Вінницький національний технічний університет
(повне найменування вищого навчального закладу)
факультет інформаційних технологій та комп'ютерної інженерії
(повне найменування інституту, назва факультету (відділення))
Кафедра програмного забезпечення
(повна назва кафедри (предметної, циклової комісії))

МАГІСТЕРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«Удосконалення методів і засобів острівної архітектури побудови веб додатків»

Виконав: студент 2-го курсу, групи 1ПІ-23м
спеціальності 121 – Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)
Осипенко К.С
(прізвище та ініціали)
Керівник: к.т.н., доц. каф. ПЗ:
Черноволик Г.О.
(прізвище та ініціали)
«06» грудня 2024 р.
Опонент:
к.т.н., доц. каф. ОТ Колесник І.С.
(прізвище та ініціали)
«06» грудня 2024 р.

Допущено до захисту
Завідувач кафедри ПЗ
д.т.н, проф. Романюк О.Н.
(прізвище та ініціали)
«06» грудня 2024 р.

ВНТУ – 2024

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра програмного забезпечення
Рівень вищої освіти ІІ-й (магістерський)
Галузь знань 12 – Інформаційні технології
Спеціальність 121 – Інженерія програмного забезпечення
Освітньо-професійна програма – Інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О. Н.
«18» вересня 2024 р.

З А В Д А Н Н Я
НА МАГІСТЕРСЬКУ КВАЛІФІКАЦІЙУ РОБОТУ СТУДЕНТУ

Осипенку Костянтину Сергійовичу

1. Тема роботи – Удосконалення методів і засобів острівної архітектури побудови веб додатків.

Керівник роботи: Черноволик Галина Олександрівна, к.т.н., доц. кафедри ПЗ, затверджені наказом вищого навчального закладу від «17» вересня 2024 р. № 310.

2. Строк подання студентом роботи
3 грудня 2024 року

3. Вихідні дані до роботи: середовище розробки Webstorm, мова розробки Javascript, операційна система – Windows 10.

4. Зміст текстової частини: вступ; аналіз стану розвитку архітектури вебзастосунків, розробка програмного модулю бібліотеки для створення вебзастосунків на основі архітекрути островів, тестування програми; економічна частина; висновки, список використаних джерел, додатки.

5. Перелік графічного матеріалу: модель роботи бібліотеки, блок-схеми алгоритмів роботи бібліотеки; тестування сайту на основі бібліотеки.

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-4	Черноволик Г.О., к.т.н., доцент кафедри ПЗ	19.09.2024	02.12.2024
5	Причепа І.В, к.е.н., доцент кафедри ЕПВМ	22.11.2024	30.11.2024

7. Дата видачі завдання 19 вересня 2024 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області архітектур побудови вебзастосунків	20.09.2024 30.09.2024	
2	Розробка методу та моделей системи	01.10.2024 17.10.2024	
3	Розробка програмного модулю бібліотеки для створення вебзастосунків на основі архітектури островів	18.10.2024 04.11.2024	
4	Тестування програмного додатку	05.11.2024 21.11.2024	
5	Економічна частина	22.11.2024 30.11.2024	

Студент _____ Осипенко К.С.
(підпис) (прізвище та ініціали)

Керівник магістерської кваліфікаційної роботи _____ Черноволик Г.О.
(підпис) (прізвище та ініціали)

Анотація

УДК 004.912.032.26

Осипенко К.С. Методи та програмні засоби розробки острівної архітектури веб додатків. Магістерська кваліфікаційна робота зі спеціальності 121 – Інженерія програмного забезпечення, освітня програма – Інженерія програмного забезпечення. Вінниця: ВНТУ, 2024. 165 с. На укр. мові. Бібліогр.: назв: 34; рисунків 47; таблиць: 13.

У магістерській кваліфікаційній роботі подано результати дослідження технологій розробки вебдодатків з використанням острівної архітектури для підвищення ефективності та продуктивності роботи вебзастосунків.

Магістерська кваліфікаційна робота містить аналіз існуючих вебфреймворків і бібліотек, які використовують острівну архітектуру, особливостей їхньої реалізації та функціональних можливостей, а також методів оптимізації взаємодії з користувачем. Оцінено методи дегідрації та гідрації даних для компонентів острівної архітектури та обґрунтовано вибір технологій кешування для покращення продуктивності.

Розроблено бібліотеку для реалізації острівної архітектури з використанням Web Worker, що дозволяє здійснювати дегідрацію та гідрацію даних у вебдодатку, а також управління кешуванням. Реалізовано основні функціональні модулі для забезпечення роботи островів, включаючи підтримку автономного функціонування у режимі офлайн. Вебдодаток розроблено з використанням мови програмування JavaScript та бібліотеки Eleventy для генерації статичних сайтів.

Ключові слова: острівна архітектура, вебдодаток, дегідрація, гідрація, кешування, JavaScript, Web Worker.

Abstract

Osypenko K.S. Methods and Software Tools for Developing Island Architecture of Web Applications. Master's qualification thesis in specialty 121 – Software Engineering, educational program – Software Engineering. Vinnytsia: VNTU, 2024. 165p. In Ukrainian. Bibliography: 34 titles; figures: 47; tables: 13.

The master's qualification thesis presents the results of research on web application development technologies using island architecture to improve the efficiency and performance of web applications. The thesis includes an analysis of existing web frameworks and libraries that utilize island architecture, their implementation features, functional capabilities, and user interaction optimization methods. The methods of data dehydration and rehydration for island architecture components are evaluated, and the choice of caching technologies for improving performance is justified.

A module for implementing island architecture using Web Workers has been developed, allowing data dehydration and rehydration in the web application, as well as cache management. The main functional modules have been implemented to ensure the functioning of islands, including support for offline mode. The web application is developed using JavaScript and the Eleventy library for static site generation.

Keywords: island architecture, web application, dehydration, rehydration, caching, JavaScript, Web Worker.

ЗМІСТ

ВСТУП

Обґрунтування вибору теми дослідження. У сучасних умовах розвитку цифрових технологій все більше уваги приділяється оптимізації та підвищенню продуктивності вебдодатків, що зумовлюється високими вимогами користувачів до швидкодії, інтерактивності та гнучкості інформаційних систем. Сучасні вебдодатки є основою для багатьох бізнес-процесів, оскільки вони забезпечують зручність взаємодії з клієнтами, доступ до послуг та інформації, а також можливість автоматизації багатьох процесів. У зв'язку з цим постає питання пошуку нових архітектурних рішень, які можуть забезпечити оптимальну продуктивність, особливо у випадках масштабування системи під високі навантаження.

Архітектура «островів» (Island Architecture) є новітнім підходом до розробки вебдодатків, який дозволяє поєднати переваги статичних та динамічних вебтехнологій, що забезпечує високу продуктивність, простоту і модульність[1]. Вона передбачає поділ вебсторінки на незалежні частини — так звані "острови", які можуть гідруватися у різний час або навіть залишатися статичними. Це дозволяє мінімізувати час завантаження сторінки, підвищуючи її швидкодію та полегшуючи інтерактивність. Крім того, використання такого підходу дозволяє розробникам створювати вебдодатки, які легко підтримувати та масштабувати.

Актуальність дослідження зумовлена тим, що сучасні користувачі очікують швидке завантаження контенту та стабільну роботу додатків навіть при великій кількості відвідувачів або при слабкому інтернет-з'єднанні. Це ставить перед розробниками серйозні виклики, оскільки необхідно знайти оптимальні методи для підтримки продуктивності додатка під час росту його складності та навантаження. Острівна архітектура дозволяє вирішити ці проблеми шляхом оптимізації завантаження окремих частин сторінки, забезпечуючи мінімальне споживання ресурсів та ефективне управління компонентами.

Значний інтерес до цього підходу пояснюється також необхідністю інтеграції з популярними фреймворками, такими як React, Vue, Preact, що вже є усталеними інструментами для створення динамічних вебінтерфейсів. У межах дослідження запропоновано фреймворк-агностік підхід, що дає можливість працювати з різними технологіями через адаптери, які забезпечують гнучкість і сумісність. Це дозволяє розробникам вільно вибирати найбільш підходящі інструменти для конкретних завдань, не обмежуючись технологічними рамками конкретного фреймворку.

Тому актуальною є розробка власної бібліотеки, що забезпечить розробника необхідним функціоналом для створення додатку на основі острівної архітектури.

Зв'язок роботи з науковими програмами, планами, темами. Робота виконувалася згідно плану виконання наукових досліджень на кафедрі програмного забезпечення.

Мета та завдання дослідження. Метою магістерської кваліфікаційної роботи є удосконалення методів і засобів острівної архітектури побудови вебдодатків для підвищення їхньої продуктивності, гнучкості та зручності використання шляхом розробки власної бібліотеки, що забезпечить розробника необхідним функціоналом для реалізації.

Основними задачами роботи є:

- аналіз існуючих підходів до побудови вебдодатків з використанням острівної архітектури;
- розробка методів гідрації і динамічного завантаження компонентів;
- створення адаптерів для використання популярних фреймворків у межах архітектури;
- реалізація механізму розумного кешування;
- тестування розробленої системи.

Об’єктом дослідження є процес розробки вебдодатків з використанням острівної архітектури.

Предметом дослідження є методи і засоби підвищення продуктивності та оптимізації роботи вебдодатків на основі острівної архітектури.

Методи дослідження. У процесі досліджень використовувались методи дослідження:

- методи створення бібліотек для створення додатків з острівною архітектурою;
- методи об’єктно-орієнтованого програмування для створення компонентів системи;
- методи кешування для оптимізації роботи додатків;
- методи тестування для перевірки працездатності розробленого рішення.

Наукова новизна отриманих результатів:

1. Подальшого розвитку отримав метод оптимізації завантаження вебсторінки, який, на відміну від існуючих, за рахунок розумного кешування, допомагає оптимізувати подальші завантаження статичних частин додатку.

2. Подальшого розвитку отримав метод покращення продуктивності роботи вебзастосунку, який, на відміну від існуючих, виносить гідрацію і дегідрацію в окремий поток, що зменшує навантаження на основний поток.

Практичне значення одержаних результатів. Розроблена бібліотека може використовуватися розробниками для створення власних додатків.

Особистий внесок здобувача. Усі наукові результати, викладені у магістерській кваліфікаційній роботі, отримані автором особисто. У науковій роботі[2], автору належать такі результати: аналіз архітектур для побудови вебзастосунків, постановка задач дослідження, визначення методів їх розв’язання і програмна реалізація системи.

Апробація матеріалів магістерської кваліфікаційної роботи. Результати магістерської кваліфікаційної роботи обговорювалися на міжнародній науково-практичній Інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ» (Суми/Вінниця, 2024).

Публікації. Результати дослідження опубліковані в тезах доповіді міжнародної науково-практичної Інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ, 2024»[2].

Структура та обсяг роботи. Магістерська кваліфікаційна робота складається зі вступу, п’яти розділів, висновків, списку використаних джерел, що містить 34 найменування, 4 додатки. Робота містить 47 ілюстрацій, 13 таблиць.

1 АНАЛІЗ СТАНУ РОЗВИТКУ АРХІТЕКТУРИ ПОБУДОВИ ВЕБЗАСТОСУНКІВ

1.1 Аналіз предметної області архітектур побудови вебзастосунків

Історія архітектур вебзастосунків — це історія еволюції технологій, що почалася з простих, статичних сторінок і пройшла довгий шлях до складних динамічних вебдодатків. Протягом десятиліть ці архітектури змінювалися, відповідаючи на виклики продуктивності, зручності використання і швидкості.

Першим етапом розвитку були повністю статичні сторінки[3]. У 1990-х роках вебзастосунки складалися переважно з статичних HTML-сторінок. Такі сторінки створювалися вручну й відображали однаковий контент для всіх користувачів. Це був початковий етап розвитку Інтернету, коли більшість сайтів функціонували як інформаційні вітрини. Ранні вебдодатки були дуже простими, а технології, як-от HTML, CSS і трохи пізніше JavaScript, почали набувати популярності.

Наступним етапом був Server-Side Rendering (SSR)[4]. На рубежі 2000-х років почала зростати потреба в більш інтерактивних додатках, які могли адаптувати контент під кожного користувача. Це стало можливим завдяки SSR (Server-Side Rendering) — підходу, де вебсторінки динамічно генеруються на сервері. Технології, як-от PHP, ASP.NET, і JSP, стали популярними, оскільки дозволяли створювати інтерактивні вебзастосунки, де користувачі могли виконувати дії, а сервери — генерувати відповідні сторінки.

Наступним етапом є поява Single page application [5]. З поширенням швидкісного Інтернету і збільшенням потужностей браузерів у середині 2000-х років з'явилися перші SPA (Single-Page Applications). Це стало революцією в веброзробці, оскільки додатки почали працювати як десктопні програми, але в браузері. Замість того, щоб перезавантажувати сторінку після кожної дії

користувача, SPA дозволяли оновлювати лише необхідні частини інтерфейсу. Це стало можливим завдяки використанню JavaScript та технологій на кшталт AJAX.

У цей період з'явилися популярні JavaScript-фреймворки, такі як AngularJS, а пізніше React і Vue.js, які дозволили створювати складні інтерфейси з мінімальними затримками для користувачів.

З розвитком обчислювальної техніки та глобалізацію використання сайтів наступним етапом є перехід на мікросервіси. У 2010-х роках із зростанням складності вебзастосунків і збільшенням обсягу користувачів виникла потреба в більш масштабованих і гнучких архітектурах. З'явилася мікросервісна архітектура, що дозволяла розбивати великий вебзастосунок на невеликі, незалежні сервіси. Це дозволило розробникам працювати над окремими частинами додатка, що полегшувало масштабування й підтримку.

Паралельно з мікросервісами популярною стала контейнеризація за допомогою технологій на кшталт Docker[6]. Вона дозволила ізолювати застосунки в контейнери, що полегшувало їх розгортання і забезпечення стабільної роботи в різних середовищах.

І останнім етапом є популяризація Jamstack та Island architecture [7]. У 2020-х роках з'явилися нові тенденції в архітектурі вебдодатків, орієнтовані на швидкість завантаження і продуктивність. Island architecture стала однією з таких концепцій, яка поєднує підходи SSR і SPA: сторінки рендеряться на сервері, але деякі частини (острови) можуть бути інтерактивними і оновлюватися без перезавантаження.

Ще однією популярною архітектурою став Jamstack — підхід, де клієнтська частина додатка генерується заздалегідь і відображається як статичні файли, а всі динамічні запити виконуються через API.

У сучасному світі розробники вебдодатків стикаються з низкою проблем, пов'язаних із підвищенням складності архітектур. Однією з головних проблем є гідрація — процес, під час якого клієнтська частина SPA застосунку має «оживити» сторінку після початкового рендерингу на сервері (SSR). Це може призводити до затримок у взаємодії з користувачем, оскільки браузеру потрібно виконати значний обсяг JavaScript-коду, перш ніж сторінка стане повністю інтерактивною.

Ще однією проблемою є надмірне споживання ресурсів. Багато сучасних фреймворків і бібліотек вимагають великих обсягів даних і ресурсів для рендерингу складних інтерфейсів, що збільшує час завантаження та знижує продуктивність, особливо на мобільних пристроях або в умовах поганого з'єднання.

Інші виклики включають масштабованість застосунків і безпеку. Оскільки вебдодатки стають все більш інтерактивними та складними, збільшується потреба в архітектурах, які можуть підтримувати велику кількість одночасних користувачів і забезпечувати безпеку даних.

Отже, розвиток архітектур вебдодатків пройшов від простих статичних сторінок до складних, інтерактивних і масштабованих систем. Кожен етап розвитку був відповіддю на нові виклики: продуктивність, взаємодія з користувачем, масштабованість і легкість підтримки. Сучасні архітектури продовжують розвиватися, використовуючи нові технології для покращення досвіду користувачів і продуктивності додатків.

1.2 Порівняння з аналогами

Архітектури побудови вебзастосунків еволюціонували з роками, і кожна нова концепція мала на меті вирішити певні проблеми попередніх підходів. У сучасному веброзробленні використовуються різні архітектури, серед яких

найбільш поширеними є SSR (Server-Side Rendering), SPA (Single-Page Applications), MPA (Multi-Page Applications) та Island architecture. Кожен підхід має свої сильні та слабкі сторони, і їх вибір залежить від специфіки проєкту, вимог до продуктивності та досвіду користувача.

Server-side rendering **(**SSR) надає вебсторінки, що генеруються на сервері перед тим, як бути відправленими користувачеві. Це дозволяє прискорити перше завантаження сторінки, але для подальших дій користувача кожен запит потребує з'єднання з сервером, що може вплинути на швидкість. Однак підхід і має недолік, а саме, генерація сторінки займає час на сервері, що може сповільнити видачу сторінки при високих навантаженнях.

Single-page Applications **(**SPA) є потужною архітектурою для створення динамічних, інтерактивних додатків, де весь контент завантажується за один раз, а подальші взаємодії оновлюють лише окремі частини сторінки без перезавантаження. Проте цей підхід може спричиняти проблеми з SEO та повільним початковим завантаженням, оскільки велика частина JavaScript-коду завантажується відразу, з ростом додатку може значно зрости і розмір бандлу, і великий проект запросто може займати він 3 до 10 мегабайт, що значно сповільнує скачування сторінки на девайсі користувача.

Multi-page Applications (MPA) використовує класичний підхід, де кожна сторінка вебдодатка існує окремо і генерується індивідуально на сервері [8]. Це забезпечує зручну SEO-оптимізацію, але призводить до збільшення часу на перезавантаження кожної сторінки. Так як кожна сторінка по суті незалежна, це шкодить плавності переходів за часто призводить до подвійного завантаження данних, що при комплексній логіці та слабкій мережі може призвести до значного сповільнення сайту.

Island architecture — це новий підхід, який намагається поєднати переваги SSR і SPA. Основна ідея полягає в тому, що статичні частини сторінки рендеряться на сервері, тоді як «острови» інтерфейсу, які потребують динамічної взаємодії, залишаються інтерактивними та оновлюються клієнтською частиною.

Це дозволяє зменшити обсяг JavaScript-коду, прискорити перше завантаження сторінки та забезпечити швидку інтерактивність.

Таблиця 1.1 – Порівняльна таблиця архітектур вебзастосунків

Показник	SSR	SPA	MPA	Island architecture
Продуктивність першого рендеру	+	-	+	+
Час на завантаження сторінки	+/-	-	+/-	+
SEO-оптимізація	+	-	+	+
Кількість JavaScript-коду	+/-	-	+	+
Інтерактивність	-	+	+/-	+
Масштабованість	+/-	+	+/-	+
Гідрація	+	-	+	+/-
Висновок	4	2	5.5	6.5

Отже, Island architecture виграє за багатьма показниками, особливо в контексті сучасних потреб вебдодатків. Вона поєднує переваги SSR у швидкому початковому рендерингу та SEO-оптимізації з інтерактивністю SPA. Крім того, Island architecture дозволяє мінімізувати гідрацію та зменшити обсяг JavaScript-коду, що позитивно впливає на продуктивність та досвід користувача.

1.3 Аналіз методів розв’язання задачі

Острівна архітектура вебдодатків, відома також як *Island Architecture*, є одним із найбільш інноваційних підходів у сучасній розробці. Її концепція базується на ідеї поділу користувацького інтерфейсу на автономні модулі — "острови". Кожен "острів" є самостійною функціональною одиницею, яка відповідає за виконання певних завдань. Це дозволяє ізолювати модулі один від одного, що мінімізує ризик помилок у системі, підвищує продуктивність і спрощує масштабування додатку.

Острівна архітектура передбачає низку ключових принципів, які визначають її ефективність:

1. Модульність. Кожен компонент інтерфейсу створюється як автономний блок з власною логікою, стилями і залежностями. Це дозволяє зменшити взаємозалежність між частинами додатку.
2. Локалізація залежностей. Залежності "островів" зберігаються локально, що дозволяє завантажувати лише ті ресурси, які потрібні для роботи конкретного модуля.
3. Оптимізація рендерингу. "Острови" можуть завантажуватися та рендеритися на вимогу, що зменшує час першого завантаження сторінки.

Зокрема, острівна архітектура популяризувалася у зв’язку з необхідністю зменшення обсягу JavaScript, який виконується на клієнті.

Сучасні вебдодатки часто страждають від надмірного обсягу JavaScript [9], що впливає на швидкість завантаження сторінки.

Використання острівної архітектури дозволяє розв’язати цю проблему, оскільки кожен "острів" містить лише необхідний мінімум коду.

JavaScript є основною мовою програмування для створення острівної архітектури, оскільки він дозволяє створювати динамічні, інтерактивні

інтерфейси. Його популярність зумовлена широким спектром інструментів і бібліотек, таких як React, Vue.js та Svelte, які дозволяють організувати додаток за принципом автономних компонентів.

JavaScript забезпечує модульність і динамічне виконання коду, що є критичним для острівної архітектури [10].

Node.js доповнює JavaScript, забезпечуючи серверне середовище для виконання коду, рендерингу сторінок на сервері (SSR) та роботи з API. Використання Node.js дозволяє значно оптимізувати продуктивність, оскільки частина роботи виконується ще до завантаження сторінки у браузері.

Це особливо важливо для додатків, які орієнтовані на глобальну аудиторію, де час завантаження має вирішальне значення.

Сучасні браузери відіграють роль універсального середовища для виконання коду. Їхня підтримка передових API, таких як Intersection Observer чи Web Components, дає змогу створювати автономні "острови", які взаємодіють з іншими частинами додатку лише за необхідності.

Правильна взаємодія між клієнтським і серверним кодом є ключем до створення продуктивних вебдодатків [11].

Окрім традиційного підходу з використанням JavaScript і Node.js, існують фреймворки, які спрощують впровадження острівної архітектури.

Наприклад, Astro (рис. 1.1) — інструмент, орієнтований на серверну генерацію сторінок із мінімальним обсягом JavaScript на клієнті [12].

Такий підхід дозволяє знизити навантаження на клієнтську частину та покращити продуктивність [13].

Рисунок 1.1 – Логотип фреймворку Astro

Іншим прикладом є Qwik (рис 1.2), який впроваджує концепцію миттєвої інтерактивності. Qwik завантажує JavaScript лише тоді, коли це потрібно для конкретної взаємодії [14]. Це дозволяє досягти високої продуктивності навіть на слабких пристроях. Аналогічно, SvelteKit забезпечує реактивну модель, яка спрощує створення автономних компонентів.

Рисунок 1.2 – Логотип фреймворку Qwik

Для розробників, які віддають перевагу іншим мовам, існують альтернативи, такі як Blazor [15], від Microsoft, логотип якого зображений на рисунку 1.3, що дозволяє будувати додатки на основі C#. Цей підхід може бути корисним у випадках, коли проєкт інтегрується в екосистему Microsoft. Недоліками є те що додаток не дає перевикористовувати код і кодова база може значно вирости і збільшити кількість проблемних місць. Також різні внутрішні протоколи Javascript і C# можуть викликати різнобіжності в очікуваній роботі і реальному результаті, тому це потребує кваліфікованого спеціаліста який досконало знає 2 мови на достатньому рівні.

Рисунок 1.3 – Логотип Blazor

Острівна архітектура може бути доповнена мікросервісною архітектурою на серверній стороні. Використання GraphQL чи REST API дає змогу кожному "острову" взаємодіяти лише з необхідними йому сервісами, що зменшує залежність від інших частин системи. Важлива незалежностість окремих сервісів у розподілених системах [16]. Цей підхід має як сильні так і слабкі

сторони, взаємодія з різними серверами для рендеру сторінки може прискорити за рахунок того що навантаження розприділяється, проте це може призводити до неконсистентності рендеру, певні куски можуть підгружатись набагато пізніше інших. Однак такий підхід має і свої переваги, зокрема мікросервізи забезпечують повну автономність островів, тож якщо один зі островів буде затриманий, вся інша сторінка буде зарендерена без жодних проблем. Мікросервіси дають можливість критичні островки віддавати статично, або використовувать найшвидші технології для таких задач, наприклад писати це на мовах без сбірника сміття, таких як Rust чи C++, що може забезпечити значний приріст швидкості та зменшити затрати на оплату інфраструктури. Мікросервіси безпроблемно масштабуються, більш того, за потреби можна розгорнути для критичних компонентів 2 сервіса, які будуть оркеструватись і працювати по черзі, на основі Round robin scheduling [17].

Обрання JavaScript у поєднанні з Node.js та сучасними браузерами як основного технологічного стека зумовлене їхньою універсальністю, широкою підтримкою спільноти і багатством екосистеми. Вибір технологій має ґрунтуватися на їхній здатності підтримувати основні принципи архітектури: модульність, гнучкість і масштабованість[18].

Альтернативи, такі як Astro чи Qwik, є доцільними для специфічних задач, однак для більшості проєктів універсальність і доступність JavaScript-орієнтованого підходу робить його оптимальним вибором.

Для реалізації динамічних частин контенту при високих навантаження, може бути важливим розробки критичних місць на інших мовах, в такому випадку можна використати комбінацію острівної архітектури з мікросервісами. Однак для більності випадків статичних частин контенту достатньо швидкодії Javascript в комбінації з її багатим вибором готових рішень різних проблем та

асинхронністю. Тому для виконання задачу була обрана саме вона. Можливе використання фреймворків для того, щоб забезпечити.

1.4 Постановка задач дослідження

Провівши дослідження існуючих підходів і архітектур вебдодатків, було визначено задачі, які необхідно вирішити у процесі розробки:

- розробити методи та модель острову;
- розробити інтерфейс адаптеру;
- розробити файл налаштувать для інтеграції з сбірником;
- розробка покращеної системи кешування;
- розробити web worker для роботи з гідрацією;
- провести тестування прототипу вебдодатка на основі острівної архітектури розробити зручний та зрозумілий графічний інтерфейс;
- підготувати методичні рекомендації для впровадження острівної архітектури.

1.5 Висновки

У першому розділі було детально розглянуто принципи острівної архітектури для побудови вебдодатків, а також аналіз популярних підходів і технологій, які реалізують цей архітектурний підхід. Були проаналізовані сучасні інструменти та платформи, такі як Astro, Qwik, Blazor, з урахуванням їхніх переваг і недоліків для розробки автономних, ізольованих компонентів. Висновки показали, що використання JavaScript і Node.js є найбільш оптимальним вибором завдяки їхній поширеності, широкій підтримці

екосистеми, а також здатності легко інтегруватися з іншими сучасними інструментами.

Крім того, було досліджено можливості мікросервісної архітектури як способу забезпечення гнучкості, масштабованості та модульності системи. Аналіз підтвердив, що мікросервіси ідеально підходять для підтримки острівної архітектури, забезпечуючи автономність кожного функціонального блоку і легкість інтеграції з іншими частинами додатка. Було також проведено аналіз способів взаємодії між компонентами за допомогою API, подієвої моделі та мережі служб, що дозволяє підвищити продуктивність і надійність системи.

У результаті аналізу було обрано сучасний підхід до розробки острівної архітектури з використанням мікросервісів і технологій JavaScript/Node.js. Також сформульовано основні задачі, які потрібно вирішити для удосконалення методів розробки вебдодатків: створення ізольованих модулів з ефективним управлінням їхнім станом, оптимізація взаємодії між сервісами і "островами", а також впровадження засобів моніторингу і масштабування системи.

2 РОЗРОБКА МЕТОДІВ ТА АЛГОРИТМІВ РОБОТИ БІБЛІОТЕКИ

2.1. Архітектурне проєктування бібліотеки

Вебкомпоненти стали ключовим вибором у розробці вдосконаленої архітектури, оскільки вони забезпечують високу гнучкість, модульність і сумісність із сучасними вебтехнологіями. Як частина відкритого стандарту, вебкомпоненти дозволяють створювати багаторазово використовувані модулі з ізольованою логікою та стилями, що повністю відповідає концепції острівної архітектури. Завдяки своїй незалежності від конкретних фреймворків чи бібліотек, вони гарантують тривалу підтримку і сумісність з майбутніми технологіями, що стало вирішальним аргументом при їхньому виборі.

Альтернативами вебкомпонентів є численні популярні фреймворки, такі як React, Angular, Vue.js чи Svelte. Кожен із них пропонує власну реалізацію компонентної моделі, яка зазвичай базується на специфічному рантаймі, що виконує логіку та рендеринг компонентів. Наприклад, у React компоненти обробляються через віртуальний DOM [19], а у Vue.js використовується двонапрямний зв’язок для управління станом [20]. Однак такий підхід має суттєві недоліки, які стають особливо помітними в масштабних проєктах.

По-перше, використання фреймворків вимагає завантаження додаткових рантайм-бібліотек, що збільшує розмір додатка та час завантаження сторінок. Це суперечить основній меті вдосконалення архітектури – створити легку і швидкодіючу систему. Наприклад, фреймворк React додає до проєкту кілька десятків кілобайтів залежностей, навіть для простих реалізацій, що є критичним у контексті оптимізації.

По-друге, фреймворки з часом змінюють свої API та підходи до роботи, що може призводити до складнощів у підтримці старих проєктів. Залежність від конкретного фреймворку робить систему менш гнучкою для адаптації до нових

стандартів чи інструментів, тоді як вебкомпоненти, як частина самого вебстандарту, залишаються стабільними і довговічними.

Остаточним аргументом на користь вебкомпонентів стало те, що багато сучасних фреймворків у кінцевому підсумку намагаються імітувати або наблизитися до концепції вебкомпонентів, але ці спроби супроводжуються великими накладними витратами. В результаті розробники часто стикаються з необхідністю ручної оптимізації для досягнення показників, які вебкомпоненти пропонують "з коробки".

Вебкомпоненти у вдосконаленій архітектурі забезпечили:

- Швидкодію – відсутність потреби у завантаженні важких рантаймів. Логіка виконується нативно в браузері, що суттєво зменшує затримки.
- Ізоляцію – Shadow DOM гарантує, що стилі та логіка компонентів не конфліктують із глобальними стилями чи іншими модулями [21].
- Сумісність як частину стандарту, вебкомпоненти підтримуються всіма сучасними браузерами, без необхідності залежності від сторонніх бібліотек.

Враховуючи вищезазначене, вебкомпоненти стали природним вибором для розробки вдосконаленої архітектури. У поєднанні з надійною основою генератора Eleventy, вони забезпечили потужний інструментарій для створення ефективної, довговічної та стандартизованої системи, яка легко масштабується та адаптується до нових викликів веброзробки.

Ще однією ключовою ідеєю стало використання вебворкерів як основного каналу комунікації для ручної регідрації. Вебворкери дозволяють переносити складні обчислення і обробку даних у окремі потоки, звільняючи основний потік браузера для обробки взаємодії з користувачем. Це рішення значно підвищило швидкодію та знизило ризик блокування інтерфейсу під час виконання ресурсомістких завдань. Використання вебворкерів також забезпечило більш

контрольований процес регідрації компонентів, дозволяючи оновлювати тільки ті частини, які потребують змін, що мінімізує витрати ресурсів і покращує загальний досвід користувача.

Кешування стало ще однією важливою складовою вдосконаленої архітектури. Було реалізовано комбінований підхід до кешування на стороні клієнта та сервера. На клієнтській стороні використовувалися технології, такі як IndexedDB і LocalStorage, що дозволило зберігати дані локально для швидшого доступу без необхідності повторного завантаження з сервера. Серверне кешування, у свою чергу, було реалізовано через інтеграцію із сучасними платформами для зберігання й доставки статичних ресурсів, що скоротило час завантаження та зменшило навантаження на сервер. Такий підхід забезпечив швидкий доступ до часто використовуваних даних і значно підвищив продуктивність системи в умовах високого навантаження.

Для базової інфраструктури було вирішено взяти за основу перевірений і надійний генератор статичних сайтів Eleventy [22] . Цей інструмент уже добре зарекомендував себе завдяки своїй простоті, стабільності та високій швидкості роботи. Використання Eleventy дозволило зосередитися на вдосконаленні ключових аспектів архітектури, не витрачаючи ресурси на створення базового каркаса системи. Платформа Eleventy забезпечила ефективний механізм генерації статичних ресурсів, який легко інтегрується з іншими сучасними інструментами і технологіями. Завдяки своїй мінімалістичній природі та гнучкості цей генератор став основою для реалізації покращеної архітектури з акцентом на швидкість і надійність.

Ставка на простоту та швидкодію в поєднанні з використанням передових технологій, таких як вебкомпоненти, вебворкери та комбіноване кешування, дозволила створити архітектуру, яка не тільки задовольняє сучасні вимоги до продуктивності, але й відкриває широкі можливості для подальшого вдосконалення. Інтеграція надійного інструменту Eleventy забезпечила стабільну

основу, на якій були побудовані всі покращення, що дозволило досягти високої ефективності системи та створити міцну платформу для майбутніх інновацій.

На рисунку 2.1 зображено принцип завантаження сайта, динамічна компоненти відділені від статичних, завантажуються по умові, коли вони стають необхідними, а статичні компоненти віддаються візразу.

При завантаженні клієнтського компоненту виконується гідрація, що забезпечить консистентність даних, при тому вона відбувається лише в тих місцях, де це необхідно.

Рисунок 2.1 – Принцип розбиття сайту

Для зручності розробки розробнику важливо мати інструменти, які забезпечать його необхідним функціоналом, більшість цих речей можуть реалізовувати фреймворки, система проектувалась таким чином, що розробник може інтегрувати будь-який фреймворк написавши для цього адаптер, однак основними рекомендованими фреймворками є так звані фреймворки без рантайму, тобто ті фреймворки які не створюють свій віртуальний DOM і не абстраговані від браузеру, таким чином це і дозволяє збільшити швидкодію, і зменшує необхідну роботу, яку буде виконувати вебворкер.

Для створення додатку необхідно мати так званий білдер – систему, що збере і скомпілює додаток у набір файлів, які потім і буде віддавати сервер. Існує багато підходів зі своїми плюсами та мінусами, перевага була надана Eleventy, як простомі і надійному генератору статичних сайтів. На його основі було створено бібліотеку, яка розширює можливості та дозволяє реалізувати острівну архітектуру без інших зовнішніх залежностей.

2.2 Розробка методу покращення гідрації компонентів

Для того щоб отримати найкраще з двох світів, статичного і динамічного, необхідно мати ефективний механізм гідрації, який не буде використовувати забагато ресурсів системи і тим самим тормозити швидкість роботи вебдодатку.

Гідрація є процесом активації інтерактивності компонентів після їх статичного завантаження, також вона виконує роль зберігача консистентності даних. Основними її проблемами є кількість завантажень і виконання великої кількості коду перед тим як сторінка стане повноцінно інтерактивною [23]. Часткова гідрація дозволяє гідрувати лише інтерактивні компоненти, тим самим зменшуючи кількість необхідної роботи для браузера.

Web Worker є потужним інструментом для оптимізації роботи вебдодатків, особливо у контексті гідрації компонентів. Головною перевагою використання Web Worker є можливість виконання обчислень у фоновому режимі, ізоляція цих

процесів від основного потоку браузера. Це означає, що навіть складні завдання, такі як обробка даних або гідрація динамічних компонентів, не блокуватимуть рендеринг сторінки або взаємодію користувача з інтерфейсом. У контексті острівної архітектури Web Worker може відповідати за підготовку інтерактивних елементів, обробку їх стану та передачу готових результатів у DOM. Такий підхід забезпечує більш плавну взаємодію з користувачем, зменшує час затримки, сприяє рівномірному розподілу обчислювальних навантажень та дозволяє повністю використати апаратні ресурси пристрою. Крім того, Web Worker створює ізольоване середовище, що мінімізує ризик конфліктів між компонентами, і надає можливість паралельної роботи, покращуючи загальну масштабованість системи.

Основними покращеннями є інтеграція гідрації у web-worker та надання розробнику контроля за процесом гідрації.

Покращений метод гідрації включає в себе обробку гідрації вебворкером, таким чином основний потік подій буде менше навантажений що дозволить збільшити швидкодію і плавність користувацького інтерфейсу. На кожен вебворкер браузер при потребі виділяє окремий потік виконання і таким чином можна зберегти швидкодію і зробити виникнення помилок при гідрації локальним для одного острівця, помилка не завадить дії інших частин додатку.

На рисунку 2.2 зображено діаграму послідовності [24] покращення на основі інтеграції вебворкера, після отримання даних про острови згенеровані на сервері, клієнт створює вебворкера, якому надається завдання гідрації дерева компонентів, гідрація може відбуватись рекурсивно якщо компонент має підкомпоненти. Вебворкери за рахунок того що дії відбуваються асинхронно забезпечують швидку гідрацію і завантаження сторінки. Після виконання роботи вони віддають дані по шині подій, які компонент відловлює, після чого використовує. Роль сервера мінімальна, при тому за потреби сервер може

віддавати посилання на клієнтський компонент який браузер буде отримувати з іншого серверу.

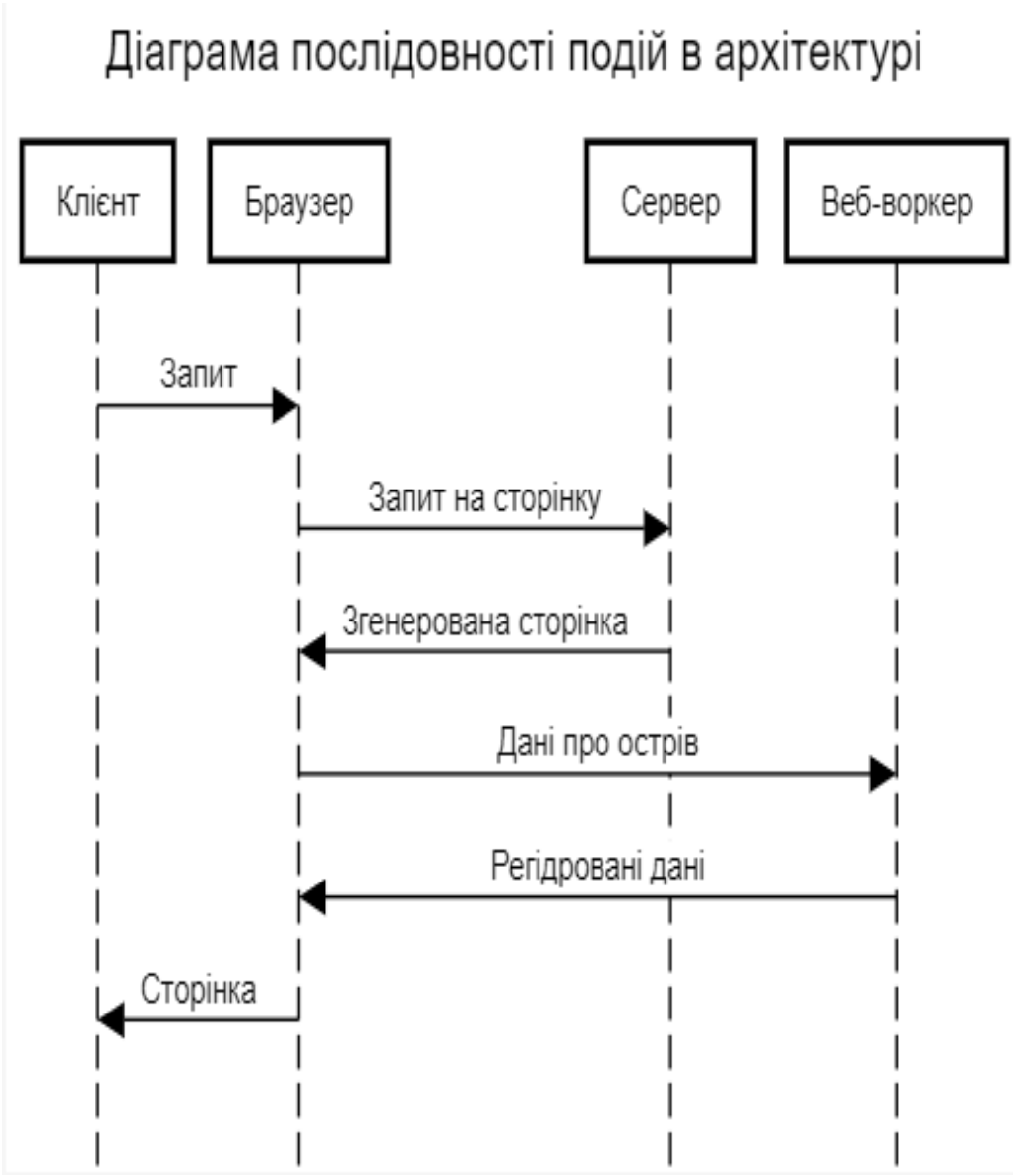


Рисунок 2.2 – Діаграма послідовності при покращеній гідрації

Процес того як буде відбуватись гідрація певною мірою підконтрольний розробнику, по перше розробник за допомогою вебкомпоненту може визначити компонент який необхідно гідрувати, та спеціальним синтаксисом надати послідовність дій та що показувати поки відбувається гідрація. Це відбувається за допомогою вебкомпонентів та синтаксису html дата атрибутів, які уже система

розпізнає і працює з ними. Хоча вебкомпоненти не дуже часто використовуються сьогодні, але це зазначений стандарт, який підтримує більшість браузерів в останній версіях, стандарт розробляється і буде пітримуватись і надалі, тому вибір такого підходу забезпечить надійність і стандартизацію підходу.

2.3 Розробка методу оптимізації завантаження сторінок

Оптимізація завантаження вебсторінок є комплексним процесом, який включає зменшення часу завантаження, поліпшення користувацького досвіду та ефективне використання системних ресурсів. У сучасній веброзробці цей процес часто базується на інтеграції вебкомпонентів із такими технологіями, як часткова гідрація, лінійне завантаження (lazy loading), попереднє рендерення (pre-rendering) та використання Web Workers. Ці підходи забезпечують ізоляцію компонентів, їх модульність та адаптивність до змінюваних умов роботи системи.

Вебкомпоненти виступають ядром цього підходу, оскільки вони базуються на стандартизованих можливостях браузера і забезпечують уніфікований спосіб створення та управління інтерактивними елементами інтерфейсу. Технології, які лежать в основі вебкомпонентів, включають кастомні елементи (Custom Elements), тіньовий DOM (Shadow DOM) та шаблони HTML (HTML Templates). Кастомні елементи дозволяють створювати власні HTML-теги з визначеною розробником поведінкою, що робить систему гнучкішою і більш налаштованою на конкретні потреби. Тіньовий DOM ізолює стилі та структуру компонента, запобігаючи конфліктам із іншими елементами на сторінці. Використання шаблонів HTML забезпечує можливість завантаження в DOM лише необхідних компонентів, що позитивно впливає на загальну продуктивність.

Однією з ключових стратегій оптимізації завантаження є попереднє рендерення. На етапі генерації сторінок створюється статичний HTML із мінімально необхідними стилями та скриптами, що суттєво скорочує час

відображення першого вмісту (First Contentful Paint). У цьому контексті кастомні елементи інтегруються як статичні компоненти, які пізніше отримують динамічну функціональність.

Інший важливий підхід — лінійне завантаження. За допомогою API Intersection Observer компоненти завантажуються тільки тоді, коли вони потрапляють у видиму область екрану користувача. Такий метод дозволяє уникнути завантаження зайвих даних і суттєво економить ресурси системи, особливо на сторінках із великим обсягом контенту.

В рамках розробки цього методу була розроблена інтеграція з найбільш поширеними сучасними браузерними API, які надають можливість слухати події та можуть знадобитись розробнику під час розробки власного додатку.

На рисунку 2.3 зображено блок-схему оптимізованого процесу завантаження при отримані сторінки:

1. Отримуються статичні дані сторінки.
2. Статичні дані відображаються, в статичних даних знаходяться метадані острову та так званий fallback, даті що відображаються в той час поки дані острову все ще не отримані.
3. Потім відбувається процес слухання браузера (в асинхронному форматі) на те чи відбулась подія задана в острові на відображення контенту.
4. Якщо подія відбулась то відбувається підвантаження острову, перехід на пункт 6.
5. Якщо не відбулась вона очікується надалі.
6. Гідрація компоненту.
7. Відображення острову.

Рисунок 2.3 – Блок-схема оптимізованого процесу завантаження

Часткова гідрація також стала невід'ємною частиною сучасних оптимізацій. Вона дозволяє розділити сторінку на незалежні інтерактивні "острови", які гідруються лише за необхідності. Це знижує навантаження на браузер і значно зменшує обсяг JavaScript, що завантажується. Таким чином, кожен компонент отримує динамічну функціональність тільки тоді, коли це дійсно потрібно.

Важливу роль у цьому процесі відіграють Web Workers, які дозволяють переносити важкі обчислення або процеси гідрації компонентів в окремі потоки (рис. 2.4). Це забезпечує вільність основного потоку браузера для швидкого рендерингу сторінки та реакції на дії користувача. Наприклад, складні обчислення стану компонента можуть виконуватися у Worker, а їх результати передаватися назад у DOM, мінімізуючи затримки.

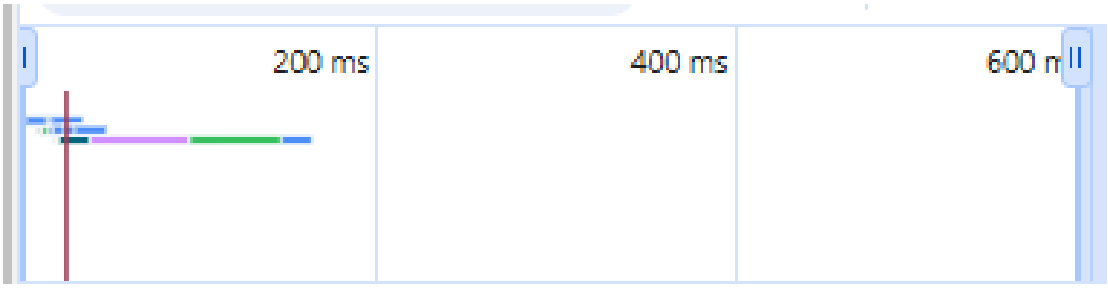


Рисунок 2.4 – Діаграма завантаження сторінки в браузері

Окрім цього, стратегія пріоритизації ресурсів дозволяє забезпечити ефективне завантаження. Критично важливі файли (наприклад, ключові стилі та скрипти) завантажуються першочергово, тоді як менш важливі ресурси обробляються асинхронно. Використання Shadow DOM сприяє ізоляції стилів, усуваючи необхідність у великих глобальних CSS-файлах і значно скорочуючи час рендерингу сторінки.

Динамічне імпортування модулів, підтримуване браузерами через ES-модулі, дозволяє завантажувати логіку компонентів тільки тоді, коли вона потрібна. Наприклад, якщо компонент використовується залежно від взаємодії користувача, його код завантажується безпосередньо під час цієї взаємодії, зменшуючи початковий обсяг завантаження.

Результатом застосування таких підходів є значне прискорення завантаження сторінок, зниження обсягу переданих даних і підвищення

загальної продуктивності. Модульність вебкомпонентів у поєднанні з частковою гідрацією, лінійним завантаженням та використанням Web Workers забезпечує системі гнучкість, масштабованість і довговічність. Ці переваги роблять вебкомпоненти ідеальним вибором для сучасної архітектури, особливо в умовах необхідності зменшення залежності від рантаймів і глобальних бібліотек, які часто стають перешкодою для оптимізації.

2.4 Розробка методу взаємодії між компонентами "островів"

Розробка методу взаємодії між компонентами "островів" є ключовим етапом побудови сучасної вебархітектури, яка базується на принципах ізольованості, модульності та ефективного управління ресурсами. У контексті острівної архітектури, де кожен інтерактивний блок функціонує як окремий модуль, взаємодія між цими блоками повинна бути організована таким чином, щоб забезпечити якнайменше втручання у внутрішню логіку компонентів і водночас підтримувати стабільність та надійність системи.

Основна проблема полягає в тому, що інтерактивні "острови" є незалежними компонентами, які можуть бути гідровані у різний час і функціонувати автономно. Це створює виклик у забезпеченні безпечної, уніфікованої та ефективної комунікації між ними. Традиційні підходи, такі як використання глобальних подій або централізованих сховищ стану, мають низку недоліків. Наприклад, глобальні події можуть спричиняти конфлікти між компонентами, особливо у великих проєктах, а централізовані сховища додають складність у розгортанні і підтримці масштабованості.

З огляду на ці виклики, було вирішено розробити протокол взаємодії між компонентами, який базується на стандартних можливостях браузера, зокрема на механізмі postMessage. Цей стандарт дозволяє забезпечити безпечну передачу повідомлень між різними частинами системи, зокрема між окремими iframe, Web

Workers або навіть між окремими ве-компонентами, що працюють у межах одного документу.

Кожен "острів" отримує унікальний ідентифікатор, який використовується як ключ для ідентифікації джерела або адресата повідомлення. Завдяки цьому підходу компоненти можуть взаємодіяти один із одним, не створюючи жорстких зв’язків, що є важливим принципом для підтримання модульності. Наприклад, якщо один "острів" представляє список доступних закладів харчування, а інший відповідає за карту, на якій відображаються ці заклади, взаємодія відбувається через передачу повідомлень. Список може надсилати дані про обраний заклад, а карта — динамічно оновлювати своє відображення відповідно до отриманого повідомлення.

Процес організації цієї взаємодії включає кілька важливих етапів. По-перше, всі компоненти реєструються в системі через їх унікальні ідентифікатори. Це дозволяє визначати, які саме модулі мають бути адресатами повідомлень. По-друге, кожен "острів" створює локальний обробник, який прослуховує повідомлення, надіслані через postMessage, і виконує відповідні дії, залежно від отриманих даних. Нарешті, кожен компонент має можливість ініціювати повідомлення, використовуючи методи, які доступні через стандартний API.

Таке рішення має низку переваг. По-перше, використання postMessage як базового протоколу гарантує сумісність із усіма сучасними браузерами. Це означає, що система не залежить від специфічних фреймворків або бібліотек, що позитивно впливає на її масштабованість і довговічність. По-друге, механізм унікальних ідентифікаторів дозволяє уникнути конфліктів між компонентами навіть у випадках, коли система має багато інтерактивних "островів"(рис 2.5).

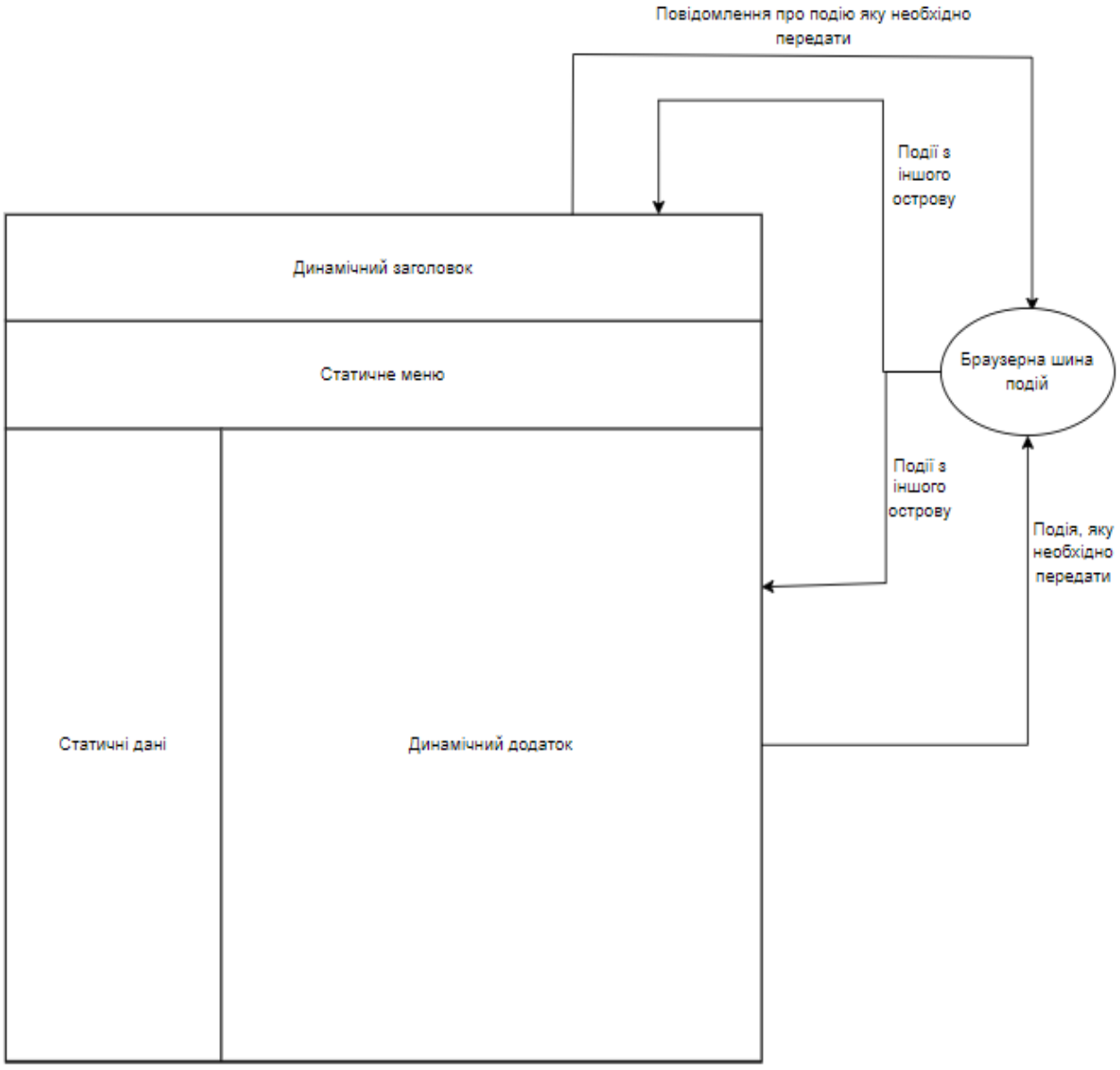


Рисунок 2.5 – Демонстрація принципу шини подій для спілкування компонентів

Крім того, запропонований підхід забезпечує високу гнучкість. Наприклад, якщо у майбутньому потрібно додати новий компонент, він може бути легко інтегрований у систему без необхідності змін у вже існуючих "островах". Достатньо лише призначити новому компоненту ідентифікатор і налаштувати обробники для обміну повідомленнями.

У результаті розроблений протокол на основі postMessage став надійним і масштабованим рішенням для взаємодії між "островами". Це дозволяє підтримувати незалежність кожного компонента, одночасно забезпечуючи необхідну інтеграцію для спільної роботи. Такий підхід відповідає сучасним вимогам до модульності, ефективності та сумісності вебдодатків, роблячи його оптимальним вибором для реалізації острівної архітектури.

2.5 Розробка методу покращення продуктивності та масштабованості

Розробка методу покращення продуктивності та масштабованості сучасних вебдодатків з острівною архітектурою була спрямована на забезпечення зручності для розробників і оптимізації роботи браузерів. Головним завданням стало створення інструментів, які дозволяють легко позначати інтерактивні компоненти для динамічного завантаження за певних умов, водночас покращуючи продуктивність сторінок за рахунок ефективного управління статичним і динамічним контентом.

Ключовою ідеєю було впровадження механізму роботи з "островами" — ізольованими частинами сторінки, що можуть бути незалежно завантажені та ініціалізовані. Для розробників цей підхід надає виняткову простоту: достатньо лише позначити компонент як динамічний чи статичний і вказати умови його завантаження. Наприклад, розробник може налаштувати, щоб певний острів завантажувався лише тоді, коли користувач прокручує сторінку до його області видимості, або щоб інтерактивна логіка активувалася лише після певної події, наприклад кліку чи наведення курсора.

З технічної точки зору це стало можливим завдяки інтеграції механізмів лінивого завантаження (lazy loading) і часткової гідрації (partial hydration). Ліниве завантаження дозволяє уникати завантаження зайвого JavaScript, забезпечуючи лише ту функціональність, яка актуальна в конкретний момент. Наприклад, великі таблиці, слайдери або інтерактивні карти не ініціалізуються до моменту,

поки вони не будуть потрібні. Це не лише зменшує обсяг переданих даних, але й дозволяє браузеру сфокусуватися на швидкому рендерингу базового контенту.

Часткова гідрація, зі свого боку, дозволяє завантажувати інтерактивну логіку для окремих частин сторінки, не впливаючи на інші компоненти. Наприклад, якщо острів представляє собою динамічну форму, її JavaScript-код завантажується і ініціалізується незалежно від інших компонентів, таких як статичні блоки тексту чи зображення. Це створює ефект мінімальної залежності між різними частинами додатка, що особливо важливо для складних систем з великою кількістю інтерактивних елементів.

Особлива увага приділялася статичним островам, які не потребують динамічної логіки, але можуть бути повторно використані на різних сторінках. Для таких компонентів було розроблено систему розумного кешування, яка забезпечує можливість завантаження їх з локального кешу браузера. Це дозволяє уникати повторних запитів до сервера навіть при переході між сторінками, що суттєво прискорює навігацію. Наприклад, якщо статичний банер, меню або футер використовується на кількох сторінках, він завантажується лише один раз і зберігається в кеші для подальшого використання.

Service Worker є ключовою технологією для реалізації розумного кешування в острівній архітектурі, що дозволяє суттєво покращити продуктивність вебдодатків і забезпечити безперебійний досвід користувача[25]. Основна перевага Service Worker полягає у його здатності діяти як посередник між браузером і сервером, дозволяючи управляти мережею та кешем з високим рівнем контролю. У контексті острівної архітектури Service Worker може бути використаний для кешування статичних "островів" на клієнтській стороні, зберігаючи попередньо відрендерений HTML, CSS і JavaScript компоненти. Завдяки цьому повторне завантаження сторінки або навігація між її частинами здійснюється значно швидше, оскільки браузеру не потрібно повторно звертатися до сервера для отримання вже кешованих даних (рис 2.6).

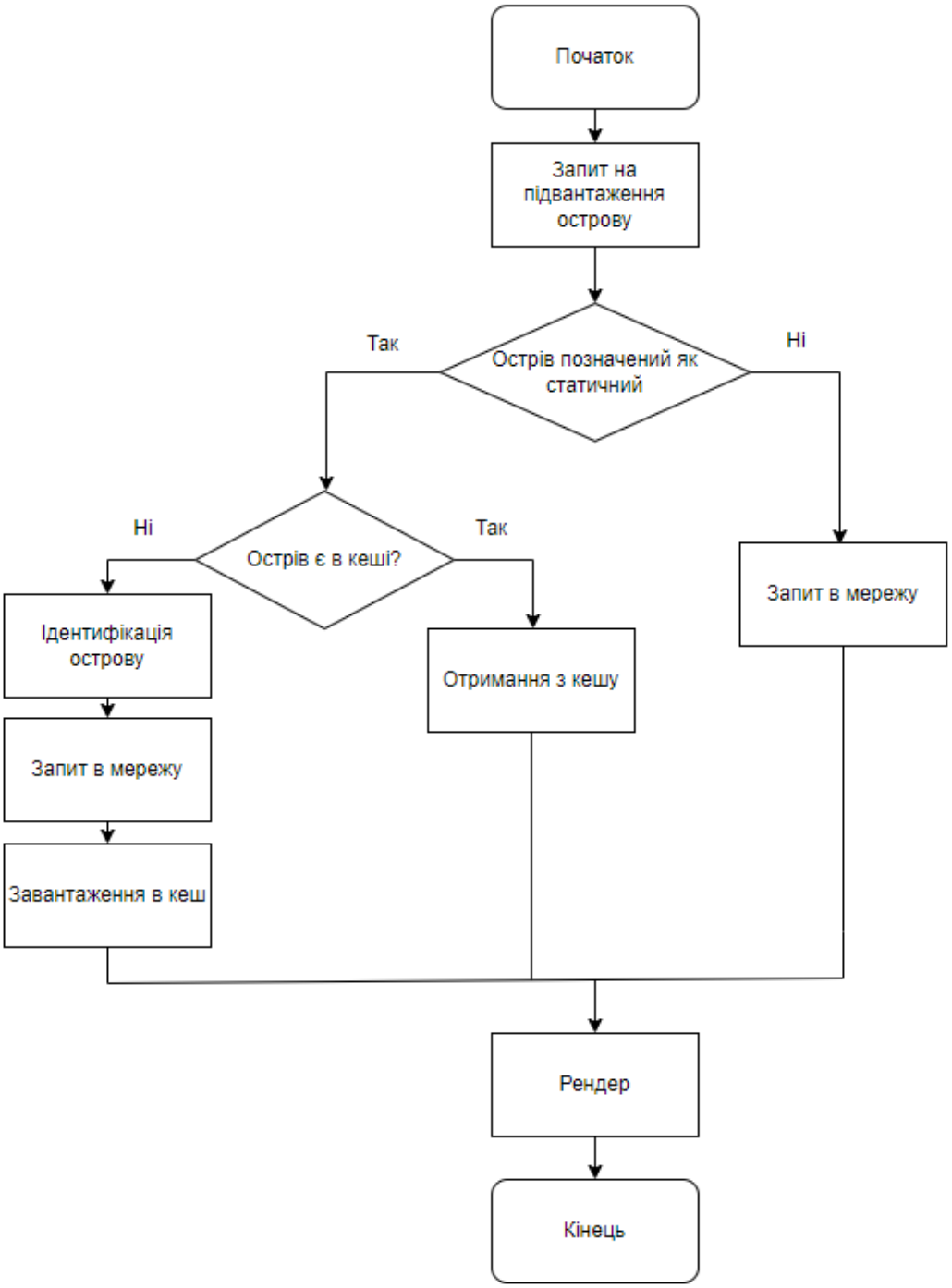


Рисунок 2.6 – Блок схема кешування за допомогою Service Woker

Розглянемо алгоритм роботи Service Worker, перша за все йде перевірка на те чи статичний це остів, статичні острови позначаються специфічно, у випадку якщо він не статичнй то відбувається підвантаження та рендер, зберігати його не має сенсу.

Якщо острів статичний йде перевірка його в кеші, у випадку якщо його немає він туди підвантажується після чого йде його відображення.

Додатково Service Worker дозволяє впровадити умовне оновлення кешу. Наприклад, при оновленні контенту сервер може надсилати сигнали оновлення або індикатори змін, які Service Worker використовує для оновлення лише тих островів, які зазнали змін. Це зменшує обсяг мережевого трафіку і знижує час оновлення. Інтелектуальне управління кешем також включає стратегічний вибір, коли завантаження відбувається з кешу, а коли — з мережі, залежно від поточного стану підключення чи критичності даних.

Ще однією значною перевагою є можливість роботи в режимі офлайн. Оскільки статичні "острови" кешуються на пристрої користувача, вони залишаються доступними навіть без активного інтернет-з'єднання. Це покращує загальну стійкість додатку і створює більш зручний досвід для користувача.

Розумне кешування також враховує динаміку оновлення контенту. Якщо статичний острів потребує змін, система автоматично інвалідизує застарілі дані в кеші та оновлює їх новою версією. Це стало можливим завдяки унікальним ідентифікаторам, які генеруються для кожного статичного компонента. Такий підхід забезпечує актуальність даних і водночас дозволяє уникнути зайвих завантажень.

З точки зору розробника, робота з такими механізмами виглядає максимально простою. Достатньо позначити острів як статичний або динамічний і визначити умови його завантаження. Наприклад, компонент може мати атрибут lazy для лінивого завантаження або атрибут cacheable для інтеграції з системою кешування. Усі складні операції, пов’язані з управлінням ресурсами, відбуваються автоматично, без потреби в додаткових зусиллях.

Для браузерів запропонований метод забезпечує значні переваги. Розділення логіки компонентів і впровадження лінивого завантаження дозволяє зменшити обсяг JavaScript, який необхідно обробити на етапі початкового

завантаження сторінки. Це прискорює час відображення першого контенту (First Contentful Paint) і забезпечує більш плавну взаємодію з додатком навіть на пристроях з низькою продуктивністю.

Загалом, розроблений метод покращення продуктивності та масштабованості поєднує простоту розробки з технічною ефективністю. Він надає розробникам інтуїтивні інструменти для оптимізації роботи додатка, одночасно забезпечуючи користувачам швидке завантаження сторінок і плавну взаємодію з контентом. Це створює базу для сучасних вебдодатків, які відповідають вимогам високої продуктивності та здатності масштабуватися під потреби користувачів.

2.6 Розробка адаптерів для бібліотеки

Розширення можливостей використання різних фреймворків у веброзробці часто вимагає адаптації специфічних особливостей цих фреймворків до існуючих архітектурних рішень. Для цього використовуються адаптери — спеціалізовані компоненти, які забезпечують інтеграцію між фреймворком і базовою інфраструктурою проєкту.

Адаптер реалізує ключовий принцип сумісності: він дозволяє використовувати компоненти фреймворку у рамках системи Eleventy, забезпечуючи плавну інтеграцію шляхом передрендерення компонентів на сервері і створення готових HTML-структур. Основна мета такого підходу полягає у збереженні продуктивності і простоти обробки даних, що є критичним для розробки бібліотеки.

Зазвичай більшість фреймворків реалізують певну «точку входу», зазвичай це функція render, яка використовується для того щоб додаток зарендерився в потрібне місце. Кожен динамічний острів і буде використовувати таку функцію, якщо є необхідність використання фреймворків, у випадку якщо фреймворк не

має точки входу а використовує якусь специфічну систему втиснення в HTML, такий адаптер розробнику доведеться написати самому. Це дозволяє створювати статичний HTML-контент, що зменшує залежність від клієнтського рендерингу і скорочує час до відображення першого вмісту. Це рішення підходить для застосунків, де важливо забезпечити SEO-оптимізацію і швидкість завантаження.

Важливим аспектом є динамічне підключення JavaScript на клієнті за допомогою адаптованої системи імпортів. У цьому прикладі файл, який містить логіку клієнтського компонента, зберігається у структурі проєкту і автоматично доступний через URL-адресу, що генерується адаптером. Цей підхід дозволяє забезпечити модульність і масштабованість рішення, оскільки розробник може легко налаштувати, які ресурси потрібні для завантаження.

Використання базових компонентів Node.js достатньо для того щоб надати високу гнучкість у розробці. Наприклад, компонент може бути написаний у вигляді окремого файлу, а його інтеграція до Eleventy виконується без необхідності додаткових змін у вихідному коді компонентів.

Потенціал розширення цього підходу є значним. Завдяки універсальній структурі адаптера, можна легко додати підтримку інших популярних фреймворків, таких як React, Vue чи Svelte. Це забезпечує можливість створення проєктів із багатофреймворковою екосистемою, дозволяючи розробникам вибирати оптимальні інструменти для конкретних задач без втрати сумісності з основною архітектурою. Такий підхід також сприяє зменшенню залежності від обраного фреймворка, забезпечуючи довготривалу стабільність і гнучкість системи.

Загалом, створення адаптерів, як у наведеному прикладі, дозволяє значно спростити інтеграцію між різними технологіями, зберігаючи при цьому високу продуктивність і зручність для розробників. Це рішення відкриває нові

можливості для адаптації архітектури до змінюваних потреб, що є важливим аспектом сучасної веброзробки.

2.7 Розробка загального алгоритма роботи бібліотеки

Архітектура «островів» (Island Architecture) являє собою передовий підхід у розробці вебдодатків, що поєднує гнучкість, модульність і високу продуктивність. Система забезпечує швидке завантаження сторінок, інтерактивність і зручність для кінцевих користувачів за рахунок розподілу сторінки на окремі автономні компоненти — так звані "острови". У цьому розділі буде розглянуто загальний алгоритм роботи системи, підкріплений описом діаграм UML, які наочно ілюструють структуру, взаємодію компонентів і послідовність дій.

Базовий алгоритм роботи архітектури "островів" складається з декількох ключових етапів, які забезпечують поступове завантаження і ініціалізацію контенту з метою покращення продуктивності та забезпечення плавної взаємодії з користувачем. Процес починається з ініціації запиту користувачем, що може здійснюватися через введення URL-адреси або взаємодію з інтерфейсом. На етапі серверного рендерингу статичний контент генерується і передається клієнту для забезпечення швидкого відображення базового контенту. Після цього починається завантаження динамічних компонентів, що забезпечують взаємодію та інтерактивність сторінки. Особливу роль у цьому процесі відіграють Service Workers, що забезпечують кешування і підвищення продуктивності шляхом організації розумного доступу до збережених даних.

Для кращого розуміння роботи системи і її основних компонентів було створено UML-діаграми, що демонструють структуру та взаємодію різних частин системи. UML (Unified Modeling Language) є інструментом для візуалізації, що полегшує розробку і аналіз складних систем.

Діаграма випадків використання (рис. 2.7) наочно демонструє ключові взаємодії користувачів із системою, акцентуючи увагу на основних процесах.

Основними акторами на діаграмі є "User" (користувач) та "Service Worker", кожен із яких виконує певні дії, необхідні для роботи системи. Користувач ініціює початковий запит для отримання контенту, який у подальшому рендериться і відображається. Після цього відбувається завантаження динамічного контенту та його кешування для подальшого використання. Актор "Service Worker" взаємодіє з кешем, відповідаючи за зберігання статичних даних і забезпечення доступу до них у режимі офлайн.

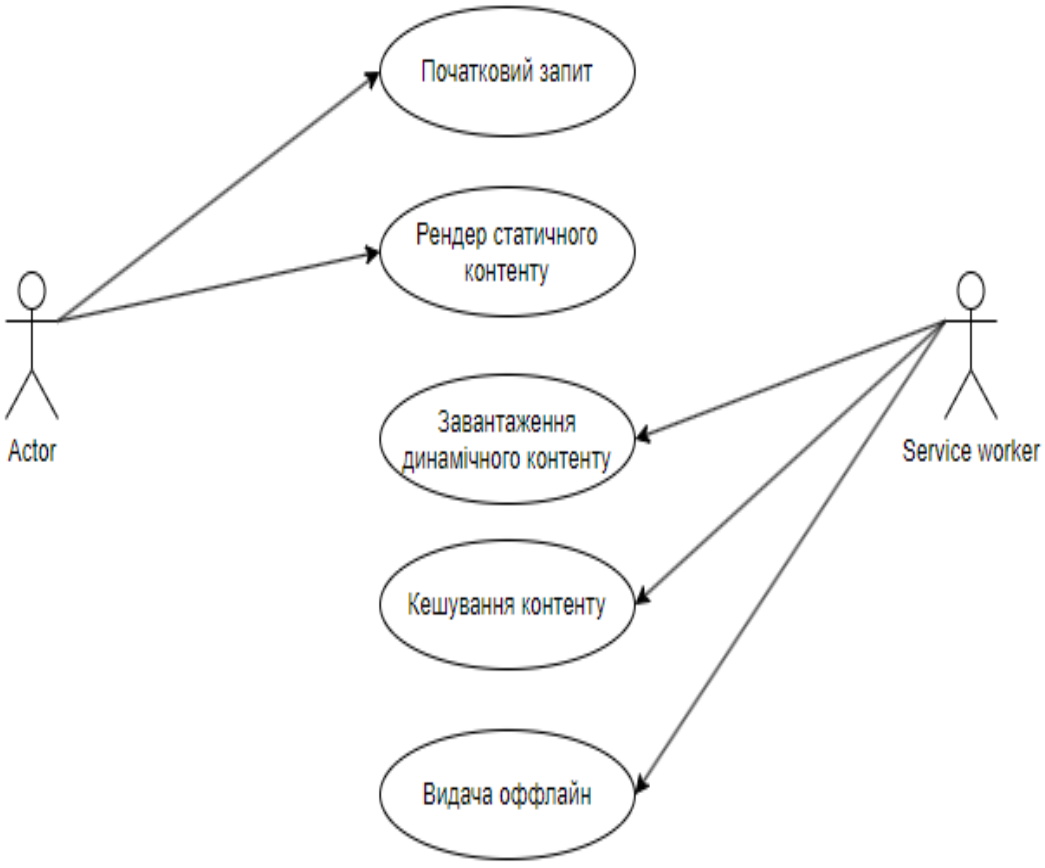


Рисунок 2.7 – Діаграма випадків використання

На діаграмі показано, як "User" ініціює запит, отримує статичний контент, а "Service Worker" відповідає за збереження і обслуговування запитів до кешу.

Така модель забезпечує плавну і безперебійну роботу додатка незалежно від доступності інтернет-з'єднання.

Діаграма послідовності (рис. 2.8) ілюструє порядок виконання різних етапів алгоритму та дозволяє побачити, як відбувається взаємодія між компонентами системи у часі. Вона відображає, як користувач, клієнт, сервер, Service Worker і Cache взаємодіють між собою для забезпечення безперебійної роботи системи.

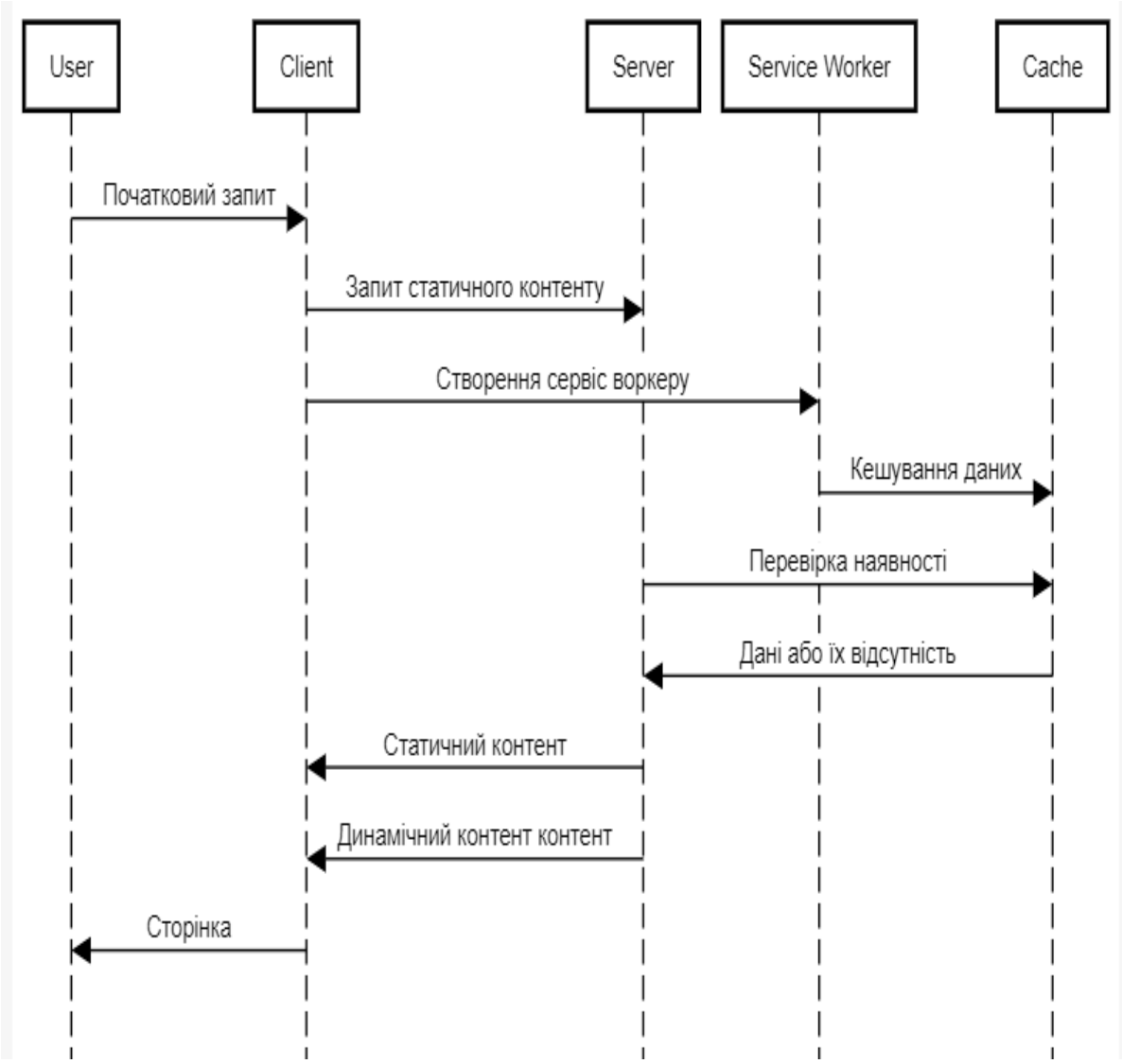


Рисунок 2.8 – Діаграма послідовності

1. Користувач ініціює запит на отримання контенту через клієнтський додаток.
2. Клієнт надсилає запит на сервер для отримання статичного контенту, який забезпечує швидке відображення основних елементів сторінки. Після цього частина сторінки уже вважається готовою до використання.
3. Після цього клієнт реєструє Service Worker, що відповідає за обробку мережевих запитів та організацію кешування.
4. Service Worker перевіряє наявність запитуваних даних у кеші та кешує нові дані для подальшого використання.
5. У випадку, якщо користувач працює офлайн, клієнт звертається до Service Worker, який надає доступ до кешованого контенту.
6. Service worker перевіряє у кеші доступність контенту, зазвичай статичні острови будуть закешовані, а отже розмітка і стилі будуть відображені користувачу. Такий підхід гарантує, що навіть у випадках втрати з'єднання користувач зможе продовжувати роботу з додатком без затримок.
7. Сервер відповідає контентом.
8. Дані віддаються користувачу.

Важливим є те що при помилці або затримці на етапі підвантаження островів, інший контент відображається нормально, тим самим це надає певні гарантії безпеки.

Ця діаграма наочно показує, як системні компоненти взаємодіють між собою та які процеси ініціюються кожним учасником. Завдяки цій моделі розробники можуть легко ідентифікувати вузькі місця або точки, що потребують оптимізації, а також зрозуміти, як відбувається забезпечення актуальності кешованих даних.

Діаграма класів (рис. 2.9) описує структуру основних компонентів архітектури «островів» і показує взаємозв'язки між ними. Вона представляє ключові класи, що складають архітектуру системи, зокрема: User, Client, Server, Service Worker, Cache і Dynamic Component, Island, Conditions. Кожен клас відповідає за певну функціональність, має свої атрибути і методи, які визначають його роль у загальній системі. На діаграмі можна побачити основні принципи взаємодії класів між собою, і той момент що система не має високої зв’язності, і може вільно розширюватись за допомогою адаптерів.

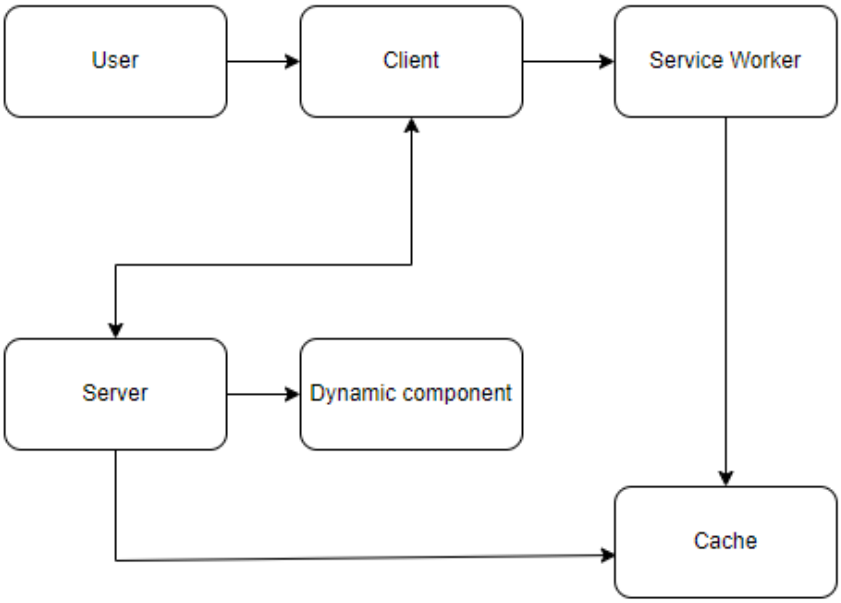


Рисунок 2.9 – Діаграма класів архітектури «Острови»

Розглянемо класи, що використовуються в додатку:

- Клас User відображає користувача системи, який ініціює запити та взаємодіє з інтерфейсом. Він має атрибути, такі як ID, ім'я, та методи для виконання запитів.

1. Клас Client є основною точкою взаємодії з користувачем на боці браузера і відповідає за обробку запитів, отримання статичного контенту та інтеграцію з Service Worker.
2. Клас Server виконує функцію обробки запитів та генерації відповідей. Він надає HTML-сторінки, які включають статичний контент для швидкого відображення.
3. Клас Service Worker відповідає за кешування даних і забезпечення роботи додатка в офлайн-режимі. Цей клас має методи для реєстрації та обробки кешу, а також для отримання кешованих даних.
4. Клас Cache є абстракцією системи зберігання даних, яка використовується для зберігання статичного контенту, що може бути використаний повторно без звернення до сервера.
5. Клас Dynamic Component містить методи і атрибути для роботи з динамічними елементами сторінки, які забезпечують інтерактивність.
6. Клас Island: Реалізує концепцію "островів", які є окремими компонентами сторінки. Забезпечує динамічну гідрацію цих компонентів, включаючи методи для визначення умов завантаження, управління станом та рендерингом.
7. Клас Conditions: Відповідає за управління умовами завантаження і гідрації компонентів. Має методи, які дозволяють визначати умови, такі як видимість компонента (visible), взаємодія з користувачем (interaction), режим очікування (idle), тощо. Ці методи використовуються для того, щоб компоненти завантажувались лише тоді, коли це необхідно, що значно покращує продуктивність системи.

Ця діаграма класів дозволяє зрозуміти структуру системи на рівні об'єктів і методів, і показує, як різні компоненти співпрацюють між собою, щоб забезпечити стабільність та продуктивність вебдодатка. Вона також підкреслює

модульність системи, де кожен компонент виконує свої обов'язки ізольовано, забезпечуючи легкість у підтримці та масштабуванні додатка.

2.8 Висновки

У другому розділі здійснено архітектурне проєктування та моделювання системи, спрямоване на вдосконалення острівної архітектури для сучасних вебдодатків. Було розроблено цілісну модель архітектури, що враховує використання вебкомпонентів як базової технології завдяки їхній модульності, ізольованості та відповідності стандартам вебу. Це дозволило забезпечити високий рівень гнучкості, сумісності та довговічності системи.

Особливу увагу було приділено впровадженню механізмів кешування на стороні клієнта та сервера для підвищення продуктивності, а також використанню вебворкерів як ефективного інструменту для оптимізації процесу регідрації компонентів. Таке рішення дозволило мінімізувати затримки та зменшити навантаження на основний потік браузера, що сприяє покращенню користувацького досвіду.

Ключовим технічним рішенням стало використання генератора статичних сайтів Eleventy як основи для реалізації архітектури. Завдяки його стабільності, гнучкості та сумісності з іншими сучасними інструментами, Eleventy надав надійну платформу для впровадження оптимізацій, спрямованих на підвищення швидкодії системи.

У підсумку, було розроблено концепцію архітектури, змодульовано ключові компоненти та обґрунтовано вибір технологічних рішень, що стали основою для побудови продуктивної, масштабованої та адаптивної системи, яка відповідає сучасним вимогам до веброзробки.

3 РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ БІБЛІОТЕКИ ДЛЯ СТВОРЕННЯ ВЕБЗАСТОСУНКІВ НА ОСНОВІ АРХІТЕКРУТИ ОСТРОВІВ

3.1 Варіантний аналіз і обґрунтування вибору засобів для реалізації бібліотеки

Для реалізації острівної архітектури, яка ставить на меті розподіл інтерфейсу на незалежні функціональні блоки, було проведено аналіз можливих мов програмування та фреймворків. Враховуючи вимоги до гнучкості, продуктивності та зручності розробки, розглядалися такі мови: JavaScript, Python, C#, Java та C++. Кожна з цих мов має свої переваги й недоліки, в контексті реалізації проєкту, які вплинули на остаточний вибір технології для реалізації даної архітектури.

JavaScript є основною мовою для розробки фронтенд-частини вебдодатків[26]. Основна перевага JavaScript полягає у його природній підтримці браузерами, що дозволяє виконувати код безпосередньо на стороні клієнта. Ця мова має широку екосистему бібліотек і фреймворків, таких як React, Vue.js, Svelte, що дають змогу створювати динамічні інтерфейси, легко розділяти їх на компоненти та забезпечувати їх взаємодію, також за рахунок того що існує серверне середовище виконання Node.js, можна в одному проекті писати на одній мові.

До основних переваг JavaScript можна віднести:

- широку підтримку браузерами – єдиний стандарт для клієнтської частини;
- динамічність, що полягає у можливості легкого створювання інтерактивних елементів й адаптування їх під вимоги користувачів;
- велику екосистему – доступ до безлічі бібліотек і фреймворків, які спрощують розробку.

Серед недоліків JavaScript можна виділити:

- асинхронність і складність, що полягають у тому, що розуміння та управління асинхронними операціями можуть бути складним для новачків;
- відсутність сильної типізації, що може ускладнювати підтримку великих проєктів.

Python є універсальною мовою програмування, яка активно використовується для бекенд-розробки, автоматизації процесів та аналізу даних [27]. Для розробки вебдодатків використовуються фреймворки, такі як Django або Flask, які забезпечують зручний серверний рендеринг. Однак у контексті острівної архітектури Python має обмеження, оскільки не може безпосередньо виконуватися в браузері і вимагає додаткових інструментів для інтеграції з клієнтською частиною.

До основних переваг Python можна віднести:

- простоту і читабельність коду, що ідеально підходить для швидкого прототипування.
- широкий спектр застосувань, що можна використовувати для аналітики, машинного навчання, бекенд-розробки.

Серед недоліків Python можна виділити:

- відсутність підтримки на стороні клієнта, неможливість виконання в браузері обмежує його використання для клієнтської частини;
- низьку продуктивність у порівнянні з іншими мовами, Python має нижчу швидкодію, особливо для багатокористувацьких додатків.

C# – мова програмування, створена компанією Microsoft, яка часто використовується для розробки десктопних, мобільних та вебдодатків у рамках екосистеми Microsoft [28]. Використовуючи платформу Blazor, можна

створювати вебдодатки, що виконуються в браузері за допомогою WebAssembly. Це забезпечує можливість використання C# як альтернативи JavaScript для клієнтської сторони.

До основних переваг C# можна віднести:

- сильну типізація, яка допомагає уникнути багатьох помилок на етапі розробки.
- підтримку великих проєктів, добре інтегрується в корпоративні середовища, має розвинену підтримку і зручні інструменти.

Серед недоліків C# можна виділити:

- залежність від екосистеми Microsoft, яка обмежує вибір хостингів і платформ.
- підвищені вимоги до обчислювальних ресурсів; Blazor має великі накладні витрати, що може впливати на продуктивність.

Java є однією з найпоширеніших мов програмування, яка використовується як для бекенд-розробки, так і для створення великих корпоративних вебдодатків [29]. Вона пропонує високий рівень безпеки та продуктивності, а також добре підходить для створення надійних серверних рішень. Однак у контексті острівної архітектури Java має певні обмеження через складність інтеграції з фронтендом.

До основних переваг Java можна віднести:

- стабільність та продуктивність, високу продуктивність для розподілених систем і серверних обчислень;
- сильну типізацію, яка забезпечує надійність і простоту підтримки коду.

Серед недоліків Java можна виділити:

- складність у використанні на стороні клієнта, Java не є нативною для браузерів, що ускладнює інтеграцію з клієнтською частиною;

- трудомісткість у налаштуванні, потребує більше зусиль для конфігурації та підтримки порівняно з JavaScript.

C++ – це мова програмування, відома своєю високою продуктивністю і широким спектром застосувань у системному програмуванні, розробці ігор та високопродуктивних систем [30]. Однак для розробки вебдодатків C++ не є оптимальним вибором, оскільки потребує складної інтеграції з іншими технологіями для роботи в браузері.

До основних переваг C++ можна віднести:

- високу продуктивність, дозволяє створювати надзвичайно швидкі додатки;
- гнучкість, можливість роботи на низькому рівні з пам’яттю та апаратним забезпеченням.

Серед недоліків C++ можна виділити:

- складність розробки, C++ вимагає більше зусиль для написання та підтримки коду, особливо в контексті вебдодатків;
- непридатність для клієнтської сторони (як і Python, C++ не підтримується браузерами, що робить його малопридатним для реалізації острівної архітектури).

Існує гібридний підхід – це реалізація на сервері для однієї з мов і для клієнтської частини на Javascript, компіляція через WASM або реалізація на Javascript.

Зважаючи на всі проаналізовані варіанти, JavaScript було обрано як основну мову для реалізації острівної архітектури. JavaScript забезпечує можливість виконання коду безпосередньо в браузері, має широку екосистему інструментів і фреймворків, а також забезпечує високу інтерактивність і динамічність вебдодатків. JavaScript підтримується всіма браузерами, що робить

його єдиним реальним стандартом для клієнтської частини вебдодатків і дозволяє забезпечити незалежність кожного "острова", швидкодію і гнучкість.

Обґрунтуємо вибір реалізації острівної архітектури.

Після визначення мови програмування було проведено аналіз можливих фреймворків і бібліотек для реалізації острівної архітектури. Основними кандидатами стали Astro, Qwik, Next.js та Eleventy. Кожен з цих підходів має свої переваги та недоліки, які були враховані при прийнятті рішення.

Astro є інноваційною платформою, яка дозволяє створювати компоненти, що рендеряться на сервері з мінімальним JavaScript на клієнті. Це забезпечує високу швидкість завантаження сторінок, проте має певні обмеження у роботі з інтерактивними елементами.

Qwik пропонує концепцію миттєвої гідрації, що дозволяє завантажувати мінімальний JavaScript і робити додаток інтерактивним лише тоді, коли це необхідно. Це значно зменшує обсяг завантажуваних даних, проте Qwik має певні виклики з інтеграцією та навчанням розробників.

Next.js є популярним фреймворком, що використовує React і пропонує можливості серверного рендерингу. Хоча Next.js є універсальним і потужним, його використання передбачає великий рантайм, що збільшує розмір додатка і негативно впливає на продуктивність, що суперечить концепції острівної архітектури.

Eleventy — це генератор статичних сайтів, який відзначається простотою, гнучкістю та швидкодією. Використання Eleventy забезпечує повний контроль над рендерингом без накладних витрат на додатковий рантайм, що робить його чудовим вибором для реалізації концепції "островів". Eleventy також легко інтегрується з сучасними технологіями, забезпечуючи підтримку компонентів та інструментів для побудови ізольованих функціональних блоків.

Результати порівняння розглянутих мов програмування за обраними критеріями наведені у таблиці 3.1.

Таблиця 3.1 – Порівняльні характеристики мобільних додатків

Фреймворк / Критерій	Astro	Qwik	Next.js	Elevenity
Простота використання	+/-	-	+	+
Швидкодія	+	+	+/-	+
Гнучкість налаштувань	-	+/-	+/-	+
Розмір рантайму	+	+/-	-	+
Підтримка острівної архітектури	+	+	-	+
Підсумковий результат	3.5	3	2.5	5

Таким чином, було вирішено використовувати Eleventy для реалізації базової інфраструктури проєкту та створити бібліотеку, яка забезпечить реалізацію острівної архітектури. Eleventy став найкращим вибором завдяки своїй простоті, високій продуктивності, відсутності накладних витрат на рантайм і гнучкості в налаштуваннях. Це дозволило максимально ефективно інтегрувати острівну архітектуру та реалізувати всі необхідні покращення без значних витрат ресурсів.

3.2 Розробка програмного модуля острову

Проєкт, спрямований на реалізацію "островів" у контексті острівної архітектури для вебдодатків на базі Eleventy, був побудований із дотриманням

певних концептуальних принципів, орієнтованих на підвищення ефективності, продуктивності та гнучкості системи. Основне завдання полягало у створенні незалежних, ізольованих компонентів користувацького інтерфейсу, які можуть функціонувати автономно та взаємодіяти з іншими елементами системи тільки тоді, коли це необхідно. Такий підхід дозволив досягти значного підвищення продуктивності системи, скоротити час завантаження сторінок і забезпечити оптимальне управління ресурсами.

Одним із ключових принципів є принцип автономності та ізоляції компонентів. Кожен компонент інтерфейсу, або так званий "острів", був реалізований як окремий модуль із повністю ізольованою логікою, стилями та залежностями. Це означає, що кожен компонент може працювати незалежно від інших, що спрощує його розробку, тестування та підтримку. Ізольованість забезпечує мінімальну взаємодію між компонентами, а будь-які зміни в одному з них не впливають на решту системи. Такий підхід сприяє підвищенню стабільності додатка та полегшує його масштабування та оновлення. Завдяки цьому кожен "острів" може бути розроблений, замінений або оновлений без необхідності втручання у функціонування інших частин системи, що забезпечує високий рівень гнучкості та адаптивності.

Іншим важливим аспектом проєктування архітектури є впровадження принципу відкладеного завантаження компонентів. Відкладене завантаження, або lazy-loading, дозволяє активувати компонент лише тоді, коли він стає необхідним для користувача, наприклад, коли певний елемент з'являється у видимій області екрана або коли користувач починає з ним взаємодіяти. Використання цього принципу значно зменшує початковий обсяг ресурсів, які необхідно завантажити при першому завантаженні сторінки, що суттєво покращує продуктивність і зменшує час завантаження. Для реалізації цієї концепції використовуються механізми, такі як Intersection Observer, що дозволяють визначати, коли компонент стає видимим, і активувати його у цей момент. Таким чином, ресурси використовуються більш ефективно,

забезпечуючи високий рівень продуктивності та оптимальне завантаження інтерфейсу.

Гідрація компонентів на основі потреб є ще одним важливим елементом реалізації острівної архітектури. Гідрація передбачає процес перетворення статичного контенту, згенерованого на сервері, в інтерактивний компонент на стороні клієнта. Це дозволяє користувачам отримувати повноцінне відображення сторінки ще до того, як динамічна логіка буде повністю завантажена. Завдяки такому підходу досягається швидке перше відображення сторінки (First Contentful Paint), що позитивно впливає на користувацький досвід. Гідрація компонентів проводиться у моменти, коли це необхідно, наприклад, коли користувач починає взаємодіяти з компонентом, або на основі інших умов, таких як стан мережі чи енергозбереження. Цей підхід забезпечує динамічне управління компонентами, уникає зайвого завантаження та підвищує реактивність системи навіть при великих навантаженнях.

На рисунку 3.1 зображена блок-схема, що демонструє процес гідрації компонента в острівній архітектурі, починаючи від ініціації до завершення оновлення даних. Процес розпочинається зі зчитування умов гідрації для конкретного компонента. Далі перевіряється наявність батьківського елемента, оскільки, якщо такий елемент існує, необхідно додати умову очікування готовності батьківського компонента перед початком гідрації поточного елемента. Після цього відбувається перевірка виконання всіх умов, які були задані для компонентів. Якщо всі умови виконані, надається завдання воркеру для обробки рендеру компонента. На цьому етапі воркер виконує необхідні обчислення або дії, пов'язані з підготовкою компонента до інтерактивного стану. Після рендеру компонента від воркера очікується повідомлення про завершення задачі. У разі отримання цього повідомлення здійснюється оновлення даних компонента, і компонент стає інтерактивним, що завершує процес гідрації. Такий підхід дозволяє ефективно розподіляти обчислювальні ресурси та

зменшувати навантаження на систему, забезпечуючи плавну і поетапну гідрацію кожного компонента в острівній архітектурі.

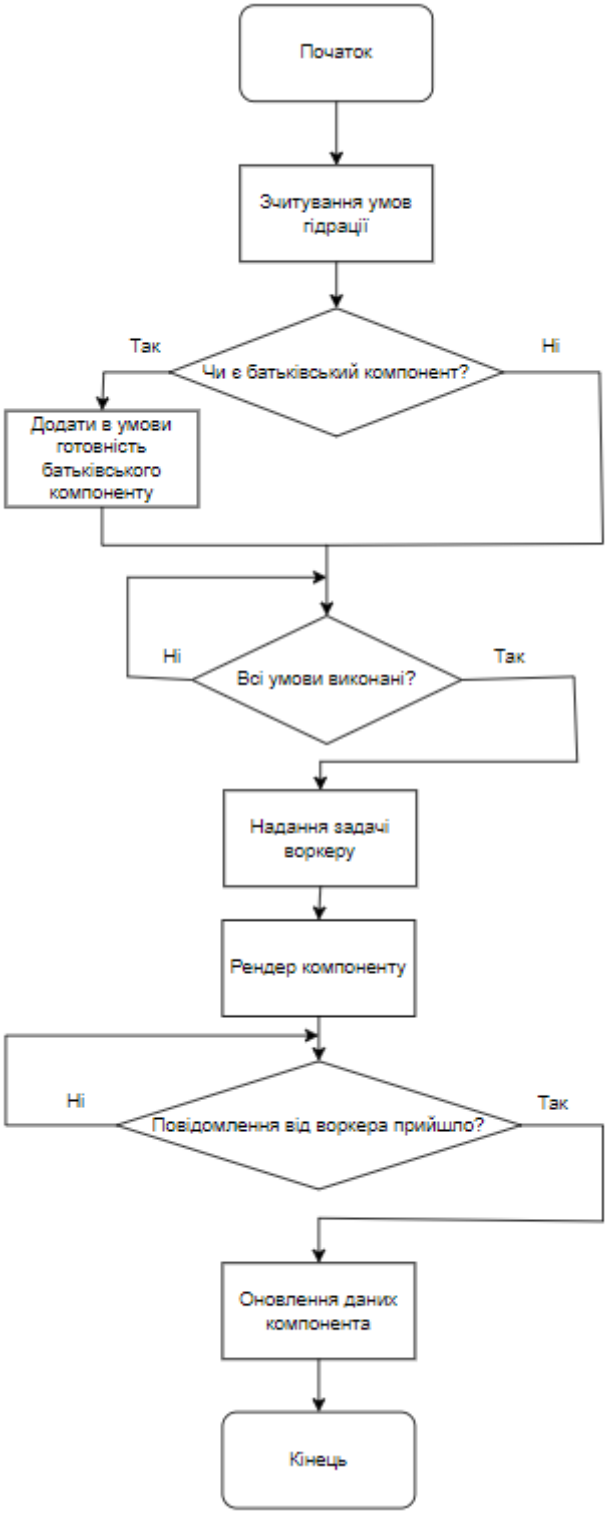


Рисунок 3.1 – Блок-схема процесу гідрації

Також був впроваджений підхід, заснований на серверному рендерингу компонентів із подальшою активацією на клієнтській стороні. Серверний рендеринг дозволяє формувати HTML-код ще на сервері, що забезпечує максимально швидке початкове відображення сторінки у браузері користувача. Завдяки цьому користувач отримує доступ до контенту одразу, не очікуючи, доки клієнтська логіка буде повністю завантажена та оброблена. Це значно скорочує час, необхідний для взаємодії з основним контентом, і підвищує продуктивність системи. Подальша гідрація на стороні клієнта додає необхідну інтерактивність, дозволяючи "островам" реагувати на взаємодію користувача в реальному часі. Такий підхід забезпечує високий рівень інтерактивності, а також баланс між швидкодією та функціональністю системи.

Використання зазначених концептуальних принципів — автономності, відкладеного завантаження, динамічної гідрації та серверного рендерингу — дозволяє створити ефективну архітектуру, у якій кожен "острів" функціонує незалежно, завантажується лише тоді, коли це потрібно, і забезпечує високий рівень інтерактивності без зайвого навантаження на браузер. Це дозволяє не лише підвищити загальну продуктивність додатка, але й зробити його гнучким, легким у підтримці та масштабуванні. Дотримання принципів оптимізації ресурсів, ізоляції компонентів і гнучкого управління контентом дозволяє створити вебдодаток, здатний швидко реагувати на зміну потреб користувачів і забезпечити стабільну роботу навіть у великих проєктах. Завдяки цим рішенням створена архітектура є надійною базою для майбутніх удосконалень і легко адаптується до нових викликів веброзробки.

3.3 Розробка адаптера

У даному розділі висвітлюється процес розробки адаптерів для інтеграції популярних фреймворків у контекст острівної архітектури на базі Eleventy. Зокрема, увага зосереджена на адаптації таких фреймворків, як Svelte,

Vue та Preact, з метою підвищення гнучкості, продуктивності та забезпечення незалежності компонентів вебдодатку. Основним завданням розробки адаптерів є забезпечення можливості створення автономних модулів ("островів") для різних частин інтерфейсу, які можуть взаємодіяти з системою лише тоді, коли це необхідно. Такий підхід сприяє підвищенню масштабованості, зменшенню часу завантаження та оптимальному управлінню ресурсами.

Eleventy надає інтерфейс для роботи з адаптерами, на діаграмі класів (рис 3.2) можна побачити інтерфейс EleventyAdapterInterface, на основі цього інтерфейсу платформа може рендерити різні типи фреймворків, в бібліотеці були реалізовані SvelteAdapter, VueAdapter, PreactAdapter.

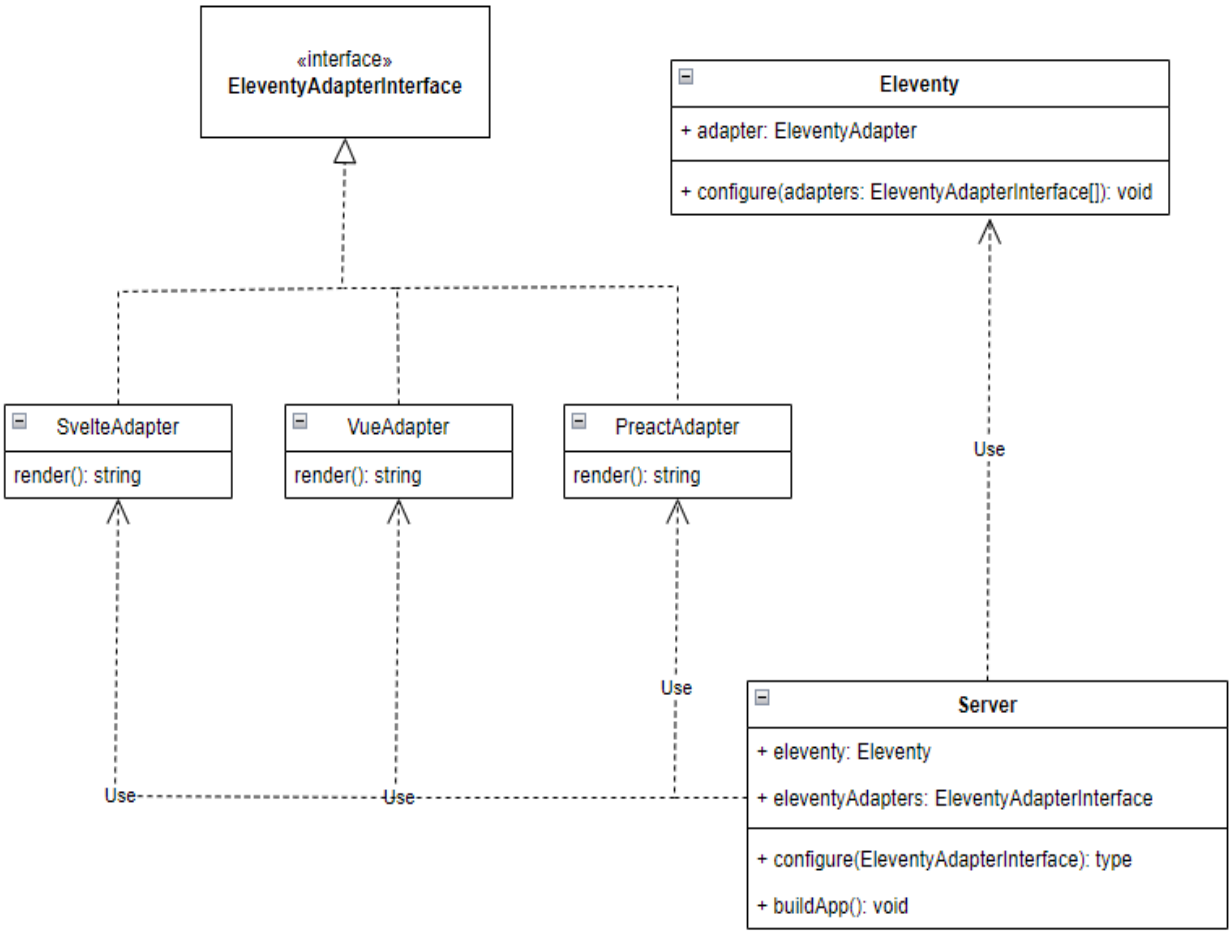


Рисунок 3.2 – Діаграма класів роботи з адаптером

Розробка адаптерів ґрунтувалася на кількох ключових принципах, таких як забезпечення незалежності компонентів, зниження обчислювальних витрат шляхом застосування серверного рендерингу, а також реалізація гнучкої гідрації компонентів на клієнтській стороні. Усі адаптери були реалізовані з урахуванням специфіки фреймворків, для яких вони розроблялися, з метою максимально ефективного використання переваг кожного з них у межах острівної архітектури. Це дозволяє легко інтегрувати їх з Eleventy та забезпечувати незалежне функціонування компонентів, зводячи взаємодію між ними до мінімуму. Проте користувач без проблем може реалізувати свій адаптер, більшість фреймворків мають вхідну точку render, і зачасту потрібно буде просто її використати в адаптері.

У випадку Preact, адаптер реалізовано з використанням серверного рендерингу для генерування HTML, що потім інтегрується з Eleventy. Процес включає читання файлу з компонентом, підготовку необхідних залежностей для рендерингу, безпосередній рендеринг компонента у форматі HTML, а також генерацію URL для клієнтських JavaScript-файлів, що забезпечують гідрацію на стороні користувача. Завдяки цьому підходу досягається швидке початкове відображення сторінки, що покращує продуктивність додатку, тоді як JavaScript-код, який завантажується клієнтом, додає необхідну інтерактивність.

Адаптер для Svelte підтримує як серверний рендеринг, так і гідрацію на клієнтській стороні, що дозволяє створювати високопродуктивні інтерактивні компоненти. Компіляція компонентів реалізована у двох режимах — серверний рендеринг для створення HTML-сторінок, що забезпечує швидке перше відображення контенту, та клієнтське виконання, яке додає можливість інтерактивності. Згенеровані компоненти забезпечують необхідну гідрацію при взаємодії з користувачем, що сприяє ефективному управлінню ресурсами. Окрім цього, використовується функція управління CSS, що дозволяє коректно підключати стилі до кожної сторінки, забезпечуючи ізоляцію компонентів. На рисунку 3.3 можна побачити що через те що фреймворк використовує

компілятор, перед тим як відправити компонент, його необхідно скопмілювати, тому в такому незвичному випадку потрібно знати особливості кожного фреймворка.

```
1 usage
generateClientBundle() : FileTarget  {
  let options : {...}  = {
    filename: this.filename,
    generate: "dom",
    hydratable: this.isSSR ? true : false,
    css: this.isSSR ? false : true,
  };

  let component : {...}  = svelte.compile(this.content, options);

  let target : FileTarget  = new FileTarget(this.filename);
  target.write(component.js.code, this.clientImportMap);
  return target;
}
```

Рисунок 3.3 – Фрагмент коду генерації клієнтського бандла

Vue адаптер побудований на основі серверного рендерингу з використанням Vue SSR (server-side rendering), що дозволяє швидко генерувати HTML для початкового відображення сторінок. Це зменшує час завантаження та забезпечує оптимальне відображення сторінки для користувача ще до того, як клієнтська логіка буде повністю завантажена. Подальша інтерактивність досягається за допомогою гідрації, яка забезпечується клієнтськими скриптами. Таким чином, досягається баланс між швидким серверним рендерингом та динамічною інтерактивністю, необхідною для сучасних вебдодатків.

Розробка адаптерів для Eleventy передбачала реалізацію кількох основних принципів, які забезпечують ефективну інтеграцію популярних фреймворків і автономну роботу кожного компонента. По-перше, здійснювалося розділення рендерингу на серверний та клієнтський. Серверний рендеринг забезпечував швидке початкове відображення сторінок, що було важливим для зниження часу завантаження. Після цього на клієнтській стороні проводилася гідрація, яка додавала інтерактивність. Такий підхід дозволяє забезпечити продуктивність та швидке реагування додатку.

Другим важливим принципом була гнучка гідрація компонентів. Вона реалізовувалася лише тоді, коли це вимагалося поточною ситуацією, наприклад, коли компонент ставав видимим або коли користувач починав взаємодію з ним. Це зменшувало початковий обсяг завантажуваних ресурсів і сприяло покращенню швидкодії системи. Ізольованість компонентів також мала вирішальне значення, оскільки дозволяла уникнути небажаної взаємодії між різними елементами додатку, забезпечуючи таким чином стабільність і надійність роботи системи.

Забезпечення клієнтських залежностей було ще одним важливим аспектом при створенні адаптерів. Адаптери мали генерувати URL-адреси для імпорту JavaScript-бібліотек, необхідних для гідрації на стороні клієнта. Це гарантувало, що компоненти мали доступ до необхідних скриптів для виконання інтерактивних функцій. Крім того, під час компіляції компонентів для серверного та клієнтського використання забезпечувалася оптимізація згенерованого коду, що сприяло підвищенню продуктивності.

Для створення власних адаптерів можуть знадобитися такі компоненти та етапи. По-перше, необхідно глибоко розуміти особливості фреймворку, для якого створюється адаптер, включаючи механізми рендерингу та гідрації. Також важливо забезпечити наявність інструментів для серверного рендерингу, що дозволяє створювати HTML-код для першого відображення сторінки, та

реалізувати засоби для генерації клієнтських скриптів, які будуть використовуватися для гідрації. Крім того, важливо передбачити обробку CSS для кожного компонента, що гарантує ізоляцію стилів і правильне відображення на кожній сторінці. Нарешті, адаптери мають бути розроблені з можливістю асинхронної гідрації, що дозволяє здійснювати її лише після виконання необхідних умов.

Таким чином, розробка адаптерів для Svelte, Vue та Preact дозволила інтегрувати ці фреймворки в острівну архітектуру на базі Eleventy, що забезпечує високу продуктивність, гнучкість та незалежність компонентів. Основні принципи розробки адаптерів зосереджувалися на ізоляції компонентів, серверному рендерингу та асинхронній гідрації, що забезпечило ефективне використання ресурсів і високий рівень взаємодії з користувачем. Створена архітектура дозволяє зберегти модульність системи, забезпечуючи легке розширення та підтримку компонентів, а також зберігає гнучкість і масштабованість для майбутніх потреб веброзробки.

3.4 Розробка файлу налаштувань для інтеграції з Elevenity

Для розробник проекту був використаний збірних статичних сайтів Evelevnity, для коректної роботи проекту в результаті розробки був створений шаблон, на основі якого можуть створюватись проекти.

Розробка даного конфігураційного файлу для Eleventy (рисунок 3.4) передбачала автоматизацію процесу налаштування проєкту, включаючи автоматичний імпорт плагінів, копіювання необхідних файлів і визначення структури проєкту. Основною метою було забезпечення гнучкості та зручності під час розгортання середовища для роботи з Eleventy, мінімізуючи ручне втручання розробника.

```
const fs :  = require("fs");
const path :PlatformPath | path  = require("path");

module.exports = function(eleventyConfig) :{…}  {
    eleventyConfig.setQuietMode(true);
    const pluginsDir :string  = path.join(__dirname, "11ty");
    fs.readdirSync(pluginsDir).forEach(file :string  => {
        if (file.endsWith(".plugin.cjs")) {
            const pluginPath :string  = path.join(pluginsDir, file);
            const plugin = require(pluginPath);
            eleventyConfig.addPlugin(plugin);
        }
    });
    eleventyConfig.addPassthroughCopy( fileOrDir: "lib/**/*.{css,png,svg,js}");
    eleventyConfig.addPassthroughCopy( fileOrDir: "/**/*.{css,js}");
    eleventyConfig.addPassthroughCopy( fileOrDir: {
        "**/*.worker.js": "workers/"
    });
    eleventyConfig.setServerOptions( options: {
        domdiff: false,
    });
    eleventyConfig.ignores.add("README.md");
    eleventyConfig.addGlobalData( name: "permalink",  data: () => {
        return (data) :string  => `${data.page.filePathStem}.${data.page.outputFileExtension}`;
    });

    return {
        dir: {
            input: "index",
            output: "_site",
            includes: "_includes",
            data: "_data",
        },
        htmlTemplateEngine: "liquid",
        markdownTemplateEngine: "liquid",
    };
};
```

Рисунок 3.4 – Створення файлу налаштувань

Для реалізації автоматичного імпорту плагінів використано підхід, що передбачає динамічне завантаження всіх плагінів із папки 11ty, файли яких мають розширення .plugin.cjs. Це рішення дозволяє уникнути необхідності імпортувати кожен плагін окремо, що значно скорочує час налаштування та

зменшує ймовірність помилок при імпорті. Завдяки використанню модуля fs здійснюється сканування вмісту папки, що спрощує процес підтримки та оновлення плагінів.

Додатково було розроблено механізм автоматичного перенесення воркерів. Усі файли з розширенням .worker.js автоматично копіюються до спеціальної папки workers/ під час збірки проєкту. Це дозволяє чітко організувати структуру вихідних файлів і полегшує роботу з ними в процесі розробки.

Файл також містить налаштування для копіювання різних статичних ресурсів, таких як CSS, JS, PNG, SVG, з визначених директорій у фінальну структуру збірки. Це забезпечує правильне розміщення статичних файлів для їх використання у вебдодатку.

Ще одним ключовим аспектом стало налаштування конфігурації сервера, де було визначено параметри для відключення механізму DOM-дифінгу (domdiff), що може бути необхідно для зменшення навантаження на сервер у специфічних випадках.

Налаштування включають ігнорування деяких файлів, таких як README.md, що є типовою практикою для запобігання випадковому включенню файлів документації у фінальну збірку. Крім того, було додано глобальне налаштування для структури посилань, що автоматично визначає формат URL-адрес для кожної сторінки.

У розробленому файлі було передбачено кілька допущень, що стосуються розміщення різних типів файлів у відповідних директоріях, що дозволяє оптимізувати структуру проєкту та забезпечити легкість його підтримки. Було розроблено структуру директорій, яка включає вхідну папку demo, вихідну папку _site, а також спеціальні папки для включень (_includes) та даних (_data), що спрощує організацію проєкту та підвищує його масштабованість.

Таким чином, файл збірки забезпечує автоматизацію основних процесів налаштування проєкту Eleventy, що полегшує підтримку та розширення проєкту в майбутньому, дозволяючи зосередитись на безпосередньо розробці функціональності, а не на ручному налаштуванні середовища.

3.5 Розробка системи кешування

Розроблена система кешування з використанням Service Worker володіє низкою переваг і особливостей, які роблять її ефективною для сучасних вебдодатків, що потребують стабільної роботи навіть за відсутності підключення до мережі. Система базується на розділенні кешу на статичний і динамічний, що сприяє оптимізації зберігання ресурсів і підвищенню загальної продуктивності.

На рисунку 3.5 можна побачити фрагмент коду, що отримує дані з кеша у випадку статичних даних, таким чином відбувається оптимізація, і файли завантажуються з кеша, а не з окремого запиту, забезпечуючи підтримку оффлайн режиму.

```
self.addEventListener( type: 'fetch', listener: (event : Event ) : void => {
    const url : URL = new URL(event.request.url);
    if (staticAssets.includes(url.pathname) || url.pathname.endsWith('.html') || url.pathname.endsWith('.css') || url.pathname.endsWith('.js')) {
        event.respondWith(
            caches.match(event.request).then((cachedResponse : Response | undefined ) : Response | Promise<...> => {
                if (cachedResponse) {
                    return cachedResponse;
                }
                return fetch(event.request).then((networkResponse : Response ) : Promise<Response> => {
                    return caches.open(DYNAMIC_CACHE_NAME).then((cache : Cache ) : Response => {
                        cache.put(event.request, networkResponse.clone());
                        return networkResponse;
                    });
                }).catch(() : Promise<...> | undefined => {
                    if (event.request.headers.get('accept').includes('text/html')) {
                        return caches.match( request: '/offline.html');
                    }
                });
            })
        );
    } else {
        event.respondWith(
            fetch(event.request).catch(() : Promise<...> => caches.match( request: '/offline.html'))
        );
    }
});
```

Рисунок 3.5 – Фрагемнт коду діставання з кешу

Однією з ключових особливостей даного рішення є використання двох окремих кешів: статичного (STATIC_CACHE_NAME) і динамічного (DYNAMIC_CACHE_NAME). Статичний кеш зберігає критично важливі ресурси, зокрема HTML, CSS, JavaScript і інші файли, що потрібні для початкового завантаження додатку. Завдяки цьому користувачі отримують можливість швидко завантажувати основний контент навіть без доступу до мережі, що підвищує доступність вебдодатку.

Динамічний кеш, натомість, зберігає ресурси, що завантажуються під час роботи додатку. Ці ресурси можуть не входити до початкового набору файлів, але бути важливими для подальшого використання. Використання динамічного кешу дозволяє значно знизити навантаження на сервер, оскільки частина даних зберігається локально та повторно використовується при наступних запитах.

На рисунку 3.6 показаний процес подій що відбувається під час завантаження сторінки.

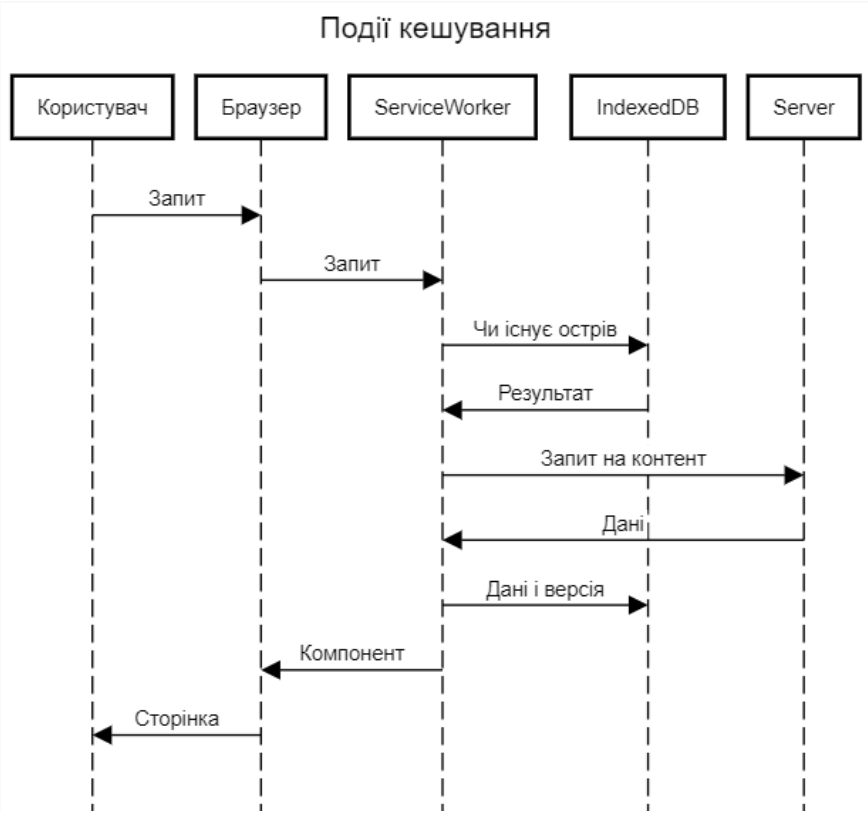


Рисунок 3.6 – Послідовність подій кешування

Клієнт робить запит, після чого цей запит браузер передає сервіс воркеру, який відповідає за видачу контенту, сервіс воркер робить запит у базу даних, де зберігається потенційний статичний компонент, якщо він там наявний він віддається, якщо ж його немає, або версія не співпадає, у випадку як показано на діаграмі, робиться запит серверу на отримання нового контенту, та збереження його в базу, після чого дані віддаються користувачу у вигляді повноцінної сторінки. Важливо помітити що версія застосунку задається розробником, і коли сайт оновлюється розробник має вручну оновити версію сайта, або використовувати автоматизовані засоби для версіонування.

Стратегія "кешування при першому завантаженні" (Cache First) для статичних ресурсів є ще однією важливою перевагою. Згідно з нею, при запиті ресурсу Service Worker спочатку звертається до кешу, і якщо ресурс уже збережений, він одразу надається користувачу, що суттєво скорочує час завантаження. Якщо ж ресурсу немає в кеші, здійснюється запит до мережі, а отриманий ресурс додається до динамічного кешу для подальшого використання. Це забезпечує баланс між продуктивністю та гнучкістю системи.

Також важливою є реалізація механізму очищення старих кешів під час активації нового Service Worker. Очищення старих даних дозволяє уникнути накопичення застарілої інформації і забезпечує, що користувачі отримують найактуальніші версії файлів. Це досягається шляхом порівняння імені поточного кешу із заздалегідь визначеними назвами і видаленням тих кешів, що більше не використовуються.

Для покращення користувацького досвіду було реалізовано механізм фолбеку на випадок відсутності мережевого підключення. Якщо користувач запитує HTML-сторінку і з'єднання з мережею відсутнє, Service Worker повертає раніше збережену офлайн-сторінку. Це забезпечує безперервний досвід використання додатку і мінімізує незручності для користувачів у випадку нестабільного інтернету.

3.6 Розробка програмного модуля Web worker

Розробка модуля Web Worker для острівної архітектури мала на меті створення механізму дегідрації, гідрації та кешування даних для ізольованих компонентів — так званих "островів". Основна мета цього підходу полягала в тому, щоб забезпечити оптимальне управління станом кожного компонента, підвищити продуктивність додатку та надати можливість працювати з даними навіть за відсутності підключення до мережі.

Web Worker було обрано для реалізації цих функцій з метою розвантаження основного потоку виконання JavaScript і забезпечення асинхронної обробки запитів, пов'язаних із зберіганням і відновленням даних для кожного з островів. Це дозволяє виконувати складні операції з даними, не впливаючи на швидкодію та інтерфейс користувача.

Одним із ключових аспектів роботи воркера є дегідрація даних. Дегідрація передбачає перетворення стану компонента в компактний формат для подальшого зберігання. Воркера використовували для збереження цих дегідрованих даних у локальному сховищі браузера (localStorage). Це дозволяє зберегти стан кожного острова, щоб у майбутньому можна було відновити його без необхідності повторного запиту до сервера.

Гідрація є процесом відновлення стану компонента на основі збережених раніше даних. У реалізованому воркері використовується логіка, яка зчитує дані з локального сховища і, якщо дані для певного острова існують, вони відправляються назад до основного потоку для гідрації. Це сприяє забезпеченню інтерактивності та збереженню стану компонентів, навіть коли додаток завантажується після перерви.

Кешування даних здійснюється через Cache API. Для кожного острова створюється запит, який зберігається в кеші, що дозволяє швидко отримувати дані без необхідності їх повторного запиту з сервера. Це рішення особливо корисне в умовах обмеженого доступу до інтернету, оскільки дані залишаються

доступними для використання. Зберігання відбувається із забезпеченням контролю над типами даних та їх призначенням, що допомагає підтримувати консистентність даних між різними станами додатку.

На рисунку 3.7 зображено участь WebWorker в процесі гідрації, при запиті зі сторони клієнта і отримані даних через ServiceWorker запускається гідрація компоненту, в окремому потоці, під час гідрації користувачу показується так званий fallback, код який показується поки йде завантаження, після того як веб-воркер згідрував дані він через шину подій направляє повідомлення про те що острів з певним індентифікатором закінчив гідрацію і в повідомленні направляє його дані. Браузер отримує інформацію і відображає вірні згідровані дані.

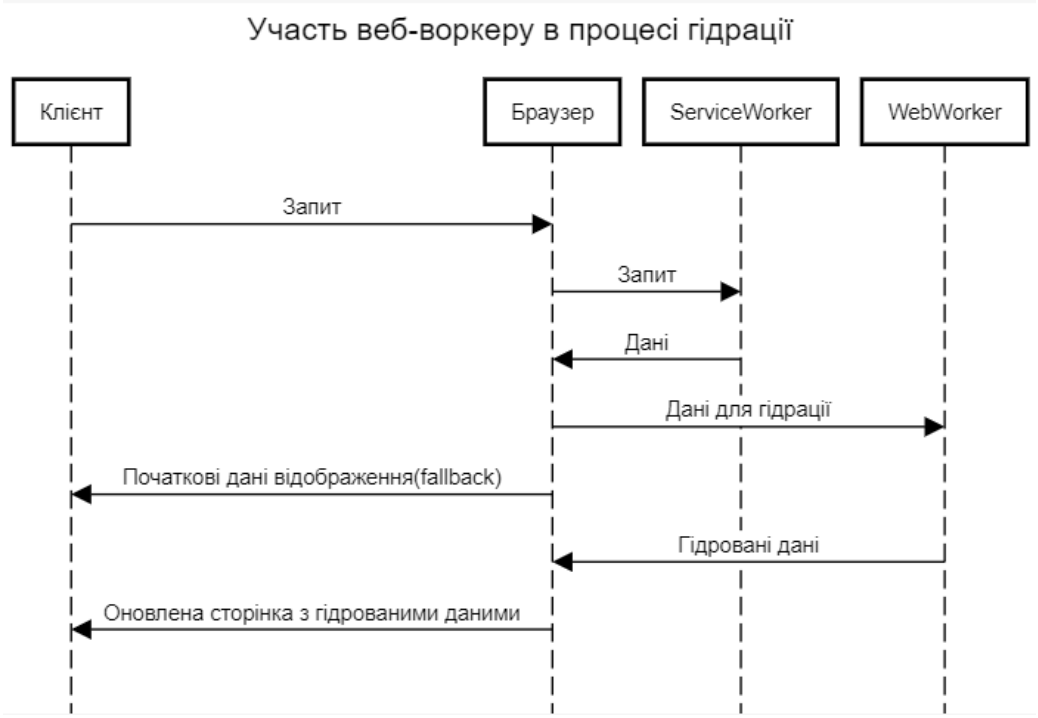


Рисунок 3.7 – Участь вебворкеру в процесі гідрації

Окрім зберігання, воркер також забезпечує очищення кешу для островів, що дозволяє уникати накопичення застарілих або непотрібних даних, тим самим

оптимізуючи використання ресурсів браузера. Під час очищення перевіряється наявність кешу для конкретного острова, і якщо він існує, його видаляють. Це гарантує, що у сховищі залишаються лише актуальні дані, а користувач отримує найновішу інформацію.

Модуль Web Worker забезпечує обробку різних типів запитів (дегідрація, гідрація, кешування, очищення) і виконує їх асинхронно, що дозволяє підвищити загальну продуктивність додатку та покращити користувацький досвід. Завдяки використанню цього підходу вдалося реалізувати більш гнучке управління станом компонентів у межах острівної архітектури, що робить додаток більш стійким і продуктивним, особливо в умовах нестабільного або відсутнього підключення до мережі.

3.7 Висновки

У третьому розділі було обґрунтовано вибір засобів програмування та технологій, використаних для розробки модулів додатку, а також було визначено підхід до роботи з острівною архітектурою. Для реалізації функціоналу було обрано використання JavaScript та Web Worker для асинхронного управління станом компонентів ("островів"). Розроблено декілька ключових модулів, зокрема модуль дегідрації даних компонентів для зберігання стану в локальному сховищі, модуль гідрації для відновлення стану компонента з кешу, а також модуль кешування, що дозволяє зберігати дані у Cache API для оптимізації роботи в режимі офлайн. Всі розроблені модулі спрямовані на забезпечення стабільної роботи вебдодатку, навіть в умовах обмеженого або нестабільного підключення до мережі.

4 ТЕСТУВАННЯ ПРОГРАМНОГО ДОДАТКУ

4.1 Аналіз методів тестування

Тестування острівної архітектури, орієнтованої на ефективну роботу в умовах динамічного середовища, потребує особливого підходу, який передбачає оцінку продуктивності, завантаження та використання ресурсів додатком. Для аналізу ефективності системи було обрано методи тестування, що зосереджені на замірі часу виконання операцій, обчисленні обсягу завантажених даних і моніторингу використання кешу. Особлива увага приділялася дегідрації та гідрації даних островів, а також ефективності роботи кешу в різних умовах використання додатку.

Тестування замірів часу. Для острівної архітектури надзвичайно важливо оцінити час, який витрачається на дегідрацію та гідрацію кожного острова. Для цього застосовувалися інструменти для заміру часу виконання таких операцій, як збереження даних у локальному сховищі, відновлення цих даних і гідрація компонентів на основі отриманих даних. Замір часу є критично важливим для оцінки затримок, які можуть впливати на загальний користувацький досвід.

У процесі тестування було виявлено, що використання Web Worker для дегідрації та гідрації значно знижує навантаження на основний потік виконання JavaScript. Операції, пов'язані зі зберіганням та відновленням даних, відбуваються у фоновому режимі, що дозволяє основному інтерфейсу залишатися інтерактивним і реагувати на дії користувача без затримок.

Тестування обсягу завантажених даних. В острівній архітектурі, де кожен "острів" має власний стан і залежності, важливо контролювати обсяг завантажених даних, щоб уникнути перевантаження мережі та підвищити продуктивність додатку. Тестування було спрямоване на визначення кількості кілобайтів, що завантажуються під час гідрації кожного компонента, а також на оцінку ефективності використання кешу.

Під час тестування було встановлено, що стратегія кешування значно зменшує обсяг завантажених даних. Дані, що раніше були збережені у кеші (Cache API), можуть бути використані повторно, що дозволяє уникнути дублювання запитів до сервера. Це особливо важливо у випадку, коли компонент має складну структуру або вимагає завантаження великих ресурсів. За допомогою аналізу обсягу завантажених даних можна оцінити ефективність кешування і визначити, які саме ресурси можна оптимізувати для зменшення навантаження на мережу.

Тестування використання кешу. Використання Cache API для зберігання даних островів дозволяє забезпечити швидкий доступ до необхідних ресурсів без повторних звернень до серверу. Тестування кешу орієнтоване на оцінку його ефективності, зокрема, чи зберігаються всі необхідні дані, чи вчасно оновлюються, а також на оцінку процесу очищення кешу від застарілих даних.

У ході тестування використовувалася стратегія "кешування при першому завантаженні" (Cache First), що дозволяє спочатку звертатися до кешу і лише за необхідності виконувати запит до сервера. Було виявлено, що такий підхід значно знижує час завантаження додатку, особливо у випадках, коли користувач має нестабільне або повільне інтернет-з'єднання. Очищення кешу під час активації нового Service Worker дозволяє запобігти накопиченню застарілих даних, що покращує продуктивність і знижує використання дискового простору.

Порівняльний аналіз методів тестування. Аналіз наведених методів тестування показав, що кожен із них має свої переваги для оцінки ефективності острівної архітектури. Замір часу виконання операцій дозволяє оцінити швидкість роботи системи та знижувати затримки, які впливають на користувацький досвід. Аналіз обсягу завантажених даних забезпечує контроль за використанням мережевих ресурсів, а тестування використання кешу дозволяє оптимізувати зберігання даних і забезпечити швидкий доступ до них.

Після порівняльної характеристики методів було прийнято рішення використовувати комплексний підхід до тестування, що включає замір часу, аналіз обсягу завантажених даних і оцінку ефективності кешування. Такий підхід дозволяє отримати повне уявлення про продуктивність системи, забезпечує ефективну роботу острівної архітектури навіть в умовах обмежених мережевих ресурсів, а також гарантує стабільну і швидку взаємодію користувача з додатком.

4.2 Тестування сайту, створеного на основі розробки

Для тестування були розроблені 2 версії максимально простого сайту, де продемонстровано переваги підходу, важливо примітити що переваги помітні уже на невеликому масштабі даних, коли даних стає більше новітні підходи будуть ще більш помітними.

Спершу розглянемо завантаження сайту на якому помітні лише статичні дані, динамічних даних користувач поки не бачи, а отже за принципом lazy-loading, вони й не відвантажуються, що дає зекономити кількскість часу для завантаження та кількості біт. На рисунку 4.1 показано граф завантаження, завантажується сторінка, стилі і фреймворк, цього достатньо для того щоб показати корисну для користувача, з графу видно що завантажується html файл, що і є по суті сайтов ,після чого, з невеличкою затримкою у 6мс почав завантажуватись CSS файл, потім завантажилась бібліотека, що необхідно для роботи архітектури, та автоімпортер для роботи з фреймворками. Загалом контент завантажився швидко, за 62мс, при чому найбільше часу зайняло зображення.

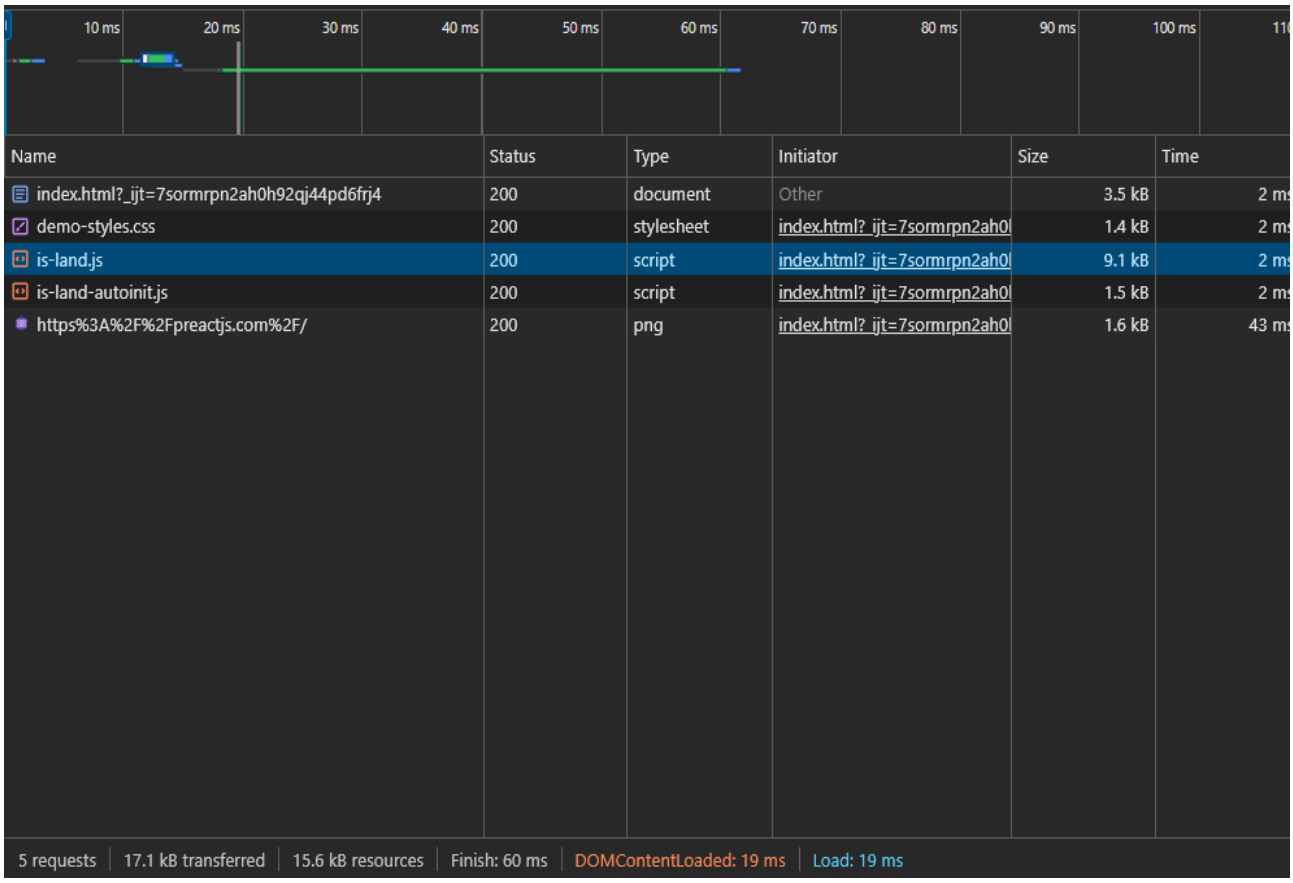


Рисунок 4.1 – Граф завантаження початкової сторінки сайту

У випадку повноцінного використання сайту(рис 4.2), загальний розмір скачаних файлів виріс (через бібліотеку), однак важливо помітити що навантаження для першого відображення контента знизилось, порція додаткового контенту почала завантажуватись лише коли стала потрібна, через 400мс часу, тим самим виходячи вдалось зменшити кількість даних які були потрібні спочатку. Таким чином збільшено ефективність використання ресурсів за зменшено час для First content paint, що є важливою метрикою для виведення сайтів вгору в пошукових запитах та дозволяє покращити користувацький досвід. Користувачі які не долистують до низу сайту отримають тільки необхідний контент. Хоч і загальна кількість часу і загальний розмір бандлу виріс від використання бібліотеки, проте це лише разова плата, у випадку використання багатьох островів це компенсується перевагами lazy-loading, що надає такий підхід.

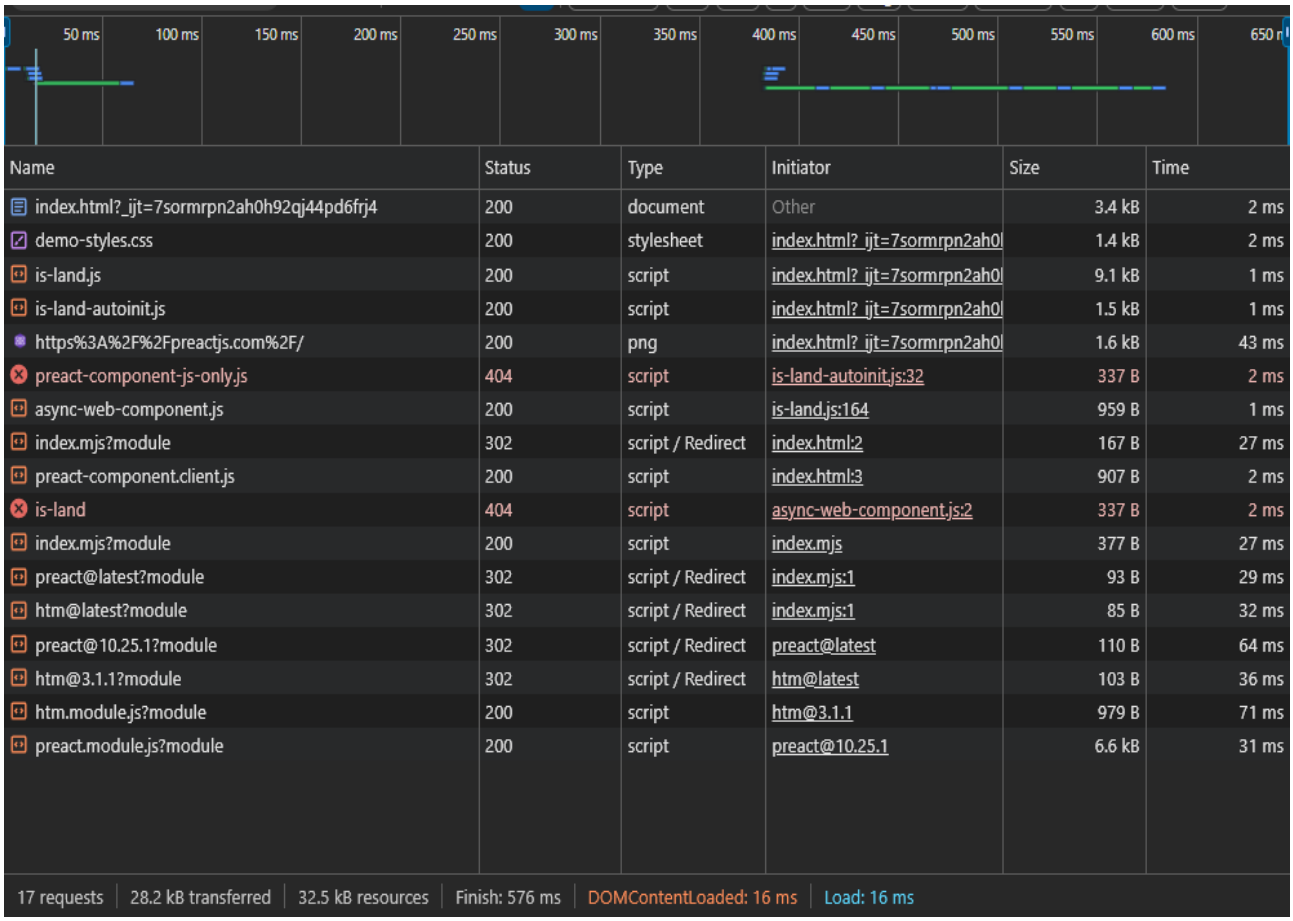
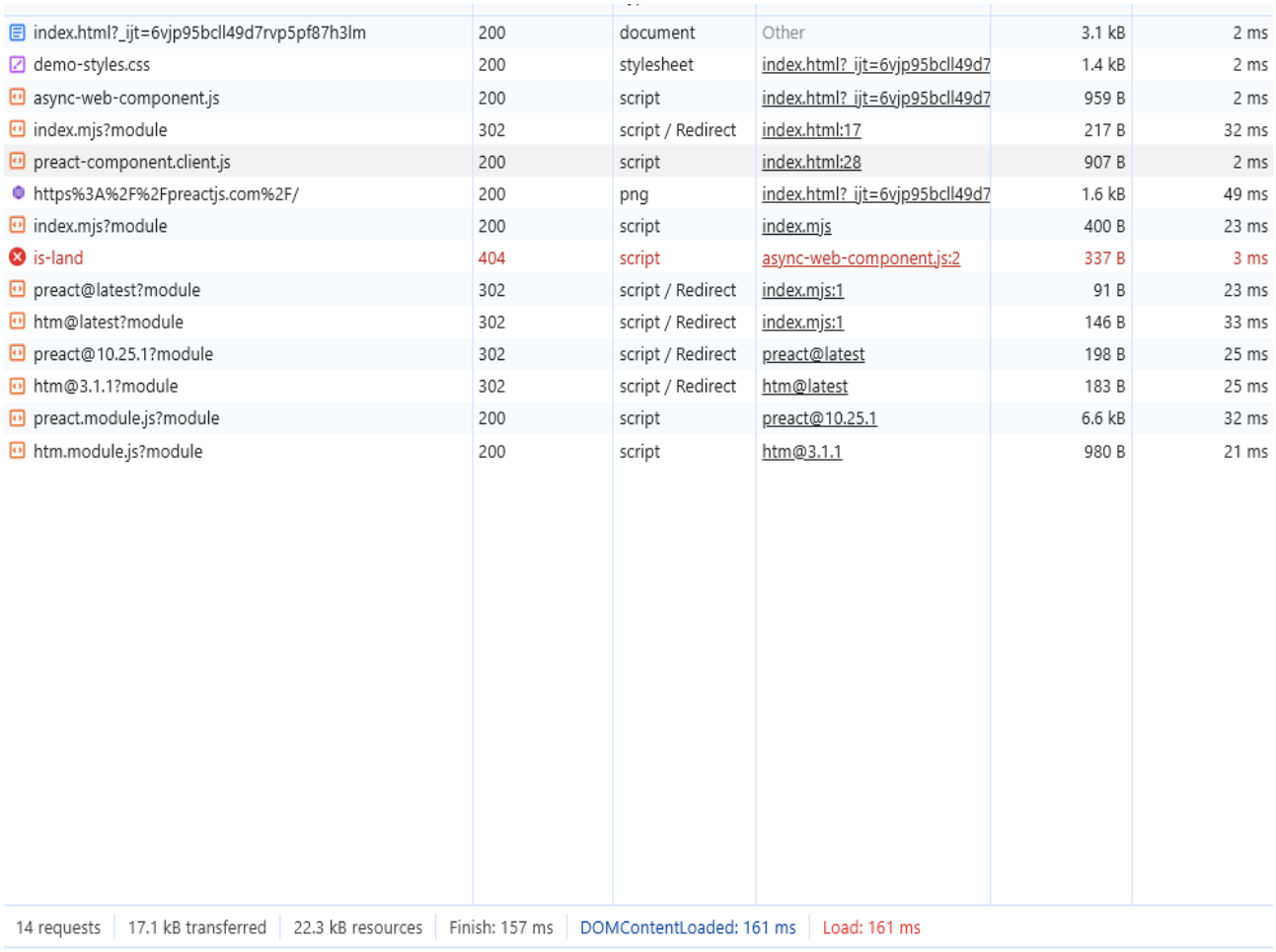


Рисунок 4.2 – Граф завантаження початкової сторінки та завантаження при довантаженні островів

Для прикладу було продемонстровано (рис 4.3) підхід без використання архітектури, де весь контент сторінки завантажується разом з сторінкою, такий підхід має певні переваги, по перше бібліотека не завантажується, а отже і загальний розмір завантаження зменшується на 10 кілобайт, проте у випадку коли контенту сайту більше ніж на один екран, страждають певні метрики сайту, такі як швидкість завантаження, час до First content paint, що може виразитись на метриках сайту та здвинути його популярність нижче по рейтингу пошукових видач. Для великих додатків використання такого підходу може значно збільшити розмір бандлу. Модель завантажень ресурсів є повністю каскадною, і до завантаження всіх ресурсів сайт є неінтерактивним статичним сайтом.



index.html?_ijt=6vjp95bcll49d7rvp5pf87h3lm	200	document	Other	3.1 kB	2 ms
demo-styles.css	200	stylesheet	index.html?_ijt=6vjp95bcll49d7	1.4 kB	2 ms
async-web-component.js	200	script	index.html?_ijt=6vjp95bcll49d7	959 B	2 ms
index.mjs?module	302	script / Redirect	index.html:17	217 B	32 ms
preact-component.client.js	200	script	index.html:28	907 B	2 ms
https%3A%2F%2Fpreactjs.com%2F/	200	png	index.html?_ijt=6vjp95bcll49d7	1.6 kB	49 ms
index.mjs?module	200	script	index.mjs	400 B	23 ms
is-land	404	script	async-web-component.js:2	337 B	3 ms
preact@latest?module	302	script / Redirect	index.mjs:1	91 B	23 ms
htm@latest?module	302	script / Redirect	index.mjs:1	146 B	33 ms
preact@10.25.1?module	302	script / Redirect	preact@latest	198 B	25 ms
htm@3.1.1?module	302	script / Redirect	htm@latest	183 B	25 ms
preact.module.js?module	200	script	preact@10.25.1	6.6 kB	32 ms
htm.module.js?module	200	script	htm@3.1.1	980 B	21 ms

14 requests | 17.1 kB transferred | 22.3 kB resources | Finish: 157 ms | DOMContentLoaded: 161 ms | Load: 161 ms

Рисунок 4.3 – Граф завантаження сторінки без використання острівної архітектури та оптимізацій

Не зважаючи на мінуси зазначеного підходу, варто помітити, що при створенні односторінкових сайтів, в яких весь контент поміщається на девайсі користувача відразу, цей підхід має значні переваги, хоча це і специфічний випадок.

4.3 Розробка інструкції користувача

Інструкція для користувача передбачає ознайомлення з мінімальними технічними характеристиками для встановлення шаблону. Детальний опис характеристик наведений у таблиці 4.1.

Таблиця 4.1 – Мінімальні технічні характеристики

Характеристика	Опис
Операційна система	Windows 10, Windows 11, Linux, OSX
Процесор	Мінімум 1,2 ГГц, двоядерний або вище
Оперативна пам'ять	Мінімум 500 МБ RAM
Вільне місце	Мінімум 500 МБ
Інтерфейси	Wi-Fi, LTE

Для створення додатку на базі шаблону необхідно встановити архів та розпакувати його, після чого встановити необхідні модулі.

У результаті попередніх дій у вас має з’явитись готовий до запуску проєкт, для ознайомлення з базовими випадками використання рекомендується уважно прочитати README.md файл, в якому будуть описані найпоширеніші способи використання та практичні рекомендації щодо імплементації певних речей, що можуть знадобитись розробнику.

Для успішної інтеграції важливо також налаштувати взаємодію між основним потоком додатку та Web Worker. Це передбачає обробку повідомлень, які надходять від воркера, для правильного реагування на події, такі як завершення дегідрації чи гідрації даних. У процесі налаштування воркера рекомендується використовувати механізми обробки помилок, щоб своєчасно виявляти і вирішувати можливі проблеми, що виникають під час взаємодії між різними компонентами додатку.

Також необхідно врахувати, що для роботи з даними, які зберігаються у кеші або локальному сховищі, потрібен контроль за обсягами та актуальністю інформації. Наприклад, варто періодично очищувати кеш від застарілих або непотрібних даних, щоб уникнути накопичення зайвих ресурсів, що можуть вплинути на продуктивність додатку. Розробники повинні також враховувати

можливість створення власних політик кешування та використання сховища для максимізації продуктивності та мінімізації використання мережевих ресурсів.

4.4 Висновки

У четвертому розділі було проведено тестування програмних модулів, що реалізують острівну архітектуру для вебдодатків. Тестування охоплювало перевірку коректної роботи механізмів гідрації компонентів, відкладеного завантаження (lazy-loading), кешування, а також взаємодії між автономними компонентами ("островами"). Було перевірено, як кожен "острів" виконує свої функції незалежно від інших, забезпечуючи високу продуктивність і гнучкість системи.

Тестування продемонструвало правильність роботи розроблених механізмів гідрації, ефективність використання вебворкерів для обробки ресурсомістких задач у фоновому режимі, а також оптимізацію часу завантаження сторінок завдяки попередньому завантаженню критичних ресурсів. Крім того, було підтверджено, що використання Shadow DOM для ізоляції стилів і логіки дозволяє уникнути конфліктів між компонентами, забезпечуючи стабільність роботи кожного модуля.

Розроблено інструкцію для розробників, що описує основні аспекти впровадження та налаштування острівної архітектури з використанням Eleventy. Окремо було описано мінімальні технічні вимоги для серверного оточення, що забезпечують стабільну та безперебійну роботу системи.

Результати тестування підтвердили, що розроблена архітектура відповідає поставленим вимогам щодо продуктивності та гнучкості. Система демонструє стабільну роботу в умовах різного рівня навантаження та забезпечує високий рівень інтерактивності для користувачів без збільшення накладних витрат на виконання додаткових рантайм-бібліотек.

5 ЕКОНОМІЧНА ЧАСТИНА

Науково-технічна розробка має право на існування та впровадження, якщо вона відповідає вимогам часу, як в напрямку науково-технічного прогресу та і в плані економіки. Тому в процесі проведення науково-дослідної роботи важливо проводити оцінку економічної ефективності результатів виконаних досліджень та розробок.

Магістерська кваліфікаційна робота за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» відноситься до науково-технічних робіт, які орієнтовані на виведення на ринок (або рішення про виведення науково-технічної розробки на ринок може бути прийнято у процесі проведення самої роботи), тобто коли відбувається так звана комерціалізація науково-технічної розробки. Комерціалізація науково-технічних розробок є важливим напрямком, який дозволяє забезпечити широке використання результатів розробки іншими споживачами, отримуючи при цьому певний економічний ефект. Необхідним кроком для виконання такої задачі є знаходження потенційного інвестора, який би взявся за реалізацію цього проекту і переконати його в економічній доцільності реалізації проєкту.

Для наведеного випадку мають бути виконані такі етапи робіт:

1) проведено комерційний аудит науково-технічної розробки, тобто встановлення її науково-технічного рівня та комерційного потенціалу;

2) розраховано загальні витрати на здійснення науково-технічної розробки;

3) розрахована економічна ефективність науково-технічної розробки у випадку її впровадження і комерціалізації потенційним інвестором і проведено обґрунтування економічної доцільності комерціалізації потенційним інвестором.

5.1 Проведення комерційного та технологічного аудиту науково-технічної розробки

Метою проведення комерційного і технологічного аудиту дослідження за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» є оцінювання науково-технічного рівня та рівня комерційного потенціалу розробки, створеної в результаті науково-технічної діяльності.

Оцінювання науково-технічного рівня розробки та її комерційного потенціалу рекомендується здійснювати із застосуванням 5-ти бальної системи оцінювання за 12-ма критеріями, наведеними в табл. 5.1 [31].

Таблиця 5.1 – Рекомендовані критерії оцінювання науково-технічного рівня і комерційного потенціалу розробки та бальна оцінка

Бали (за 5-ти бальною шкалою)					
	0	1	2	3	4
Технічна здійсненність концепції					
1	Достовірність концепції не підтверджена	Концепція підтверджена експертними висновками	Концепція підтверджена розрахунками	Концепція перевірена на практиці	Перевірено працездатність продукту в реальних
Ринкові переваги (недоліки)					
2	Багато аналогів на малому ринку	Мало аналогів на малому ринку	Кілька аналогів на великому ринку	Один аналог на великому ринку	Продукт не має аналогів на великому ринку
3	Ціна продукту значно вища за ціни аналогів	Ціна продукту дещо вища за ціни аналогів	Ціна продукту приблизно дорівнює цінам аналогів	Ціна продукту дещо нижче за ціни аналогів	Ціна продукту значно нижче за ціни аналогів

Продовження таблиці 5.1

	0	1	2	3	4
4	Технічні та споживчі властивості	Технічні та споживчі властивості продукту	Технічні та споживчі властивості	Технічні та споживчі властивості	Технічні та споживчі властивості
5	Експлуатаційні витрати значно вищі, ніж в аналогів	Експлуатаційні витрати дещо вищі, ніж в аналогів	Експлуатаційні витрати на рівні експлуатаційн	Експлуатаційні витрати трохи нижчі, ніж в аналогів	Експлуатаційні витрати значно нижчі, ніж в аналогів
Ринкові перспективи					
6	Ринок малий і не має позитивної	Ринок малий, але має позитивну динаміку	Середній ринок з позитивною	Великий стабільний ринок	Великий ринок з позитивною динамікою
7	Активна конкуренція великих	Активна конкуренція	Помірна конкуренція	Незначна конкуренція	Конкурентів немає
Практична здійсненність					
8	Відсутні фахівці як з технічної, так і з комерційної реалізації ідеї	Необхідно наймати фахівців або витрачати значні кошти та час на навчання	Необхідне незначне навчання фахівців та збільшення їх штату	Необхідне незначне навчання фахівців	Є фахівці з питань як з технічної, так і з комерційної реалізації ідеї
9	Потрібні значні фінансові ресурси, які відсутні. Джерела фінансування	Потрібні незначні фінансові ресурси. Джерела фінансування відсутні	Потрібні значні фінансові ресурси. Джерела фінансування є	Потрібні незначні фінансові ресурси. Джерела фінансування є	Не потребує додаткового фінансування

Продовження таблиці 5.1

	0	1	2	3	4
10	Необхідна розробка нових матеріалів	Потрібні матеріали, що використовуються у військово промисловому комплексі	Потрібні дорогі матеріали	Потрібні досяжні та дешеві матеріали	Всі матеріали для реалізації ідеї відомі та давно використовуються у виробництві
11	Термін реалізації ідеї більший за 10 років	Термін реалізації ідеї більший за 5 років. Термін окупності інвестицій більше 10-ти років	Термін реалізації ідеї від 3-х до 5-ти років. Термін окупності інвестицій більше 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій від 3-х до 5-ти років	Термін реалізації ідеї менше 3-х років. Термін окупності інвестицій менше 3-х років
12	Необхідна розробка регламентних документів та отримання великої кількості дозвільних документів на виробництво та реалізацію продукту	Необхідно отримання великої кількості дозвільних документів на виробництво та реалізацію продукту, що вимагає значних коштів та часу	Процедура отримання дозвільних документів для виробництва та реалізації продукту вимагає незначних коштів та часу	Необхідно тільки повідомлення відповідним органам про виробництво та реалізацію продукту	Відсутні будь-які регламентні обмеження на виробництво та реалізацію продукту

Результати оцінювання науково-технічного рівня та комерційного потенціалу науково-технічної розробки потрібно звести до таблиці.

Таблиця 5.2 – Результати оцінювання науково-технічного рівня і комерційного потенціалу розробки експертами

Критерії	Експерт (ПІБ, посада)		
	1	2	3
	Бали:		
1. Технічна здійсненність концепції	4	3	4
2. Ринкові переваги (наявність аналогів)	3	3	4
3. Ринкові переваги (ціна продукту)	3	3	3
4. Ринкові переваги (технічні властивості)	4	4	4
5. Ринкові переваги (експлуатаційні витрати)	3	3	3
6. Ринкові перспективи (розмір ринку)	3	4	3
7. Ринкові перспективи (конкуренція)	3	3	4
8. Практична здійсненність (наявність фахівців)	3	3	3
9. Практична здійсненність (наявність фінансів)	3	4	4
10. Практична здійсненність (необхідність нових матеріалів)	3	4	3
11. Практична здійсненність (термін реалізації)	3	3	3
12. Практична здійсненність (розробка документів)	3	3	3
Сума балів	38	40	41
Середньоарифметична сума балів $СБ_c$	39,7		

За результатами розрахунків, наведених в таблиці 5.2, зробимо висновок щодо науково-технічного рівня і рівня комерційного потенціалу розробки. При цьому використаємо рекомендації, наведені в табл. 5.3 **[**32**]**.

Таблиця 5.3 – Науково-технічні рівні та комерційні потенціали розробки

Середньоарифметична сума балів СБ , розрахована на основі висновків	Науково-технічний рівень та комерційний потенціал розробки
41…48	Високий
31…40	Вище середнього
21…30	Середній
11…20	Нижче середнього
0…10	Низький

Згідно проведених досліджень рівень комерційного потенціалу розробки за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» становить 39,7 бала, що, відповідно до таблиці 5.3, свідчить про комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вище середнього).

5.2 Розрахунок витрат на проведення науково-дослідної роботи

Витрати, пов’язані з проведенням науково-дослідної роботи на тему «Удосконалення методів і засобів острівної архітектури побудови вебдодатків», під час планування, обліку і калькулювання собівартості науково-дослідної роботи групуємо за відповідними статтями.

5.2.1 Витрати на оплату праці

До статті «Витрати на оплату праці» належать витрати на виплату основної та додаткової заробітної плати керівникам відділів, лабораторій, секторів і груп,

науковим, інженерно-технічним працівникам, конструкторам, технологам, креслярам, копіювальникам, лаборантам, робітникам, студентам, аспірантам та іншим працівникам, безпосередньо зайнятим виконанням конкретної теми, обчисленої за посадовими окладами, відрядними розцінками, тарифними ставками згідно з чинними в організаціях системами оплати праці.

Основна заробітна плата дослідників

Витрати на основну заробітну плату дослідників ($З_о$) розраховуємо у відповідності до посадових окладів працівників, за формулою 5.1 [32]:

$$З_о = \sum_{i=1}^{k} \frac{M_{ni} \cdot t_i}{T_p}, \quad (5.1)$$

де k – кількість посад дослідників залучених до процесу досліджень;

M_{ni} – місячний посадовий оклад конкретного дослідника, грн;

t_i – число днів роботи конкретного дослідника, дні;

T_p – середнє число робочих днів в місяці, T_p=24 дні.

$$З_о = 50000{,}00 \cdot 110 / 24 = 229166{,}67 \text{грн.}$$

Проведені розрахунки зведемо до таблиці.

Таблиця 5.4 – Витрати на заробітну плату дослідників

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на заробітну плату, грн
Керівник проекту	50000	2083,33	110	229166,67
Інженер-розробник програмного забезпечення	40000	1666,67	100	166666,67

Продовження таблиці 5.4

Найменування посади	Місячний посадовий оклад, грн	Оплата за робочий день, грн	Число днів роботи	Витрати на заробітну плату, грн
Науковий співробітник дослідження проблем програмного забезпечення	18000	750,00	25	18750,00
Всього				414583,33

Основна заробітна плата робітників

Витрати на основну заробітну плату робітників ($З_р$) за відповідними найменуваннями робіт НДР розраховуємо за формулою 5.2:

$$З_р = \sum_{i=1}^{n} C_i \cdot t_i , \qquad (5.2)$$

де C_i – погодинна тарифна ставка робітника відповідного розряду, за виконану відповідну роботу, грн/год;

t_i – час роботи робітника при виконанні визначеної роботи, год.

Погодинну тарифну ставку робітника відповідного розряду C_i можна визначити за формулою 5.3:

$$C_i = \frac{M_M \cdot K_i \cdot K_c}{T_p \cdot t_{зм}} , \qquad (5.3)$$

де M_M – розмір прожиткового мінімуму працездатної особи, або мінімальної місячної заробітної плати (в залежності від діючого законодавства), приймемо $Мм$=8000,00 грн;

K_i – коефіцієнт міжкваліфікаційного співвідношення для встановлення тарифної ставки робітнику відповідного розряду (табл. Б.2, додаток Б) [32];

K_c – мінімальний коефіцієнт співвідношень місячних тарифних ставок робітників першого розряду з нормальними умовами праці виробничих об'єднань і підприємств до законодавчо встановленого розміру мінімальної заробітної плати.

T_p – середнє число робочих днів в місяці, приблизно T_p = 24 дні;

$t_{зм}$ – тривалість зміни, год.

$C_1 = 8000{,}00 \cdot 1{,}10 \cdot 1{,}65 / (24 \cdot 8) = 75{,}63$ грн.

$З_{р1} = 75{,}63 \cdot 6{,}00 = 453{,}75$ грн.

Таблиця 5.5 – Величина витрат на основну заробітну плату робітників

Найменування робіт	Тривалість роботи, год	Розряд роботи	Тарифний коефіцієнт	Погодинна тарифна ставка, грн	Величина оплати на робітника грн
Установка електронно-обчислювального обладнання	6	2	1,1	75,63	453,75
Підготовка робочого місця дослідника	2,4	2	1,1	75,63	181,50
Інсталяція програмного забезпечення	2,2	5	1,7	116,88	257,13

Аналіз покращень	5	5	1,7	116,88	584,38
Аналіз потоку даних	5	3	1,3	89,38	446,88

Продовження таблиці 5.5

Тестування на різних додатках	15	5	1,7	116,88	1753,13
Всього					3676,75

Додаткова заробітна плата дослідників та робітників

Додаткову заробітну плату розраховуємо як 10 … 12% від суми основної заробітної плати дослідників та робітників за формулою 5.4:

$$З_{дод} = (З_о + З_р) \cdot \frac{Н_{дод}}{100\%}, \quad (5.4)$$

де $Н_{дод}$ – норма нарахування додаткової заробітної плати. Приймемо 10%.

$З_{дод}$ = (414583,33+ 3676,75) · 10 / 100% = 41826,01грн.

5.2.2 Відрахування на соціальні заходи

Нарахування на заробітну плату дослідників та робітників розраховуємо як 22% від суми основної та додаткової заробітної плати дослідників і робітників за формулою5.5:

$$З_н = (З_о + З_р + З_{дод}) \cdot \frac{Н_{зп}}{100\%} \quad (5.5)$$

де $Н_{зп}$ – норма нарахування на заробітну плату. Приймаємо 22%.

Зн = (414583,33+ 3676,75+ 41826,01) · 22 / 100% = 101218,94 грн.

5.2.3 Сировина та матеріали

До статті «Сировина та матеріали» належать витрати на сировину, основні та допоміжні матеріали, інструменти, пристрої та інші засоби і предмети праці, які придбані у сторонніх підприємств, установ і організацій та витрачені на проведення досліджень.

Витрати на матеріали (*М*), у вартісному вираженні розраховуються окремо за кожним видом матеріалів за формулою 5.6:

$$M = \sum_{j=1}^{n} H_j \cdot Ц_j \cdot К_j - \sum_{j=1}^{n} B_j \cdot Ц_{вj}\,, \qquad (5.6)$$

де H_j – норма витрат матеріалу *j*-го найменування, кг;

n – кількість видів матеріалів;

$Ц_j$ – вартість матеріалу *j*-го найменування, грн/кг;

$К_j$ – коефіцієнт транспортних витрат, ($К_j$ = *1,1 … 1,15*);

B_j – маса відходів *j*-го найменування, кг;

$Ц_{вj}$ – вартість відходів *j*-го найменування, грн/кг.

$М_1 = 2{,}00 \cdot 180{,}00 \cdot 1{,}1 - 0{,}000 \cdot 0{,}00 = 396{,}0$ грн.

Проведені розрахунки зведемо до таблиці.

Таблиця 5.6 – Витрати на матеріали

Найменування матеріалу, марка, тип, сорт	Ціна за од, грн	Норма витрат, од	Величина відходів, кг	Ціна відходів, грн/кг	Вартість витраченого матеріалу, грн
Папір офісний Maestro Standard+ B клас A4 80 г/м2, уп	180	2	0	0	396

Продовження таблиці 5.6

Найменування матеріалу, марка, тип, сорт	Ціна за од, грн	Норма витрат, од	Величина відходів, кг	Ціна відходів, грн/кг	Вартість витраченого матеріалу, грн
Папір для записів YES Tetra Pak 8bit UA 200 арк., уп	110	1	0	0	121
Органайзер офісний Настільний набір Buromax 16 items, black (BM.6302-01), шт	210	2	0	0	462
Канцелярське приладдя (набір працівника), шт	170	2	0	0	374
Картридж для принтера Canon LBP6500	1100	1	0	0	1210
Диск оптичний NewLine CD-RW	15	3	0	0	49,5
Flesh-пам'ять Kingston 16 GB	165	1	0	0	181,5
Тека для паперів CALIPSO BOX	82	2	0	0	180,4
Всього					3490,63

5.2.4 Розрахунок витрат на комплектуючі

Витрати на комплектуючі *($К_в$)*, які використовують при проведенні НДР на тему «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» відсутні.

5.2.5 Спецустаткування для наукових (експериментальних) робіт

До статті «Спецустаткування для наукових (експериментальних) робіт» належать витрати, пов’язані з виготовленням та придбанням спеціального устаткування, необхідного для проведення досліджень, експериментів чи наукових випробувань. До цих витрат також входять витрати на проектування, виготовлення, транспортування, монтаж, встановлення, а також підготовку устаткування до роботи, що забезпечує його належне функціонування відповідно до мети дослідження.

Балансову вартість спецустаткування розраховуємо за формулою 5.7:

$$B_{спец} = \sum_{i=1}^{k} Ц_i \cdot C_{пр.i} \cdot K_i \quad , \qquad (5.7)$$

де $Ц_i$ – ціна придбання одиниці спецустаткування даного виду, марки, грн;

$C_{пр.i}$ –кількість одиниць устаткування відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує доставку, монтаж, налагодження устаткування тощо, ($K_i = 1,10…1,12$);

k – кількість найменувань устаткування.

$B_{спец} = 25000 \cdot 1 \cdot 1,1 = 27500$ грн.

Отримані результати зведемо до таблиці:

Таблиця 5.7 – Витрати на придбання спецустаткування по кожному виду

Найменування устаткування	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
ПК ASUS S500SC-51140F0030 SFF (90PF02K2-M02E5016), монітор	1	25000	27500
ПК HP Pro 400-G9 SFF, (8N8V2AA), монітор	1	30000	33000
Всього			60500

5.2.6 Програмне забезпечення для наукових (експериментальних) робіт

До статті «Програмне забезпечення для наукових (експериментальних) робіт» належать витрати на розробку та придбання спеціальних програмних засобів і програмного забезпечення, (програм, алгоритмів, баз даних) необхідних для проведення досліджень, також витрати на їх проектування, формування та встановлення.

Балансову вартість програмного забезпечення розраховуємо за формулою5.8:

$$B_{прг} = \sum_{i=1}^{k} Ц_{iпрг} \cdot C_{прг.i} \cdot K_i \ , \qquad (5.8)$$

де $Ц_{iпрг}$ – ціна придбання одиниці програмного засобу даного виду, грн;

$C_{прг.i}$ – кількість одиниць програмного забезпечення відповідного найменування, які придбані для проведення досліджень, шт.;

K_i – коефіцієнт, що враховує інсталяцію, налагодження програмного засобу

тощо, (K_i = *1,10…1,12);*

k – кількість найменувань програмних засобів.

$B_{прг}$ = *12500,00 2 · 1,12 = 28000 грн.*

Отримані результати зведемо до таблиці:

Таблиця 5.8– Витрати на придбання програмних засобів по кожному виду

Найменування програмного засобу	Кількість, шт	Ціна за одиницю, грн	Вартість, грн
Операційна система Microsoft Windows 10 Pro for Workstations	2	12500	28000
Webstorm (ліцензія/за міс.)	4	5230	23430,4
Всього			51430,4

Природробці також використовувалось і безкоштовне програмне забезпечення Unity та Visual Studio.

5.2.7 Амортизація обладнання, програмних засобів та приміщень

У спрощеному вигляді амортизаційні відрахування по кожному виду обладнання, приміщень та програмному забезпеченню тощо, розраховуємо з використанням прямолінійного методу амортизації за формулою:

$$A_{обл} = \frac{Ц_б}{T_в} \cdot \frac{t_{вик}}{12}, \quad (5.9)$$

де $Ц_б$ – балансова вартість обладнання, програмних засобів, приміщень тощо, які використовувались для проведення досліджень, грн;

$t_{вик}$ – термін використання обладнання, програмних засобів, приміщень під час досліджень, місяців;

$T_в$ – строк корисного використання обладнання, програмних засобів, приміщень тощо, років.

$A_{обл} = (27500{,}00 \cdot 4) / (4 \cdot 12) = 2291{,}67$ грн.

Проведені розрахунки зведемо до таблиці.

Таблиця 5.9 – Амортизаційні відрахування по кожному виду обладнання

Найменування обладнання	Балансова вартість, грн	Строк корисного використання, років	Термін використання обладнання, місяців	Амортизаційні відрахування, грн
Персональний комп'ютер ASUS	27500	4	4	2291,67
Персональний комп'ютер HP	33000	4	4	2750,00
Робоче місце дослідника	7880	5	3	394,00
Оргтехніка	8675	4	3	542,19
Операційна система Microsoft Windows 10 Pro for Workstations	28000	3	3	2333,33
Webstorm (ліцензія/за міс.)	23430,4	2	4	3905,07
Всього				12216,25

5.2.8 Паливо та енергія для науково-виробничих цілей

Витрати на силову електроенергію ($B_е$) розраховуємо за формулою 5.10:

$$B_e = \sum_{i=1}^{n} \frac{W_{yi} \cdot t_i \cdot Ц_e \cdot K_{вni}}{\eta_i}, \qquad (5.10)$$

де W_{yi} – встановлена потужність обладнання на визначеному етапі розробки, кВт;

t_i – тривалість роботи обладнання на етапі дослідження, год;

$Ц_e$ – вартість 1 кВт-години електроенергії, грн; (вартість електроенергії визначається за даними енергопостачальної компанії), приймемо $Ц_e = 11$ грн;

$K_{вп i}$ – коефіцієнт, що враховує використання потужності, $K_{вп i} < 1$;

η_i – коефіцієнт корисної дії обладнання, $\eta_i < 1$.

$B_e = 0,3 \cdot 880,0 \cdot 11,0 \cdot 0,95 / 0,97 = 2844,12$ *грн.*

Проведені розрахунки зведемо до таблиці.

Таблиця 5.10 – Витрати на електроенергію

Найменування обладнання	Встановлена потужність, кВт	Тривалість роботи, год	Сума, грн
ПК ASUS	0,3	880	2844,12
ПК HP	0,3	880	2844,12
Робоче місце дослідника	0,15	880	1422,06
Оргтехніка	0,25	30	80,80
Всього			7191,11

5.2.9 Службові відрядження

До статті «Службові відрядження» дослідної роботи належать витрати на відрядження штатних працівників, працівників організацій, які працюють за договорами цивільно-правового характеру, аспірантів, зайнятих розробленням досліджень, відрядження, пов’язані з проведенням випробувань машин та приладів, а також витрати на відрядження на наукові з’їзди, конференції, наради, пов’язані з виконанням конкретних досліджень.

Витрати за статтею «Службові відрядження» розраховуємо як 20…25% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{св} = (З_о + З_р) \cdot \frac{Н_{св}}{100\%}, \quad (5.11)$$

де $Н_{св}$ – норма нарахування за статтею «Службові відрядження», приймемо $Н_{св}$ = 20%.

$В_{св}$ = (414583,33+ 3676,75) · 20 / 100% = 83652,02 грн.

5.2.10 Витрати на роботи, які виконують сторонні підприємства, установи і організації

Витрати за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації» розраховуємо як 30…45% від суми основної заробітної плати дослідників та робітників за формулою:

$$B_{сп} = (З_о + З_р) \cdot \frac{Н_{сп}}{100\%}, \quad (5.12)$$

де $Н_{сп}$ – норма нарахування за статтею «Витрати на роботи, які виконують сторонні підприємства, установи і організації», приймемо $Н_{сп}$= 30%.

$В_{сп}$ = (414583,33+ 3676,75) · 30 / 100% = 125478,03 грн.

5.2.11 Інші витрати

До статті «Інші витрати» належать витрати, які не знайшли відображення у зазначених статтях витрат і можуть бути віднесені безпосередньо на собівартість досліджень за прямими ознаками.

Витрати за статтею «Інші витрати» розраховуємо як 50…100% від суми основної заробітної плати дослідників та робітників за формулою5.13:

$$I_в = (З_о + З_р) \cdot \frac{Н_{ів}}{100\%}, \quad (5.13)$$

де $Н_{ів}$ – норма нарахування за статтею «Інші витрати», приймемо $Н_{ів}$ = 50%.

$I_в$ = (414583,33+ 3676,75) · 50 / 100% = 209130,04 грн.

5.2.12 Накладні (загальновиробничі) витрати

До статті «Накладні (загальновиробничі) витрати» належать: витрати, пов'язані з управлінням організацією; витрати на винахідництво та раціоналізацію; витрати на підготовку (перепідготовку) та навчання кадрів; витрати, пов'язані з набором робочої сили; витрати на оплату послуг банків; витрати, пов'язані з освоєнням виробництва продукції; витрати на науково-технічну інформацію та рекламу та ін.

Витрати за статтею «Накладні (загальновиробничі) витрати» розраховуємо як 100…150% від суми основної заробітної плати дослідників та робітників за формулою 5.14:

$$В_{нзв} = (З_о + З_р) \cdot \frac{Н_{нзв}}{100\%}, \quad (5.14)$$

де $Н_{нзв}$ – норма нарахування за статтею «Накладні (загальновиробничі) витрати», приймемо $Н_{нзв}$ = 100%.

$В_{нзв}$ = (414583,33+ 3676,75) · 120 / 100% = 418 260,08грн.

Витрати на проведення науково-дослідної роботи на тему «Підвищення захищеності процесу розробки проектів на основі удосконаленої моделі розподілу секрету та розмежування доступу між учасниками проекту» розраховуємо як суму всіх попередніх статей витрат за формулою 5.15:

$$В_{заг} = З_о + З_р + З_{дод} + З_н + М + К_в + В_{спец} + В_{прг} + А_{обл} + В_е + В_{св} + В_{сп} + І_в + В_{нзв}. \quad (5.15)$$

$В_{заг}$= 1532653,59.

Загальні витрати *ЗВ* на завершення науково-дослідної (науково-технічної) роботи та оформлення її результатів розраховується за формулою:

$$ЗВ = \frac{B_{заг}}{\eta}, \quad (5.16)$$

де η - коефіцієнт, який характеризує етап (стадію) виконання науково-дослідної роботи, приймемо η =0,9.

ЗВ = 1532653,59/ 0,9 = 1702948,43 грн.

5.3 Розрахунок економічної ефективності науково-технічної розробки при її можливій комерціалізації потенційним інвестором

В ринкових умовах узагальнюючим позитивним результатом, що його може отримати потенційний інвестор від можливого впровадження результатів тієї чи іншої науково-технічної розробки, є збільшення у потенційного інвестора величини чистого прибутку.

Результати дослідження проведені за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» передбачають комерціалізацію протягом 3-х років реалізації на ринку.

В цьому випадку майбутній економічний ефект буде формуватися на основі таких даних:

ΔN – збільшення кількості споживачів продукту, у періоди часу, що аналізуються, від покращення його певних характеристик;

Показник	1-й рік	2-й рік	3-й рік
Збільшення кількості споживачів, користувачів вебдодатку	10000	8000	6000

N – кількість споживачів які використовували аналогічний продукт у році до впровадження результатів нової науково-технічної розробки, приймемо 50000 користувачів;

$Ц_б$ – вартість програмного продукту у році до впровадження результатів розробки, приймемо 500 грн/користування протягом року;

$\pm\Delta Ц_о$ – зміна вартості програмного продукту від впровадження результатів науково-технічної розробки, приймемо 100,00 грн.

Можливе збільшення чистого прибутку у потенційного інвестора $\Delta П_i$ для кожного із 3-х років, протягом яких очікується отримання позитивних результатів від можливого впровадження та комерціалізації науково-технічної розробки, розраховуємо за формулою **[**32**]:**

$$\Delta П_i = (\pm\Delta Ц_о \cdot N + Ц_о \cdot \Delta N)_i \cdot \lambda \cdot \rho \cdot (1 - \frac{\vartheta}{100}), \qquad (5.17)$$

де λ – коефіцієнт, який враховує сплату потенційним інвестором податку на додану вартість. У 2024 році ставка податку на додану вартість складає 20%, а коефіцієнт $\lambda = 0{,}8333$;

ρ – коефіцієнт, який враховує рентабельність інноваційного продукту. Приймемо ρ =45%;

ϑ – ставка податку на прибуток, який має сплачувати потенційний інвестор, у 2024 році $\vartheta = 18\%$;

Збільшення чистого прибутку 1-го року:

$\Delta П_1$ = (100,00·50000,00+600,00·10000)·0,83·0,45·(1-0,18/100%)= 3381147 грн.

Збільшення чистого прибутку 2-го року:

$\Delta П_2$ =(100,00·50000,00+600,00·(10000+8000))·0,83·0,45·(1-0,18/100%)= 4856556,6 грн.

Збільшення чистого прибутку 3-го року:

$\Delta П_3$ =(100,00·50000,00+600,00·(10000+8000+6000))·0,83·0,45·(1-0,18/100%)= 5963113,8 грн.

Приведена вартість збільшення всіх чистих прибутків ПП (формула 5.18), що їх може отримати потенційний інвестор від можливого впровадження та комерціалізації науково-технічної розробки:

$$ПП = \sum_{i=1}^{T} \frac{\Delta П_i}{(1+\tau)^t}, \qquad (5.18)$$

де $\Delta П_i$ – збільшення чистого прибутку у кожному з років, протягом яких виявляються результати впровадження науково-технічної розробки, грн;

T – період часу, протягом якого очікується отримання позитивних результатів від впровадження та комерціалізації науково-технічної розробки, роки;

τ – ставка дисконтування, за яку можна взяти щорічний прогнозований рівень інфляції в країні, τ =0,25;

t – період часу (в роках) від моменту початку впровадження науково-технічної розробки до моменту отримання потенційним інвестором додаткових чистих прибутків у цьому році.

$ПП =$

3381147/(1+0,25)^1+4856556,6/(1+0,25)^2+5963113,8/(1+0,25)^3=8866228,09грн

Величина початкових інвестицій PV (формула 5.19), які потенційний інвестор має вкласти для впровадження і комерціалізації науково-технічної розробки:

$$PV = k_{інв} \cdot ЗВ, \tag{5.19}$$

де $k_{інв}$ – коефіцієнт, що враховує витрати інвестора на впровадження науково-технічної розробки та її комерціалізацію, приймаємо $k_{інв}$=2;

$ЗВ$ – загальні витрати на проведення науково-технічної розробки та оформлення її результатів, приймаємо 1702948,43 грн.

$PV = k_{інв} \cdot ЗВ$ = 2 · 1702948,43= 3405896,87грн.

Абсолютний економічний ефект $E_{абс}$ (формула 5.20) для потенційного інвестора від можливого впровадження та комерціалізації науково-технічної розробки становитиме:

$$E_{абс} = ПП - PV \tag{5.20}$$

де $ПП$ – приведена вартість зростання всіх чистих прибутків від можливого впровадження та комерціалізації науково-технічної розробки, 8866228,09грн;

PV – теперішня вартість початкових інвестицій, 3405896,87 грн.

$E_{абс} = ПП - PV$ = 8866228,09- 3405896,87= 5460331,22грн.

Внутрішня економічна дохідність інвестицій $E_в$ (формула 5.21), які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$E_в = \sqrt[Tж]{1 + \frac{E_{абс}}{PV}} - 1, \quad (5.21)$$

де $E_{абс}$ – абсолютний економічний ефект вкладених інвестицій, 5460331,22грн;

PV – теперішня вартість початкових інвестицій, 3405896,87 грн;

$T_ж$ – життєвий цикл науково-технічної розробки, тобто час від початку її розробки до закінчення отримування позитивних результатів від її впровадження, 3 роки.

$E_в = \sqrt[Tж]{1 + \frac{E_{абс}}{PV}} - 1$ = (1+5460331,22/3405896,87)$^{1/3}$= 0,38.

Мінімальна внутрішня економічна дохідність вкладених інвестицій $\tau_{мін}$ (формула 5.22):

$$\tau_{мін} = d + f, \quad (5.22)$$

де d – середньозважена ставка за депозитними операціями в комерційних банках; в 2024 році в Україні d =0,11;

f – показник, що характеризує ризикованість вкладення інвестицій, приймемо 0,15.

$\tau_{мін}$ = 0,11+0,15 = 0,26 < 0,38 свідчить про те, що внутрішня економічна дохідність інвестицій $E_в$, які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки вища мінімальної внутрішньої дохідності. Тобто інвестувати в науково-дослідну роботу за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» доцільно.

Період окупності інвестицій $T_{ок}$ які можуть бути вкладені потенційним інвестором у впровадження та комерціалізацію науково-технічної розробки:

$$T_{ок} = \frac{1}{E_в}, \tag{5.23}$$

де $E_в$ – внутрішня економічна дохідність вкладених інвестицій.

$T_{ок}$ = 1 / 0,38 = 2,66 року.

$T_{ок}$ < 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

5.4 Висновок

Згідно проведених досліджень рівень комерційного потенціалу розробки за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків» становить 39,7 бала, що, свідчить про комерційну важливість проведення даних досліджень (рівень комерційного потенціалу розробки вище середнього).

Також термін окупності становить 2,66 р., що менше 3-х років, що свідчить про комерційну привабливість науково-технічної розробки і може спонукати

потенційного інвестора профінансувати впровадження даної розробки та виведення її на ринок.

Отже можна зробити висновок про доцільність проведення науково-дослідної роботи за темою «Удосконалення методів і засобів острівної архітектури побудови вебдодатків».

ВИСНОВКИ

У ході виконання магістерської кваліфікаційної роботи було розроблено методи та програмні засоби для впровадження острівної архітектури вебдодатків. Робота була виконана та оформлена відповідно до методичних вказівок [33, 34].

Було проаналізовано сучасний стан питання острівної архітектури. Розглянуто основні існуючі підходи, фреймворки та бібліотеки, визначено їхні особливості та недоліки, а також проведено порівняння з розробленим програмним рішенням. У результаті аналізу було обрано мову програмування JavaScript, генератор статичних сайтів Eleventy, а також створено власну бібліотеку для підтримки острівної архітектури. Це дозволило досягти високої продуктивності та простоти реалізації компонентів.

Для реалізації системи було використано вебкомпоненти як базову одиницю острівної архітектури, що забезпечує ізоляцію логіки і стилів, а також можливість багаторазового використання незалежних модулів. Для оптимізації роботи системи було інтегровано ве-воркери для виконання ресурсомістких обчислень у фоновому режимі та реалізовано комбіноване кешування як на стороні клієнта, так і на стороні сервера.

Було розроблено платформу для створення вебдодатків, яка включає механізми часткової та адаптивної гідрації компонентів, оптимізації

завантаження сторінок через відкладене завантаження (lazy-loading) та попереднє завантаження (preloading). Крім того, було впроваджено механізми взаємодії між "островами" через подієво-орієнтовану архітектуру, що дозволило забезпечити ефективний обмін даними між компонентами з мінімальними накладними витратами.

Розроблено бібліотеку для інтеграції острівної архітектури на основі Eleventy, що дозволяє легко створювати автономні компоненти для вебдодатків із високою продуктивністю. Основні модулі бібліотеки включають методи для гідрації компонентів, оптимізації завантаження ресурсів та механізми комунікації між компонентами.

Було також розроблено метод масштабування системи використання вебворкерів. Це забезпечило можливість гнучкого розширення системи у разі збільшення користувацького навантаження та підвищення вимог до продуктивності.

Тестування розробленої системи показало її повну працездатність та відповідність поставленим технічним завданням. Розроблено інструкцію для розробників щодо інтеграції та використання острівної архітектури у вебдодатках на основі Eleventy.

У результаті роботи вдалося створити бібліотеку, що забезпечує високу швидкодію та гнучкість при розробці сучасних вебдодатків, завдяки чому було досягнуто поставлені цілі щодо покращення структури, продуктивності та можливості масштабування вебсистем на основі острівної архітектури.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Island architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.patterns.dev/vanilla/islands-architecture
2. Войтко В.В. Удосконалення методу "острови" для підвищення швидкості роботи програмних застосунків у браузері / В.В. Войтко, Г.О. Черноволик, Н.Є. Барчук, О.В. Гаврилюк, К.С. Осипенко // Електронні інформаційні ресурси: створення, використання, доступ та управління. Збірник матеріалів Міжнародної науково-практичної Інтернет конференції 20-21 листопада 2024р. – Суми/Вінниця: НІКО / КЗВО «Вінницька академія безперервної освіти», 2024. – С. 31-34.
3. The Evolution of Web Development: From 90s to Today [Електронний ресурс] – Режим доступу до ресурсу: https://www.sapientcodelabs.com/blogs/evolution-of-web-development-journey-from-90s-to-now
4. Server Side Rendering in JavaScript – SSR vs CSR Explained [Електронний ресурс] – Режим доступу до ресурсу: https://www.freecodecamp.org/news/server-side-rendering-javascript/
5. Single Page Web Applications: JavaScript end-to-end / Michael Mikowski, Josh Powel: Manning Publications, 2013, 13 с.
6. Bootstrapping Microservices with Docker, Kubernetes, GitHub Actions, and Terraform / Ashley Davis: Manning Publications: 2021, 440 с.
7. Dynamic Imports: The Real MVP of Islands Architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.jacobmilhorn.com/posts/dynamic-imports-the-mvp-of-islands-architecture/
8. Single-Page Application vs. Multi-Page Application [Електронний ресурс] – Режим доступу до ресурсу: https://www.bitstudios.com/blog/single-page-application-vs-multi-page-application/

9. JavaScript: The Definitive Guide: Master the World's Most-Used Programming Language / David Flanagan: O'Reilly Media, 2020, 23 с.

10. You Don't Know JS Yet: Scope & Closures / Kyle Simpson, Simon St.Laurent, Sarah Drasner : GetiPub & Leanpub, 2020, 56 с.

11. High Performance Browser Networking / Ilya Grigorik: O'Reilly Media, 2013, 113с

12. Astro [Електронний ресурс] – Режим доступу до ресурсу: https://astro.build

13. Designing Data-Intensive Applications / Martin Kleppmann: O'Reilly Media, 2017, 82 с.

14. Qwik resumable [Електронний ресурс] – Режим доступу до ресурсу: https://qwik.dev/docs/concepts/resumable/

15. Blazor [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/aspnet/core/blazor/webassembly-build-tools-and-aot?view=aspnetcore-9.0

16. Building Microservices / Sam Newman: O'Reilly Media, 2021, 15 с.

17. Round Robin Scheduling for the same arrival time [Електронний ресурс] – Режим доступу до ресурсу: https://www.geeksforgeeks.org/program-for-round-robin-scheduling-for-the-same-arrival-time/

18. Clean Architecture: A Craftsman's Guide to Software Structure and Design / Robert C. Martin, Prentice Hall: Pearson, 2017, 61 с.

19. Virtual DOM and Internals [Електронний ресурс] – Режим доступу до ресурсу: reactjs.org/docs/faq-internals.html

20. Reactivity in Depth time [Електронний ресурс] – Режим доступу до ресурсу: https://vuejs.org/guide/extras/reactivity-in-depth.html

21. Using shadow DOM [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM

22. Elevenity documentation [Електронний ресурс] – Режим доступу до ресурсу: https://www.11ty.dev/docs/

23. Hydration: What is it? [Електронний ресурс] – Режим доступу до ресурсу: https://dev.to/costamatheus97/hydration-what-is-it-1lgl

24. UML Distilled: A Brief Guide to the Standard Object Modeling Language / Martin Fauler: Addison-Wesley, 2003, 75 с.

25. Service Worker API [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

26. Javascript [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript

27. Python [Електронний ресурс] – Режим доступу до ресурсу: https://www.python.org

28. C# [Електронний ресурс] – Режим доступу до ресурсу: https://learn.microsoft.com/en-us/dotnet/csharp/

29. Java [Електронний ресурс] – Режим доступу до ресурсу: https://docs.oracle.com/en/java/

30. C++ [Електронний ресурс] – Режим доступу до ресурсу: https://cplusplus.com/doc/

31. Кавецький В. В. Економічне обґрунтування інноваційних рішень: практикум / В. В. Кавецький, В. О. Козловський, І. В. Причепа. Вінниця : ВНТУ, 2016. 113 с.

32. Методичні вказівки до виконання економічної частини магістерських кваліфікаційних робіт / Уклад.: В. О. Козловський, О. Й. Лесько, В.В.Кавецький. Вінниця: ВНТУ, 2021. 42 с.

33. Положення про кваліфікаційну роботу на другому (вищому) рівні вищої освіти. Розробники: А.О. Семенов, Л.П. Громова, Т.В. Макарова, О.В. Сердюк. Вінниця : ВНТУ, 2021. – 60 с.

34. Методичні вказівки до виконання магістерської кваліфікаційної роботи для студентів спеціальності 121 «Інженерія програмного забезпечення» / уклад. : О. Н. Романюк, Г. О. Черноволик. – Вінниця : ВНТУ, 2022. – 50 с.

ДОДАТКИ

111

Додаток А – Технічне завдання

Міністерство освіти і науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп'ютерної інженерії

ЗАТВЕРДЖУЮ
Завідувач кафедри ПЗ
Романюк О.Н.
«19» вересня 2024 р.

Технічне завдання

на магістерську кваліфікаційну роботу «Удосконалення методів і засобів острівної архітектури побудови веб-додатків» за спеціальністю 121 – Інженерія програмного забезпечення

Керівник магістерської кваліфікаційної роботи:
к.т.н., доц. каф. ПЗ. Черноволик Г.О.
«19» вересня 2024 року.

Виконав:
студент гр.1ПІ-23м Осипенко К.С.
«19» вересня 2024 року.

Вінниця – 2024 року

1. Найменування та галузь застосування

Магістерська кваліфікаційна робота: «Удосконалення методів і засобів острівної архітектури побудови веб додатків». Галузь застосування – вебзастосунки.

2. Підстава для розробки.

Підставою для виконання магістерської кваліфікаційної роботи (МКР) є індивідуальне завдання на МКР та наказ ректора № 310 від 17 вересня 2024 р. по ВНТУ про закріплення тем МКР.

3. Мета та призначення розробки.

Метою магістерської кваліфікаційної роботи є удосконалення методів і засобів острівної архітектури побудови вебдодатків для підвищення їхньої продуктивності, гнучкості та зручності використання шляхом розробки власної бібліотеки, що забезпечить розробника необхідним функціоналом для реалізації.

Призначення роботи – методи та програмні засоби впровадження острівної архітектури у вебдодатках.

4. Вихідні дані для проведення НДР

Перелік основних літературних джерел, на основі яких буде виконуватись МКР.

1. Island architecture [Електронний ресурс] – Режим доступу до ресурсу: https://www.patterns.dev/vanilla/islands-architecture

2. High Performance Browser Networking / Ilya Grigorik: O'Reilly Media, 2013, 113с

3. Using shadow DOM [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/API/Web_components/Using_shadow_DOM

5. Технічні вимоги

Вхідні дані до роботи – проєкт вебдодатка, специфікація вебдодатка; вихідні дані – оптимізований бандл додатку.

6. Конструктивні вимоги.

Графічна та текстова документація повинна відповідати діючим стандартам України.

7. Перелік технічної документації, що пред'являється по закінченню робіт:

– пояснювальна записка до МКР;
– технічне завдання;
– лістинги програми.

8. Вимоги до рівня уніфікації та стандартизації

При розробці програмних засобів слід дотримуватися уніфікації і ДСТУ.

9. Стадії та етапи розробки:

№ з/п	Назва етапів магістерської кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області архітектур побудови вебзастосунків	20.09.2024 30.09.2024	
2	Розробка методу та моделей системи	01.10.2024 17.10.2024	

3	Розробка програмного модуля бібліотеки для створення вебзастосунків на основі архітекрути островів	18.10.2024 04.11.2024	
4	Тестування програмного додатку	05.11.2024 21.11.2024	
5	Економічна частина	22.11.2024 30.11.2024	

10. Порядок контролю та прийняття.

Виконання етапів магістерської кваліфікаційної роботи контролюється керівником згідно з графіком виконання роботи. Прийняття магістерської кваліфікаційної роботи здійснюється ДЕК, затвердженою зав. кафедрою згідно з графіком.

115

ДОДАТОК Б

Протокол перевірки на плагіат

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи: **Удосконалення методів і засобів острівної архітектури побудови веб додатків**

Тип роботи: кваліфікаційна робота
Підрозділ : кафедра програмного забезпечення, ФІТКІ, 1ПІ-23м
Науковий керівник: к.т.н., доц. каф. ПЗ Черноволик Г.О.

Turnitin	
Оригінальність	96 %
Схожість	4 %

Аналіз звіту подібності

▪ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.
▫ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.
▫ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.
Заявляю, що ознайомлений з повним звітом подібності, який був згенерований Системою щодо роботи «Удосконалення методів і засобів острівної архітектури побудови веб додатків».

Автор _____ Осипенко Костянтин Сергійович

Опис прийнятого рішення: **допустити до захисту**

Особа, відповідальна за перевірку _____ Черноволик Г. О.
(підпис) (прізвище, ініціали)

Експерт _____ _____
(за потреби) (підпис) (прізвище, ініціали, посада)

Додаток В – Лістинг коду

island-worker.js

```
self.addEventListener('message', async (event) => {
    const { action, props } = event.data;

    if (action === 'hydrate') {
        const hydratedProps = await hydrateProps(props);
        postMessage({ action: 'hydrated', hydratedProps });
    }

    if (action === 'dehydrate') {
        const dehydratedProps = await dehydrateProps(props);
        postMessage({ action: 'dehydrated', dehydratedProps });
    }
});

async function hydrateProps(props) {
    return { ...props, hydrated: true };
}

async function dehydrateProps(props) {
```

```
    return { ...props, hydrated: false };
}
```

service-worker.js

```
import {saveStaticIsland, getStaticIsland, openDatabase} from './indexeddb.js'

self.addEventListener('install', async (event) => {
    const db = await openDatabase();
});

self.addEventListener('fetch', async (event) => {
    const url = new URL(event.request.url);

    if (event.request.method === 'GET') {
        const staticIsland = await getStaticIsland(url.pathname);

        if (staticIsland) {
            // Return cached island HTML
            event.respondWith(
                new Response(staticIsland.content, {
                    headers: { 'Content-Type': 'text/html' },
                })
```

```
            );
            return;
        }
    }

    event.respondWith(
        caches.match(event.request).then((cachedResponse) => {
            return cachedResponse || fetch(event.request);
        })
    );
});

self.addEventListener('message', async (event) => {
    if (event.data.action === 'cache-static-island') {
        const { url, content } = event.data;
        await saveStaticIsland(url, content);
    }
});
```

island-autoinit.js

```
class IslandAutoinit {
    static attr = {
      autoInitType: "autoinit",
```

```
    import: "import",
  }

  static types = {
    "petite-vue": function(lib) {
      lib.createApp().mount(this);
    },
    "vue": function(lib) {
      lib.createApp().mount(this);
    },
    "svelte": function(lib) {
      new lib.default({ target: this });
    },
    "svelte-ssr": function(lib) {
      new lib.default({ target: this, hydrate: true });
    },
    "preact": function(lib) {
      lib.default(this);
    }
  };
}

Island.onReady.set("import-autoinit", async function(Island) {
  let mod;
 let importScript = this.getAttribute(IslandAutoinit.attr.import);
 if(importScript) {
URL in your script's import anyway
  mod = await import(importScript);
 }
```

```
  if(mod)                                                                    {
    let  fn  =  IslandAutoinit.types[this.getAttribute(IslandAutoinit.attr.autoInitType)  ||
importScript];
    if(fn)                                                                   {
      await                                fn.call(this,                     mod);
    }
  }
})
```

indexeddb.js

```
export          async          function          openDatabase()          {
  return          new          Promise((resolve,          reject)          =>          {
    const          request          =          indexedDB.open('IslandCacheDB',          1);

    request.onupgradeneeded          =          (event)          =>          {
      const          db          =          event.target.result;
      if          (!db.objectStoreNames.contains('staticIslands'))          {
        db.createObjectStore('staticIslands',          {          keyPath:          'url'          });
      }
    };

    request.onsuccess          =          (event)          =>          resolve(event.target.result);
    request.onerror          =          (event)          =>          reject(event.target.error);
  });
}

export          async          function          saveStaticIsland(url,          content)          {
```

```
  const db = await openDatabase();
  const transaction = db.transaction('staticIslands', 'readwrite');
  const store = transaction.objectStore('staticIslands');
  store.put({ url, content });

  return transaction.complete;
}

export async function getStaticIsland(url) {
  const db = await openDatabase();
  return new Promise((resolve, reject) => {
    const transaction = db.transaction('staticIslands', 'readonly');
    const store = transaction.objectStore('staticIslands');
    const request = store.get(url);

    request.onsuccess = (event) => resolve(event.target.result);
    request.onerror = (event) => reject(event.target.error);
  });
}
```

island.js

```
import { saveStaticIsland, getStaticIsland } from '/indexeddb.js';

class Island extends HTMLElement {
 static tagName = "is-land";
 static prefix = "is-land--";
 static attr = {
  template: "data-island",
  ready: "ready",
```

```
  defer:                                                    "defer-hydration",
  static:                                                   "data-static"
 };

 static onceCache = new Map();
 static onReady = new Map();

 static fallback = {
  ":not(is-land,:defined,[defer-hydration])": (readyPromise, node, prefix) => {
   let cloned = document.createElement(prefix + node.localName);
   for(let attr of node.getAttributeNames()) {
    cloned.setAttribute(attr, node.getAttribute(attr));
   }

   let shadowroot = node.shadowRoot;
   if(!shadowroot) {
    let tmpl = node.querySelector(":scope > template:is([shadowrootmode], [shadowroot])");
    if(tmpl) {
     let mode = tmpl.getAttribute("shadowrootmode") || tmpl.getAttribute("shadowroot") || "closed";
     shadowroot = node.attachShadow({ mode }); // default is closed
     shadowroot.appendChild(tmpl.content.cloneNode(true));
    }
   }

   if(shadowroot) {
    cloned.attachShadow({ mode: shadowroot.mode }).append(...shadowroot.childNodes);
```

```
    }

    cloned.append(...node.childNodes);
    node.replaceWith(cloned);

    return readyPromise.then(() => {
     // Restore original children and shadow DOM
     if(cloned.shadowRoot) {
       node.shadowRoot.append(...cloned.shadowRoot.childNodes);
     }
     node.append(...cloned.childNodes);
     cloned.replaceWith(node);
    });
  }
}

constructor() {
  super();

  if (navigator.serviceWorker) {
    navigator.serviceWorker.register('/service-worker.js').then((reg) => {
      this.sw = reg;
    });
  }

  this.worker = new Worker('/island-worker.js');
  this.worker.onmessage = (event) => {
    const { action, hydratedProps, dehydratedProps } = event.data;
    if (action === 'hydrated') {
```

```
    this.props = hydratedProps;
    this.dispatchEvent(new CustomEvent('hydrated', { detail: hydratedProps }));
   }
   if (action === 'dehydrated') {
    this.props = dehydratedProps;
    this.dispatchEvent(new CustomEvent('dehydrated', { detail: dehydratedProps }));
   }
  };

  this.ready = new Promise(resolve => {
   this.readyResolve = resolve;
  });
}

static getParents(el, stopAt = false) {
  let nodes = [];
  while(el) {
   if(el.matches && el.matches(Island.tagName)) {
    if(stopAt && el === stopAt) {
     break;
    }

    if(Conditions.hasConditions(el)) {
     nodes.push(el);
    }
   }
   el = el.parentNode;
  }
  return nodes;
```

```
}

cacheStaticIsland(content) {
  if (this.sw) {
    this.sw.active.postMessage({
      action: 'cache-static-island',
      content,
      url: window.location.href,
    });
  }
}

static async ready(el, parents) {
  if(!parents) {
    parents = Island.getParents(el);
  }
  if(parents.length === 0) {
    return;
  }
  let imports = await Promise.all(parents.map(p => p.wait()));
  if(imports.length) {
    return imports[0];
  }
}

forceFallback() {
  if(window.Island) {
    Object.assign(Island.fallback, window.Island.fallback);
  }
```

```
  for(let selector in Island.fallback) {
   let components = Array.from(this.querySelectorAll(selector)).reverse();

   // with thanks to https://gist.github.com/cowboy/938767
   for(let node of components) {
    if(!node.isConnected) {
     continue;
    }

    let parents = Island.getParents(node);
    // must be in a leaf island (not nested deep)
    if(parents.length === 1) {
     let p = Island.ready(node, parents);
     Island.fallback[selector](p, node, Island.prefix);
    }
   }
  }
}

wait() {
 return this.ready;
}

async connectedCallback() {
 const isStatic = this.hasAttribute(Island.attr.static);

 if (isStatic) {
  const url = window.location.href;
```

```
    const existingIsland = await getStaticIsland(url);

    if (!existingIsland) {
      const content = this.outerHTML;
      await saveStaticIsland(url, content);
      this.cacheStaticIsland(content);
    }
  }

  if(Conditions.hasConditions(this)) {
    this.forceFallback();
  }

  await this.hydrate();
}

getTemplates() {
  return this.querySelectorAll(`template[${Island.attr.template}]`);
}

replaceTemplates(templates) {
  for(let node of templates) {
    if(Island.getParents(node, this).length > 0) {
      continue;
    }

    let value = node.getAttribute(Island.attr.template);
    if(value === "replace") {
      let children = Array.from(this.childNodes);
```

```
    for(let child of children) {
      this.removeChild(child);
    }
    this.appendChild(node.content);
    break;
  } else {
    let html = node.innerHTML;
    if(value === "once" && html) {
      if(Island.onceCache.has(html)) {
        node.remove();
        return;
      }

      Island.onceCache.set(html, true);
    }

    node.replaceWith(node.content);
  }
 }
}

async hydrate() {
 let conditions = [];
 if(this.parentNode) {
  // wait for all parents before hydrating
  conditions.push(Island.ready(this.parentNode));
 }

 let attrs = Conditions.getConditions(this);
```

```
    for(let condition in attrs) {
      if(Conditions.map[condition]) {
        conditions.push(Conditions.map[condition](attrs[condition], this));
      }
    }

    await Promise.all(conditions);

    this.replaceTemplates(this.getTemplates());

    for(let fn of Island.onReady.values()) {
      await fn.call(this, Island);
    }

    this.readyResolve();

    this.setAttribute(Island.attr.ready, "");

    this.querySelectorAll(`[${Island.attr.defer}]`).forEach(node =>
node.removeAttribute(Island.attr.defer));
  }
}

class Conditions {
  static map = {
    visible: Conditions.visible,
    idle: Conditions.idle,
    interaction: Conditions.interaction,
    media: Conditions.media,
```

```
  "save-data": Conditions.saveData,
};

static hasConditions(node) {
  return Object.keys(Conditions.getConditions(node)).length > 0;
}

static getConditions(node) {
  let map = {};
  for(let key of Object.keys(Conditions.map)) {
    if(node.hasAttribute(`on:${key}`)) {
      map[key] = node.getAttribute(`on:${key}`);
    }
  }

  return map;
}

static visible(noop, el) {
  if(!('IntersectionObserver' in window)) {
    return;
  }

  return new Promise(resolve => {
    let observer = new IntersectionObserver(entries => {
      let [entry] = entries;
      if(entry.isIntersecting) {
        observer.unobserve(entry.target);
        resolve();
```

```
      }
    });

    observer.observe(el);
  });
}

static idle() {
  let onload = new Promise(resolve => {
    if(document.readyState !== "complete") {
      window.addEventListener("load", () => resolve(), { once: true });
    } else {
      resolve();
    }
  });

  if(!("requestIdleCallback" in window)) {
    return onload;
  }

  return Promise.all([
    new Promise(resolve => {
      requestIdleCallback(() => {
        resolve();
      });
    }),
    onload,
  ]);
}
```

```
static interaction(eventOverrides, el) {
 let events = ["click", "touchstart"];
 if(eventOverrides) {
  events = (eventOverrides || "").split(",").map(entry => entry.trim());
 }

 return new Promise(resolve => {
  function resolveFn(e) {
   resolve();

   for(let name of events) {
    el.removeEventListener(name, resolveFn);
   }
  }

  for(let name of events) {
   el.addEventListener(name, resolveFn, { once: true });
  }
 });
}

static media(query) {
 let mm = {
  matches: true
 };

 if(query && ("matchMedia" in window)) {
  mm = window.matchMedia(query);
```

```
    }

    if(mm.matches) {
      return;
    }

    return new Promise(resolve => {
      mm.addListener(e => {
        if(e.matches) {
          resolve();
        }
      });
    });
  }

  static saveData(expects) {
    if(!("connection" in navigator) || navigator.connection.saveData === (expects !==
"false")) {
      return;
    }

    return new Promise(() => {});
  }
}

if("customElements" in window) {
  window.customElements.define(Island.tagName, Island);
  window.Island = Island;
```

```
 if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js').then((registration) => {
   console.log('Service Worker registered:', registration);
  });
 }
}

export {
 Island,
 Island as component, // Backwards compat only: recommend `Island` export
};

export const ready = Island.ready; // Backwards compat only: recommend `Island` export
```

elevenity.cjs

```
const EleventySveltePlugin = require("./11ty/SveltePlugin.cjs");
const EleventyVuePlugin = require("./11ty/VuePlugin.cjs");
const EleventyPreactPlugin = require("./11ty/PreactPlugin.cjs");
const EleventyIslandMarkdownPlugin = require("./11ty/MarkdownPlugin.cjs");

module.exports = function(eleventyConfig) {
 eleventyConfig.setQuietMode(true);
 eleventyConfig.addPassthroughCopy("*.js");

 eleventyConfig.setServerOptions({
  domdiff: false,
 });

 eleventyConfig.addPlugin(EleventySveltePlugin);
```

```
  eleventyConfig.addPlugin(EleventyVuePlugin);
  eleventyConfig.addPlugin(EleventyPreactPlugin);
  eleventyConfig.addPlugin(EleventyIslandMarkdownPlugin);

  eleventyConfig.addGlobalData("permalink", () => {
    return (data) => `${data.page.filePathStem}.${data.page.outputFileExtension}`;
  });

  return {
    dir: {
      input: "index.html",
    },
    htmlTemplateEngine: "liquid",
    markdownTemplateEngine: "liquid",
  }
};
```

elevenity-auto-import.js

```
const fs = require("fs");
const path = require("path");

module.exports = function(eleventyConfig) {
  eleventyConfig.setQuietMode(true);
  const pluginsDir = path.join(__dirname, "11ty");
  fs.readdirSync(pluginsDir).forEach(file => {
    if (file.endsWith(".plugin.cjs")) {
      const pluginPath = path.join(pluginsDir, file);
      const plugin = require(pluginPath);
```

```
        eleventyConfig.addPlugin(plugin);
    }
  });
  eleventyConfig.addPassthroughCopy("lib/**/*.{css,png,svg,js}");
  eleventyConfig.addPassthroughCopy("/**/*.{css,js}");
  eleventyConfig.addPassthroughCopy({
    "**/*.worker.js": "workers/"
  });
  eleventyConfig.setServerOptions({
    domdiff: false,
  });
  eleventyConfig.ignores.add("README.md");
  eleventyConfig.addGlobalData("permalink", () => {
    return (data) => `${data.page.filePathStem}.${data.page.outputFileExtension}`;
  });

  return {
    dir: {
      input: "index",
      output: "_site",
      includes: "_includes",
      data: "_data",
    },
    htmlTemplateEngine: "liquid",
    markdownTemplateEngine: "liquid",
  };
};
```

server-side.js

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

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

const app = express();
const port = 3000;

const componentCache = {};

function readComponent(filePath) {
  return new Promise((resolve, reject) => {
    fs.readFile(filePath, 'utf8', (err, data) => {
      if (err) reject(err);
      else resolve(data);
    });
  });
}


async function getComponent(componentName) {
  if (componentCache[componentName]) {
    console.log(`Serving component "${componentName}" from cache.`);
    return componentCache[componentName];
  }

  const htmlPath = path.join(__dirname, 'components', `${componentName}.html`);
  const jsPath = path.join(__dirname, 'components', `${componentName}.js`);

  try {
    if (fs.existsSync(htmlPath)) {
```

```
        console.log(`Reading component "${componentName}" from HTML file.`);
        const html = await readComponent(htmlPath);
        componentCache[componentName] = html;
        return html;
      }

      if (fs.existsSync(jsPath)) {
        console.log(`Reading component "${componentName}" from JS file.`);
        const js = await readComponent(jsPath);
        componentCache[componentName] = js;
        return js;
      }

      throw new Error(`Component "${componentName}" not found.`);
    } catch (error) {
      throw new Error(`Error loading component "${componentName}": ${error.message}`);
    }
}

app.get('/component/:name', async (req, res) => {
  const componentName = req.params.name;

  try {
    const component = await getComponent(componentName);
    res.send(component);
  } catch (error) {
    res.status(404).send(`Component "${componentName}" not found.`);
  }
```

```
});

app.get('/', async (req, res) => {
  try {
    const header = await getComponent('header');
    const footer = await getComponent('footer');
    const dynamicContent = `<div><p>Dynamic Content: ${new Date().toLocaleTimeString()}</p></div>`;

    const html = `
  <!DOCTYPE html>
  <html lang="en">
  <head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Dynamic Page</title>
  </head>
  <body>
    ${header}
    <main>
      <h1>Welcome to the dynamic page!</h1>
      ${dynamicContent}
    </main>
    ${footer}
  </body>
  </html>
  `;

    res.send(html);
```

```
  } catch (error) {
    res.status(500).send(`Error rendering page: ${error.message}`);
  }
});

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

FileTarget.cjs

```
const fs = require("fs");
const path = require("path");
const { ImportTransformer } = require("esm-import-transformer");

class FileTarget {
 constructor(filename) {
  this.filename = filename;
  this.parsed = path.parse(filename);
  this.suffix = ".client";
 }

 setSuffix(suffix) {
  this.suffix = suffix;
 }

 getFilename() {
  return this.parsed.name + this.suffix + ".js";
 }
```

```
  getImportUrl() {
    return "/" + path.join(this.parsed.dir, this.getFilename());
  }

  getOutputFile() {

    return path.join("./_site", this.parsed.dir, this.getFilename());
  }

  augmentContent(content, importMap) {
    let transformer = new ImportTransformer();
    let code = transformer.transform(content, importMap);
    return code;
  }

  write(content, importMap) {
    if(importMap) {
      content = this.augmentContent(content, importMap);
    }

    let outputFile = this.getOutputFile();
    let {dir} = path.parse(outputFile);
    fs.mkdirSync(dir, {recursive: true});

    fs.writeFileSync(outputFile, content, "utf8");
  }
}

module.exports = FileTarget;
```

MarkdownPlugin.cjs

```
const markdownIt = require("markdown-it");

module.exports = function(eleventyConfig) {
 let md = markdownIt({ html: true })

 md.disable('code');

 const proxy = (tokens, idx, options, env, self) => self.renderToken(tokens, idx, options);
 const defaultRenderer = md.renderer.rules.html_block || proxy;

 function contentCheck(token) {
  return token.content.trim().startsWith("<is-land ") || token.content.trim().startsWith("<is-land>");
 }

 md.renderer.rules.html_block = function(tokens, idx, options, env, self) {
  tokens = tokens.map((token, index) => {
   if(token.type === "paragraph_open") {
    if(contentCheck(tokens[index + 1])) {
     token.type = "html_block";
     token.tag = "";
    }
   }
   if(token.type === "paragraph_close") {
    if(contentCheck(tokens[index - 1])) {
     token.type = "html_block";
     token.tag = "";
```

```
      }
    }

    if(token.content.startsWith("<is-land ")) {
      token.type = "html_block";
      token.children = null;
      token.level = 0;
      token.content = `\n${token.content}\n\n`;
    }
    return token;
  });

  return defaultRenderer(tokens, idx, options, env, self)
 };

 eleventyConfig.setLibrary("md", md);
};
```

PreactPlugin.cjs

```
const fs = require("fs");
const path = require("path");
const render = require('preact-render-to-string');

const FileTarget = require("./FileTarget.cjs");

module.exports = function(eleventyConfig) {
 eleventyConfig.addFilter("preact", async (filename) => {
  let content = fs.readFileSync(filename, "utf8");
  let target = new FileTarget(filename);
```

```
  target.write(content, {
    imports: {
      "htm/preact": "https://unpkg.com/htm/preact/index.mjs?module"
    }
  });

  let appDefinition = await import(path.join("../", filename));
  let output = render(appDefinition.default());

  return {
    html: output,
    clientJsUrl: target.getImportUrl(),
  }
 });
};
```

Svelte.component.cjs

```
const fs = require("fs");
const path = require("path");
const { Module } = require("module");
const svelte = require('svelte/compiler');

const FileTarget = require("./FileTarget.cjs");

class EleventySvelteComponent {
 constructor(filename, options = { ssr: true }) {
  this.filename = filename;
  this.parsed = path.parse(filename);
  this.content = fs.readFileSync(filename, "utf8");
```

```
  this.isSSR = options.ssr;

  this.clientImportMap = {
    imports: {
      "svelte/internal": "https://unpkg.com/svelte@3.48.0/internal/index.mjs"
    }
  };
}

generateClientBundle() {
  let options = {
    filename: this.filename,
    generate: "dom",
    hydratable: this.isSSR ? true : false,
    css: this.isSSR ? false : true,
  };

  let component = svelte.compile(this.content, options);

  let target = new FileTarget(this.filename);
  target.write(component.js.code, this.clientImportMap);
  return target;
}

renderOnServer() {
  if(!this.isSSR) {
    return {
      html: "",
      css: "",
```

```
    };
  }

  let component = svelte.compile(this.content, {
    filename: this.filename,
    generate: "ssr",
    format: "cjs",
  });

  let m = new Module();
  m.paths = module.paths;
  m._compile(component.js.code, this.filename);

  let fn = m.exports.default.render;
  let result = fn();
  return {
    html: result.html.replace(/\n/g, ""),
    css: result.css.code
  };
}

get() {
  let target = this.generateClientBundle();
  let {html, css} = this.renderOnServer();

  return {
    html: html.replace(/\n/g, ""),
    css,
    clientJsUrl: target.getImportUrl()
```

```
    };
  }
}

module.exports = EleventySvelteComponent;
```

SveltePlugin.cjs

```
const EleventySvelteComponent = require("./SvelteComponent.cjs");

class CssManager {
  constructor() {
    this.reset();
  }

  reset() {
    this.pages = {};
  }

  addPageCss(pageUrl, css) {
    if(!this.pages[pageUrl]) {
      this.pages[pageUrl] = new Set();
    }
    this.pages[pageUrl].add(css);
  }

  getCssForPage(pageUrl) {
    return Array.from(this.pages[pageUrl] || []).join("\n");
  }
}
```

```
module.exports = function(eleventyConfig) {
  let pageCss = new CssManager();
  eleventyConfig.on("eleventy.before", () => pageCss.reset());
  eleventyConfig.addFilter("getSvelteCss", pageUrl =>
pageCss.getCssForPage(pageUrl));

  eleventyConfig.addFilter("svelte", async function(filename, pageUrl) {
    let ssr = !!pageUrl;

    let c = new EleventySvelteComponent(filename, {
      ssr,
    });

    let component = c.get();

    if(pageUrl) {
      pageCss.addPageCss(pageUrl, component.css);
    }

    return component;
  });
};
```

Vue.plugin.cjs

```
const path = require("path");
const { createSSRApp } = require('vue');
const { renderToString } = require('vue/server-renderer');

const FileTarget = require("./FileTarget.cjs");
```

```
module.exports = function(eleventyConfig) {

  eleventyConfig.addFilter("vue", async (filename) => {
    let target = new FileTarget(filename);
    target.setSuffix("");

    let appDefinition = await import(path.join("../", filename));
    let app = createSSRApp(appDefinition.default);
    let html = await renderToString(app);

    return {

      clientJsUrl: target.getImportUrl(),
    }
  });
};
```

Package.json

```
{
  "name": "island",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "type": "module",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
```

```
  "license": "ISC",
  "dependencies": {
    "@11ty/eleventy": "^3.0.0",
    "htm": "^3.1.1",
    "esm-import-transformer": "^1.0.1",
    "preact": "^10.11.3",
    "preact-render-to-string": "^5.2.6",
    "svelte": "^3.55.1",
    "vue": "^3.2.45"
  }
}
```

ДОДАТОК Г

Ілюстративна частина

ІЛЮСТРАТИВНА ЧАСТИНА

УДОСКОНАЛЕННЯ МЕТОДІВ І ЗАСОБІВ ОСТРІВНОЇ АРХІТЕКТУРИ ПОБУДОВИ ВЕБ ДОДАТКІВ

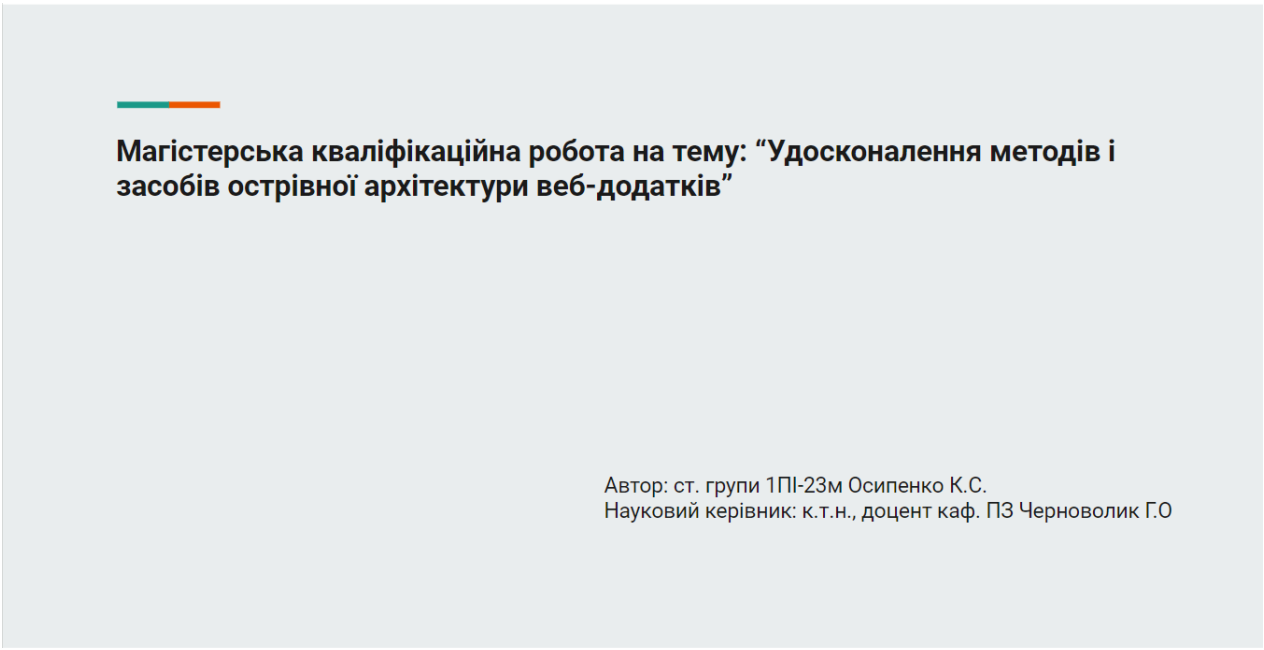


Рисунок Г.1 – Назва роботи

Рисунок Г.2 – Актуальність

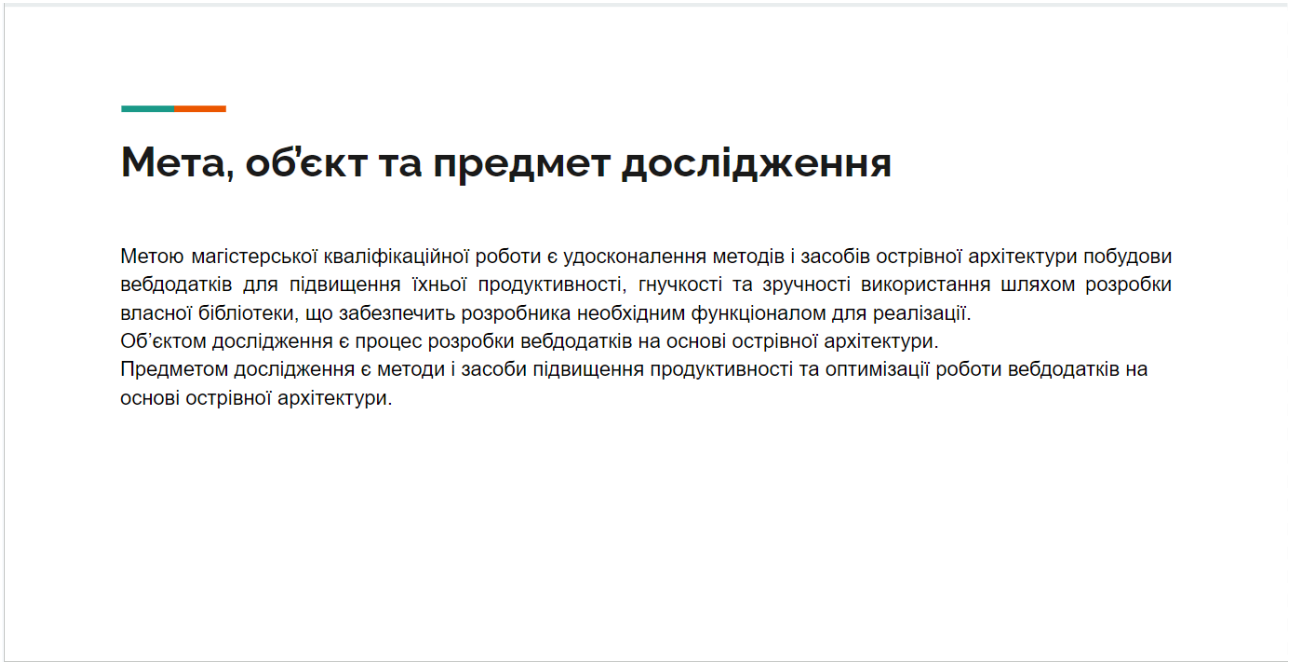


Рисунок Г.3 – Мета, об’єкт та предмет дослідження

Рисунок Г.4 – Завдання дослідження

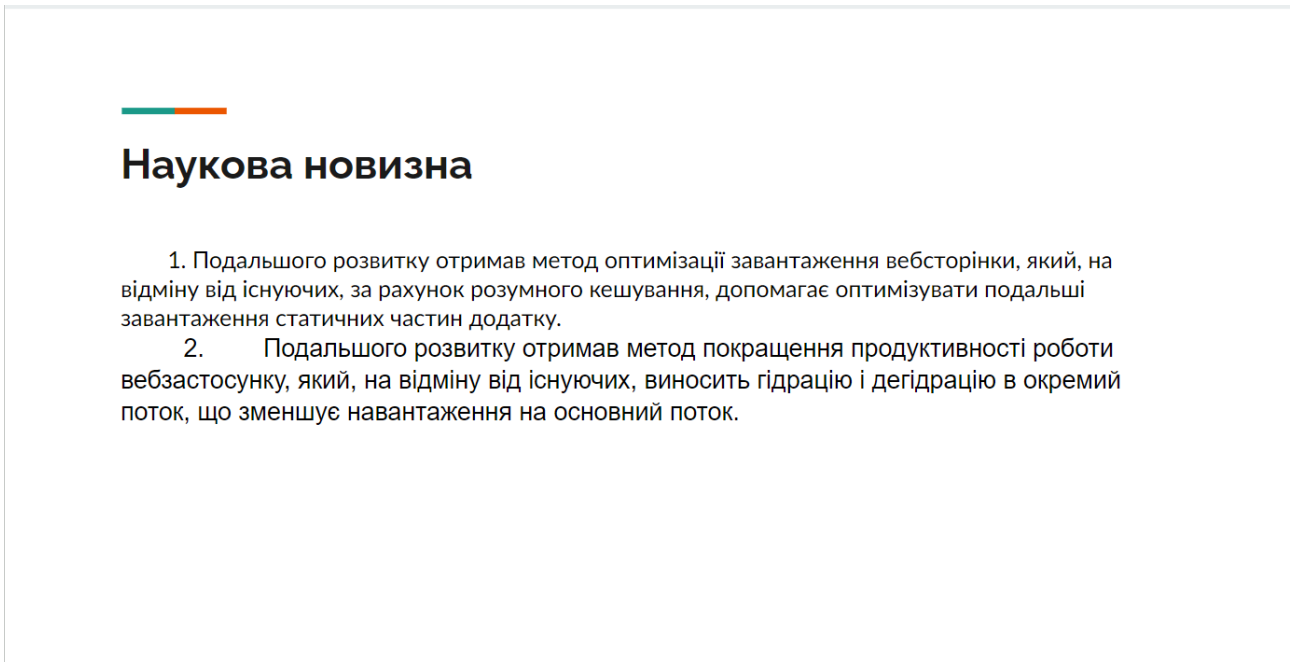


Рисунок Г.5 – Наукова новизна

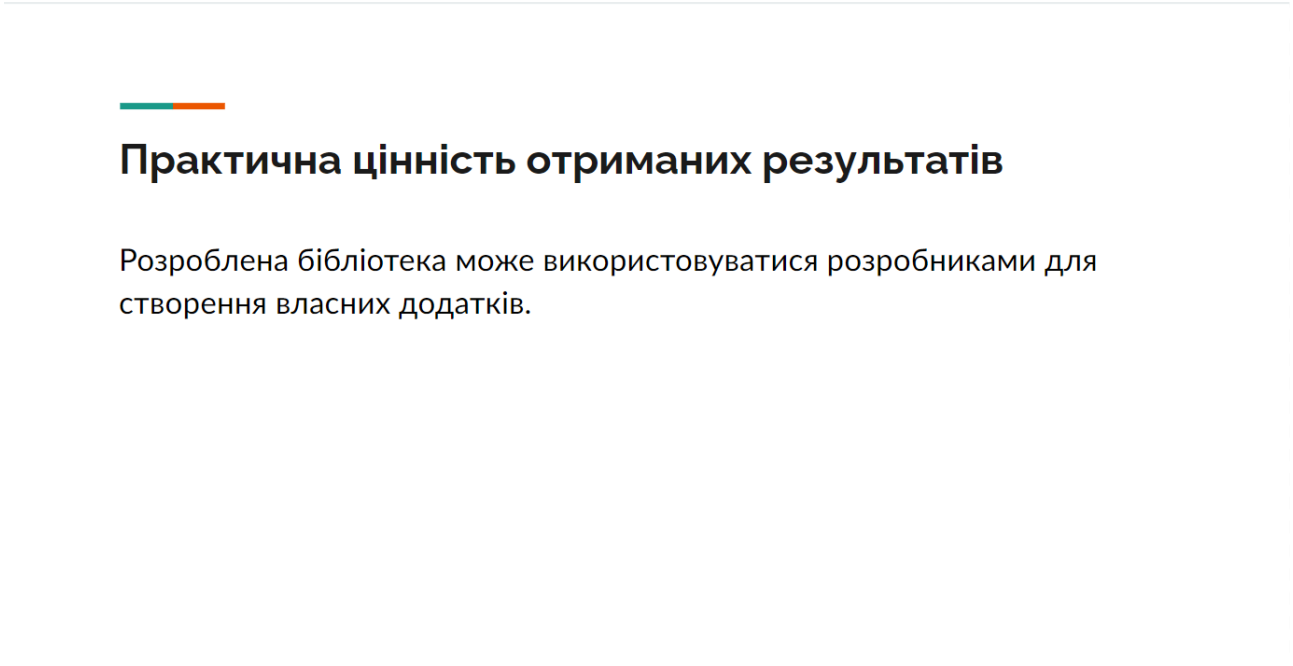


Рисунок Г.6 – Практична цінність отриманих результатів

Аналоги

Рисунок Г.7 – Аналоги

Порівняльний аналіз аналогів

	Astro	Qwik	Blazor	Власна розробка
Підтримка острівної архітектури	+	-	-	+
Гідрація	+	+	-	+
Розвинута екосистема	-	-	+	-
Система кешування	-	-	-	+

Рисунок Г.8 – Порівняльний аналіз аналогів

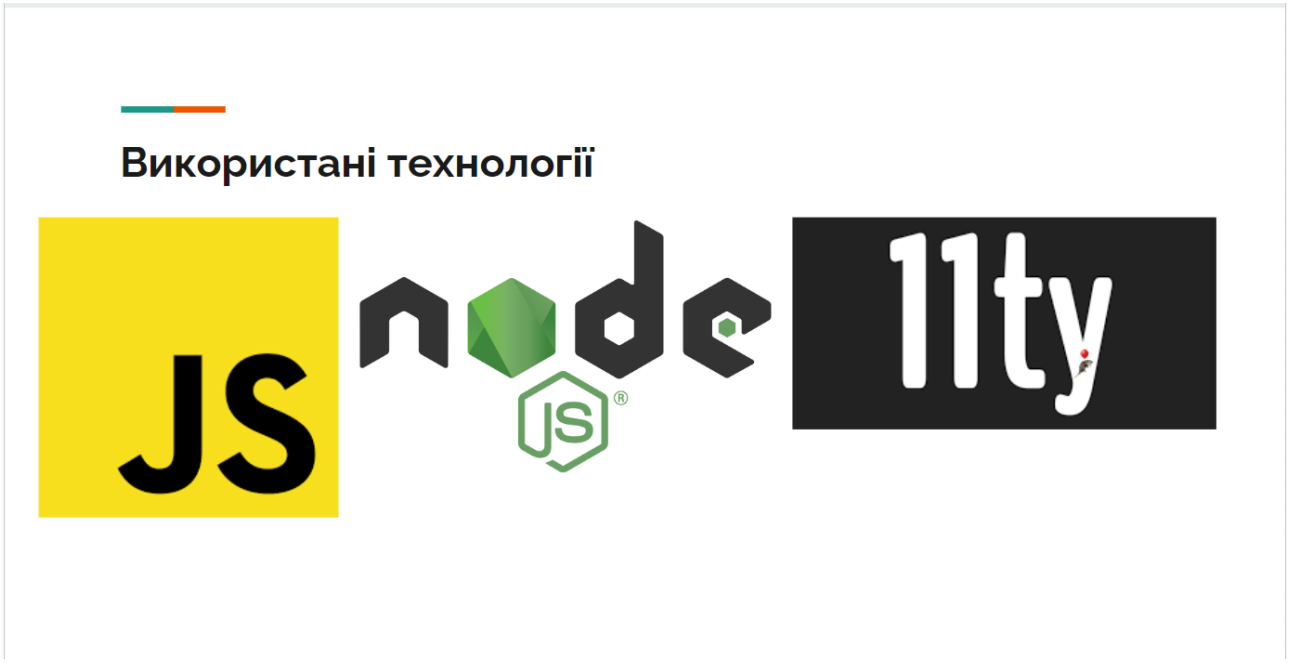


Рисунок Г.9 – Використані технології

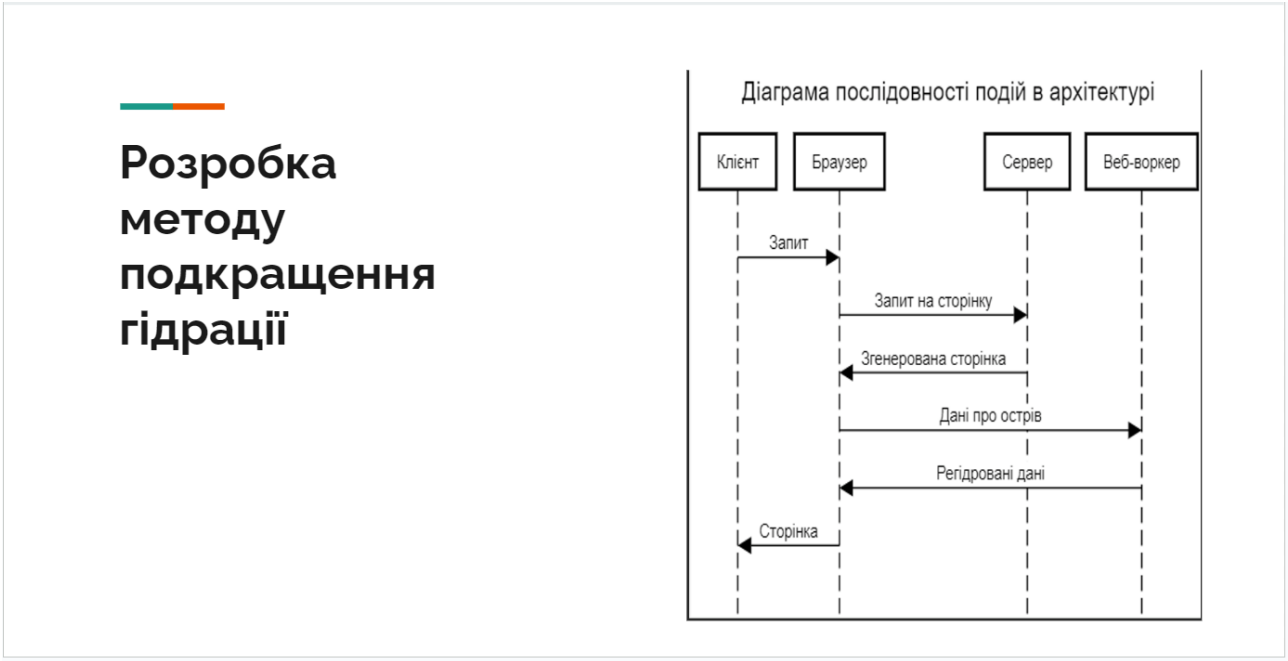


Рисунок Г.10 – Розробка методу покращення гідрації

Рисунок Г.11 – Розробка методу оптимізації завантаження сторінки

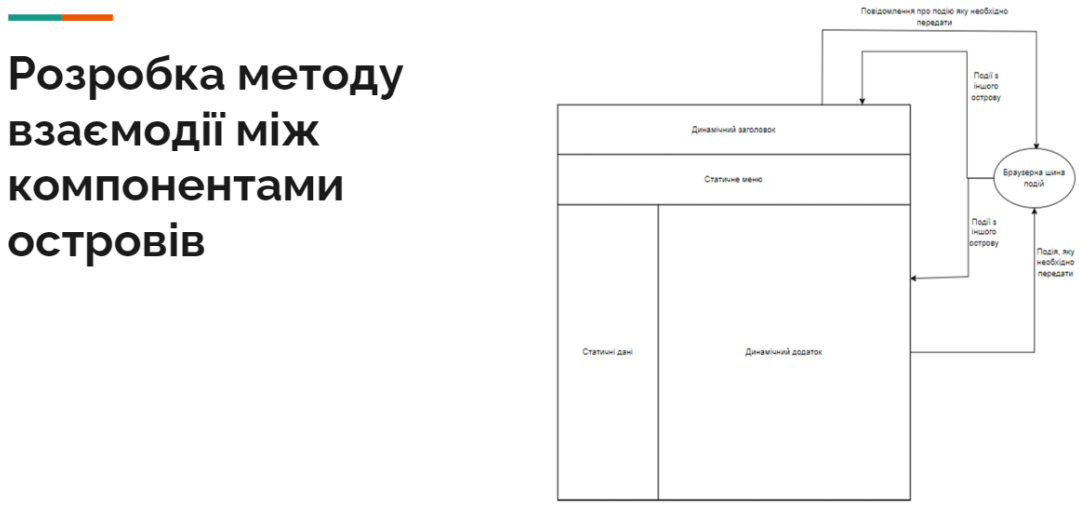


Рисунок Г.12 – Розробка методу взаємодії між компонентами островів

Рисунок Г.13 – Розробка методу покращення продуктивності та масштабованості

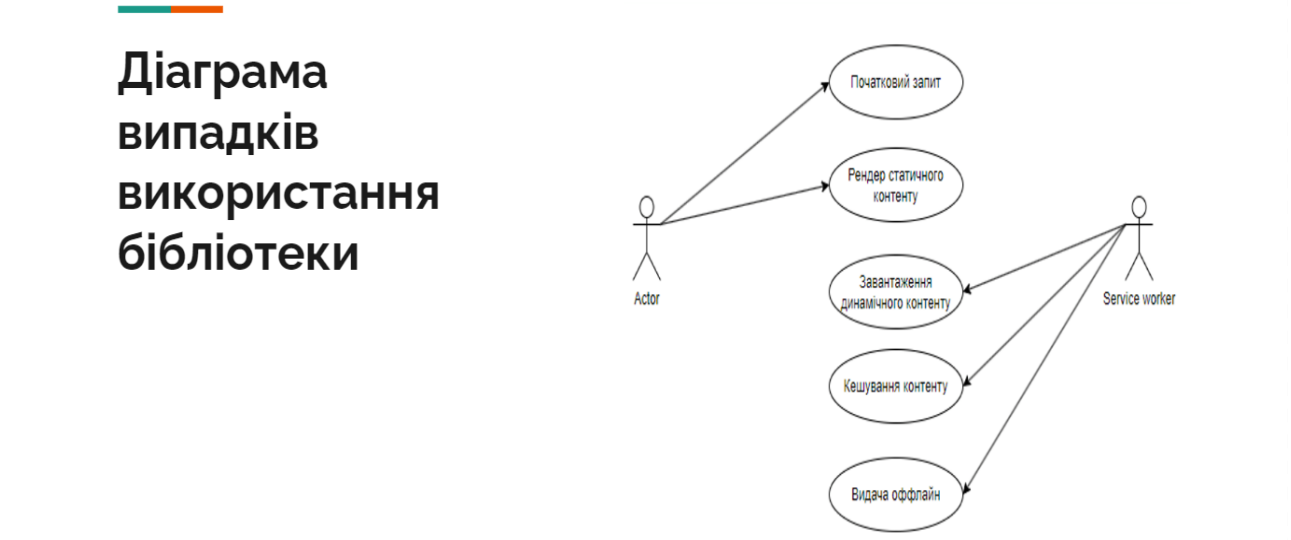


Рисунок Г.14 – Діаграма випадків використання бібліотеки

Діаграма послідовності бібліотеки

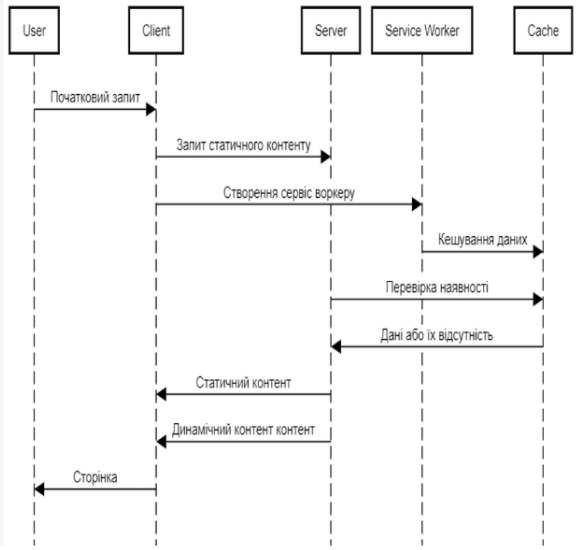


Рисунок Г.15 – Діаграма послідовності бібліотеки

Діаграма класів

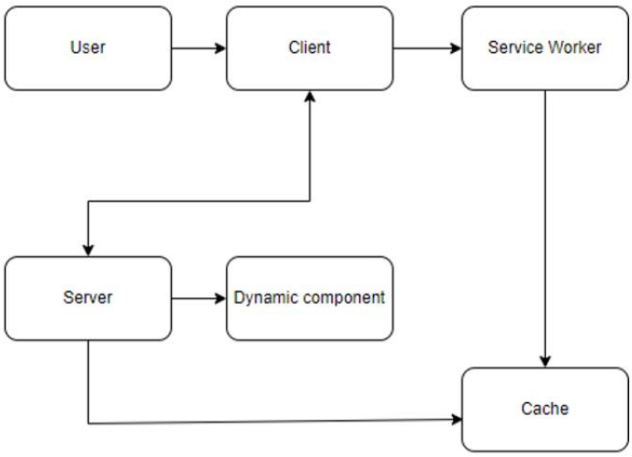


Рисунок Г.16 – Діаграма класів

Тестування системи

Граф завантаження сторінки без оптимізацій

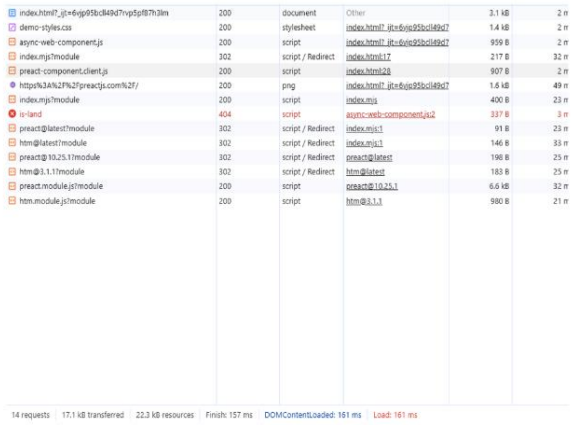


Рисунок Г.17 – Тестування системи (частина 1)

Тестування системи

Граф завантаження початкової частини сторінки з оптимізаціями

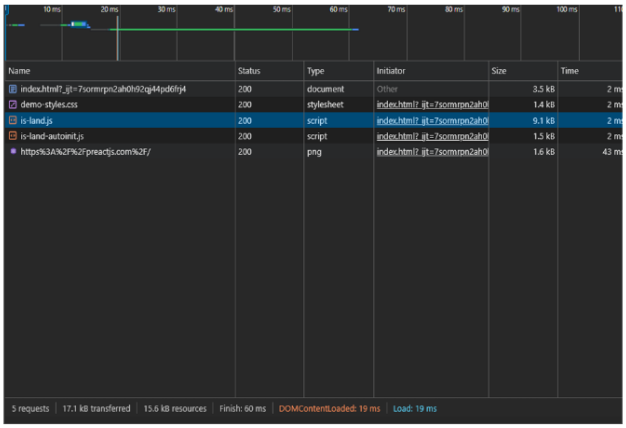


Рисунок Г.18 – Тестування системи (частина 2)

Тестування системи

Граф завантаження всієї сторінки з оптимізаціями

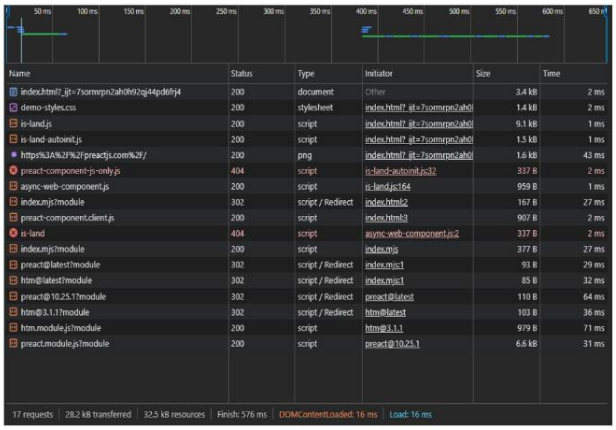


Рисунок Г.19 – Тестування системи (частина 3)

Демонстрація

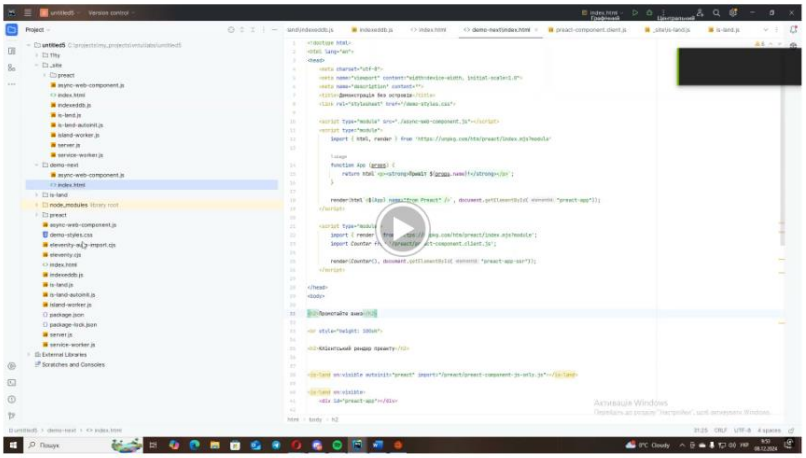


Рисунок Г.20 – Демонстрація роботи програми

Висновки

У ході виконання магістерської кваліфікаційної роботи було розроблено методи та програмні засоби для впровадження острівної архітектури вебдодатків. Робота була виконана та оформлена відповідно до методичних вказівок [33, 34].

Було проаналізовано сучасний стан питання острівної архітектури. Розглянуто основні існуючі підходи, фреймворки та бібліотеки, визначено їхні особливості та недоліки, а також проведено порівняння з розробленим програмним рішенням. У результаті аналізу було обрано мову програмування JavaScript, генератор статичних сайтів Eleventy, а також створено власну бібліотеку для підтримки острівної архітектури. Це дозволило досягти високої продуктивності та простоти реалізації компонентів.

Для реалізації системи було використано вебкомпоненти як базову одиницю острівної архітектури, що забезпечує ізоляцію логіки і стилів, а також можливість багаторазового використання незалежних модулів. Для оптимізації роботи системи було інтегровано ве-воркери для виконання ресурсомістких обчислень у фоновому режимі та реалізовано комбіноване кешування як на стороні клієнта, так і на стороні сервера.

Рисунок Г.21 – Висновки (частина 1)

Висновки

Було розроблено платформу для створення вебдодатків, яка включає механізми часткової та адаптивної гідрації компонентів, оптимізації завантаження сторінок через відкладене завантаження (lazy-loading) тапопереднє завантаження (preloading). Крім того, було впроваджено механізми взаємодії між "островами" через подієво-орієнтовану архітектуру, що дозволило забезпечити ефективний обмін даними між компонентами з мінімальними накладними витратами.

Розроблено бібліотеку для інтеграції острівної архітектури на основі Eleventy, що дозволяє легко створювати автономні компоненти для вебдодатків із високою продуктивністю. Основні модулі бібліотеки включають методи для гідрації компонентів, оптимізації завантаження ресурсів та механізми комунікації між компонентами.

Було також розроблено метод масштабування системи використання вебворкерів. Це забезпечило можливість гнучкого розширення системи у разі збільшення користувацького навантаження та підвищення вимог до продуктивності.

Рисунок Г.22 – Висновки (частина 2)

Висновки

Тестування розробленої системи показало її повну працездатність та відповідність поставленим технічним завданням. Розроблено інструкцію для розробників щодо інтеграції та використання острівної архітектури у вебдодатках на основі Eleventy.

У результаті роботи вдалося створити бібліотеку, що забезпечує високу швидкодію та гнучкість при розробці сучасних вебдодатків, завдяки чому було досягнуто поставлені цілі щодо покращення структури, продуктивності та можливості масштабування вебсистем на основі острівної архітектури.

Рисунок Г.23 – Висновки (частина 3)

Апробація результатів роботи і публікації

Апробація: Результати магістерської кваліфікаційної роботи обговорювалися на міжнародній науково-практичній Інтернет-конференції «Електронні інформаційні ресурси: створення, використання, доступ» (Суми/Вінниця, 2024).

Публікації: Войтко В.В. Удосконалення методу "острови" для підвищення швидкості роботи програмних застосунків у браузері / В.В. Войтко, Г.О. Черноволик, Н.Є. Барчук, О.В. Гаврилюк, К.С. Осипенко // Електронні інформаційні ресурси: створення, використання, доступ та управління. Збірник матеріалів Міжнародної науково-практичної Інтернет конференції 20-21 листопада 2024р. – Суми/Вінниця: НІКО / КЗВО «Вінницька академія безперервної освіти», 2024. – С. 31-34.

Рисунок Г.24 – Апробація результатів роботи і публікації

Дякую за увагу!

Рисунок Г.25 – Фінальний слайд