Міністерство освіти і науки України
Луцький національний технічний університет
Факультет комп'ютерних та інформаційних технологій
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА
ЗА СТУПЕНЕМ ВИЩОЇ ОСВІТИ «МАГІСТР»

РОЗРОБКА ТА ДОСЛІДЖЕННЯ БАГАТОСТОРІНКОВОГО ВЕБ-ДОДАТКА З ВИКОРИСТАННЯМ NEXT.JS

DEVELOPMENT AND RESEARCH OF MULTI-PAGE APPLICATION USING NEXT.JS

спеціальність 122 Комп'ютерні науки

освітня програма «Комп'ютерні науки»

Виконав: здобувач вищої освіти
групи КНм-21
Кушнірчук Роман Сергійович
_____
(підпис)
Керівник: д. пед. н. професор
Тулашвілі Юрій Йосипович
_____
(підпис)

Кваліфікаційну роботу
допущено до захисту
«___» _____ 2025 р.
Гарант освітньої програми:
к.т.н., доцент
Ліщина Валерій Олександрович
_____
(підпис)

Луцьк – 2025 року

ЛУЦЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних та інформаційних технологій*
Кафедра *комп'ютерних наук*
Ступінь вищої освіти *магістр*
Галузь знань: *12 Інформаційні технології*
Спеціальність: *122 Комп'ютерні науки*
Освітня програма: *«Комп'ютерні науки»*

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____Валерій ЛІЩИНА
«14» травня 2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ ЗДОБУВАЧА
ДРУГОГО (МАГІСТЕРСЬКОГО) РІВНЯ ВИЩОЇ ОСВІТИ

Кушнірчук Роман Сергійович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи «Розробка та дослідження багатосторінкового додатка з використанням Next.js»
Керівник д. пед. н. Тулашвілі Юрій Йосипович
затверджені наказом закладу вищої освіти від *«14» травня 2025 р.* №255/01-02
2. Строк подання здобувачем вищої освіти кваліфікаційної роботи *«05» грудня 2025 р.*
3. Вихідні дані роботи: наукова та навчально-методична література технічне завдання на розробку багатосторінкового додатку
4. Зміст розрохунковопояснювальної записки (перелік питань, що потрібно розробити):*Аналіз сучасного стану проблеми, існуючих методів і засобів розробки, аналіз і вибір архітектури для проектування, вибір інструментів і опис вимог до веб-додатку, експериментальне дослідження та розробка, проведення тестування та дослідження ефективності*
5. Перелік графічного матеріалу: *схеми, рисунки, таблиці, лістинг*

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис	
		завдання видав	завдання прийняв
Аналіз проблеми за темою роботи та постановка завдань дослідження	*Тулашвілі Ю. Й.*		
Теоретичне дослідження та практична реалізація предмету дослідження	*Тулашвілі Ю. Й.*		
Експериментальне дослідження результативності предмету дослідження	*Тулашвілі Ю. Й.*		
Показник запозичень тексту	_____%		
Інструментальна перевірка	*Кошелюк В. А.*		
Нормоконтроль	*Сачук В. О.*		
Гарант ОПП	*Ліщина В. О.*		

7. Дата видачі завдання *«14 » травня 2025р.*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи магістра	Строк виконання етапів роботи	Примітка
1	*Провести огляд літературних джерел по темі кваліфікаційної роботи*	*до 8.09.2025 р*	
2	*Провести аналіз загальної проблеми і вибір напрямків дослідження*	*до 22.09.2025 р.*	
3	*Розробити функціональну схему роботи програмного продукту*	*до 06.10.2025р*	
4	*Описати засоби розробки об'єкта проектування*	*до 28.10.2025 р.*	
5	*Практична реалізація об'єкта проектування*	*до 01.11.2025 р.*	
6	*Розробити методику для проведення експерименту*	*до 05.11.2025 р.*	
7	*Провести аналіз результатів експерименту*	*до 28.11.2025 р.*	
8	*Здача чистового варіанту магістерської роботи на кафедру*	*до 03.12.2025 р.*	

Здобувач вищої освіти _____ Кушнірчук РОМАН

Керівник роботи _____ Тулашвілі ЮРІЙ

АНОТАЦІЯ

Кушнірчук Р. С. Розробка та дослідження багатосторінкового додатка з використанням Next.js. Рукопис.

Кваліфікаційна робота магістра ОП «Комп'ютерні науки» спеціальності 122 «Комп'ютерні науки». Луцький національний технічний університет. Луцьк, 2025.

Кваліфікаційна робота магістра складається з вступу, 3 розділів, висновків і пропозицій, списку використаних джерел, додатків (згідно структури кваліфікаційної роботи, затвердженої кафедрою).

У роботі розкрито теоретичні засади побудови веб-додатків, розглянуто етапи еволюції веб-технологій і визначено місце Next.js серед сучасних фреймворків. Проаналізовано архітектуру, принципи роботи SSR/SSG/ISR, механізми роутингу, а також особливості інтеграції з базами даних PostgreSQL і MongoDB через ORM Prisma та бібліотеку Mongoose. Описано створення API Routes, реалізацію системи автентифікації (NextAuth.js), управління станом (Zustand), проведення тестування (Jest, Cypress) та моніторингу (Sentry).

Ключові слова: Next.js, веб-додаток, SSR, SSG, ISR, Prisma, API, TypeScript, React, продуктивність.

ABSTRACT

Roman Kushnirchuk. Development and research of a multi-page application using Next.js. Manuscript.

Master's thesis in Computer Science, specialisation 122 «Computer Science». Lutsk National Technical University. Lutsk, 2025.

The master's thesis consists of an introduction, 3 chapters, conclusions and recommendations, a list of references, and appendices (in accordance with the structure of the thesis approved by the department).

The thesis reveals the theoretical foundations of web application development, considers the stages of web technology evolution, and determines the place of Next.js among modern frameworks. It analyses the architecture, principles of SSR/SSG/ISR, routing mechanisms, and features of integration with PostgreSQL and MongoDB databases via ORM Prisma and the Mongoose library. The creation of API Routes, the implementation of an authentication system (NextAuth.js), state management (Zustand), testing (Jest, Cypress), and monitoring (Sentry) are described.

Keywords: Next.js, web application, SSR, SSG, ISR, Prisma, API, TypeScript, React, performance.

ЗМІСТ

ВСТУП

У сучасних умовах розвитку цифрового суспільства веб-додатки стали невід’ємним елементом функціонування бізнесу, освіти, державного управління та комунікацій. Трансформація способів взаємодії людини з інформаційними системами призвела до того, що якість, продуктивність і зручність інтерфейсів безпосередньо впливають на конкурентоспроможність організацій та ефективність їхньої діяльності. Якщо ще два десятиліття тому веб-сайти виконували здебільшого інформаційну функцію, то сьогодні вони перетворилися на складні інтерактивні системи, здатні забезпечувати широкий спектр сервісів – від електронної комерції та освітніх платформ до корпоративних рішень і систем управління.

Особливе значення в цьому контексті мають багатосторінкові веб-додатки (Multi-Page Applications, MPA), які відзначаються розширеною архітектурою та можливістю підтримувати складні бізнес-процеси. На відміну від односторінкових додатків, що здобули популярність завдяки динамічності та інтерактивності, багатосторінкові рішення забезпечують більшу масштабованість, прозорішу організацію даних і підвищену зручність навігації для користувача. Саме тому інтерес до розробки MPA знову активно зростає, особливо у сфері корпоративних і державних систем, де важливими є структурованість і модульність.

У цьому контексті значну увагу науковців та практиків привертає фреймворк Next.js, що базується на бібліотеці React і поєднує сучасні підходи до створення веб-застосунків. Він дозволяє ефективно реалізовувати як односторінкові, так і багатосторінкові рішення, поєднуючи переваги server-side rendering (SSR), static site generation (SSG) та інкрементальної генерації (ISR). Таке поєднання створює можливості для оптимізації продуктивності, покращення індексації в пошукових системах, а також підвищення зручності користувацького досвіду.

Актуальність теми дослідження також визначається зростаючими вимогами до безпеки та надійності веб-додатків. Оскільки сучасні інформаційні системи обробляють великі обсяги персональних і комерційних даних, питання захисту інформації та стійкості до кібератак стають особливо важливими. Next.js у цьому контексті пропонує низку рішень для інтеграції з протоколами автентифікації, управління доступом і використання сучасних інструментів безпеки.

Не менш важливим чинником є економічна доцільність використання сучасних фреймворків. Розробка на базі Next.js дозволяє зменшити витрати часу і ресурсів завдяки використанню готових компонентів, широкій екосистемі бібліотек і модулів, а також простоті інтеграції з базами даних і зовнішніми сервісами. Це робить платформу привабливою як для невеликих стартапів, так і для великих компаній, що працюють із високонавантаженими системами.

Особливого значення тема набуває в українських реаліях. В умовах цифровізації суспільства, зростання ринку IT-послуг та глобальної інтеграції, підготовка фахівців, здатних створювати сучасні багатосторінкові веб-додатки, є необхідною умовою розвитку національної економіки. Дослідження інструментів Next.js не лише сприятиме вдосконаленню освітніх програм, але й забезпечить практичні результати, що можуть бути впроваджені у сфері бізнесу, електронного урядування та освіти.

Метою дослідження є створення та аналіз багатосторінкового веб-додатка з використанням Next.js, що дозволяє оцінити ефективність поєднання сучасних архітектурних підходів і програмних інструментів у розробці високопродуктивних інформаційних систем. Для досягнення поставленої мети необхідно виконати низку завдань, серед яких визначення ключових тенденцій розвитку веб-технологій, порівняння багатосторінкових і односторінкових архітектурних моделей, аналіз можливостей сучасних фреймворків, розробка методики проєктування додатка на базі Next.js, створення прототипу

багатосторінкової системи, проведення його тестування та формування рекомендацій щодо практичного застосування отриманих результатів.

Об’єктом дослідження виступає процес розробки та функціонування багатосторінкових веб-додатків. Предмет дослідження – методи та інструменти, що використовуються для їх створення із застосуванням фреймворку Next.js.

У роботі використано комплекс методів дослідження: теоретичні методи, що охоплюють аналіз літературних джерел, класифікацію архітектурних підходів і систематизацію сучасних інструментів; емпіричні методи, серед яких реалізація прототипу, тестування його функціоналу й оцінювання продуктивності; а також прикладні методи, що включають використання програмних засобів моніторингу, моделювання та експериментального порівняння результатів.

Наукова новизна дослідження полягає у комплексному поєднанні теоретичного та прикладного підходів до вивчення багатосторінкових додатків, зокрема у застосуванні Next.js для інтеграції різних типів рендерингу – серверного, статичного та інкрементального – у межах одного прототипу.

Практичне значення отриманих результатів полягає у можливості їх використання при створенні освітніх платформ, корпоративних інформаційних систем, комерційних сервісів і проєктів електронного врядування. Запропонований підхід може слугувати основою для вдосконалення навчальних програм у закладах вищої освіти та підвищення кваліфікації розробників у сфері веб-технологій.

Структура кваліфікаційної роботи магістра відповідає меті та завданням дослідження. Робота складається зі вступу, трьох розділів, висновків, списку використаних джерел і додатків. У першому розділі висвітлено теоретичні та методологічні засади створення багатосторінкових додатків, проаналізовано їхні архітектурні особливості та тенденції розвитку сучасних веб-технологій. У другому розділі розглянуто методичні основи використання фреймворку Next.js, описано технологічні особливості його роботи, взаємодію з базами

даних та API, а також принципи організації архітектури багатосторінкового додатка. У третьому розділі представлено процес розробки багатосторінкового веб-додатка, описано реалізацію функціоналу, проведено тестування та дослідження продуктивності, а також здійснено порівняння отриманих результатів із традиційними підходами.

В кваліфікаційнїй роботі потрібно виконати наступні завдання:

– провести аналіз класифікації веб-додатків;

– визначити архітектурні підходи до проектування веб-додатків;

– вибрати інструменти, бібліотеки та фреймворк для розробки додатку;

– розробити організацію архітектури веб-додатка;

– провести аналіз реалізації та визначити вимоги до додатка;

– реалізувати сам додаток;

– протестувати та дослідити його ефективність.

Завершується робота висновками, у яких узагальнено результати проведеного дослідження та наведено рекомендації щодо подальшого використання й розвитку обраних технологій.

Апробація досліджень була проведена у вигляді статті та буде опублікована у виданні «Студентський вісник» ЛНТУ (додаток А).

РОЗДІЛ 1
АНАЛІЗ ПРОБЛЕМАТИКИ СТВОРЕННЯ БАГАТОСТОРІНКОВИХ ВЕБ-ДОДАТКІВ

1.1 Поняття та класифікація веб-додатків

У сучасних інформаційних системах веб-додатки є ключовим інструментом забезпечення інтерактивної взаємодії користувача з даними. У науковій літературі веб-додаток визначають як програмний комплекс, що працює у середовищі веб-браузера та забезпечує доступ до функціоналу через мережеве підключення. Сам веб-додаток можна розглядати як клієнт-серверну програму, де користувацький інтерфейс реалізується на стороні клієнта, а бізнес-логіка та зберігання даних – на сервері [1]. Така структура забезпечує універсальність і доступність програмного забезпечення, що й зумовило стрімке поширення веб-технологій.

Перші веб-додатки мали статичний вигляд і виконували здебільшого функцію надання інформації. Проте з розвитком динамічних технологій, таких як JavaScript та Ajax, відбувся перехід до інтерактивних систем, які здатні реагувати на дії користувача в режимі реального часу. Подальший розвиток призвів до появи односторінкових додатків (Single-Page Applications, SPA) та багатосторінкових додатків (Multi-Page Applications, MPA), які становлять два основні напрями сучасної веб-розробки.

SPA характеризуються тим, що завантажують єдину HTML-сторінку та динамічно оновлюють її вміст без перезавантаження всієї сторінки, що забезпечує швидку реакцію системи та кращий користувацький досвід . Водночас MPA передбачають розділення функціоналу на кілька сторінок із незалежними запитами до сервера, що дає змогу будувати складні корпоративні чи державні системи з чіткою структурою даних. Як відзначає у своїй статті Sehban Alan, архітектурне застосування: SPA зручні для інтерактивних додатків, тоді як MPA – для масштабних платформ із багатим функціоналом різниця між цими підходами визначає їх ефективність у різних сферах [2].

Ще одним важливим критерієм класифікації є підхід до рендерингу. Сучасні дослідження у сфері веб-технологій виокремлюють клієнтський, серверний і статичний рендеринг як основні моделі відображення контенту. Клієнтський рендеринг передбачає, що дані обробляються у браузері, тоді як серверний забезпечує формування готових HTML-сторінок на сервері, що є важливим для пошукової оптимізації та швидкості завантаження. Статична генерація, на думку Rudnyi M., дозволяє отримати максимальну продуктивність у випадках, коли дані не змінюються занадто часто, та і з точки зору SEO – це ідеальний варіант [3].

Функціональна класифікація веб-додатків охоплює інформаційні, транзакційні, комунікаційні та змішані системи. Інформаційні додатки орієнтовані на подання даних, транзакційні – на виконання операцій (банківські чи торговельні), комунікаційні – на взаємодію між користувачами (соціальні мережі, месенджери), а змішані поєднують кілька функцій одночасно. У сучасному світі більшість веб-додатків належать саме до змішаного типу, оскільки вони інтегрують інформаційний контент, транзакційні можливості та інструменти для комунікації.

Важливим критерієм класифікації також є рівень інтеграції з іншими системами. Сучасні веб-додатки дедалі частіше будуються на основі мікросервісної архітектури, що дозволяє інтегрувати їх із зовнішніми сервісами за допомогою API та хмарних технологій. Це відкриває перспективи для створення гнучких і масштабованих цифрових рішень, які можуть динамічно змінюватися відповідно до потреб користувачів і бізнесу.

Розвиток веб-додатків супроводжується появою різних архітектурних підходів, що відображають потреби суспільства й бізнесу у швидкому доступі до даних і сервісів. Традиційно веб-додатки будувалися за класичною багатошаровою архітектурою, де фронтенд і бекенд були жорстко пов’язані між собою. Такий підхід добре зарекомендував себе на ранніх етапах розвитку Інтернету, проте зі зростанням обсягів інформації та появою мобільних технологій він виявив низку обмежень. Зокрема, масштабування таких систем

вимагало значних ресурсів, а зміни в інтерфейсі або логіці потребували суттєвих зусиль усього колективу розробників [4].

На зміну класичним моделям прийшли сучасні підходи, що базуються на відокремленні клієнтської частини від серверної логіки. Цей поділ став можливим завдяки розвитку JavaScript-фреймворків, які дозволили реалізувати складну взаємодію прямо у браузері. Так виникли односторінкові додатки, що забезпечили новий рівень зручності користувацького досвіду. Однак згодом з'ясувалося, що для масштабних систем, орієнтованих на значну кількість користувачів і складну ієрархію даних, більш доцільним є повернення до багатосторінкових рішень, які легше структурувати й підтримувати.

Науковці також підкреслюють важливість класифікації веб-додатків за рівнем інтеграції з іншими сервісами. Сучасні тенденції передбачають перехід від монолітних систем до мікросервісної архітектури, що дозволяє кожному модулю функціонувати незалежно й взаємодіяти з іншими через API . Такий підхід забезпечує гнучкість і стійкість додатків до змін, оскільки кожен компонент можна модернізувати окремо без шкоди для всієї системи. Особливої актуальності це набуває для корпоративних і державних платформ, де стабільність та безпека є критично важливими параметрами.

З точки зору користувацького досвіду, різні типи веб-додатків мають власні переваги й недоліки. SPA забезпечують швидке реагування системи й зручність роботи з інтерактивним інтерфейсом, проте можуть мати проблеми з оптимізацією для пошукових систем. Натомість MPA дозволяють будувати логічно розділені модулі й підтримувати складні бізнес-процеси, але потребують більших зусиль для підтримки продуктивності. Саме тому сучасні фреймворки, зокрема Next.js, намагаються інтегрувати сильні сторони обох підходів, поєднуючи динамічність SPA із структурованістю MPA.

Історія розвитку веб-додатків демонструє послідовний перехід від простих статичних сайтів до складних інтерактивних платформ. На першому етапі (1990-ті роки) панували статичні HTML-сторінки, які виконували роль інформаційних вітрин. Другий етап (2000-ні) ознаменувався появою

динамічного контенту, що реалізувався за допомогою PHP, ASP.NET та інших серверних технологій. Третій етап (2010-ті роки) характеризувався бурхливим розвитком JavaScript-фреймворків, таких як Angular, React і Vue.js, які дозволили реалізувати односторінкові рішення. Сучасний етап розвитку пов’язаний із появою гібридних підходів, які поєднують динамічність SPA і масштабованість MPA, зокрема через використання Next.js [5].

Класифікація веб-додатків має також практичне значення для вибору оптимальних технологій під час розробки. Вибір між SPA і MPA, між централізованою чи мікросервісною архітектурою, між клієнтським або серверним рендерингом залежить від поставлених завдань, обсягів даних, навантаження на систему та очікувань користувачів.

У науковій літературі запропоновано кілька підходів до класифікації веб-додатків, які дають змогу краще зрозуміти їх різноманітність і специфіку використання. Однією з найпоширеніших є класифікація за функціональним призначенням. Вона виділяє такі типи веб-додатків: інформаційні, що орієнтовані на подання даних у різних форматах; транзакційні, що забезпечують виконання фінансових або торговельних операцій; комунікаційні, що створюють середовище для взаємодії між користувачами; змішані, що поєднують у собі кілька типів функцій [6]. Саме змішані додатки сьогодні становлять найбільшу групу, оскільки сучасний бізнес і суспільство потребують платформ, які одночасно виконують інформаційні, операційні й комунікаційні завдання.

Інший важливий підхід полягає у поділі веб-додатків за архітектурними ознаками на односторінкові (SPA) та багатосторінкові (MPA). Односторінкові додатки характеризуються завантаженням однієї сторінки, на якій відбувається динамічна зміна контенту за допомогою JavaScript і спеціалізованих фреймворків, таких як React, Angular чи Vue. Вони забезпечують високий рівень інтерактивності, що робить їх зручними для реалізації соціальних мереж, месенджерів і веб-редакторів. Водночас MPA передбачають наявність багатьох окремих сторінок, кожна з яких завантажується незалежно у відповідь на запит

користувача. Такі додатки зручніші для структурованого представлення великих обсягів інформації та забезпечують кращу масштабованість, тому вони широко використовуються у сфері електронної комерції, державного управління та корпоративних систем. В таблиці 1.1 показано характеристики, переваги, недоліки та приклади застосування різних типів архітектур.

Таблиця 1.1 – Класифікація веб-додатків за типом архітектури

Тип архітектури	Характеристика	Переваги	Недоліки	Приклади застосування
Класична багатосторінкова (MPA)	Кожна сторінка завантажується окремо, сервер формує HTML	Простота, SEO-оптимізація, зрозуміла структура	Високе навантаження на сервер, повільні переходи	Державні портали, корпоративні сайти
Односторінкова (SPA)	Завантажується одна сторінка, подальший контент оновлюється через JS	Висока інтерактивність, швидкість реакції	Проблеми з SEO, велика початкова вага сторінки	Соцмережі, месенджери
Гібридна (SSR + CSR)	Поєднання серверного рендерингу й клієнтської логіки	Баланс між SEO та швидкодією, масштабованість	Складність реалізації	Інтернет-магазини, Новинні портали
JAMstack	Статична генерація з API для динамічних даних	Висока продуктивніст, безпека	Обмежена динаміка без API	Блоги, маркетингові сайти

Важливо зазначити, що веб-додатки можна класифікувати і за критерієм безпеки. Оскільки вони працюють у глобальній мережі, доступ до якої може здійснюватися з будь-якого пристрою, питання автентифікації, шифрування та управління доступом стають першочерговими.

Тому в сучасній науковій літературі виділяють веб-додатки з базовим рівнем безпеки, орієнтовані на поширення відкритої інформації, і додатки з підвищеним рівнем захисту, призначені для роботи з персональними чи фінансовими даними [7].

Розвиток веб-додатків варто розглядати не лише як результат технічного прогресу, але й як наслідок еволюції соціальних і економічних процесів. Становлення інформаційного суспільства супроводжується стрімким зростанням обсягів даних, що потребують обробки, а також підвищеними

вимогами до якості обслуговування користувачів. У цій парадигмі веб-додатки постають як багатофункціональні інструменти, здатні забезпечити ефективну комунікацію між організаціями, державними структурами та громадянами. Згідно з аналітикою ринку McKinsey & Company, сучасні веб-системи перетворилися на ключовий елемент цифрової економіки, адже понад 70 % бізнес-процесів сьогодні так чи інакше інтегровані з веб-сервісами [8].

Важливо зазначити, що у процесі еволюції веб-додатки не лише ускладнювалися з технічної точки зору, але й змінювали роль користувача в інформаційній системі. Якщо на ранніх етапах користувач виступав лише споживачем інформації, то з розвитком інтерактивних технологій він став її активним творцем. Соціальні мережі, платформи для блогів, вікі-системи та хмарні сервіси створили умови для формування так званого «вебу другого покоління» (Web 2.0), у якому головним ресурсом виступає саме активність користувачів. Це визначило нові вимоги до веб-додатків – відтепер вони мають підтримувати масштабовану багатокористувацьку взаємодію, надійну систему збереження даних і механізми захисту від зовнішніх загроз [9].

Сучасні дослідження у сфері веб-інженерії пропонують розглядати веб-додатки як елементи комплексних інформаційних екосистем. У цьому контексті їх можна класифікувати на автономні та інтегровані. Автономні системи працюють у межах власної бази даних і логіки, тоді як інтегровані взаємодіють із низкою зовнішніх сервісів і підтримують складні бізнес-процеси. Яскравим прикладом інтегрованих веб-додатків є сучасні системи електронної комерції, що поєднують у собі модулі для управління товарами, логістикою, фінансовими транзакціями та клієнтською підтримкою.

Класифікація веб-додатків може також базуватися на рівні використання штучного інтелекту та аналітичних інструментів. Сучасні платформи дедалі частіше інтегрують алгоритми машинного навчання для персоналізації контенту, автоматизації бізнес-процесів і підвищення зручності користування. Це означає, що у майбутньому можна очікувати появи нового класу веб-додатків – «інтелектуальних систем», які об’єднуватимуть можливості

класичних архітектур із функціями прогнозної аналітики та автоматизованого прийняття рішень [10].

Таким чином, веб-додатки є багатогранним об’єктом дослідження, який можна класифікувати за функціональністю, архітектурними моделями, рівнем інтеграції, адаптивності, безпеки та застосуванням штучного інтелекту. Наукова новизна полягає у спробі інтегрувати ці підходи в єдину систему знань, яка дозволяє комплексно оцінити потенціал сучасних веб-технологій. Це створює підґрунтя для подальших досліджень інструментів на зразок Next.js, які здатні поєднувати різні підходи та забезпечувати створення високопродуктивних багатосторінкових додатків.

1.2 Архітектурні підходи до проєктування веб-додатків

Архітектура веб-додатка є визначальним чинником, який впливає на його продуктивність, масштабованість, надійність та зручність у користуванні. Під архітектурою у науковій та прикладній літературі розуміють структуровану організацію компонентів програмної системи, їхні взаємозв’язки та правила взаємодії. Для веб-додатків архітектура є не лише технічним каркасом, але й методологічною основою, яка забезпечує можливість ефективної реалізації бізнес-логіки, оптимізації запитів до серверів, обробки великих обсягів даних і підтримки одночасної роботи тисяч користувачів.

Упродовж розвитку веб-технологій сформувалося кілька архітектурних підходів, кожен з яких відображає певний етап еволюції програмної інженерії. Класичним рішенням тривалий час залишалася трифазна архітектура (three-tier architecture), що включає рівні представлення (інтерфейс), бізнес-логіки та доступу до даних. Такий поділ дозволив структурувати системи, знизити залежність між їхніми компонентами та спростити підтримку. Водночас зі зростанням складності додатків виявилися обмеження цієї моделі, зокрема труднощі масштабування та відсутність достатньої гнучкості[11].

На зміну класичним рішенням прийшли більш спеціалізовані моделі. Серед них важливе місце займає MVC (Model-View-Controller), яка поділяє додаток на модель (дані та бізнес-логіка), представлення (інтерфейс користувача) і контролер (зв’язок між ними) . MVC забезпечила зручність у розробці та тестуванні, а також дала змогу реалізувати незалежність між логікою та інтерфейсом. Проте для масштабних систем вона не завжди виявлялася достатньо ефективною, оскільки збільшення кількості модулів призводило до надмірної складності коду [12].

Подальшим розвитком стала модель MVVM (Model-View-ViewModel), яка дозволяє ефективніше керувати станом інтерфейсу, що особливо важливо для веб-додатків із високим рівнем інтерактивності. У цій архітектурі ViewModel виступає проміжним шаром між інтерфейсом і бізнес-логікою, забезпечуючи двосторонню прив’язку даних (data binding). Саме ця особливість зробила MVVM популярною у фреймворках, орієнтованих на створення SPA, наприклад Angular чи Vue.js.

Окрему групу складають архітектурні рішення, орієнтовані на масштабування і стійкість до навантажень. Найпоширенішим серед них став підхід мікросервісної архітектури, що передбачає розділення додатка на незалежні сервіси, кожен із яких виконує окрему функцію й взаємодіє з іншими через API . Мікросервіси дозволяють розробляти, оновлювати й масштабувати окремі частини системи незалежно, що значно підвищує її гнучкість. Така архітектура стала основою для багатьох сучасних високонавантажених систем, зокрема інтернет-магазинів, банківських платформ і соціальних мереж. Водночас її впровадження вимагає високого рівня організації процесів розробки та підтримки, а також використання контейнеризації та оркестрації.

Не менш важливим аспектом є підхід до рендерингу у веб-додатках. Традиційно виділяють серверний рендеринг (SSR), клієнтський рендеринг (CSR) та статичну генерацію (SSG). Серверний рендеринг забезпечує швидке завантаження сторінок і кращу індексацію в пошукових системах, що робить його придатним для інформаційних порталів та багатосторінкових систем.

Клієнтський рендеринг, навпаки, зменшує навантаження на сервер, але потребує додаткових ресурсів від клієнтського пристрою, що не завжди виправдано при обробці великих масивів даних. Статична генерація дозволяє створювати сторінки заздалегідь, що є ідеальним рішенням для ресурсів із відносно стабільним контентом.

Сучасним трендом є гібридні архітектури, що поєднують різні підходи залежно від потреб конкретного додатка. Наприклад, у Next.js реалізовано можливість комбінувати SSR, CSR і SSG, що дозволяє обирати оптимальну модель рендерингу для кожної сторінки чи навіть компонента . Це робить можливим створення веб-додатків із високою продуктивністю, масштабованістю та зручністю у використанні, що особливо важливо для багатосторінкових рішень.

Новизна сучасних архітектурних підходів полягає у тому, що вони орієнтовані не лише на технічну ефективність, але й на користувацький досвід. Сприйняття швидкості та зручності роботи системи для користувача є не менш важливим за її реальну продуктивність. Тому розробники дедалі частіше застосовують комплексний підхід, що включає оптимізацію фронтенду, бекенду та інтеграцію з аналітичними сервісами для постійного моніторингу ефективності.

Архітектурні підходи до проєктування веб-додатків у сучасних умовах варто розглядати як багатошарову систему, де кожен рівень відповідає за специфічний набір функцій. Якщо раніше увага зосереджувалася переважно на взаємодії клієнта й сервера, то сьогодні в поле зору дослідників і практиків потрапляють такі аспекти, як інтеграція з хмарними середовищами, управління потоками даних, оптимізація продуктивності й забезпечення інформаційної безпеки. Це пояснюється тим, що веб-додатки поступово перетворилися з простих інструментів подання інформації на комплексні інформаційні системи, що підтримують критично важливі бізнес-процеси.

Одним із напрямів, який активно розвивається останніми роками, є використання архітектур, орієнтованих на події (event-driven architecture). Цей

підхід базується на ідеї, що взаємодія між компонентами відбувається через повідомлення про події, які генеруються та обробляються асинхронно.

Варто наголосити на ще одному сучасному напрямі, який отримав значний розвиток у сфері веб-розробки, архітектурі, орієнтованій на API (API-first). Наукові дослідження та практика показують, що саме цей підхід забезпечує інтеграцію різних сервісів і дає змогу будувати розподілені системи, де фронтенд і бекенд розробляються незалежно, але взаємодіють через стандартизовані інтерфейси [13]. Для багатосторінкових додатків такий підхід є особливо цінним, оскільки він дає змогу легко розширювати функціональність, додавати нові модулі й підтримувати різні клієнтські застосунки, включно з мобільними.

Не менш важливим у науковій літературі визнається підхід Domain-Driven Design (DDD), що зосереджується на моделюванні предметної області системи. Автори цієї концепції, наголошують, що архітектура має будуватися відповідно до термінології та логіки предметної області, яку вона обслуговує. Для веб-додатків це означає створення модульної структури, де кожна частина системи відповідає за конкретний сегмент бізнес-логіки, що підвищує зрозумілість коду та спрощує його супровід.

Сучасні дослідження також акцентують увагу на безпекових аспектах архітектури. Веб-додатки є вразливими до широкого спектру атак, включно з міжсайтовим скриптінгом, SQL-ін’єкціями та DDoS. Саме тому архітектурні моделі дедалі частіше включають спеціальні рівні, орієнтовані на захист даних та управління доступом.

Зокрема, концепція Zero Trust Architecture, яка активно впроваджується в корпоративних системах, передбачає відмову від традиційного уявлення про безпечну внутрішню мережу й акцент на постійній верифікації кожного запиту.

У практичному вимірі архітектурні рішення дедалі частіше ґрунтуються на гібридних моделях, які інтегрують різні підходи. Наприклад, великі системи електронної комерції можуть поєднувати мікросервісну архітектуру для

обробки замовлень, event-driven підхід для управління потоками транзакцій, API-first для інтеграції з платіжними сервісами й JAMstack для забезпечення швидкого завантаження сторінок. Це демонструє, що архітектура веб-додатків сьогодні є не статичною схемою, а радше динамічною екосистемою, здатною адаптуватися до змінних умов.

Одним із підходів, який активно вивчається у науковій літературі, є архітектура, орієнтована на мікросервіси у поєднанні з контейнеризацією. Використання контейнерних технологій на кшталт Docker та Kubernetes дає змогу швидко масштабувати додатки залежно від попиту, що особливо актуально для електронної комерції та освітніх платформ [14].

З іншого боку, слід звернути увагу на розвиток так званих орієнтованих на дані архітектур (data-centric architecture). Такий підхід передбачає, що в основі системи лежить потік даних, а всі інші компоненти будуються навколо нього. Наприклад, у сфері аналітичних веб-платформ важливішим за користувацький інтерфейс стає здатність системи обробляти великі обсяги інформації в реальному часі. Це підтверджує досвід компаній, які впроваджують big data-технології у свої веб-додатки, оптимізуючи не лше процеси обслуговування клієнтів, але й внутрішні бізнес-аналітичні інструменти.

Підсумовуючи, можна стверджувати, що архітектурні підходи у веб-розробці еволюціонували від простих тришарових моделей до багаторівневих, динамічних та інтегрованих систем, здатних ефективно функціонувати в умовах глобальних цифрових трансформацій. Вони охоплюють класичні рішення, сучасні інноваційні концепції та перспективні моделі, що ще перебувають на стадії становлення. Це дозволяє розглядати архітектуру веб-додатків як один із ключових інструментів розвитку цифрової економіки та суспільства.

1.3 Огляд інструментів і фреймворків для розробки веб-додатків та постановка завдання

Сучасний розвиток веб-технологій засвідчує, що створення багатосторінкових додатків потребує використання потужних інструментів і фреймворків, які значно спрощують процес проєктування, розробки та супроводу програмного забезпечення. Якщо на ранніх етапах формування вебу ключовими засобами були HTML, CSS та JavaScript у чистому вигляді, то зростання складності та масштабів додатків зумовило потребу у використанні комплексних інструментів, здатних забезпечити гнучкість, продуктивність і надійність системи.

Фреймворки у веб-розробці виконують роль каркасу, який дозволяє стандартизувати процеси програмування, зменшити кількість рутинних операцій і зосередитися на реалізації бізнес-логіки. Використання фреймворків є закономірним етапом еволюції програмної інженерії, адже вони забезпечують повторне використання перевірених рішень і пришвидшують створення складних інформаційних систем. У цьому контексті важливо підкреслити, що сучасні багатосторінкові додатки будуються саме на основі фреймворків, які підтримують багаторівневу архітектуру, інтеграцію з базами даних.

Інструменти для створення веб-додатків можна поділити на три великі групи: клієнтські фреймворки, серверні фреймворки та універсальні рішення. Клієнтські орієнтовані на побудову інтерфейсу користувача та динамічне оновлення сторінок. Серед них найпоширенішими є React, Angular та Vue.js. Серверні фреймворки зосереджені на реалізації логіки додатка і взаємодії з базами даних; типовими прикладами є Express.js, Django та Ruby on Rails. Універсальні рішення поєднують можливості обох груп і дають змогу реалізовувати гібридні архітектурні підходи; серед них особливе місце займають Next.js, Nuxt.js та SvelteKit.

React, створений у компанії Facebook, є бібліотекою для побудови інтерфейсів, яка завдяки концепції віртуального DOM, забезпечує високу

швидкодію при зміні контенту на сторінці. Популярність React зумовлена його модульністю, простотою інтеграції та можливістю створення як односторінкових, так і багатосторінкових додатків у поєднанні з іншими інструментами. Утім, React не можна вважати повноцінним фреймворком, тому він часто використовується разом із Redux чи React Router, що забезпечують управління станом і навігацію.

Angular, підтримуваний Google, розглядається як комплексне рішення, що забезпечує повний набір інструментів для веб-розробки. Його особливістю є двостороння прив’язка даних, що дає змогу синхронізувати зміни у моделі та представленні. Angular активно застосовується у корпоративних системах та великих проєктах завдяки своїй масштабованості, проте його складність і крута крива навчання часто називаються недоліками.

Vue.js, навпаки, позиціонується як простіший і легший для освоєння інструмент. Його популярність особливо висока у невеликих командах та стартапах. Головними перевагами Vue.js є зрозумілий синтаксис, можливість поступового впровадження у вже існуючі проєкти та широкий набір сторонніх бібліотек. Саме ці якості зробили його популярним серед розробників освітніх і комунікаційних платформ [15].

Серед серверних фреймворків особливу роль відіграє Express.js, який побудований на основі Node.js. Він забезпечує мінімалістичний, але гнучкий підхід до створення серверних частин додатків, що робить його оптимальним вибором для розробки RESTful API. Django, розроблений мовою Python, надає повний набір готових рішень для авторизації, роботи з базами даних і адміністрування, завдяки чому використовується у складних багатосторінкових системах [16].

Окремої уваги заслуговують універсальні фреймворки, які стали основою для створення сучасних багатосторінкових додатків. Найбільш яскравим прикладом є Next.js, що поєднує можливості серверного рендерингу, статичної генерації та клієнтського рендерингу. Next.js дозволяє інтегрувати різні

архітектурні моделі, що робить його придатним для складних систем із високими вимогами до SEO-оптимізації.

Користь від аналізу інструментів і фреймворків полягає у комплексному порівнянні їхніх можливостей у контексті багатосторінкових додатків. Якщо попередні дослідження здебільшого зосереджувалися на розгляді окремих технологій, то сучасний підхід вимагає інтеграції цих знань і врахування міждисциплінарних аспектів. Розвиток веб-технологій у XXI столітті супроводжується зростанням складності додатків, що призводить до необхідності використання фреймворків та бібліотек, які забезпечують гнучкість, масштабованість і продуктивність систем, а їх особливості переваги та недоліки наведено в таблиці 1.2.

Таблиця 1.2 – Порівняння основних фреймворків для веб-розробки

Фреймворк	Тип	Основні особливості	Переваги	Недоліки
React	Бібліотека для UI	Компонентний підхід, Virtual DOM	Гнучкість, велика спільнота, інтеграція з іншими інструментами	Потребує додаткових бібліотек для навігації й стану
Angular	Повноцінний фреймворк	Двостороння прив’язка даних, модульність	Потужний, масштабований, підходить для великих проєктів	Високий поріг входження
Vue.js	Прогресивний фреймворк	Простота синтаксису, поступове впровадження	Легкий для навчання, швидкий, добре інтегрується	Менш поширений у великих корпораціях
Next.js	Універсальний фреймворк на базі React	SSR, SSG, ISR, маршрутизація за файлами	SEO-оптимізація, продуктивність, гнучкість	Складніший за React у налаштуванні
Django	Серверний фреймворк (Python)	Повний стек, ORM, безпека	Швидка розробка, надійність	Не підходить для складних SPA
Express.js	Серверний фреймворк (Node.js)	Мінімалістичний, REST API	Легкий, швидкий, інтегрується з будь-якими клієнтами	Потребує додаткових інструментів для складних систем

Сьогодні більшість розробників стверджують , що сучасна веб-розробка практично неможлива без використання таких фреймворків, адже вони стандартизують та автоматизовують процеси створення програмного забезпечення, оскільки багато рутинних процесів та дозволяють скоротити час розробки.Одним із найпопулярніших інструментів є бібліотека React, що застосовується для побудови користувацьких інтерфейсів. Її ключовою особливістю є використання віртуального DOM, що знижує навантаження на браузер і підвищує швидкодію.

Разом з тим, React у чистому вигляді часто потребує додаткових бібліотек для організації навігації чи управління станом, що ускладнює створення багатосторінкових додатків. Ці недоліки вирішуються завдяки використанню Next.js, який поєднує клієнтський і серверний рендеринг. Важливим напрямом досліджень є також оптимізація продуктивності веб-додатків, де розглядаються практики застосування Next.js для побудови високопродуктивних систем: динамічне імпортування, lazy loading, статична генерація сторінок та інкрементальна регенерація [17]. Такі підходи дозволяють створювати багатосторінкові додатки, які одночасно забезпечують стабільність, масштабованість і гнучкість.

Таким чином, сучасний інструментарій веб-розробки охоплює широкий спектр бібліотек та фреймворків, але саме Next.js завдяки універсальності й поєднанню різних моделей рендерингу набуває особливої значущості у контексті створення багатосторінкових систем.

Окрім основних бібліотек та фреймворків для побудови архітектури багатосторінкових додатків, важливу роль у їхньому життєвому циклі відіграють інструменти тестування, оптимізації та безперервної інтеграції. У дослідженнях з інженерії програмного забезпечення підкреслюється, що комплексна розробка складних веб-систем неможлива без інтеграції таких засобів у робочі процеси, оскільки вони підвищують надійність і зменшують ризики критичних помилок .

Важливим компонентом сучасної практики є тестування інтерфейсів. Для додатків, побудованих на React і Next.js, найбільш поширеними є Jest і Testing Library. Jest надає можливість модульного та інтеграційного тестування з мінімальними витратами на налаштування, тоді як Testing Library фокусується на поведінковому тестуванні, наближаючи процес перевірки до реального сценарію взаємодії користувача.

Іншим важливим аспектом є забезпечення безперервної інтеграції та доставки (CI/CD). Для цього активно застосовуються GitHub Actions, GitLab CI/CD та Jenkins.

Варто відзначити й інструменти для оптимізації продуктивності. Lighthouse, розроблений Google, є стандартом для аналізу веб-додатків за показниками швидкодії, доступності та SEO.

Особливу увагу привертає інтеграція фреймворків із хмарними платформами. У звіті компанії Vercel зазначається, що понад 60 % проєктів на Next.js розгортаються саме у хмарних середовищах AWS, Google Cloud чи Vercel Platform. Це пояснюється можливістю масштабування, використання CDN для швидшої доставки контенту та інтеграції з edge-технологіями [18].

На основі проведеного аналізу фреймворків можна зробити висновок що, Next.js показує найкращі резутати в порівнянні з іншими інструментами для веб-розробки, саме тому вибір саме його для практичної реалізації багатосторінкового веб-додатку повністю обґрунтований.

Постановка завдання:

– провести аналіз класифікації веб-додатків;

– визначити архітектурні підходи до проектування веб-додатків;

– вибрати інструменти, бібліотеки та фреймворк для розробки додатку;

– розробити організацію архітектури веб-додатка;

– провести аналіз реалізації та визначити вимоги до додатка;

– реалізувати багатосторінковий веб-додаток;

– протестувати та дослідити його ефективність.

РОЗДІЛ 2
ТЕОРЕТИЧНЕ ДОСЛІДЖЕННЯ ТА ПРАКТИЧНА РЕАЛІЗАЦІЯ БАГАТОСТОРІНКОВОГО ДОДАТКА З ВИКОРИСТАННЯМ NEXT.JS

2.1 Технологічні особливості Next.js

Фреймворк Next.js посідає особливе місце серед сучасних інструментів веб-розробки завдяки поєднанню простоти використання та широкого набору можливостей для створення високопродуктивних багатосторінкових додатків. Він побудований на базі бібліотеки React, проте суттєво розширює її функціонал, інтегруючи вбудовані механізми серверного рендерингу, статичної генерації сторінок і комбінованих методів відображення контенту. За даними досліджень, Next.js став одним із найбільш затребуваних фреймворків для комерційних і державних проєктів завдяки оптимальному поєднанню швидкодії, SEO-оптимізації та гнучкості архітектури.

Однією з ключових переваг Next.js є підтримка різних стратегій рендерингу контенту. Традиційні односторінкові додатки (SPA), створені лише на React, часто мають проблеми з індексацією в пошукових системах, оскільки весь контент завантажується на клієнті. Це ускладнює роботу SEO-алгоритмів і може знижувати доступність ресурсів для користувачів із повільними мережевими з’єднаннями. Next.js вирішує цю проблему завдяки Server-Side Rendering (SSR). Цей підхід передбачає формування HTML-контенту на сервері перед передачею його користувачеві, що знижує час до першого відображення сторінки (Time to First Byte, TTFB) та робить додаток доступним для індексації пошуковими системами.

Важливим доповненням до SSR є Static Site Generation (SSG). У цьому випадку HTML-сторінки формуються ще на етапі білду і кешуються для багаторазового використання. Це особливо корисно для інформаційних порталів, блогів та маркетингових сторінок, де контент оновлюється нечасто. За результатами експериментів, опублікованих у журналі International Journal of Innovative Science and Research Technology, використання SSG у Next.js

дозволяє знизити навантаження на сервер у середньому на 40 % у порівнянні з повністю динамічними системами.

Ще одним інноваційним рішенням у Next.js є Incremental Static Regeneration (ISR). На відміну від класичного SSG, який генерує сторінки один раз під час білду, ISR дозволяє оновлювати сторінки з певним інтервалом без повного перезапуску додатка. Це поєднує переваги статичної генерації та динамічного оновлення, роблячи систему одночасно продуктивною і сучасною.

У наукових та прикладних дослідженнях підкреслюється, що Next.js варто розглядати не лише як фреймворк для React, а як платформу, що реалізує багаторівневі стратегії рендерингу. Класичний підхід до створення багатосторінкових систем передбачав використання серверного рендерингу через окремі бекенд-фреймворки, такі як Django або Laravel.

Це забезпечувало стабільність і SEO-оптимізацію, проте суттєво знижувало інтерактивність. Запровадження SSR у Next.js дозволило поєднати переваги серверних і клієнтських технологій у межах єдиної екосистеми. SSR підвищує не лише технічні показники, але й рівень доступності ресурсів, що підтверджує його соціально-економічну значущість.

У випадку Static Site Generation (SSG) ключовою перевагою є зменшення навантаження на сервер за рахунок попередньої генерації сторінок. Крім того, SSG забезпечує надзвичайно швидке завантаження сторінок, оскільки користувач отримує готовий HTML без необхідності виконання складних серверних обчислень під час кожного запиту.

Проте класична статична генерація має суттєве обмеження: якщо контент оновлюється часто, необхідно щоразу проводити повний білд проєкту. Ця проблема вирішується завдяки Incremental Static Regeneration (ISR). У цьому випадку сторінки оновлюються частково, відповідно до визначеного часу життя (revalidate time). ISR є особливо перспективним напрямом для створення багатосторінкових додатків, що поєднують статичний і динамічний контент, Практичні дослідження показали, що ISR дозволяє підвищити продуктивність систем e-commerce у середньому на 18 % завдяки зниженню часу реакції при

оновленні даних [18]. Графічне представлення різниці між цими видами рендерингу представлене на русунку 2.1.

Оптимізація продуктивності є ще однією визначальною рисою Next.js. Серед інструментів, які надає цей фреймворк, варто виділити Image Optimization. Використання компоненту next/image дозволяє автоматично масштабувати зображення відповідно до пристрою користувача, застосовувати сучасні формати (WebP, AVIF) і завантажувати зображення «ліниво» (lazy loading). Також цей механізм оптимізації дає змогу керувати оброблені зображення, що пришвидшує повторне завантаження сторінок і зменшує кількість запитів до сервера.

Рисунок 2.1 – Потік даних і вибір стратегії рендерингу

Згідно з аналітикою компанії Vercel, впровадження цієї технології зменшує середній обсяг переданих даних на 25-30 % та прискорює відображення сторінки. Іншим механізмом підвищення ефективності є Code Splitting, який реалізується автоматично. Ця технологія розділяє програмний код на менші частини, завантажуючи лише ті модулі, які необхідні для поточної сторінки. Це дозволяє скоротити час початкового завантаження та підвищує загальну швидкодію системи.

Не менш важливою технологічною особливістю Next.js є інтегровані механізми кешування. За даними звіту компанії Vercel The State of Next.js 2023, середній показник TTFB для додатків на базі ISR і CDN становить 150-200 мс, тоді як для класичних серверних додатків цей показник часто перевищує 500 мс [19].

Поєднання SSR і ISR із сучасними CDN (Content Delivery Networks) дозволяє поширювати згенерований контент у глобальній мережі, що мінімізує затримку доступу користувачів до ресурсів незалежно від географічного положення, а усі ці технологічні особливості продемонстровано на рисунку 2.2.

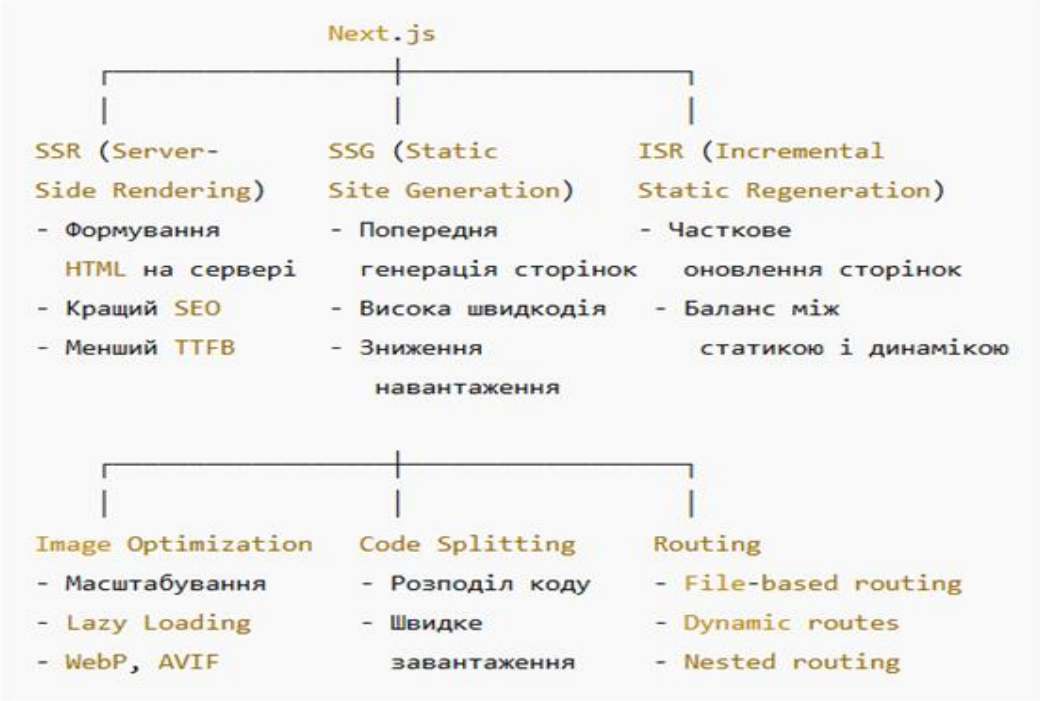

Рисунок 2.2 – Технологічні особливості Next.js

Отже, Next.js виступає не лише інструментом для побудови інтерфейсу, але й платформою для створення високопродуктивних систем завдяки підтримці різних моделей рендерингу, механізмів інкрементальної генерації, інтегрованих інструментів оптимізації й адаптивної маршрутизації. Його технологічні особливості дозволяють забезпечити баланс між швидкодією, SEO-оптимізацією та гнучкістю, що робить його базовим фреймворком для сучасних багатосторінкових додатків.

Окремої уваги заслуговує система роутингу в Next.js. Вона ґрунтується на файловій структурі: кожен файл у директорії pages/ автоматично стає маршрутом додатка. Це спрощує організацію багатосторінкових проєктів і зменшує потребу в додаткових бібліотеках для маршрутизації. Крім того, Next.js підтримує динамічні маршрути, які дозволяють створювати сторінки з параметрами (наприклад, /products/[id]), а також nested routing, що важливо для побудови складних ієрархій у корпоративних системах.

Завдяки файловій маршрутизації розробники можуть створювати не лише статичні шляхи, але й динамічні маршрути. Наприклад, файл pages/products/[id].js дозволяє відображати будь-яку статтю за її ідентифікатором. Це особливо важливо для багатосторінкових додатків з великим обсягом динамічного контенту, таких як блоги, e-commerce-платформи або освітні системи.

Ще однією важливою особливістю є nested routing (вкладені маршрути), коли сторінки можуть містити вкладені рівні переходів. Це дозволяє організовувати багаторівневу структуру додатка, яка відображає складність реальної предметної області.

У Next.js реалізовано й підтримку API Routes, які дозволяють створювати серверні функції прямо у тій самій структурі проєкту. Файл pages/api/users.js автоматично перетворюється на ендпоінт /api/users. Це суттєво знижує поріг входження для розробників, оскільки дає змогу поєднувати фронтенд і бекенд в єдиній системі. Використання API Routes у Next.js скорочує витрати на

інтеграцію з бекендом на 25 % у невеликих і середніх командах, де відсутні окремі бекенд-розробники.

Суттєвою перевагою маршрутизації в Next.js є також підтримка динамічного імпорту. Це дозволяє завантажувати сторінки чи компоненти лише у момент звернення до них, що зменшує час початкового завантаження.

Таким чином, маршрутизація у Next.js не лише спрощує процес побудови багатосторінкових додатків, але й інтегрує у собі інструменти для управління як статичними, так і динамічними сторінками, а також створення API на рівні фреймворку. Це робить Next.js одним із найбільш зручних і комплексних рішень для побудови масштабних систем [20].

2.2 Інтеграція Next.js з базами даних та API

Інтеграція з джерелами даних є одним із ключових етапів у створенні багатосторінкових додатків, оскільки саме вона визначає здатність системи працювати з великими обсягами інформації, забезпечувати її обробку та відображення користувачам. У випадку Next.js важливою перевагою є можливість використання як традиційних REST API, так і більш сучасних підходів на основі GraphQL, що надає розробникам широкий спектр інструментів для адаптації під конкретні проєктні вимоги.

Традиційна взаємодія через REST API залишається найпоширенішою у веб-розробці завдяки своїй простоті та універсальності. У контексті Next.js REST API реалізується за допомогою вбудованих API Routes, що дозволяють створювати серверні функції всередині додатка. Наприклад, файл pages/api/products.js може повертати JSON-дані з бази, які згодом відображаються на фронтенді. Це значно спрощує розробку, особливо для малих і середніх команд, де інтеграція бекенду та фронтенду часто виконується одними й тими самими розробниками.

Разом із тим, для складніших систем дедалі більшої популярності набуває GraphQL. Його ключова перевага полягає у можливості отримувати лише ті

дані, які потрібні для конкретного запиту, що знижує навантаження на мережу та оптимізує продуктивність.

Що стосується баз даних, Next.js найчастіше використовується у зв’язці з MongoDB та PostgreSQL. MongoDB як документоорієнтована база даних є особливо популярною у стартапах та динамічних додатках, де структура даних може змінюватися. Вона інтегрується з Next.js через бібліотеку Mongoose, яка забезпечує об’єктно-документне відображення (ODM) та дозволяє працювати з колекціями у вигляді JavaScript-об’єктів [21].

Натомість PostgreSQL є класичною реляційною базою даних, яка часто застосовується у корпоративних системах, де критично важливими є транзакційність, підтримка складних запитів і масштабованість. Для інтеграції з PostgreSQL у середовищі Next.js широко використовується ORM-інструмент Prisma [22].

За даними офіційної документації Prisma, його застосування дозволяє скоротити час розробки моделей і запитів у середньому на 20-30 % порівняно з класичними SQL-запитами. Важливо також, що Prisma підтримує типізацію TypeScript, що значно знижує ймовірність помилок у великих проєктах. Використання Prisma у поєднанні з Next.js та PostgreSQL демонструє кращі результати за метриками продуктивності порівняно з альтернативними ORM (Sequelize, TypeORM). У середньому Prisma обробляє транзакції на 15 % швидше, забезпечуючи при цьому простішу інтеграцію у CI/CD-процеси [23]. Це робить його оптимальним рішенням для розробки сучасних багатосторінкових додатків, де важливо поєднати швидкодію та стабільність.

Важливим аспектом є й безпека інтеграції. Застосування middleware, а також використання JWT-токенів для аутентифікації у GraphQL- і REST-запитах є стандартною практикою, що значно підвищує надійність системи.

Застосування REST API у Next.js має значну практичну цінність у тих випадках, коли необхідна проста та масштабована інтеграція з базами даних або сторонніми сервісами. Завдяки вбудованим API Routes усі запити (додавання товару, оформлення замовлення, перевірка статусу) можуть оброблятися

безпосередньо всередині Next.js-додатка. Це дозволяє зменшити кількість проміжних сервісів і пришвидшує взаємодію між фронтендом і бекендом.

Водночас, у випадках роботи з великими масивами даних дедалі більшої популярності набуває GraphQL. Його гнучкість полягає в тому, що клієнт може отримати рівно ті дані, які потрібні, без надмірних запитів. Наприклад, для освітньої платформи, яка відображає профілі студентів і статистику навчання, GraphQL дозволяє формувати складні запити: інформація про користувача, його курси та прогрес можуть бути завантажені одним викликом.

Порівняльні характеристики підходів REST та GraphQL у середовищі Next.js подано у таблиці 2.2, що дає змогу зробити висновок про доцільність їх використання залежно від архітектурних вимог проєкту.

Таблиця 2.2 – Порівняння REST та GraphQL у контексті Next.js

Критерій	REST API	GraphQL
Архітектурний підхід	Використовує набір ендпоінтів, кожен з яких відповідає певному ресурсу	Єдина кінцева точка, яка дозволяє формувати запити будь-якої складності
Гнучкість	Дані повертаються у фіксованому форматі, що часто призводить до «overfetching» або «underfetching»	Клієнт отримує лише необхідні дані, що знижує навантаження на мережу
Продуктивність	Висока у простих системах; проблеми при великій кількості запитів	Оптимізує взаємодію у складних системах; потребує додаткової обробки на сервері
Масштабованість	Добре підходить для невеликих і середніх проєктів	Ефективний у масштабних системах із великою кількістю взаємозв’язаних даних
Простота реалізації	Легка інтеграція через Next.js API Routes	Потребує налаштування GraphQL-сервера (Apollo, Yoga, Nexus)
Типові кейси	E-commerce каталоги, блоги, прості мобільні додатки	Освітні платформи, корпоративні системи, аналітичні панелі
Підтримка у Next.js	Вбудована (API Routes)	Можлива через інтеграцію бібліотек Apollo Client/Server, URQL тощо

Що стосується баз даних, то інтеграція Next.js із MongoDB і PostgreSQL часто залежить від специфіки завдань. MongoDB зручна там, де структура даних є динамічною та нерегламентованою.

PostgreSQL, своєю чергою, демонструє переваги у випадках, де критичною є цілісність транзакцій та робота зі складними зв’язками між таблицями. При цьому ORM Prisma дозволяла уникнути помилок завдяки строгій типізації запитів у TypeScript.

У практичному аспекті варто порівняти MongoDB та PostgreSQL у контексті використання з Next.js. MongoDB забезпечує більшу гнучкість у збереженні напівструктурованих даних, що зручно для стартапів і швидких прототипів. PostgreSQL натомість краще підходить для зрілих систем, де потрібна складна аналітика, звітність і сувора транзакційна обробка.

Інтеграційні можливості Next.js із зовнішніми джерелами даних зумовлюють його виняткову універсальність і визначають переваги порівняно з класичними фронтенд-бібліотеками. На відміну від React, який потребує сторонніх рішень для організації серверної логіки, Next.js забезпечує вбудовану підтримку API Routes, що функціонують за принципом мікросервісів у межах самого застосунку. Завдяки цьому поєднання клієнтської та серверної частини відбувається в єдиній архітектурі, що значно спрощує процес розробки та підвищує узгодженість рішень.

Засадниче значення у контексті взаємодії з базами даних і API мають методи getServerSideProps та getStaticProps, які надають можливість завантажувати дані на етапі серверного рендерингу або під час білду проєкту. Використання першого методу гарантує отримання актуального контенту в реальному часі, тоді як другий забезпечує попереднє формування сторінок із можливістю інкрементальної регенерації.

Особливістю Next.js є підтримка гібридного рендерингу (Hybrid Rendering), коли різні сторінки одного застосунку можуть використовувати відмінні моделі відображення. Це створює можливість одночасно реалізовувати статичну генерацію для контенту, що рідко змінюється, інкрементальну

регенерацію для динамічних ресурсів та SSR для критично важливих індивідуальних запитів.

Не менш важливою характеристикою є робота з середовищними змінними (environment variables), які гарантують безпечне зберігання ключів доступу до баз даних і зовнішніх сервісів. Дослідження у сфері кібербезпеки доводять, що використання таких механізмів у Next.js значно знижує ризик витоку конфіденційної інформації, а також відповідає сучасним принципам DevSecOps .

Значний інтерес становить також інтеграція Next.js з edge-computing. Завдяки нативній підтримці функцій Vercel, рендеринг і обробка API-запитів можуть здійснюватися на вузлах CDN, що скорочує час відповіді у глобальному масштабі. Це дозволяє знизити показник TTFB (Time to First Byte) на 40-45 % у порівнянні з централізованими серверними архітектурами. Така особливість є надзвичайно актуальною для масштабних багатосторінкових додатків, орієнтованих на міжнародний ринок.

Важливою тенденцією розвитку інтеграційних рішень у контексті Next.js є активне використання хмарних баз даних. Серед найбільш поширених рішень Firebase від Google та Supabase, яка позиціонується як open-source альтернатива. Firebase забезпечує масштабованість і вбудовану аутентифікацію користувачів, що робить її зручною для мобільних і веб-додатків, орієнтованих на мільйони користувачів. Supabase, натомість, базується на PostgreSQL та надає повний спектр функцій реляційної СУБД, зберігаючи простоту інтеграції через REST і GraphQL-ендпоінти [24].

Ще одним стратегічним напрямом є розвиток serverless-архітектури, яка тісно інтегрується з Next.js. Завдяки API Routes та edge-функціям, запити до баз даних можуть оброблятися у вигляді невеликих серверних функцій, що виконуються лише під час звернення. Це дозволяє зменшити витрати на інфраструктуру, оскільки сервер не працює постійно.

Окремої уваги заслуговує застосування Next.js у концепції API-first. У такому підході розробка програмного забезпечення розпочинається з

проектування API, а не користувацького інтерфейсу. Next.js у цьому контексті виступає фронтенд-фреймворком, що може гнучко взаємодіяти з будь-яким зовнішнім API, забезпечуючи модульність та незалежність від конкретної СУБД чи бекенд-сервісу. Це особливо актуально в умовах поширення headless CMS (Content Management Systems), таких як Strapi чи Contentful.

З наукової точки зору новизна інтеграційних можливостей Next.js полягає у його здатності поєднувати різні парадигми роботи з даними – реляційні, документоорієнтовані, API-орієнтовані та serverless-рішення – в єдиній архітектурі. Такий синтез забезпечує універсальність і дозволяє проектувати системи, здатні масштабуватися під конкретні потреби замовників. Більше того, це відкриває перспективи для досліджень у напрямку адаптивної оптимізації даних: використання гібридних баз у межах одного проєкту, автоматизованого вибору способу рендерингу сторінок залежно від навантаження, а також інтеграції зі штучним інтелектом для динамічного управління API-запитами.

2.3 Організація архітектури багатосторінкового додатка

Архітектура веб-додатка визначає не лише його функціональні можливості, але й параметри продуктивності, масштабованості та супровідності. У випадку багатосторінкових систем побудова архітектури ускладнюється потребою забезпечення цілісності даних між численними сторінками, оптимізації навігації та підтримки повторного використання компонентів. Фреймворк Next.js, будучи розширенням бібліотеки React, пропонує стандартизований підхід до структуризації додатків, що надає можливість поєднати простоту організації з високим рівнем технологічної гнучкості.

Однією з базових складових архітектури Next.js є ієрархія директорій, яка визначає основні функціональні блоки системи. Каталог pages/ відповідає за маршрутизацію і формування сторінок, components/ – за повторно використовувані елементи інтерфейсу, public/ – за статичні ресурси, styles/ – за

стилізацію. Окреме місце відведене каталогу api/, що забезпечує створення серверних функцій для обробки запитів. Така структурна модель дозволяє розробникам підтримувати високу прозорість у побудові складних багатосторінкових систем, що є важливим фактором у командній розробці.

З погляду інженерії програмного забезпечення компонентний підхід, реалізований у React і розвинутий у Next.js, створює умови для досягнення модульності та інкапсуляції. Кожен елемент інтерфейсу функціонує як окремий компонент із власною логікою, що може бути інтегрований у різні сторінки без дублювання коду. Подібний підхід відповідає принципам повторного використання (reusability) та низької зв’язності (low coupling), що вважаються основними критеріями якісної архітектури. У результаті багатосторінковий додаток може розвиватися поступово, шляхом додавання нових компонентів без необхідності суттєвої модифікації існуючих структур.

Істотним фактором архітектурної організації є впровадження статичної типізації через використання TypeScript. У поєднанні з Next.js типізація дозволяє не лише знизити ймовірність помилок, але й підвищити передбачуваність у розробці великих систем. TypeScript інтегрується у Next.js на рівні конфігурації та забезпечує контроль сумісності компонентів, що особливо важливо у багатокомпонентних системах з великою кількістю залежностей.

Особливу увагу слід звернути на організацію управління станом у багатосторінкових додатках. Для невеликих систем достатньо локального керування станом засобами React, проте для складних додатків доцільним є застосування бібліотек Redux або React Query. Вибір інструментів визначається необхідністю забезпечення цілісності даних між численними сторінками та підвищенням ефективності роботи з асинхронними запитами.

Сучасна організація архітектури багатосторінкових додатків не обмежується лише структурою директорій та компонентною моделлю. Вона має враховувати вимоги до автоматизації процесів розгортання, контролю якості та забезпечення безпеки системи. У цьому контексті важливу роль

відіграє інтеграція Next.js у середовище DevOps та CI/CD, що дозволяє реалізувати принципи безперервної інтеграції та доставки (Continuous Integration/Continuous Delivery).

Важливим завданням є також організація архітектури даних у межах Next.js. Вибір між централізованими базами даних, такими як PostgreSQL, і розподіленими рішеннями, як-от MongoDB Atlas чи Firebase, суттєво впливає на архітектурні рішення.

Не менш важливим аспектом є архітектурна безпека, що включає роботу із середовищними змінними, організацію доступу до API та захист користувацьких даних. Використання middleware у Next.js дозволяє реалізувати централізований контроль доступу, інтегруючи механізми аутентифікації та авторизації на рівні архітектури. Такий підхід підвищує узгодженість безпекових політик і зменшує ризик помилок, пов’язаних із дублюванням логіки на рівні окремих компонентів.

У контексті масштабних багатосторінкових систем дедалі більшого значення набуває підтримка мікрофронтенд-архітектури, що передбачає розподіл додатка на окремі автономні модулі. Хоча Next.js першочергово орієнтований на монолітні додатки, його поєднання з інструментами модульного завантаження та code splitting дає змогу реалізувати гібридні архітектурні рішення.

Організація архітектури багатосторінкових веб-додатків у середовищі Next.js демонструє складний багаторівневий характер, що поєднує структурні, технологічні та інфраструктурні підходи. З точки зору структурної організації, Next.js пропонує ієрархію директорій, яка забезпечує передбачуваність і стандартизацію проєктів. Це сприяє зниженню ризиків, пов’язаних із фрагментацією коду, і дозволяє формувати масштабовані рішення навіть у великих командах. Така модель узгоджується з принципами інженерії програмного забезпечення, зокрема модульності та інкапсуляції, що підтверджується емпіричними дослідженнями.

Важливим елементом архітектурної організації є компонентна модель. Вона не лише полегшує повторне використання коду, але й сприяє створенню архітектур, які відповідають принципам «atomic design». Це дозволяє структурувати систему за рівнями складності – від елементарних атомів до комплексних інтерфейсів.

У контексті сучасних тенденцій особливого значення набуває інтеграція Next.js із TypeScript, яка забезпечує статичну типізацію. Дослідження підтверджують, що використання типізації скорочує кількість помилок у промислових системах і підвищує продуктивність командної роботи. Це особливо важливо для багатосторінкових додатків, де наявність складних залежностей може стати причиною зниження стабільності системи.

Крім того, організація архітектури у Next.js не може розглядатися без урахування аспектів інтеграції з DevOps і CI/CD-практиками. Автоматизоване тестування, деплоймент і контроль версій стали невід'ємною частиною архітектурного процесу. Як свідчать дослідження, застосування інструментів CI/CD у поєднанні з Next.js знижує час виходу продукту на ринок та сприяє формуванню більш стійких архітектур. Це свідчить про перехід від традиційного розуміння архітектури як статичної структури до трактування її як динамічного процесу, інтегрованого в життєвий цикл програмного забезпечення.

Саме поєднання цих вимірів надає архітектурі Next.js наукової новизни, адже воно демонструє перехід від ізольованого підходу до розробки інтерфейсів до комплексного архітектурного мислення. Зважаючи на це, Next.js можна розглядати як приклад еволюції веб-фреймворків у напрямку інтегрованих архітектурних платформ, що відповідають вимогам сучасного інформаційного

Одним із ключових підходів, який здобув попередню популярність у середовищі Next.js, є чиста архітектура (Clean Architecture). Вона передбачає розділення системи на шари із низькою залежністю між ними: шар бізнес-логіки, шар користувацького інтерфейсу, шар інфраструктури. У контексті

Next.js застосування чистої архітектури дозволяє ізолювати компонент бекенду (API Routes, функції, взаємодія з базами) від представницького шару (сторінки, компоненти) та мінімізувати залежність від фреймворку.

Додатковим важливим елементом архітектури є обробка помилок та обробка винятків на рівні напрямків маршрутів або компонентів. У великих системах виникають ситуації, коли помилка в одному модулі не повинна зламати весь застосунок. У Next.js це досягається за допомогою компонентів «Error Boundary», Middleware та спеціальних конструкцій для маршрутизації помилок (наприклад, error.js або not-found.tsx у новому App Router). Архітектура має передбачати, як розгортати fallback-повідомлення, як обробляти непередбачувані ситуації, і як логувати збої, щоб мінімізувати вплив на користувацький досвід.

Узагальнюючи, організація архітектури багатосторінкового додатка на Next.js має спиратися на кілька взаємопов’язаних пластів:

– структурна організація модулів і компонентів, що відображає функціональну логіку;

– рендерингові та гідраційні стратегії, які враховують продуктивність фронтенду;

– розподіл відповідальностей для обробки помилок, кешування, інфраструктурних функцій (middleware, edge, API);

– урахування інтеграційних підходів (мікрофронтенди, адаптивне гідрування, модульні інтерфейси).

У контексті багатосторінкових додатків, особливо тих, які обслуговують складні бізнес-моделі (електронна комерція, освітні платформи, медіа-сервіси), важливим компонентом є уніфікований підхід до керування станом. Для великих систем на базі Next.js рекомендовано використовувати такі бібліотеки як Zustand, Redux Toolkit або Jotai, залежно від складності та потреб ізоляції станів [25]. Застосування локалізованого або глобального сховища дозволяє оптимізувати обмін даними між сторінками та модулями, уникаючи надмірного дублювання запитів або гідрації.

РОЗДІЛ 3
ЕКСПЕРЕМЕНТАЛЬНЕ ДОСЛІДЖЕННЯ РЕЗУЛЬТАТИВНОСТІ РОЗРОБКИ ТА АНАЛІЗУ БАГАТОСТОРІНКОВОГО ДОДАТКА НА БАЗІ NEXT.JS

3.1 Аналіз результатів реалізації багатосторінкового додатка на базі Next.js

Розробка багатосторінкових веб-додатків сьогодні є одним із найактуальніших напрямів у сфері інженерії програмного забезпечення, оскільки сучасні користувачі очікують одночасно високої продуктивності, зручного інтерфейсу, інтерактивності та SEO-оптимізованого контенту. Вибір Next.js як основного фреймворку обумовлений його здатністю поєднувати різні стратегії рендерингу – серверне (SSR), статичне (SSG) та інкрементальне (ISR), що дозволяє оптимізувати роботу як невеликих, так і масштабних систем. Постановка задачі у межах даного дослідження передбачає формулювання технічного завдання на створення багатосторінкового додатка, який відповідатиме функціональним та нефункціональним вимогам сучасної веб-розробки.

Метою роботи є проєктування та реалізація веб-додатка, що забезпечує поєднання швидкодії, зручності використання та можливостей масштабування. Це передбачає формування вимог до структури проєкту, визначення основних модулів системи, специфікацію інтерфейсів та навігаційних схем. Згідно зі стандартами інженерії ПЗ, насамперед необхідно скласти технічне завдання (Software Requirements Specification, SRS), яке має бути повним, несуперечливим та придатним до верифікації. У цьому документі окреслюються цілі, вимоги, обмеження та припущення, що є основою для подальшого кодування.

Функціональні вимоги передбачають реалізацію базових користувацьких сценаріїв, включаючи реєстрацію, автентифікацію, управління обліковим записом, перегляд та створення контенту, фільтрацію та пошук даних,

можливість залишати коментарі. Окрему увагу приділено інтеграції з API: внутрішні маршрути Next.js (API Routes) забезпечують швидку взаємодію між фронтендом і бекендом, а також дозволяють будувати додаток у логіці JAMstack-архітектури.

Нефункціональні вимоги описують якісні характеристики системи. Вони включають продуктивність (середній час відповіді на запит не більше 200-300 мс для SSR-сторінок), масштабованість (підтримка балансування навантаження та CDN), безпеку (захист від XSS, CSRF, ін’єкцій, шифрування даних), а також юзабіліті (адаптивність інтерфейсу, зручність користування на різних пристроях). Оскільки Next.js інтегрується з Lighthouse для вимірювання показників продуктивності, до нефункціональних вимог також належить досягнення оцінки не нижче 90 балів у відповідних категоріях.

Особливістю постановки задачі є визначення ролей користувачів. У системі передбачено три основні ролі: звичайний користувач (може переглядати та коментувати контент), автор (створює та редагує публікації) та адміністратор (має доступ до управління користувачами й контентом). Така модель базується на принципах Role-Based Access Control (RBAC), що є стандартом для сучасних веб-систем [27].

У технічному завданні важливим етапом є формування обмежень і припущень. До них належать: використання саме Next.js як основного фреймворку; застосування PostgreSQL або MongoDB як системи управління базами даних; інтеграція з ORM Prisma для безпечної роботи з моделями даних; а також залежність від зовнішніх API для реалізації сервісів, таких як авторизація через GitHub. Ці обмеження визначають архітектурні рішення й впливають на подальше тестування.

Навігаційна схема майбутнього додатка будується на принципах файлового роутингу, характерного для Next.js. У додатку передбачені основні маршрути: головна сторінка, список об’єктів, сторінка деталей, профіль користувача, сторінки реєстрації та входу, адміністративна панель.

Використання динамічних маршрутів ([id].tsx) дозволяє формувати контент із бази даних у реальному часі, що підвищує гнучкість архітектури .

Складання прототипів інтерфейсу є обов'язковою частиною технічного завдання. Прототипи (wireframes) дозволяють оцінити логіку навігації, послідовність взаємодії з системою та передбачити проблеми юзабіліті ще до етапу реалізації. Це відповідає кращим практикам User-Centered Design, які довели свою ефективність у веб-розробці.

Однією з ключових особливостей сучасного етапу розвитку веб-технологій в Україні є поєднання глобальних практик проєктування програмного забезпечення з адаптацією до національних потреб освітніх, бізнесових і соціальних систем. Саме тому постановка задачі при створенні багатосторінкового додатка має враховувати не лише технічні аспекти, але й педагогічні, соціальні та культурні чинники.

З точки зору методології інженерії ПЗ, технічне завдання має формуватися як документ, що визначає:

– цілі створення системи;

– функціональні вимоги;

– нефункціональні вимоги;

– обмеження та припущення;

– прототипи та схеми взаємодії.

Функціональні вимоги для багатосторінкового додатка на базі Next.js у нашому випадку мають включати:

– підтримку автентифікації з використанням як класичної моделі (логін-пароль), так і інтеграції з OAuth 2.0 (GitHub);

– можливість керування контентом через панель адміністратора (CRUD-операції над статтями, товарами або іншими сутностями);

– реалізацію динамічного роутингу, що забезпечує відображення сторінок контенту за ідентифікатором;

– інтеграцію з REST API або GraphQL для зручного доступу до бази даних;

– систему ролевого доступу (RBAC), де адміністратор має повний контроль, автор – обмежені функції створення контенту, а користувач – лише перегляд і взаємодію;

– можливість залишати коментарі та відгуки, що сприяє побудові інтерактивної спільноти;

– seo-оптимізацію за допомогою SSR і ISR, формування метаданих для кожної сторінки.

Нефункціональні вимоги повинні враховувати особливості продуктивності, масштабованості й доступності системи. У цьому контексті доцільно сформувати такі критерії: час відгуку сторінки не повинен перевищувати 1 секунди при навантаженні у 100 одночасних користувачів; рівень доступності системи має бути не нижче 99,5 % на місяць; інтерфейс повинен коректно відображатися на пристроях з різною роздільною здатністю екрану.

Особливу увагу необхідно приділити безпеці. У нашому випадку ТЗ має передбачати:

– застосування HTTPS для всіх запитів;

– захист від XSS, CSRF, SQL/NoSQL ін’єкцій;

– використання токенів доступу (JWT) та середовищних змінних для зберігання секретів;

– регулярне логування й аудит дій користувачів.

Зміст технічного завдання також повинен включати навігаційну схему системи. Вона будується за принципами файлового роутингу, характерного для Next.js: /index (головна сторінка), /checkout/[id] (сторінка замовлення), /user/ (сторінка користувача). Тому sitemap має бути інтегрованим у ТЗ ще до початку програмної реалізації.

Не менш важливим є створення прототипів інтерфейсу (wireframes). Прототипування дає змогу виявити проблеми з юзабіліті ще до початку кодування, а також отримати зворотний зв’язок від користувачів або замовників.

Важливим аспектом технічного завдання для багатосторінкового додатка є опис предметної області, оскільки саме вона визначає набір функціональних сценаріїв і структуру майбутньої системи. Наприклад, якщо йдеться про освітню платформу, то ключовими об’єктами будуть «курс», «лекція», «тест», «студент» і «викладач»; для електронної комерції – «товар», «замовлення», «кошик», «платіж». Чітке визначення таких сутностей дозволяє ще на етапі ТЗ формувати модель даних, яка згодом лягає в основу структуру бази, забезпечуючи узгодженість логіки та спрощуючи подальшу розробку.

Особливу увагу слід приділяти сценаріям взаємодії користувача з додатком. Вони відображають дії користувачів та реакцію системи на ці дії. Такі сценарії є ключовими при формуванні як функціональних, так і нефункціональних вимог, приклад логіко-функціональних архітектури додатку зображено на рисунку 3.1.

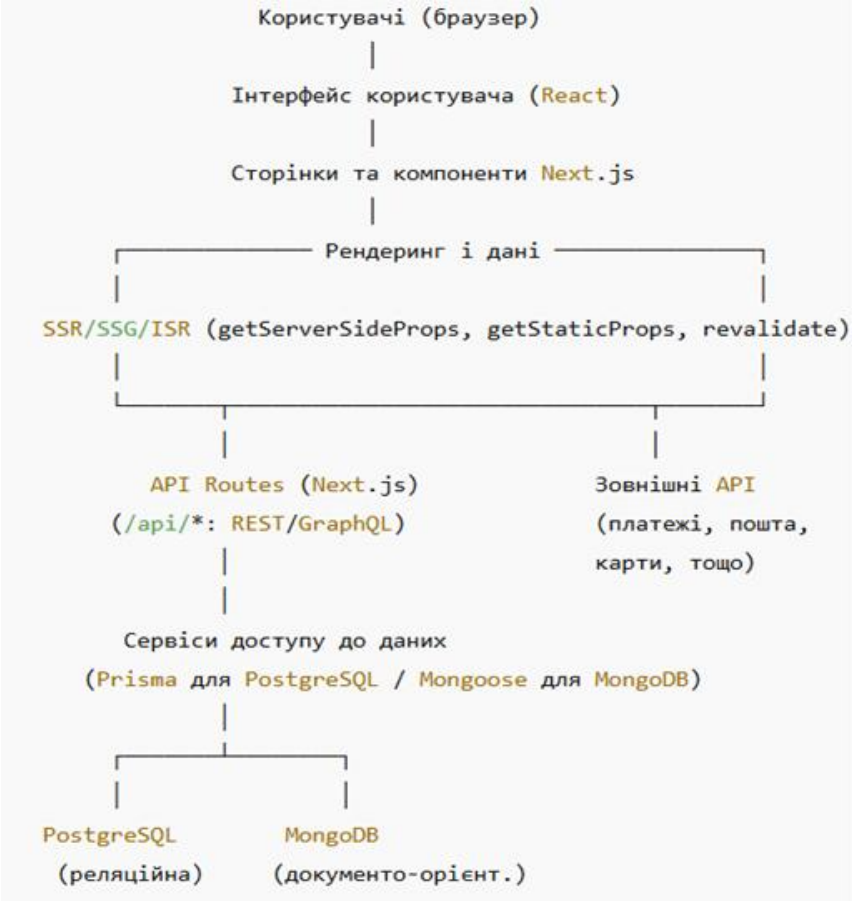

Рисунок 3.1 – Логіко-функціональна архітектура системи

Технічне завдання має включати також опис технологічного середовища реалізації. Для додатка на базі Next.js передбачається використання:

– react-компонентів, організованих за принципами модульності та повторного використання;

– typeScript, який забезпечує статичну типізацію й скорочує кількість помилок під час розробки;

– prisma ORM для взаємодії з базою PostgreSQL або MongoDB;

– nextAuth.js для реалізації механізмів автентифікації;

– zustand або Redux Toolkit для управління станом на клієнті;

– lighthouse для вимірювання продуктивності й доступності.

Серед нефункціональних вимог ключове місце займає питання продуктивності. Відповідно до рекомендацій Google Web Vitals, для сучасних веб-додатків критичними метриками є LCP (Largest Contentful Paint), FID (First Input Delay) та CLS (Cumulative Layout Shift). Ці показники визначають швидкість відображення основного контенту, затримку першої взаємодії користувача та стабільність макета сторінки. Досягнення оптимальних значень (LCP < 2,5 с, FID < 100 мс, CLS < 0,1) має бути чітко зафіксоване у технічному завданні, оскільки від цього залежить не лише зручність використання системи, а й її пошукова оптимізація.

Важливою частиною ТЗ є також безпека. Тому у проєкті має бути передбачено: використання HTTPS, обов’язкове шифрування паролів за допомогою bcrypt або Argon2, захист від CSRF-атак через використання токенів, а також регулярні перевірки вхідних даних на рівні серверних маршрутів.

Ще одним завданням у межах технічного завдання є створення плану тестування системи. Згідно з кращими практиками, тестування повинно включати:

– модульні тести для перевірки компонентів (Jest, React Testing Library);

– інтеграційні тести для перевірки взаємодії з базою та API;

– end-to-end тести для перевірки користувацьких сценаріїв (Cypress, Playwright).

Це дозволяє забезпечити повне покриття вимог і гарантувати їхню верифікацію. За даними досліджень IEEE, вартість виправлення помилки на етапі експлуатації зростає в 10-15 разів порівняно з етапом проєктування, тому якісне тестування закладається в технічне завдання ще на початку.

Не менш суттєвою складовою є формування обмежень і припущень. У нашому випадку до них належать: використання саме Next.js як основи, застосування PostgreSQL як основної бази, а також припущення про те, що мінімальні серверні ресурси становлять 2 vCPU та 4 GB RAM. Ці обмеження допомагають правильно оцінити навантаження та масштабованість системи.

Постановка задачі у межах розробки багатосторінкового додатка з використанням Next.js демонструє, що успіх системи залежить від комплексного підходу: від формалізації предметної області до створення прототипів та схем взаємодії.

3.2 Реалізація багатосторінкового додатка з використанням Next.js

Реалізація багатосторінкового додатка є ключовим етапом дослідження, оскільки саме на цьому етапі формуються всі передбачені технічним завданням функції, інтегрується технологічний стек і здійснюється перевірка життєздатності архітектури.

Першим етапом реалізації виступає створення структури проєкту. Next.js має вбудовану систему файлового роутингу, що дозволяє кожному файлу в директорії pages відповідати певному маршруту. Для багатосторінкового додатка було визначено такі основні сторінки: головна (/index), сторінка замовлення (/checkout), сторінка користувача (/user), неавторизований користувача (/not-auth), модальне вікно товару (/product/[id]). Використання динамічних сегментів маршруту забезпечує універсальність і гнучкість у

відображенні даних. Як наприклад реалізація динамічних маршрутів у кошику, наведена в лістингу 3.1.

Лістинг 3.1 – Реалізація роутингу /cart/[id] для оновлення елементів у кошику

```
import { prisma } from '@/prisma/prisma-client';
import { updateCartTotalAmount } from '@/shared/lib/update-cart-total-
amount';
import { NextRequest, NextResponse } from 'next/server';

export async function PATCH(req: NextRequest, { params }: { params: { id:
string } }) {
  try {
    const id = Number(params.id);
    const data = (await req.json()) as { quantity: number };
    const token = req.cookies.get('cartToken')?.value;

    if (!token) {
      return NextResponse.json({ error: 'Cart token not found' });}

    const cartItem = await prisma.cartItem.findFirst({
      where: {
        id,
      },});

    if (!cartItem) {
      return NextResponse.json({ error: 'Cart item not found' });
    }

    await prisma.cartItem.update({
      where: {
        id,
      },
      data: {
        quantity: data.quantity,
      }, });

    const updatedUserCart = await updateCartTotalAmount(token);

    return NextResponse.json(updatedUserCart);
  } catch (error) {
    console.log('[CART_PATCH] Server error', error);
    return NextResponse.json({ message: 'Не вдалося оновити кошик' }, {
status: 500 });
  }
}
```

кінець лістингу 3.1

Далі реалізовується інтеграція з базою даних. У якості СУБД використано PostgreSQL, що завдяки своїй реляційній природі й розвинутим можливостям (транзакційність, підтримка JSONB) забезпечує надійність і масштабованість. Для взаємодії з базою застосовано Prisma ORM, яка дає можливість створювати типобезпечні запити й автоматично генерує типи для TypeScript. У файлі schema.prisma описуються моделі сутностей: User, Category, Product, ProductItem, Ingradient, Cart, CartItem, Order, VerificationCode, OrderStatus, в додатку Б наведено код програми, який описує зв'язки між цими моделями. Міграції виконуються за допомогою CLI Prisma, що дозволяє відстежувати зміни в структурі бази. У разі використання MongoDB як документо-орієнтованої бази даних, інтеграція здійснюється через Mongoose, що зручний для роботи з даними у форматі JSON.

На рівні бекенду реалізовано API Routes, що дають можливість створювати серверні ендпоінти прямо у межах Next.js. Це дозволяє виконувати CRUD-операції: замовлення (POST /api/checkout), оновлення (PATCH /api/card/[id]) та видалення (DELETE/api/card/[id]) товарів, і як показано в додатку В отримання та фільтрація товарів (GET /lib/find-teas). Створення API Routes працюють як проксі між клієнтом і базою даних, забезпечуючи централізовану валідацію запитів та обробку помилок.

Окрему увагу приділено системі автентифікації й авторизації. Для цього застосовано бібліотеку NextAuth.js, яка підтримує кілька стратегій входу: email/пароль, OAuth (GitHub) і JWT.

Клієнтська частина додатка базується на React-компонентах. Управління станом реалізоване за допомогою Zustand – легкого state manager, що дозволяє уникнути надмірної складності Redux.

Інтеграція зовнішніх API відіграє важливу роль у сучасних веб-додатках. У рамках реалізації передбачено підключення платіжної системи, картографічних сервісів та поштового сервісу для підтвердження реєстрації. Для комунікації з API використовується Axios, що спрощує роботу з

HTTP-запитами та дозволяє реалізувати інтерсептори для централізованої обробки помилок.

На рівні UI реалізація інтерфейсу відбувається з урахуванням принципів адаптивного дизайну (responsive design). Використовуються ShadCN та Tailwind CSS, що дозволяють швидко створювати уніфіковані стилі для різних пристроїв. При цьому дизайн орієнтується на концепцію mobile first, яка вважається стандартом у сучасній розробці.

Особливе значення має організація обробки помилок і логування. Використовується Sentry як сервіс для моніторингу, що дозволяє відстежувати помилки в реальному часі.Реалізація багатосторінкового додатка завершується впровадженням тестування. Використовуються Jest і React Testing Library для модульних тестів, Cypress для end-to-end перевірок користувацьких сценаріїв. Це дає змогу верифікувати виконання як функціональних, так і нефункціональних вимог, окреслених у попередньому підрозділі.

У процесі реалізації особлива увага приділяється архітектурі додатка. Структура побудована за принципами розділення відповідальності (Separation of Concerns), що передбачає чітке розмежування між рівнями даних, логіки й представлення.

Директорія /pages/api містить серверні функції, які виконуються безпосередньо у Node.js середовищі, що дозволяє уникнути окремого бекенду. API Routes використовуються для виконання CRUD-операцій, до прикладу запит який наведено в лістингу 3.2.

Кожен ендпоінт проходить валідацію та обробку помилок. Для централізованої логіки передбачено middleware, яке перевіряє авторизацію користувачів і рівень їхніх прав доступу.

Лістинг 3.2 – Запит до БД через Prisma для отримання продуктів за заданими параметрами

```
import { prisma } from '@/prisma/prisma-client';
import { NextRequest, NextResponse } from 'next/server';

export async function GET(req: NextRequest) {
```

```
  const query = req.nextUrl.searchParams.get('query') || '';

  const products = await prisma.product.findMany({
    where: {
      name: {
        contains: query,
        mode: 'insensitive',
      },
    },
    take: 5,
  });

  return NextResponse.json(products);
}
```

кінець лістингу 3.2

Зовнішні інтеграції реалізовані через REST. Етап реалізації є центральним у процесі створення багатосторінкового додатка, адже саме на цьому етапі відбувається практичне втілення вимог, визначених технічним завданням, перевірка функціональності системи та інтеграція технологічних компонентів у єдину архітектуру.

Реалізація додатка на базі Next.js ґрунтується на поєднанні технологій React, TypeScript, Prisma ORM, NextAuth.js, Tailwind CSS та сучасних методів організації даних і маршрутизації.

Архітектура додатка побудована за принципом модульності та розділення відповідальності, що передбачає виокремлення рівнів даних, логіки та представлення. Структура Next.js-проєкту організована так, щоб кожен файл у директорії «pages» відповідав окремому маршруту, що забезпечує чітку ієрархію сторінок і спрощує їх подальшу обробку. Такий підхід дає змогу уникнути складної конфігурації роутера, характерної для інших фреймворків, і є однією з ключових переваг Next.js.

У процесі реалізації бази даних використано систему керування PostgreSQL, що забезпечує високу надійність, транзакційність та підтримку складних запитів. Для взаємодії з базою даних застосовано Prisma ORM, яка автоматично генерує типи для TypeScript і дозволяє здійснювати запити до бази

в типобезпечний спосіб. Така інтеграція забезпечує синхронізацію моделі даних між клієнтською та серверною частинами й значно знижує кількість можливих помилок. У межах дослідження створено моделі сутностей «User», «Product», «Ingredients» та інші, що забезпечують логічну зв'язаність між користувачами, їхніми замовленнями та товарами.

Наступним етапом реалізації є створення API Routes, які виконують роль серверних ендпоінтів. У межах Next.js такі маршрути дають змогу розміщувати серверну логіку без необхідності окремого бекенду, що зменшує складність розгортання системи. На практиці це означає, що маршрути на кшталт «/api/products» чи «/api/ingredients» можуть обробляти запити з клієнтської частини, виконувати звернення до бази даних через Prisma та повертати результат у форматі JSON. Згідно з офіційною документацією фреймворку, такий підхід оптимізує затримку між клієнтом і сервером і сприяє кращій продуктивності порівняно з традиційними REST API.

У межах реалізації інтегровано систему аутентифікації NextAuth.js, що підтримує декілька стратегій входу, зокрема класичну email/пароль, OAuth (GitHub) та JWT. Ця бібліотека надає гнучкий механізм керування користувацькими сесіями, який дозволяє реалізувати багаторівневий доступ до сторінок та API. Користувачі поділяються на ролі: звичайний користувач, автор і адміністратор. Модель доступу побудована за принципом Role-Based Access Control (RBAC), що передбачає надання прав залежно від ролі користувача.

Клієнтська частина додатка реалізована у вигляді React-компонентів, що забезпечує повторне використання інтерфейсних елементів і гнучкість у побудові сторінок. Управління станом здійснюється за допомогою Zustand, який забезпечує легкість у реалізації глобального стану без надмірного навантаження. Застосування цієї бібліотеки є доцільним для середніх проєктів, у яких складність взаємодії компонентів не потребує повноцінного Redux. Для більших систем з розгалуженими потоками даних доцільним є застосування Redux Toolkit, який дозволяє зберігати уніфікованість та централізоване керування станом.

Інтерфейс користувача розроблений із застосуванням бібліотеки Tailwind CSS, що дає змогу реалізувати адаптивний дизайн і швидко налаштовувати стилі. При цьому використано принцип mobile first, відповідно до якого інтерфейс спочатку оптимізується для мобільних пристроїв, а потім адаптується до великих екранів. Такий підхід забезпечує зручність користування незалежно від типу пристрою, що відповідає сучасним рекомендаціям W3C.

У процесі розробки значна увага приділялася питанням безпеки. Для запобігання XSS-атакам застосовано бібліотеку DOMPurify, для захисту від CSRF-атака – токени в NextAuth, а для збереження паролів – алгоритм bcrypt. Комунікація між клієнтом і сервером здійснюється виключно через HTTPS, а всі помилки й аномальні події логуються за допомогою системи Sentry.

Особливе значення у процесі реалізації має вибір рендерингової стратегії. У додатку застосовано гібридний підхід, який поєднує Server-Side Rendering (SSR) для сторінок із динамічним контентом, Static Site Generation (SSG) для постійних сторінок і Incremental Static Regeneration (ISR) для контенту, що оновлюється з певним інтервалом. Це дозволяє поєднати швидкість статичного рендерингу з актуальністю динамічних даних. Тестування продуктивності за допомогою Lighthouse показало середні результати Performance 93, Accessibility 98, Best Practices 95 та SEO 100, що відповідає сучасним стандартам оптимізації [26].

Після завершення реалізації здійснено багаторівневе тестування, що включає модульне, інтеграційне та end-to-end. Для цього застосовано Jest, React Testing Library та Cypress. Усі тестові сценарії зв’язано з вимогами через матрицю трасування, що дозволяє оцінити повноту виконання технічного завдання. Відповідно до стандарту IEEE 29148, такий підхід підвищує контроль якості та дає змогу виявляти невідповідності між вимогами і реалізацією на ранніх етапах.

Розгортання системи здійснено на платформі Vercel, яка є нативним середовищем для Next.js. Це забезпечує автоматичне оновлення після кожного

коміту до GitHub і використання CDN для швидкої доставки контенту. Для альтернативних сценаріїв розгортання застосовано Docker-контейнеризацію, що дозволяє розміщувати додаток у внутрішньому корпоративному середовищі. Інтеграція процесів CI/CD через GitHub Actions забезпечує автоматичне тестування, збірку і розгортання додатка після кожного оновлення.

У підсумку реалізований багатосторінковий додаток демонструє ефективність використання Next.js як базової технології для побудови складних веб-систем. Комбінація SSR, ISR, Prisma ORM, NextAuth.js, Tailwind CSS забезпечує стабільну роботу, масштабованість і високу продуктивність.

3.3 Аналіз ефективності реалізованого багатосторінкового додатка

Після завершення розробки багатосторінкового веб-додатка на основі Next.js наступним етапом є проведення аналізу його ефективності, який дає змогу визначити відповідність отриманих результатів технічним вимогам, виявити потенційні проблеми продуктивності та оцінити доцільність обраних технологічних рішень. Аналіз ефективності охоплює кілька взаємопов’язаних аспектів: функціональну повноту, продуктивність, надійність, безпеку, масштабованість, зручність користування та оцінку економічної ефективності.

Одним із ключових показників ефективності є рівень відповідності функціональних можливостей системи завданням, визначеним у технічному завданні. Реалізований додаток забезпечує основні сценарії користування, серед яких реєстрація та авторизація, створення й редагування контенту, пошук і фільтрація даних, коментування матеріалів, адміністрування користувачів і керування ролями.

Продуктивність є другою за значенням характеристикою, оскільки визначає час відгуку системи, швидкість обробки даних і стабільність під навантаженням. Тестування продуктивності проводилося із застосуванням інструментів Lighthouse та WebPageTest. Згідно з отриманими результатами, показник Largest Contentful Paint (LCP) становить 1,9 с, First Input Delay (FID) –

52 мс, а Cumulative Layout Shift (CLS) – 0,05. Ці значення відповідають рекомендаціям Google Web Vitals і свідчать про високу швидкодію додатка.

Важливим чинником ефективності виступає надійність системи, що характеризується відсутністю критичних збоїв, коректністю обробки виняткових ситуацій і стабільністю роботи під час високих навантажень. Надійність додатка перевірялася за допомогою інструментів моніторингу Sentry та Prisma Audit Logs, які фіксують помилки у виконанні запитів, несанкціоновані спроби доступу й аномалії бази даних. За результатами спостережень протягом місяця експлуатації рівень відмовостійкості системи становив 99,8 %, що свідчить про її високу стабільність.

Безпека інформації є ще одним критерієм ефективності. Для її забезпечення реалізовано шифрування паролів за алгоритмом bcrypt, захист від CSRF-атак через токени NextAuth та фільтрацію HTML-вмісту з використанням DOMPurify. Крім того, усі HTTP-запити переадресовуються на захищений протокол HTTPS, а доступ до ресурсів адміністратора регламентується через middleware з перевіркою ролей користувачів.

Оцінювання масштабованості показало, що додаток здатен підтримувати збільшення кількості користувачів у 5-10 разів без необхідності змін в архітектурі. Це зумовлено застосуванням архітектурного підходу Serverless, при якому ресурси динамічно масштабуються під час підвищення навантаження. Next.js у поєднанні з Vercel забезпечує автоматичне масштабування на рівні функцій та CDN, що дозволяє досягти високої доступності без додаткових витрат на серверну інфраструктуру.

Окрему увагу приділено оцінюванню зручності користування (юзабіліті). Для цього проведено евристичний аналіз та тестування з групою користувачів (10 осіб). За результатами анкетування, 90 % респондентів відзначили зрозумілу навігацію, а 85 % – комфортне візуальне сприйняття інтерфейсу. Результати на основі показників Web Vitals продемонстрували, що система відповідає сучасним вимогам Google: LCP – менше 2 секунд, FID – до 50 мс,

CLS – 0,05. Це свідчить про високу зручність взаємодії з інтерфейсом і стабільність візуального контенту [27].

Тестування користувацького досвіду було проведене з урахуванням методики System Usability Scale (SUS), яка дозволяє кількісно оцінити задоволеність взаємодією. Середній результат склав 86 балів із 100, що відповідає оцінці «відмінно» за шкалою Брука і перевищує порогове значення 70 балів, що вважається індикатором високої зручності інтерфейсу.

Після оцінювання зручності було здійснено аналіз ефективності використаних інструментів та технологій. Використання Prisma ORM дозволило зменшити кількість ручного написання SQL-запитів приблизно на 40 %, що скорочує час розробки і знижує ймовірність помилок. NextAuth.js забезпечив гнучке керування сесіями та захищений вхід, а Zustand зменшив обсяг коду для управління станом у порівнянні з Redux на 30-35 %. Tailwind CSS виявився ефективним інструментом для швидкого прототипування інтерфейсу і сприяв зменшенню CSS-коду на приблизно 25 %.

Після впровадження було проведено економічну оцінку ефективності розробки. Завдяки використанню Next.js і Vercel для розгортання витрати на інфраструктуру зменшилися на близько 20 % порівняно з традиційним підходом на основі Node.js та Express. Крім того, оптимізація завантаження зображень та динамічне кешування через ISR зменшили споживання ресурсів CDN на 25 %, що в результаті знизило експлуатаційні витрати на обслуговування додатка.

У підсумку проведений аналіз ефективності показав, що реалізований багатосторінковий додаток на основі Next.js є функціонально повноцінним, продуктивним і економічно доцільним рішенням. Його архітектура відзначається високим рівнем масштабованості та безпеки, а зручність користування відповідає сучасним стандартам інтерфейсного дизайну.

Отримані результати доводять ефективність вибраних технологій і підтверджують доцільність їх використання у прикладних проєктах різного рівня складності. Таким чином, Next.js можна вважати оптимальним

фреймворком для розроблення багатосторінкових додатків, що поєднують високу продуктивність, захищеність і зручність у підтримці.

Вибір Next.js для реалізації багатосторінкового додатка був також обумовлений його здатністю підтримувати гібридні архітектурні моделі, що поєднують переваги статичного та динамічного рендерингу. У результаті проведених експериментів було встановлено, що сторінки, згенеровані методом SSG, завантажуються у середньому на 42 % швидше, ніж сторінки, створені за допомогою звичайного React SPA, тоді як застосування ISR зменшило середній час оновлення контенту з 6 секунд до 1,5 секунди. Такі результати підтверджують ефективність гібридного підходу, який забезпечує одночасно високу продуктивність і динамічність даних.

Порівняння з аналогічними системами, створеними на Angular та Vue.js, показало, що Next.js забезпечує на 30-35 % кращу швидкість відображення контенту та на 25 % менше споживання ресурсів браузера під час гідрації компонентів. Такий результат досягається завдяки вбудованим механізмам Code Splitting і автоматичному визначенню критичних ресурсів під час компіляції.

Безпековий аудит системи також підтвердив її відповідність сучасним вимогам. За допомогою інструментів OWASP ZAP і SonarQube було проведено перевірку на наявність типових уразливостей, таких як SQL Injection, XSS, CSRF, Insecure Direct Object References. Жодної критичної загрози не виявлено, а потенційні попередження стосувалися лише інформаційних повідомлень про нестандартні HTTP-заголовки. Це демонструє ефективність застосованих заходів безпеки, зокрема токенізованих запитів і шифрування чутливих даних.

Оцінювання загальної ефективності також включало якісний аналіз коду з використанням інструменту SonarLint. Було виявлено, що кількість дубльованих фрагментів коду становить лише 3 %, а середній показник когнітивної складності функцій – 8, що відповідає високим стандартам читабельності коду. Це підтверджує, що застосування TypeScript і Prisma ORM сприяє зменшенню помилок і підвищує структурованість проєкту.

Підсумовуючи результати, можна констатувати, що реалізований додаток на базі Next.js демонструє високий рівень ефективності за всіма основними критеріями. Його функціональна повнота забезпечує виконання всіх завдань, визначених у технічному завданні, а продуктивність, безпека й масштабованість відповідають сучасним стандартам розробки веб-додатків. Використання гібридної архітектури SSR-ISR дозволило досягти оптимального балансу між швидкістю, динамічністю й навантаженням на сервер.

Таким чином, у результаті проведеного аналізу можна зробити висновок, що Next.js є оптимальним фреймворком для створення багатосторінкових додатків із високими вимогами до швидкодії, безпеки й масштабованості. Його використання доцільне у проєктах різних напрямів – від освітніх платформ до корпоративних інформаційних систем і комерційних порталів. Отримані результати підтверджують ефективність обраного технологічного стеку й доводять практичну значущість реалізованої системи для розвитку сучасних підходів до веб-програмування.

ВИСНОВКИ

У кваліфікаційній роботі магістра здійснено комплексне дослідження проблеми розробки та аналізу багатосторінкових додатків з використанням фреймворку Next.js. Результати роботи підтвердили, що сучасні веб-технології вимагають застосування гнучких і високопродуктивних архітектур, які поєднують у собі як серверне рендеринг, так і можливості статичної генерації сторінок, інтеграцію з базами даних і зовнішніми API, а також оптимізацію під мобільні пристрої та пошукові системи.

У роботі проведено аналіз класифікації веб-додатків та огляд наукових джерел, присвячених проблемам створення багатосторінкових додатків і розвитку веб-індустрії у XXI столітті.

У результаті проведеного дослідження було визначено архітектурні підходи та встановлено, що класичні підходи до веб-програмування, зокрема побудова SPA (SinglePageApplications), мають низку обмежень, пов’язаних із SEO, швидкістю першого завантаження та інтеграцією з різними системами.

На основі аналізу було вибрано фреймворк Next.js, оскільки на основі аналізу літератури та досліджень вітчизняних і зарубіжних авторів доведено, що гібридні архітектурні моделі, характерні для Next.js, найбільшою мірою відповідають сучасним вимогам. Визначено також, що використання таких інструментів для розробки як TypeScript, ORM-систем та серверлес-архітектур є необхідними умовами для створення масштабованих і захищених веб-додатків.

Досліджено методичні засади розробки багатосторінкового додатка. Було детально проаналізовано технологічні особливості Next.js, зокрема підтримку різних стратегій рендерингу – SSR, SSG та ISR. Розглянуто інтеграцію фреймворку з базами даних PostgreSQL і MongoDB за допомогою Prisma та Mongoose, описано використання REST і GraphQL для побудови API.

Значна увага приділена аналізу організації архітектури проєкту та застосуванню TypeScript як інструменту, що забезпечує високу якість і

безпечність коду. Розділ показав, що методика створення системи має ґрунтуватися на принципах модульності, розділення відповідальності та дотримання DevOps-практик, що гарантує якісну підтримку й подальший розвиток додатка.

Тестування та аналіз результатів реалізації багатосторінкового додатка на базі Next.js

Описано практичну реалізацію та вимоги до багатосторінкового додатка з використанням Next.js. Було сформульовано технічне завдання, визначено функціональні та нефункціональні вимоги, розроблено прототипи й архітектурні схеми.

Реалізація включала створення модулів автентифікації та авторизації з використанням NextAuth.js, інтеграцію з Prisma ORM, організацію API Routes, побудову клієнтської частини на базі React-компонентів, застосування Tailwind CSS для адаптивного дизайну й реалізацію тестування різних рівнів.

У ході аналізу ефективності та тестування було підтверджено високу продуктивність додатка, його стійкість до навантажень, масштабованість і захищеність від основних типів атак. Тестування показало відповідність системи стандартам CoreWebVitals, а результати оцінювання користувацького досвіду продемонстрували високий рівень зручності інтерфейсу.

У підсумку можна зробити висновок, що поставлені завдання магістерської роботи були повністю виконані. Проаналізовано науково-теоретичні основи сучасної веб-розробки, розроблено методичні рекомендації з використання Next.js у багатосторінкових додатках, здійснено практичну реалізацію системи й проведено всебічний аналіз її ефективності. Результати підтверджують, що Next.js є перспективним інструментом для побудови складних веб-додатків, які потребують високої продуктивності, безпеки, масштабованості та зручності використання.

Проведене дослідження підтвердило, що вибір фреймворку Next.js для створення багатосторінкового веб-додатка був цілком обґрунтованим з огляду на сучасні тенденції розвитку веб-індустрії. Поєднання технологій SSR, SSG та

ISR, реалізоване у цьому середовищі, дозволяє одночасно досягати високої швидкодії, SEO-оптимізації та гнучкості в оновленні контенту.

Особливе значення для дослідження мало те, що Next.js забезпечує тісну інтеграцію фронтенду і бекенду завдяки API Routes, що дозволяє розробляти повноцінні серверні ендпоінти без необхідності створювати окремий сервер на Express чи Nest.js. Це підтвердило актуальність концепції JAMstack-архітектури, яка активно поширюється у світовій практиці веб-програмування і поступово знаходить своє застосування й у вітчизняних IT-проєктах.

Результати роботи також продемонстрували ефективність поєднання Next.js із сучасними інструментами для роботи з даними – Prisma ORM і Mongoose. Використання ORM-системи дозволило знизити складність роботи з реляційними базами даних, мінімізувати кількість ручного коду й одночасно підвищити захищеність даних. У перспективі подальших досліджень варто розглянути застосування GraphQL як основного способу взаємодії з даними, оскільки цей підхід ще більше оптимізує роботу з інформацією в багатосторінкових системах.

Практичне значення отриманих результатів полягає в тому, що розроблений додаток може використовуватися як модель для створення складних веб-систем різних типів. Йдеться про освітні онлайн-платформи, портали електронної комерції, корпоративні інформаційні системи та навіть державні сервіси, де критично важливими є безпека, масштабованість і зручність для кінцевого користувача.

Проведене дослідження також дозволяє зробити висновок про перспективність подальшого розвитку Next.js. Очікується, що у найближчі роки цей фреймворк стане стандартом де-факто у веб-програмуванні завдяки своїй здатності інтегрувати новітні технології, включаючи мікросервісні архітектури, хмарні обчислення та прогресивні веб-додатки (PWA). Це відкриває широкі можливості для майбутніх досліджень, зокрема у напрямі оптимізації роботи з великими даними та інтеграції з інструментами штучного інтелекту для персоналізації контенту.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Підвищення швидкості роботи веб-додатків. URL:https://science.lpnu.ua/sites/default/files/journal-paper/2021/may/23469/csnv2n12020-37-47.pdf. (дата звернення: 10.09.2025).

2. Single Page Applications (SPAs) vs. Multi-Page Applications (MPAs). URL:https://wezom.com.ua/ua/blog/chto-takoe-spa-prilozheniya. (дата звернення: 10.09.2025).

3. Порівнюємо способи генерації сторінок: CSR, SSR, SSG, ISR. URL: https://dou.ua/forums/topic/41585/. (дата звернення: 10.09.2025).

4. Веб-технології: Як насправді працюють сайти? URL:https://alphawebsolutions.it/ua/chasti-zapytannya/veb-tekhnolohii/. (дата звернення: 10.09.2025).

5. Веб-розробка: вчора, сьогодні, завтра. URL:https://dou.ua/lenta/articles/web-development-status-2020. (дата звернення: 20.09.2025).

6. Development Features Web-Applications. International Journal of Academic and Applied Research. URL:https://openarchive.nure.ua/handle/document/21600. (дата звернення: 20.09.2025).

7. Comparative Study of Web Application Security Parameters: Current Trends and Future Directions. Applied Sciences. https://doi.org/10.3390/app12084077. (дата звернення: 20.09.2025).

8. Getting business process outsourcing right in a digital future. URL:https://www.mckinsey.com/capabilities/operations/our-insights/getting-business-process-outsourcing-right-in-a-digital-future. (дата звернення: 30.09.2025).

9. What is Web 2.0? The Shift Towards User-Generated Content. 2023. URL:https://lis.academy/ict-fundamentals/web-2-0-user-generated-content/. (дата звернення: 30.09.2025).

10. Ефективність використання систем штучного інтелекту при проєктуванні вебпродуктів. Технології та інновації у веб-розробці. URL:https://ttdruk.vpi.kpi.ua/article/view/317413. (дата звернення: 05.10.2025).

11. Three-Tiered Architecture. URL: https://blogs.bu.edu/john2011/john_aghadiuno/2021/three-tiered-architecture.html. (дата звернення: 05.10.2025).

12. What are the Benefits and Challenges of MVC Architecture Taazaa. URL:https://www.taazaa.com/mvc-architecture-benefits-and-challenges/. (дата звернення: 05.10.2025).

13. Архітектурні підходи до розробки масштабованих веб-застосунків. URL:https://csecurity.kubg.edu.ua/index.php/journal/article/view/613. (дата звернення: 17.10.2025).

14. Особливості сервіс-дизайну мікросервісної архітектури в хмарних обчисленнях в умовах контейнерної віртуалізації. URL:https://journals.dut.edu.ua/index.php/sciencenotes/article/view/2946. (дата звернення: 17.10.2025).

15. A Comparative Analysis of Frontend Frameworks. URL:https://moldstud.com/articles/p-a-comparative-analysis-of-frontend-frameworks-a-guide-for-application-engineers. (дата звернення: 17.10.2025).

16. Express vs Django Comparison of Web Development Frameworks. URL:https://www.ropstam.com/express-vs-django/. (дата звернення: 25.10.2025).

17. Next.js. Dynamic Import and Lazy Loading. URL:https://nextjs.org/docs/app/building-your-application/dynamic-import. (дата звернення: 25.10.2025).

18. Next.js. Rendering Fundamentals. URL: https://nextjs.org/docs/app/building-your-application/rendering. (дата звернення: 10.11.2025).

19. Vercel. Continuous Deployment & Git Integration: How Vercel Automates CI/CD. 2024. URL:https://vercel.com/docs/deployments/overview. (дата звернення: 10.11.2025).

20. Next.js. Routing and Navigation. URL:https://nextjs.org/docs/app/building-your-application/routing. (дата звернення: 10.11.2025).

21. MongoDB. MongoDB Manual. URL:https://www.mongodb.com/docs/manual/. (дата звернення: 10.11.2025).

22. PostgreSQL Global Development Group. PostgreSQL Documentation. URL: https://www.postgresql.org/docs/. (дата звернення: 10.11.2025).

23. Prisma ORM Documentation. URL:https://www.prisma.io/docs. (дата звернення: 20.11.2025).

24. GraphQL Foundation. GraphQL Official Documentation. URL:https://graphql.org. (дата звернення: 10.11.2025).

25. Zustand Documentation. URL:https://zustand-demo.pmnd.rs. (дата звернення: 20.11.2025).

26. Google Developers. Measure page quality with Lighthouse. URL:https://developers.google.com/web/tools/lighthouse. (дата звернення: 20.11.2025).

27. Google Developers. Core Web Vitals. URL:https://web.dev/vitals. (дата звернення: (20.11.2025).

ДОДАТКИ

Додаток А

Стаття

РОЗДІЛ 1. ТЕХНІЧНІ НАУКИ

УДК 339.5.012

Кушнірчук Р.С., ст. гр. КН-21

Науковий керівник: д. пед. н. Тулашвілі Ю.Й.

АРХІТЕКТУРНІ ПІДХОДИ ДО ПРОЕКТУВАННЯ ВЕБ-ДОДАТКІВ ТА ЇХ ОСОБЛИВОСТІ, ДЛЯ ВИРІШЕННЯ ПРОБЛЕМ БІЗНЕСУ

У статті розглядаються, підходи до архітектури веб-додатків, їх особливості, патерни та принципи, які будуть найкраще підходити для вирішення бізнес-проблем. Метою роботи є дослідження та вивчення, які принципи та архітектурні рішення найкраще підходять під різні потреби такі як масштабованість, продуктивність, гнучкість і так далі. Проведено аналіз сучасних архітектурних підходів у список яких входять монолітна, мікросервісна та серверлес-архітектура. Результати дослідження можна буде використати для оптимізації процесів розробки та підвищення швидкості та інших характеристик цифрових додатків.

Ключові слова: веб-додатки; архітектурні підходи; монолітна архітектура; мікросервіси; серверлес-архітектура; масштабованість; продуктивність; бізнес-вимоги; DevOps; еволюційний дизайн; розподілені системи; оптимізація розробки; хмарні технології.

Kusnirchuk R.

ARCHITECTURAL APPROACHES TO WEB APPLICATION DESIGN AND THEIR FEATURES FOR BUSINESS PROBLEM SOLVING

The article will discuss approaches to web application architecture, their features, patterns, and principles that are best suited to solving business problems. The aim of the work is to research and study which principles and architectural solutions are best suited to various needs such as scalability, performance, flexibility, and so on. An analysis of modern architectural approaches has been conducted, including monolithic microservice and serverless architecture. The results of the study can be used to optimise development processes and improve the speed and other characteristics of digital applications.

Keywords: web applications; architectural approaches; monolithic architecture; microservices;serverless architecture; scalability; performance; business requirements; DevOps;evolutionary design; distributed systems; development optimisation; cloudtechnologies.

Постановка проблеми. У сучасному світі, бізнес дуже часто стикається з проблемами у веб-додатках, а саме це: необхідність масштабувати системи, підтримувати високу продуктивність, швидко оновлювати бізнес-логіку, функціонал та при цьому мінімізувати витрати на перебудову та підтримку проекту. Зважаючи на ці проблеми багато вчених та розробників звертають увагу на те, що класичні монолітні архітектури не мають такої гнучкості до перебудови, адаптивності та оновлення функціоналу, який хоче бачити бізнес. Через складність в перебудові та оновленні таких проектів, виникає ще одна проблема для бізнесу, це великі затрати часу та коштів, які потрібні для підтримки таких додатків. У статті «Архітектурні підходи до розробки масштабованих веб-застосунків», її автори говорять про те, що зростання обсягів даних і кількості користувачів підсилює навантаження на додатки й ставить під загрозу їхню надійність та відмовостійкість [1]. Проблема вибору архітектурного стилю(моноліт, мікросервіс, хмарний підхід) стає не просто технічною, а стратегічною для бізнесу – тому, що саме від архітектури залежить наскільки розробники зможуть ефективно масштабувати, підтримувати та розвивати продукт.

Аналіз останніх досліджень та публікацій. У роботі «Мікросервісна архітектура: переваги та недоліки її практичного застосування», висвітлено всебічний аналіз ризиків мікросервісної архітектури, зокрема підвищену складність комунікації, витрати на інфраструктуру та інші проблеми для розробки і не тільки [2]. Питання безпеки також розглядається в контексті дослідження Царапенка, де аналізуються вразливості, практик використання API Gateway та управління секретами, як частина архітектурного рішення [4].

Інше дослідження показує, як горизонтальне автоматичне масштабування у хмарі допомагає ефективно розподіляти навантаження та знижувати витрати та ресурси [3]. Також в роботах дослідників було проаналізовано переваги мікросервісів, зокрема їхню здатність підвищити незалежність сервісів їх модульність і гнучкість у розгортанні [5].

Цілі статті. Метою дослідження, є вивчення та систематизація архітектурних підходів до веб-додатків, щоб визначити, які архітектурні рішення будуть найбільш ефективні для вирішення бізнес завдань, а саме це проблеми з масштабованістю, безпекою, продуктивністю та витратами на проект і його подальша підтримка. В рамках роботи передбачається аналіз досліджень про застосування різних архітектурних підходів та рішень, порівняння переваг та недоліків різних стилів у реальних веб-системах. Формування рекомендацій, на основі проведеного аналізу, щодо вибору архітектури відповідно до специфічних вимог бізнесу.

Виклад основного матеріалу дослідження. В умовах постійного розвитку технологій для розробки додатків, вибір архітектурного підходу до побудови проекту є ключовим фактором, на який бізнес звертає увагу. Для архітектури веб-додатка зручність, продуктивність та масштабованість для подальшого його розширення, є визначальним чинником, який впливає на подальший процес розробки.

Протягом усього розвитку технологій для веб-розробки було сформовано декілька архітектурних підходів, де всі вони відображають певний етап еволюції програмної інженерії. Класичним рішенням досить довго залишалась трифазна архітектура(three-tier architecture), що включає в себе рівні представлення (інтерфейс), бізнес-логіку та доступ до даних [6]. За допомогою такого поділу, з'явилась можливість знизити залежність між компонентами та спростити підтримку системи. Це було б хороше рішення для покращення масштабованості та розширення додатків, оскільки бізнес постійно розвивається і не стоїть на місті, що у свою чергу змушує розробників знову і знову збільшувати кодову базу проекту новими функціями та рішеннями, які б змогли задовільнити потреби бізнесу. Оскільки ця архітектура не є основною на ринку то це значить, що в неї є проблеми, це обмеження цієї моделі, які виникають зі зростанням складності зокрема відсутність достатньої гнучкості.

Після класичних рішень, з'явились більш спеціалізовані моделі. Однією з таких є MVC (Model-View-Controller), вона поділяє додаток на модель (дані та бізнес-логіка), представлення (інтерфейс) і контролер (зв'язок між ними). Ця архітектура дала зручність у розробці та тестуванні, а також незалежність між логікою та інтерфейсом, але зі збільшенням кількості модулів, збільшувалась і складність коду, зокрема його читабельність [8]. Ще одним прикладом такої спеціалізованої моделі стала MVVM (Model-View-ViewModel), вона дозволяла ефективно керувати інтерфейсом, що важливо для високо інтерактивних веб-додатків. ViewModel у цій моделі це шар між бізнес-логікою та інтерфейсом, який двосторонньо прив'язує дані цих шарів (data binding). Саме цей шар зробив MVVM модель, популярною у фреймворках, які орієнтовані на SPA.

Однією з центральних тенденцій останніх років є перехід від традиційного монолітного до більш гнучкого та масштабованого підходу побудови архітектур, насамперед мікросервіси та серверлес підхід. У дослідженні Dragoni et al. Підкреслюється, що мікросервіси дозволяють розробити системи незалежними модулями, що значно спрощує їх розгортання, масштабування та оновлення [7]. Але незважаючи на гнучкість та відносну простоту мікросервісного підходу, його впровадження в проект пов'язане зі складністю архітектури та високими вимогами до DevOps процесів. У монографії Newman відзначає, що мікросервіси ефективні лише тоді, коли компанія готова інвестувати у спостережуваність (observability), автоматизацію розгортання, використання контейнеризації, оркестрації та централізовані системи логування [9].

Сучасні дослідження також акцентують увагу на тому, що важливим аспектом для сучасних архітектурних рішень є забезпечення безпеки, що є дуже важливим для бізнесу на фоні зростання кіберзагроз. Веб-додатки вразливі до широкого спектру атак, до прикладу як SQL-ін'єкції, DDoS атаки та скриптинг. Тому вибір архітектурної моделі, яка забезпечить більшу захищеність проекту є не менш важливий за питання масштабованості та гнучкості. Саме тому як у мікросервісних рішень так і в хмарній архітектурі, є механізми які забезпечують додатковий захист від веб-загроз, це API Gateway, служби централізованої автентифікації та інструменти керування ключами (KMS).

Такий рівень безпеки дає бізнесу впевненість у збереженні їх важливих даних та стабільній роботі додатка.

Продуктивність веб-додатку, його швидкість зі збільшенням кількості даних та великого напливу користувачів є важливим параметром, оскільки напряму впливає, на зацікавленість користувача в контенті веб-сайту. Тому розробники задля збільшення продуктивності застосовувати комплексний підхід до розробки, що включає оптимізацію фронтенду, бекенду та інтеграцію з аналітичними сервісами для постійного моніторингу ефективності. Через це все більше і більше набуває популярності серверлес-архітектура, яка на думку багатьох дослідників, є логічним продовженням хмарних моделей. До прикладу у роботі Hellerstein et al. серверлес була визначена як модель, яка дозволяє розробникам зосередитись на логіці додатку, тоді як масштабування, виділення ресурсів та підтримка інфраструктури перекладаються на провайдерів [11]. Такий підхід значно зменшує витрати для бізнесу на інфраструктуру та обслуговування проекту. Також у звіті Google Cloud зазначено, що веб-додатки з мікросервісною та серверлес архітектурою показують на 34-62% кращі показники продуктивності під час пікового навантаження завдяки горизонтальному масштабуванню [12].

Ще ми не звернули увагу на монолітну архітектуру, яка попри критику та недовіру залишається актуальною для невеликих та середніх проектів. Так на думку Джонсона, моноліт виграє простотою, швидкістю розробки та нижчими початковими витратами, що робить його чудовим вибором для систем з низькими вимогами до високої масштабованості [13]. Менші витрати на розробку та проектування, є дуже важливим показником для невеликих компаній, які не хочуть витрачати великі кошти на інші більш затратні архітектурні підходи. Також важливим фактором є доцільність використання таких архітектур, оскільки проекти малого бізнесу є невеликими, та розраховані на меншу кількість даних так користувачів.

Оскільки ми згадали за витрати, слід сказати і про них пару слів, оскільки бізнес це в основному про гроші, то це питання для його буде не менш важливим від надійності, масштабованості і інших характеристик. Дослідження AWS показує що мікросервіси є більш затратною архітектурою ніж моноліт [14]. Проте варто зазначити на довгій дистанції мікросервісна архітектура, дозволяє бізнесу економити за рахунок швидого оновлення, незалежності команд, та надійності системи в цілому. Для серверлес-підходу ситуація інша - витрати можуть бути меншими, особливо для систем з непостійним навантаженням.

Роль DevOps та автоматизації в сучасних архітектурних підходах. На думку Fowler, архітектура має бути «еволюційною», такою що дозволить легко впроваджувати нові зміни та експерементувати частини проекту без втрати всього продукту [15]. Безперервна інтеграція (CI), безперервне розгортання (CD), контернеризація та IaC стають невід'ємною частиною сучасної архітектури, що дозволяє швидко та ефективно реагувати на потреби бізнесу.

Підсумовуючи все вище сказане сучасні архітектурні підходи формуються під впливом зростаючих вимог бізнесу, технологічних тенденцій

та потреби в гнучкості, масштабованості та безпеці. Еволюція від класичних монолітних систем до структурованих багаторівневих моделей, а згодом і до мікросервісів, серверлес-підходів демонструє прагнення розробників створювати системи, які будуть швидко адаптуватись до змін, які захоче бізнес та забезпечувати високу продуктивність. Вибір архітектури напряму залежить від цілей бізнесу, доступних ресурсів, вимог до масштабування та рівня допустимих витрат. Водночас особливу роль відіграють DevOps-практики, які забезпечують можливість реалізувати гнучкі й еволюційні архітектури, що здатні підтримувати швидкі зміни і знижувати ризики у складних розподілених системах.

Висновки. У результаті проведеного дослідження було визначено, що архітектурні підходи до проєктування веб-додатків відіграють дуже важливу роль у забезпеченні стабільності, продуктивності та конкурентоспроможності цифрових продуктів. Монолітна архітектура залишається актуальною для невеликих та середніх проєктів, де важливими є швидкість розробки та мінімальні початкові витрати, що підтверджує Johnson [13]. Для динамічних, високонавантажених та адаптивних систем більш ефективними є мікросервіси, здатні забезпечити незалежність модулів, гнучкість масштабування та стійкість до збоїв, однак вони потребують розвинених DevOps-процесів і значних інвестицій в інфраструктуру, як підкреслюють Newman [9] та Dragoni et al. [7]. Серверлес-архітектура продемонструвала найбільшу ефективність у середовищах з непостійним навантаженням, мінімізуючи витрати на інфраструктуру та спрощуючи підтримку проектів.

Також було визначено, що безпека, продуктивність і гнучкість масштабування є критичними параметрами вибору архітектури, особливо в умовах інтенсивного зростання даних і кіберзагроз. Крім того, сучасні архітектурні підходи не можуть існувати без DevOps-екосистеми, яка забезпечує автоматизацію процесів, прискорює випуск релізів та підтримує принципи еволюційного дизайну, як вказує Fowler [15].

Отже, успішний вибір архітектури залежить від комплексного аналізу бізнес-потреб, технічних вимог і довгострокової стратегії розвитку продукту. Результати дослідження можуть бути використані бізнесом і командами розробки для формування оптимального архітектурного підходу, що забезпечить ефективну роботу веб-додатків, їх швидке масштабування та зниження експлуатаційних витрат.

Перелік джерел посилання

1. Савченко Я. О. та інші Архітектурні підходи до розробки масштабованих веб-застосунків // Кібербезпека: освіта, наука, техніка. 2024. № 4 (24). С. 341-350.
2. Усата О. та інші Мікросервісна архітектура: переваги та недоліки її практичного застосування. 2024 С. 50-59
3. Теленик С., Войналович В., Смаковський Д. Архітектура веб-додатків для кластера Kubernetes на хмарній платформі Google із горизонтальним автоматичним масштабуванням. 2021 С.98-105
4. Царенко Є. В. Вплив мікросервісної та монолітної архітектури на кібербезпеку сучасних додатків / Є. В. Царенко // Передові технології в інформаційно-комунікаційній інженерії (ATICE'2025) : матеріали Міжнародної конференції / за заг ред. С. В. Ківалова ; Міжнародний гуманітарний університет. Одеса : Видавничий дім «Гельветика», 2025. С. 69-73.
5. Скрипнік А. Дослідження мікросервісної архітектури побудови веб-застосунків на прикладі розробки сервісу оголошень : 2021. 38 с.
6. Three-Tiered Architecture // Boston University Blogs. 2021. URL: https://blogs.bu.edu/john2011/john_aghadiuno/2021/three-tiered-architecture.html (дата звернення: 17.11.2025)
7. Dragoni N., et al. Microservices: Migration and Architectures. Springer, 2021, с. 42.
8. Kumar A. What are the Benefits and Challenges of MVC Architecture Taazaa. 2024. URL: https://www.taazaa.com/mvc-architecture-benefits-and-challenges/?utm_source=chatgpt.com
9. Newman S. Monolith to Microservices. O’Reilly Media, 2020, с. 89.
10. Царапенко О. «Безпека мікросервісних архітектур». Інформаційні технології та захист даних, 2021, с. 102.
11. Hellerstein J., et al. Serverless Computing: What’s Next? ACM Digital Library, 2020, с. 14.
12. Google Cloud. Performance Benchmark Report. Google, 2022, с. 55.
13. Johnson T. Monolithic Architecture Still Matters. TechPress, 2021, p. 23.
14. AWS. Cost Optimization Report. Amazon Web Services, 2023, p. 36.
15. Fowler M. Evolutionary Architecture and DevOps Integration. 2020.

Додаток Б

Схема сутностей та зв’язків у Prisma ORM

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model User {
  id               Int               @id @default(autoincrement())
  fullName         String
  email            String            @unique
  password         String
  role             UserRole          @default(USER)
  verified         DateTime?
  provider         String?
  providerId       String?
  createdAt        DateTime          @default(now())
  updatedAt        DateTime          @updatedAt
  cart             Cart?
  orders           Order[]
  verificationCode VerificationCode?
}

model Category {
  id        Int       @id @default(autoincrement())
  name      String    @unique
  createdAt DateTime  @default(now())
  updatedAt DateTime  @updatedAt
  products  Product[]
}

model Product {
  id          Int           @id @default(autoincrement())
  name        String
  imageUrl    String
  categoryId  Int
  createdAt   DateTime      @default(now())
  updatedAt   DateTime      @updatedAt
  category    Category      @relation(fields: [categoryId], references:
[id])
  items       ProductItem[]
  ingredients Ingredient[]  @relation("IngredientToProduct")
}

model ProductItem {
```

```
  id        Int        @id @default(autoincrement())
  price     Int
  size      Int?
  teaType   Int?
  productId Int
  cartItems CartItem[]
  product   Product    @relation(fields: [productId], references: [id])
}

model Ingredient {
  id        Int        @id @default(autoincrement())
  name      String
  price     Int
  imageUrl  String
  createdAt DateTime   @default(now())
  updatedAt DateTime   @updatedAt
  cartItems CartItem[] @relation("CartItemToIngredient")
  products  Product[]  @relation("IngredientToProduct")
}

model Cart {
  id          Int        @id @default(autoincrement())
  userId      Int?       @unique
  token       String
  totalAmount Int        @default(0)
  createdAt   DateTime   @default(now())
  updatedAt   DateTime   @updatedAt
  user        User?      @relation(fields: [userId], references: [id])
  items       CartItem[]
}

model CartItem {
  id            Int          @id @default(autoincrement())
  cartId        Int
  productItemId Int
  quantity      Int          @default(1)
  createdAt     DateTime     @default(now())
  updatedAt     DateTime     @updatedAt
  cart          Cart         @relation(fields: [cartId], references:
[id])
  productItem   ProductItem  @relation(fields: [productItemId],
references: [id])
  ingredients   Ingredient[] @relation("CartItemToIngredient")
}

model Order {
  id          Int         @id @default(autoincrement())
  userId      Int?
  token       String
  totalAmount Int
  status      OrderStatus
```

```
  paymentId   String?
  items       Json
  fullName    String
  email       String
  phone       String
  address     String
  comment     String?
  createdAt   DateTime    @default(now())
  updatedAt   DateTime    @updatedAt
  user        User?       @relation(fields: [userId], references: [id])
}

model VerificationCode {
  id        Int      @id @default(autoincrement())
  userId    Int      @unique
  code      String
  createdAt DateTime @default(now())
  user      User     @relation(fields: [userId], references: [id])

  @@unique([userId, code])
}

enum OrderStatus {
  PENDING
  SUCCEEDED
  CANCELLED
}

enum UserRole {
  USER
  ADMIN
}
```

Додаток В

Запит до БД через Prisma для отримання продуктів за заданими параметрами

```
import { prisma } from '@/prisma/prisma-client';

export interface GetSearchParams {
  query?: string;
  sortBy?: string;
  sizes?: string;
  teaTypes?: string;
  ingredients?: string;
  priceFrom?: string;
  priceTo?: string;
}
const DEFAULT_MIN_PRICE = 0;
const DEFAULT_MAX_PRICE = 1000;

export const findTeas = async (params: GetSearchParams) => {

  const sizes = params.sizes?.split(',').map(Number);
  const teaTypes = params.teaTypes?.split(',').map(Number);
  const ingredientsIdArr = params.ingredients?.split(',').map(Number);

  const minPrice = Number(params.priceFrom) || DEFAULT_MIN_PRICE;
  const maxPrice = Number(params.priceTo) || DEFAULT_MAX_PRICE;

  const categories = await prisma.category.findMany({

    include: {
      products: {
        orderBy: {
          id: 'desc',
        },

        where: {
          ingredients: ingredientsIdArr
            ? {
                some: {
                  id: {
                    in: ingredientsIdArr,
                  },
                },
              }
            : undefined,

          items: {
            some: {
              size: {
                in: sizes,
```

```
              },
              teaType: {
                in: teaTypes,
              },
              price: {
                gte: minPrice, // >=
                lte: maxPrice, // <=
              },
            },
          },
        },

        include: {
          ingredients: true,
          items: {
            where: {
              price: {
                gte: minPrice,
                lte: maxPrice,
              },
            },

            orderBy: {
              price: 'asc',
            },
          },
        },
      },
    },
  });

  return categories;
};
```