Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Дипломна робота

освітній ступінь – магістр

на тему: «**Розробка спеціалізованого фреймворку серверного рендерингу React-додатків з оптимізацією завантаження сторінок** »

Виконав: студент 2-го року навчання,

Спеціальності

121 «Інженерія Програмного Забезпечення»

Войлов Богдан

Керівник Афонін А.О.

доцент

«05» червня 2025 р.

Київ – 2025

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра мультимедійних систем

Спеціальність 121 «Інженерія Програмного Забезпечення»

Освітня програма магістр

ЗАТВЕРДЖУЮ

Завідувач кафедри мультимедійних систем

Жежерун О. П.

«_____» _____ 2025 р.

ЗАВДАННЯ

ДЛЯ ДИПЛОМНОЇ РОБОТИ СТУДЕНТУ

Войлову Богдану

1. Тема роботи **«Розробка спеціалізованого фреймворку серверного рендерингу React-додатків з оптимізацією завантаження сторінок»**, керівник роботи Афонін Андрій Олександрович, доцент

2. Строк подання студентом роботи 5 червня 2025

3. План роботи

ЗМІСТ

ЗМІСТ

Анотація

Ця робота присвячена розробці фреймворку загального призначення для серверного рендерингу React-додатків, орієнтованого на високу швидкість завантаження сторінок.

У першій частині присвячена можливостям рендерингу React-додатків. Розглянуто які існують типи рендерингу, як вони працюють, та коли їх доцільно використовувати.

У другій частині розглянуто Core Web Vitals(CWV) метрики. Вплив різних типів рендеренгу на них. Також проаналізовано вплив розміру клієнтського JavaScript коду на CWV метрики на даних з датасету HTTPArchive CWV Technology Report.

У третій частині розглянуто інсуючі фреймворки для створення React-додатків, зокрема Next.js, Remix, Astro та Fresh. Проаналізовано їх функціональні можливості та статистику додатків що їх використовують з датасету HTTPArchive CWV Technology Report.

У четвертій частині описано архітектуру та важливі деталі імплементації прототипу фреймворку.

П’ята частина містить порівняльний аналіз метрик завантаження існуючих фреймворків та запропонованого фреймворку.

Вступ

У сучасній фронтенд розробці одним із головних викликів є забезпечення високої продуктивності та швидкого завантаження сторінок. Найпоширенішою бібліотекою для створення фронтенду є React.

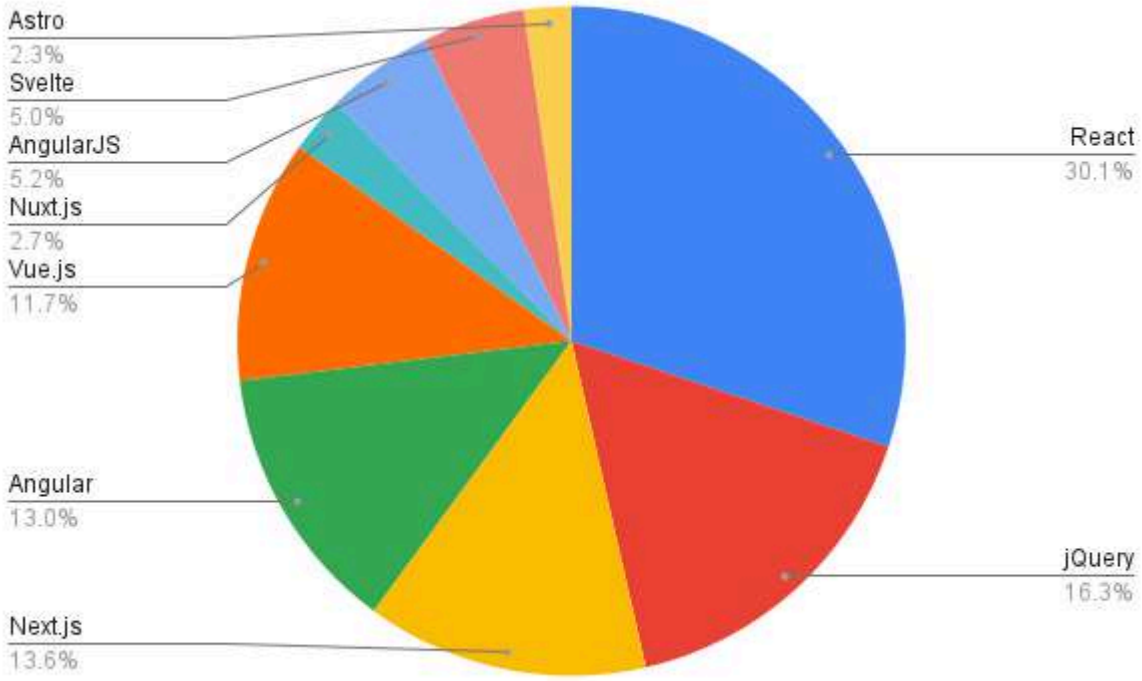

Рисунок 1. Використання інструментів у веб-розробці

Основна ідея бібліотеки React - є композиція компонентів. Компоненти, написані різними людьми, повинні добре працювати разом. Для авторів React важливо, щоб розробники могли додавати функціональність до компонента, не викликаючи ланцюгових змін у всій кодовій базі.

Абсолютна більшість веб додатків створених з використанням React мають низькі показники Core Web Vitals.

Фреймворки для серверного рендерингу, зокрема NextJS, Astro та Fresh, у деяких випадках обмежені як у функціональності, так і у швидкості завантаження сторінок.

Основною метою цієї роботи є створення прототипу фреймворку для розробки фронтенду веб додатків за допомогою бібліотеки React з оптимізацією завантаження сторінок.

Фреймворк має задовольняти наступним критеріям:

1. Надавати можливість використовувати сторонні React UI компоненти створені без використання цього фреймворку.
2. Мінімізувати час до відображення перших UI елементів.
3. Мінімізувати обсяг надлишкового JS коду що завантажує браузер на першій сторінці веб додатку.
4. Мінімізувати обсяг даних, що завантажує браузер при переході між сторінками веб додатку.

Завдання роботи включають аналіз можливостей рендерингу в React, аналіз Lighthouse метрик, аналіз існуючих фронтенд фреймворків, розробку архітектури нового фреймворку, враховуючи вище описані критерії, розробку прототипу фреймворку, та перевірку його ефективності.

Документація значної кількості функцій React потрібних для реалізації фреймворку публічно практично не доступна. Частково, важлива інформація не згадується взагалі. В певних розділах, зустрічається позиція команди авторів React, що користувачам не потрібно знати певні деталі. Приклад такої позиції наведено на рисунку 2.2. Тому частина цієї роботи присвячена дослідженню незадокументованих функцій React.

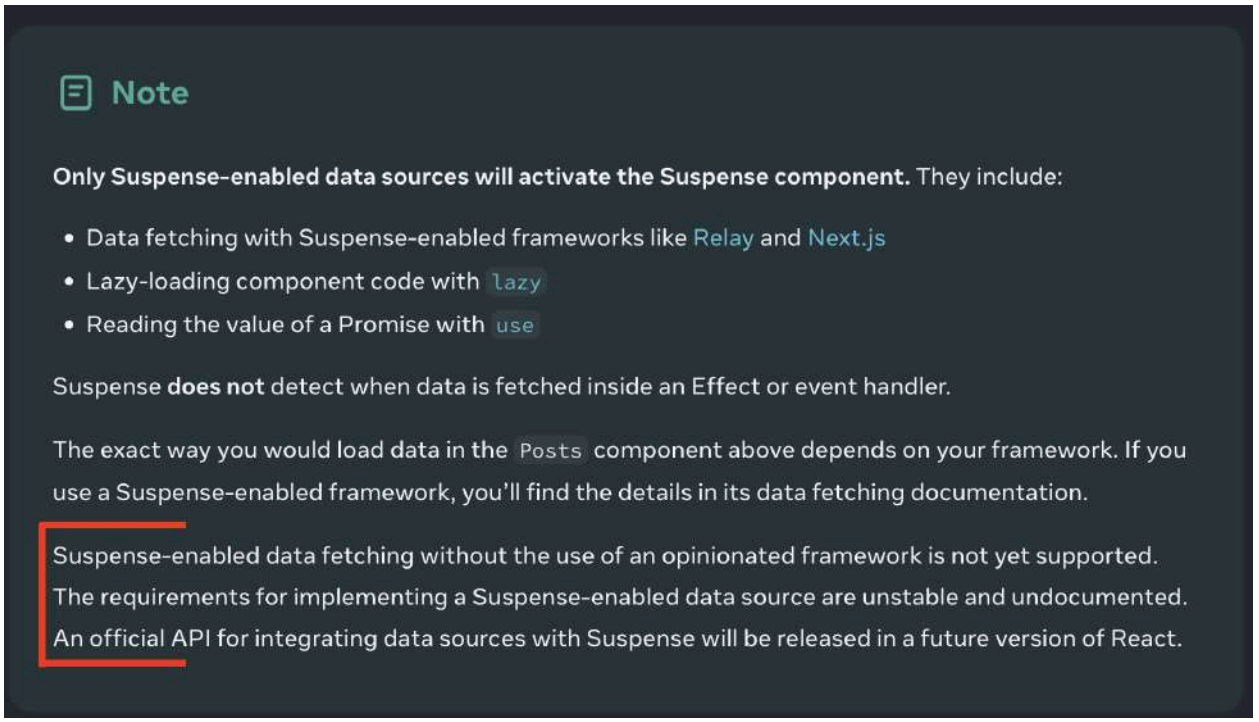

Рисунок 2. Приклад того як автори React не надають доступ до інформації важливої для імплементації власного фреймворку

Наукова новизна роботи полягає у розробці прототипу фреймворку, що має переваги у функціональних можливостях та швидкості завантаження сторінок перед існуючими фреймворками для створення React-додатків.

Результати цієї роботи можуть бути корисними фронтенд-розробникам, архітекторам і техлідам, які планують або вже реалізують React-додатки з високими вимогами до швидкодії та SEO. Також розробникам фронтенд бібліотек і фреймворків, що зможуть застосувати представлені в роботі підходи для вдосконалення власних інструментів, або як джерело натхнення для нових ідей щодо створення фреймворків забезпечуючих кращий користувацький досвід.

1. Огляд рендеренгу у React

React-додатки складаються з компонентів. Компонент — це частина UI, яка має власну логіку та зовнішній вигляд. Компонент може бути малим, як кнопка, або великим, як ціла сторінка. В сучасних версіях компоненти реалізуються JS функціями, що приймають параметри та повертають розмітку UI, як показано в лістингу 1.1.

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

Лістинг 1.1. Приклад React компонента

Компоненти можуть використовувати в розмітці інші компоненти, як показано в лістингу 3.2.

```
function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
    </div>
  );
}
```

Лістинг 1.2. Приклад використання інших React компонентів в розмітці

Таким чином React-додатки являють собою дерево компонентів. Далі розглянемо деталі реалізації рендерингу, що будуть дуже важливі для розділу

1.1. Рендеринг в браузері

Для того щоб відобразити UI в браузері, з розмітки що повертають функції-компоненти React створює Virtual DOM - дерево що за структурою

нагадує DOM браузера. Потім коли стан компонентів змінюється - функції-компоненти чий стан було змінено викликаються та повертають нову розмітку. React вносить цю розмітку у Virtual DOM, порівнює останню версію Virtual DOM з DOM браузера, та вносить відмінності у DOM браузера. Така кількість надлишкових дій потрібна через застарілий алгоритм обробки зміни стану, що робить багато надлишкових змін. Внесення цих змін напряму в DOM призвело б до значного погіршення швидкодії React-додатків.

1.2. Рендеринг на сервері

Модуль `react-dom/server` надає можливість з відобразити компонент у HTML рядок та у потік HTML рядків. Таким чином можуть створювати готові HTML сторінки з React компонентів на сервері. Це корисно коли є потреба зробити статичні сторінки з використовуючи готові бібліотеки компонентів, або коли є потреба використати дані або ресурси для рендерингу сторінки не відсилаючи їх клієнту.

Для того щоб додати клієнтську логіку до згенерованого HTML у React передбачена гідрація - процес прив'язування існуючого DOM до дерева компонентів на клієнті. Гідрація імплементована в функції `hydrateRoot(domNode, reactNode, options?)` з модуля `react-dom/client`.

Гідрація - часто фундаментально досить не ефективний процес[7]. Велика кількість сайтів має декілька компонентів з клієнтською логікою. Розглянемо як приклад додаток для ведення блогів, де користувачі можуть читати статті та переглядати коментарі та залишати коментарі. Схематично додаток зображено на рисунку 1.2.1. У результаті реалізації такого додатку на React у класичному стилі отримаємо дерево компонентів зображене на рисунку 1.2.2.

Бачимо що майже здебільшого додаток має статичні компоненти що не потребують гідрації.

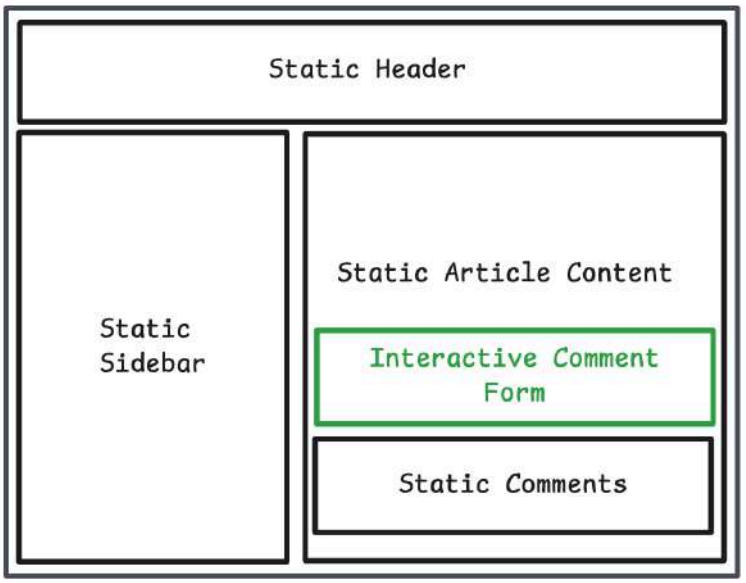

Рисунок 1.2.1. Схема веб-додатку блогу

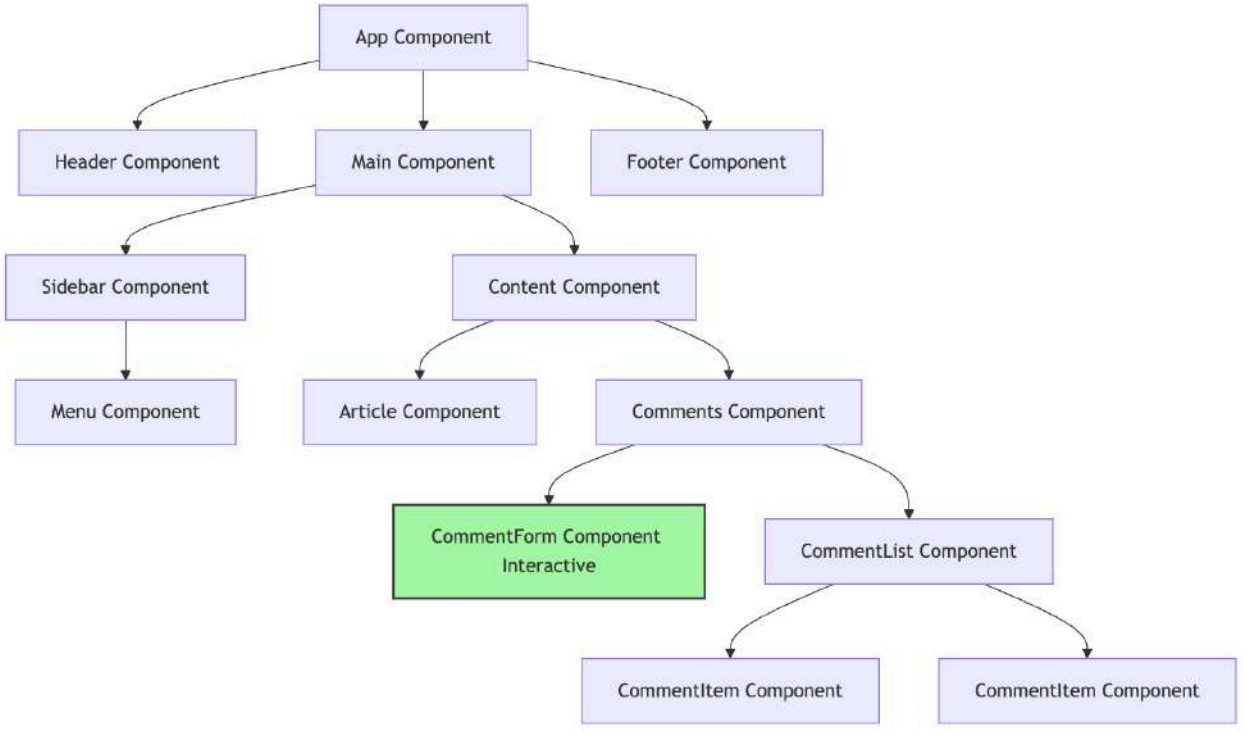

Рисунок 1.2.2. Діаграма компонентів веб-додатку блогу

Але в глубині дерева маємо форму для додавання коментаря. За технічним завданням форма повинна підтримувати редагування форматованого тексту. Приклад зображено на рисунку 1.2.3. Для такої форми найпростіше буде використати готовий React компонент з бібліотеки react-rte.

Рисунок 1.2.3. Форма редагування форматованого тексту

Якщо імплементувати цей додаток у найпопулярніших фреймворку серверного рендеренгу на React, Next.js, в контексті гідрації маємо два варіанти:

1. Для гідрації потрібно буде завантажити на клієнт код всіх статичних компонентів що використовуються на сторінці, та виконати його.
2. Використати спеціальну функцію React що підтримується в Next.js - серверні компоненти. Тоді потрібно буде завантажити лише код інтерактивного компоненту, але натомість для кожного серверного компоненту завантажиться повторно його розмітка в спеціальному форматі. В наступному підрозділі розглянемо це детальніше.

1.3. Серверні компоненти

Серверні компоненти - це компоненти чий код не завантажується на клієнт. Замість цього для гідрації їх гідрації потрібно завантажити їх розмітку в JSON форматі [1][14][15][19]. Наприклад для компонента з лістингу 1.3.1 на клієнт буде завантажено розмітку на лістингу 1.3.2.

```
function Greeting(props){
  return <span>Hello {props.name}</span>
}
```

Лістинг 1.3.1. Простий приклад коду серверного компонента

```
J0:["$","@1",null,{"children":["$","span",null,{"children":
"Hello from server land"}]}]
```

Лістинг 3.3.2. Дані що будуть завантажені на клієнт для серверного компонента з лістингу 1.3.1

Бачимо що це по суті повне дублювання HTML розмітки. Відповідно серверні компоненти відносно ефективні тільки у випадку коли вони мало повторюються, та мають багато JS коду, що генерує невелику розмітку. Для всіх інших випадків, особливо при відображені масивів, наприклад масиву коментарів як у прикладу на рисунку 1.2.2 - серверні компоненти тільки збільшують розмір даних потрібних для завантаження сайту.

Також серверні компоненти не ефективні в багатосторінкових додатках, де компоненти повторюються між різними сторінками. У Next.js коли користувач перейде на іншу сторінку, і там зустрінуться компоненти що були завантажені для відображення минулих сторінок - вони не будуть завантажуватися повторно. У випадку з серверними компонентами - їх розмітка буде завжди завантажуватись повністю як вперше.

Таким чином навіть офіційний приклад використання серверних компонентів від Next.js (https://github.com/vercel/next-react-server-components), що має демонструвати їх ефективність, по суті працює швидше якщо не використовувати серверні компоненти[4]. Саме тому автори Remix, другого за популярністю фреймворка для серверного рендеренгу, відмовились додавати підтримку серверних компонентів у Remix.

1.4. Потоковий серверний рендеринг

Часто під час серверного рендеренгу виникає ситуація коли HTML певних компонентів вже готовий, а інших ще генерується. Різні компоненти наприклад можуть залежати від різних даних, одні отримуються швидше, інші - довше, а деяким компонентам взагалі не не залежать від жодних даних і їх можна відобразити одразу[6].

Модуль `react/server-dom` надає можливість зменшити час до відображення перших UI елементів, шляхом потокової серверного генерації HTML. Замість того, щоб чекати поки повністю згенерується один великий HTML-файл, ми можемо розділити його на менші частини. Потоки Node дозволяють нам передавати дані по частинам об'єкт відповіді, що означає, що ми можемо відправляти частини HTML по мірі готовності. Оскільки браузер може почати відображати UI, поки ще отримує HTML, ми можемо створити дуже ефективний досвід першого завантаження.

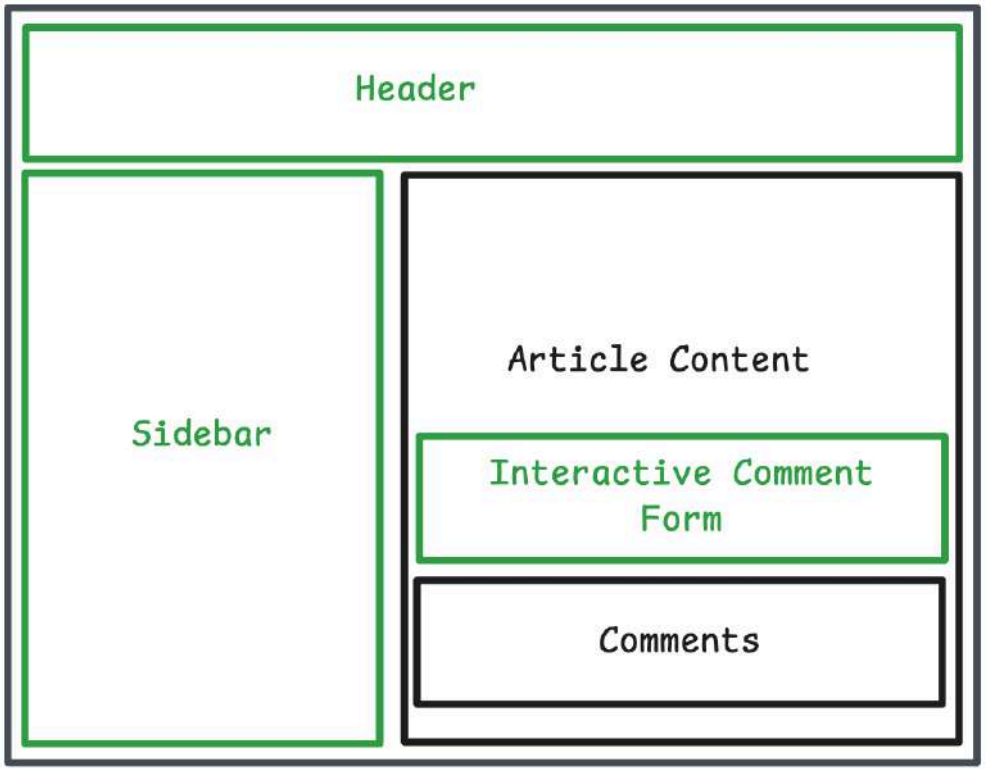

Рисунок 1.4.1. Частини блогу що не залежать від даних

На рисунку 1.4.1 зеленим кольором відображено компоненти що не потребують даних, і можуть бути відображені відразу.

Для того щоб відобразити сторінку зі заглушками на місці не готових частин, їх потрібно реалізувати асинхронними компонентами, та в місці використання обгорнути в компонент Suspense[9], як показано в лістингу 3.4.2.

```
function Article() {
  return (
    <div>
      <Suspense fallback={<span>Article loading...</span>}>
        <ArticleContent  />
      </Suspense>
      <CommentForm/>
      <Suspense fallback={<span>Comments
loading...</span>}>
        <Comments />
      </Suspense>
    </div>
  );
}
```

Лістинг 3.4.2. Компонент статті блогу з використанням Suspense

В самому компоненті, якщо це серверний комопнент, достатньо повернути розмітку обгорноту в Promise, в рамках Promise-у відповідно можна зробити всі необхідні асинхроні дії, на кшталт запиту на бекенд або в базу даних. Поки умовна база даних буде обробляти запити на отримання тексту статті та коментарів, користувач побачить сторінку з заглушками на місці цих частин.

Для отримання самого потоку HTML рядків, компонент сторінки потрібно передати у функцію `renderToPipeableStream(reactNode, options?)`. Функція поверне поток, в якому Suspense відобразиться в розмітку заглушки, та буде спеціальний коментар на елемент template з спеціальним id, як показано на лістингу 1.4.3.

```
<!--$?-->
<template id="B:0"></template>
<span>Article loading...</span>
<!--/$-->
```

Лістинг 1.4.3. Приклад HTML розмітки Suspense в серверній генерації

Після HTML коду Suspense-а далі в поток додається решта готового HTML коду. Коли генерація компонента загорнутого в Suspense завершиться, в поток додається згенерована розмітка всередині елемента div та елемент script, зі JS кодом що знайде елемент template зі спеціальним id та замінить його та заглушку на згенеровану розмітку, як показано на лістингу 3.4.4.

```
<div hidden id="S:0">
  <p>Article content goes here</p>
</div>
<script>
  $RC = function (b, c, e) {....}
  $RC("B:0", "S:0")
</script>
```

Лістинг 1.4.4. Приклад розмітки що додається в поток після завершення генерації компонента загорнутого в Suspense

Гідрація при потоковій генерації може відбуватися одночасно з тим як на клієнт досилаються згенеровані частини. Гідрація згенерованих частин по суті викликається в кінці функції `$RC`. Тому важливо до цього завантажити для серверного компоненту - JSON об’єкт його розмітки, а для звичайного компонента - його JS код.

2. Огляд CVW метрик та можливостей їх оптимізації

Core Web Vitals(CVW) — це підмножина показників Web Vitals, які стосуються всіх веб-сторінок, мають вимірюватися всіма власниками сайтів і відображаються у всіх інструментах Google[18]. Кожен із показників Core Web Vitals представляє окремий аспект користувацького досвіду, може бути виміряний у реальних умовах і відображає реальний досвід користувача щодо важливого результату, орієнтованого на користувача.

Core Web Vitals поділяє швидкість завантаження сторінки на наступні складові: First Contentful Paint (FCP), Largest Contentful Paint (LCP), Total Blocking Time (TBT), Cumulative Layout Shift (CLS), Interaction to Next Paint (INP). В рамках цієї роботи не розглядається CLS, оскільки ця метрика оптимізується через CSS стилі.

2.1 FCP та INP

First Contentful Paint вимірює час від моменту, коли користувач вперше перейшов на сторінку, до моменту, коли будь-яка частина вмісту сторінки відобразилася на екрані[17]. Для цієї метрики «контент» стосується тексту, зображень (включно з фоновими зображеннями), елементів <svg> або небілих елементів <canvas>.

На рисунку 2.1.1. показано як перше відображення контенту (FCP) відбувається у другому кадрі, оскільки саме тоді перші текстові та графічні елементи відображаються на екрані. Пришвидшити FCP в рамках React можна за допомогою SSR, як буде показано далі.

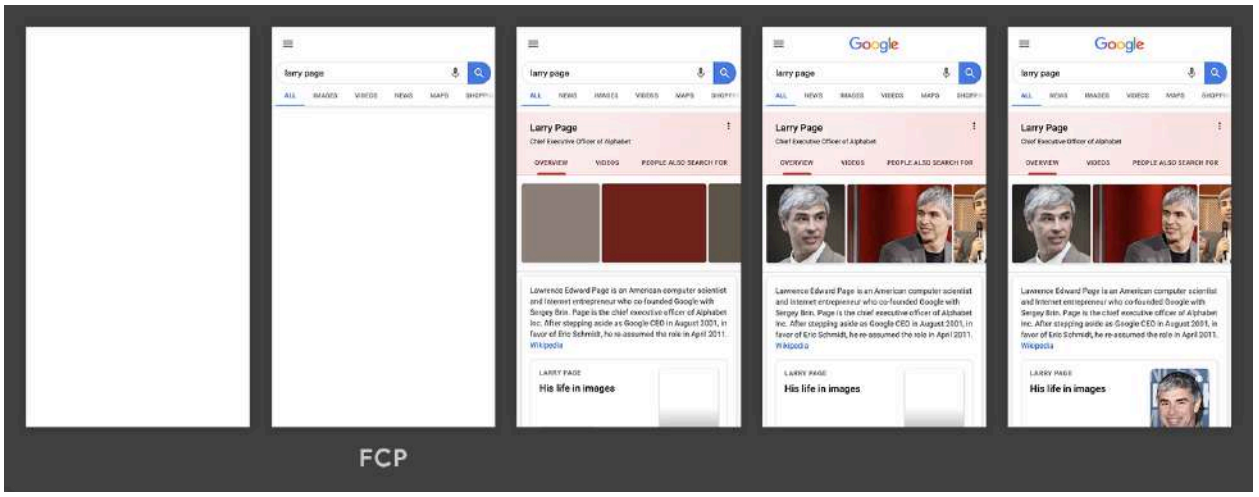

Рисунок 2.1.1. Демонстрація FCP на хронології завантаження веб сторінки

Взаємодія до наступного відмалювання (Interaction to Next Paint, INP) — це показник Core Web Vitals, який оцінює швидкість реакції сторінки, використовуючи дані з Event Timing API[16]. INP відстежує затримку всіх взаємодій користувача зі сторінкою та повідомляє одне значення, під яким були всі (або майже всі) взаємодії. Низький INP означає, що сторінка стабільно могла швидко реагувати на всі або переважну більшість дій користувача. Частиною INP є затримка введення (First Input Delay, FID) - показник який вимірює час до першої можливості взаємодії користувача зі сторінкою. Пришвидшити FID в рамках SSR React додатків можна зменшуючи розмір JS коду що потребує завантаження для роботи сайту.

2.2. Вплив клієнтського та серверного рендерингу на CVW

На рисунку 2.2.1. схематично зображено хронологію від початку HTTP запиту на HTML сторінку до FCP у звичайному клієнтському додатку на React. Бачимо що для того щоб щось відобразити спочатку потрібно завантажити весь JS код, потім побудувати Virtual DOM, і тільки потім в DOM потрапляють елементи інтерфейсу.

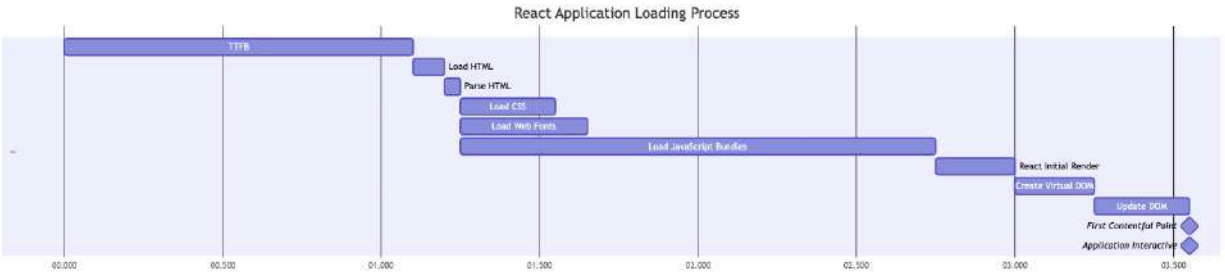

Рисунок 2.2.1. Діаграма хронології завантаження

Серверний рендеринг дозволяє відразу надіслати на клієнт готовий HTML. Таким чином клієнт відразу побачить якійсь UI, але не зможе з ним взаємодіяти поки не завантажить необхідні для гідрації дані та не проведе гідрацію. Таким чином значно зменшується FCP, але час до інтерактивності суттєво не змінюється, як показано на рисунку 4.2.2.

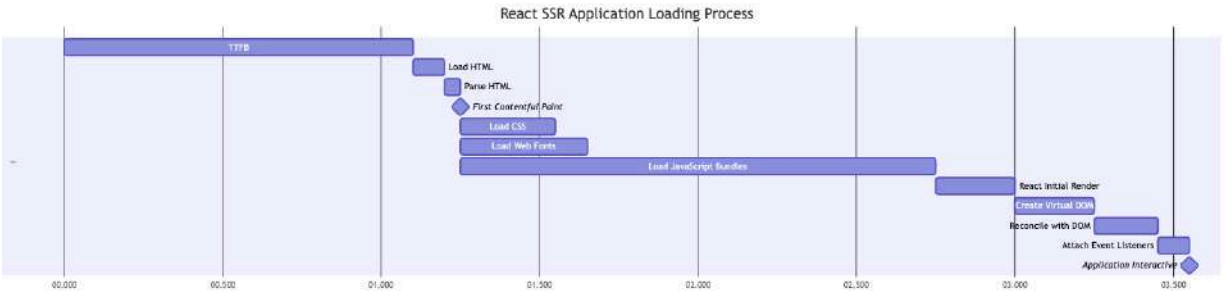

Рисунок 2.2.2. Діаграма хронології завантаження SSR додатку

2.3. Експериментальні дані

Розглянемо різницю в завантаженні на прикладі сторінки додатку “TODO Manager”, створеного з використанням популярної бібліотеки компонентів MUI, зображеної на рисунку 2.3.1. На хронологіях 2.3.2. та 2.3.3. зображено по кадру на кожні 460 мілісекунд з моменту відкриття сторінки в браузері, що відповідає значенню `window.performance.timing.navigationStart`. Детальніше методологія описана в розділі 5. Бачимо що FCP без використання SSR займає на 2.3с або на 170% більше часу. Але час з момента переходу на сторінку до інтерактивності UI - лише на 0.6с або на 20% довше. В таблиці 4.3.4. бачимо що без використання SSR, FCP наступає після того як обробники подій додані

в DOM, а з використанням SSR - раніше. Це обумовлено тим що без використання SSR - елементи в DOM додаються з обробниками, а лише потім відмальовуються браузером.

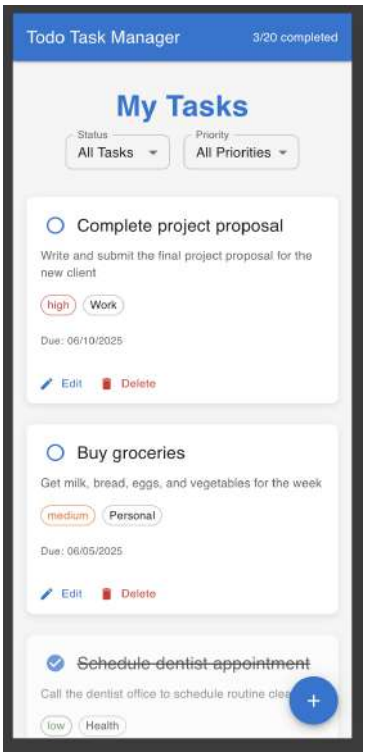

Рисунок 2.3.1. Сторінка додатку “TODO Manager”

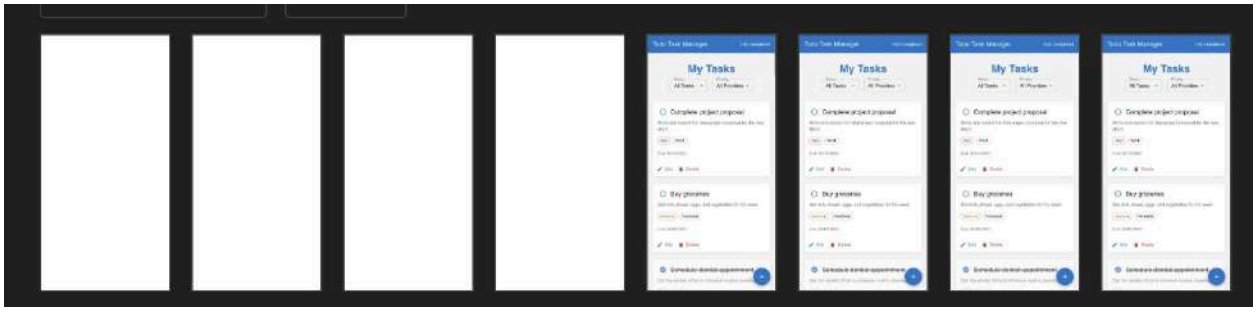

Рисунок 2.3.2. Хронологія завантаження веб сторінки з SSR

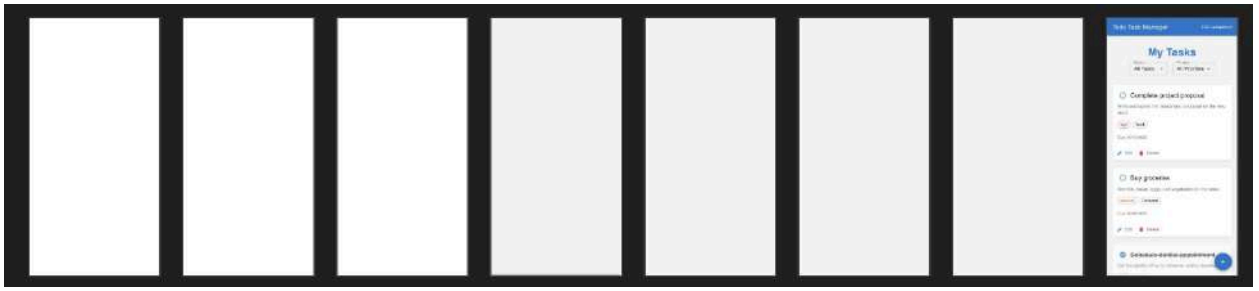

Рисунок 2.3.3. Хронологія завантаження веб сторінки без SSR

	SSR	Без SSR
Median FCP	1.3s	3.6s
Median Navigation Start to Listeners Attached	3s	2.1s
Median Navigation Start to Interactive	3s	3.6s

Таблиця 2.3.4. Порівняння метрик завантаження сторінок з використанням SRR та без

2.4. HTTPArchive CWV Technology Report

Звіт про CWV технологій є поєднанням двох джерел даних: CrUX та HTTP Archive, та містить дані по 18 мільйонам веб сайтів: які технології використовує веб сайт, розмір JS коду що він завантажує, його CWV метрики та багато іншого[8].

Датасет CrUX базується на даних користувачів Chrome, які дозволили надсилання статистики використання. Їхній досвід взаємодії з публічно доступними вебсайтами агрегується у 28-денні вікна, а результати публікуються у вигляді щомісячних дампів даних, доступних для запитів у BigQuery, а також через CrUX API.

HTTP Archive — це лабораторний інструмент, тобто він вимірює, як саме побудовані окремі вебсторінки, і він фактично базується на тих самих вебсайтах, що і корпус CrUX. HTTP Archive працює на основі WebPageTest, який інтегрується з іншими лабораторними інструментами, такими як Lighthouse та Wappalyzer, щоб отримувати детальні дані про сторінку.

На рисунку 2.4.1. бачимо що існує залежність між Lighthouse(LH) score, метрика від 0 до 1 що є зваженою сумою CWV метрик, та розміром

завантаженого сторінкою JS коду. Залежність підтверджується коефіцієнтом кореляції Пірсона, він становить -0.59. На рисунку 2.4.2. бачимо що медіана розміру JS коду сторінок з LH score <0.7 на 685% більша за медіану сторінок з LH score >0.7. Серед сторінок що використовують React ця медіана на 1008% більше за медіану сторінок з LH score <0.7. Медіана LH score сторінок що використовують React складає 0.45. Отже бачимо величезний простір для покращень.

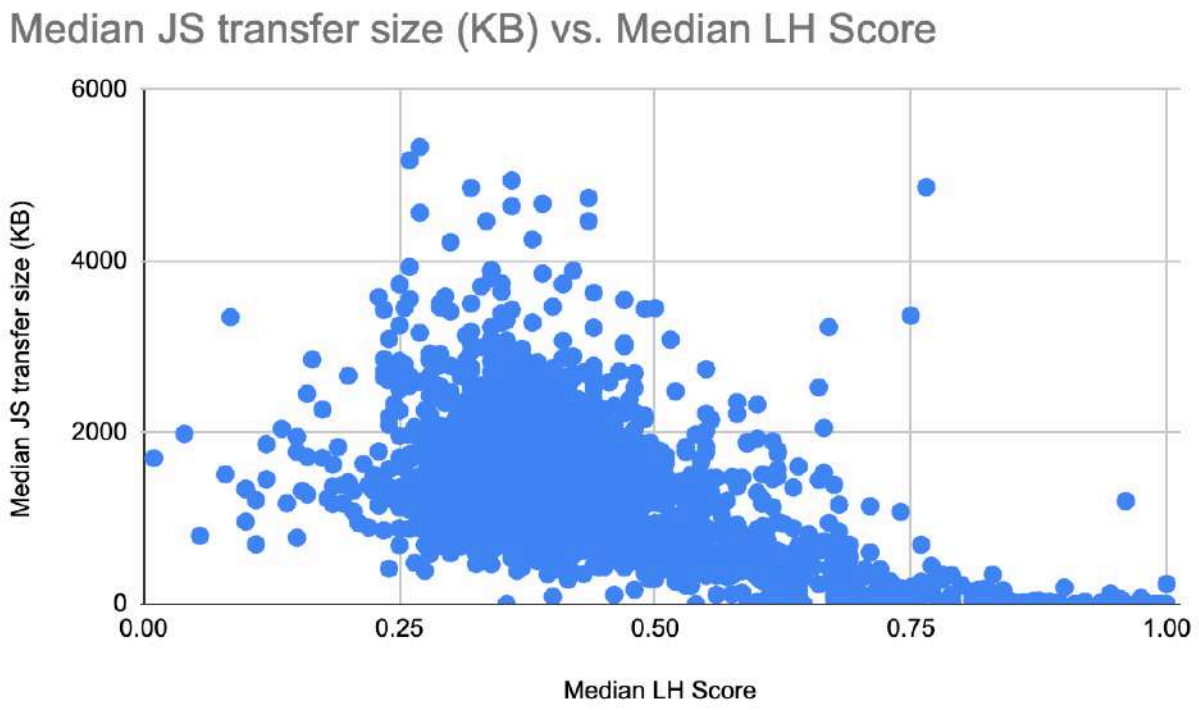

Рисунок 2.4.1. Залежність медіани LH Score до медіани загального розміру JS коду

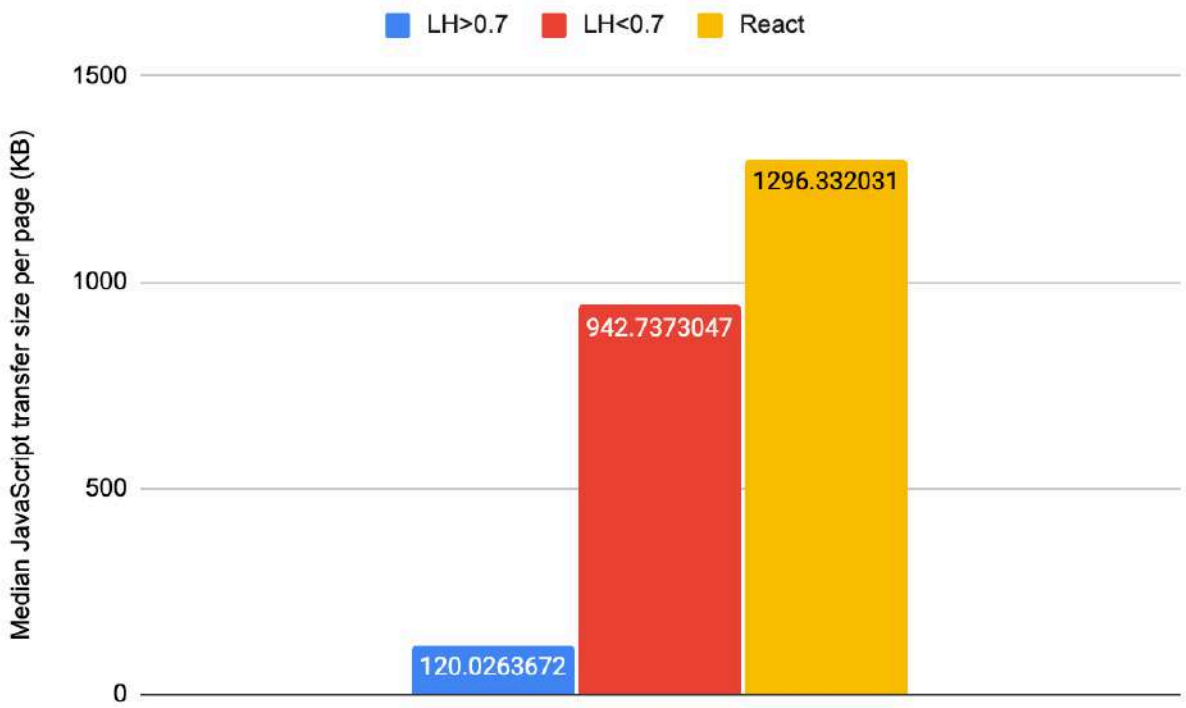

Рисунок 2.4.2. Порівняння медіан розміру JS коду сторінок

3. Огляд існуючих фреймворків

	Next.js	Remix	Astro
Кількість доменів	204 768	5 067	12 000
Медіана LH score	0.47	0.57	0.7
Медіана JS transfer size(KB)	933	677	311
Підтримка React	Нативна	Нативна	На рівні плагіна
Підтримка Suspense в SSR	Нативна у Server Components	Адаптована	Відсутня
Керування клієнтським JS	Server Components	Відсутня	Islands

Таблиця 3.1. Порівняння фреймворків серверного рендерингу

Основними фреймворками для створення React-додатків з серверним рендерингом є Next.js та Remix. Обидва гідрують всю сторінку після завантаження. Next.js дає можливість не завантажувати код певних компонентів на клієнт за допомогою Server Components, та використовувати Suspense в них, як і передбачено в React[10].

На фоні Next.js та Remix за LH score виділяється Astro. Astro це фреймворк для створення контентних веб додатків[2]. Його основною особливістю є острови(islands). Острів це компонент що має бути інтерактивним. Лише JS код островів буде завантажено на клієнт, що як ми бачимо з таблиці 3.1. суттєво зменшує розмір клієнтського JS сторінки, та суттєво підвищує LH score[13]. Astro має свої нативні компоненти, але завдяки можливості створювати плагіни, в якості островів дозволяє використовувати компоненти створені з використанням будь якої технології в

тому числі React. Однак Astro не дозволяє використовувати острови в React компонентах, лише в Astro компонентах.

Окремо ще слід зазначити Fresh. Це фреймворк що підтримує створення островів та нативно підтримує React, на відміну від Astro[5]. Однак Fresh не підтримує Suspense, та в цілому рідко використовується. В датасеті HTTPArchive CWV Technology Report зазначено лише 12 доменів що використовували Fresh у квітні 2025.

4. Опис розробленого прототипу фреймворку

Уточнимо вимоги описані у вступі враховуючи проведений аналіз в розділах 1, 2 та 3.

1. Підтримка SSR - для того щоб користувач швидше бачив інтерфейс.
2. Підтримка островів - щоб завантажувати лише JS код інтерактивних компонентів.
3. Підтримка Suspense - можливість асинхронно відображати окремі компоненти на сервері протягом відображення сторінки в HTML. Це зручно щоб отримувати дані з бази даних потрібні для відображення компонента, а поки дані в процесі отримання - продовжити відображати решту компонентів сторінки.
4. Можливість комунікації між островами - тобто інтерактивними частинами сторінки.
5. При переході між сторінками - завантаження та перемальовування в браузері не повністю всієї сторінки, а лише частини що відрізняються від попередньої.

Архітектурно запропонований фреймворк складається з трьох частин:

1. Роутер сторінок
2. Менеджер монтування островів
3. Пакувальник коду островів

Сценарій завантаження першої сторінки показано на рисунку 4.1., та сторінки при переході - на рисунку 4.2.

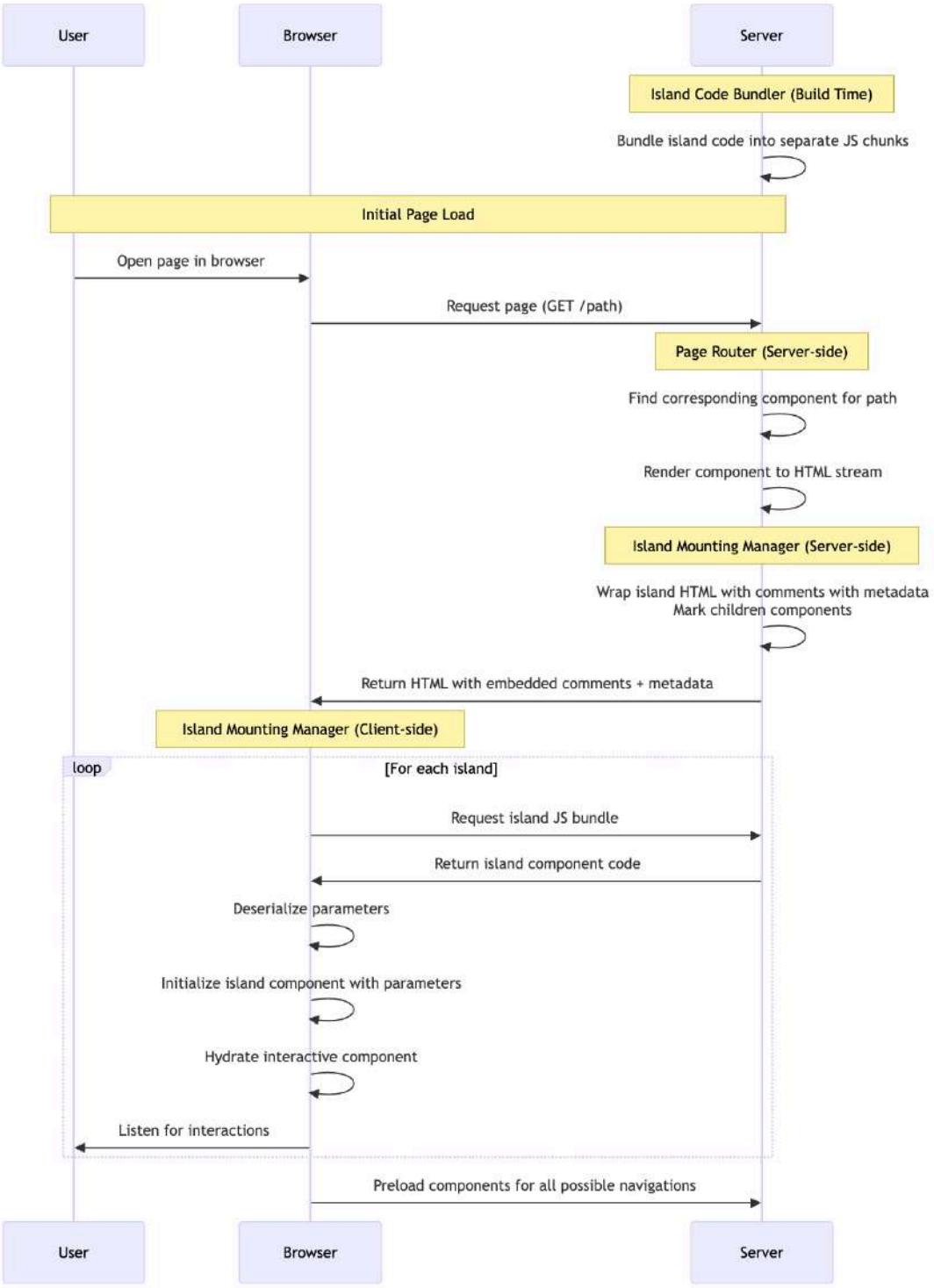

Рисунок 4.1. Діаграма завантаження першої сторінки додатку з використанням фреймворку

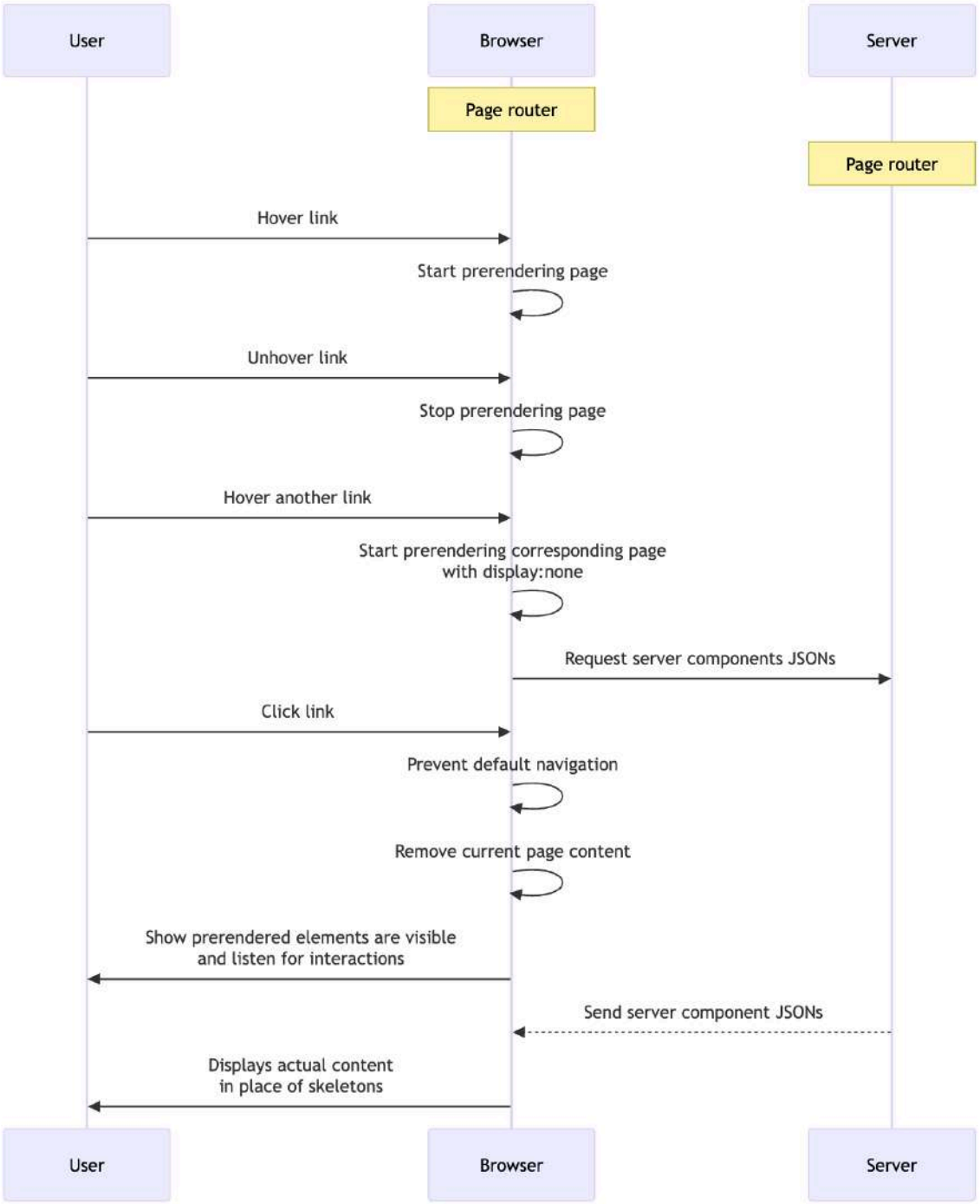

Рисунок 4.1. Діаграма завантаження сторінок при навігації в додатку з використанням фреймворку

4.1. Роутер сторінок

Оскільки при переході між сторінками статичні компоненти швидше за все будуть повторюватися, має сенс використовувати CSR для відображення сторінок при навігації.

Роутер сторінок складається з двох частин - серверної та клієнтської.

Серверна частина має 2 функції:

1. При запиті на сервер за певним шляхом має знаходити відповідний компонент та відображати його в потік HTML рядків які пересилати клієнту. Для співставлення шляху URL та компоненту сторінки використано клас FileSystemRouter з пакета bun. Таким чином розробник що використовує фреймворк, має в директорії pages створювати файли <url_path>/<page_name>.tsx.
2. Ендпоінт що відображає необхідний серверний компонент у JSON об’єкт необхідний для клієнтського відображення серверного компонента.

Клієнтська частина має також дві функції

1. При завантаженні сторінки - фоном завантажувати JS код компонентів сторінок на які має посилання поточна сторінка.
2. Для кожного внутрішнього посилання
    a. При наведені на посилання - починати рендерити попередньо завантажений компонент відповідної сторінки у елемент з CSS стилем display:none.
    b. При знятті наведення - припиняти попередній шаг.
    c. При натисканні на посилання, прибирати поточну сторінку та показувати попередньо зрендерену сторінку з шагу a.

4.2. Менеджер монтування островів

Менеджер монтування також складається з серверної та клієнтської частин.

Розробник що користується фреймворком має функцію island в яку огортає інтерактивні компоненти. Вона додає протягом серверного рендерингу в HTML коментарі перед початком та після кінця HTML відображення цього компонента. В коментарі вказує шлях за яким на сервері зберігається JS пакет цього компонента та серіалізовані параметри передані цьому компоненту. Також коментарями виділяє дітей острова, це також знадобиться на клієнті.

На клієнті розглянемо дві реалізації, кожна має свої плюси на мінуси.

Перша - менеджер створює React елементи повторюючи структуру HTML що повернув сервер, та коли натикається на коментар початку острова - дістає шлях до JS пакета та використовуючи React lazy import скачує код компоненту з сервера, десеріалізує параметри та ініціалізує компонент з ними. Таким чином всі острови знаходяться в одному React дереві, отже можемо використовувати React.Context для комунікації, а також потенційно додавати інші функції.

Друга - як в Astro, менеджер лише знаходить острови в DOM, так само завантажує їх код але ініціалізує кожен острів в окремому React дереві. В такій реалізації потрібно використовувати окремий механізм для комунікації між островами, але для сторінок що мають великі DOM дерева інтерактивність наступить значно швидше.

Оскільки перша імплементація надає додаткові функціональні можливості - в прототипі реалізовано її.

4.3. Пакувальник коду островів

Остання складова архітектури - пакувальник коду островів забезпечує збір коду островів в окремі JS файли, таким чином щоб клієнт міг завантажити лише необхідний код.

Також є можливість позначити інтерактивні або динамічні частини в файлі островах. Відповідно всі компоненти що знаходяться за рамками компоненту Dynamic, як на слайді компонент Typography - не будуть

імпортовані на клієнт, зменшуючи мережеве навантаження та пришвидшуючи настання інтерактивності. Dynamic помічає свої дочірні компоненти на етапі серверного рендерингу аналогічно функції island.

```
<Typography variant="body2">{props.note.text}</Typography>
<Dynamic>
  <CardActions>
    <Button size="small">Learn More</Button>
    <IconButton aria-label="delete" size="small"
      onClick={props.onDelete}>
      <DeleteIcon fontSize="inherit" />
    </IconButton>
  </CardActions>
</Dynamic>
```

Лістинг 4.3.1. Приклад використання компонента Dynamic

Для імплементації використано Vite - набір гнучких інструментів для створення серверів для розробки, що включає Rollup - гнучкий пакувальник JS коду. Rollup надає можливість писати власні плагіни, на основі цього імплементовано видалення залежностей за рамками компоненту Dynamic. Одна з функцій плагіну - transform(code:string, path:string). Плагіни виконуються в порядку що визначений в файлі vite.config.ts, тож для простоти імплементації варто додати наш плагін до плагіна @vitejs/plugin-react-swc, що транспілює JSX в JS. Тож в функції transform наш плагін для кожного файлу що закінчується на .island.tsx

1. Парсить код в ast за допомогою пакету @babel/parse
2. Проходить по ast дереву за допомогою пакету @babel/traverse
3. Видаляє компоненти за рамками <Dynamic> та їх імпорти з ast дерева
4. Замість <Dynamic> залишає <React.Fragment> - об’єднує елементи, для випадку якщо в один <Dynamic> огорнуто декілька сусідніх елементів.
5. Огортає JSX що повертає компонент острова у спеціальний компонент <DynamicInStaticRenderer myHtml={props.html}>, що з переданого html створює аналогічні React елементи поки не знайде коментар-мітку

компонента Dynamic. Далі замість створяння React елементів з помічених DOM елементів - використовує відповідний по порядку child React елемент, тобто саме той елемент що був огорнутий в Dynamic.

6. Зі зміненого ast генерує код за допомогою @babel/generate в файл <name>ClientGenerated.island.tsx, передає html елементи острова в параметри

Далі клієнтська частина менеджера монтування островів використовує саме цей файл для завантаження коду компонента острова.

4.4. Схема директорій проекту з використанням фреймворка

Виходячи з імплементації описаної в попередніх підрозділах, організація директорій на папок в проекті з використанням фреймворку буде виглядати як зображено на рисунку 4.4.1.

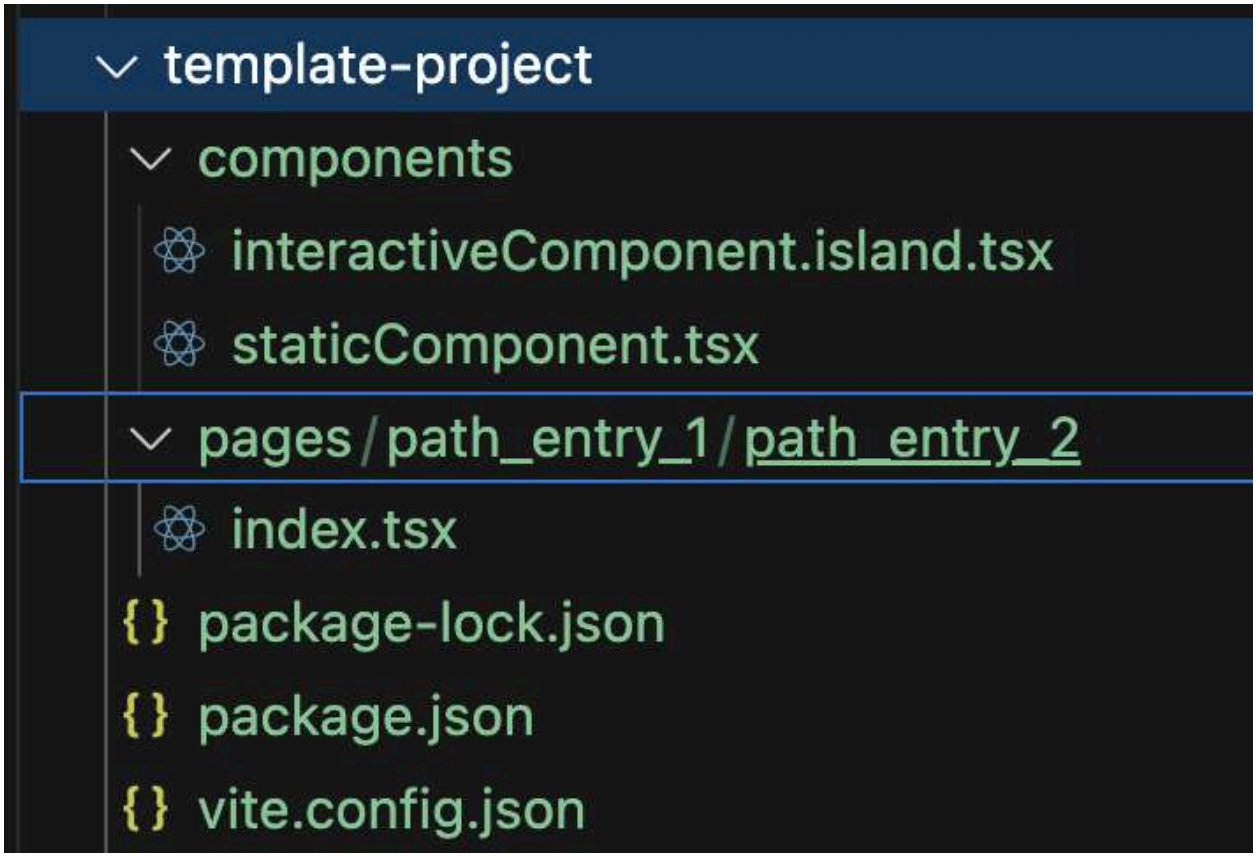

Рисунок 4.4.1. Структура директорій проєкту з використанням фреймворку

5. Порівняння метрик завантаження

Для вимірювання метрик використано Chromium Performance API[11]. FCP можна виміряти за допомогою PerformanceObserver, як показано в лістингу 5.1.

```
if ('PerformanceObserver' in window) {
  const observer = new PerformanceObserver((list) => {
    for (const entry of list.getEntries()) {
      if (entry.name === 'first-contentful-paint') {
        console.log('FCP:', entry.startTime);
        observer.disconnect();
      }
    }
  });

  observer.observe({ type: 'paint', buffered: true });
}
```

Лістинг 5.1. Приклад використання PerformanceObserver

Для вимірювання часу до додавання обробників подій використано інстанс класу Performance та хук React.useEffect що викликається після того як React вніс зміни що стосуються компонента в DOM, в нашому випадку - додав обробники подій. Метод now() класу Performance повертає час в мілісекундах від початку завантаження сторінки, тобто від моменту коли користувач відкриває сторінку в браузері[12].

```
useEffect(() => {
      const timeToReady =  window.performance.now()
      console.log(`Time to DOM updated:
${timeToReady.toFixed(2)}ms`)
    }
  }, [])
```

Лістинг 5.2. Приклад вимірювання часу до внесення React-ом перших змін в DOM

Конфігурація запуску тестів:

- Chromium версії 137.0.7151.56.
- Dev Tools Network throttling “Slow 4G”
- Dev Tools CPU throttling “4x”
- MacBook Pro
    - Chip Apple M1 Pro
    - Memory 16GB
    - macOS 15.5 (24F74)

Для запуску тестів використано Playwright, це бібліотека з відкритим вихідним кодом для автоматизації тестування в браузері та веб-скрапінгу, розроблена компанією Microsoft[3]. Для кожної з версій імплементації “TODO Manager” додатку описаного в підрозділі 2.3 зроблено 100 ітерацій запуску тесту.

	Next.js SSR	CSR	Astro	Фреймворк
Median FCP	1.3s	3.6s	1.3s	1.2s
Median Navigation Start to Listeners Attached	3.1s	2.1s	3s	2.8s
Median Navigation Start to Interactive	3.1s	3.6s	3s	2.8s
MUI components loaded on client	27	27	24	16
MUI component total size	78KB	78KB	75KB	69KB

Таблиця 5.1. Порівняння імплементацій додатку “TODO Manager”

Додаток “TODO Manager” було обрано через те що його імплементація досить погано оптимізується з використанням островів Astro. Розмітка сторінки містить багато статичних елементів але вони так змішані з інтерактивним що важко виділити окремі острови і по суті виходить один великий острів, та вдається запобігти завантаженню JS кода лише 3х MUI компонентів - AppBar, Toolbar та Container. Але використовуючи компонент Dynamic з запропонованого фреймворку, показану в лістингу 4.3.1 вдається запобігти завантаженню 11 MUI компонентів. Бачимо що не зважаючи на те що кількість завантажених MUI компонентів зменшилась на 40%, загальний розмір коду MUI компонентів завантажений на клієнт зменшився лише на 11%. Такий результат обумовлено тим що різні компоненти мають кратно різний розмір коду.

Слід зазначити що в імплементацій інших веб сторінках - можливості оптимізації можуть сильно відрізнятися в залежності співвідношення розміру JS коду статичних та динамічних компонентів. Загалом час завантаження буде як мінімум на рівні Astro або меньше для більшості випадків. Час завантаження може бути більшим ніж в Astro лише у випадку коли сторінка має значну кількість статичних DOM елементів, наприклад якщо зі сторінки перегляду коду пул реквесту в GitHub прибрати можливість взаємодіяти з кодом, імплементація такої сторінки на Astro буде значно швидшою. Десять сторінок змін в пул реквесті можуть відображатись у десятки тисяч DOM елементів. І якщо за ТЗ ці сторінки можуть бути не інтерактивні - клієнтська частина менеджера монтування островів марно витратить значний час на створення інстансів React.Element.

6. Висновки

В рамках роботи було досліджено можливості рендерингу React-додатків. Проаналізовано їх вплив на CWV метрики. Також було проаналізовано вплив розміру JS коду що завантажує сторінка на CWV метрики, використовуючи дані датасету HTTPArchive CWV Technology Report. Було проведено огляд існуючих фреймворків серверного рендерингу. Порівняно їх функціональні можливості та статистика метрик завантаження додатків побудованих з використанням цих фреймворків. В ході аналізу обгрунтовано переваги серверного рендерингу та островів.

На основі проведеного аналізу визначено які функції має підтримувати фреймворк для того щоб давати можливість створювати додатки зі швидким завантаженням сторінок та мати переваги над кожним з розглянутих існуючих фреймворків.

Розроблено архітектуру що забезпечує підтримку визначених функцій. Розглянуто дві можливі реалізації менеджера монтування островів.

Для реалізації прототипу фреймворку було використано мову програмування TypeScript та Vite - набір гнучких інструментів для створення серверів для розробки, що включає Rollup - гнучкий пакувальник JS коду на основі якого імплементовано пакування коду островів. Імплементовано розроблену архітектуру на рівні Proof of Concept.

Проведено вимірювання метрик завантаження сторінки на основі розроблених реалізацій MVP додатку “TODO Manager” з використанням клієнтського рендерингу, серверного рендерингу Next.js, серверного рендерингу та островів Astro та розробленого фреймворку. На основі вимірювання підтверджено переваги запропонованої архітектури фреймворку.

Список використаних джерел

1. Abrodi T. Why are React Server Components actually beneficial?. tigerabrodi. URL: https://tigerabrodi.blog/why-is-react-server-components-actually-beneficial-full-history (date of access: 24.04.2025).

2. Astro. Astro. URL: https://astro.build/ (date of access: 03.04.2025).

3. Fast and reliable end-to-end testing for modern web apps | Playwright. Fast and reliable end-to-end testing for modern web apps | Playwright. URL: https://playwright.dev/ (date of access: 05.03.2025).

4. Florence R. React Server Components and Remix. Remix - Build Better Websites. URL: https://remix.run/blog/react-server-components (date of access: 21.03.2025).

5. Fresh - The simple, approachable, productive web framework. Fresh - The simple, approachable, productive web framework. URL: https://fresh.deno.dev/ (date of access: 16.04.2025).

6. Gonchar D. How and Components Streaming works in Next.js. hackernoon. URL: https://hackernoon.com/how-lesssuspense-greater-and-components-streaming-works-in-nextjs (date of access: 01.05.2025).

7. hydrateRoot – React. React. URL: https://18.react.dev/reference/react-dom/client/hydrateRoot (date of access: 10.04.2025).

8. New dashboard: the Core Web Vitals technology report. HTTP Archive. URL: https://discuss.httparchive.org/t/new-dashboard-the-core-web-vitals-technology-report/2178 (date of access: 07.05.2025).

9. New Suspense SSR Architecture in React 18. GitHub. URL: https://github.com/reactwg/react-18/discussions/37 (date of access: 13.05.2025).

10. Next.js by Vercel - The React Framework. Next.js by Vercel - The React Framework. URL: https://nextjs.org/ (date of access: 07.04.2025).

11. Performance APIs - Web APIs | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance_API (date of access: 09.05.2025).

12. Performance: now() method - Web APIs | MDN. MDN Web Docs. URL: https://developer.mozilla.org/en-US/docs/Web/API/Performance/now (date of access: 05.03.2025).

13. Schott F. 2023 Web Framework Performance Report | Astro. Astro. URL: https://astro.build/blog/2023-web-framework-performance-report/ (date of access: 14.05.2025).

14. Server Components – React. React. URL: https://react.dev/reference/rsc/server-components (date of access: 02.04.2025).

15. Understanding React Server Components | Tony Alicea. Tony Alicea. URL: https://tonyalicea.dev/blog/understanding-react-server-components/ (date of access: 17.04.2025).

16. Wagner J. Interaction to Next Paint (INP) | Articles | web.dev. web.dev. URL: https://web.dev/articles/inp (date of access: 05.06.2025).

17. Walton P. First Contentful Paint (FCP) | Articles | web.dev. web.dev. URL: https://web.dev/articles/fcp (date of access: 09.05.2025).

18. Walton P. Web Vitals | Articles | web.dev. web.dev. URL: https://web.dev/articles/vitals (date of access: 08.05.2025).

19. Wu C. How React server components work: an in-depth guide. Plasmic Blog. URL: https://www.plasmic.app/blog/how-react-server-components-work#consuming-the-rsc-format (date of access: 09.04.2025).