Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра Інформатики Факультету Інформатики

Розробка веб-застосунку на основі дослідження підходів SSR та CSR (SPA)

Текстова частина до курсової роботи за спеціальністю
«Інженерія програмного забезпечення» - 121

Керівник курсової роботи
Кравченко О.В

_____
(підпис)
“____”
_____ 2022 р.

Виконала студентка БП ІПЗ-4
Горуля К. І.
“____”
_____ 2022 р.

ЗМІСТ

ВСТУП ... **3**

1. АРХІТЕКТУРА ВЕБ-ДОДАТКІВ ... **4**

1.1 Клієнт-серверна архітектура веб-додатків ... **4**

1.2 Відображення (рендеринг) контенту у браузері. DOM-дерево ... **6**

1.3 Сучасні клієнтські архітектури веб-додатків ... **7**

1.3.1 Server-side Rendering (SSR) ... 8

1.3.2 Single-page Application (SPA) ... 9

1.3.3 Популярність SSR і CSR (SPA) ... 11

1.4 Технології. React. Virtual DOM. Next.js ... **12**

2. ДОСЛІДЖЕННЯ І ТЕСТУВАННЯ ПІДХОДІВ SSR І SPA НА ПРИКЛАДІ РОЗРОБКИ ІДЕНТИЧНИХ ЗАСТОСУНКІВ ... **13**

2.1 Підхід SPA ... **13**

2.1.1 Розробка міні-проекту на React ... 13

2.1.2 Результати роботи та тестування швидкості ... 16

2.2 Підхід SSR ... **19**

2.2.1 Розробка міні-проекту на React та Next.js ... 19

2.2.2 Результати роботи та тестування швидкості ... 22

2.3 Висновок. Порівняння підходів. Застосування для кожного підходу. Вибір архітектури для власного додатку. ... **24**

3. РОЗРОБКА ПРОЕКТУ ... **27**

3.1 Актуальність ... **27**

3.2 Архітектура проекту. Монорепозиторій. ... **28**

3.3 Back-end розробка ... **28**

3.3.1 Використані технології ... 28

3.3.2 Структура проекту ... 29

3.3.3 База даних ... 30

3.3.4 Опис запитів до БД ... 32

3.4 Front-end розробка ... **34**

3.4.1 Використані технології ... 34

3.4.2 Структура проекту ... 34

3.4.3 Логін та Реєстрація ... 36

3.4.4 Отримання даних з back-end ... 39

3.5 Демонстрація проекту ... **40**

ВИСНОВОК ... **45**

ВСТУП

У сучасному світі технології змінюються кожен день. Те, що було популярним учора, сьогодні вже є застарілим. Проте, більшість нових технологій насправді – це лише надбудови над існуючими базовими принципами розробки, які допомагають оптимізувати та покращити виконання коду. Професійному розробнику потрібно досконало розуміти та знати свою область роботи, а також бути у курсі всіх новинок, щоб досягати результату найефективнішим способом. Так, наприклад, у front-end розробці React – це JavaScript бібліотека, яка допомагає зручно оновлювати HTML у браузері. Це лише надбудова над *існуючими* способами відображати контент у браузері. Розуміючи це, розробник може відмовитись від цієї бібліотеки, якщо потрібно, або легко її замінити іншими, більш вдалими варіантами.

Для front-end розробників актуальним питанням є вибір архітектури для веб-застосунків, а саме підходу для відображення контенту у браузері. Існує декілька таких підходів: SSR, SPA, SSG, але щоб обрати кращий варіант у конкретному випадку, потрібно чітко усвідомлювати, як кожен з них працює і чим вони відрізняються. Це питання дуже мене зацікавило, адже вибір архітектури додатку – найголовніше при його створенні. Вибір неправильного підходу для розробки застосунку може мати негативні наслідки для бізнесу, адже він напряму впливає на швидкість реалізації, вартість і час розробки, можливість додавати оновлення у майбутньому і підтримувати існуючий код. У світі, де все дуже швидко змінюється, вибір правильного підходу є критичним для успіху проекту.

Метою моєї курсової роботи є дослідити підходи до відображення контенту у браузері – SPA і SSR, зрозуміти як вони працюють, в чому їх різниця і в яких випадках потрібно використовувати кожен з підходів. За результатами дослідження обрати відповідний підхід для розробки мого власного веб-додатку для запису на косметичні послуги, ідею, цінності та дослідження якого, я описувала у своїй минулій курсовій роботі[1].

1. АРХІТЕКТУРА ВЕБ-ДОДАТКІВ

1.1 Клієнт-серверна архітектура веб-додатків

Будь-який веб-додаток є прикладом клієнт-серверної архітектури. Модель “клієнт-сервер” — це структура програми, яка розподіляє завдання або робочі навантаження між постачальниками ресурсу або послугами, які називаються серверами, і замовниками цих послуг, які називаються клієнтами[2]. Клієнт-серверна архітектура (рисунок 1) описує взаємодію між клієнтом, прикладом якого може бути локальний комп’ютер, телефон, браузер тощо, та сервером. Розробники клієнтської частини додатку зазвичай називаються front-end розробниками. А серверної частини – backend розробниками.

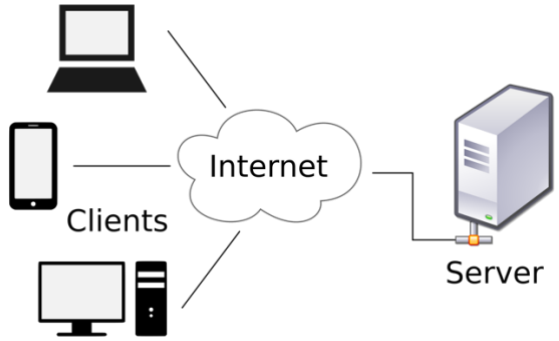


Рисунок 1 - Клієнт-серверна архітектура

Клієнт ініціює отримання інформації від серверу у вигляді запитів (request). Сервер надає цю інформацію одному або декільком клієнтам (response). Клієнтські комп’ютери забезпечують інтерфейс, який допомагає користувачу зручно робити запит та отримувати візуалізовану інформацію. Спілкуються клієнт і сервер між собою через комп’ютерну мережу або на окремих обладнаннях, або в одній системі.

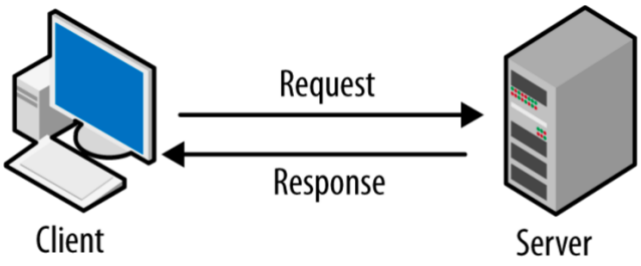


Рисунок 2 - Взаємодія клієнт-сервер

У випадку веб-додатків клієнтом виступає браузер, а сервером – веб-сервер. Щоб детальніше зрозуміти їх взаємодію потрібно додати інформацію про деякі терміни[3]:

- TCP/IP

  Transmission Control Protocol/Internet Protocol – це комунаційні протоколи зв’язку у мережі Інтернет, які дають можливість спілкуватись комп’ютерам між собою на великій відстані та визначають спосіб передачі даних по мережі. TCP – збирає пакети з даними, IP – доставляє у місце призначення[4].

- DNS

  Domain Name System – систему доменних імен часто називають телефонною книжкою інтернету. Веб-браузери взаємодіють між собою за адресами Інтернет-протоколу IP. Оскільки IP адреси важко запам’ятати, вони незручні для застосування клієнтами, використовують доменні імена. Користувачі заходять на веб-сайти за допомогою доменних імен (google.com). DNS перетворює доменні імена сайтів у IP адреси, щоб браузери могли отримувати інформацію[5].

- HTTP

  Hypertext Transfer Protocol – протокол для передачі гіпермедійних документів, наприклад HTML. За допомогою нього відбувається зв’язок між браузерами (клієнтами) та веб-серверами.

Описавши ці поняття, можемо прослідкувати, як користувач потрапляє на веб-сайт та отримує інформацію на ньому. Спочатку він вводить адресу веб-сайту у пошуковий рядок. Браузер звертається до DNS-серверу, щоб за доменним іменем отримати справжню IP адресу сайту. Потім браузер за допомогою протоколу TCP/IP по інтернет-з’єднанню передає HTTP запит на веб-сервер для отримання копії сайту. У випадку успішного звернення сервер надсилає клієнту “status: 200 OK” і починає відправляти файли сайту (index.html, main.css, script.js тощо) у вигляді пакетів. Потім ці пакети опрацьовуються браузером та

збираються у повноцінний сайт, який ми звикли бачити. Якщо запит не успішний, сервер відправляє текст помилки та її статус.

1.2 Відображення (рендеринг) контенту у браузері. DOM-дерево

Для кращого розуміння front-end розробки, варто детальніше розглянути, як браузери відображають веб-сторінки. В минулому розділі вже описано, що браузер на запит користувача отримує від серверу пакети з даними. Але як ці пакети стають веб-сайтом? Цей процес називається рендерингом веб-сторінок. Відображення (рендеринг) контенту у браузері – процес перетворення HTML, CSS, JavaScript в інтерактивну сторінку, який завершується механізмом візуалізації веб-сторінки. Це відбувається наступним чином:

1. Браузер оброблює HTML та будує DOM-дерево (рисунок 3)

   Механізми браузера перетворюють дані (байти), отримані з пакетів від веб-сервера в символи. Потім розподіляють ці символи на лексеми (токени)[6], які комбінуються у вузли[7]. Далі ці вузли групуються у деревоподібну структуру, яка називається Document Object Modal. DOM – це представлення HTML сторінки у вигляді JavaScript об’єктів, в якому кожен тег, вузол дерева є окремим об’єктом;

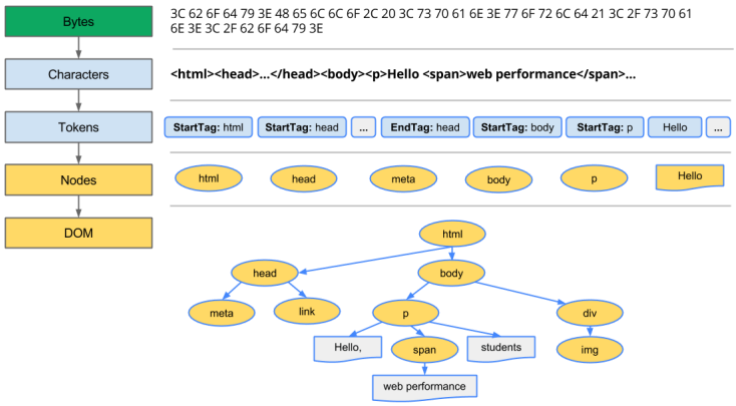


Рисунок 3 - Створення DOM-дерева

2. Браузер оброблює CSS та будує CSSOM-дерево (рисунок 4)

   Аналогічно до HTML у минулому пункті, CSS оброблюється браузером та формується у деревоподібну структуру JavaScript об’єктів, яка називається CSS Object Modal;

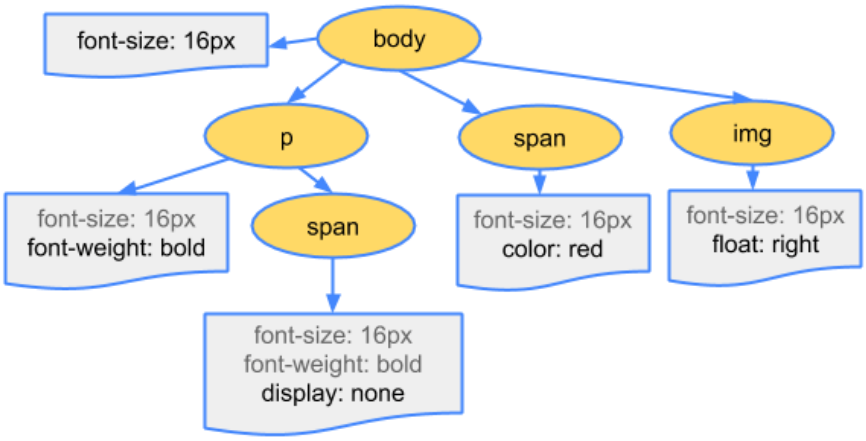


Рисунок 4 - Створення CSSOM-дерева

3. Поєднання DOM і CSSOM-дерев у дерево візуалізації (рендеру)

   Це дерево містить у собі всю необхідну інформацію для наступних кроків;
4. Завдяки дереву рендеру, браузер обчислює позицію кожного елемента (вузла) сторінки;
5. Відмальовування (додавання) кожного елемента на веб-сторінку.

1.3 Сучасні клієнтські архітектури веб-додатків

Оскільки, HTML та CSS за допомогою механізмів браузера перетворюються відповідно у JavaScript об’єкти, ми можемо доступатись до будь-якого елементу сторінки за допомогою JavaScript. Таким чином у розробників з’являється можливість робити будь-які зміни існуючої сторінки. Наприклад, можна змінити колір фону сторінки. Доступатися до елементів DOM-дерева можна через об’єкт “document”. Завдяки команді “document.body” ми можемо доступитись до тегу <body><body /> нашої HTML сторінки, а за допомогою команди “document.body.style.background = ‘blue’” змінити фон усієї сторінки на синій колір.

Існує багато селекторів, за якими можна зручніше доступатись до тегів, класів, id тощо. А також багато функцій, які дають можливість змінювати текст, стилі, додавати слухачі подій, динамічно оновлювати цілі сторінки елементів. Раніше front-end розробники так і створювали веб-сайти. Але даний спосіб

розробки є вкрай незручним, адже після будь-якої зміни приміром додавання елемента або видалення його, потрібно очищати DOM-дерево та заново додавати на нього елементи (робити перерендер). Окрім того, що такий спосіб розробки робить код нечитабельним, це спричиняє велике навантаження на браузер і, як наслідок, провокує затримки у відображенні даних для клієнтів. Операція перерендеру DOM-дерева є однією з найважчих за часом і пам'яттю операцій для браузера. Саме тому з часом способи та підходи до розробки клієнтських застосунків змінились. В наступних розділах я детальніше розкажу про кожну з популярних нині архітектур для розробки, а також про засоби, які дозволяють оптимізувати операції з DOM-деревом.

1.3.1 Server-side Rendering (SSR)

Server-side rendering – тип дизайну архітектури веб-додатку, при якому весь процес відтворення (рендерингу) контенту сайту (HTML, CSS, JS, дані з API) відбувається на сервері (рисунок 5). Користувач робить запит на отримання веб-сторінки, сервер готує HTML сторінку: додає на неї стилі, завантажує дані (якщо є запити на сторонні API) і після цього відправляє браузерові, який відображає вже повністю готову сторінку користувачеві. Таким чином юзер за лічені секунди (адже сервер працює значно швидше за браузер) отримує вже провантажену сторінку. Йому не доводиться додатково чекати на підвантаження картинок або додаткових даних. Це значно покращує досвід користувача. Використання такого типу архітектури також покращує SEO-оптимізацію, оскільки браузер отримує цілісну HTML сторінку з контентом, який можна легко індексувати.

До недоліків цього підходу зазвичай відносять перенавантаження серверу, адже користувач, відвідуючи нову сторінку, кожен раз відправляє запити на сервер для отримання її. А також, низький рівень взаємодії з кінцевим користувачем. Використовуючи SSR, неможливо створити клієнтський продукт, який містить багато інтерактиву для юзера та потребує постійного оновлення даних. Для будь-якої зміни даних потрібно перезавантажувати *всю* сторінку, що

спричиняє навантаження на сервер та змушує користувача чекати, навіть якщо зміна даних – це одне слово.

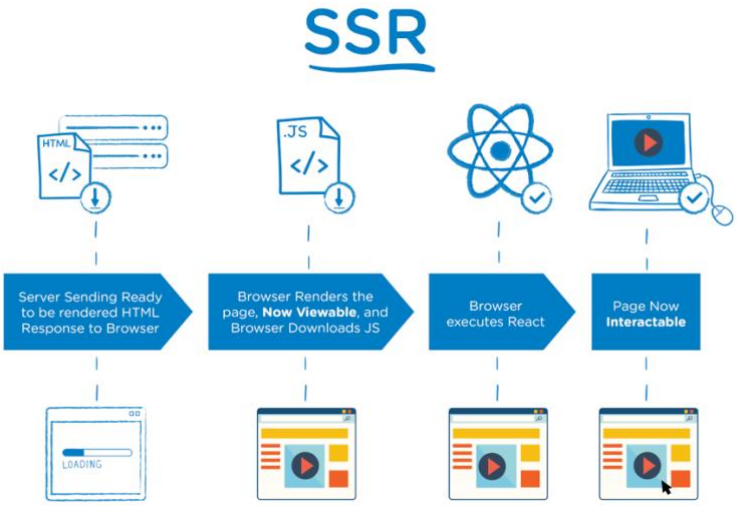


Рисунок 5 - Server-side rendering

1.3.2 Single-page Application (SPA)

Single-page Application – вид веб-додатків, у яких взаємодія з користувачем відбувається шляхом динамічного переписування поточної HTML-сторінки, без її оновлення та завантаження нових сторінок. Такі додатки є прикладом CSR – Client-side Rendering (рисунок 6). На відміну від SSR, де браузер отримує одразу весь вміст HTML-документу, у CSR підході браузер отримує стандартний HTML-файл з контейнером для майбутнього DOM-дерева і файли JavaScript, які після початкового завантаження відтворюють решту веб-сторінки та оновлюють DOM. Тобто на початку користувач отримує порожню сторінку, і вже “на його очах” відбувається додавання усіх елементів та отримання даних від сторонніх API. Перехід на будь-яку нову сторінку не змінює і не оновлює початкову HTML-сторінку, а здійснюється шляхом очищення DOM-дерева до початкового стану і додаванням нових елементів у нього завдяки JavaScript. Це дозволяє зробити перехід між сторінками миттєвим та непомітним для користувача.

Такий спосіб рендеру дозволяє зменшити кількість запитів на сервер та здійснювати оновлення даних лише у відповідь на дії користувача. При тому перезавантажуються у такому випадку лише ті елементи DOM, які містять змінені дані. Інші ж при цьому лишаються статичними та незмінними, адже

JavaScript з ними не взаємодіє. Хоч перше завантаження таких додатків є повільнішим (користувач чекає поки JavaScript відпрацює у браузері), подальша взаємодія з ним відбувається швидко за рахунок відсутності постійних запитів до серверу. Архітектура SPA підходить для застосунків, у яких планується швидка зміна даних на окремих частинах сторінки, а також широка взаємодія з користувачем, наприклад веб-платформи, панелі відображення.

Найбільшим мінусом SPA, окрім швидкості першого завантаження, є SEO-оптимізація[8]. Оскільки спочатку у браузер завантажується порожня сторінка HTML і лише потім JavaScript додає на неї елементи, це робить неможливим одразу відслідкувати кількість посилань та унікальних сторінок. Тому їх індексація очікує своєї черги після завантаження відповідних сторінок JavaScript-ом та додавання цих гіперпосилань на них. Більш за те, для ранжування в пошуку кожна унікальна сторінка додатку має мати власну чисту[9] URL-адресу, не зважаючи на те, що кожна сторінка SPA насправді візуалізується в одній і тій самій HTML-сторінці. Для пошукових систем, таких як Google, та для ботів соціальних мереж метадані мають бути отримані зі статичного HTML, щоб індексувати та передзавантажити сторінки для користувача. З SPA додатком це зробити важче та довше. Проте, не зважаючи на всі ці мінуси, розробники знаходять різноманітні рішення для покращення SEO-оптимізації.

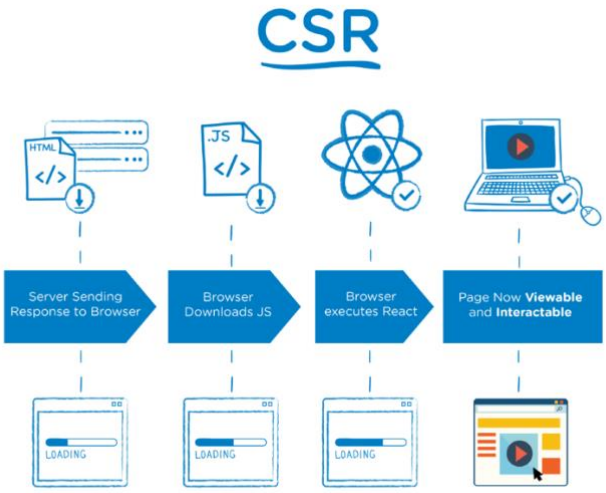


Рисунок 6 - Client-side Rendering

1.3.3 Популярність SSR і CSR (SPA)

Server-side rendering був однією з перших технологій для веб-розробки і довго тільки цей підхід використовувався програмістами. На стороні сервера, зазвичай це був PHP-сервер, збиралась сторінка та вже готовою віддавалась клієнту. Це було швидко та ефективно. Але з часом застосунки у вебі ставали дедалі комплексними та обширними, вони потребували широкої взаємодії з користувачем та оптимізації в навантаженні серверів. Тому на заміну серверному рендеру прийшов клієнтський разом з SPA та різними фреймворками для роботи з ним. Такі додатки стали дуже популярними та з часом майже повністю витіснили серверний спосіб рендеру. Програмісти використовували SPA не задумуючись, чи справді він потрібен. Всі нові технології та фрейворки розроблялись для SPA додатків. До таких можна віднести React, AngularJS, Vue.js, Ember.js, Meteor.js. Та останнім часом в спільноті front-end розробників з'явився новий тренд – розробка SSR додатків. Сталось це в тому числі через появу фреймворків, які дозволяють використовувати вже популярні для SPA технології, наприклад Next.js + React. Але факт у тому, що розробка повернулася до такої, якою була 10 років тому. Це означає, що SPA не є ідеальним варіантом для розробки веб-додатків і програмісти, шукаючи щось інше, повернулись до забутого старого.

Проте, досі постає питання, особливо у молодих веб-розробників, який саме підхід обрати для створення додатку? Яка архітектура допоможе досягти найкращого результату у конкретному випадку? Проаналізувавши відповіді своїх колег, я прийшла до висновку, що переважаюча більшість не усвідомлює різницю SPA і SSR, та не може дати відповідь, у яких випадках використовувати кожен з підходів. Так само плутаюсь у цих поняттях і я. Саме тому у наступних розділах я протестую обидва способи рендеру контенту у браузері – SSR і CSR (SPA), проаналізую їх роботу та вимоги до свого проекту. Таким чином, я прийду до висновку, який з цих підходів оптимальний для мого власного додатку та почну роботу над ним.

1.4 Технології. React. Virtual DOM. Next.js

У своїй роботі я вже не раз зазначала, що сучасні програмісти використовують різноманітні бібліотеки та фреймворки для розробки веб-додатків. Роблять вони це для оптимізації та спрощення написання коду. Прикладами таких технології є вже згадані раніше React, AngularJS, Vue.js, Ember.js, Meteor.js. Та найпопулярнішою серед них є React[10]. За статистикою[11] 40% розробників використовують у своїй роботі React, 23% Angular і 19% Vue.js. React – це JavaScript бібліотека для створення клієнтських інтерфейсів на основі компонентів. Саме ця бібліотека популяризувала використання віртуального DOM-дерева (VDOM). Я вже згадувала, що будь-які операції напряму з деревом DOM, перевізуалізація його елементів становлять найбільше навантаження для браузеру. Алгоритми Virtual DOM створюють копію DOM-дерева із JavaScript об'єктів, яка займає значно менше пам'яті. Кожному елементові DOM-дерева існує відповідний VDOM елемент. Всі зміни елементів та їх даних відбуваються спочатку у віртуальній копії дерева. Потім змінений VDOM порівнюється з поточним DOM-деревом, і лише тоді відбувається перевідображення елементів справжнього DOM, які зазнали змін.

Раніше React використовувався для клієнтського рендерингу елементів, але у 2016 році світ побачив фреймоворк Next.js. Цей фреймворк створений для розробки SSR веб-додатків на Node.js та React. Він швидко сподобався програмістам, адже з використанням Next.js можна було продовжити писати код на вже звичному React, а також користуватись перевагами серверного рендеру сторінок.

Саме через можливість дослідити два різних підходи SSR і SPA на одній і тій самій бібліотеці React з використанням додаткового фреймворку Next, а також через їх популярність, я обрала ці технології для своїх тестових проектів.

2. ДОСЛІДЖЕННЯ І ТЕСТУВАННЯ ПІДХОДІВ SSR І SPA НА ПРИКЛАДІ РОЗРОБКИ ІДЕНТИЧНИХ ЗАСТОСУНКІВ

В минулих розділах я вже визначила предмет і мету дослідження, технології, які буду використовувати. В цьому розділі я опишу сам процес дослідження, свої кроки, здобуту інформацію та висновки. Для отримання об'єктивних результатів та досвіду від тестування підходів є сенс розробити два однакових невеличких проекти для кожного типу рендерингу. На невеликих проектах я зможу швидко протестувати роботу і React, і Next, і проаналізувати, як працюють SSR і SPA. Для цієї мети підійде найпростіший функціонал, як от запит до стороннього API і виведення результатів. Тестові додатки будуть складатись з двох сторінок – домашня та новинна. На новинній сторінці будуть відображатись 10 картинок і 10 результатів запитів до публічного API – NewsAPI[12]. В процесі розробки я буду звертати увагу на легкість у використання технологій (React vs Next.js + React), зручність маршрутизації, швидкість завантаження сторінок тощо.

2.1 Підхід SPA

2.1.1 Розробка міні-проекту на React

Розробка більшості додатків на React починається з команди "*npx create-react-app my-app*". Результат виконання цієї команди інсталює React та створює в проекті стандартні папки і файли для запуску елементарного застосунку, приклад якого зображений на рисунку 7. Завдяки цій команді розпочати розробку можна в лічені хвилини, не витрачаючи час на додаткові налаштування webpack[13] тощо.

Для свого тестового додатку я обрала структуру, зображену на рисунку 8. Всі папки розробник може створювати та називати на свій смак, але прийнято мати папку *components* для зберігання компонентів. Папку *pages* іноді не створюють, додаючи всі сторінки в папку *components*, проте мені здалось логічним все ж відокремити їх у папку.

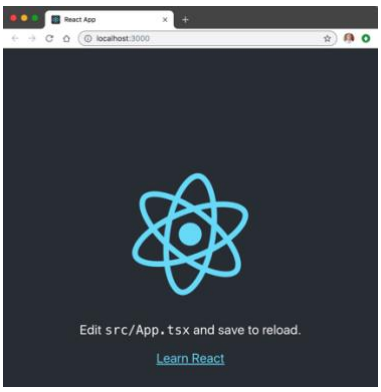


Рисунок 7 - Стандартний create-react-app проект

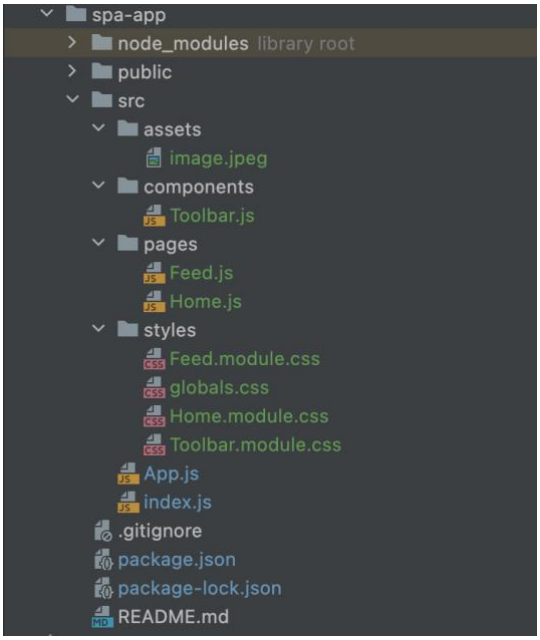


Рисунок 8 - Структура SPA проекту

Мій проект складається з папок *public*, *src* та файлу *package.json*. Папка *src* містить: папку *assets* для сторонніх картинок; *components* для компонентів; *pages* для компонентів-обгорток, які виконують роль сторінок; *styles* для стилів; компонент *App.js* – обгортку для всіх компонентів, яка містить в собі router для навігації по сторінкам та файл index.js, який рендерить через ReactDOM компонент App у div з id "*root*".

Навігація по веб-сайту описується в головному компоненті App. Для цього в React використовуються окрема бібліотека "*react-router-dom*", з якої можна отримати компоненти Switch і Route. Далі ці компоненти складаються в наступну структуру:

```
<Switch>
  <Route exact path="/" component={Home}/>
  <Route path="/feed" component={Feed}/>
```

```
  <Route path="/feed/:id" component={CustomFeed}/>
</Switch>
```

Компонент Switch використовується як обгортка для компонентів Route. Кожен окремий Route відповідає за посилання, на яке може перейти користувач. У випадку, якщо посилання має бути динамічним та містить параметр до нього додається двокрапка та назва параметру, наприклад /feed/:id.

Запити до API в SPA підході з використанням React найчастіше відбуваються двома способами – миттєво при потраплянні користувача на сторінку або у відповідь на його дію (натискання на кнопку). У другому випадку це звичайна функція, яка викликається при спрацюванні обробника подій. Для першого ж випадку використовується один з методів життєвого циклу компонентів React. В сучасному стилі написанні коду – функціональному, а не класовому підході до компонентів, назви та спосіб використання цих методів трошки змінився, але їх зміст залишився тим самим. В бібліотеці React існує 3 основних цикли[14] (рисунок 9) :

- Mounting – додавання компоненту в DOM-дерево. Метод *componentDidMount* виконується після того, як компонент був доданий у DOM. В цей метод зручно додавати функції, які мають виконуватись одразу при відвідуванні сторінки. Наприклад запит до API з виводом статей. В функціональному підході такий метод реалізовується за допомогою хука *useEffect*[15] с порожнім масивом залежностей.
- Updating – оновлення компоненту, яке може бути викликане змінами властивостей або стану. Додавши в масив залежностей хуку *useEffect* стан чи змінну, ми можемо “змусити” функцію в цьому хукові повторити своє виконання.
- Unmounting – видалення компоненту з DOM-дерева. В методі *componentWillUnmount* зручно видаляти слухачі подій та таймери. У функціональному підході цей метод замінюється додаванням return {} до хуку *useEffect*.

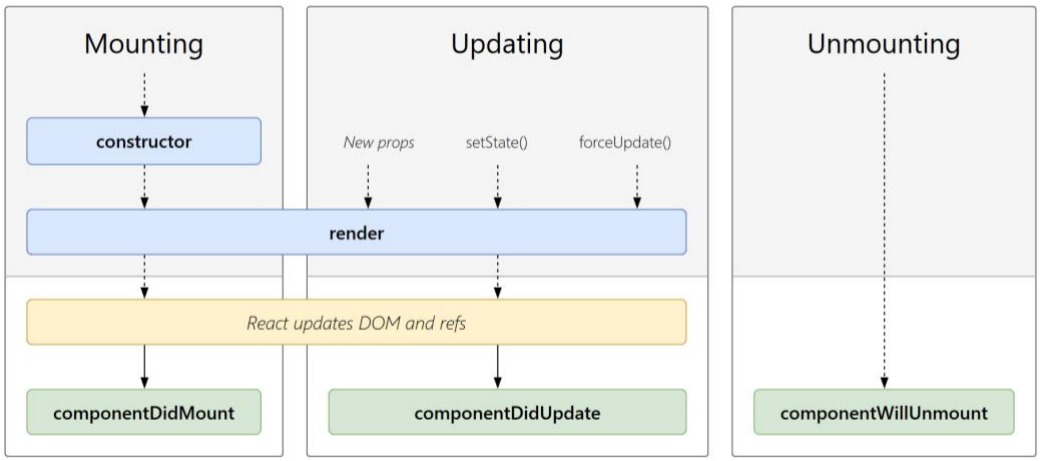


Рисунок 9 - Цикли життя компонентів React

Можемо зробити висновок, що для такої задачі, як відображення стрічки новин на сторінці, нам потрібно додати в компонент, який відповідає за сторінку новин (Feed.js), хук *useEffect* з порожнім масивом залежностей та викликом запиту до API на отримання статей. Отриманий результат записати в стан[16] компоненту та вивести на екран.

```
const [articles, setArticles] = useState([]);

useEffect( () => {
  async function fetchData() {
    const apiResponse = await fetch(
      `https://newsapi.org/v2/everything?q=apple`,
      {
        headers: {
          Authorization: `Bearer af8be6e906d241a5827bc1058a686df0`,
        },
      },
    ).then(res => res.json());
    const { articles } = apiResponse;
    setArticles(articles);
  }

  fetchData();

}, [])
```

Рисунок 10 - Приклад запиту в useEffect

2.1.2 Результати роботи та тестування швидкості

Результатом роботи став дуже простий веб-сайт зі статичною сторінкою Home, яка містить кілька речень, та більш навантаженою сторінкою Feed з запитом до API та виводом 10 картинок і 10 статей (рисунок 11). Розробка такого додатку не зайняла багато часу, проте дала змогу ознайомитись з головними принципами роботи React.

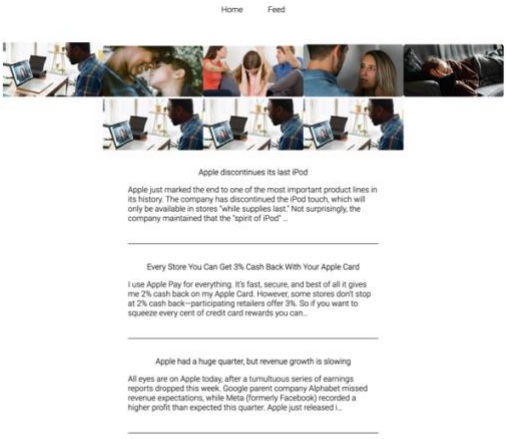


Рисунок 11 - Результат роботи над SPA додатком

Якщо на сторінці натиснути на кнопку “view page source” в браузері відкриється HTML файл нашої сторінки. Як видно на рисунку 12 в ньому абсолютно нічого немає, окрім контейнеру. Саме в цей контейнер з id “root” завдяки JavaScript додається контент. Тим не менш, цей контент не відображається на page source, що і спричиняє погану SEO-оптимізацію SPA додатків.

Рисунок 12 – Page source SPA додатку

Протестувати цей міні-додаток легко і зручно використовуючи Lighthouse[17]. Це інструмент від Google для відслідковування показників, таких як: performance (продуктивність), accessibility (доступність), SEO тощо. Після тестування сторінки з новинами, я отримала наступні результати – 52% продуктивності, 80% доступності і 92% SEO (рисунок 13). На рисунку 14 відображається послідовність, в якій користувач бачив контент. Чітко видно, що спочатку він бачив 2-3 картинки, лише потім додались інші та текст. Зазвичай, у

більш великих додатках суттєво відрізняється перший скріншот від третього, а третій від п’ятого, бо контент повільно дозавантажується. Це спричиняє поганий відсоток performance.

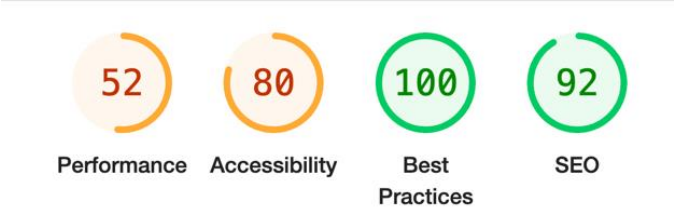


Рисунок 13 - Результат тестування Lighthouse

Рисунок 14 - Послідовність завантаження контенту

На рисунку 15 я додала час завантаження кожної картинки. Вони займають багато пам’яті, і Lighthouse рекомендує додавати картинки в форматі webp, проте робити це досить довго по часу та незручно без сторонніх бібліотек. Тому маємо час завантаження кожної картинки 30-70 ms, що теж погіршує performance. Рисунок 16 свідчить про те, скільки вантажився запит до API. Користувач чекав 152 ms на отримання новин, хоча картинки завантажились за 40 ms. Це спричиняє негативний UX, адже користувач не розуміє, чому у нього відображаються тільки картинки, хоча він чекав на новини. Проте, у React можна застосувати додаткові перевірки, які не додають на сторінку нічого, поки не завантажиться основний контент.

Name	Status	Type	Initia..	Size	Time	Waterfall
image.865af8f9c29...	304	jpeg	reac...	302 B	8 ms	
1.e83ee9b2a59ae3...	304	png	reac...	302 B	38 ms	
2.561e9e72eebd04...	304	png	reac...	301 B	21 ms	
3.07e8860cb03944...	304	png	reac...	302 B	23 ms	
4.04380663aa60f77...	304	png	reac...	302 B	24 ms	
image2.cb2137584...	304	jpeg	reac...	302 B	72 ms	
image3.cb2137584...	304	jpeg	reac...	302 B	77 ms	
image1.cb2137584...	304	jpeg	reac...	302 B	36 ms	

Рисунок 15 - Час завантаження картинок

Рисунок 16 - Час виконання запиту

На рисунку 17 бачимо, що користувач чекав 4 секунди на те, щоб побачити хоч щось на сторінці, і 72.2 секунди, щоб побачити найважчі за обсягом елементи. Загальний час блокування сторінки 940ms.

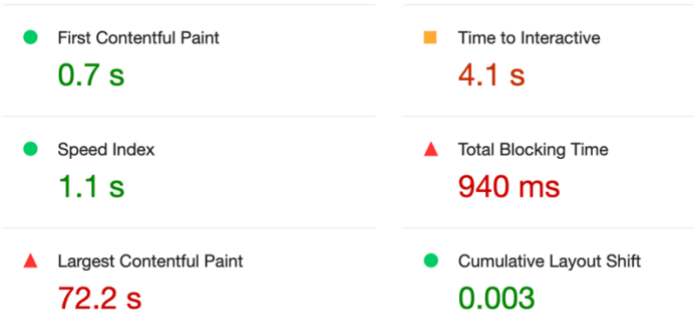


Рисунок 17 - Додаткова аналітика Lighthouse

2.2 Підхід SSR

2.2.1 Розробка міні-проекту на React та Next.js

В минулому розділі я описала особливості роботи з “чистим” React, тож зараз я зосереджусь на фреймоворку Next.js для розробки на React. Створення проекту починається з команди “*npx create-next-app*”. Аналогічно до минулого прикладу, виконання команди створює простенький застосунок, з якого швидко можна почати роботу (рисунок 18).

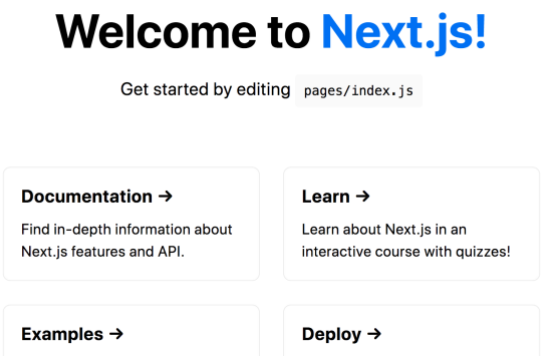


Рисунок 18 - Стандартний create-next-app проект

На рисунку 19 я додала структуру своїх файлів. В Next.js обов’язковим є використання папки *pages*. Всі файли в цій папці – окремі сторінки. Навігація по ним відбувається по назві цих файлів. Наприклад, файл з назвою *feeds.js* відповідає за сторінку з новинами, яку можна побачити за посиланням /feeds. Якщо ж посилання має бути динамічним та залежати від параметрів в назву додаються квадратні дужки []. Текст написаний в них відповідає параметру. Так сторінку для статті по посиланню /feed/article_1 ми знайдемо в папці *feed*, у файлі *[slug].js*, де slug – id статті.

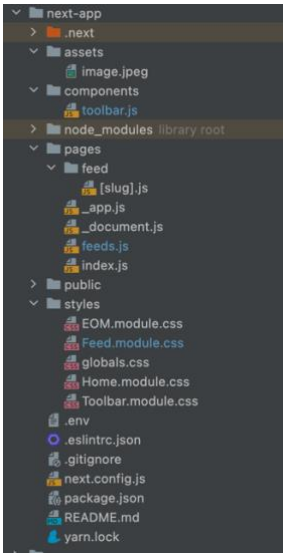


Рисунок 19 - Структура SSR додатку

Фреймоворк Next.js містить в собі декілька функцій для отримання даних різними типами рендеру:

- ***getServerSideProps*[18]** - коли користувач заходить на сторінку, ця функція автоматично виконується безпосередньо *під час запиту*. Сторінка попередньо відтворюється (pre-render) вже з отриманими даними. Працює лише на стороні сервера і ніколи не запускається у браузері. Додати її можна лише на окрему сторінку (папка pages). Цю функцію варто використовувати тоді, коли дані мають бути отримані обов’язково *під час запиту*, наприклад заголовки для авторизації.

```
export async function getServerSideProps(context) {
  return {
    props: {}, // буде передано компоненту сторінки як props
  }
}
```

- ***getStaticProps*[19]** - виконання цієї функції відбувається під час збірки проекту (build time). Next.js попередньо відтворює сторінку під час білду, використовуючи дані в props, які отримуються після виконання функції getStaticProps. Цю функцію слід використовувати, коли дані, потрібні для відображення, доступні *під час білду* (до запиту користувача), сторінка має бути оптимізованою для CEO і дуже швидкою. Оскільки getStaticProps викликається один раз під час збірки, дані теж отримуються лише один раз. Проте це можна змінити, додавши додаткову статичну регенерацію[20] на вказаний період часу. Наприклад оновлювати виклик функції раз на хвилину. Для проектів, які не плануються часто оновлюватись (інтернет-магазин) це чудове рішення.

```
export async function getStaticProps(context) {
  return {
    props: {}, // буде передано компоненту сторінки як props
    revalidate: 60, // оновлення раз на хвилину
  }
}
```

- ***getStaticPaths*[21]** - якщо у сторінки є динамічні шляхи, наприклад відображення сторінки зі статтею з унікальним id, та на ній використовується *getStaticProps* для отримання даних, має бути додана функція *getStaticPaths*. В тілі функції визначається список шляхів, які будуть статично згенеровані. Таким чином можна попередньо відтворити сторінки, наприклад, найпопулярніших статей, на які користувач скоріш за все натисне та скоротити час їх завантаження.

```
export async function getStaticPaths() {
  return {
    paths: [
      { params: { ... } }
    ],
    fallback: true // false or 'blocking'
  };
}
```

- **client-side with *useEffect*[22]** - стандартний спосіб отримання даних React. У випадку, коли сторінка не потребує індексації SEO, а її вміст потрібно часто оновлювати зручно використовувати useEffect, який був вже раніше описаний.

Для тестового застосунку, де нам потрібен вивід даних з API одразу при потраплянні на сторінку, найкраще використовувати функцію *getStaticProps*, адже це забезпечить хороше SEO та швидке завантаження.

Ще однією особливістю Next є використання компоненту <Image /> з бібліотеки *next/image*[23] для автоматичної оптимізації картинок. Я використала цей компонент для кожної з картинок для тестування.

2.2.2 Результати роботи та тестування швидкості

Результатом нашої роботи став аналогічний до попереднього застосунок. Проте, якщо ми подивимось page source (рисунок 20), ми побачимо повністю весь код сторінки, включно з JavaScript та CSS. Саме це впливає на оптимальність SSR для SEO.

Рисунок 20 - Page source SSR додатку

Протестувавши його в Lighthouse маємо наступні показники:

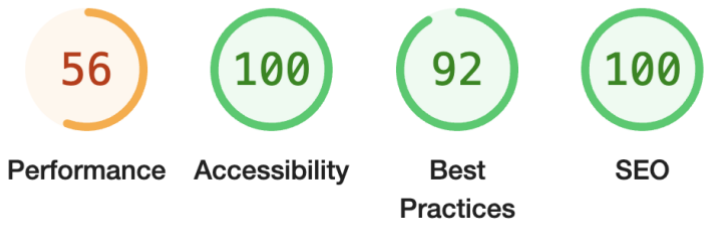
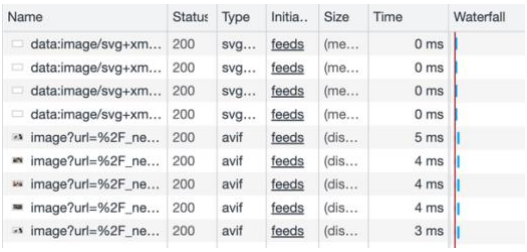


Рисунок 21 - Результат тестування Lighthouse

Одразу порівнюючи з попередніми, можемо зауважити, що відсоток SEO збільшився до 100, а performance лише на декілька відсотків відрізняється. Проте дещо більше про нього нам вказує рисунок 22, на якому вказана послідовність відображення контенту. Основна відмінність у тому, що користувач спочатку не бачив нічого, а потім одразу отримав всю сторінку. Lighthouse оцінив це як негативний варіант, хоча все залежить від того, який UX на думку власників проекту буде краще для користувача.

Рисунок 22 - Послідовність завантаження контенту

У вкладці Network бачимо наскільки змінився час завантаження картинок. З 40-70 ms він знизився до 0. Формат картинок теж змінився. Автоматично завдяки Next з jpeg та png відбулось форматування в svg та avif. Запиту до API в Network відповідно не було, адже він відбувся під час білду, а не на клієнті.



Name	Status	Type	Initia..	Size	Time	Waterfall
data:image/svg+xm...	200	svg...	feeds	(me...	0 ms	
data:image/svg+xm...	200	svg...	feeds	(me...	0 ms	
data:image/svg+xm...	200	svg...	feeds	(me...	0 ms	
data:image/svg+xm...	200	svg...	feeds	(me...	0 ms	
image?url=%2F_ne...	200	avif	feeds	(dis...	5 ms	
image?url=%2F_ne...	200	avif	feeds	(dis...	4 ms	
image?url=%2F_ne...	200	avif	feeds	(dis...	4 ms	
image?url=%2F_ne...	200	avif	feeds	(dis...	4 ms	
image?url=%2F_ne...	200	avif	feeds	(dis...	3 ms	

Рисунок 23 - Час завантаження картинок

Додаткові метрики Lighthouse показують, що у випадку SSR час до моменту першої взаємодії та час отримання найважчих даних однаковий, при чому останній з 72s знизився до 8s, а загальний час блокування менший в 2 рази. Загалом тестування додатку на Next показало кращі результати по швидкості завантаження.

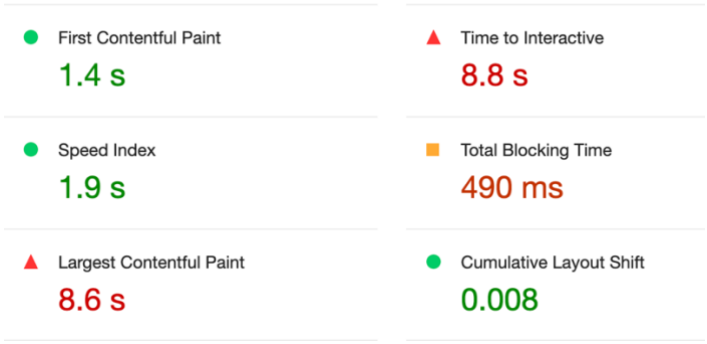


Рисунок 24 - Додаткові метрики Lighthouse

2.3 Висновок. Порівняння підходів. Застосування для кожного підходу. Вибір архітектури для власного додатку.

Дослідивши на практиці обидва підходи, можна їх порівняти та проаналізувати, в яких випадках краще кожен застосовувати.

Client-side rendering (SPA)

Плюси:

- Швидкий рендеринг сайту після першого завантаження, перехід між сторінками;
- Оновлення лише частин сторінки в залежності від змінених даних;
- Активна взаємодія з користувачем;

Мінуси:

- Низький рівень SEO оптимізації;
- Довший час початкового завантаження сторінки;

Цей підхід варто застосовувати, якщо питання SEO та швидкого першого завантаження не є важливим. Найчастіше це буває, коли веб-додаток не потребує реклами і є B2B застосунком. Будувати SPA додаток доцільно, якщо в ньому буде багато вкладок, графіків, взаємодії з користувачем, постійних змін даних, наприклад Dashboard, панель користувача.

Server-side rendering

Плюси:

- Краще SEO для пошукових систем за рахунок наявності статичного коду всієї HTML сторінки;
- Швидке провантаження контенту;

- Користувач має кращий UX досвід, адже отримує одразу цілісну сторінку;

Мінуси:

- Часті запити до серверу (кожен раз, коли користувач заходить на будь-яку сторінку);
- Перезавантаження всієї сторінки у випадку оновлення даних;
- Низька взаємодія з користувачем. Обмеженість дій;

Такий тип архітектури чудово підходить для статичних веб-сайтів, лендінгів, новинних сайтів, блогів, інтернет магазинів (де контент оновлюється раз на день). Тобто для таких типів веб-додатків, де важливим є швидкість першого завантаження (для того, щоб зацікавити клієнта), SEO (щоб сайт відображався при пошуку та в соціальних мережах), де немає необхідності в швидкому та постійному оновленні даних. Тому, наприклад, реалізація панелі відображення з безліччю таблиць і графіків на сторінці через SSR буде значно повільніша, ніж при використанні іншого підходу – SPA.

Варто додати, що фреймворк Next дозволяє поєднувати в собі два види рендерингу. Використовуючи його можна комбінувати SSR та CSR, що є безмежно зручним для будь-яких додатків. Але тим не менш, потрібно розуміти між ними різницю, щоб знати, у яких випадках потрібно використовувати кожен з підходів, навіть у межах одного фреймворку.

Результатом мого дослідження стало чітке усвідомлення різниці в підходах до відображення контенту, а також тестування їх на практиці для розуміння інструментів для роботи з ними. Ґрунтуючись на отриманих результатах, я можу вибрати найкращий підхід для свого проекту. Оскільки реалізовувати в рамках цієї роботи я буду веб-платформу, на якій буде можливість обрати майстра, записатись до нього на послуги, переглянути пропозиції інших майстрів по різним напрямкам тощо, доцільним, на мою думку, є використання SSR підходу та фрейворку Next.js. Більшість сторінок сайту будуть містити дані, отримані по запитам з API. Ці сторінки зручно попередньо відобразити та оновлювати раз на певний час. Основними функціями для користувача є запис на послуги та

перегляд цих записів. Вони не потребують широкої взаємодії та оновлення даних. До того ж моїй платформі важливо відображатись у пошукових системах, адже є мета – залучити користувача, коли він буде шукати майстрів в пошуковій стрічці. Також такі переваги, як оптимізації картинок, швидкість першого завантаження та можливість працювати з запитами даних трьома способами, заохочують почати роботу саме з фреймворком Next.js.

3. РОЗРОБКА ПРОЕКТУ

3.1 Актуальність

Ідея проекту, який я планую реалізувати в рамках цієї роботи, полягає в створенні веб-платформи для майстрів у сфері краси та їх клієнтів. Головна мета цього продукту – полегшити та автоматизувати запис на послуги до приватних майстрів, оптимізувати зберігання всіх записів в одному місці зі зручним відображенням. Свою минулу курсову роботу[1] я присвятила дослідженню підходу Design Thinking для проектування та створення цифрового продукту. В цій роботі я детально описала гіпотезу, аналіз ринку, провела тестування, опитування, визначила біль користувача та знайшла оптимальний варіант для вирішення її. Результатом моєї роботи став детальний набір прототипів[30], окремо протестованих потенційними користувачами та покращених опісля.

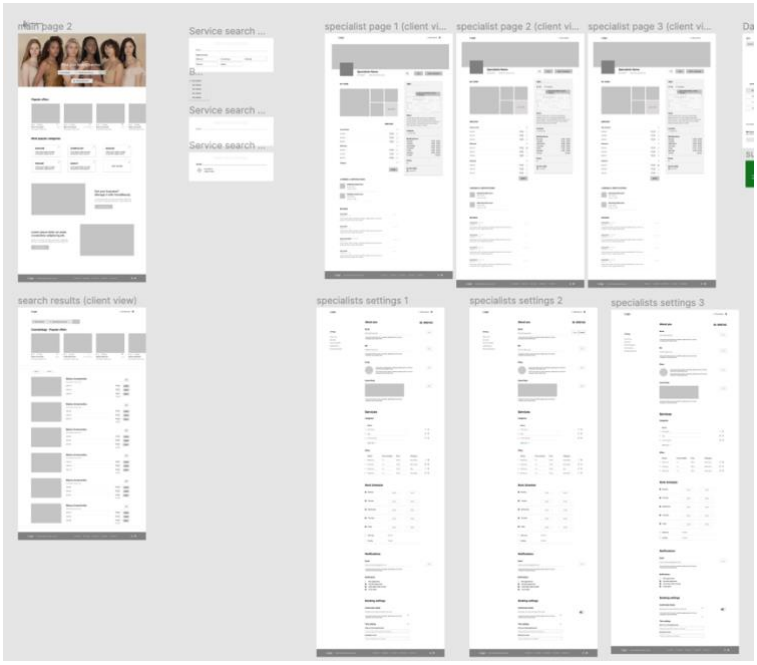


Рисунок 25 - Прототипи веб-додатку

Для детальної інформації щодо актуальності продукту, моїх досліджень предметної області, проведеним опитуванням тощо пропоную переглянути свою минулу роботу[1]. В цій курсовій роботі я зосереджусь саме на розробці свого веб-продукту.

3.2 Архітектура проекту. Монорепозиторій.

Для реалізації описаної вище веб-платформи потрібно написати back-end для роботи з базою даних та front-end для користувацького інтерфейсу. Таким чином проект складатиметься з двох папок, які я вирішила зберігати в одному монорепозиторії. Монорепозиторій – це стратегія, в якій код декількох проектів зберігається в одному git-репозиторії[24].

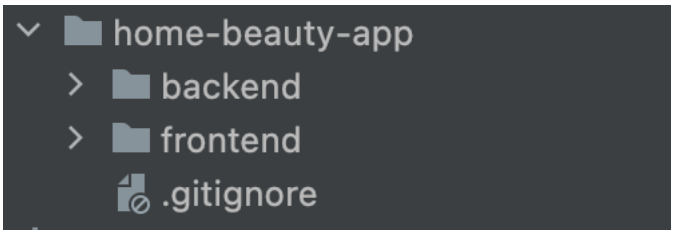


Рисунок 26 - Структура проекту

Для back-end я створила порожню папку та поступово наповнювала її файлами, для front-end я скористалась командою для створення Next додатку та отримала базовий набір файлів, які потім редагувала. Детальна інформація по запуску кожного проекту, встановленні бази даних тощо міститься в файлах README.md.

3.3 Back-end розробка

3.3.1 Використані технології

Мовою для написання програмного коду я обрала TypeScript[25]. Це строго типізована мова програмування, яка базується на JavaScript. Мені подобається функціональна парадигма мови JavaScript, тому у розробці я надаю перевагу саме їй, проте відсутність типізації сприяє великій кількості помилок. TypeScript допомагає уникнути помилок шляхом строгого визначення типів та інтерфейсів. Платформою для виконання коду JavaScipt є Node.js[26]. Щоб відбувалось виконання TypeScript коду у середовищі Node.js, потрібно кожен раз компілювати його в JavaScript.

У якості сервера бази даних для свого застосунку я обрала PostgreSQL. В університеті я працювала саме з реляційними базами даних, тож в проекті буду використовувати їх. Для роботи з реляційними базами зазвичай використовують

MySQL або PostgreSQL. Другий варіант є популярнішими, має більше можливостей та підтримує тип даних JSON. Для зручної роботи з базою даних я буду використовувати ORM Prisma[27]. ORM (Object-relation Modal) – це об’єктно-орієнтована система для управління базою даних. Використання ORM дозволяє розробникам не писати довгі та одноманітні SQL запити. Prisma найпопулярніша ORM для роботи з Node.js і TypeScript. На рисунку 27 зображений приклад створення сутності User.

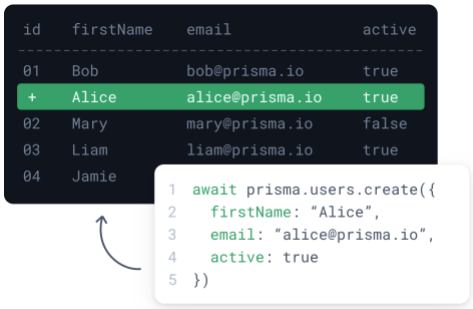


Рисунок 27 - Створення User у Prisma

3.3.2 Структура проекту

Мій back-end проект складається з папки *src* та файлів конфігурації, таких, як *package.json, tsconfig.json* тощо. Папка *src* містить файл *server.ts*, в якому запускається веб-сервер; папку *logger*, в якій міститься конфігурації всіх текстових виводів для помилок та повідомлень; папку *db*, в якій зберігаються всі міграції та схема моделей бази данних у Prisma – *schema.prisma*; папка *dal* для підключення PrismaClient. Також в папці *src* є набільша за обсягом папка *api*. В ній міститься папка з user-dto, папка з функціями-утилітами, такими як валідація, обгортки для запитів, функції для middleware. Папка *middlewares* для функцій проміжної обробки авторизаційних запитів, обробки помилок тощо. Та найбільша папка *endpoints*, яка містить в собі файл *endpoint.ts*, в якому ініціалізується кожен endpoint з вкладених папок. В кожній вкладеній папці теж є файл, в якому описані всі шляхи запитів. На рисунку 29 зображений файл *user-endpoints*. З нього видно, що за шляхом “/users/popular” відпрацьовує *getPopularHandler*, який зберігається в папці *get-popular-users* та повертає усіх

users, оцінка яких вища за 4.8. Аналогічну до цього прикладу логіку мають всі інші файли папки *endpoints*.

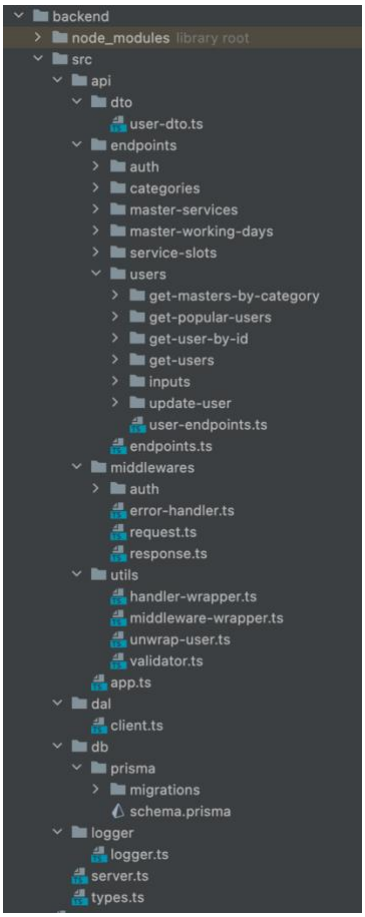


Рисунок 30 - Структура back-end

```
const path = '/users';

export function initUserEndpoints(app: Application): void {
  app.get(path, getUsersHandler);
  app.get(`${path}/popular`, getPopularHandler);
  app.get(`${path}/:id`, getUserByIdHandler);
  app.get(`${path}/category/:id`, getMastersByCategoryHandler);
  app.put( path: `${path}/:id`, updateUserHandler);
}
```

Рисунок 29 - Файл user-endpoints.ts

```
  const users = await prisma.user.findMany({
    where: { rate: { gt: 4.8}}
  });
  logger.info( message: 'Fetched users from DB');

  res.send(users);
}

export const getPopularHandler = wrapHandler(handler);
```

Рисунок 28 - Запит до БД

3.3.3 База даних

Перед розробкою бази даних та написання запитів до неї, я розробила схему реляційної бази даних для свого проекту в онлайн застосунку Lucidchart[28] (рисунок 31). Детальний опис цієї схеми я робила у своїй минулій роботі[1]. В процесі розробки назви деяких атрибутів були змінені, додані або видалені, адже іноді під час написання коду з’являються кращі ідеї.

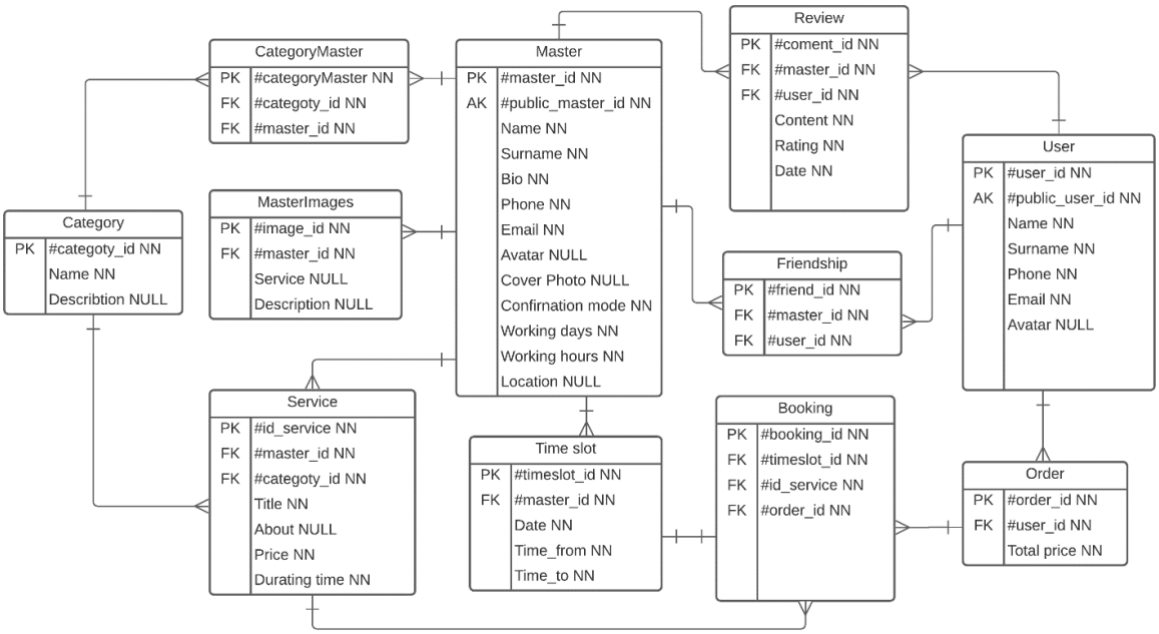


Рисунок 31 - Схема бази даних

Кожну модель бази даних я описала в файлі *schema.prisma*, щоб ORM згенерувала класи для роботи з SQL запитами. На рисунку 31 описана модель ServiceSlot, в якій зберігається вся інформація про запис на послугу. На рисунку 32 модель MasterService. Ці дві моделі поєднуються зв’язком “one to many”. У ServiceSlot може бути лише один *service*. А у MasterService може бути багато *masterSlots*. В Prisma це описується додаванням до ServiceSlot id послуги по якому буде відбуватись зв’язок (*serviceId*), а також визначенням послуги, додавши @relation до відповідної моделі (*service MasterService? @relation(fields: [serviceId], references: [id])*). До MasterService додається масив з записами (*masterSlots*). Повну документацію по опису зв’язків можна прочитати в офіційній документації Prisma.

```
model ServiceSlot {
  id Int @id @default(autoincrement())

  date DateTime
  slotStartTime String
  slotEndTime String

  serviceId Int?
  service MasterService? @relation(fields: [serviceId], references: [id])

  masterId Int
  master User? @relation("master", fields: [masterId], references: [id])

  clientId Int?
  client User? @relation("client", fields: [clientId], references: [id])
}
```

Рисунок 32 - Модель ServiceSlot

```
model MasterService {
  id Int @id @default(autoincrement())

  name String
  duration String
  price Int

  masterId Int
  master User? @relation(fields: [masterId], references: [id])

  masterSlots ServiceSlot[]
}
```

Рисунок 33 - Модель MasterService

Після кожного оновлення моделі в файлі *schema.prisma* потрібно створювати нову міграцію командою "*npx prisma migrate dev --schema ./src/db/prisma/schema.prisma*". Після цього потрібно перегенерувати ORM моделі, використовуючи команду "*npm run generate*". Ці команди описані у файлі README.md для зручності під час розробки.

3.3.4 Опис запитів до БД

Як я вже описувала раніше, всі endpoints та шляхи до них описані в папці *src->api->endpoints*. Ця папка логічно розділена на 5 менших. Окремо відокремлені маршрути для авторизації (папка *auth*), взаємодії з моделлю User (папка *users*), моделлю ServiceSlot (*service-slots*), моделлю Category (*categories*), моделлю MasterService (*master-services*), моделлю MasterWorkingDay (*master-working-days*). Нижче опишу типові приклади створення endpoints та запитів до БД.

- Приклад маршрутів для service-slots endpoints;

```
const path = '/service-slots';

export function initServiceSlotsEndpoints(app: Application): void {
  app.get(path, getSlotsHandler);
  app.get(`${path}/by-clients/:clientId`, getSlotsByClient);
  app.get(`${path}/by-masters/:masterId`, getSlotsByMaster);
  app.get(`${path}/:masterId/:date`, getSlotsByDateHandler);
  app.post(path, addSlotHandler);
  app.post( path: `${path}/:id`, bookSlotHandler);
}
```

- Приклад запиту на отримання усіх service-slots по id клієнта;

```
async function handler(req: Request, res: Response): Promise<void> {
  const {clientId} = req.params;

  const serviceSlots = await prisma.serviceSlot.findMany({
    where: {
      clientId: parseInt(clientId),
    }
  });
  logger.info( message: 'Fetched serviceSlots from DB');

  res.send(serviceSlots);
}
```

- Запит на бронювання послуги (оновлення);

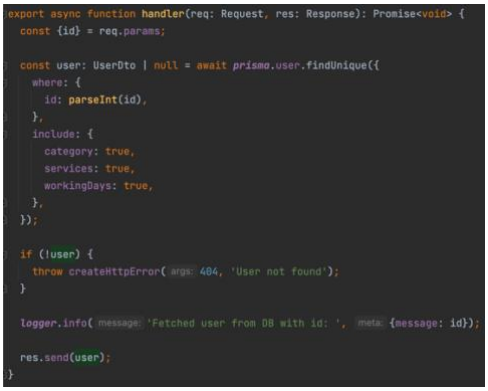
```
async function handler(req: Request, res: Response): Promise<void> {
  const {id} = req.params;

  const updatedSlot = await prisma.serviceSlot.update({
    where: {id: parseInt(id)},
    data: req.body
  });

  logger.info( message: 'Updated serviceSlot into DB with id',  meta: {message: updatedSlot.id});

  res.send(updatedSlot);
}
```

- Запит на отримання користувача по id, а також усіх його категорій, послуг та днів роботи;



```
export async function handler(req: Request, res: Response): Promise<void> {
  const {id} = req.params;

  const user: UserDto | null = await prisma.user.findUnique({
    where: {
      id: parseInt(id),
    },
    include: {
      category: true,
      services: true,
      workingDays: true,
    },
  });

  if (!user) {
    throw createHttpError( args: 404, 'User not found');
  }

  logger.info( message: 'Fetched user from DB with id: ',  meta: {message: id});

  res.send(user);
}
```

- Запит на додавання доступного запису на послугу;

```
async function handler(req: Request, res: Response): Promise<void> {
  const {date, slotStartTime, slotEndTime, serviceId, masterId, clientId} = req.body;

  const serviceSlot = await prisma.serviceSlot.create({
    data: {
      date, slotStartTime, slotEndTime, serviceId, masterId, clientId
    },
  });
  logger.info( message: 'Inserted new serviceSlot into DB with id',  meta: {message: serviceSlot.id});

  res.send(serviceSlot);
}
```

3.4 Front-end розробка

3.4.1 Використані технології

У минулих розділах я аргументувала та детально описала використання JavaScript бібліотеки React та фреймворку Next.js для розробки front-end застосунку. Основною мовою розробки я обрала TypeScript по тим же причинам, що і на бекенді. А також через додаткову зручність визначати інтерфейси об'єктам, отриманим з бекенду та тим самим зменшувати кількість помилок.

Для форматування коду, виявлення помилок, дотримання одного шаблону та код-стилю, покращення якості коду я використовую ESLint[29]. Це технологія, яка статично аналізує код та автоматично виправляє всі помилки або залишає вказівки до виправлення. Конфігурації для ESLint можна обрати стандартні, наприклад конфіг Airbnb "*eslint-config-airbnb*", а також можна їх розширювати власними правилами та плагінами. Перш за все я додала плагін для перевірки помилок для TypeScript.

Для зручності роботи зі стилями у проекті я використовую SCSS для написання стилів і CSS-модулі для їх зберігання. CSS-модулі – це файли, які мають назву на кшталт "*PageStyle.modules.scss*", і у яких назви класів мають локальну область. До кожного класу додається унікальний хеш, який відображається у браузері наступним чином:

```
▶ <div class="SingleChoiceAnswer_checkmark_wrapper__v6680">…</div>
```

Такий підхід дозволяє мати безліч однакових за назвою класів з різними стилями.

3.4.2 Структура проекту

Файлова структура фронтенд-проекту (рисунок 37) складається з файлів-конфігурацій Next, ESLint, TypeScript, *package.json*, файлу *_types.ts* з визначенням типів та папок.

У файлі *_types.ts* визначені інтерфейси, які використовуються по всьому проектові. В своїй назві вони мають приставку "I", а в тілі містять поля і їх типізацію.

```
export interface ISlot {
  id: number;
  date?: string;
  slotStartTime: string;
  slotEndTime: string;
  clientId?: number;
}
```

Рисунок 34 - Файл types.ts

Серед директорій проекту є (рисунок 37): окрема папка *styles* для загальних стилів, змінних та *breakpoints*; папка *assets* для збереження картинок; папка *data* для масивів з локальними, зазвичай тестовими, даними; папка *services* для відокремлення запитів до API.

Детальніше варто роздивитись папку *pages* (рисунок 36). Кожен файл цієї папки – окрема сторінка, а його назва – шлях до неї. Наприклад, перейти на сторінку логіну можна за шляхом /login. Вкладені у папки файли відображають вкладеність шляху. Назви файлів у квадратних лапках означають, що шлях до них має динамічну зміну. Так файл *[masterId].tsx* є сторінкою окремого майстра, на яку можна перейти за шляхом /masters/master_id. На рисунку 36 зображені всі сторінки мого веб-додатку: домашня, сторінка логіну та реєстрації, особисті кабінети майстра та клієнта, сторінка майстра з детальною інформацію на ній та можливістю записатись на послугу та сторінка категорії, на якій запропоновані майстри цієї категорії. В файлах папки *pages* компоненти виконують роль обгорток. Їх основна функція – отримати дані з бекенду та передати їх у внутрішні компоненти, які вже відображають ці дані відповідно до дизайну та зберігаються в папці *components*.

На рисунку 35 зображена папка *components*. В ній є під-папка *atoms* для збереження універсальних маленьких UI компонентів, таких як кнопка. А також під-папка для кожної зі сторінок. Всередині цих папок для кожного внутрішнього компонента теж створюється окрема директорія, в якій вже зберігається файл tsx та файл зі стилями. Є багато підходів для організації файлової структури фронтенд-додатків, проте цей, на мою думку, найпростіший та найзручніший.

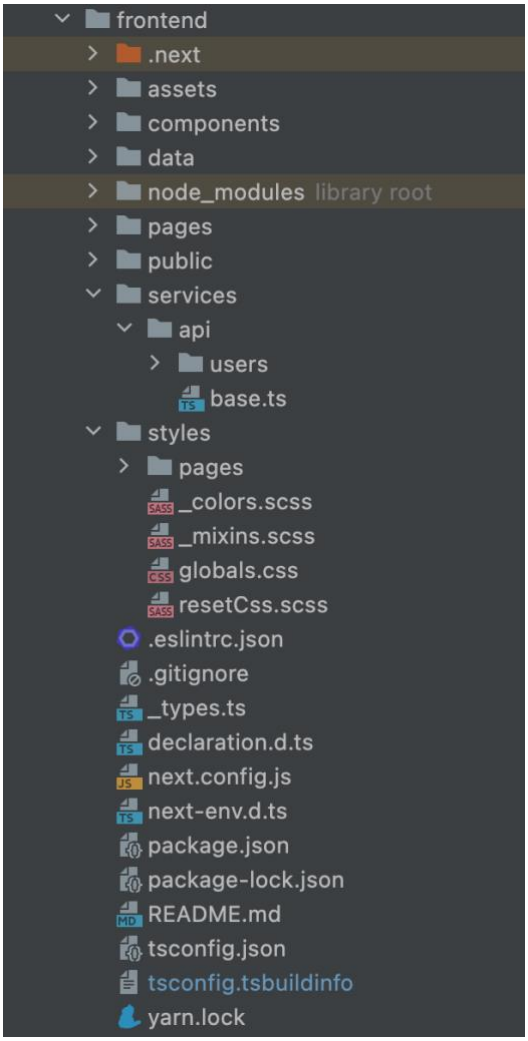


Рисунок 37 - Структура проекту

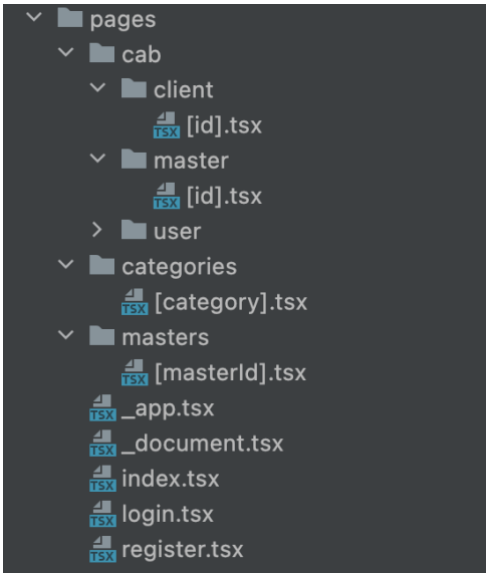


Рисунок 36 - Папка Pages

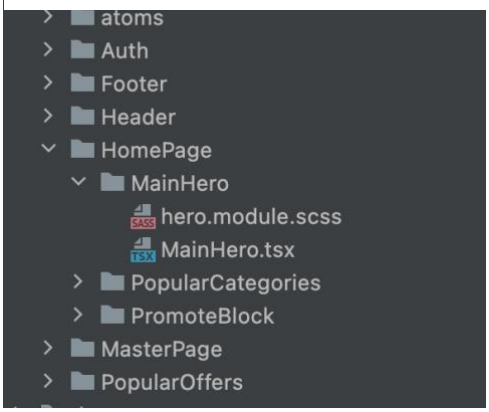


Рисунок 35 - Папка components

3.4.3 Логін та Реєстрація

Основна логіка авторизації користувачів відбувається на сторінках Login та Register. На першій можна увійти в систему, на другій відповідно зареєструватись у системі. На рисунку 38 зображений приклад обробки натискання на кнопку підтвердження реєстрації. Якщо всі поля введені успішно, користувач реєструється у системі та потрапляє на сторінку логіну. У випадку некоректних даних або серверних помилок на кшталт дублювання email, юзер отримує повідомлення про помилку. На рисунку 39 – приклад запиту до back-end з папки *services*. В даному випадку це запит на створення (реєстрацію) нового користувача.

```
const handleSubmit = async (
  e: ChangeEvent<HTMLInputElement>
): Promise<void> => {
  e.preventDefault();

  createUser( data: {
    ...values,
  }) Promise<any>
    .then((data) => {
      if (data?.error) {
        throw new Error(data?.error?.message);
      }
      router.push( url: '/login');
    }) Promise<void>
    .catch((err) => {
      setError(err.toString());
    });
};
```

Рисунок 38 - Обробка кліку на реєстрацію

```
export const createUser = async (data: any): Promise<any> => {
  const url = `${API_URL}/register`;

  const response = await fetch(url, init: {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
  });

  return response.json();
};
```

Рисунок 39 - Запит на створення нового користувача

На сторінці логіну користувача обробка входу по email та пароль відбувається так, як зображено на рисунку 42. Надсилається запит на бекенд для спроби авторизації, у разі успішної відповіді від сервера разом із нею на клієнт відправляється унікальний jwt-токен. Цей токен декодується за допомогою бібліотеки jwr_decode. З нього отримується user.id, user.role, які разом з токеном додаються у localStorage функціями setLokalToken (рисунок 40), setUserRole (рисунок 41), setUserId. Після успішної авторизації користувач потрапляє на головну сторінку.

```
const handleSubmit = async (
  e: ChangeEvent<HTMLInputElement>
): Promise<void> => {
  e.preventDefault();

  authorizeUser(values) Promise<any>
    .then(async (data) => {
      if (data?.error) {
        throw new Error(data?.error?.message);
      }

      const user: USER = jwt_decode(data.token);

      setUserId(user.id);
      setUserRole(user.role);
      setLocalToken(data.token);

      await router.push( url: '/');
      router.reload();
    }) Promise<void>
    .catch((err) => {
      setError(err.toString());
    });
};
```

Рисунок 42 - Обробка входу в акаунт

```
export const setLocalToken = (value: string): void => {
  return localStorage.setItem('home_user_token', value);
};

export const getLocalToken = (): string | null => {
  return localStorage.getItem( key: 'home_user_token');
};
```

Рисунок 40 - Функції збереження токену користувача

```
export const setUserRole = (value: string): void => {
  return localStorage.setItem('home_user_role', value);
};

export const getUserRole = (): string | null => {
  return localStorage.getItem( key: 'home_user_role');
};
```

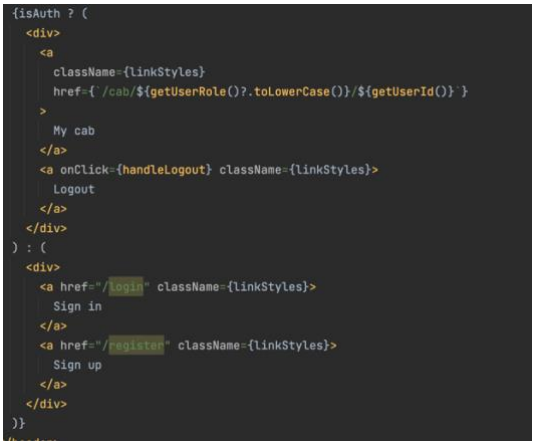
Рисунок 41 - Функції збереження ролі користувача

На головній сторінці, а також у місцях, де це потрібно, визначається роль користувача та його статус (залогінений чи ні). За допомогою функцій getLokalToken (рисунок 40), getUserRole (рисунок 41) отримуються дані з localStorage або порожній рядок, якщо користувач неавторизований і localStorage порожній. На рисунку 43 зображена функція перевірки, чи користувач авторизований та зміни стану компоненту в залежності від цього. Якщо isAuth має булеве значення true, це означає, що користувач залогінений та через шапку сайту може потрапити у свій кабінет або вийти з акаунту. Якщо isAuth має значення false, користувачу пропонується перейти на сторінку логіну або реєстрації. Токен, отриманий з localStorage використовується для авторизованих запитів. Він додається в Header запиту.

При виході з акаунті очищаються всі дані з localStorage (рисунок 44).

```
const checkToken = (): void => {
  if (getLocalToken()) {
    setAuth( value: true);
  }
};
```

Рисунок 43 - Перевірка, чи користувач авторизований




```
const handleLogout = (): void => {
  removeLocalToken();
  removeUserId();
  removeUserRole();
  window.location.reload();
  router.push( url: '/');
};
```

Рисунок 44 - Очищення LocalStorage після виходу

Рисунок 45 - Логіка опрацювання значення isAuth

3.4.4 Отримання даних з back-end

В розділі 2.2.1 я описувала, що Next.js підтримує декілька варіантів отримання даних з API. Здебільшого у проекті я використовувала функцію *getStaticProp*, приклад використання якої зображений на рисунку 14. Це скріншот з домашньої сторінки, під час білду якої відпрацьовує *getStaticProp* та відправляються два запити на бекенд для отримання категорій та популярних майстрів. Результати, отримані у відповідь від API, передаються в props компоненту домашньої сторінки під назвами *categories, popularMasters*.

```
export const getStaticProps: GetStaticProps = async () => {
  const categoriesJson = await fetch( input: `${API_URL}/categories`);
  const categories = await categoriesJson.json();

  const popularMastersJson = await fetch( input: `${API_URL}/users/popular`);
  const popularMasters = await popularMastersJson.json();
  return {
    props: { categories, popularMasters },
  };
};
```

Рисунок 46 - Використання getStaticProps

Рисунок 47 ілюструє використання функції *getStaticProps* на сторінці, яка має динамічну зміну – *[category].tsx*. Для попереднього відображення сторінок з id – 1, 2, 3 використовується функція *getStaticPaths*. Так користувач може миттєво попасти на визначені сторінки. Це досить зручно, адже найчастіше можна спрогнозувати, куди натисне користувач на головній сторінці. Аналогічно до цих двох випадків відбуваються всі запити веб-платформи.

```
export const getStaticProps: GetStaticProps = async (context : GetStaticPropsContext<ParsedUrlQuery, PreviewData> ) => {
  const categoryId = context?.params?.category;
  const mastersRes = await fetch( input: `${API_URL}/users/category/${categoryId}` );
  const masters = await mastersRes.json();

  const categoryRes = await fetch( input: `${API_URL}/categories/${categoryId}` );
  const category = await categoryRes.json();

  const popularMastersJson = await fetch( input: `${API_URL}/users/popular` );
  const popularMasters = await popularMastersJson.json();
  return {
    props: { masters, category, popularMasters },
  };
};

export const getStaticPaths: GetStaticPaths = () => {
  return {
    paths: [
      { params: { category: '1' } },
      { params: { category: '2' } },
      { params: { category: '3' } },
    ],
    fallback: false,
  };
};
```

Рисунок 47 - Використання getStaticPaths

3.5 Демонстрація проекту

1. Сторінка реєстрації користувача /register

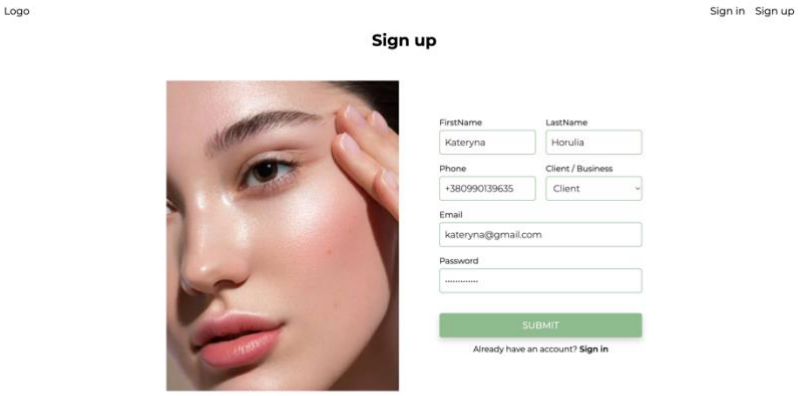


Рисунок 48 - Реєстрація користувача

2. Сторінка авторизації /login

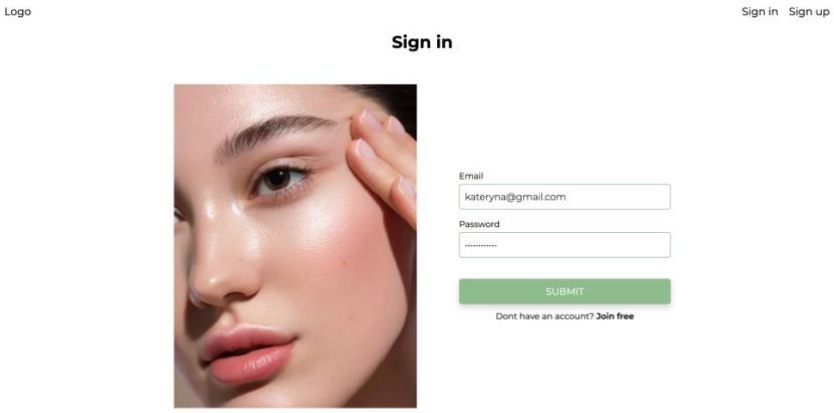
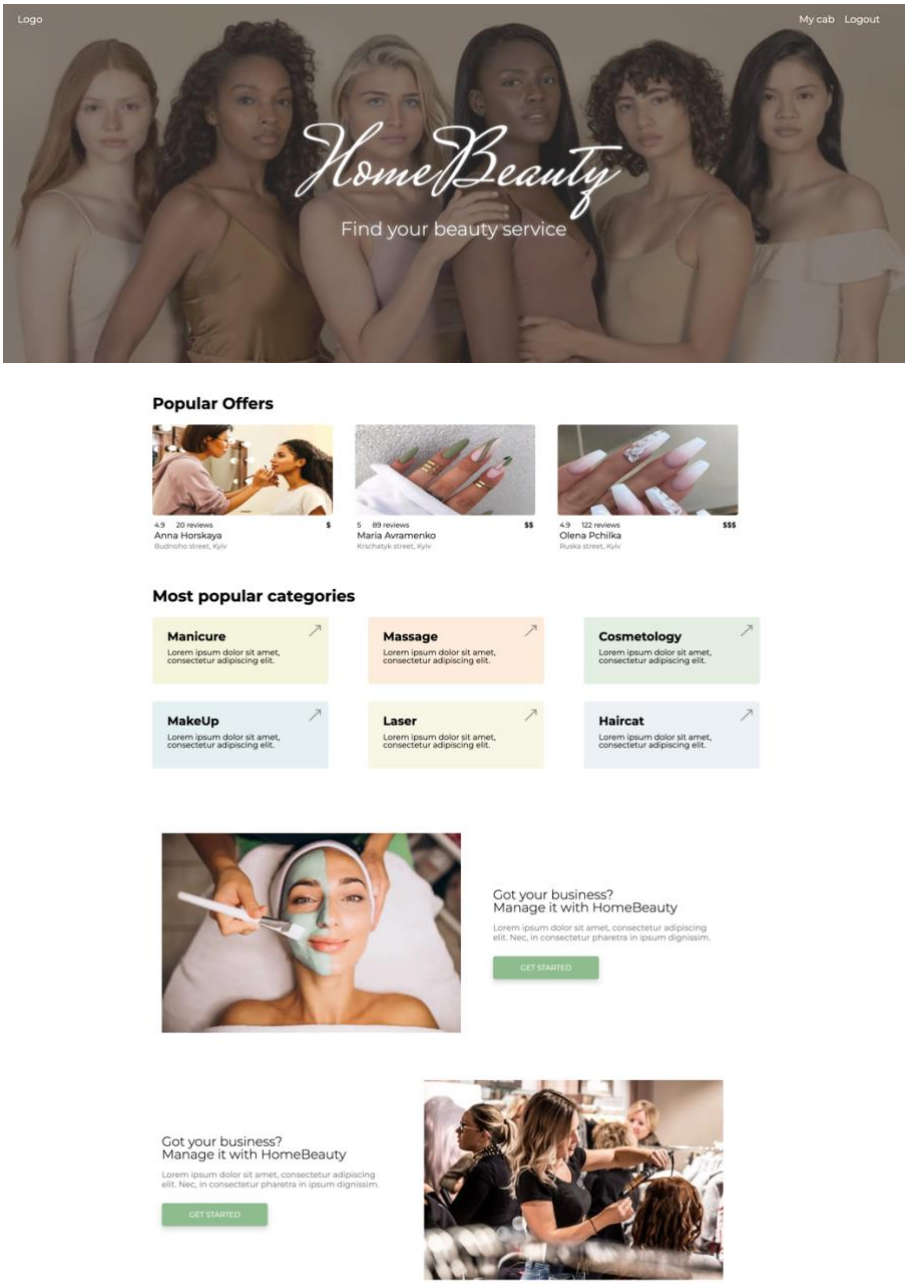


Рисунок 49 - Авторизація користувача

3. Домашня сторінка, на яку користувач потрапляє після успішної авторизації. Якщо він не авторизований, йому також доступна ця сторінка, але праворуч згори недоступний вхід в особистий кабінет. На цій сторінці користувачеві пропонуються популярні майстри різних категорій, а також основні категорії, за якими можна обрати потрібну послугу та знайти майстра.



4. Персональна сторінка майстра /masters/master_id. На рисунку 50 у майстра ще немає фотографій, тому квадратики заповнені сірим кольором. Проте, у цього майстра доступні послуги, на які можна записатись. Вказані години роботи, адреса тощо.

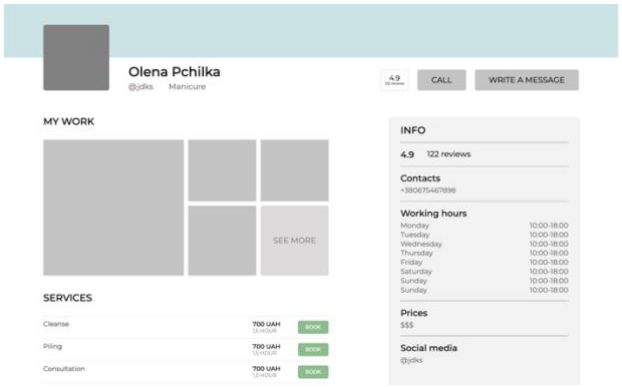


Рисунок 50 - Персональна сторінка майстра

5. Сторінка окремої категорії /categories/categoryId. На рисунку 51 зображена категорія – Manicure. На сторінку додані популярні майстри цієї категорії, а також перелік майстрів, до яких можна одразу записатись.

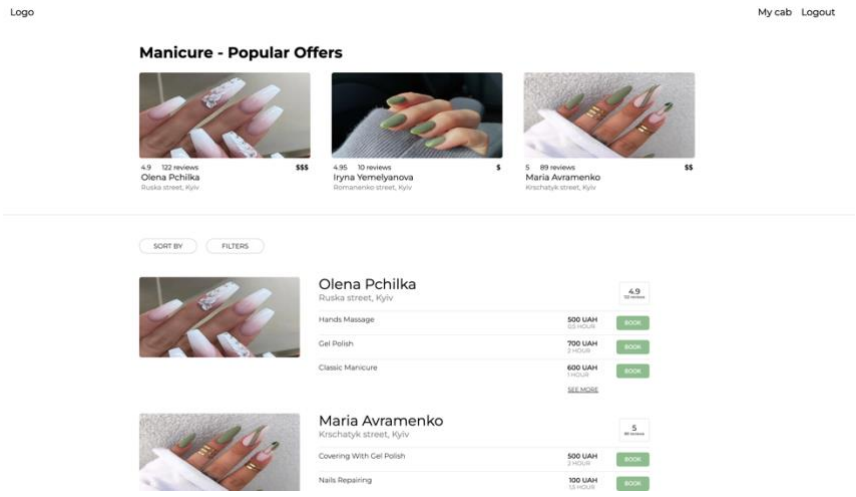


Рисунок 51 - Сторінка категорії Manicure

6. При натисканні на кнопку BOOK відкривається вікно для бронювання відповідної послуги. В залежності від обраної дати, користувачу пропонуються вільні слоти, які створив майстер. Обравши зручний час можна підтвердити запис на послугу, натиснувши на кнопку CONFIRM. На рисунку 52 представлені вільні місця для запису на 21 травня. На рисунку 53 – на 22 травня. Як бачимо, 22 травня більш популярний день для відвідувань і кількість вільних слотів для запису менша. На рисунку 54 – скріншот повідомлення про успішний запис на послугу та основна інформація про нього.

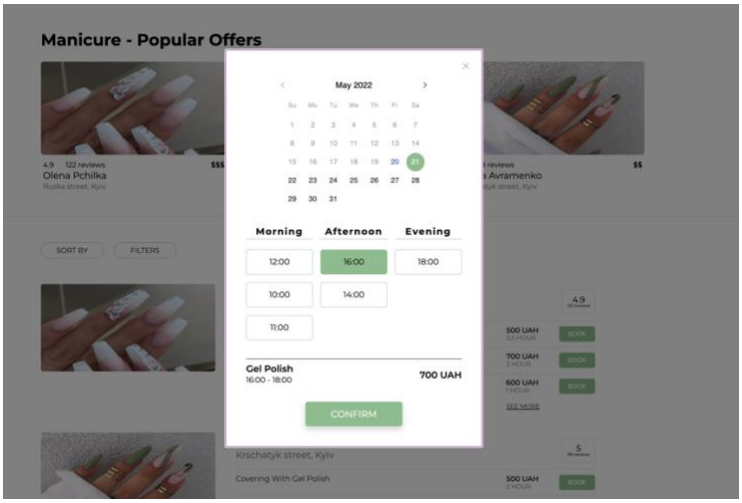


Рисунок 52 - Вікно бронювання послуги 21.05

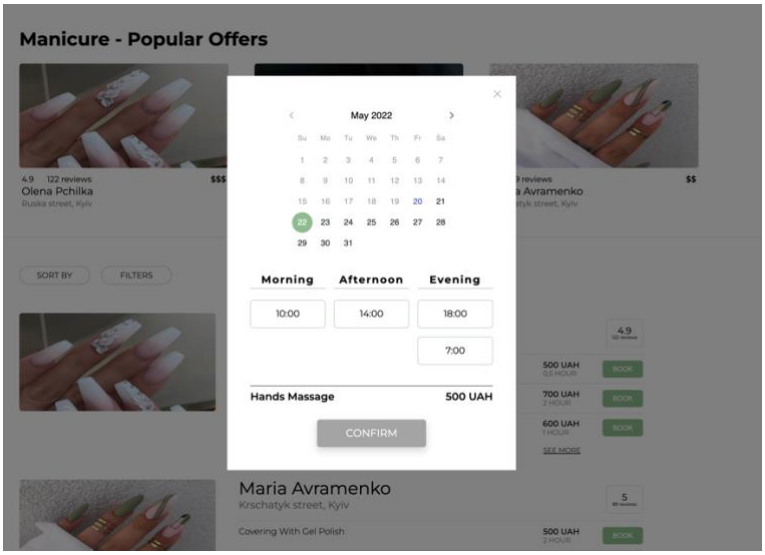


Рисунок 53 - Вікно бронювання послуги 22.05

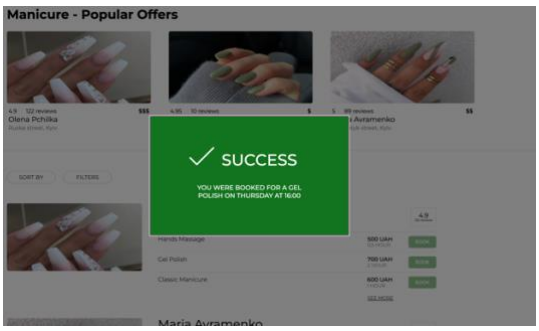


Рисунок 54 - Успішне бронювання послуги

7. Сторінка особистого кабінету користувача з переліком запланованих записів та тих, які вже відбулися /cab/client/clientId.

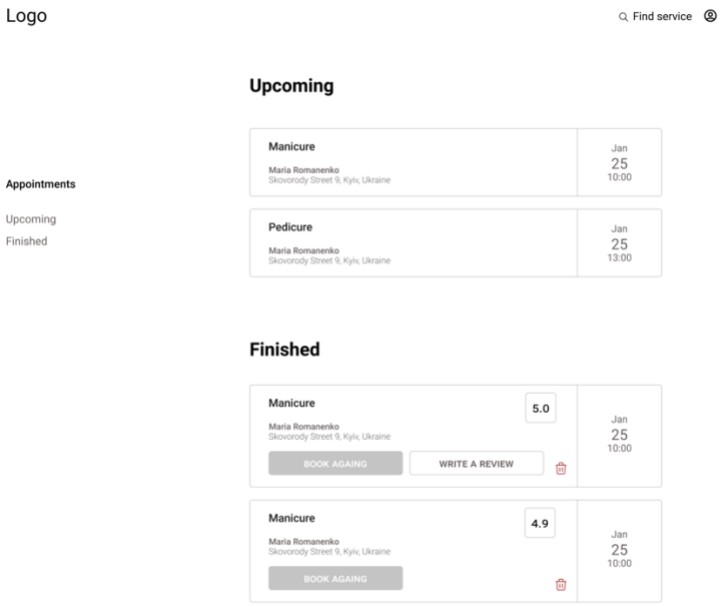


Рисунок 55 - Особистий кабінет

ВИСНОВОК

Під час написання цієї курсової роботи, я детально розібралась в роботі браузера і взаємодії клієнт-сервер у веб-додатках. Я ознайомилась з тим, як відбувається відмальовування сторінок у браузері, що таке DOM і CSSOM дерева, як оновлюються елементи на сторінці. З'ясувавши, що рендеринг елементів становить найбільше навантаження для роботи браузера, я зрозуміла причини використання сучасних фрейморків і бібліотек, в яких на заміну реального DOM-дерева прийшла віртуальна копія JavaScript об'єктів VDOM.

Основним предметом моїх дослідженнь були підходи до рендерингу контенту у браузері – SSR та CSR (SPA). Я вивчила та описала відмінності їх роботі. Головна різниця полягає в тому, що в SSR підході користувач отримує готову відмальовану на сервері HTML-сторінку, а в SPA – чекає, поки сторінка за допомогою JavaScript відмалюється у браузері. Ця різниця впливає на такі характеристики веб-сайту, як SEO, швидкість завантаження, технології для розробки. Щоб краще дослідити різницю цих клієнтських архітектур, я провела своє тестування кожної з них. Я створила два ідентичних проекти, використовуючи різні підходи. Для реалізації я обрала React та Next.js, попередньо описавши особливості цих технологій. Після завершення розробки, я протестувала свої міні-проекти у Lighthouse. В результаті цього дослідження, я познайомилась з новими технологіями, самостійно на практиці переконалась у різниці між підходами, протестувала їх швидкість. Найголовніше, після свого дослідження я змогла аргументовано обрати підхід для свого веб-додатку і у майбутньому теж зможу це зробити з будь-яким застосунком.

Обравши найкращий підхід для розробки мого проекту – веб-платформи для запису на косметичні послуги, я почала написання коду. В рамках курсової роботи я створила реляційну базу даних PostgreSQL, додала ORM Prisma для полегшення взаємодії з нею, написала back-end для запитів до бази даних на TypeScript та Node.js. Я розробила front-end відповідно до протестованих прототипів у минулій курсовій роботі, використовуючи для цього React, Next.js, TypeScript, ESLint.

В результаті своєї роботи, я отримала готову функціональну платформу, на якій можна зареєструватись як клієнт, переглядати пропозиції майстрів, записуватись до них на послуги, шукати майстрів по категоріям, тримати свої записи в зручному особистому кабінеті з приємним інтерфейсом. Також на ній можна зареєструватись як майстер, створювати слоти для записів на послуги, додавати інформацію про себе, свої роботи тощо. У майбутньому я планую розширювати функціональність проекту, продовжити front-end та back-end розробку. Окрім готового full-stack додатку, я, завдяки своєму дослідженню, здобула знання, як легко та аргументовано визначати потребу у використанні SSR чи SPA підходу.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Дослідження підходу Design Thinking для створення успішного цифрового продукту [Електронний ресурс] // Горуля Катерина – 2021. – Режим доступу до ресурсу: http://ekmair.ukma.edu.ua/bitstream/handle/123456789/22052/Horulia_Bakalavrska_robota.pdf
2. Distributed Application Architecture [Електронний ресурс] // Sun Microsystem – 2011. – Режим доступу до ресурсу: https://web.archive.org/web/20110406121920/http:/java.sun.com/developer/Books/jdbc/ch07.pdf
3. How the Web works [Електронний ресурс] // MDN contributors – 2022. – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Learn/Getting_started_with_the_web/How_the_Web_works
4. Transmission Control Protocol/Internet Protocol [Електронний ресурс] // The Editors of Encyclopaedia Britannica – Режим доступу до ресурсу: https://www.britannica.com/technology/TCP-IP
5. What is DNS? | How DNS works [Електронний ресурс] // Cloudflare – Режим доступу до ресурсу: https://www.cloudflare.com/learning/dns/what-is-dns
6. Anatomy of a Compiler [Електронний ресурс] // The Editors of Encyclopaedia Britannica – 2020. – Режим доступу до ресурсу: http://www.cs.man.ac.uk/~pjj/farrell/comp3.html
7. What is Node [Електронний ресурс] // Techmonitor – Режим доступу до ресурсу: https://techmonitor.ai
8. SEO vs React response // Barry Adams – 2017. – Режим доступу до ресурсу: https://medium.com/@badams/another-very-late-response-youre-confusing-googlebot-the-crawler-with-caffeine-google-s-b9ef24d81524
9. CLEAN URLS – Режим доступу до ресурсу: https://www.hacksplaining.com/glossary/clean-urls

10. React Documentation – Режим доступу до ресурсу: https://reactjs.org/docs/hooks-effect.html
11. Most used web frameworks among developers worldwide, as of 2021// Lionel Sujay Vailshery – 2022. – Режим доступу до ресурсу: https://www.statista.com/statistics/1124699/worldwide-developer-survey-most-used-frameworks-web
12. NewsAPI Documentation – Режим доступу до ресурсу: https://newsapi.org/
13. Webpack Documentation – Режим доступу до ресурсу: https://webpack.js.org/
14. Стан та життєвий цикл компонентів // React Documentation – Режим доступу до ресурсу: https://uk.reactjs.org/docs/state-and-lifecycle.html
15. Using the Effect Hook // React Documentation – Режим доступу до ресурсу: https://reactjs.org/docs/hooks-effect.html
16. Using the State Hook // React Documentation – Режим доступу до ресурсу: https://reactjs.org/docs/hooks-state.html
17. Lighthouse Documentation – Режим доступу до ресурсу: https://developers.google.com/web/tools/lighthouse
18. getServerSideProps // Next Documentation – Режим доступу до ресурсу: https://nextjs.org/docs/basic-features/data-fetching/get-server-side-props
19. getStaticProps // Next Documentation – Режим доступу до ресурсу: https://nextjs.org/docs/basic-features/data-fetching/get-static-props
20. Incremental Static Regeneration // Next Documentation – Режим доступу до ресурсу: https://nextjs.org/docs/basic-features/data-fetching/incremental-static-regeneration
21. getStaticPaths // Next Documentation – Режим доступу до ресурсу: https://nextjs.org/docs/basic-features/data-fetching/get-static-paths
22. Client-side data fetching // Next Documentation – Режим доступу до ресурсу: https://nextjs.org/docs/basic-features/data-fetching/client-side
23. Next/image // Next Documentation – Режим доступу до ресурсу: https://nextjs.org/docs/api-reference/next/image

24. Linux Game Programming // Mark "Nurgle." Collins – 2001
25. TypeScipt Documentation – Режим доступу до ресурсу: https://www.typescriptlang.org/
26. Node Documentation – Режим доступу до ресурсу: https://nodejs.org/en/
27. Prisma Documentation – Режим доступу до ресурсу: https://www.prisma.io/
28. Lucidchart Documentation – Режим доступу до ресурсу: https://www.lucidchart.com/pages/
29. Eslint Documentation – Режим доступу до ресурсу: https://eslint.org/
30. Прототипи веб-платформи https://www.figma.com/file/viCbL3RZzDibJeAv1BiGxL/Home-Beauty?node-id=0%3A1