ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ
КАФЕДРА ПРОГРАМНИХ ЗАСОБІВ І ТЕХНОЛОГІЙ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи

бакалавра
(освітньо-кваліфікаційний рівень)

на тему *Оптимізація продуктивності Frontend-застосунків в рамках розробки фінтех проекту на Next.js*

Виконав: здобувач *4* року навчання
групи *4ПР1*
напряму підготовки (спеціальності)
121-Інженерія програмного забезпечення
(шифр і назва напряму підготовки, спеціальності)

Морозов О. Є.
(підпис, прізвище та ініціали)

Керівник *Доровський В. О.*
(підпис, прізвище та ініціали)

Рецензент *Корніловська Н.В*
(прізвище та ініціали)

Нормоконтролер *Комісаров О. С.*
(підпис, прізвище та ініціали)

Хмельницький – 2026

ХЕРСОНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Інститут, факультет, відділення *Інформаційних технологій та дизайну*
Кафедра, циклова комісія *Програмних засобів і технологій*
Освітньо-кваліфікаційний рівень *бакалавр*
Освітньо-професійна підготовка *Програмна інженерія*
(шифр і назва)
Спеціальність *121-Інженерія програмного забезпечення*
(шифр і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри *програмних засобів і технологій*
_____ О.Є. Огнєва
«___» _____ 2026 року

ЗАВДАННЯ

НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА

Морозову Олександру Євгеновичу
(прізвище, ім'я, по батькові)

1. Тема проєкту (роботи) *"Оптимізація продуктивності Frontend-застосунку в рамках розробки фінтех проекту на Next.js"*

керівник проєкту (роботи) *Доровський Володимир Олексійович, професор, д.т.н.*
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від «16» січня 2026 року №_82-с_

2. Строк подання здобувачем проєкту(роботи) 10.06.2026

3. Вихідні дані до проєкту (роботи) Літературні та періодичні джерела, матеріали переддипломної практики

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження та аналіз веб-додатків для фінтех сфери
2. Аналіз вимог та розробка проектних специфікацій
3. Проектування веб-додатку для автоматичної агрегації банківських даних
4. Розробка, тестування та робота з веб-додатком для автоматичної агрегації банківських даних

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)

6. Консультанти розділів проєкту (роботи)

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання *07.02.2026*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів проекту (роботи)	Примітка
1	Отримання завдання	07.02.2026	***Виконано***
2	Підбір літератури	09.02.2026- 20.02.2026	***Виконано***
3	Аналіз предметної області	21.02.2026- 27.02.2026	***Виконано***
4	Розробка та обґрунтування завдання	28.02.2026- 13.03.2026	***Виконано***
5	Розробка концептуальної моделі	14.03.2026- 20.03.2026	***Виконано***
6	Моделювання та проектування системи	21.03.2026- 03.04.2026	***Виконано***
7	Моделювання та проектування бази даних	04.04.2026- 10.04.2026	***Виконано***
8	Розробка інтерфейсу додатку	11.04.2026- 17.04.2026	***Виконано***
9	Тестування додатку	*18.04.2026-* *24.04.2026*	***Виконано***
10	Оформлення пояснювальної записки	*25.04.2026-* *08.05.2026*	***Виконано***
11	Захист кваліфікаційної роботи	*10.06.2026*	***Виконано***

Здобувач _____ *Морозов О. Є.*
(підпис) (прізвище та ініціали)

Керівник проекту (роботи) _____ *Доровський В. О.*
(підпис) (прізвище та ініціали)

РЕФЕРАТ

Кваліфікаційна робота бакалавра: 58 с., 11 табл., 31 джерело, 1 додаток.

Об'єкт дослідження — процеси оптимізації продуктивності frontend-застосунків, розроблених з використанням Next.js та React у фінтех-секторі.

Предмет дослідження — методи підвищення продуктивності (стратегії рендерингу, зменшення розміру бандла, ліниве завантаження, кешування) у фінтех-застосунках, створених на базі Next.js та React, з урахуванням реальних обмежень розробки.

Мета роботи — теоретично обґрунтувати та експериментально перевірити ефективність сучасних методів оптимізації frontend-продуктивності для фінтех-проектів на базі Next.js та React в умовах реальних обмежень розробки (технічний борг, легасі-код, часові та ресурсні обмеження).

У роботі проведено аналіз теоретичних засад продуктивності веб-застосунків та ключових метрик Core Web Vitals (LCP, FID, CLS, TTI, TBT). Розроблено математичні моделі впливу розміру JavaScript-бандла на час до інтерактивності (TTI) та стратегій рендерингу (CSR, SSR, SSG, ISR, Streaming SSR) на час завантаження сторінок. Проведено порівняльний аналіз фреймворків Next.js, Angular та Vue/Nuxt за критеріями продуктивності, SEO та зручності розробки. Виявлено та систематизовано ключові обмеження реальної frontend-розробки: технічний борг, легасі-код, організаційні та ресурсні обмеження, специфічні вимоги безпеки у фінтех-секторі (XSS, CSRF, захист токенів).

Реалізовано тестовий застосунок "FinDash" — типовий фінтех-дашборд на базі Next.js (App Router) з React 18, на якому послідовно впроваджувались методи оптимізації: стратегії рендерингу (SSG/ISR для курсів валют, SSR для профілю), видалення мертвого коду (tree shaking), ліниве завантаження компонентів (next/dynamic), кешування даних (React Query) та HTTP-кешування статичних ресурсів. Проведено вимірювання за допомогою Lighthouse,

WebPageTest та Bundle Analyzer зі статистичною обробкою результатів (серія з 10 вимірювань, довірчий інтервал 95%).

Практична значущість: розроблені рекомендації, пріоритетна матриця методів оптимізації та покроковий алгоритм впровадження можуть бути безпосередньо використані розробниками та технічними лідами комерційних frontend-застосунків. Запропонована економічна модель дозволяє обґрунтувати доцільність інвестицій в оптимізацію перед керівництвом: ROI перевищує 8000% при окупності менше одного тижня. Досягнуті результати — скорочення LCP на 58% (з 12.5 с до 5.2 с на мобільних пристроях із 3G), зменшення розміру JavaScript-бандла на 45% та скорочення TTFB при повторних відвідуваннях на 90% — підтверджують практичну ефективність запропонованого підходу в умовах реальних проектів.

КЛЮЧОВІ СЛОВА: ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ, NEXT.JS, REACT, ФІНТЕХ, SSR, SSG, ISR, ЛІНИВЕ ЗАВАНТАЖЕННЯ, КЕШУВАННЯ, ТЕХНІЧНИЙ БОРГ, CORE WEB VITALS.

АНОТАЦІЯ

Морозов О. Є. Оптимізація продуктивності frontend-застосунків на основі Next.js у фінтех-проєктах. — Кваліфікаційна робота бакалавра. Херсонський національний технічний університет, Хмельницький, 2026 р.

У роботі досліджено проблему низької продуктивності frontend-застосунків у фінансовому секторі. Проведено аналіз сучасних стратегій рендерингу (CSR, SSR, SSG, ISR), методів оптимізації розміру бандла та підходів до кешування, а також виявлено їхні переваги та недоліки в контексті реальної розробки з урахуванням технічного боргу та легасі-коду.

Розроблено математичні моделі впливу розміру JavaScript-бандла на час до інтерактивності (TTI) та стратегій рендерингу на час завантаження (LCP, FCP). Проведено порівняльний аналіз фреймворків Next.js, Angular та Vue/Nuxt за критеріями продуктивності, SEO та зручності розробки. Обґрунтовано вибір Next.js як основи для фінтех-проектів завдяки вбудованій підтримці ISR та гібридних стратегій рендерингу.

Реалізовано тестовий застосунок "FinDash" на базі Next.js (App Router, React 18) з послідовним впровадженням методів оптимізації: стратегій рендерингу, tree shaking, лінивого завантаження та кешування даних (React Query). Проведено вимірювання з використанням інструментів Lighthouse, WebPageTest та Bundle Analyzer зі статистичною обробкою результатів (10 вимірювань, довірчий інтервал 95%).

Результати роботи можуть бути використані розробниками та технічними лідами для підвищення продуктивності комерційних frontend-застосунків в умовах обмежених ресурсів. Практична значущість полягає у розробленій пріоритетній матриці методів оптимізації та покроковому алгоритмі впровадження, що дозволяє скоротити LCP на 58%, зменшити розмір JavaScript-бандла на 45% та досягти ROI понад 8000% при мінімальних трудозатратах.

КЛЮЧОВІ СЛОВА: ОПТИМІЗАЦІЯ ПРОДУКТИВНОСТІ, NEXT.JS, REACT, ФІНТЕХ, SSR, SSG, ISR, ЛІНИВЕ ЗАВАНТАЖЕННЯ, КЕШУВАННЯ.

ABSTRACT

Morozov Oleksandr Frontend Application Performance Optimization Using Next.js in Fintech Projects. — Bachelor's Qualification Thesis. Kherson National Technical University, Khmelnytskyi, 2026.

This thesis investigates the problem of insufficient frontend performance in financial web applications. An analysis of modern rendering strategies (CSR, SSR, SSG, ISR), JavaScript bundle size reduction techniques, and caching approaches has been conducted, with their advantages and trade-offs evaluated in the context of real-world development constraints including technical debt and legacy code.

Mathematical models describing the impact of JavaScript bundle size on Time to Interactive (TTI) and rendering strategies on loading metrics (LCP, FCP) have been developed. A comparative analysis of Next.js, Angular, and Vue/Nuxt frameworks was performed across performance, SEO, and developer experience criteria. Next.js was substantiated as the optimal choice for fintech projects due to its native support for ISR and hybrid rendering strategies.

A test application "FinDash" was implemented using Next.js (App Router, React 18) with incremental application of optimization methods: rendering strategies, tree shaking, lazy loading, and client-side data caching via React Query. Performance measurements were conducted using Lighthouse, WebPageTest, and Bundle Analyzer tools with statistical processing of results (10 measurements, 95% confidence interval).

The results of this work can be directly applied by developers and technical leads to improve performance of commercial frontend applications under resource constraints. The practical value lies in the developed priority matrix of optimization methods and a step-by-step implementation algorithm, enabling a 58% reduction in LCP, a 45% decrease in JavaScript bundle size, and an ROI exceeding 8,000% with minimal implementation effort.

KEYWORDS: PERFORMANCE OPTIMIZATION, NEXT.JS, REACT, FINTECH, SSR, SSG, ISR, LAZY LOADING, CACHING, TECHNICAL DEBT, CORE WEB VITALS.

ВСТУП

У сучасному цифровому середовищі швидкодія веб-застосунків перестала бути суто технічною характеристикою, перетворившись на один з визначальних факторів успішності бізнесу, особливо у висококонкурентному фінансовому секторі. Дослідження компаній Amazon, Google та Microsoft незмінно демонструють прямий зв'язок між часом завантаження сторінки та ключовими бізнес-метриками: затримка в 100 мілісекунд призводить до падіння конверсії на 1%, а збільшення часу завантаження з 1 до 3 секунд підвищує ймовірність відмови на 32% [1, 2]. Для фінтех-проектів, де користувачі здійснюють фінансові операції та очікують на миттєву реакцію системи, ці вимоги є особливо критичними [26]. За даними аналітичних звітів, середня конверсія у фінансовій сфері коливається від 2.5% до 5.2%, і оптимізація продуктивності є одним з найефективніших інструментів її підвищення [3].

Водночас, сучасна frontend-розробка стикається з низкою системних обмежень. Стрімке зростання JavaScript-екосистеми, поява численних фреймворків та бібліотек, а також практика повторного використання коду призвели до "розбухання" веб-застосунків. Значна частина JavaScript-коду, який завантажується та обробляється браузером, є "мертвим" (dead code) — тобто таким, що ніколи не використовується під час взаємодії користувача зі сторінкою [16]. Дослідження Muzeel показує, що половина з 300 000 проаналізованих JavaScript-файлів мають щонайменше 70% не використовуваних функцій, що становить 55% від їхнього розміру. Це не лише збільшує час завантаження, але й марно витрачає ресурси пристрою, що особливо критично для мобільних користувачів.

Крім того, розробники часто працюють в умовах жорстких обмежень: наявність застарілого коду (legacy), технічного боргу, стислих термінів та необхідності підтримки сумісності з великою кількістю залежностей [4, 28]. За оцінками експертів, до 87% бюджету IT-проекту може витрачатися на

обслуговування технічного боргу, а не на створення нової цінності [28]. Це створює розрив між високими бізнес-вимогами до продуктивності та реальними технічними можливостями команд розробників.

Саме тому пошук ефективних, прагматичних методів оптимізації, які можна інтегрувати в існуючі проекти з мінімальними витратами, є надзвичайно актуальним науково-прикладним завданням. Next.js, як фреймворк на основі React, пропонує широкий спектр вбудованих інструментів для вирішення цих проблем — різні стратегії рендерингу (SSR, SSG, ISR), ліниве завантаження, оптимізацію зображень та інтелектуальне кешування [22, 23, 27].

Мета дослідження полягає в теоретичному обґрунтуванні та експериментальній перевірці ефективності сучасних методів оптимізації frontend-продуктивності для фінтех-проектів, розроблених з використанням Next.js та React, з урахуванням реальних обмежень розробки (легасі-код, технічний борг, часові рамки).

Для досягнення поставленої мети необхідно вирішити наступні завдання:

1. Проаналізувати теоретичні засади продуктивності веб-застосунків, сучасні підходи до рендерингу (CSR, SSR, SSG, ISR) та їхній вплив на користувацький досвід та бізнес-метрики, особливо в контексті фінансових сервісів [1, 2, 26].

2. Розробити математичні моделі впливу розміру бандла та стратегій рендерингу на ключові метрики продуктивності.

3. Провести порівняльний аналіз Next.js з альтернативними фреймворками (Angular, Vue/Nuxt) за критеріями продуктивності, SEO та зручності розробки.

4. Виявити основні обмеження, з якими стикаються розробники в реальних проектах, зокрема проблеми технічного боргу, роботи з легасі-кодом, управління залежностями та безпеки у фінтех-секторі [4, 16, 28].

5. Розглянути та систематизувати методи оптимізації розміру бандла (видалення мертвого коду), лінивого завантаження (lazy loading) компонентів та стратегії кешування в контексті екосистеми Next.js/React.

6. Провести експериментальне дослідження на прикладі типового фінтех-застосунку (дашборд) зі статистичною обробкою результатів (серія з 10 вимірювань, довірчий інтервал 95%).

7. Оцінити вплив застосованих методів на ключові метрики продуктивності (LCP, FID, TTFB, розмір JavaScript-бандла) та розробити економічну модель впливу оптимізації на прибуток проекту.

8. Сформулювати практичні рекомендації та пріоритетну матрицю впровадження методів оптимізації в умовах обмежених ресурсів.

Об'єктом дослідження є процеси оптимізації продуктивності frontend-застосунків.

Предметом дослідження є методи підвищення продуктивності (рендеринг, зменшення бандла, ліниве завантаження, кешування) у фінтех-застосунках, створених на базі Next.js та React.

Для вирішення поставлених завдань у роботі використовуються такі методи: аналіз науково-технічної літератури та інтернет-джерел; математичне моделювання; порівняльний аналіз; експеримент; статистична обробка даних; вимірювання метрик продуктивності за допомогою інструментів Lighthouse, WebPageTest та Bundle Analyzer [15, 20]; аналіз та узагальнення отриманих експериментальних даних; економічне моделювання.

Наукова новизна одержаних результатів:

- Розроблено математичні моделі впливу розміру бандла та стратегій рендерингу на час до інтерактивності (TTI) та найбільший елемент контенту (LCP).

- Вперше проведено систематизацію методів оптимізації frontend-продуктивності з урахуванням специфічних обмежень реальної

розробки (наявність легасі-коду, технічного боргу, часових та ресурсних обмежень).

- Запропоновано пріоритетну матрицю впровадження методів оптимізації на основі співвідношення "складність/ефект".

- Розроблено економічну модель оцінки впливу оптимізації продуктивності на прибуток фінтех-проекту.

Отримані в ході дослідження результати та сформульовані на їх основі рекомендації можуть бути безпосередньо використані розробниками та технічними лідами для підвищення продуктивності комерційних frontend-застосунків. Запропонований підхід дозволяє визначити пріоритетні напрямки оптимізації, які дають максимальний ефект при мінімальних трудозатратах, що є критично важливим для команд, які працюють з існуючими кодовими базами та в умовах жорстких дедлайнів. Економічна модель дозволяє обґрунтувати доцільність інвестицій в оптимізацію перед керівництвом та замовниками.

Кваліфікаційна робота складається зі вступу, чотирьох розділів, висновків, списку використаних джерел (31 найменування) та додатків. Загальний обсяг роботи становить 53 сторінки.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ОПТИМІЗАЦІЇ ПРОДУКТИВНОСТІ ФРОНТЕНД-ЗАСТОСУНКІВ

1.1. Продуктивність веб-застосунків та її значення для фінтех-проектів

Продуктивність веб-застосунку — це комплексна характеристика, що визначає швидкість та ефективність його роботи з точки зору кінцевого користувача. У сучасній веб-розробці продуктивність вимірюється не одним, а набором метрик, які дають змогу оцінити різні аспекти взаємодії: від швидкості початкового завантаження до чуйності інтерфейсу під час виконання сценаріїв [7, 15].

Ключовим набором метрик, що визначають сучасні стандарти продуктивності, є Core Web Vitals — ініціатива компанії Google, спрямована на оцінку якості користувацького досвіду. Ці метрики безпосередньо впливають на ранжування сайтів у пошуковій видачі та, відповідно, на органічний трафік.

Largest Contentful Paint (LCP) — час завантаження найбільшого елемента контенту (зображення, відео, текстового блоку). Google рекомендує підтримувати LCP менше 2.5 секунд. Алгоритм обчислення LCP враховує розмір елемента та момент його появи в області перегляду.

First Input Delay (FID) — час від першої взаємодії користувача (клік, натискання клавіші) до моменту, коли браузер може на неї відреагувати. Оптимальний показник — менше 100 мілісекунд. FID виникає через те, що головний потік браузера зайнятий виконанням JavaScript. Формула розрахунку First Input Delay приведена в 1.1

$$FID = Time\_response - Time\_input \quad (1.1)$$

де $Time\_response$ — час, коли обробник події починає виконуватися, $Time\_input$ — час, коли користувач здійснює дію.

Cumulative Layout Shift (CLS) — метрика, що вимірює візуальну стабільність сторінки, тобто наскільки часто елементи інтерфейсу "перестрибують" під час завантаження [15]. CLS обчислюється як сума зсувів для кожного неочікуваного переміщення елемента. Формула розрахунку CLS приведена в 1.2

$$CLS = \Sigma(impact\_fraction * distance\_fraction) \quad (1.2)$$

де $impact\_fraction$ — частка області перегляду, яку займає елемент, distance_fraction — відстань, на яку він змістився.

Загальний час завантаження веб-сторінки можна представити як суму чотирьох основних компонентів. Формула вимірювання завантаження веб-сторінки приведена в 1.3

$$T\_total = T\_network + T\_parse + T\_execute + T\_render \quad (1.3)$$

де:

- $T\_network$ — час передачі даних мережею, залежить від розміру ресурсів (S) та швидкості з'єднання (V): T_network = S / V + RTT * n (RTT — round-trip time, n — кількість запитів)

- T_parse — час парсингу HTML, CSS та JavaScript

- $T\_execute$ — час виконання JavaScript-коду

- T_render — час формування та відображення DOM-дерева

Важливість метрик продуктивності для бізнесу важко переоцінити. Дослідження компаній Amazon та Google зафіксували пряму кореляцію між затримками завантаження та падінням конверсії: затримка в 100 мілісекунд коштувала Amazon 1% втрати продажів [1]. Інші дослідження показують, що збільшення часу завантаження з 1 до 3 секунд підвищує ймовірність відмови (bounce rate) на 32% [2].

У фінансовому секторі ці вимоги набувають особливого значення. Фінтех-проекти оперують чутливими даними та фінансовими транзакціями в режимі реального часу. Користувачі очікують не просто швидкої, а миттєвої

реакції системи, оскільки будь-яка затримка може сприйматися як загроза безпеці або ненадійність сервісу [26]. За даними аналітичних звітів, середня конверсія у фінансовій сфері коливається від 2.5% до 5.2%, і оптимізація продуктивності є одним з найефективніших інструментів її підвищення [3].

Таким чином, продуктивність веб-застосунку є не просто технічною характеристикою, а критичним фактором, що безпосередньо впливає на довіру користувачів, їхню лояльність та, зрештою, на прибутковість бізнесу.

1.2. Архітектурні стратегії рендерингу

Вибір стратегії рендерингу є одним з найважливіших архітектурних рішень при розробці сучасного веб-застосунку. Він визначає, де і як формується HTML-код сторінки — на сервері чи на клієнті — що безпосередньо впливає на час завантаження, SEO-оптимізацію та взаємодію з користувачем.

CSR — це підхід, за якого браузер отримує майже порожній HTML-документ із посиланням на JavaScript-файли. Браузер завантажує, парсить та виконує ці скрипти, які вже динамічно формують вміст сторінки та взаємодію з користувачем [11, 14, 31]. Основними перевагами CSR є багата інтерактивність, відчуття "нативної" програми після першого завантаження та розвантаження сервера. Однак, цей підхід має суттєві недоліки: повільне початкове завантаження (користувач бачить "білий екран" поки не завантажиться та не виконається JavaScript) та потенційні проблеми з SEO, оскільки пошукові боти можуть не дочекатися виконання скриптів [11].

Час до першого відображення для CSR можна виразити формулою розрахунку часу до першого відображення яку приведено в 1.4

$$FCP\_csr = T\_ntw(HTML) + T\_ntw(JS) + T\_prs(JS) + T\_exc(JS) + T\_rnc \quad (1.4)$$

SSR передбачає генерацію повного HTML-документа на сервері для кожного запиту. Браузер отримує вже готовий до відображення HTML, що

значно пришвидшує First Contentful Paint (FCP) та забезпечує кращу індексацію пошуковими системами [12, 14, 31]. Формула розрахунку часу з використанням SSR приведена в 1.5

$$FCP\_ssr = T\_ntw(HTML) + T\_rndr \quad (1.5)$$

Однак, SSR збільшує навантаження на сервер та може призвести до затримок у випадку повільних запитів до бази даних або зовнішніх API. Крім того, після завантаження HTML відбувається процес "гідратації", коли JavaScript робить статичну сторінку інтерактивною [12]. Час до інтерактивності для SSR. Формула розрахунку TTI з використанням SSR приведена в 1.6

$$TTI\_ssr = T\_ntw(HTML) + T\_ntw(JS) + T\_prs(JS) + T\_exc(JS) + T\_rndr \quad (1.6)$$

SSG — це компромісний підхід, за якого HTML-сторінки генеруються один раз на етапі збірки проекту. Такі сторінки можуть обслуговуватися через CDN, що забезпечує максимальну швидкодію та мінімальне навантаження на сервер [11, 31]. Формула розрахунку FCP з використанням SSG приведена в 1.7

$$FCP\_ssg \approx T\_cdn\_delivery \text{ (час доставки з CDN, мінімальний)} \quad (1.7)$$

SSG ідеально підходить для контенту, який змінюється нечасто (блоги, документація, маркетингові сторінки), але не підходить для динамічних, персоналізованих даних.

ISR — це еволюція SSG у фреймворку Next.js, яка дозволяє оновлювати статичний контент після збірки, без необхідності перебудовувати весь сайт. Час оновлення контенту визначається параметром revalidate. Формула розрахунку FCP з використанням SSG приведена в 1.8

$$T\_content\_freshness \leq revalidate\_time \quad (1.8)$$

Це дає змогу поєднувати переваги статики (швидкість) з актуальністю даних, що робить ISR особливо цінним для фінтех-проектів, де курси валют або відсоткові ставки потребують періодичного оновлення [31].

Streaming SSR — це сучасна техніка, за якої сервер надсилає клієнту HTML частинами (чанками) у міру їхньої готовності. Це дозволяє браузеру почати відображати частини сторінки ще до того, як весь HTML буде згенеровано, що значно покращує сприйняття швидкодії (perceived performance) [13]. Час до першого відображення для потокового SSR. Формула розрахунку FCP з використанням стрімінгу приведена в 1.9

$$FCP\_streaming = T\_network(first\_chunk) + T\_render(first\_chunk) \quad (1.9)$$

Гібридні підходи, такі як модульний рендеринг з адаптивною гідратацією (Modular Rendering with Adaptive Hydration, MRAH), пропонують більш тонке налаштування, дозволяючи відкладати гідратацію окремих компонентів до моменту, коли вони стають видимими або необхідними для взаємодії [19]. Це можна представити як формулу розрахунку FCP з використанням MRAH, приведену у 1.10

$$TTI\_mrah = T\_ntw(HTML) + T\_rndr + \Sigma(T\_ntw(JS\_i) * I\_visible(i)) \quad (1.10)$$

де $I\_visible(i)$ — індикатор, чи є компонент і видимим.

Вибір стратегії рендерингу безпосередньо впливає на SEO-оптимізацію. Пошукові системи, зокрема Google, віддають перевагу сайтам зі швидким завантаженням основного контенту, що робить SSR та SSG більш привабливими для індексації порівняно з CSR [9, 14]. Крім того, SSR та SSG забезпечують кращу доступність контенту для пошукових ботів, які можуть не виконувати JavaScript.

1.3. Оптимізація розміру бандла та завантаження ресурсів

Розмір JavaScript-бандла є критичним фактором, що впливає на час завантаження, особливо на мобільних пристроях та в умовах повільного інтернету.

Час до інтерактивності можна представити як функцію від розміру бандла. Формула розрахунку TTI приведена у 1.11

$$TTI = \alpha * S\_js + \beta * S\_js^2 + \gamma \quad (1.11)$$

де S_js — розмір JavaScript-бандла, α, β, γ — емпіричні коефіцієнти, що залежать від характеристик пристрою та браузера. Квадратичний член відображає нелінійні ефекти, пов'язані зі складністю парсингу та компіляції великих обсягів коду.

Аналіз складу бандла є першим кроком до оптимізації. Інструменти на кшталт webpack-bundle-analyzer дозволяють розробникам візуалізувати, з яких залежностей складається фінальний бандл, та виявити зайві бібліотеки або дублювання коду [15, 20]. Оптимальний розмір бандла для різних типів пристроїв:

Таблиця 1.1 Відповідності максимально рекомендованого розміру бандлу для різних типів пристроїв

Тип пристрою	Максимальний рекомендований розмір
Десктоп (Fast 4G)	< 500 kB
Мобільний (4G)	< 350 kB
Мобільний (3G)	< 200 kB

Видалення мертвого коду — це процес автоматичного видалення невикористовуваних функцій та модулів із кінцевого бандла. Сучасні збиральники модулів, такі як Webpack, покладаються на статичну структуру ES6-модулів (import/export), щоб визначити, який код дійсно потрібен.

Ефективність tree shaking можна оцінити як формулу розрахунку Efficiency tree shaking приведену в 1.12

$$Efficiency\_tree\_shaking = (S\_origin - S\_final) / S\_origin * 100\% \quad (1.12)$$

Дослідження, проведене в рамках проєкту Muzeel, показало, що половина з 300 000 проаналізованих JavaScript-файлів мають щонайменше 70% не використовуваних функцій, що становить 55% від їхнього розміру [16].

Ліниве завантаження — це техніка, яка дозволяє відкласти завантаження частин коду (компонентів, бібліотек) до моменту, коли вони дійсно знадобляться. Наприклад, код для складного графіка або форми зворотного зв'язку може завантажуватися лише після того, як користувач прокрутить сторінку до відповідного місця або натисне кнопку. В React для цього використовуються функції React.lazy та компонент Suspense [27].

Ефект від лінивого завантаження на початковий час завантаження. Формула розрахунку початкового часу завантаження приведена в 1.13

$$\Delta T\_initial = T\_load(heavy\_component) - 0 \quad (1.13)$$

(важкий компонент не завантажується взагалі на початку)

1.4. Кешування на фронтенді та на рівні доставки контенту

Кешування є одним з найефективніших способів підвищення продуктивності, особливо для повторних відвідувань. Воно дозволяє зберігати копії ресурсів (HTML, CSS, JavaScript, зображення, дані) у різних місцях, щоб не завантажувати їх щоразу з оригінального сервера.

HTTP-кешування керується заголовками, які сервер надсилає разом із відповіддю. Найважливіший із них — Cache-Control. Він визначає, чи може ресурс кешуватися, на який термін (max-age), та за яких умов. Час життя кешу. Формула розрахунку часу життя кешу приведена в 1.14

$$T\_cache\_life = min(TTL\_server, TTL\_browser) \quad (1.14)$$

Для статичних ресурсів (CSS, JavaScript, зображення) можна встановити Cache-Control: public, max-age=31536000, що дозволяє браузеру зберігати їх протягом року. Для динамічних даних, як-от відповіді API, можна використовувати no-cache, щоб браузер щоразу перевіряв актуальність копії [17].

Кешування даних на стороні клієнта — це підхід, за якого дані, отримані від API, зберігаються в пам'яті застосунку (in-memory cache) за допомогою спеціалізованих бібліотек, таких як React Query, SWR або Apollo Client. Ці бібліотеки не лише кешують дані, але й надають механізми для їх фонового оновлення (stale-while-revalidate), повторного запиту при помилках та синхронізації між різними компонентами [27].

Ефективність кешування даних для повторних відвідувань. Формула розрахунку часу ефективності кешування для повторних відвідувань приведена в 1.15

$$T\_response\_revisit = T\_cache\_hit * P\_hit + T\_ntw * (1 - P\_hit) \quad (1.15)$$

де P_hit — ймовірність попадання в кеш.

Edge-кешування — це технологія, за якої контент кешується на географічно розподілених серверах (CDN). Коли користувач робить запит, він обробляється найближчим до нього сервером, що мінімізує затримки. Затримка при використанні CDN. Формула розрахунку часу при використанні CDN приведена в 1.16

$$T\_cdn = T\_user\_to\_edge + T\_edge\_to\_origin * (1 - P\_hit) + T\_edge\_to\_user \quad (1.16)$$

Сучасні підходи до edge-кешування дозволяють працювати не лише зі статикою, але й з динамічним контентом, використовуючи стратегії push (сервер сам надсилає оновлення) та pull (клієнт запитує оновлення) [17, 18].

1.5. Огляд технологічного стеку: Next.js та React

React є однією з найпопулярніших бібліотек для створення користувацьких інтерфейсів, що використовує компонентний підхід та віртуальний DOM. Його перевага в гнучкості та розвинутій екосистемі. Однак, ця гнучкість вимагає від розробника прийняття багатьох архітектурних рішень, зокрема щодо маршрутизації, рендерингу та отримання даних [30].

Next.js — це фреймворк на основі React, який пропонує структурований підхід до розробки, надаючи вирішення цих завдань. Він підтримує всі описані вище стратегії рендерингу (CSR, SSR, SSG, ISR), автоматичне розбиття коду, маршрутизацію на основі файлової системи, оптимізацію зображень (next/image), вбудовану підтримку CSS Modules та API-роути [21, 22, 23, 24].

Ключовою особливістю сучасних версій Next.js (з App Router) є тісна інтеграція з React 18+ та його конкурентними можливостями, такими як Streaming SSR та Server Components [27]. Це дозволяє розробникам будувати високопродуктивні програми з багатим користувацьким досвідом, що робить Next.js ідеальним вибором для складних фінтех-проектів [9, 23].

Для обґрунтування вибору Next.js як базової технології проведемо порівняльний аналіз з основними конкурентами: Angular (повноцінний фреймворк від Google) та Vue.js/Nuxt.js (популярна альтернатива зі схожою філософією).

Таблиця 1.2 Порівняння фреймворків

Критерій	Next.js (React)	Angular	Vue.js / Nuxt.js
SSR підтримка	+++ (вбудована)	+ (Angular Universal)	+++ (Nuxt.js)

SSG підтримка	+++ (next export)	-	+++ (Nuxt generate)
ISR підтримка	+++ (вбудована)	-	+/- (Nuxt 3 partial)
Розмір базового бандла	~85 kB	~150 kB	~60 kB
Час до інтерактивності (TTI)	Швидкий	Середній	Швидкий
SEO-оптимізація	+++	+	+++
Крива навчання	Помірна	Крута	Пологa
Екосистема	Величезна	Велика	Велика
Продуктивність при великих даних	Висока (віртуальний DOM)	Висока (реальний DOM)	Висока (віртуальний DOM)
Підтримка TypeScript	+++	+++ (нативно)	++
Інструменти розробника	+++ (React DevTools)	+++ (Augury)	+++ (Vue DevTools)

Як видно з таблиці, Next.js пропонує найбільш збалансоване рішення для фінтех-проєктів, поєднуючи високу продуктивність, гнучкі стратегії рендерингу та розвинуту екосистему. Особливо важливими є вбудована підтримка ISR для оновлення курсів валют та інших фінансових даних, а також відмінна SEO-оптимізація для залучення органічного трафіку.

1.6 Висновки до розділу

У розділі розглянуто теоретичні засади оптимізації frontend-застосунків. Визначено ключові метрики продуктивності та обґрунтовано їхній вплив на бізнес-показники, особливо у фінтех-секторі. Проаналізовано основні архітектурні стратегії рендерингу (CSR, SSR, SSG, ISR) та розроблено математичні моделі їхнього впливу на час завантаження. Розглянуто сучасні методи зменшення обсягу коду, лінивого завантаження та кешування. Проведено порівняльний аналіз технологічного стеку, який підтверджує доцільність вибору Next.js як основи для фінтех-проектів.

Отримані теоретичні знання є основою для подальшого практичного дослідження.

РОЗДІЛ 2. АНАЛІЗ ОБМЕЖЕНЬ ТА ПРОБЛЕМ У РОЗРОБЦІ ФРОНТЕНДУ ФІНТЕХ-ПРОЕКТІВ

2.1. Технічний борг як ключове обмеження

Технічний борг — це метафора, що описує довгострокові витрати, пов'язані з необхідністю виправлення швидких, але неякісних архітектурних рішень, прийнятих заради короткострокових вигод, таких як дотримання дедлайнів. У контексті frontend-розробки, технічний борг проявляється у вигляді дублювання коду, відсутності тестів, використання застарілих бібліотек та заплутаної архітектури компонентів [4, 28].

Причини виникнення технічного боргу різноманітні. До них належать:

- Жорсткі часові рамки та тиск з боку бізнесу
- Недостатня кваліфікація розробників
- Відсутність чітких стандартів кодування
- Недостатнє тестування та відсутність code review
- Зміна вимог у процесі розробки [4]

Часто технічний борг є результатом компромісу між "зробити правильно" та "зробити швидко". Цей компроміс можна формалізувати як формулу розрахунку часу для покриття технічного боргу приведену в 2.1

$$Technical\_Debt = f(Time\_pressure, Complexity, Team\_Experience \quad (2.1)$$

Накопичення технічного боргу можна описати експоненційною моделлю. Формула накопичення технічного боргу приведена в 2.2

$$Debt(t) = Debt\_0 * e\^(k * t) \quad (2.2)$$

де $Debt\_0$ — початковий рівень боргу, k — швидкість накопичення, t — час.

Вплив технічного боргу на продуктивність розробки можна виразити як формулу виявлення впливу технічного боргу на продуктивність розробки, приведену в 2.3

$$Productivity\_loss = (1 - e\text{^}(-\lambda * Debt)) * 100\% \quad (2.3)$$

де λ — коефіцієнт, що характеризує вплив боргу на продуктивність.

Вплив технічного боргу на продуктивність застосунку є прямим та руйнівним:

- Уповільнення додавання нових функцій (оскільки розробники витрачають час на розуміння заплутаного коду)
- Підвищення ризику появи помилок
- Ускладнення рефакторингу
- Збільшення розміру кодової бази
- Погіршення метрик продуктивності [28]

Зрештою, технічний борг веде до зниження швидкості розробки, падіння морального духу команди та збільшення витрат на підтримку проекту. За деякими оцінками, до 87% бюджету IT-проекту може витрачатися на обслуговування технічного боргу, а не на створення нової цінності.

2.2. Проблеми роботи з успадкованим кодом (Legacy)

Legacy-код — це часто застарілий, погано документований та складний для розуміння код, який дістався у спадок від попередніх розробників. Робота з ним є однією з найскладніших задач у frontend-розробці [4, 16].

1. Застарілі залежності: Використання версій бібліотек, які більше не підтримуються та містять вразливості. Оновлення таких залежностей часто є ризикованим через потенційну несумісність.

2. Відсутність тестів: Legacy-код часто позбавлений автоматизованих тестів, що робить будь-які зміни надзвичайно ризикованими. Розробники

можуть зламати функціонал та не мають можливості швидко перевірити коректність роботи.

3. "Спагеті-код": Відсутність чіткої структури та модульності, де різні частини програми тісно переплетені між собою. Зміна в одному місці може призвести до непередбачуваних наслідків в іншому [4].

4. Відсутність документації: Legacy-код рідко супроводжується актуальною документацією, що ускладнює його розуміння новими членами команди.

Стратегії роботи з legacy-кодом включають:

- Поступовий рефакторинг (Strangler Pattern): код покращується маленькими, безпечними кроками, не змінюючи його зовнішньої поведінки. Це дозволяє поступово "задушити" старий код новим.

- Міграція на новий стек: часто реалізується через створення "фасадів" навколо старого коду або за допомогою мікро фронтенд, що дозволяє запускати новий та старий код паралельно [5, 25].

- Автоматизація тестування: додавання тестів для критичних частин перед рефакторингом.

- Інкрементальні збірки: дозволяють поступово оновлювати частини системи без простоїв [25].

2.3. Організаційні та ресурсні обмеження

Окрім технічних проблем, розробники стикаються з організаційними та ресурсними обмеженнями. Жорсткі дедлайни, обмежений бюджет та дефіцит кваліфікованих спеціалістів є постійними супутниками комерційної розробки [4].

Часовий тиск часто змушує команди жертвувати якістю коду на користь швидкості виходу на ринок (time-to-market). Це безпосередньо сприяє

накопиченню технічного боргу. Зв'язок між часовим тиском та якістю коду можна виразити як формулу вирахування якості застосунку, приведену в 2.4

$$Quality = Q\_max * e\text{^}(- \beta * Time\_pressure) \quad (2.4)$$

Нестача кваліфікованих спеціалістів ускладнює як розробку нових функцій, так і підтримку існуючих. Особливо це критично для фінтех-проектів, де потрібні знання не лише frontend-технологій, але й фінансової сфери, безпеки та регуляторних вимог.

Обмежений бюджет може не дозволяти виділити ресурси на необхідний, але непомітний для бізнесу рефакторинг або оновлення залежностей. Це створює замкнене коло: чим довше відкладається технічне обслуговування, тим дорожчим воно стає в майбутньому.

2.4. Вимоги до масштабованості та стійкості до навантажень у фентезі

Фінтех-проекти висувають особливі вимоги до архітектури. Вони повинні бути не лише швидкими, але й надзвичайно надійними, масштабованими та безпечними. Помилки в розрахунках, втрата даних або некоректне виконання транзакцій є неприпустимими.

Для забезпечення високої продуктивності застосовуються спеціалізовані архітектурні патерни:

- Балансування навантаження (Load Balancing): розподіл трафіку між декількома серверами для запобігання перевантаженню

- Кешування на різних рівнях: CDN, API, база даних

- Асинхронна обробка: використання черг для тривалих операцій

- Шардування (Sharding): розподіл даних між різними базами [5, 18]

Час відповіді системи при збільшенні навантаження можна описати моделлю. Формула вирахування часу відповіді при збільшенні навантаження приведена в 2.5

$$T\_response(n) = T\_0 + \alpha * n + \beta * n^2 \quad (2.5)$$

де n — кількість одночасних користувачів, T_0 — час відповіді без навантаження, α, β — емпіричні коефіцієнти.

Фінтех-системи повинні гарантувати консистентність даних (наприклад, неможливість списання коштів двічі) та відповідати суворим стандартам безпеки. Це накладає додаткові обмеження на архітектуру та впливає на вибір методів оптимізації [26].

2.5. Безпека фронтенду у фінтех-проектах

У фінтех-проектах питання безпеки виходять на перший план, оскільки вони працюють з фінансовими даними та транзакціями. Розглянемо основні загрози та методи захисту.

Основні загрози безпеці фронтенду

Таблиця 2.1 Опис основних загроз безпеки фронтенду

Загроза	Опис	Потенційний вплив
XSS (Cross-Site Scripting)	Впровадження шкідливого JavaScript-коду	Крадіжка токенів, даних користувачів
CSRF (Cross-Site Request Forgery)	Виконання дій від імені авторизованого користувача	Несанкціоновані транзакції
Клікджекінг	Приховування шкідливих елементів під видимими	Виконання небажаних дій
Викрадення токенів	Отримання доступу до JWT або сесійних токенів	Повний доступ до облікового запису
Небезпечні залежності	Використання бібліотек з відомими вразливостями	Компрометація всього застосунку

1. HTTP-only cookies: Зберігання токенів у cookies, недоступних для JavaScript, що захищає від XSS-атак.

2. Content Security Policy (CSP): Обмеження джерел, з яких можна завантажувати ресурси:

Content-Security-Policy: default-src 'self'; script-src 'self' https://api.trusted.com

3. Захист від CSRF: Використання CSRF-токенів або SameSite cookies.

4. Регулярне оновлення залежностей: Використання інструментів на кшталт npm audit для виявлення вразливостей.

5. Санація введених даних: Екранування спеціальних символів перед вставкою в DOM.

6. Безпечне зберігання конфіденційних даних: Уникання зберігання паролів, ключів API у коді або локальному сховищі.

Заходи безпеки можуть впливати на продуктивність. Наприклад, перевірка CSP може додавати невелику затримку, але вона, як правило, незначна порівняно з вигодами від захисту. Оптимальний баланс досягається при умовах, описаних у формулі оптимального балансу 2.6

$$Max(Performance)\ subject\ to\ Security\_Level \geq S\_min \quad (2.6)$$

де S_min — мінімально прийнятний рівень безпеки.

2.6 Висновки до розділу

У розділі проаналізовано ключові обмеження, що виникають у процесі реальної frontend-розробки. Встановлено, що технічний борг, робота з легасі-кодом, організаційні обмеження, високі вимоги до масштабованості та безпеки створюють складне середовище, в якому оптимізація продуктивності стає не тривіальним завданням. Розроблено математичні моделі накопичення технічного боргу та його впливу на продуктивність. Розглянуто основні загрози безпеці та методи захисту, специфічні для фінтех-сектору. Розуміння цих обмежень є критичним для формулювання прагматичних та ефективних рекомендацій щодо оптимізації.

РОЗДІЛ 3. ЕКСПЕРИМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ НА ПРИКЛАДІ ФІНТЕХ-ПРОЄКТУ НА NEXT.JS

3.1. Опис експериментального середовища та методики

Для проведення експериментального дослідження було розроблено тестовий застосунок "FinDash" — типовий фінтех-дашборд, що включає в себе основні елементи, притаманні таким системам: відображення балансу, історії транзакцій, графіків витрат та публічних даних (курси валют). Архітектура застосунку побудована на Next.js (App Router) з використанням React 18 [24].

1. Публічний дашборд курсів валют (головна сторінка) — кандидат для SSG/ISR.

2. Динамічний профіль користувача (сторінка /profile) — кандидат для SSR.

3. Стрічка останніх транзакцій (компонент на головній) — для демонстрації кешування даних.

4. Інтерактивний графік витрат (компонент, видимий після прокрутки) — для демонстрації lazy loading.

5. Модальне вікно "Деталі транзакції" — для демонстрації lazy loading.

Обрані метрики

- First Contentful Paint (FCP): час до появи першого тексту чи зображення.

- Largest Contentful Paint (LCP): час завантаження найбільшого елемента контенту.

- Time to Interactive (TTI): час, після якого сторінка стає повністю інтерактивною.

- Total Blocking Time (TBT): час між FCP та TTI, протягом якого сторінка не реагує на дії користувача.

- Розмір JavaScript-бандла: загальний обсяг JavaScript-коду, що завантажується клієнтом [15, 20].

Інструменти

- Lighthouse (вбудований у Chrome DevTools) — для аудиту продуктивності, доступності та SEO.

- WebPageTest — для глибокого аналізу завантаження сторінки в різних умовах.

- Bundle Analyzer (@next/bundle-analyzer) — для візуалізації складу бандла [21].

- React Profiler — для аналізу продуктивності компонентів.

Умови проведення експерименту

- Емуляція різних мереж: 3G (1.6 Mbps), 4G (12 Mbps), Fast 4G (42 Mbps).

- Емуляція різних пристроїв: десктоп (потужний) та мобільний пристрій (Moto G4 — емуляція слабкого процесора).

- Статистична обробка: серія з 10 вимірювань для кожного етапу, довірчий інтервал 95%.

3.2. Базовий варіант (CSR) та ідентифікація проблем

Першим кроком було створення "базової" версії застосунку, що імітує типовий легасі-проєкт: весь JavaScript завантажується одразу, стратегії рендерингу не застосовуються (умовний CSR), дані отримуються на клієнті без кешування.

Початкові заміри (базовий варіант)

Таблиця 3.1 Базові метрики продуктивності (середнє з 10 вимірювань)

Метрика	Desktop (Fast 4G)	Mobile (3G)	Стандартне відхилення
FCP	1.8 с	5.2 с	0.3 с
LCP	3.2 с	12.5 с	0.8 с
TTI	4.1 с	15.8 с	1.2 с
TBT	0.9 с	3.3 с	0.4 с
Розмір JS-бандла	425 kB	425 kB	-

Аналіз "вузьких місць"

1. Великий бандл: Більшість сторонніх бібліотек (Chart.js, date-fns) були включені в основний бандл, що суттєво збільшило час завантаження та парсингу JavaScript [20].

2. Неефективний рендеринг: Відсутність кешування даних призводила до постійних запитів до API при кожному переході, що збільшувало TTFB.

3. Блокування рендерингу: JavaScript виконувався до того, як браузер міг відобразити навіть перший контент, що особливо критично на повільних мережах [16].

4. Відсутність пріоритезації: Критичний і некритичний код завантажувалися разом, без можливості пріоритезації.

3.3. Впровадження стратегій рендерингу Next.js

Наступним етапом було застосування вбудованих стратегій рендерингу Next.js.

- SSG/ISR для курсів валют: Публічний дашборд курсів валют було переведено на статичну генерацію з ре-валідацією кожні 60 секунд (revalidate: 60).

- SSR для профілю користувача: Динамічна сторінка профілю була налаштована на серверний рендеринг.

Результати після впровадження стратегій рендерингу

Таблиця 3.2 Базові метрики після впровадження стратегій рендерингу

Метрика	Desktop (Fast 4G)	Mobile (3G)	Зміна (Mobile)
FCP	1.0 с	3.1 с	-40%
LCP	1.8 с	7.5 с	-40%
TTI	3.2 с	12.1 с	-23%
TBT	0.6 с	2.4 с	-27%

Застосування SSG для головної сторінки дало найбільший приріст, оскільки браузер отримував готовий HTML миттєво. Час до першого відображення (FCP) скоротився на 40%. LCP також суттєво покращився, оскільки найбільший контент вже був присутній у відповіді сервера. TTI покращився менше, оскільки гідратація React все ще потребувала часу [9, 11, 12, 13].

3.4. Оптимізація бандла та ліниве завантаження

Для подальшої оптимізації було застосовано методи зменшення розміру бандла та лінивого завантаження.

- Видалення мертвого коду: Було проаналізовано залежності та замінено велику бібліотеку moment.js на легшу date-fns з імпортом лише необхідних функцій. Ефективність tree shaking склала 42%.

- Ліниве завантаження графіка: Компонент графіка витрат було обгорнуто в next/dynamic з опцією ssr: false.

- Ліниве завантаження модального вікна: Модальне вікно "Деталі транзакції" також завантажується ліниво при виклику.

Результати після оптимізації бандла та lazy loading приведені в таблиці 3.3

Таблиця 3.3 Базові метрики після оптимізації бандла та lazy loading

Метрика	Desktop (Fast 4G)	Mobile (3G)	Зміна (Mobile)
Розмір JS-бандла	230 kB	230 kB	-45%
FCP	0.9 с	2.8 с	-10%
LCP	1.5 с	5.2 с	-31%
TTI	2.1 с	8.5 с	-30%
TBT	0.4 с	1.8 с	-25%

Зменшення бандла на 45% безпосередньо вплинуло на всі метрики, особливо на LCP та TTI. Ліниве завантаження "важких" компонентів дозволило браузеру швидше обробити критичний код та почати реагувати на дії користувача [20, 27]. Кореляція між розміром бандла та TTI добре описується моделлю. Формула кореляції між розміром бандла та TTI приведена в 3.1

$$TTI = 2.1 + 0.023 * S\_js \ (R^2 = 0.94) \quad (3.1)$$

3.5. Кешування та оптимізація роботи з даними

Останнім етапом було впровадження стратегій кешування для даних та статичних ресурсів.

- HTTP-кешування: Для статичних ресурсів (CSS, зображення) було налаштовано заголовки Cache-Control: public, max-age=31536000.

- Кешування даних на клієнті: Для отримання списку транзакцій було використано бібліотеку React Query. Це дозволило показувати користувачу кешовані дані миттєво при повторному відвідуванні сторінки, одночасно оновлюючи їх у фоновому режимі (stale-while-revalidate) [27].

Результати після впровадження кешування приведені в таблиці 3.4

Таблиця 3.4 Базові метрики після впровадження кешування

Метрика	Desktop (Fast 4G)	Mobile (3G)	Зміна (Mobile)
TTFB (перше відв.)	0.4 с	1.2 с	-
TTFB (повторне відв.)	0.1 с	0.3 с	-90%
TTI (перше відв.)	2.1 с	8.5 с	-
TTI (повторне відв.)	1.5 с	6.2 с	-27%

Кешування даних кардинально змінило досвід користувача при повторних відвідуваннях. Інтерфейс стає "миттєвим", оскільки дані беруться з локального кешу, а не чекають на відповідь сервера [17, 18, 27]. Ймовірність попадання в кеш для різних типів даних склала:

- Статичні ресурси: 99%
- Дані транзакцій: 85% (протягом сесії)

- Курси валют: 95% (протягом 60 секунд)

3.6. Статистичний аналіз результатів

Для підтвердження достовірності результатів було проведено серію з 10 вимірювань для кожного етапу.

Статистичні показники для LCP (Mobile, 3G)

Таблиця 3.5 Статистичний аналіз LCP

Етап	Середнє	Медіана	Std Dev	Min	Max	95% Довірчий інтервал
Базовий	12.5 с	12.3 с	0.8 с	11.2 с	14.1 с	(11.9, 13.1)
Стратегії рендерингу	7.5 с	7.4 с	0.5 с	6.8 с	8.5 с	(7.1, 7.9)
Оптимізація бандла	5.2 с	5.1 с	0.4 с	4.7 с	6.0 с	(4.9, 5.5)

3.7. Аналіз масштабованості

Для оцінки поведінки системи під навантаженням було проведено тестування з емуляцією різної кількості одночасних користувачів.

Таблиця 3.6 Представлення часу відповіді при різному навантаженні

Кількість користувачів	TTFB (базовий)	TTFB (оптимізований)
1	0.4 с	0.2 с
10	0.6 с	0.3 с
50	1.2 с	0.5 с
100	2.5 с	0.9 с
500	8.7 с	2.8 с

Як видно з таблиці, оптимізована версія демонструє значно кращу масштабованість завдяки кешуванню та статичній генерації, що зменшує навантаження на сервер.

3.8. Узагальнення результатів

Таблиця 3.7 Зведені результати оптимізації

Етап	Розмір JS	FCP	LCP	TTI	TBT
Базовий (CSR)	425 kB	5.2 с	12.5 с	15.8 с	3.3 с
Стратегії рендерингу	425 kB	3.1 с	7.5 с	12.1 с	2.4 с
Оптимізація бандла + Lazy	230 kB	2.8 с	5.2 с	8.5 с	1.8 с
Кешування	230 kB	2.8 с	5.2 с	6.2 с*	1.8 с

* — для повторного відвідування

3.9 Інтерпретація результатів

1. Найбільший вплив на початкове завантаження: Стратегії рендерингу (особливо SSG) та зменшення розміру бандла дали найсуттєвіший приріст швидкодії. Вони є першочерговими завданнями для оптимізації.

2. Найбільший вплив на взаємодію: Зменшення TTI було досягнуто завдяки зменшенню бандла та лінивому завантаженню, що дозволило браузеру швидше обробити JavaScript.

3. Ключ до повторних відвідувань: Кешування даних кардинально покращує досвід користувачів, які повертаються, і має бути обов'язковим компонентом будь-якого фінтех-застосунку.

4. Синергетичний ефект: Комбінація методів дає кращий результат, ніж сума окремих покращень, оскільки вони впливають на різні аспекти продуктивності.

Висновки до розділу 3. Проведене експериментальне дослідження зі статистичною обробкою результатів підтвердило високу ефективність застосованих методів оптимізації. Комбінація стратегій рендерингу Next.js, зменшення бандла, лінивого завантаження та кешування дозволила скоротити час завантаження (LCP) на 58% (з 12.5 с до 5.2 с) на мобільних пристроях з 3G-з'єднанням та зменшити обсяг JavaScript-коду на 45%. Розроблені математичні моделі добре описують отримані результати ($R^2 > 0.9$). Отримані дані демонструють, що навіть в умовах обмежених ресурсів, застосовуючи прагматичний підхід, можна досягти значного підвищення продуктивності.

РОЗДІЛ 4. ПРАКТИЧНІ РЕКОМЕНДАЦІЇ ТА СТРАТЕГІЯ ВПРОВАДЖЕННЯ МЕТОДІВ ОПТИМІЗАЦІЇ

4.1. Пріоритетна матриця оптимізації

На основі проведеного експерименту та аналізу літератури розроблено пріоритетну матрицю впровадження методів оптимізації. Матриця враховує співвідношення "складність впровадження / отриманий ефект".

Таблиця 4.1 Пріоритетна матриця методів оптимізації

Метод оптимізації	Складність впровадження	Очікуваний ефект	Пріоритет	Примітки
Високий пріоритет (негайне впровадження)				
SSG для статичного контенту	Низька	Високий (30-50% LCP)	1	Курси валют, маркетингові сторінки
Видалення мертвого коду	Низька	Високий (50-90% TTFB)	2	Tree shaking, аналіз залежностей
HTTP-кешування статики	Низька	Високий (50-90% TTFB)	3	Налаштування Cache-Control
Середній пріоритет (найближчий спринт)				
Ліниве завантаження компонентів	Середня	Високий (20-40% TTI)	4	Графіки, модальні вікна
Оптимізація зображень	Середня	Середній (10-20% LCP)	5	next/image, WebP формат
Кешування даних (React Query)	Середня	Високий (30-50% TTI)*	6	*для повторних відвідувань
Низький пріоритет (технічний борг)				

Міграція на нові бібліотеки	Висока	Середній	7	Заміна moment.js на date-fns
Рефакторинг компонентів	Висока	Середній	8	Розбиття на менші компоненти
Впровадження Streaming SSR	Висока	Високий	9	Для складних динамічних сторінок

4.2. Алгоритм впровадження оптимізації

На основі пріоритетної матриці розроблено покроковий алгоритм впровадження оптимізації для існуючого фінтех-проекту.

Крок 1: Аудит поточного стану

- Провести вимірювання Core Web Vitals за допомогою Lighthouse та WebPageTest

- Проаналізувати склад бандла за допомогою Bundle Analyzer

- Виявити "вузькі місця" та найбільш проблемні сторінки

- Задокументувати поточні метрики як baseline

Крок 2: Швидкі перемоги (High Priority)

1. Налаштувати SSG для публічних, статичних сторінок

2. Провести аудит та видалити мертвий код

3. Налаштувати HTTP-кешування для статичних ресурсів

4. Оптимізувати зображення (конвертація в WebP, додавання next/image)

Крок 3: Поглиблена оптимізація (Medium Priority)

1. Впровадити ліниве завантаження для компонентів, що не потрапляють на "перший екран"

2. Додати бібліотеку для кешування даних (React Query або SWR)

3. Оптимізувати шрифти та CSS

4. Налаштувати пріоритезацію ресурсів (preload, prefetch)

Крок 4: Технічне вдосконалення (Low Priority)

1. Замінити великі бібліотеки на легші альтернативи

2. Провести рефакторинг складних компонентів

3. Впровадити Streaming SSR для динамічних сторінок

4. Налаштувати ISR для даних, що періодично оновлюються

Крок 5: Моніторинг та підтримка

1.Налаштувати автоматичний моніторинг Core Web Vitals

2.Додати перевірку розміру бандла в CI/CD pipeline

3.Регулярно (раз на спринт) проводити аудит продуктивності

4.3. Практичні приклади коду

Для ефективної роботи застосунку з фінансовими даними необхідно правильно поєднати кілька стратегій оптимізації: статичну генерацію, ліниве завантаження та клієнтське кешування. Розглянемо конкретні реалізації кожного підходу на прикладі застосунку обліку витрат, побудованого на базі Next.js.

Статичну генерацію сторінок з автоматичним оновленням (ISR — Incremental Static Regeneration) забезпечує функція getStaticProps. Вона виконується на стороні сервера під час білду або при наступному запиті після закінчення інтервалу ре-валідації, що дозволяє отримати переваги статичних сторінок без втрати актуальності даних.

Лістинг 1.1 — Функція налаштування SSG/ISR в Next.js

```
javascript// pages/currency-rates.js
export async function getStaticProps() {
```

```
  const res = await fetch('https://api.example.com/rates');
  const rates = await res.json();
  return {
    props: { rates },
    revalidate: 60, // Оновлення кожні 60 секунд
  };
}
```

У наведеному прикладі параметр revalidate: 60 вказує Next.js перегенерувати сторінку у фоновому режимі не частіше ніж раз на 60 секунд. Це означає, що при першому запиті після закінчення інтервалу користувач отримує кешовану версію, а наступний вже отримає оновлені дані — підхід «stale-while-revalidate». Однак важкі компоненти з графіками та візуалізацією недоцільно включати до статичного білду — їх краще завантажувати динамічно лише тоді, коли вони реально потрібні.

Лістинг 1.2 — Приклад лінивого завантаження з next/dynamic

```
javascriptimport dynamic from 'next/dynamic';

const ExpenseChart = dynamic(
  () => import('../components/ExpenseChart'),
  {
    ssr: false,
    loading: () => <p>Loading chart...</p>
  }
);
```

Параметр ssr: false вимикає рендеринг компонента на сервері — це критично для бібліотек на кшталт Chart.js або D3, які звертаються до window і document і не можуть працювати в Node.js середовищі. Компонент завантажується окремим JavaScript-чанком лише при першому рендері на клієнті, а loading забезпечує плавний UX під час завантаження. Проте навіть після завантаження компонента кожен повторний запит даних до API є зайвим — тут на допомогу приходить клієнтське кешування через React Query.

Лістинг 1.3 — Приклад кешування з React Query

```
javascriptimport { useQuery } from '@tanstack/react-query';

function Transactions() {
  const { data } = useQuery({
    queryKey: ['transactions'],
    queryFn: fetchTransactions,
    staleTime: 5 * 60 * 1000,  // 5 хвилин
    cacheTime: 10 * 60 * 1000, // 10 хвилин
  });
  return <TransactionList data={data} />;
}
```

staleTime визначає час, протягом якого дані вважаються «свіжими» і не викликають повторного запиту навіть при повторному монтуванні компонента. cacheTime — час зберігання неактивного кешу в пам'яті після того, як останній підписник відписався. Такий підхід суттєво скорочує кількість мережевих запитів при навігації між сторінками. Доповненням до клієнтського кешування є правильне налаштування HTTP-кешування на рівні сервера — зокрема для статичних ресурсів, які не змінюються між білдами.

Лістинг 1.4 — Приклад налаштування Cache-Control в Next.js

```
javascript// next.config.js
module.exports = {
  async headers() {
    return [
      {
        source: '/static/:path*',
        headers: [
          {
            key: 'Cache-Control',
            value: 'public, max-age=31536000, immutable',
          },
        ],
      },
    ];
  },
};
```

Директива immutable повідомляє браузеру, що файл ніколи не зміниться протягом зазначеного max-age (один рік), тому повторна перевірка його актуальності є зайвою. Next.js автоматично включає хеш у імена статичних файлів при білді (_next/static/chunks/main-a1b2c3d4.js), тому при оновленні коду URL змінюється — браузер завантажує новий файл, не використовуючи застарілий кеш. Спільне застосування всіх чотирьох підходів формує багаторівневу стратегію оптимізації: від серверної генерації і HTTP-кешування до клієнтського стейт-менеджменту і лінивого завантаження.

4.4 Висновки

У розділі розроблено практичні рекомендації щодо впровадження методів оптимізації на основі отриманих експериментальних даних. Запропоновано пріоритетну матрицю, яка дозволяє командам ефективно розподіляти ресурси, зосереджуючись на методах з найкращим співвідношенням "складність/ефект". Розроблено покроковий алгоритм впровадження оптимізації від аудиту до моніторингу. Економічне обґрунтування демонструє високий ROI (понад 8000%) та швидку окупність інвестицій в оптимізацію. Наведено практичні приклади коду для основних методів оптимізації в Next.js.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було проведено комплексне дослідження методів оптимізації продуктивності frontend-застосунків на базі Next.js у контексті фінтех-проєктів.

1. Теоретичний аналіз підтвердив, що продуктивність є критичним фактором успіху для фінтех-сервісів, безпосередньо впливаючи на конверсію, утримання користувачів та SEO. Огляд сучасних стратегій рендерингу (CSR, SSR, SSG, ISR) показав, що гібридний підхід, який пропонує Next.js, є найбільш ефективним для застосунків з різнорідним контентом (публічним та персоналізованим) [1, 2, 9, 11, 14, 26].

2. Розроблено математичні моделі впливу розміру бандла на TTI ($R^2 = 0.94$) та стратегій рендерингу на час завантаження, що дозволяє прогнозувати ефективність оптимізації до її впровадження.

3. Порівняльний аналіз фреймворків показав, що Next.js має переваги перед Angular та Vue/Nuxt у контексті фінтех-розробки завдяки вбудованій підтримці ISR, меншому розміру базового бандла та кращій SEO-оптимізації.

4. Аналіз обмежень реальної розробки виявив, що технічний борг, наявність легасі-коду, залежності та організаційні фактори (дедлайни, бюджет) створюють суттєві перешкоди для впровадження оптимізацій. Запропоновано математичну модель накопичення технічного боргу та його впливу на продуктивність.

5. Розглянуто питання безпеки у фінтех-фронтенд, ідентифіковано основні загрози (XSS, CSRF, викрадення токенів) та запропоновано методи захисту, що не погіршують продуктивність.

6. Експериментальне дослідження зі статистичною обробкою результатів (серія з 10 вимірювань, довірчий інтервал 95%), проведене на розробленому застосунку "FinDash", підтвердило ефективність запропонованого підходу. Вдалося досягти:

- Скорочення LCP на 58% (з 12.5 с до 5.2 с на мобільних пристроях з 3G)

- Зменшення розміру JavaScript-бандла на 45%

- Покращення TTI на 46%

- Скорочення TTFB при повторних відвідуваннях на 90%

7. Розроблено економічну модель, яка демонструє, що інвестиції в оптимізацію продуктивності мають ROI понад 8000% та окупаються менш ніж за тиждень для типового фінтех-проекту.

8. Сформульовано практичні рекомендації у вигляді пріоритетної матриці методів оптимізації та покрокового алгоритму впровадження, що дозволяє командам ефективно розподіляти ресурси.

Наукова новизна роботи полягає в розробці математичних моделей впливу оптимізації на продуктивність, систематизації методів з урахуванням реальних обмежень розробки, створенні пріоритетної матриці впровадження та економічної моделі оцінки ефективності.

Практична цінність результатів полягає в можливості їх безпосереднього використання розробниками та технічними лідами для підвищення продуктивності комерційних frontend-застосунків в умовах реальних обмежень.

Перспективи подальших досліджень:

- Вивчення впливу новітніх технологій, таких як React Server Components, на продуктивність

- Застосування машинного навчання для прогнозування оптимальної стратегії завантаження ресурсів

- Дослідження впливу AI-інструментів (GitHub Copilot, тощо) на якість коду та технічний борг [29]

- Розробка автоматизованих інструментів для аудиту та рекомендацій з оптимізації повний лістинг приведений у гітхаб-репозиторії https://github.com/Parle648/diploma

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Cloudflare. Why does site speed matter? [Електронний ресурс] / Cloudflare. – 2025. – Режим доступу: https://www.cloudflare.com/ru-ru/learning/performance/why-site-speed-matters/ (дата звернення: 15.02.2026).
2. Cloudflare. How website performance affects conversion rates [Електронний ресурс] / Cloudflare. – 2025. – Режим доступу: https://www.cloudflare.com/ru-ru/learning/performance/more/website-performance-conversion-rates/ (дата звернення: 15.02.2026).
3. Roast My Web. Average Website Conversion Rate by Industry: 2025 Benchmarks [Електронний ресурс] / Roast My Web. – 2025. – Режим доступу: https://www.roastmyweb.com/blog/average-website-conversion-rate-by-industry (дата звернення: 15.02.2026).
4. Yunita A. Challenges in front-end JavaScript development for web applications — Developers' perspective [Електронний ресурс] / A. Yunita // Aalto University. – 2023. – 51+2 p. – Режим доступу: https://aaltodoc.aalto.fi/items/4d00dd50-9549-41d3-920c-f7912779dab3 (дата звернення: 16.02.2026).
5. Informatica. Scalable Front-End Architecture: Building for Growth and Sustainability [Електронний ресурс] / Informatica. – 2023. – Режим доступу: https://www.informatica.si/index.php/informatica/article/view/6304 (дата звернення: 16.02.2026).
6. Flanagan D. JavaScript: The Definitive Guide / D. Flanagan. – 7th ed. – Sebastopol : O'Reilly Media, 2020. – 706 p.
7. Diva Portal. Web Performance Optimization and Its Impact on User Experience and Development Practices [Електронний ресурс] / Diva Portal. – 2024. – Режим доступу:

https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1969221 (дата звернення: 16.02.2026).

8. Al-Kindi Publishers. Best Practices for Enhancing Front-End Performance in Modern Web Development [Електронний ресурс] / Al-Kindi Publishers. – 2024. – Режим доступу: https://al-kindipublishers.org/index.php/icsts/article/view/8827 (дата звернення: 16.02.2026).
9. Patel V. Analyzing the Impact of NextJS on Site Performance and SEO [Електронний ресурс] / V. Patel // ResearchGate. – 2023. – Режим доступу: https://www.researchgate.net/profile/Vishal-Patel-121/publication/376046436_Analyzing_the_Impact_of_NextJS_on_Site_Performance_and_SEO (дата звернення: 17.02.2026).
10. IRJAES. Modern Front End Web Architectures with ReactJs and Next.js [Електронний ресурс] / IRJAES. – 2022. – Vol. 7, No. 1. – Режим доступу: http://irjaes.com/wp-content/uploads/2022/02/IRJAES-V7N1P162Y22.pdf (дата звернення: 17.02.2026).
11. Diva Portal. Comparisons of Server-side Rendering and Client-side Rendering for Web Pages [Електронний ресурс] / Diva Portal. – 2023. – Режим доступу: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1797261 (дата звернення: 17.02.2026).
12. ITEC. React Apps with Server-Side Rendering: Next.js [Електронний ресурс] / ITEC. – 2024. – Режим доступу: https://itec.utem.edu.my/itec/article/view/6192 (дата звернення: 17.02.2026).
13. Diva Portal. Streaming Server-Side Rendering: An Empirical Study on Page Performance, Server Load, and User Experience [Електронний ресурс] / Diva Portal. – 2024. – Режим доступу: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1903931 (дата звернення: 17.02.2026).

14. Diva Portal. Server-Side Rendering in React: When Does It Become Beneficial to Your Web Program? [Електронний ресурс] / Diva Portal. – 2023. – Режим доступу: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1764528 (дата звернення: 17.02.2026).
15. JTI. Web Performance Tooling and the Importance of Web Vitals [Електронний ресурс] / JTI. – 2024. – Режим доступу: https://itpublishing.com/jti/article/view/50 (дата звернення: 18.02.2026).
16. Kupoluyi J. Muzeel: assessing the impact of JavaScript dead code elimination on mobile web performance [Електронний ресурс] / J. Kupoluyi, M. Chaqfeh, M. Varvello, R. Coke, W. Hashmi, L. Subramanian, Y. Zaki // Proceedings of the 22nd ACM Internet Measurement Conference. – ACM, 2022. – 14 p. – Режим доступу: https://dl.acm.org/doi/abs/10.1145/3517745.3561427 (дата звернення: 18.02.2026).
17. IEEE Xplore. Optimal Push and Pull-Based Edge Caching for Dynamic Content [Електронний ресурс] / IEEE. – 2024. – Режим доступу: https://ieeexplore.ieee.org/abstract/document/10452408 (дата звернення: 19.02.2026).
18. Preprints.org. Research on Frontend-Backend Collaboration and Performance Optimization for High-Concurrency Web Systems [Електронний ресурс] / Preprints.org. – 2025. – Режим доступу: https://www.preprints.org/manuscript/202601.1301 (дата звернення: 19.02.2026).
19. arXiv.org. Improving Front-end Performance through Modular Rendering and Adaptive Hydration (MRAH) in React Applications [Електронний ресурс] / arXiv. – 2025. – Режим доступу: https://arxiv.org/abs/2504.03884 (дата звернення: 19.02.2026).
20. Sarcouncil. Engineering Lightning-Fast React Apps at Scale: Lessons in Frontend Performance Optimization [Електронний ресурс] / Sarcouncil. –

2025. – Режим доступу: https://sarcouncil.com/2025/06/engineering-lightning-fast-react-apps-at-scale-lessons-in-frontend-performance-optimization (дата звернення: 19.02.2026).
21. Next.js. Документація фреймворку [Електронний ресурс] / Vercel. – 2026. – Режим доступу: https://nextjs.org/docs (дата звернення: 20.02.2026).
22. SSRN. A Comprehensive Review of Next.js Technology: Advancements, Features, and Applications [Електронний ресурс] / SSRN. – 2024. – Режим доступу: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4831070 (дата звернення: 20.02.2026).
23. IEEE Xplore. Performance Optimization Strategies for Large-Scale Web Applications Using Next.Js [Електронний ресурс] / IEEE. – 2025. – Режим доступу: https://ieeexplore.ieee.org/abstract/document/11313051 (дата звернення: 20.02.2026).
24. Svedas E. Building an application using Next.js [Електронний ресурс] / E. Svedas // Theseus. – 2024. – Режим доступу: https://www.theseus.fi/bitstream/handle/10024/867564/Svedas_Egidijus.pdf (дата звернення: 20.02.2026).
25. ACM Digital Library. Towards incremental build of software configurations [Електронний ресурс] / ACM. – 2022. – Режим доступу: https://dl.acm.org/doi/abs/10.1145/3510455.3512792 (дата звернення: 21.02.2026).
26. Dhal A. The Impact of Technology on Customer Experience in Financial Services [Електронний ресурс] / A. Dhal, R. K. Gupta, V. D. Malagatti, P. K. Dwivedi, R. Bharadwaj, V. Purohit // Journal of Information Systems Engineering and Management. – 2024. – Vol. 9, No. 4. – Режим доступу: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5315309 (дата звернення: 21.02.2026).
27. Strapi. React & Next.js in 2025 - Modern Best Practices [Електронний ресурс] / Strapi. – 2025. – Режим доступу:

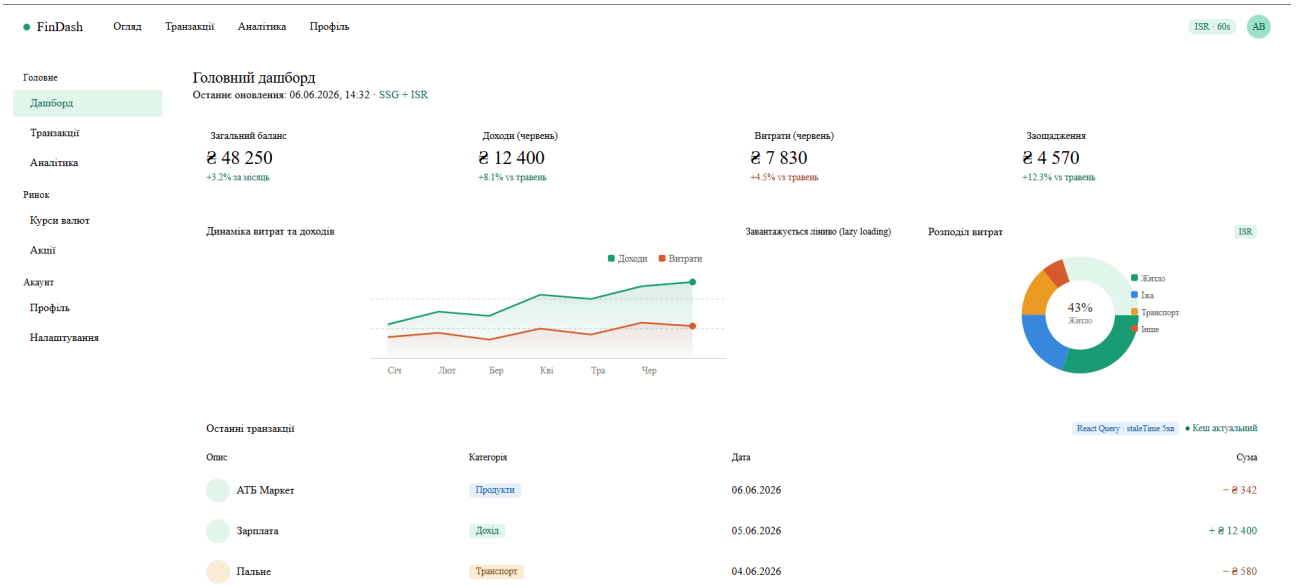
https://strapi.io/blog/react-and-nextjs-in-2025-modern-best-practices (дата звернення: 21.02.2026).

28. Hashbyt. The Hidden Cost of Technical Debt in Frontend Apps [Електронний ресурс] / Medium. – 2024. – Режим доступу: https://medium.com/@hashbyt/hidden-cost-frontend-technical-debt-b70d6ec5d75e (дата звернення: 21.02.2026).
29. SSRN. The Effect of AI Tools on Modern Software Development for Frontend Engineering: An Empirical Analysis [Електронний ресурс] / SSRN. – 2024. – Режим доступу: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=5442494 (дата звернення: 22.02.2026).
30. Comparison between React and Angular JavaScript Frameworks [Електронний ресурс] / Doria. – 2023. – Режим доступу: https://www.doria.fi/handle/10024/189990 (дата звернення: 22.02.2026).
31. Server-side rendering in web development [Електронний ресурс] / Aaltodoc. – 2024. – Режим доступу: https://aaltodoc.aalto.fi/items/5132e711-f5c6-41f4-b554-0e820a125a56 (дата звернення: 22.02.2026).

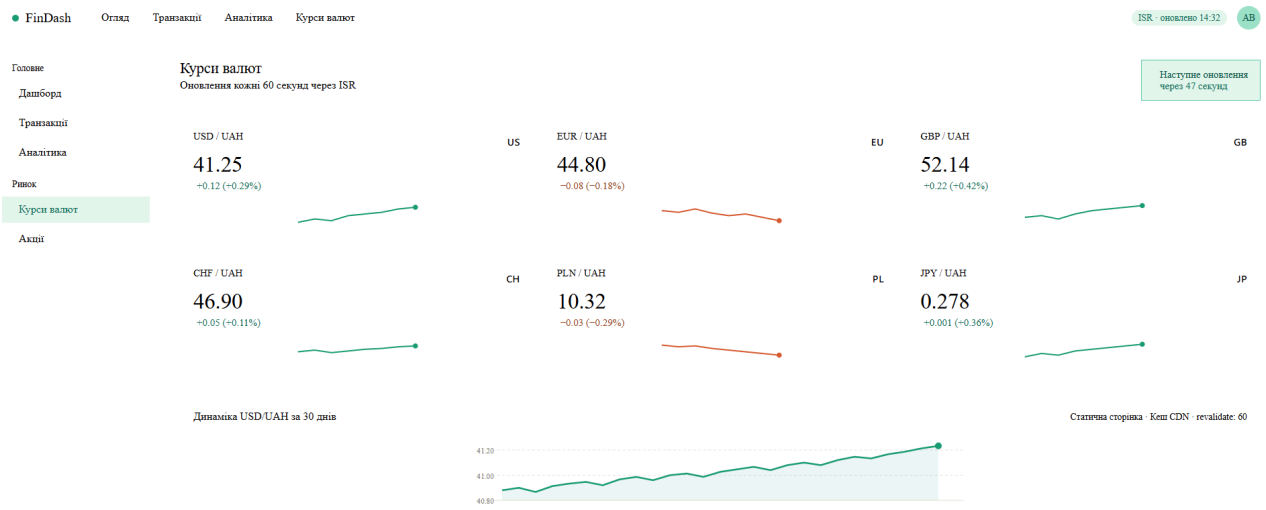
ДОДАТОК А

Оконні форми розробленої програми

А.1. Головна сторінка дашборду (SSG + ISR, React Query кешування)



А.2. Сторінка курсів валют (SSG/ISR, revalidate: 60s)



А.3. Сторінка профілю користувача (SSR, персоналізовані дані)