися на вимоги до додатку і на переваги і недоліки кожної з цих архітектур.

2.2 Аналіз методів оптимізації

На даний момент, проблеми з продуктивністю є одні з найбільш поширених. Їх рішення вимагає дотримання низки правил, сформованих в результаті вивчення побудови веб-додатків. Глобально проблеми з продуктивністю веб-застосунків можна розділити на дві категорії: передача даних і runtime.

- передача даних – завантаження будь-яких ресурсів, необхідніх роботи додатку;
- runtime – робота додатку, рендеринг і обробка введення користувача.

Кожна з цих категорій містить у собі нюанси, що відрізняють якісні програми від неякісних. Найпопулярніші метрики продуктивності веб-додатків:

- TTFB – час до отримання, першого байту;
- FCP – час до першого відображення контенту;
- FMP – час до першої значущого відображення;
- TTI – час до початку інтерактивності елементів сторінки.

Серед іншого можна також виділити поширені причини проблем з продуктивністю:

- погано написаний код може призвести до багатьох проблем з веб-додатками, включаючи неефективні алгоритми, витоку пам'яті та взаємоблокування програм. Старі версії програмного забезпечення або інтегровані застарілі системи також можуть знижувати продуктивність;
- неоптимізовані бази даних. Оптимізована база даних забезпечує найвищий рівень безпеки та продуктивності, у той час як неоптимізована база даних може зруйнувати додаток. Відсутні індекси уповільнюють виконання SQL-запитів, що може призвести до зниження працездатності всього сайту;
- DNS, брандмауер та мережеве підключення. DNS-запити становлять більшу частину веб-трафіку. Ось чому проблема з DNS може викликати стільки проблем, не дозволяючи відвідувачам отримати доступ до вашого сайту та призводячи до помилок, помилок 404 та неправильних шляхів. Так само мережеве підключення та ефективність брандмауера мають вирішальне значення для доступу та продуктивності [6];
- зовнішні сервіси. Проблема використання зовнішніх сервісів у тому, що вони знаходяться поза контролем;
- повільний сервер;
- поганий розподіл навантаження. Поганий розподіл навантаження може призвести до збільшення часу відгуку через неправильний розподіл нових відвідувачів сайту між серверами [6].

Серед загальних методів оптимізації веб-додатків, можна виділити наступні.

Передача даних

CDN – допоможе прискорити завантаження, для теориторіально розподілених клієнтів.

Resource prioritization – прискорення завантаження сторінки з допомогою правильної стратегії завантаження ресурсів. Браузери дозволяють встановлювати пріоритети для різних типів ресурсів і завантажувати раніше те, що є важливим для першого малювання.

Static compression – це алгоритм стиснення, який зменшить вагу статики та, відповідно, збільшить швидкість завантаження.

WebP vs Png vs Jgp. Webp - відмінна альтернатива Png. Крім меншої ваги зображень, Webp практично не поступається якістю і має швидкий час завантаження [7].

TTFB. Час до отримання першого байта сторінки додатку після надсилання запиту з боку клієнта. Є важливою метрикою. Це комплексний показник, що в першу чергу залежить від того, які операції виконуються на сервері під час обробки запиту. Великий час відповіді може бути пов'язаний з десятками факторів: логіка програми, повільна робота з базою даних, маршрутизація, програмна платформа, бібліотеки, нестача процесорної потужності або пам'яті.

HTTP2 може прискорити завантаження сторінки за рахунок мультиплексування або стиснення заголовків [8].

Час виконання (Runtime)

requestIdleCallback – функція, яка дозволяє виконати код у вільний час наприкінці кадру (фрейму/тіка) або коли користувач неактивний.

requestAnimationFrame дозволяє правильно запланувати анімацію та максимально збільшити шанси на рендеринг 60 кадрів на секунду.

Маніпуляції з DOM – дорогі, виконувати їх потрібно обережно та осмислено. Vue, передбачає оптимізовану роботу з DOM завдяки використанню більш легковагої версії – VirtualDOM.

Віртуальний скрол – розумний спосіб не рендерувати великі списки, а генерувати їх під час прокручування. Дозволяє споживати менше пам'яті та полегшити скролінг списків.

60 FPS by pointer-events: none – за допомогою цієї можливості можна досягти 60 FPS при скролі сторінки. Працює за таким принципом: на час прокручування відключаються всі обробники миші.

Збірка

Одним з найкращих збирачів проектів є Webpack. Він аналізує модулі програми, створює граф залежностей, потім збирає модулі у правильному порядку. Має можливості оптимізації збірки, що у свою чергу підвищує продуктивність.

Розділення коду – розділивши код на чанки, можна оптимізувати перше завантаження.

Мініфікація – зменшення розміру фінального білду, за рахунок видалення, непотрібних символів з HTML, JS, CSS, які не потрібні для відображення сторінки: коментарі, відступи.

Мертвий код – код, що не використовується, з фінального білда прибирається тим самим прискорюючи завантаження сторінок.

Архітектура

Ліниве завантаження модулів / маршрутів – інструмент, який є у всіх популярних фреймворках та бібліотеках. Дозволяє «ліниво» підвантажувати шматки функціональності програми.

Кешування файлів – дозволяє зберігати файли в браузері та не завантажувати їх щоразу з сервера.

Ліниве завантаження картинок та відео.

Використання індексів.

Тестування швидкості додатку.

Крім загальних методів, можна виділити низку тих, що використовують технологію Vue:

– функціональні компоненти. Припустимо, що є простий, невеликий компонент, завдання якого відображати той чи інший тег, залежно від

переданого значення. Цей компонент можна оптимізувати, додавши функціональний атрибут – functional. Функціональний компонент компілюється у просту функцію і не має локального стану;

- keep-alive - компонент обгортка над роутером. Всі компоненти в роутері будуть створюватися та знищуватись при переході між маршрутами. Якщо компоненти важкі, можна отримати підвисання інтерфейсу на момент перемикання. Vue надає можливість не знищувати, а кешувати і перевикористовувати, тим самим зберігаючи стан компонента. Така оптимізація призведе до збільшення споживання пам'яті, оскільки Vue необхідно підтримувати їх живими. Такий підхід не слід застосовувати бездумно [9];
- ліниве завантаження компонентів. При традиційному імпорті компонентів, дочірній компонент завантажується одразу, як браузер дійде до інструкції import. Однак якщо компонент великий, то є сенс завантажувати його асинхронно. Vue надає цю можливість із коробки. Він завантажить компонент тільки тоді, коли це знадобиться і відобразить, коли буде готовий. При використанні збирача, наприклад Webpack, дочірній компонент буде винесений в окремий файл, що зменшить вагу файлу при початковому завантаженні. Ліниве завантаження, можна також застосувати і для компонентів, що використовуються для маршрутів. Кожен із маршрутів буде завантажений тоді, коли буде запрошений;
- vuex. Робота із комітами. Коміти, на відміну від дій, є синхронними. Якщо припустимо, є необхідність зберегти великий масив даних, то його обробка заблокує інтерфейс на час роботи. Для вирішення проблеми можна розбити масив на частини і додавати їх по черзі, даючи браузеру час на відображення. З таким підходом можна додати індикатор завантаження, що покращить UX;
- вимкнення реактивності. У сховищі лежить масив об'єктів, з великим рівнем вкладеності і Vue згідно з поведінкою, виконає рекурсивний обхід, всіх вкладених полів. Якщо додаток побудований таким чином, що

залежить від об'єкта верхнього рівня вкладеності та не посилається на реактивні дані на кілька рівнів нижче, то цю реактивність можна прибрати, полегшивши роботу Vue.

Оптимізація якості взаємодії з користувачем – ключ до довгострокового успіху будь-якого додатку. Важливість оптимізації сайту неможливо переоцінити. Це допомагає забезпечити безперебійну та швидку роботу веб-сайту та допомагає задовольнити потреби клієнтів.

Web Vitals – це ініціатива Google, мета якої надати єдиний посібник із сигналів якості, необхідним для забезпечення гарної взаємодії з користувачем. Мета ініціативи Web Vitals – спростити ситуацію, що склалася, і допомогти зосередитися на найбільш важливих показниках Core Web Vitals (рисунок 2.7).

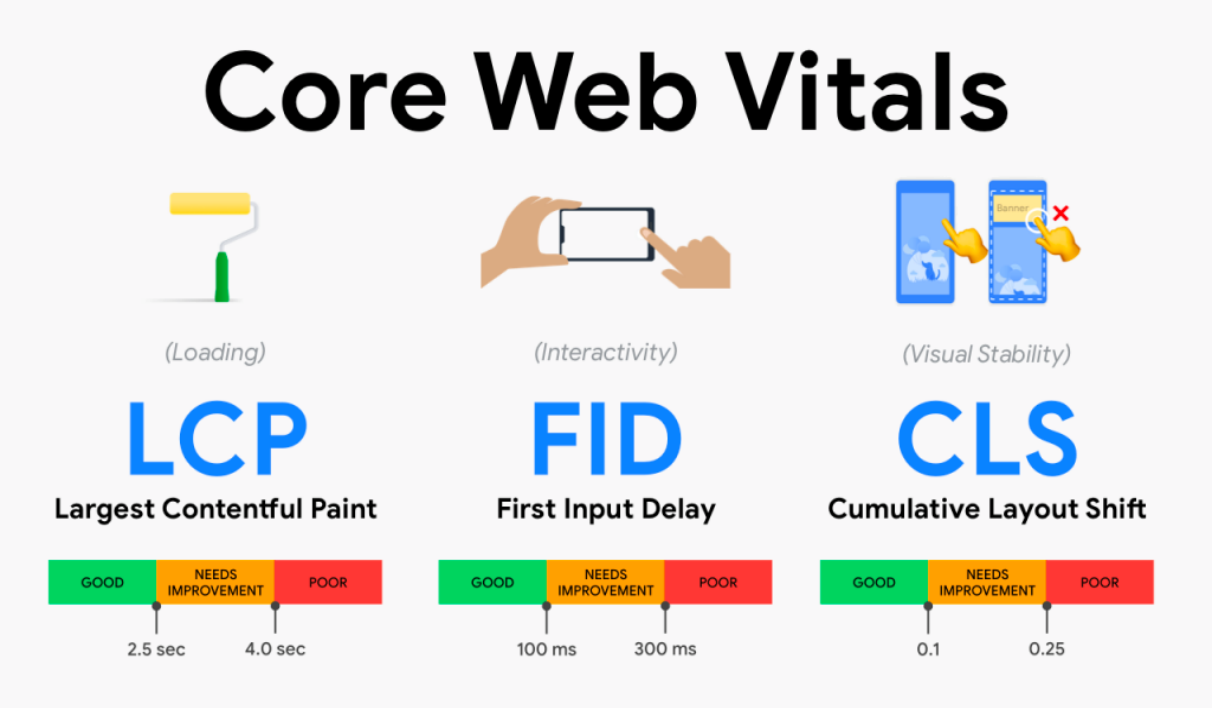

Рисунок 2.7 – Метрики Core Web Vitals

Core Web Vitals – це частина метриків Web Vitals, які використовуються для оцінки веб-сторінок і включені у всі інструменти Google. Кожен показник Core Web Vitals є окремим аспектом досвіду взаємодії користувача з сайтом, що вимірюється в польових умовах і відображає реальні дії з досягнення критично важливого результату, орієнтованого на користувача.

Поточний набір за 2023 рік фокусується на трьох аспектах взаємодії з користувачем: швидкості завантаження сторінок сайту, інтерактивності та візуальної стабільності та включає показники (і їх відповідні граничні значення), див. 2.7.

Метрики:

- LCP (Largest Contentful Paint) – швидкість завантаження основного контенту: вимірює продуктивність завантаження. Щоб забезпечити зручність роботи користувачів, показник LCP повинен бути в межах 2,5 секунд від початку завантаження сторінки.
- FID (First Input Delay) – час очікування до першої взаємодії із контентом: вимірює інтерактивність. Щоб забезпечити зручність роботи користувачів, показник FID у сторінок не повинен перевищувати 100 мілісекунд.
- CLS (Cumulative Layout Shift) – сукупне усунення макета: вимірює візуальну стабільність. Щоб забезпечити зручність роботи користувачів, показник CLS не повинен перевищувати 0,1.

Хоча показники Core Web Vitals є критично важливими для розуміння та забезпечення гарної взаємодії з користувачем, є й інші важливі метрики. Вони часто служать проміжними або додатковими показниками для Core Web Vitals, і допомагають повніше охопити взаємодію з користувачем або діагностувати конкретну проблему.

Наприклад, метрики Time to First Byte (TTFB): час до першого байта і First Contentful Paint (FCP). Перше відображення контенту є критично важливими аспектами завантаження та корисні для діагностики проблем з LCP (повільний час відгуку сервера або ресурси, що блокують рендеринг).

Аналогічно, такі показники, як Total Blocking Time (TBT): загальний час блокування та Time to Interactive (TTI), час до інтерактивності є вкрай важливими лабораторними метриками для виявлення та діагностики потенційних проблем з інтерактивністю, які можуть вплинути на FID. Однак вони не входять до набору Core Web Vitals, тому що не піддаються виміру в польових умовах і не

відображають результати, орієнтовані на користувача. Метрики можна побачити на рисунку 2.8.

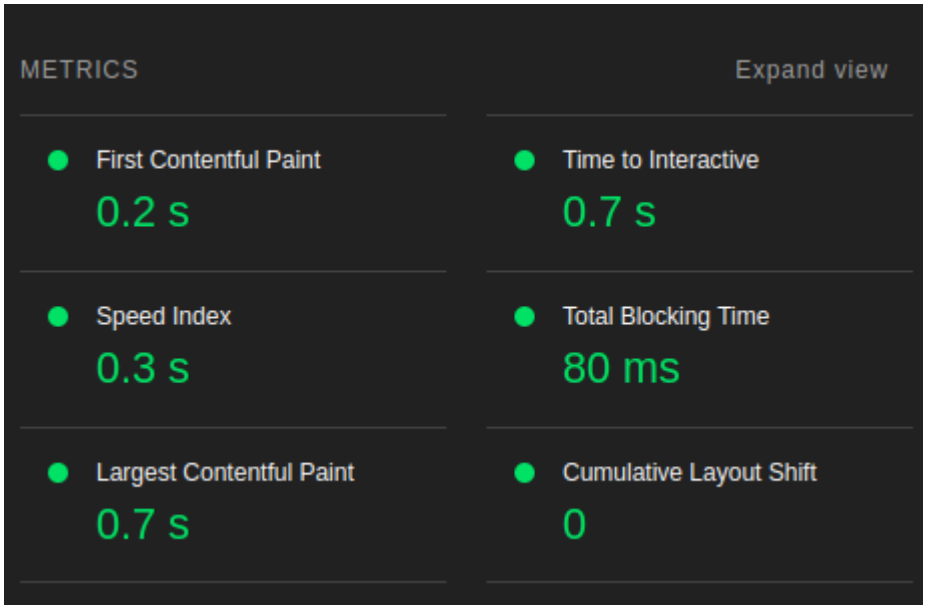

Рисунок 2.8 – Інші метрики

Інструменти, подібні до Lighthouse, які завантажують сторінки в змодельованому середовищі без участі користувача, не можуть виміряти FID (немає введення користувача). Тим не менш, показник Total Blocking Time (TBT) (загальний час блокування) піддається лабораторним вимірюванням і є відмінною метрикою для FID. Оптимізація продуктивності, що призводить до поліпшення TBT у лабораторних умовах, має покращити FID у польових умовах.

Серед іншого, можна виділити низку проблем, які виникають із вищепереліченими метриками та дії щодо їх оптимізації (табл. 1).

Таблиця 1 – Вплив дій на метрики

Дії	Вплив на метрики			
	FCP	TBT	LCP	CLS
Правильний розмір зображень	-	-	+	-

Кінець таблиці 1

Дії	Вплив на метрики			
	FCP	TBT	LCP	CLS
Закадрові зображення	-	-	+	-
Завантажені ліниво LCP зображення	-	-	+	-
Зображення не мають визначеної ширини та висоти	-	-	-	+
Зображення у форматах нового покоління	-	-	+	+
Видимий текст, під час завантаження веб-шрифту	+	-	+	+
Розділення коду та мініфікація	+	+	+	-
Блокування рендерингу	+	-	+	-
Зменшення початкового часу відповіді	+	+	+	+
Кешування	+	+	+	-

Правильний розмір зображень (Properly size images). Це трапляється, якщо зображення, яке обслуговує браузер, перевищує розмір свого контейнера. В ідеалі сторінка ніколи не повинна показувати зображення, розмір яких перевищує версію,

відображену на екрані користувача. Все, що перевищує цей розмір, призводить до марної витрати байтів і уповільнює час завантаження сторінки. Стратегії:

- основна стратегія показу зображень відповідного розміру називається «адаптивними зображеннями». З адаптивними зображеннями створюється кілька версій кожного зображення, а потім вказується, яку версію використовувати у своєму HTML або CSS за допомогою медіа-запитів, розмірів вікна перегляду тощо;
- інша стратегія полягає у використанні векторних форматів зображень, таких як SVG. За допомогою обмеженої кількості коду, зображення можна масштабувати до будь-якого розміру;
- існують також CDN зображення, які дозволяють створювати кілька версій, коли зображення завантажується або виконується запит.

Закадрові зображення (Defer offscreen images). Зображення поза екраном, які не є критичними для початкового досвіду користувача, можуть завантажуватися відкладено. Це техніка, яка відкладає завантаження зображення, доки користувач не прокрутить його, що може зменшити мережеву боротьбу за критичні ресурси та в деяких випадках покращити LCP.

LCP зображення було завантажено ліниво (Largest Contentful Paint image was lazily loaded). Відкладене завантаження сьогодні є стандартною практикою та одним із найпростіших способів пришвидшити роботу. За допомогою відкладеного завантаження зображень можна гарантувати, що вони не будуть завантажуватися, поки вони не знадобляться, коли користувач прокручує сторінку вниз. Важливі зображення у верхній частині сторінки, наприклад зображення LCP, не слід завантажувати ліниво. Це може призвести до затримки завантаження зображення, який є елементом LCP [10].

Елементи зображення не мають чіткої ширини та висоти (Image elements do not have explicit width and height). Коли надходить запит на сторінку, браузер не знає пропорцій зображення, доки не почнеться завантаження. З цієї причини сторінка візуалізується, потім запитує зображення та з’ясовує його висоту. Контейнер для зображення потім змінить розмір, щоб розмістити це зображення, і у результаті

отримаємо зміщення макета, сприяючи сукупному зсуву макета. Щоб обмежити сукупний зсув макета, викликаний зображеннями без розмірів, необхідно додати атрибути розміру ширини та висоти до зображень і відео елементів, або використати контейнер співвідношення сторін (aspect-ratio). Цей підхід гарантує, що браузер може виділити правильний обсяг місця в документі під час завантаження зображення.

Подавати зображення у форматах нового покоління (Serve images in next-gen formats). Це попередження виникає, коли використовуються зображення у традиційному форматі PNG/JPEG, замість форматів нового покоління WebP/AVIF, які забезпечують кращі функції стиснення - є більш оптимізовані.

- AVIF: Формат файлів зображень AV1 (AVIF) — це формат зображень із відкритим кодом для зберігання нерухомих та анімованих зображень. AVIF підтримує стиснення з втратами та без втрат для отримання високоякісних фотографій і стискає набагато краще, ніж у більшості популярних сьогодні форматів в Інтернеті (JPEG, PNG WebP тощо). AVIF дозволяє заощадити до 50% від загального розміру файлу;
- WebP також є сучасним форматом зображень, який забезпечує чудове стиснення без втрат і втрат для зображень. Зображення WebP зі стисненням з втратами може бути на 25-34% менше, ніж те ж саме зображення у форматі JPEG.

Переконайтеся, що текст залишається видимим під час завантаження веб-шрифту (Ensure text remains visible during webfont load). Шрифти часто являють собою великі файли з повільним часом завантаження. Забезпечення того, щоб текст залишався видимим під час завантаження веб-шрифту, позначає, що текст на веб-сторінці ніколи не можна приховувати, навіть якщо веб-шрифт ще не завантажено. Щоб усунути цю прогалину при завантаженні, деякі браузери приховують текст, доки не завантажаться шрифти, викликаючи спалах невидимого тексту (FOIT).

FOIT (спалах невидимого тексту) виникає, якщо браузер надто довго чекає, щоб завантажити веб-шрифт. У наведеному нижче прикладі фактичний текст

завантажено, і його можна виділити, але він залишається невидимим, оскільки веб-шрифт ще не повністю завантажено (рисунок 2.9).

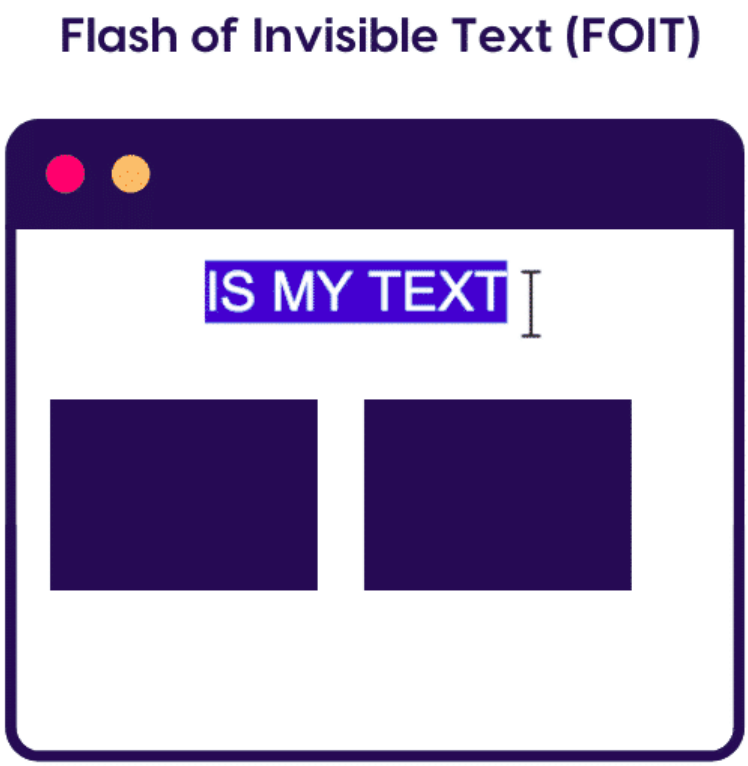

Рисунок 2.9 – FOIT

Текст (веб-шрифти, значки шрифтів тощо) повинні завжди залишатися видимими під час завантаження. Це впливає як на перше малювання вмісту (FCP), так і на малювання найбільшого вмісту (LCP), а також на взаємодію з користувачем. Це також покращує оцінку сукупного зміщення макета (CLS), у якому необхідно уникати раптових зрушень макету.

Усунути ресурси, які блокують рендеринг (Eliminate render-blocking resources). Мета полягає в тому, щоб зменшити вплив URL-адрес на отримання ресурсів, які блокують рендерінг, шляхом включення критичних ресурсів, відкладення некритичних ресурсів і видалення всього, що не використовується. Один із способів відстежити, що не використовується, є критичним, або ні – це інструмент Google Chrome Coverage (рисунок 2.10) [11].

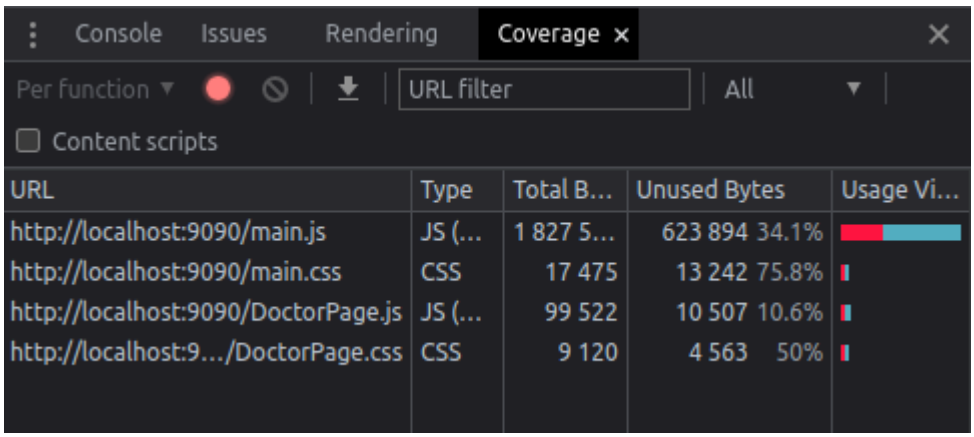

Рисунок 2.10 – Google Chrome Coverage

Стилі у файлах CSS і код у файлах JavaScript позначені двома кольорами:

– зелений (критично). Код, який має вирішальне значення для основних функцій сторінки; стилі, необхідні для першого фарбування;
– червоний (некритичний). Код не використовується в основних функціях сторінки; стилі, які застосовуються до вмісту, який не відображається одразу.

Якщо є код, який блокує рендерінг і не є критичнім, то можна зберегти його в окрему URL адресу (script) і позначити атрибутами async, defer. Це позначає, що JavaScript завантажується під час синтаксичного аналізу HTML і буде виконано після завершення завантаження сторінки (коли парсер завершить роботу).

Подібно до вбудовування коду в тег <script>, вбудовані критичні стилі, необхідні для першого малювання всередині блоку <style> на початку сторінки HTML. Потім можна завантажити решту стилів асинхронно за допомогою preload link. Ще одним способом усунення ресурсів, які блокують рендерінг, є видалення невикористаного css (minify css). Все це, має вплив, як на LCP так і на FCP.

Зменшити початковий час відповіді сервера (Reduce initial server response time). Виникає коли час до першого байта (TTFB), який також називають часом очікування, перевищує 600 мс. Високий TTFB виникає, коли:

– кешування налаштовано неправильно або веб-сайт не завантажує кеш;
– поганий постачальник веб-хостингу, особливо якщо використовується спільний, а не виділений хостинг;

– хостинг-сервіс та/або налаштування сервера можуть бути неправильно оптимізовані;
– на веб-сайті багато сторінок, зображень, плагінів, розширень і вони не структуровані належним чином, це може значно сповільнити додаток.

Це також має великий вплив на FCP і LCP.

У той час як сервісні працівники та PWA стають стандартами сучасних веб-додатків, кешування ресурсів стало більш складним, ніж будь-коли.

Важливо розуміти схему рівнів, якими проходить запит для отримання ресурсу. Схему можна побачити рисунку 2.11.

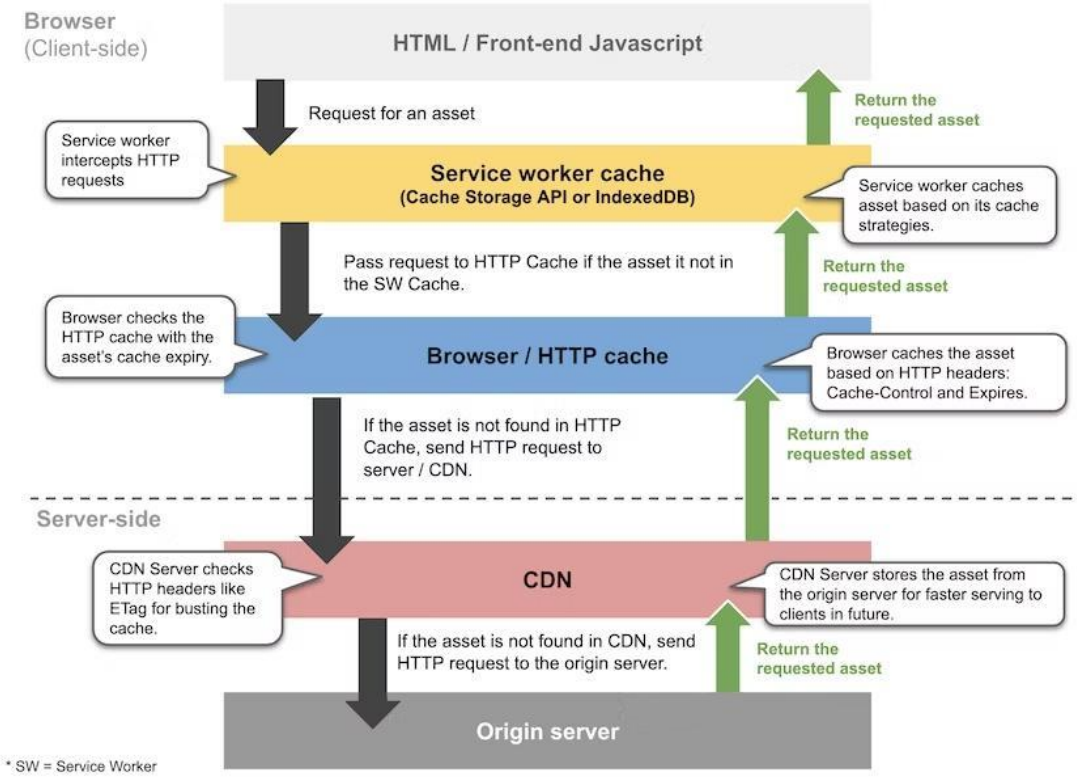

Рисунок 2.11 – Схема отримання ресурсу

На високому рівні браузер дотримується наведеного нижче порядку кешування, коли запитує ресурс:

– Service Worker та CacheApi. Service Worker перевіряє, чи є ресурс у своєму кеші, і вирішує, чи повертати сам ресурс на основі стратегій кешування. Це не відбувається автоматично. Для цього необхідно створити обробник

подій (fetch) у сервісному воркері та перехоплювати мережеві запити, щоб запити обслуговувалися з кешу сервісного воркера, а не з мережі.

- HTTP-кеш (також відомий як кеш-пам’ять браузера). Якщо ресурс знайдено в HTTP-кеші та термін його дії ще не минув, браузер автоматично використовує ресурс із HTTP-кешу.
- Server-side. Якщо нічого не знайдено в кеші Service Worker або кеші HTTP, браузер переходить до мережі, щоб запитати ресурс. Якщо ресурс не кешується в CDN, запит має повернутися до початкового сервера.

Також треба розуміти, що інтерфейс Cache у сумісництві з Service Worker, не пов'язаний з HTTP кешем. Інтерфейс Cache – це механізм кешування, повністю окремий від кешу HTTP. Яку б конфігурацію Cache-Control ви не використовували для впливу на HTTP-кеш, це не впливає на те, які активи зберігатимуться в інтерфейсі Cache. Це допомагає думати про кеш браузера як про багаторівневий. Кеш HTTP – це низькорівневий кеш, що керується парами ключ-значення з директивами, вираженими в заголовках HTTP. Навпаки, інтерфейс Cache це високорівневий кеш, керований API JavaScript. Це забезпечує більшу гнучкість, ніж при використанні відносно простих пар ключ-значення HTTP, і є половиною того, що робить можливими стратегії кешування [12].

Переваги кешування з використанням Service Worker:

- більше пам’яті та місця для зберігання джерела. Браузер розподіляє ресурси кешу HTTP для кожного джерела. Іншими словами, якщо є кілька субдоменів, усі вони мають спільний HTTP-кеш. Немає гарантії, що вміст джерела/домену залишатиметься в HTTP-кеші протягом тривалого часу. Наприклад, користувач може очистити кеш, очистивши вручну інтерфейс налаштувань веб-переглядача або запустивши апаратне перезавантаження сторінки. З кеш-пам’яттю Service Worker ви маєте набагато більшу ймовірність того, що ваш кешований вміст залишиться в кеші;
- вища гнучкість з нестабільними мережами або офлайн. Із HTTP-кешем є лише двійковий вибір: або ресурс кешувати, або ні. За допомогою кешування Service Worker за стратегією «Stale-while-revalidate», можна

обслуговувати ресурс із кешу, а потім «у фоновому режимі» повторно запитувати його з мережі та оновлювати запис у кеші, тим самим підтримувати актуальний стан ресурсу, або запропонувати повний автономний доступ.

Враховуючи складність поєднання сценаріїв кешування, неможливо розробити одне правило, яке охоплює всі випадки. Логіка кешування Service Worker не повинна узгоджуватися з логікою закінчення терміну дії кешування HTTP. Якщо можливо, треба використовувати логіку довшого терміну дії в сервісному воркері, щоб надати йому більше контролю.

Кешування HTTP все ще відіграє важливу роль, але воно ненадійне, коли мережа нестабільна або не працює.

Таким чином використовуючи всі перераховані вище практики, можна досягти значних показників Core Web Vitals і значно покращити продуктивність додатку.

3 АНАЛІЗ ПРОГРАМНИХ РІШЕНЬ

3.1 Опис інструментарію

У цій роботі дослідження проводиться в рамках середи виконання JS – Node.js і його фреймворку Nest.js, а також фреймворку на стороні клієнта Vue.js. Дані технології були обрані, тому що є одними з найпопулярніших, для створення веб-додатків. Мови програмування: JavaScript та TypeScript.

3.1.1 Backend

Node.js – середовище виконання JavaScript, засноване на JavaScript движку V8, який вміє скомпілювати JavaScript код у машинний код. Є кросплатформним середовищем з відкритим вихідним кодом для розробки серверних і мережевих додатків, яке використовує керовану подіями, неблокуючу модель введення-виведення, що робить її простою і ефективною для додатків з інтенсивним використанням даних в реальному часі, що працюють через розподілені пристрої.

Нижче представлено низку можливостей, які надає Node.js:

- масштабованість. Забезпечує широку масштабованість програм. Node.js, будучи однопотоковим, здатний обробляти безліч одночасних підключень з високою пропускною здатністю;
- швидкість. Неблокуюче виконання потоків робить Node.js ще швидше та ефективніше;
- пакети. Доступний широкий набір пакетів Node.js з відкритим вихідним кодом, які можуть спростити вашу роботу. Сьогодні в екосистемі NPM налічується понад мільйон пакетів;
- потужний бекенд – Node.js написаний мовами C і C++, що робить його швидким і додає такі функції, як підтримка мережі;

– підтримуваність – Node.js є простим вибором для розробників, оскільки як інтерфейсом, так і сервером можна керувати за допомогою JavaScript як однієї мови.

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

– головний потік не блокуватиметься операціями введення/виводу;
– сервер продовжуватиме обслуговувати запити;
– робота з асинхронним кодом.

Node.js має безліч фреймворків, серед яких можна виділити такі як: express.js, nest.js, koa.js, adonis.js.

Було проведено невеликий порівняльний аналіз.

3.1.1.1 Express.js

Express.js є гнучким та мінімалістичним фреймворком додатків. Він не побудований навколо конкретних компонентів, що дає розробникам можливість експериментувати. Вони отримують блискавичне налаштування та чистий досвід роботи з JS.

Express може бути використаний, для: багатосторінкових, односторінкових, гібридних додатків.

Характеристики Express.js:

– дозволяє розробникам швидше створювати RESTful API;
– швидка розробка на стороні сервера;
– підтримка NoSQL баз даних з коробки;
– підтримує архітектуру MVC;
– дозволяє динамічно відображати HTML-сторінки на основі передачі аргументів до шаблонів.

Express.js ідеально підходить для швидкого створення веб-додатків та сервісів, оскільки має доступні інструменти генерації API.

3.1.1.2 Nest.js

Nest – серверна платформа, створена для підтримки продуктивності розробників та полегшення їхнього життя. Повністю написаний на TypeScript та підтримує JavaScript. Легко тестується і містить великий функціонал з «коробки».

У Nest під капотом крутиться express. Nest значно розширює його функціональність, додає декларативності, а також допомагає розробнику будувати додаток відповідно до кращих архітектурних практик.

Nest.js можна бути використаний для:

- написання чистого та повторно використовуваного коду;
- написання коду за допомогою конструкцій високого рівня: інтерцептори, фільтри, пайпи;
- написання масштабованих та тестованих додатків.

Характеристики Nest.js:

- дозволяє використовувати чистий JS;
- легко розширюється – може використовуватись з іншими бібліотеками;
- поєднує в собі риси функціонального програмування, ООП та функціонального реактивного програмування;
- надає API фреймворків, які допомагають використовувати різні сторонні модулі, доступні для будь-якої платформи;
- має докладну та підтримувану документацію.

Nest.js використовується для написання масштабованих, тестованих додатків. Він забезпечує правильний баланс структури та гнучкості, щоб ефективно керувати кодом.

3.1.1.3 Koa.js

Koa.js – відкритий веб-фреймворк, написаний розробниками Express.js. За допомогою Koa вони прагнули створити меншу та надійнішу платформу для веб-додатків та API. Він пропонує широкий спектр ефективних методів для прискорення процесу створення серверів [13].

Характеристики Koa.js:

- сучасне та перспективне рішення;
- вбудований catchall помилок, що запобігають збоям;
- використання контекстного об'єкта, який містить об'єкти запиту та відповіді.

Використовується для створення серверів, маршрутів, обробки відповідей та помилок.

3.1.1.4 Adonis.js

MVC-фреймворк для Node.js який може працювати на всіх ОС. Він забезпечує стабільну екосистему для написання серверних веб-додатків. Ідеально підходить для розробників Laravel, які намагаються перейти на Node.js.

Характеристики Adonis.js:

- API та система аутентифікації на основі сеансів;
- валідація вхідних даних користувача;
- відмінна система безпеки;
- розширюваний макет програми;
- потужний ORM, що допомагає створювати безпечні SQL-запити.

У результаті, при проведенні порівняльного аналізу був обраний фреймворк Nest.js. Він використовує правильні архітектурні підходи і поряд з техніками

оптимізації, дозволить на виході отримати масштабований, структурований і продуктивний backend додаток.

3.1.2 Frontend

На даний момент існує достатня кількість фреймворків на клієнті. Але варто виділити три найбільш використовувані: React, Vue, Angular.

Vue.js – це прогресивний, легкий фреймворк для створення реактивних веб-інтерфейсів користувача. Vue розширює стандартні HTML і CSS, створюючи набір потужних інструментів для створення зовнішнього вигляду інтерактивних веб-додатків. Цей фреймворк, який підходить для створення односторінкових додатків (Single-Page Applications) будь якої складності, якщо використовувати його спільно з сучасними інструментами та додатковими бібліотеками. Хоча Vue не пов'язаний із патерном MVVM (Model-View-ViewModel), його дизайн був частково натхненний ним. У Vue робота зосереджена на рівні ViewModel, забезпечуючи реактивну синхронізацію між моделлю та шарами перегляду через двостороннє зв’язування даних[14].

React – це бібліотека JavaScript, яка використовується для створення інтерфейсу користувача. ReactJs вводить спеціальний код JavaScript, крок препроцесора, що додає XML-подібний синтаксис JavaScript. Це допомагає вести складання коду, що читається, зберігати його в один перевірений файл. Можливість відкидати HTML у функції рендерингу без конкатенації рядків. Завдяки використанню певного JSX Transformer HTML перетворюється на функції.

Angular – повномасштабний front-end фреймворк, що надає власні інструменти для всіх пов'язаних з розробкою веб-додатків функцій.

Важливою особливістю Angular є те, що він створювався для роботи саме з великими програмами. При цьому Vue та React підходять для проектів будь-якої складності [14].

Щодо оптимізації, як React, так і Vue і Angular досить швидкі [14].

Кожен фреймворк має різну архітектуру, продуктивність у різних сценаріях, екосистему та інструменти. Всі три фреймворки можуть використовуватися практично взаємозамінно для створення компонентних frontend-додатків з розширеними можливостями інтерфейсу користувача.

Виходячи з переваг, як клієнтський фреймворк / бібліотека, був обраний Vue.js.

3.2 Проектування архітектури ПЗ

Архітектура веб-додатку представляє макет із усіма компонентами програмного забезпечення (такими як бази даних, програми та проміжне програмне забезпечення) і те, як вони взаємодіють один з одним.

Зазвичай архітектура веб-додатків складається з 3 основних компонентів (3 рівня):

- клієнтська частина (Presentation Layer / Client layer) – це ключовий компонент, який взаємодіє з користувачем, отримує вхідні дані та керує логікою презентації, одночасно контролюючи взаємодію користувача з додатком. За потреби також перевіряються введені користувачем дані;
- серверна частина (Application Layer / Business Layer) – обробляє бізнес-логіку та запити користувачів, направляючи запити до потрібного компонента та керуючи всіма операціями додатку. Він може запускати та контролювати запити від широкого кола клієнтів;
- сервер бази даних (Data Layer) надає необхідні дані додатку. Він виконує завдання, пов’язані з даними. У багаторівневій архітектурі сервери баз даних можуть керувати бізнес-логікою за допомогою збережених процедур.

Система матиме монолітну клієнт-серверну архітектуру. Спосіб взаємодії фронтенду з бекендом відбуватиметься через API, за протоколом HTTP. Як архітектурний стиль взаємодії використаний REST, концепція, парадигма, яка визначає набір правил, згідно з якими системи обмінюються даними з можливістю масштабування. Для відображення клієнт-серверної архітектури, було прийнято застосувати діаграму розгортання. Діаграму можна побачити на рисунку 3.1.

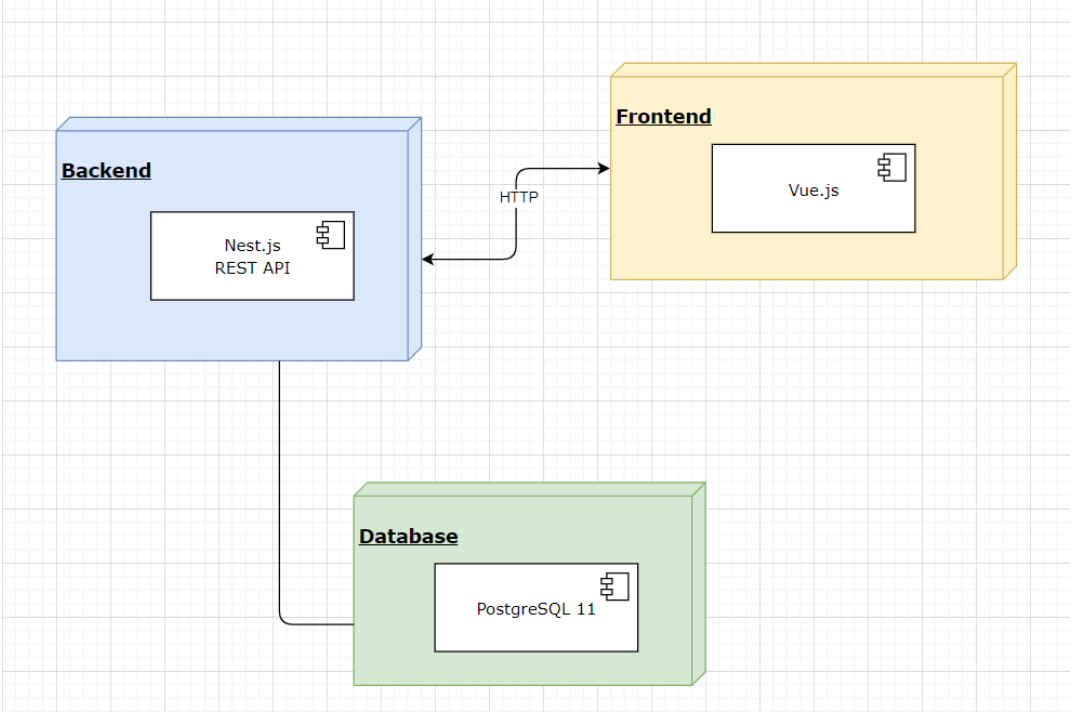

Рисунок 3.1 – Діаграма розгортання

Архітектура серверної частини поділена на сервіси в рамках одного моноліту. Кожен сервіс, буде містити: dto, modules, controllers, providers, models, interfaces, middlewares, pipes, guards, exception filters. Для підтримки загального стилю коду буде використано наступний підхід: ім'я_сервісу.назва_сутності.розширення.

Визначення:

- dto (data transfer object). Ця сутність оголошується, як клас та використовується для опису вхідних параметрів запиту, для подальшої передачі у сервіс (провайдер);
- module. Клас, анотований декоратором, @Module, де @Module передає метадані, для організації структури додатку. Кожен додаток, має хочаб один модуль – кореневий модуль, який є відправною точкою, для

побудови графу додатку – внутрішньої структури даних, яку Nest використовує для вирішення взаємозв’язку та залежностей між модулями та провайдерами. Завдяки цьому механізму, кожен модуль інкапсулює тісно пов'язаний набір можливостей. Кожен модуль, може імпортувати інший модуль, впроваджувати провайдери, контролери та надсилати в інші модулі;

- controllers. Виконують роль кінцевих точок / маршрутів, для обробки запиту;
- providers. Фундаментальна концепія. Провайдерами можуть бути: сервіси, репозиторії, фабрики, помічники тощо. Основна ідея полягає в тому, що його можна впровадити як залежність, з використанням DI (Dependency Injection) та IoC (Inversion of Control) контейнеру. Це означає, що об’єкти можуть створювати різноманітні зв’язки. Наприклад, можна інтегрувати об’єкт моделі у сервіс, для виконання запитів до БД, або інтегрувати сервіс до контролеру, для взаємодії з методами сервісу, для виконання іншої логіки;
- models. Ця сутність, уявляє собою схему таблиці, або документу у БД;
- interfaces. Тип даних, що визначається згідно з конструкцією interface. Використовується як шаблон, з методами та властивостями, для об’єкту безпосередньо або через прошарок у вигляді класу;
- middlewares. Функція яка викликається перед обробником маршруту – проміжне програмне забезпечення. Вони можуть виконати зміни об’єкту запиту / відповіді; завершити цикл запит / відповдь; викликати інше проміжне програмне забезпечення в стеку. Може використовуватися наприклад для перевірки токена авторизації, і у разі, якщо токен є валідним, відпускати виконання далі;
- pipes. Клас який використовується, у двох випадках: перетворення вхідних даних (парсинг), у потрібну форму (наприклад з строки до числа або до UUID) та валідації вхідних даних. У разі, якщо дані пройшли валідацію, то викликається обробник машруту, якщо ні, то повертається помилка.

Дані можна валідувати згідно об’єктної схеми, або з використанням класового валідатору;

- guards. Клас, який несе єдину відповідальність. Вони визначають, чи буде запит оброблено обробником маршруту чи ні, залежно від певних умов (наприклад, дозволів, ролей, ACL тощо), присутніх під час виконання. На відміну від проміжного програмного забезпечення, точно знає, який обробник буде виконаний далі;
- exception filters. Nest із коробки надає вбудований рівень винятків (фільтр), який відповідає за обробку всіх необроблених винятків у додатку. Крім можливості створення кастомного класу помилки (наприклад 401), він також надає можливість використовувати підготовлені для цього класи, наприклад BadRequestException. Для повного контролю обробки винятків, замість покладатися на вбудований, можна створити свій власний фільтр, який оброблятиме всі або ряд помилок і перевизначити повернення схеми JSON.

Архітектура клієнтської частини поділена на компоненти в рамках одного моноліту, з використанням Vuex та Composition API.

Vuex – централізоване сховище (стейт менеджер) для всіх компонентів у додатку. Використовує в собі модульну систему, в кожному з яких знаходяться 4 блоки, state, getters, mutations і actions:

- state. Зберігає стан. У даному блоці описуються дані, над якими надалі будуть виконані будь-які дії;
- getters. Аналог обчислюваних властивостей (computed). Дозволяє виконувати будь-які дії над станом, наприклад, відфільтрувати список і повернути його. Як і computed кешує значення, тобто якщо стан не було змінено, поверне значення з кеша замість повторних обчислювальних дій;
- mutations. Синхронні функції, призначені для зміни стану;
- actions. Функції, які можуть виконувати асинхронні операції та викликати мутацій для зміни стану.

Класичний сценарій передбачає використання логіки в рамках однофайлового компонента, за винятком можливості використання mixins та store. Усі хуки життєвого циклу так само використовуються у рамках цього компонента. Коли компонент стає занадто великим, код стає розмазаним по всьому компоненту. Composition API дозволяє декомпозувати або винести логіку в окремі файли і об'єднати її в рамках опції setup. Таким чином, стан і методи, що працюють з цим станом, знаходяться в окремих блоках. Хуки життєвого циклу також можуть бути декомпозовані.

4 РОЗРОБКА ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка ПЗ здійснювалася шляхом реалізації двох веб-систем, з серверною та клієнтською складовою. Перша система є поганою, як архітектурно, так і оптимізаційно, і як наслідок з поганою продуктивністю і масштабованістю. Друга – показник того, наскільки система може бути добре організована та оптимізована. Система реалізована у межах сфери медицини.

4.1 Реалізація

Для реалізації веб-системи, було прийнято рішення, використовувати монолітну архітектуру, як на сервері, так і на клієнті. Взаємодія між двома частинами додатку відбувається через протокол передачі даних HTTP, згідно з архітектурним стилем REST. Клієнт являє собою SPA, що використовує єдиний HTML-документ як оболонку для всіх веб-сторінок і організовує взаємодію з користувачем через HTML, CSS, JavaScript, що динамічно завантажуються. При реалізації згідно з попереднім проектуванням, для серверної частини, виконувались наступні кроки:

- ініціалізація модулів, контролерів, сервісів, сидів, міграцій;
- реалізація моделей бази даних, та їх провайдерів на основі токенів, для подальшого впровадження залежностей у сервіси;
- реалізація кінцевих точок та подальша логіка у сервісах;
- взаємодія сервісного шару та шару репозиторію;
- реалізація пайпів (pipes), для валідації вхідних параметрів;
- реалізація фільтру для перехоплення HTTP помилок, які можуть виникнути в процесі виконання;

– окремим є використання захисників (guards) для перевірки доступу користувача до кінцевої точки. Наприклад на основі JWT токену, або ролі.

При реалізації клієнтської частини виконувались такі крокі:

– ініціалізація файлової структури;
– ініціалізація початкового html, менеджера стану (store), роутинг, http, хуків, конфігурації, компонентів svg;
– ініціалізація ассетів, таких як, картинки, шрифти, файли стилів. Серед файлів стилів, такі як: обнулення, організація верхніх шарів, наприклад контейнера, змінних, іконковий шрифт, шаблони для помилкових форм тощо;
– ініціалізація збирача проектів webpack, для dev та production складання;
– реалізація власних глобальних UI компонентів, які будуть використані у будь-якому іншому компоненті. Серед них такі як: notification, autocomplete, select, pagination, snackbar;
– реалізація сторінок, компонентів, Composition API та взаємодію з менеджером стану.

Далі за необхідності функціонал доповнювався.

4.1.1 Серверна частина

Ця система включає 4 ролі: admin, clinic, doctor, patient. Всі користувачі, крім адміну, мають поля, які відносяться тільки до них і загальні, наприклад email. І постало питання поділу сутностей. Тому у зв'язку зі специфікою системи при проектуванні бази даних було висунуто кілька варіацій поділу користувачів:

– створити для кожної сутності свою таблицю. Тоді з'являється проблема пов'язана з повторенням полів у кожній з них, що з погляду архітектурного стилю та подальшої реалізації є некоректним та скрутним;

– створити таблицю користувачів, яка вміщатиме спільні поля для всіх і окремі таблиці для кожної сутності, які включатимуть ті поля, які належать їй. Після цього зв'язати конкретні сутності зовнішнім ключем із загальною таблицею користувачів.

Другий варіант є більш архітектурно правильним, з можливістю подальшого масштабування. Схему можна побачити на рисунку 4.1.

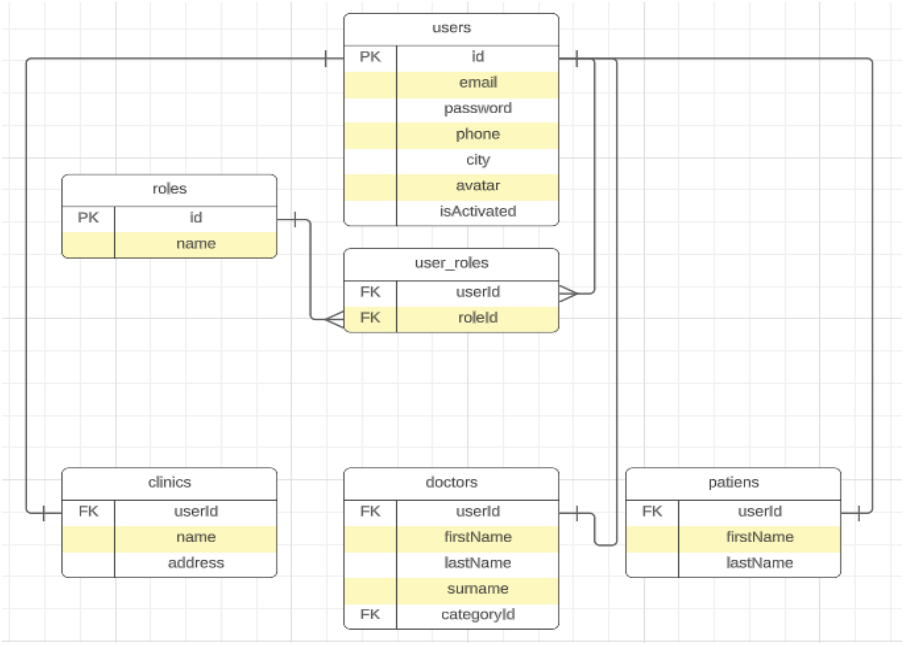

Рисунок 4.1 – ER-діаграма розділення користувачів

Нижче представлено структуру серверної частини, двох різних систем (рисунок 4.2, рисунок 4.3).

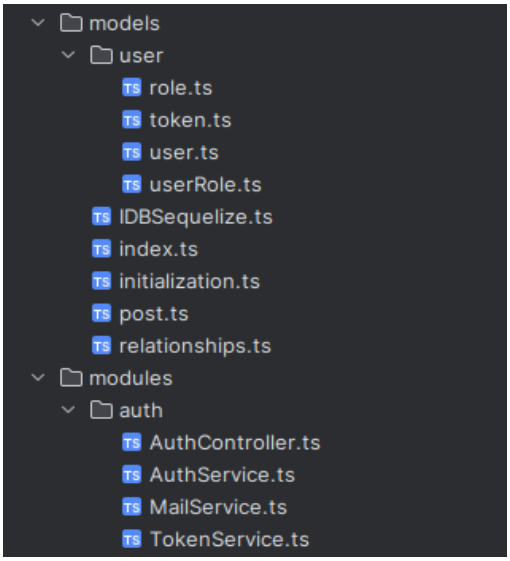

Рисунок 4.2 – Частина поганої структури

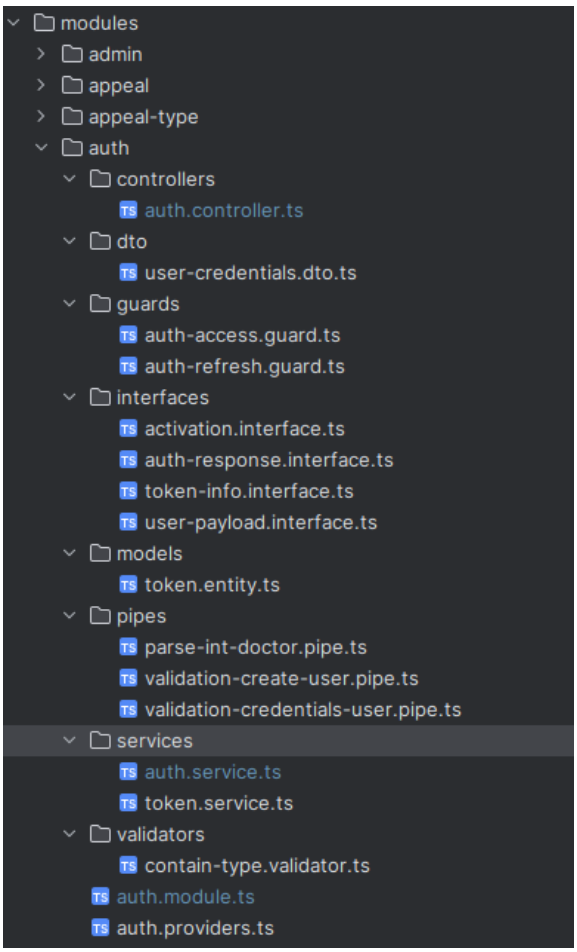

Рисунок 4.3 – Частина гарної структури

З представлених структур, можна дійти такого висновку. Погано спроектована система немає чіткого файлового розподілу. Сервіси знаходяться разом із контролером, сутності лежать окремо від модулів, яким вони належать. Все це разом перетворюється на велику грудку нерозподіленої структури. Не сутності мають бути над модулями, а модулі над сутностями. Так само однією з особливостей Nest, та в свою чергу рекомендованим для використання є визначення іменування сутностей, де файл ділиться на три частини: ім'я сутності, тип і формат.

4.1.2 Клієнтська частина

Ця система, як описано вище, має кілька ролей.

При реєстрації з'являється проблема, коли форма для клініки, пацієнта та лікаря має однакові поля, наприклад: email, password тощо. Існує ризик використання "повторних" компонентів, які включатимуть велику кількість повторюваних полів і однотипну логіку. Для того, щоб уникнути цієї повторюваності, можна виконати реалізацію з використанням слотів. Слоти дозволяють вбудовувати частину контенту (html), що передається із батьківського компонента. Приклад такого зображено нижче на рисунку 4.4.

```
<template>
    <div class="registration-doctor">
        <div class="container">
            <div class="registration-doctor__body">
                <h1 class="registration-doctor__title">Registration Doctor</h1>
                <registration-user-form
                        :v$="v$"
                        :city-errors="cityErrors"
                        :password-errors="passwordErrors"
                        :email-errors="emailErrors"
                        :phone-errors="phoneErrors"
                        v-model:city="user.city"
                        v-model:password="user.password"
                        v-model:email="user.email"
                        v-model:phone="user.phone"
                        v-model:image="image"
                        @registration="registrationUser"
                >
                    <registration-doctor-form
                            :v$="v$"
                            :category-id-errors="categoryIdErrors"
                            :experience-errors="experienceErrors"
                            :surname-errors="surnameErrors"
                            :last-name-errors="lastNameErrors"
                            :first-name-errors="firstNameErrors"
                            :specialty-errors="specialtiesErrors"
                            :categories="categories"
                            :specialties-from-db="specialtiesFromDb"
                            v-model:category-id="user.doctor.categoryId"
                            v-model:experience="user.doctor.experience"
                            v-model:surname=user.doctor.surname
                            v-model:last-name="user.doctor.lastName"
                            v-model:first-name="user.doctor.firstName"
                            v-model:specialties="user.doctor.specialties"
                    />
                </registration-user-form>
            </div>
        </div>
    </div>
</template>
```

Рисунок 4.4 – Використання слотів

При рендерингу компоненту, doctor-form буде вбудований замість слота, в компоненті user-form, який в свою чергу вбудовується в головний батьківський

компонент. Таким чином здійснюється архітектурний прийом і тепер кожна форма, чи то пацієнта чи лікаря, буде вбудовуватись у форму користувача.

Загальну структуру клієнтської частини можна побачити на рисунку 4.5.

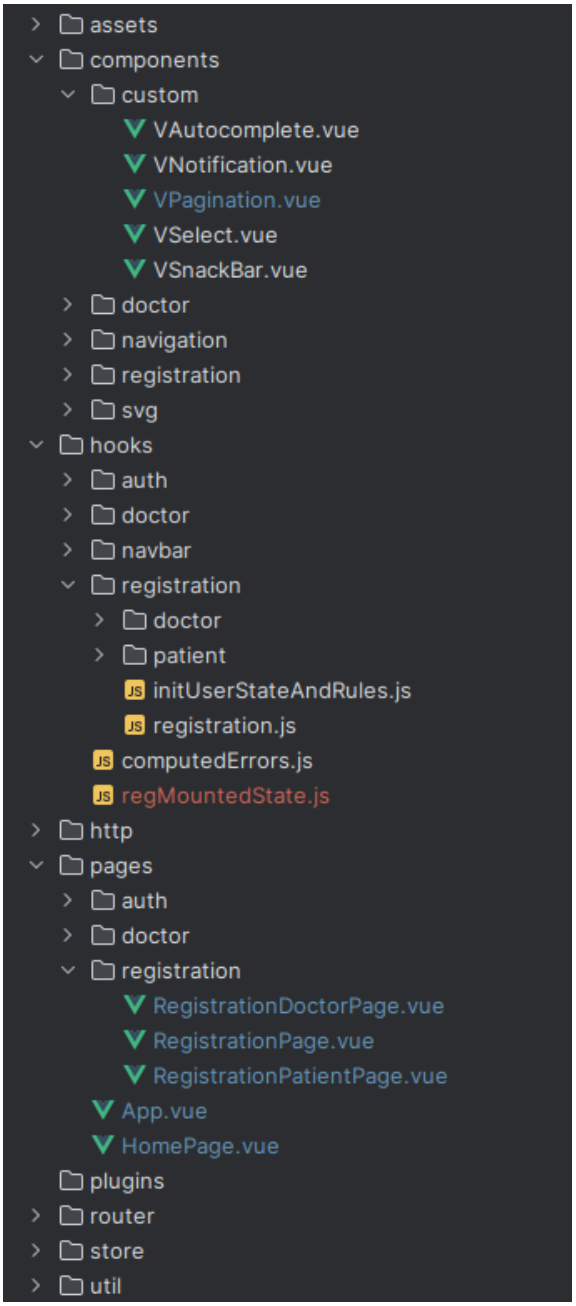

Рисунок 4.5 – Коректна структура клієнтської частини

Імена сторінок закінчуються на page, даючи чітке визначення, що це сторінка. Базові компоненти, які застосовують специфічні для застосування стилі або угоди іменуються з префіксом V. Іменування однофайлових компонентів повинно бути завжди або в PascalCase, або в kebab-case. Найменування компонентів, що

використовуються в єдиному екземплярі, наприклад navbar або footer, іменуються з префіксом the. Все це є запорукою, гарної файлової структури.

4.1.3 Оптимізація продуктивності

Для оптимізації було застосовано цілу низку методів, яку було описано під час аналізу. Як збирач проектів, виступає webpack 5, який вносить значну частку, у підвищенні продуктивності. Нижче перераховані всі методи, які були застосовані і тією чи іншою мірою впливають на продуктивність:

- мінімізування (зменшення ваги) результуючого бандла, що досягається шляхом мініфікації js, css коду. Крім того, для зменшення ваги, так само було переглянуто код на наявність імпорту різних методів з бібліотек, наприклад таких як lodash. Найчастіше можна не помітити, як відбувається імпорт усіх методів із бібліотеки, коли звідти необхідно лише кілька. Це може значно зменшити загальну вагу;
- розбиття чанків з використанням параметру cacheGroups, яке застосовується до node_modules. Цей спосіб використовується для того, щоб вирішити проблему дублювання коду, що може призвести до завантаження надлишкового контенту;
- використання кешування з допомогою: Service Worker та CacheApi. Стратегії, які застосовувались: CacheFirst (для js, css, картинок, шрифтів), Stale-while-revalidate (це чудова стратегія для речей, які важливо підтримувати у актуальному стані, наприклад аватарки користувачів);
- позбавлення від спалаху невидимого тексту (FOIT). Для цього при підключенні локальних шрифтів, використовувався font-display: swap;
- розподілення критичног о та не критичного css, з використанням плагіну до вебпаку critters. Це плагін, який вбудовує критично важливий CSS додатку та ліниво завантажує інший;

- стиснення та перетворення на новий формат картинок, формату JPEG/PNG. Для цього були використані плагіни imagemin, imagemin-mozjpeg, imagemin-webp та ін.;
- виправлення зсуву макета було здійснено шляхом додавання необхідної ширини і висоти картинок. У випадку з контентом, що додається динамічно, передбачено використання скелетонів. Скелетони, це елементи, які є повною копією динамічного контенту і заміщають його до тих пір, доки він завантажується, тим самим позбавляючи сторінку зайвих зрушень;
- запити на отримання лише необхідних даних;
- використання local storage для збереження стану на випадок перезавантаження сторінки. У випадку, якщо користувач захоче перезавантажити сторінку, дані, які зберігаються наприклад в централізованому сховищі store, будуть обнулені і замість того, щоб кожного разу посилати запит на сервер, дані зберігаються в local storage і беруться від туди при необхідності;
- пагінація або нескінченний список та virtual scroller. Пагінація дозволяє розбити великий список елементів на певну кількість сторінок, зменшивши навантаження на сторінку. Нескінченне підвантаження, повинно супроводжуватися virtual scroller, для відображення тільки видимих елементів на сторінці, тому що врешті-решт кількість елементів може стати дуже великою, що значно знизить продуктивність;
- ліниве завантаження маршрутів та компонентів. Ліниве завантаження маршрутів та компонентів, яке надає Vue, у сумісництві з webpack. Наприклад, є такий компонент, як модальне вікно. Зазвичай необхідності в його початковому відображенні з усією сторінкою немає, і цей той випадок, коли необхідне ліниве підвантаження. Модальне вікно буде завантажено тоді, коли воно знадобиться.

Отже, із застосуванням вищеописаних методів, способів, рішень, зокрема під час аналізу, було реалізовано одне із двох додатків. Другий додаток у свою чергу майже не використовував представлені методи.

4.2 Тестування

Для тестування було використано інструмент Lighthouse, який дозволяє відобразити загальний рівень продуктивності сторінки, з візуалізацією певних метрик та їх показниками.

У тестуванні застосовувалася і мобільна версія, і десктопна.

Для виконання тестування була обрана сторінка із завантаженням списку докторів, тобто динамічного контенту.

Результат виконання для оптимізованого додатку у десктоп версії можна побачити на рисунку 4.6.

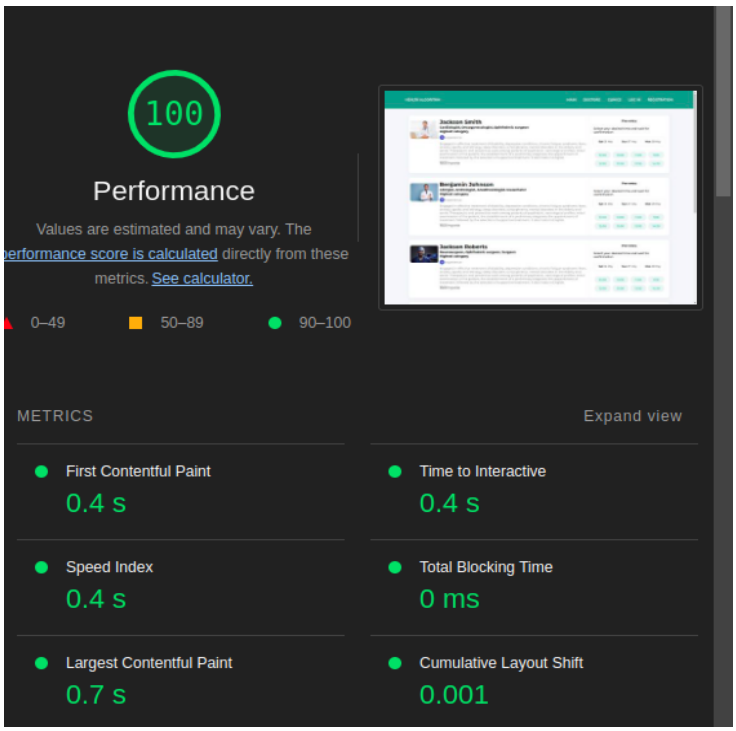

Рисунок 4.6 – Десктоп версія оптимізованого додатку

Результат виконання для оптимізованого додатку у мобільній версії можна побачити на рисунку 4.7.

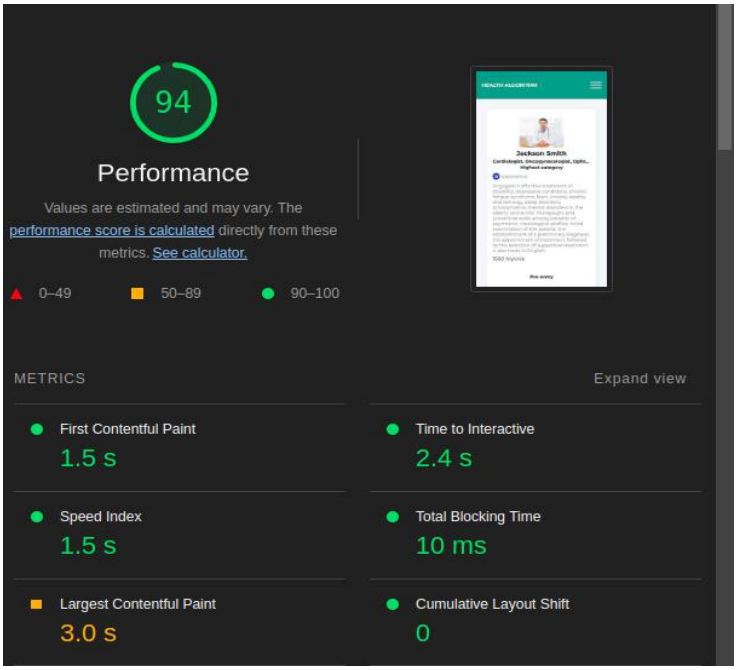

Рисунок 4.7 – Мобільна версія оптимізованого додатку

Результат виконання для не оптимізованого додатку у десктоп версії можна побачити на рисунку 4.8.

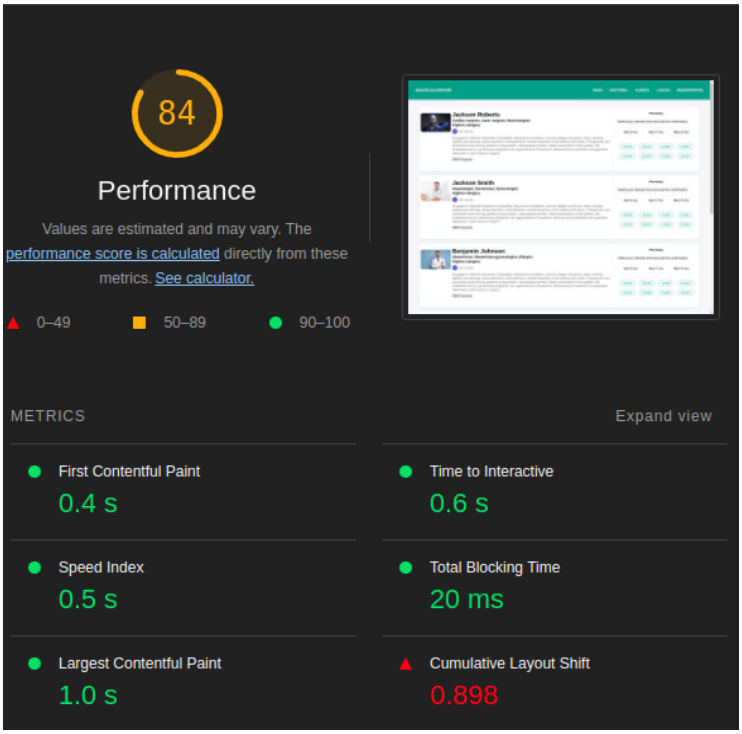

Рисунок 4.8 – Десктоп версія не оптимізованого додатку

Результат виконання для не оптимізованого додатку у мобільній версії можна побачити на рисунку 4.9.

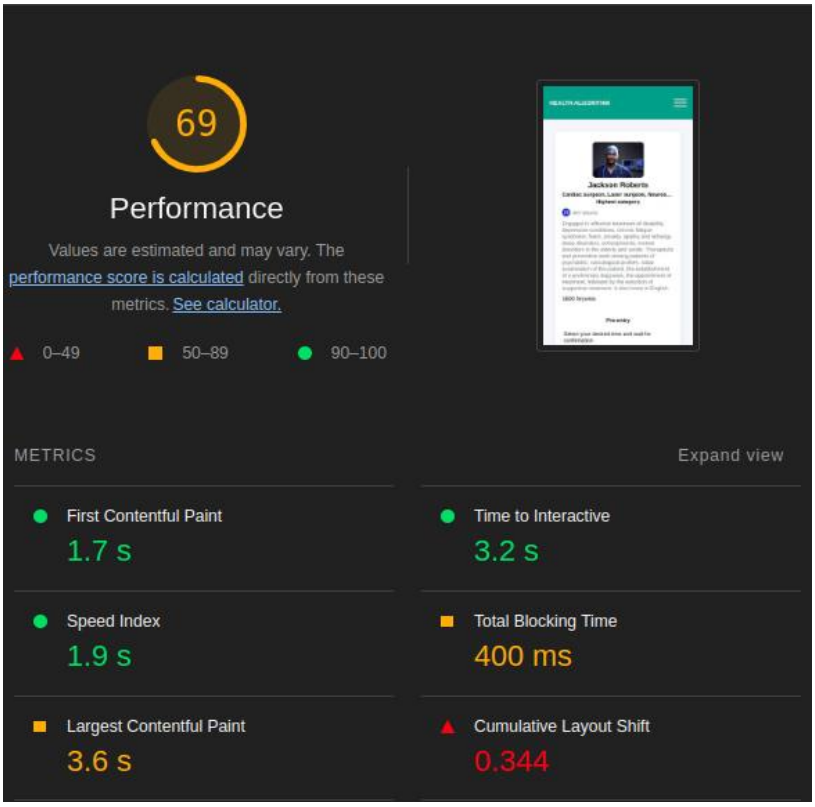

Рисунок 4.9 – Мобільна версія не оптимізованого додатку

В результаті виконання тестування виходить наступне. Для оптимізованого додатка на робочому столі, показники є позитивними, але в свою чергу, на мобільних пристроях, додаток потребує невеликих доопрацювань, а саме поліпшення показника LCP. Як з'ясувалося, це пов'язано з великим описом у картці лікаря. Тому для його поліпшення, можна застосувати зменшення кількості тексту, або взагалі прибрати.

Для неоптимізованого додатка, в декстопній версії спостерігаються значні зрушення макета, виходячи з показника CLS, а в мобільній версії, буквально всі показники, як і очікувалося, були гіршими за оптимізовану версію, особливо це стосується TBT, LCP, CLS.

Надалі, якщо не приділяти уваги оптимізації продуктивності, це може призвести до значно гірших наслідків.

ВИСНОВКИ

У результаті виконання даної роботи, був проведений аналіз предметної галузі, визначена актуальність роботи та постановка задачі.

У ході аналізу методів та архітектурних рішень було описано, які підходи використовуються при проектуванні архітектури, яка архітектура є коректною, а яка ні, її компанування, які найбільш застосовувані архітектурні патерни та їх відмінності, потоки даних у додатку та ін. При аналізі методів оптимізації, були розглянуті основні метрики продуктивності, а також методи оптимізації, включно.

При аналізі програмних рішень, в якому був проведений порівняльний аналіз фреймворків, в рамках Node.js, їх переваги та недоліки, у результаті чого, був обраний Nest.js. У ході аналізу технологій для клієнтської частини, кожен з аналізованих фреймворків / бібліотек, мають свої переваги та недоліки, вони можуть використовуватися взаємозамінно та в оптимізації, вони всі досить швидкі, тому виходячи з уподобань, був обраний Vue.js. Також було спроектовано архітектуру ПЗ, спираючись на аналіз архітектурних рішень. Крім цього, були визначені архітектурні рішення, з використанням функціональних можливостей фреймворків.

Також було розроблено дві веб-системи, однією з яких було приділено увагу оптимізаційним заходам, побудові коректної монолітної архітектури, з дотриманням розподілу зони відповідальності кожного модуля, де кожен сервіс відповідав тільки за ту сутність, якої він належить, а також побудові файлової структури. На клієнтській частині, з використанням Composition API, вдалося домогтися розбиття коду на блоки та їх подальше компанування у методі setup. Розбиття на модулі (блоки), виконувалося й у менджері стану (store), замість накопичення величезних шматків коду одному місці. Крім цього, в рамках даної системи, була необхідність усунення повторюваності шаблонів, і це завдання було вирішено з використанням слотів.

Тестування в десктопній та мобільній версії було проведено з використанням інструменту Lighthouse, який визначає такі важливі метрики, як Web Core Vitals та ряд інших, що допомогло визначити продуктивність додатку. У ході тестування з'ясувалося, наскільки різниться продуктивність між оптимізованою веб-системою та ні.

Для створення додатків, які люди хочуть використовувати, які залучають, утримують користувача, необхідно створити хороший досвід користувача і частиною такого досвіду є швидке завантаження контенту і чуйність на взаємодію з користувачем (час відгуку). Тому використання методів поліпшення продуктивності є важливою частиною продукту, що розробляється. Грамотно збудована архітектура та гарна продуктивність, запорука якісного продукту.

Незважаючи на всі методи, які були застосовані для реалізації системи, мобільна версія виявилася не до кінця оптимізована і, як наслідок, підвищення показника LCP. Для цього необхідне незначне виправлення, але все ж таки. До всього іншого, система знаходиться на localhost, замість того, щоб бути завантаженою на віддалений сервер, з відповідним налаштуванням зворотнього проксі-сервера Nginx, де можна було б використовувати наприклад gzip стиснення, а також використанням CDN сервера, який являє собою деякий механізм кешування та доставку контенту територіально розподіленим користувачам, що прискорює процес завантаження веб-сторінок.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Керівництво Microsoft з проектування архітектури, 2 видання // Edu – 2009. URL: https://dut.edu.ua/uploads/l_1507_99407341.pdf (дата звернення: 09.02.2023).
2. Sus, B., Revenchuk, I., Bauzha, O., Zagorodnyuk, S. Virtual laboratory as custom e-learning implementation and design solution.- CEUR Workshop Proceedings, 2021, 2833, P. 177–187. (дата звернення 10.02.2023).
3. Sus, B., Revenchuk, I., Tmienova, N., Bauzha, O., Chaikivskyi, T. Software System for Virtual Laboratory Works.- 2020 IEEE 15th International Scientific and Technical Conference on Computer Sciences and Information Technologies, CSIT 2020 - Proceedings, 2020, 1, Pp. 396–399, 9322046 (дата звернення: 14.02.2023).
4. Роберт Сесіл Мартін, Чиста архітектура – мистецтво розроблення програмного забезпечення // 2019 – 368 с. (дата звернення: 15.02.2023).
5. Як обрати архітектуру для Web-додатку // Ithillel – 2022. URL: https://blog.ithillel.ua/articles/web-application-architecture (дата звернення: 15.02.2023).
6. Web Application Performance: 7 Common Problems // Stackify – 2018. URL: https://stackify.com/web-application-problems/. (дата звернення: 16.02.2023).
7. Оптимізація веб-сторінок та додатків // CodeGuida – 2014. URL: https://codeguida.com/post/189 (дата звернення: 16.02.2023).
8. 18 Tips for Website Performance Optimization // Keycdn – 2022. URL: https://www.keycdn.com/blog/website-performance-optimization (дата звернення: 16.02.2023).
9. KeepAlive, Rendering Mechanism, Performance Vue // Vuejs – 2022. URL: https://vuejs.org/guide/introduction.html (дата звернення: 18.02.2023).
10. Largest Contentful Paint image was lazily loaded warning. // Perfmatters. URL: https://perfmatters.io/docs/largest-contentful-paint-image-was-lazily-loaded/ (дата звернення: 20.02.2023).

11. 9 tricks to eliminate render blocking resources. // BlogRocket – 2022. URL: https://blog.logrocket.com/9-tricks-eliminate-render-blocking-resources/#load-custom-fonts-locally (дата звернення: 22.02.2023)
12. Mastering browser cache. // Vueschool – 2020. URL: https://vueschool.io/articles/vuejs-tutorials/vue-js-performance-mastering-cache/ (дата звернення: 24.02.2023).
13. Comparison of Node.js Frameworks // Inventorsoft – 2022. // Invetorsoft. URL: https://inventorsoft.co/blog/top-14-node-js-frameworks-comparisson (дата звернення: 01.03.2023).
14. Порівняння Vue з іншими клієнтськими фреймворками / бібліотеками // Vuejs – 2022. URL: https://v2.vuejs.org/v2/guide/comparison.html?redirect=true (дата звернення: 04.03.2023).