Міністерство освіти і науки України
Національний технічний університет
«Дніпровська політехніка»

Факультет інформаційних технологій
(факультет)

Кафедра Інформаційних технологій та комп'ютерної інженерії
(повна назва)

ПОЯСНЮВАЛЬНА ЗАПИСКА

кваліфікаційна робота ступеня бакалавра
(бакалавра, спеціаліста, магістра)

Здобувача Нікіфорової Юлії Олегівни
(ПІБ)

академічної групи 126-21з-1
(шифр)

спеціальності 126 «Інформаційні системи та технології»
(код і назва спеціальності)

за освітньо-професійною програмою «Інформаційні системи та технології»
(офіційна назва)

на тему Розробка вебсайту інтернет-магазину “Next-Shopping” з використанням передового фреймворка Next.js
(назва за наказом ректора)

Керівник	**Прізвище, ініціали**	**Оцінка за шкалою**		**Підпис**
		рейтинговою	**інституційною**	
кваліфікаційної роботи	проф. Коротенко Г.М.			
розділів:				
Рецензент	доц. Ширін А.Л			
Нормоконтролер	проф. Коротенко Г.М.			

Дніпро
2025

ЗАТВЕРДЖЕНО:

завідувач кафедри

інформаційних технологій
та комп'ютерної інженерії
(повна назва)

_____ Гнатушенко В.В.
(підпис) (прізвище)

«___» _____ 2025 року

ЗАВДАННЯ
на кваліфікаційну роботу
ступеня бакалавра
(бакалавра, спеціаліста, магістра)

здобувачу Нікіфорова Ю.О. **академічної групи** 126-21з-1
(прізвище та ініціали) (шифр)

спеціальності 126 «Інформаційні системи та технології»

за освітньо-професійною програмою _____

на тему Розробка вебсайту інтернет-магазину "Next-Shopping" з використанням передового фреймворка Next.js

затверджену наказом ректора НТУ «Дніпровська політехніка» від 05.05.2025 № 337-с

Розділ	Зміст	Термін виконання
Розділ 1	Дослідження предметної області	17.04.2025-30.04.2025
Розділ 2	Огляд відомих проєктних рішень	17.04.2025-18.05.2025
Розділ 3	Реалізація проєкту	17.04.2025-20.06.2025

Завдання видано _____ Коротенко Г.М.
(підпис керівника) (прізвище, ініціали)

Дата видачі 03.02.2025

Дата подання до екзаменаційної комісії _____

Прийнято до виконання _____ Нікіфорова Ю.О.
(підпис студента) (прізвище, ініціали)

РЕФЕРАТ

Пояснювальна записка: 69 сторінок, 17 рисунків, 1 додаток, 17 джерел.

Тема дипломної роботи: “Розробка вебсайту інтернет-магазину “Next-Shopping” з використанням передового фреймворка Next.js”.

Предмет дослідження: архітектурні патерни, сучасні фреймворки та технології, що застосовуються для створення веб-додатків електронної комерції. Зокрема, розглядаються засоби розробки (Next.js, TypeScript, MongoDB), інтеграція з зовнішніми сервісами (Cloudinary, NextAuth.js), забезпечення масштабованості, безпеки та адаптивності інтерфейсу.

Об'єкт дослідження: процес розробки масштабованого веб-додатку для інтернет-магазину, що відповідає актуальним вимогам сучасної електронної комерції.

Об’єкт розроблення: демо-версія (MVP) веб-застосунку інтернет-магазину з реалізацією базової функціональності.

Мета кваліфікаційної роботи: проєктування та розробка веб-додатку інтернет-магазину з використанням сучасного стеку технологій (Next.js, TypeScript, MongoDB), який демонструє практичне застосування сучасних методів і засобів розробки, а також забезпечує високу продуктивність, безпеку, масштабованість та зручність користування.

Ключові слова: Next.js, TypeScript, Cloudinary, MongoDB, БАЗА ДАНИХ, E-COMMERCE, ВЕБ-ДОДАТОК, JWT, RESTful API.

ABSTRACT

Explanatory note: 69 pages, 17 figures, 1 appendix, 17 sources.

Topic of the diploma work: “Development of the E-commerce Website 'Next-Shopping' Using the Advanced Framework Next.js”.

Subject of research: Architectural patterns, modern frameworks, and technologies used for the development of e-commerce web applications. In particular, the research focuses on development tools such as Next.js, TypeScript, and MongoDB, integration with external services (Cloudinary, NextAuth.js), and ensuring scalability, performance, security, and responsive design.

Object of research: The process of developing a scalable web application for an online store that meets the current demands of modern e-commerce.

Development object: A demo version (MVP) of an e-commerce web application.

Goal of the qualification work: To design and develop an e-commerce web application using a modern technology stack (Next.js, TypeScript, MongoDB) that demonstrates the practical application of contemporary software development methods and tools, while ensuring high performance, security, scalability, and user convenience.

Keywords: Next.js, TypeScript, Cloudinary, MongoDB, DATABASE, E-COMMERCE, WEB APPLICATION, JWT, RESTful API.

ЗМІСТ

ВСТУП

Сучасний світ характеризується стрімким розвитком цифрових технологій, що кардинально змінюють усі сфери життя, включаючи торгівлю. Електронна комерція стала невід'ємною частиною глобальної економіки, пропонуючи споживачам безпрецедентну зручність та доступність товарів і послуг. Інтернет-магазини відіграють центральну роль у цьому процесі, виступаючи як основний канал взаємодії між продавцями та покупцями у віртуальному просторі [1]. Вони не просто дублюють функції традиційних магазинів, а створюють нові можливості для бізнесу, такі як персоналізація пропозицій, аналіз поведінки клієнтів, оптимізація логістики та розширення географії продажів без значних інвестицій у фізичну інфраструктуру.

Розвиток інтернет-магазинів пройшов шлях від простих каталогів товарів до складних інтегрованих платформ, що пропонують широкий спектр функціональних можливостей: від зручного пошуку та фільтрації товарів до безпечних платіжних систем, інтеграції з соціальними мережами, програм лояльності та ефективної післяпродажної підтримки [2]. Зростання конкуренції на ринку електронної комерції вимагає від розробників постійного вдосконалення технологій та підходів до створення таких платформ, забезпечення їх високої продуктивності, надійності, безпеки та зручності використання на будь-яких пристроях.

Особливої актуальності набуває розробка рішень, які можуть швидко адаптуватися до мінливих вимог ринку та потреб користувачів. Використання сучасних фреймворків та бібліотек, що підтримують прогресивні архітектурні патерни та забезпечують високу швидкість розробки, стає ключовим фактором успіху. Технології, що дозволяють ефективно працювати з даними, забезпечувати масштабованість та інтегруватися зі сторонніми

сервісами, є основою для створення конкурентоспроможних рішень у сфері електронної комерції.

У контексті цих тенденцій, вибір правильного стеку технологій для розробки інтернет-магазину є критично важливим. Поява таких інструментів, як Next.js, який поєднує переваги серверного рендерингу та статичної генерації, TypeScript для підвищення надійності коду, а також гнучких баз даних типу MongoDB, відкриває нові можливості для створення високопродуктивних та масштабованих веб-додатків [3]. Інтеграція з хмарними сервісами, такими як Cloudinary для управління медіа контентом, та використання сучасних рішень для аутентифікації, як NextAuth.js, дозволяє зосередитися на розробці основної функціональності, делегуючи рутинні завдання спеціалізованим сервісам.

Ця кваліфікаційна робота присвячена дослідженню та практичній реалізації підходу до створення інтернет-магазину з використанням сучасного стеку технологій, що відповідає актуальним вимогам ринку та забезпечує високу якість кінцевого продукту.

Метою кваліфікаційної роботи є розробка та дослідження архітектурно-технологічних рішень для створення високопродуктивного, масштабованого та безпечного веб-додатку інтернет-магазину з використанням сучасного стеку технологій, зокрема Next.js, TypeScript та MongoDB. Робота спрямована на практичне застосування отриманих теоретичних знань та навичок для вирішення комплексної задачі розробки повноцінної платформи електронної комерції, включаючи реалізацію ключових функціональних модулів та інтеграцію зі сторонніми сервісами.

Предметом дослідження є архітектурні патерни, технології та інструменти, що використовуються для створення сучасних інтернет-магазинів, а також особливості їх застосування для забезпечення високої продуктивності, масштабованості, безпеки та зручності користування.

Результатом кваліфікаційної роботи є розроблена ДЕМО версія (MVP) інтернет-магазину, що демонструє реалізацію ключових функціональних можливостей, таких як:

- Система авторизації користувачів через Google OAuth;
- Персоналізований профіль користувача з історією замовлень та списком улюблених товарів;
- Функціональність кошика з можливістю оформлення замовлення;
- Каталог товарів з пошуком, фільтрацією та детальними сторінками товарів;
- Адміністративна панель для управління товарами;
- Інтеграція з базою даних MongoDB для зберігання даних;
- Інтеграція з сервісом Cloudinary для управління медіа контентом;
- Забезпечення безпеки даних користувачів за допомогою JWT токенів;
- Реалізація адаптивного дизайну для коректного відображення на різних пристроях.

Розроблений додаток слугує практичною демонстрацією застосування обраного стеку технологій та архітектурних рішень, а також може бути використаний як основа для подальшого розвитку та масштабування до повноцінної комерційної платформи.

РОЗДІЛ 1. ДОСЛІДЖЕННЯ ПРЕДМЕТНОЇ ОБЛАСТІ. ПОСТАНОВКА ЗАВДАНЬ ДОСЛІДЖЕННЯ

1.1 Аналіз стану області рішення задачі

Сфера електронної комерції (e-commerce) є однією з найбільш динамічних та швидкозростаючих галузей світової економіки. Щороку обсяг онлайн-продажів збільшується, а технології, що лежать в основі інтернет-магазинів, постійно вдосконалюються [4]. Це створює як значні можливості для бізнесу, так і виклики для розробників, які повинні створювати ефективні, безпечні та зручні платформи, що відповідають очікуванням сучасних споживачів.

1.1.1 Типи рішень для створення інтернет-магазинів

На ринку існує кілька основних підходів до створення інтернет-магазинів, кожен з яких має свої переваги та недоліки:

- **SaaS-платформи (Software as a Service):** Це готові рішення, які надаються за моделлю підписки. Прикладами є Shopify, BigCommerce, Wix eCommerce. Вони пропонують простий інтерфейс для налаштування магазину, інтегровані платіжні системи, хостинг та підтримку. Головні переваги – швидкий старт, відсутність необхідності в технічних знаннях для базового налаштування, регулярні оновлення та підтримка. Недоліки – обмежені можливості кастомізації, залежність від провайдера, що може обмежувати функціональність або вимагати додаткових витрат на розширення, а також щомісячна або щорічна плата [5];

- **Open Source платформи:** Це рішення з відкритим вихідним кодом, такі як WooCommerce (для WordPress), Magento Open Source, OpenCart, PrestaShop. Вони надають повний контроль над функціональністю та дизайном, широкі можливості для кастомізації та розширення за допомогою плагінів та тем. Переваги – гнучкість, відсутність ліцензійних платежів за сам софт (хоча можуть бути витрати на хостинг, розробку та підтримку), велика спільнота розробників. Недоліки – вимагають технічних знань для встановлення, налаштування, підтримки та забезпечення безпеки, а також можуть виникати проблеми сумісності при використанні великої кількості сторонніх розширень [6];
- **Custom Development (Індивідуальна розробка):** Створення інтернет-магазину “з нуля” або на базі сучасних фреймворків (наприклад, React, Vue, Angular для фронтенду; Node.js, Python (Django/Flask), Ruby on Rails, PHP (Laravel/Symfony) для бекенду). Цей підхід дозволяє створити унікальне рішення, повністю адаптоване до специфічних потреб бізнесу, з максимальною продуктивністю та масштабованістю. Переваги – повний контроль, можливість реалізації будь-якої функціональності, висока продуктивність. Недоліки – значно вищі витрати на розробку та підтримку, довший час до запуску проєкту, необхідність у висококваліфікованих розробниках [7].

Вибір між цими підходами залежить від багатьох факторів, включаючи бюджет проєкту, терміни реалізації, необхідний рівень кастомізації, технічні можливості команди та плани щодо масштабування бізнесу. Для стартапів або малих бізнесів, які потребують швидкого запуску з обмеженим бюджетом, SaaS-платформи можуть бути оптимальним вибором. Середні та великі

бізнеси, яким потрібна більша гнучкість та контроль, часто обирають Open Source рішення або індивідуальну розробку.

1.1.2 Ключові технології та тенденції у веб-розробці інтернет-магазинів

Сучасна веб-розробка для e-commerce активно використовує передові технології для покращення користувацького досвіду, оптимізації продуктивності та забезпечення безпеки. Серед ключових технологій та тенденцій можна виділити:

- **Прогресивні веб-додатки (PWA):** Технологія, що дозволяє веб-сайтам працювати як мобільні додатки, пропонуючи офлайн-доступ, push-сповіщення та швидке завантаження. PWA покращують залученість користувачів та можуть слугувати альтернативою або доповненням до нативних мобільних додатків [8];
- **Архітектура Microservices:** Розбиття монолітного додатку на невеликі, незалежні сервіси, що взаємодіють між собою. Це підвищує масштабованість, стійкість до збоїв та спрощує розробку та розгортання окремих компонентів [9];
- **JAMstack (JavaScript, APIs, Markup):** Сучасна архітектура для створення швидких та безпечних веб-сайтів та додатків. Фронтенд (JavaScript) взаємодіє з API для отримання даних та функціональності, а контент попередньо генерується (Markup). Цей підхід забезпечує високу продуктивність та кращу безпеку [10];
- **Headless Commerce:** Відділення фронтенду (інтерфейсу користувача) від бекенду (логіки електронної комерції). Це дозволяє використовувати один

бекенд для різних каналів продажу (веб-сайт, мобільний додаток, IoT-пристрої) та надає розробникам повну гнучкість у створенні користувацького інтерфейсу [11];

- **Використання API та інтеграції:** Активне використання API для інтеграції зі сторонніми сервісами, такими як платіжні шлюзи, служби доставки, CRM-системи, маркетингові платформи. Це дозволяє розширювати функціональність магазину та автоматизувати бізнес-процеси [12];
- **Серверний рендеринг (SSR) та статична генерація (SSG):** Техніки рендерингу веб-сторінок на сервері перед відправленням клієнту. SSR забезпечує актуальність даних для динамічного контенту, тоді як SSG ідеально підходить для статичного контенту, забезпечуючи надзвичайно високу швидкість завантаження. Фреймворки, такі як Next.js, поєднують ці підходи, оптимізуючи продуктивність та SEO [3].

Вибір стеку технологій для індивідуальної розробки інтернет-магазину часто включає сучасні JavaScript-фреймворки для фронтенду (React, Vue, Angular), Node.js або інші мови для бекенду, а також різні типи баз даних (реляційні, NoSQL) залежно від структури даних та вимог до масштабованості. MongoDB, як документо-орієнтована база даних, здобула популярність у e-commerce завдяки своїй гнучкості схеми, масштабованості та високій продуктивності для роботи з великими обсягами різнорідних даних, що ідеально підходить для зберігання інформації про товари з різними атрибутами, замовлення та користувачів [13].

1.2 Аналіз технології Next.js

Next.js – це відкритий React-фреймворк для розробки веб-додатків, створений компанією Vercel. Він надає розробникам потужні інструменти та оптимізації “з коробки”, що значно прискорює процес розробки та покращує продуктивність кінцевого продукту. Next.js базується на React, але розширює його можливості, додаючи функціональність серверного рендерингу, статичної генерації, маршрутизації файлової системи, розділення коду та багато іншого [3].

1.2.1 Особливості та можливості Next.js

- **Серверний рендеринг (SSR):** Next.js дозволяє рендерити React-компоненти на сервері для кожного запиту. Це забезпечує швидке завантаження сторінок, оскільки користувач отримує вже готовий HTML, а не порожню сторінку, яку потім потрібно заповнювати JavaScript. SSR також покращує SEO, оскільки пошукові роботи бачать повністю сформований контент [3];
- **Статична генерація (SSG):** Для сторінок зі статичним контентом Next.js може генерувати HTML-файли під час збірки проєкту. Ці файли можуть бути кешовані та подаватися з CDN, що забезпечує надзвичайно високу швидкість завантаження та знижує навантаження на сервер [3];
- **Гібридний підхід:** Next.js дозволяє використовувати SSR, SSG та клієнтський рендеринг (CSR) в одному проєкті, обираючи найбільш підходящий метод для кожної сторінки. Це надає максимальну гнучкість та можливість оптимізувати продуктивність залежно від типу контенту [3];
- **Маршрутизація на основі файлової системи:** Маршрути в Next.js визначаються структурою папок та файлів у директорії pages (або app в нових

версіях). Це спрощує організацію коду та робить маршрутизацію інтуїтивно зрозумілою [3];

- **API Routes:** Next.js дозволяє створювати бекенд-функціональність (API endpoints) безпосередньо в проєкті, використовуючи файли в директорії pages/api (або app/api). Це зручно для створення невеликих API або інтеграції з базами даних та сторонніми сервісами [3];
- **Автоматичне розділення коду (Code Splitting):** Next.js автоматично розділяє JavaScript-код на невеликі частини (chunks), які завантажуються лише тоді, коли вони потрібні. Це зменшує час першого завантаження сторінки [3];
- **Оптимізація зображень:** Вбудований компонент next/image автоматично оптимізує зображення, змінюючи їх розмір, формат та використовуючи lazy loading, що покращує продуктивність та користувацький досвід [3];
- **Підтримка TypeScript та ESLint:** Next.js має вбудовану підтримку TypeScript для статичної типізації та ESLint для аналізу коду та виявлення помилок, що підвищує якість та надійність розробки [3].

1.2.2 Переваги та недоліки Next.js

Переваги:

- Висока продуктивність завдяки SSR та SSG;
- Покращене SEO;
- Швидка розробка завдяки вбудованим функціям та конвенціям;
- Гнучкість у виборі методу рендерингу для кожної сторінки;
- Можливість створення бекенду за допомогою API Routes;

- Автоматична оптимізація зображень та розділення коду;
- Активна спільнота та хороша документація.

Недоліки:

- Може бути надмірним для дуже простих статичних сайтів;
- Вимагає розуміння концепцій SSR та SSG;
- Певні обмеження при використанні специфічних бібліотек, які працюють лише на клієнті, з SSR;
- Залежність від екосистеми React.

1.2.3 Порівняння Next.js з іншими фронтенд-фреймворками

Порівняльна характеристика обраного фреймворку Next.js з іншими відомими фреймворками наведено у табл. 1.1.

Таблиця 1.1 – Порівняльна характеристика відомих JS фреймворків

Характеристика	**Next.js**	**Create React App (CRA)**	**Vue.js**	**Angular**
Тип фреймворку	Фреймворк для React (з SSR, SSG)	Набір інструментів для React (CSR)	Прогресивний фреймворк (CSR, SSR через Nuxt.js)	Повноцінний фреймворк (CSR, SSR через Angular Universal)
Рендеринг	SSR, SSG, CSR	CSR	CSR (SSG/SSR через Nuxt.js)	CSR (SSR через Angular Universal)

Маршрутизація	На основі файлової системи	Потребує бібліотек (react-router-dom)	Потребує бібліотек (Vue Router)	Вбудована
API Routes	Вбудовано	Потребує окремого бекенду	Потребує окремого бекенду	Потребує окремого бекенду
Оптимізація	Автоматична (зображення, код)	Потребує ручного налаштування/бібліотек	Потребує ручного налаштування/бібліотек	Потребує ручного налаштування/бібліотек
Використання	Додатки з SSR/SSG, e-commerce, блоги	Односторінкові додатки (SPA)	SPA, невеликі додатки, інтеграція	Великі корпоративні додатки
Складність вивчення	Середня	Низька	Низька/Середня	Висока

Next.js є чудовим вибором для проєктів, де важливі продуктивність, SEO та швидкість розробки, особливо для додатків з динамічним та статичним контентом, таких як інтернет-магазини.

1.3 Аналіз технології MongoDB

MongoDB – це документо-орієнтована NoSQL база даних з відкритим вихідним кодом. Замість традиційних таблиць та рядків, як у реляційних базах даних, MongoDB зберігає дані у гнучких, JSON-подібних документах

(BSON), що об'єднуються у колекції [13]. Ця модель даних забезпечує високу гнучкість та масштабованість, що робить MongoDB популярним вибором для сучасних веб-додатків, включаючи платформи електронної комерції.

1.3.1 Особливості та можливості MongoDB

- **Документо-орієнтована модель даних:** Дані зберігаються у вигляді документів, що дозволяє групувати пов'язану інформацію в одному місці. Схема документів може бути гнучкою, що спрощує роботу з різнорідними даними [13];
- **Висока доступність:** MongoDB підтримує реплікацію (Replica Sets), що забезпечує автоматичне перемикання на резервний сервер у разі збою основного, підвищуючи доступність даних [13];
- **Горизонтальна масштабованість:** MongoDB підтримує шардинг (Sharding), що дозволяє розподіляти дані по декількох серверах (шардах). Це забезпечує можливість масштабування бази даних при зростанні обсягів даних та навантаження [13];
- **Висока продуктивність:** MongoDB оптимізована для швидких операцій читання та запису. Вона підтримує індекси для прискорення пошуку та агрегаційні конвеєри для складних запитів [13];
- **Гнучкість схеми (Schema-less):** На відміну від реляційних баз даних, де схема жорстко визначена, MongoDB дозволяє документам в одній колекції мати різний набір полів. Це спрощує внесення змін до структури даних без необхідності міграції всієї бази [13];
- **Широкий набір драйверів:** MongoDB має офіційні та спільнотою підтримувані драйвери для багатьох мов програмування, включаючи JavaScript (для Node.js), Python, Java, C#, PHP та інші [13].

1.3.2 Переваги та недоліки MongoDB

Переваги:

- Гнучка схема даних, що спрощує роботу з різнорідними даними;
- Висока масштабованість та доступність;
- Висока продуктивність для операцій читання та запису;
- Швидка розробка завдяки гнучкості та документо-орієнтованій моделі;
- Добре підходить для додатків з великими обсягами даних та високим навантаженням.

Недоліки:

- Може бути менш ефективною для складних реляційних зв'язків порівняно з реляційними базами даних;
- Гнучкість схеми може призвести до проблем з цілісністю даних, якщо не використовувати валідацію на рівні додатку або схеми (наприклад, за допомогою Mongoose);
- Споживання пам'яті може бути вищим порівняно з деякими іншими базами даних;
- Не підтримує ACID-транзакції на рівні декількох документів (хоча в останніх версіях з'явилася підтримка транзакцій на рівні колекцій).

1.3.3 Порівняння MongoDB з реляційними базами даних (наприклад, PostgreSQL, MySQL)

Результат наведено у табл. 1.2.

Таблиця 1.2 – Порівняльна таблиця характеристик NoSQL (MongoDB) з реляційними БД

Характеристика	**MongoDB (NoSQL)**	**Реляційні БД (SQL) (PostgreSQL, MySQL)**
Модель даних	Документо-орієнтована (JSON-подібні документи)	Реляційна (таблиці, рядки, стовпці)
Схема	Гнучка (Schema-less)	Жорстко визначена
Масштабованість	Горизонтальна (шардинг)	Вертикальна (збільшення потужності сервера), горизонтальна через складні рішення
Транзакції	Транзакції на рівні документа/колекції	ACID-транзакції на рівні декількох таблиць
Мова запитів	MongoDB Query Language (на основі JSON)	SQL (Structured Query Language)
Зв'язки між даними	Вбудовування документів, посилання	Зв'язки між таблицями (JOIN)
Використання	Великі обсяги даних, швидкий розвиток, гнучка структура даних	Складні реляційні зв'язки, високі вимоги до цілісності даних

MongoDB є відмінним вибором для проєктів, де важливі гнучкість, масштабованість та швидкість обробки великих обсягів даних, особливо для додатків з мінливою структурою даних, таких як профілі користувачів,

каталоги товарів з різними атрибутами та журнали подій.

1.4 Технології та інструменти розробки

Проєкт розробки ДЕМО версії інтернет-магазину був реалізований з використанням сучасного стеку технологій, що забезпечує ефективність розробки, високу продуктивність та масштабованість. Вибір цих інструментів ґрунтувався на їх популярності, активній підтримці спільнотою, наявності необхідної функціональності та відповідності вимогам проєкту.

Середовище виконання: Node.js 16+. Використання Node.js дозволило використовувати JavaScript як для фронтенду, так і для бекенду, що спрощує процес розробки та обміну даними.

Фреймворк: Next.js 14. Як зазначено у ВСТУПі та розділі 1.2, Next.js був обраний завдяки підтримці SSR та SSG, автоматичному розділенню коду, вбудованій підтримці TypeScript та API Routes, що є критично важливим для створення високопродуктивного та SEO-оптимізованого інтернет-магазину.

Мова програмування: TypeScript. Використання TypeScript додало статичну типізацію до JavaScript, що значно підвищило надійність коду, спростило його підтримку та виявлення помилок на ранніх етапах розробки. Це особливо важливо для великих проєктів, де працює команда розробників.

Бібліотека стилів: Tailwind CSS. Цей утилітний CSS-фреймворк дозволив швидко та ефективно створювати адаптивні та сучасні інтерфейси користувача без необхідності написання великої кількості власного CSS-коду.

Підхід “utility-first” прискорює процес стилізації та робить CSS більш передбачуваним [14].

База даних: MongoDB та ORM система Mongoose. MongoDB була обрана як гнучка NoSQL база даних, що ідеально підходить для зберігання різнорідних даних інтернет-магазину. Mongoose як бібліотека для Node.js надала зручний інтерфейс для взаємодії з MongoDB, включаючи моделювання даних та валідацію [13].

CDN (Content Delivery Network): Cloudinary з безкоштовним планом. Використання CDN для зберігання та оптимізації медіаконтенту (зображень товарів) значно прискорило завантаження сторінок та зменшило навантаження на основний сервер. Cloudinary надав зручні інструменти для завантаження, трансформації та доставки зображень [15].

Авторизація: Next-auth. Ця бібліотека спростила реалізацію системи аутентифікації в Next.js додатку, надавши готові рішення для авторизації через різні провайдери, включаючи Google OAuth, та управління сесіями користувачів за допомогою JWT токенів [16].

Менеджер стану на клієнті: Zustand. Легка та швидка бібліотека для управління станом в React-додатках. Вона була використана для створення глобального стану AppStore, що синхронізує дані між сервером та клієнтськими компонентами, спрощуючи взаємодію та оновлення інтерфейсу [17].

Використання цього стеку технологій дозволило ефективно реалізувати поставлені завдання, створити функціональну ДЕМО версію інтернет-магазину та продемонструвати можливості сучасних підходів до веб-розробки. Кожен з обраних інструментів відіграв важливу роль у досягненні цілей проєкту, забезпечуючи необхідний рівень продуктивності, гнучкості та зручності розробки.

1.5 Як працює Next.js

Next.js є фреймворком, побудованим поверх React, що додає структуру та функціональність для створення повноцінних веб-додатків. Його основна відмінність полягає у підході до рендерингу та маршрутизації [3].

1.5.1 Архітектура рендерингу

Next.js підтримує кілька стратегій рендерингу, які можна комбінувати в одному проєкті:

- **Client-Side Rendering (CSR):** Стандартний підхід React, де JavaScript завантажується в браузері та рендерить інтерфейс користувача безпосередньо на клієнті. Це добре підходить для інтерактивних частин додатку або сторінок, що потребують даних, отриманих після завантаження сторінки.
- **Server-Side Rendering (SSR):** Сторінка рендериться на сервері для кожного запиту користувача. Сервер надсилає готовий HTML-код браузеру, який потім “оживляється” за допомогою JavaScript (процес, що називається hydration). SSR покращує час до першого відображення контенту (First Contentful Paint) та є корисним для SEO [3].

- **Static Site Generation (SSG):** Сторінки генеруються в HTML під час збірки проєкту. Це ідеально для сторінок зі статичним контентом, який не змінюється часто (наприклад, сторінки товарів, блог-пости). Згенеровані файли можуть бути розміщені на CDN для надзвичайно швидкої доставки [3].
- **Incremental Static Regeneration (ISR):** Дозволяє оновлювати статично згенеровані сторінки після їх розгортання без необхідності повної перебудови проєкту. Це комбінує переваги SSG з можливістю відображати свіжіші дані [3].

Next.js автоматично визначає, яку стратегію рендерингу використовувати для кожної сторінки, або дозволяє явно вказати це за допомогою функцій getStaticProps, getServerSideProps або getStaticPaths.

1.5.2 Маршрутизація

Маршрутизація в Next.js базується на файловій системі. Кожен файл з розширенням .js, .jsx, .ts, або .tsx у директорії pages (або app) автоматично стає маршрутом. Наприклад, pages/products.js буде доступний за адресою /products, а pages/products/[slug].js дозволить створювати динамічні маршрути, де [slug] є параметром [3].

1.5.3 API Routes

Next.js також дозволяє створювати API endpoints, розміщуючи файли в директорії pages/api (або app/api). Ці файли експортують функції-обробники, які отримують об'єкти запиту (request) та відповіді (response), дозволяючи

реалізувати бекенд-логіку, таку як взаємодія з базою даних, обробка форм або інтеграція зі сторонніми сервісами [3].

1.5.4 Оптимізації

Next.js включає низку вбудованих оптимізацій:

- **Автоматичне розділення коду:** Код кожної сторінки розділяється на окремі файли, що завантажуються лише при переході на цю сторінку [3].
- **Оптимізація зображень:** Компонент next/image автоматично оптимізує зображення, використовуючи сучасні формати (WebP), змінюючи розмір залежно від пристрою та застосовуючи lazy loading [3].
- **Попереднє завантаження (Prefetching):** Next.js автоматично попередньо завантажує JavaScript для сторінок, на які є посилання на поточному екрані, що прискорює переходи між сторінками.

1.6 Можливості Next.js для розробки інтернет-магазинів

Next.js чудово підходить для розробки інтернет-магазинів завдяки своїм можливостям:

- **SEO-оптимізація:** SSR та SSG забезпечують кращу індексацію контенту пошуковими системами, що є критично важливим для видимості інтернет-магазину [3].
- **Висока продуктивність:** Швидке завантаження сторінок завдяки SSR/SSG та оптимізаціям покращує користувацький досвід та зменшує показник відмов.

- **Зручна розробка:** Маршрутизація на основі файлової системи та API Routes спрощують структуру проєкту та розробку як фронтенду, так і бекенду в межах одного репозиторію.
- **Масштабованість:** Можливість вибору стратегії рендерингу та підтримка API Routes дозволяють будувати масштабовані додатки.
- **Екосистема React:** Використання React дозволяє залучати розробників зі знанням цієї популярної бібліотеки та використовувати її багату екосистему компонентів та бібліотек.

1.7 Висновки до Розділу 1

У першому розділі було проведено глибокий аналіз предметної області електронної комерції, розглянуто актуальні тенденції веб-розробки інтернет-магазинів та порівняно різні технологічні підходи. Особливу увагу приділено фреймворку Next.js, який поєднує переваги SSR, SSG і CSR, що є оптимальним рішенням для створення швидких, SEO-оптимізованих веб-застосунків. Також досліджено можливості MongoDB як гнучкої NoSQL бази даних, яка ідеально підходить для проєктів з нерегламентованою структурою даних. Проведений аналіз дозволив обґрунтовано обрати стек технологій Next.js + TypeScript + MongoDB для реалізації інтернет-магазину.

1.8 Постановка задач дослідження

На основі проведеного аналізу предметної області, сучасних технологій та інструментів, що використовуються у сфері розробки інтернет-магазинів, сформульовано наступні задачі дослідження:

1. Проаналізувати існуючі технологічні підходи до створення інтернет-магазинів (SaaS, open source, індивідуальна розробка).
2. Визначити переваги та доцільність використання фреймворку Next.js для реалізації високопродуктивного веб-додатку електронної комерції.
3. Обґрунтувати вибір бази даних MongoDB як оптимального рішення для зберігання динамічно структурованих даних інтернет-магазину.
4. Дослідити можливості інтеграції з зовнішніми сервісами (Cloudinary, NextAuth.js) для управління медіаконтентом та реалізації аутентифікації.
5. Сформувати архітектурну концепцію майбутнього веб-застосунку з урахуванням вимог до масштабованості, безпеки та продуктивності.
6. Підготувати технічне підґрунтя для практичної реалізації MVP-версії інтернет-магазину.

РОЗДІЛ 2. ОГЛЯД ВІДОМИХ ПРОЄКТНИХ РІШЕНЬ

2.1 Платформи для розробки інтернет-магазинів

У сучасному світі існує велика кількість платформ для створення інтернет-магазинів, які надають розробникам різноманітні інструменти та можливості для реалізації своїх проєктів. Найпопулярніші серед них включають:

1. **Shopify**:
   - *Платформа:* SaaS (Software as a Service).
   - *Мова програмування:* Ruby on Rails.
   - *Особливості:* Shopify надає зручний інтерфейс для створення та управління інтернет-магазином, підтримку багатьох платіжних систем, велику кількість тем та плагінів. Платформа закрита, але пропонує API для розширення функціональності.

2. **WooCommerce**:
   - *Платформа:* Плагін для WordPress.
   - *Мова програмування:* PHP.
   - *Особливості:* WooCommerce дозволяє легко інтегрувати інтернет-магазин з блогами та іншими функціями WordPress. Це відкрите рішення, що надає велику кількість розширень і тем, але вимагає хостингу та налаштування.

3. **Magento**:

- *Платформа:* Відкрите ПЗ, доступне у версіях Open Source та Commerce (SaaS).
- *Мова програмування:* PHP.
- *Особливості:* Magento відома своєю гнучкістю та масштабованістю, підходить для великих інтернет-магазинів з високим трафіком. Має широку екосистему розширень і модулів.

4. **BigCommerce**:
   - *Платформа:* SaaS.
   - *Мова програмування:* Здебільшого JavaScript (для створення кастомних рішень можна використовувати різні мови через API).
   - *Особливості:* BigCommerce пропонує потужні інструменти для управління інтернет-магазином, багатоканальні продажі, інтеграції з соціальними мережами та маркетплейсами. Платформа закрита, але має розширюваний API.

2.2 Відкритість та мови програмування

Багато платформ для розробки інтернет-магазинів є відкритими або надають можливості для розширення функціональності через API та SDK.

Деякі з них:

1. **OpenCart**:
   - *Платформа:* Відкрите ПЗ.
   - *Мова програмування:* PHP.

- *Особливості:* Легка у використанні, має багато розширень і тем, активно підтримується спільнотою розробників.

2. **PrestaShop**:
   - *Платформа:* Відкрите ПЗ.
   - *Мова програмування:* PHP.
   - *Особливості:* Гнучка платформа з великою кількістю модулів та тем, підтримує багатомовність та багатовалютність.
3. **Saleor**:
   - *Платформа:* Відкрите ПЗ.
   - *Мова програмування:* Python (Django).
   - *Особливості:* Сучасна платформа з GraphQL API, фокус на продуктивність та гнучкість, підходить для масштабованих проєктів.
4. **Sylius**:
   - *Платформа:* Відкрите ПЗ.
   - *Мова програмування:* PHP (Symfony).
   - *Особливості:* Підходить для створення кастомних рішень, підтримує REST API, модульна архітектура.

2.3 Порівняння відкритих та закритих рішень

Відкриті рішення (open-source) для інтернет-магазинів зазвичай надають більше можливостей для кастомізації та розширення функціональності, що дозволяє розробникам створювати унікальні рішення, повністю відповідні потребам клієнтів. Вони активно підтримуються

спільнотами розробників, що сприяє швидкому вирішенню проблем та появі нових функцій.

Закриті платформи (SaaS-рішення) часто мають простіший інтерфейс та швидше налаштування, але обмежують можливості кастомізації. Вони підходять для малих та середніх бізнесів, які не мають ресурсів для розробки та підтримки власних рішень.

У виборі платформи важливо враховувати вимоги проєкту, ресурси для розробки та підтримки, а також перспективи масштабування бізнесу.

Відкриті рішення (наприклад, OpenCart, PrestaShop, Sylius, Saleor):

- Дозволяють повний контроль над кодом та архітектурою проєкту;
- Підтримують глибоку кастомізацію — можна змінювати функціональність відповідно до унікальних бізнес-потреб;
- Активні спільноти розробників забезпечують швидкий доступ до оновлень, рішень проблем та великої кількості плагінів і модулів;
- Потребують більше технічних знань або окремої команди для налаштування, розгортання та підтримки;
- Вимагають власного хостингу та відповідальності за безпеку й стабільність роботи.

Закриті рішення (наприклад, Shopify, Wix):

- Забезпечують швидкий старт — користувач може створити інтернет-магазин без глибоких технічних знань;

- Вся інфраструктура, оновлення, безпека й техпідтримка надаються провайдером;
- Обмежені можливості кастомізації — користувач прив'язаний до функціоналу, який доступний у межах платформи;
- Часто мають підписку або комісію за транзакції;
- Підходять для малого та середнього бізнесу, стартапів, які не мають ресурсів для повноцінної розробки.

Додаткове порівняння закритих та відкритих рішень наведено у табл. 2.1.

Таблиця 2.1 – Порівняння відкритих та закритих рішень платформ інтернет-магазинів

Критерій	**Відкриті рішення**	**Закриті рішення (SaaS)**
Доступ до коду	Повний	Обмежений або відсутній
Кастомізація	Максимальна	Лімітована
Технічні вимоги	Високі	Мінімальні
Початкова вартість	Низька/безкоштовна (але час на розробку)	Абонплата або тарифи
Швидкість запуску	Повільніше (налаштування, хостинг)	Дуже швидкий старт

Масштабованість	Висока, але залежить від реалізації	Може бути обмежено платформою
Безпека та підтримка	Залежить від власника/команди	Відповідальність на провайдері

2.4 Вибір Next.js для розробки інтернет-магазину

Next.js був обраний як основний фреймворк для розробки інтернет-магазину через його численні переваги, які відповідають вимогам та завданням проєкту. Нижче наведено детальний огляд причин, чому Next.js підходить для цього проєкту.

2.4.1 Серверний рендеринг (SSR) та статична генерація (SSG)

Однією з головних переваг Next.js є підтримка серверного рендерингу (SSR) та статичної генерації (SSG). Це дозволяє оптимізувати швидкість завантаження сторінок, покращити SEO та забезпечити швидку реакцію користувача на дії.

- **Серверний рендеринг (SSR):** Сторінки рендеряться на сервері при кожному запиті, що дозволяє відображати актуальні дані для кожного користувача. Це особливо важливо для сторінок з динамічним контентом, таких як профілі користувачів та корзини покупок.
- **Статична генерація (SSG):** Сторінки рендеряться під час побудови проєкту, що забезпечує швидке завантаження та високу продуктивність для статичних сторінок, таких як сторінка товару чи публічна сторінка з

товарами.

2.4.2 Автоматичне розділення коду

Next.js автоматично розділяє код на частини, що зменшує розмір завантажуваних файлів та покращує продуктивність додатку. Це означає, що користувачі завантажують лише необхідні частини коду для конкретної сторінки, що пришвидшує час першого завантаження та взаємодії.

2.4.3 Підтримка TypeScript

Next.js має вбудовану підтримку TypeScript, що дозволяє використовувати статичну типізацію для підвищення надійності та зручності розробки. Використання TypeScript допомагає виявляти помилки на етапі розробки, забезпечує автодоповнення та покращує читабельність коду.

2.4.4 Інтеграція з Tailwind CSS

Next.js легко інтегрується з Tailwind CSS, що дозволяє швидко створювати стильні та адаптивні інтерфейси користувача. Tailwind CSS надає зручний набір утиліт для стилізації, що спрощує розробку та підтримку CSS-коду.

2.4.5 Підтримка API Routes

Next.js надає можливість створювати API маршрути, що дозволяє об'єднувати фронтенд та бекенд у єдиній кодовій базі. Це спрощує розробку та обслуговування проєкту, а також забезпечує легку інтеграцію з іншими сервісами та базами даних.

2.4.6 Вбудована підтримка аутентифікації

Next.js має відмінну інтеграцію з NextAuth.js, що дозволяє легко реалізувати аутентифікацію через різні провайдери, зокрема Google OAuth. Це забезпечує безпечний та зручний вхід для користувачів, що є важливим для інтернет-магазину.

2.4.7 Масштабованість та гнучкість

Next.js пропонує високу масштабованість та гнучкість, що дозволяє проєкту рости разом із бізнесом. Він підтримує модульну архітектуру, що полегшує додавання нових функцій та інтеграцію з іншими технологіями.

2.5 Обгрунтування вибору БД - MongoDB

1. **Гнучкість схеми**

MongoDB використовує гнучку схему даних (schema-less), що дозволяє зберігати документи у форматі JSON. Це забезпечує велику гнучкість у зберіганні різнорідних даних, що особливо корисно для інтернет-магазину з різноманітними продуктами та змінними атрибутами.

- *Динамічні структури:* Можливість змінювати структуру документів без необхідності перероблювати всю базу даних.
- *Різнорідні документи:* Легке зберігання різних типів даних в одному і тому ж колекції.

2. **Масштабованість**

MongoDB пропонує високу горизонтальну масштабованість завдяки підтримці шардингу. Це дозволяє розподіляти дані по декількох серверах, що забезпечує стабільну роботу системи при зростанні обсягів даних та збільшенні навантаження.

- *Шардінг:* Розподіл даних між різними серверами для підвищення продуктивності та доступності.
- *Реплікація:* Використання реплік для підвищення надійності та забезпечення безперебійної роботи.

3. **Висока продуктивність**

MongoDB забезпечує високу продуктивність завдяки своєму документо-орієнтованому підходу та індексації. Це дозволяє швидко виконувати операції зчитування та запису, що є критичним для інтернет-магазину з великим обсягом транзакцій та запитів.

- *Індексація:* Підтримка складних індексів для прискорення пошуку та фільтрації даних.

- *Документо-орієнтована модель:* Оптимізована для швидких операцій зчитування та запису.

4. **Інтеграція з TypeScript та Node.js**

MongoDB чудово інтегрується з JavaScript та Node.js, що дозволяє легко використовувати його в Next.js проєктах. Використання Mongoose як ORM спрощує роботу з базою даних, забезпечуючи зручний та інтуїтивно зрозумілий інтерфейс для взаємодії з даними.

- *Mongoose:* Потужна бібліотека для роботи з MongoDB, яка надає інструменти для валідації, створення схем та взаємодії з базою даних.
- *JSON формат:* Використання JSON для зберігання даних дозволяє легко маніпулювати ними у JavaScript.

5. **Надійність та безпека**

MongoDB надає потужні механізми для забезпечення надійності та безпеки даних. Це включає в себе функції для резервного копіювання, відновлення даних, управління доступом та шифрування.

- *Резервне копіювання та відновлення:* Інструменти для створення резервних копій та відновлення даних у разі збою.
- *Управління доступом:* Розширені можливості для контролю доступу до даних, включаючи аутентифікацію та авторизацію.
- *Шифрування:* Підтримка шифрування даних як на стороні сервера, так і на стороні клієнта.

2.6 Проєктування бази даних

Діаграми “сутність-зв'язок”, або ER-діаграми, показують, як “сутності” (люди, об'єкти або поняття) пов'язані один з одним у рамках системи. Кардинальність, представлена за допомогою нотації «вороні лапки», вказує на кількісні атрибути відносин між сутностями або наборами сутностей.

Загальна структура сутностей та їх зв’язків представлена на рис. 2.1.

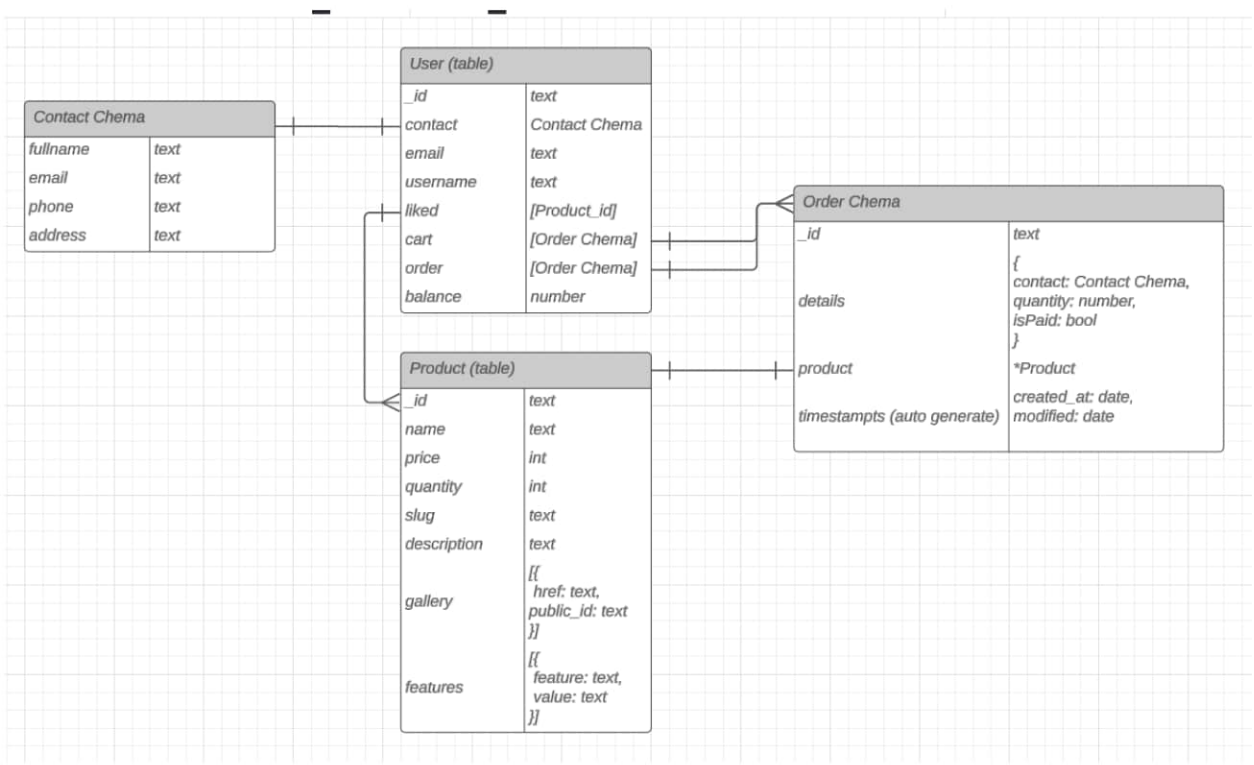


Рисунок 2.1 – схема бази даних

1. Схема користувача має як базові поля email, username що даються при авторизації в google auth автоматично, так і повний стан дій користувача.

Таких як:

- Лайкнуті товари (масив зі строчками _id продуків);
- Корзина (масив з об’єктами за схемою OrderChema);
- Замовлення (масив з об’єктами за схемою OrderChema);
- Контактні данні (за схемою ContactChema);
- Баланс рахунку (звичайна цифра що імітує рахунок).

Детальну структуру колекції користувачів представлено на рис. 2.2 у вигляді коду.

```
import { Schema, model, models } from 'mongoose'
import { ContactChema, OrderSchema } from './orders'

const UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'Email already exists!'],
    required: [true, 'Email is required!'],
  },
  username: {
    type: String,
    required: [true, 'Username is required!'],
},
  image: { type: String },
  liked: [{ type: String, default: [] }],

  cart: [OrderSchema],
  order: [OrderSchema],
  contact: ContactChema,

  balance: { type: Number, default: 0 },
})

const User = models.User || model('User', UserSchema)

export default User

```

Рисунок 2.2 – Схема колекції користувача

1. Схема продуктів підтримає усі операції (CRUD) через сторінку **/admin** або сторонній API запити. Наприклад, Postman.

- slug – це посилання на товар у форматі назви продукта та дефісом замість пробілів. Наприклад: /products/tws-pro-4.
- price та quantity це ціна за одиницю продукта та кількість у наявності.
- name та description – це назва товару та його опис.
- gallery – містить обьєкт з ключами href, public_id. Перший для посилання на стороні клієнта, други для видалення зі сторони бекенда.
- features – як таблиця з полями title, value, тобто назва та значення характеристики.

Схема колекції продуктів подана на рис. 2.3 у вигляді коду.

```
import { ProductType } from 'product-types'
import mongoose, { models, model } from 'mongoose'

// Extend the Mongoose Document interface with your custom fields
export interface ProductDocument extends ProductType, Document {}
// Define the schema for the product
const productSchema = new mongoose.Schema<ProductDocument>({
  name: { type: String, required: true },
  price: { type: Number, required: true },
  quantity: { type: Number, required: true, min: 0 },
  slug: { type: String, required: true, unique: true },
  description: { type: String, required: true, default: 'Опис' },

  gallery: {
    type: [
      {
        href: String,
        public_id: String,
      },
    ],
    default: [],
  },

  features: [
    {
      title: String,
      value: String,
    },
  ],
})

// Create the model based on the schema
const Product =
  models.Product || model<ProductDocument>('Product', productSchema)

export default Product

```

Рисунок 2.3 – Схема колекції продуктів

2. В цьому файлі зазначено дві схеми – контакти та замовлення:

- Схема контактів мають поля **fullname, email, phone, address** – усі поля текст та усі обов’язково повинні бути заповненими
- Timestamps – автоматично створює поля **created_at / modified_at**.

- Схема замовлення це лише об’єкт з полями product, details.

Поле product – це посилання на поточний товар по полю **_id** продукта.

Поле details – це об’єкт що містить деталі замовлення. Контактні дані, кількість, та чи оплатив користувач покупку.

Візуальне представлення цих двох колекцій показано на рис. 2.4 у вигляді коду.

```
import mongoose, { models, model, Schema } from 'mongoose'

export const ContactChema = new Schema({
  fullName: { type: String, required: true },
  email: { type: String, required: true },
  phone: { type: String, required: true },
  address: { type: String, required: true },
})

// Define the schema for the product
export const OrderSchema = new Schema(
  {
    product: {
      type: Schema.Types.ObjectId,
      ref: 'Product',
      required: true,
    },
    details: {
      contact: { type: ContactChema },
      quantity: { type: Number, required: true, default: 1 },
      isPaid: { type: Boolean, required: true, default: false },
    },
},

  { timestamps: true },
)

// Create the model based on the schema
const Orders = models.Orders || model('Orders', OrderSchema)

export default Orders
```

Рисунок 2.4 – схема колекції замовлень та контактів

2.7 Архітектурний паттерн розробки застосунку

2.7.1 Server та сторонні сервіси

Загальна схема взаємодії серверної частини з зовнішніми сервісами наведена на рис. 2.5.

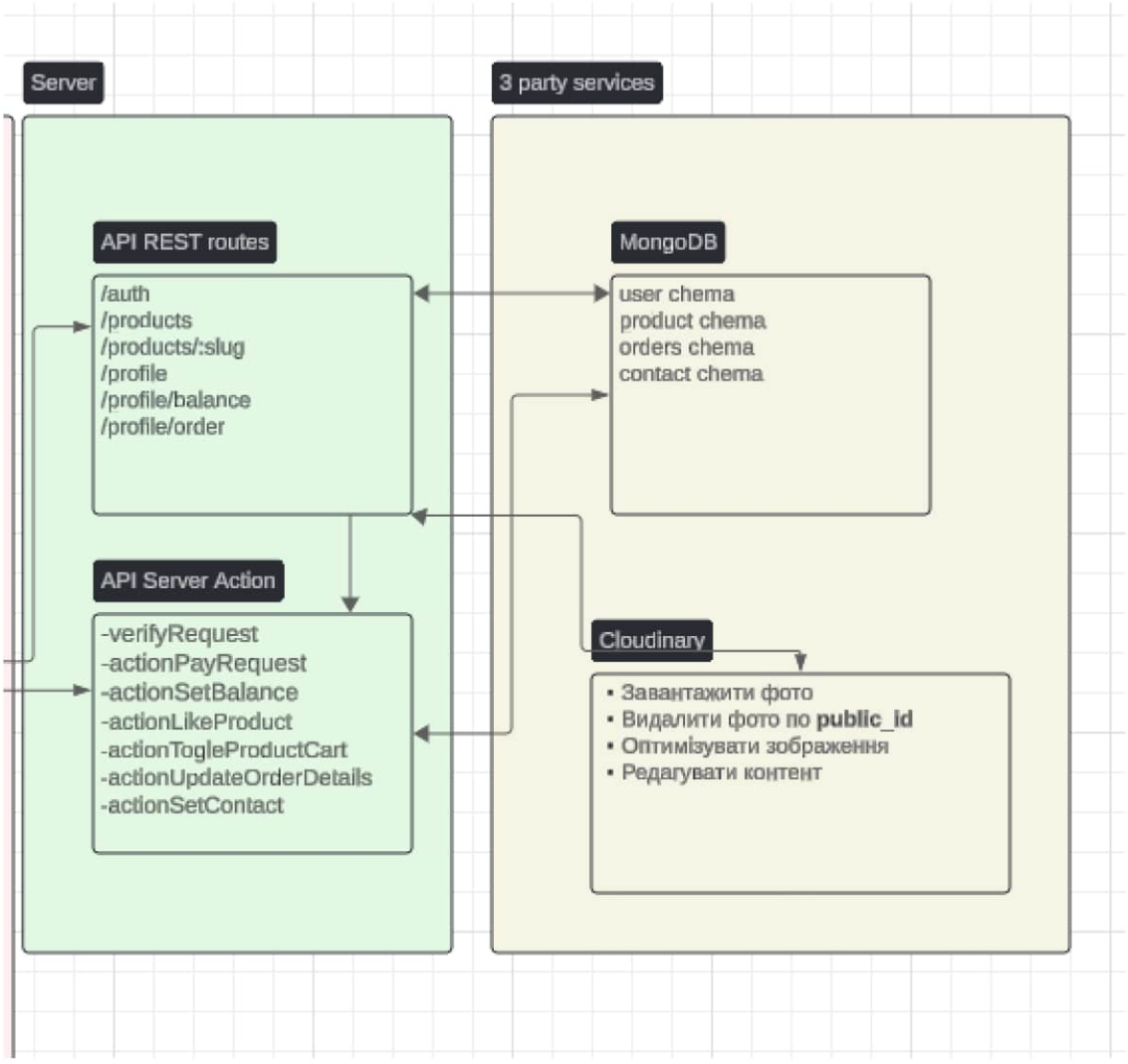


Рисунок 2.5 – Взаємодія серверного блока та сторонніх сервісів

Rest API routes.

Серверна частина має Rest API routes що автоматично генеруються залежно від шляху файлу, наприклад:

Рисунок 2.6 – папки проєкту api

Згідно рис. 2.6 маємо відповідні роути /api/auth; /api/products; /api/products/:slug. Але для цього необхідно створити **route.ts** файл та експортувати роути як функції з іменем GET/POST/DELETE і тд.

Також серверна частина автоматично відправляє усі сторінки які знаходяться в каталозі проєкту. Наприклад, маємо сторінку /products (рис. 2.7):

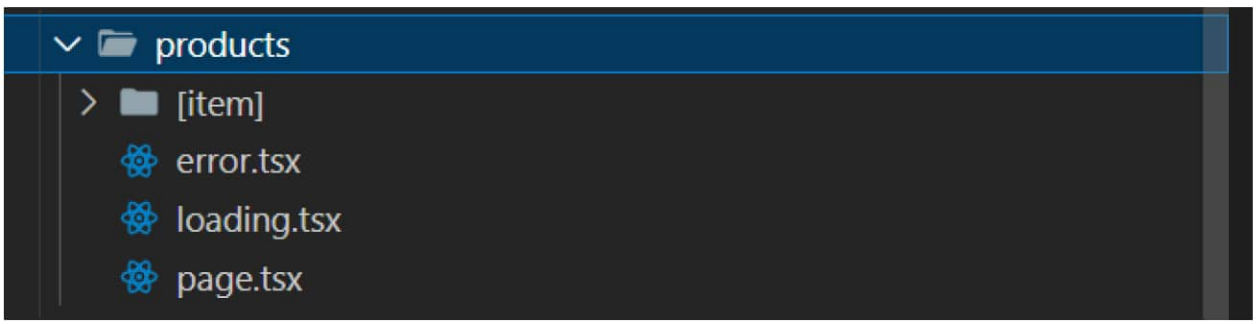


Рисунок 2.7 – приклад модулів на стороні клієнта

При переході на сторінки www.examplesite/products скомпілюється автоматично на стороні серверу файл **page.tsx** та скомпільована сторінка відправиться клієнту. Такий підхід значно підвищує SEO оптимізацію та загрузку сторінки. Також автоматично рендериться на клієнті компоненти loading.tsx при довгому завантаженні та error.tsx при помилці.

Папка /products/[slug] означає що ми чекаємо перехід по посиланню: www.examplesite/products/**slug** – тобто в кінці додається slug товару щоб отримати сторінку конкретного товару.

Важливо відмітити, що в одному каталозі може бути лише або route.ts, або page.tsx. Для розподілення клієнт-серверної архітектури.

Server Actions

Серверні actions – це звичайні функції які знаходяться в одній кодовій базі, але викликаються на стороні клієнта, передаючи певні параметри в функцію. Даний запит автоматично оброблюється на стороні серверу та повертається певна відповідь. Даний підхід спрощує відправку запитів з клієнта на сервер, так як виглядає це як звичайний виклик функцій.
наприклад (рис. 2.8):

Рисунок 2.8 – оголошені файли з server actions

```
'use server'

import { decode } from 'next-auth/jwt'
import { cookies } from 'next/headers'
import { NextRequest } from 'next/server'

export async function VerifyRequest(request?: NextRequest) {
  const cookie =
    cookies().get('next-auth.session-token') ??
    cookies().get('__Secure-next-auth.session-token')

  const decodedToken = await decode({
    secret: process.env.NEXTAUTH_SECRET!,
    token: cookie?.value,
  })

  return decodedToken
}
```

Рисунок 2.9 – приклад сервер функції

На першій строчці оголошено директиву “use server” (рис. 2.9) – це означає, що даний файл буде використовуватися лише на сервері та оголошений як Server Action. Його можна викликати як на сервері з інших роутів чи actions функцій, так і на стороні клієнта. Для використання необхідно лише імпортувати server action функцію та викликати її передавши необхідні параметри.

Party services

Сторонні сервіси у даному проєкті – **MongoDB** та **Cloudinary**.

MongoDB – дозволяє зберігати дані згідно їхніх схем, які були наведені раніше, модифікувати , видаляти та додавати їх.

Cloudinary – CDN сервіс для зберігання фото та відео. Був налаштований так, щоб автоматично групувати усі медіа файли по папкам згідно назви продукту, зберігав назви згідно назви продукту з порядковим номером. Схематичне представлення структури зберігання медіа в Cloudinary показано на рис. 2.10.

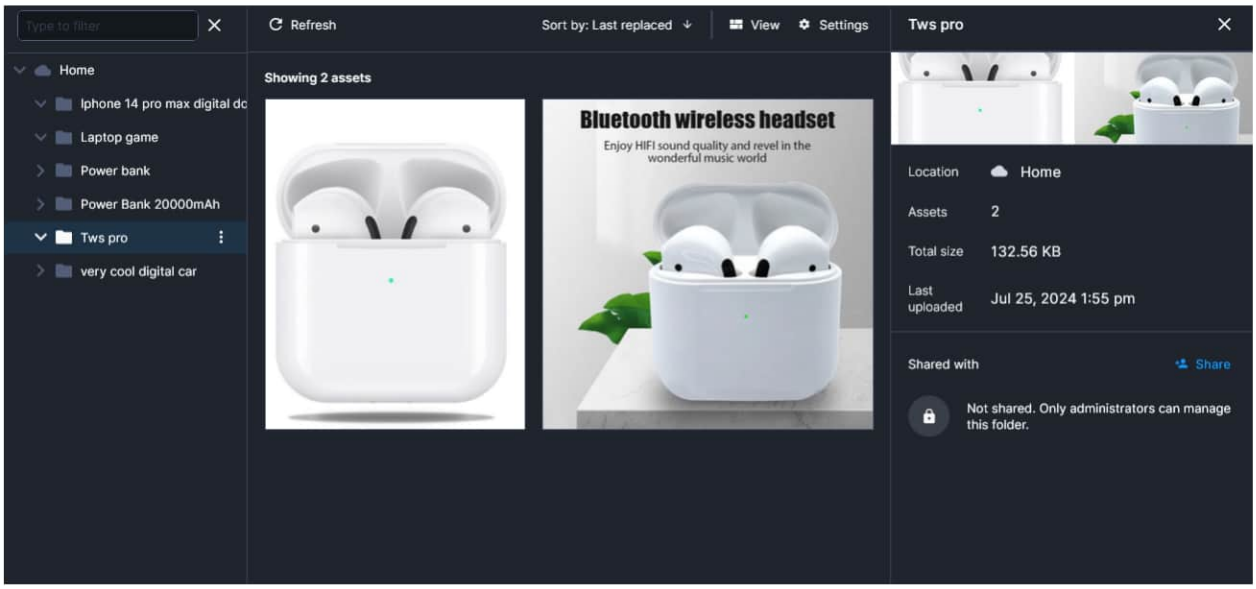


Рисунок 2.10 – формат зберігання медіа зображень

Cloudinary дозволяє в удобній формі переглядати, модифікувати, видаляти та додавати картки. Що робить роботу з медіа контентом дуже зручним. При створенні товару та збереженні в базу через адмін сторінку, автоматично додаються фотографії (якщо вони є в цей сервіс).

2.7.2 Клієнт частина

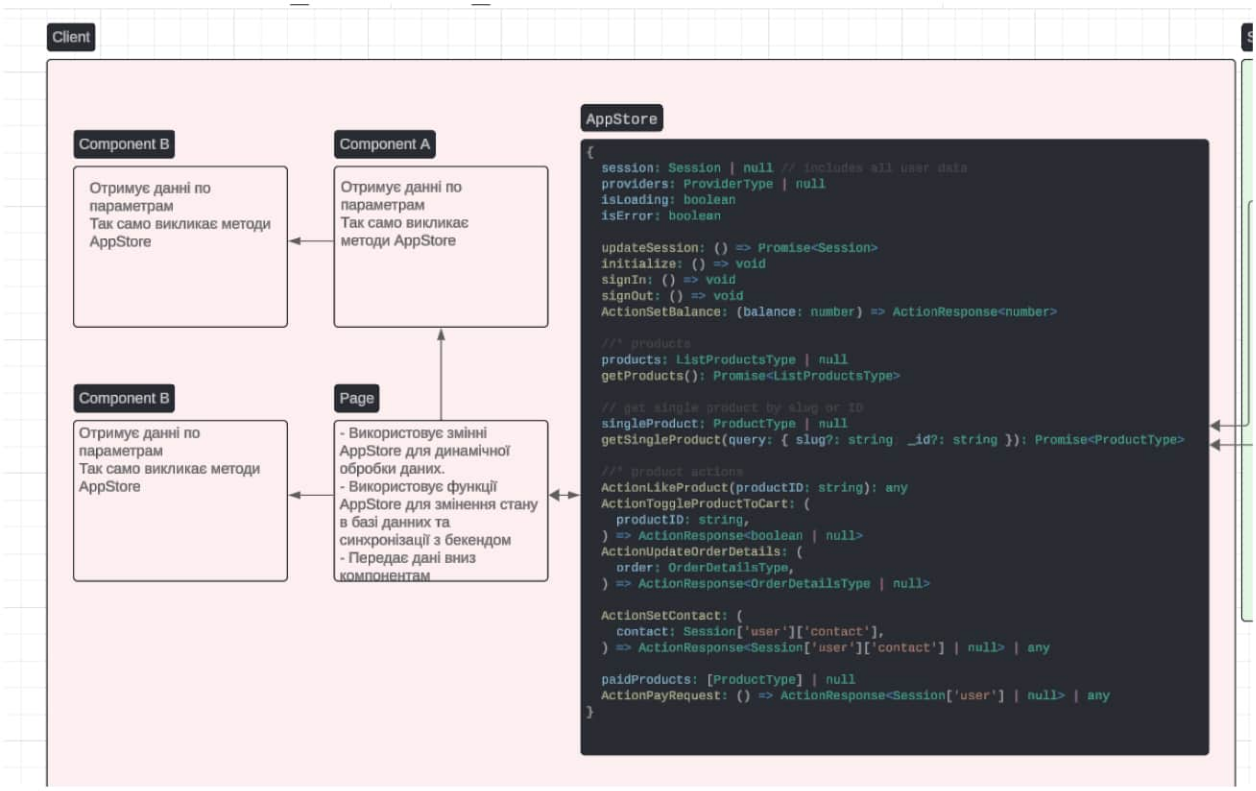


Рисунок 2.11 – модель клієнтської частини

AppStore

В якості менеджеру стану на стороні клієнта було використано бібліотеку **Zustand.** Цей модуль являє собою ключовою точкою між взаємодією між клієнтом та сервером на стороні клієнта. Усі запити на сервер повинні проходити тільки через AppStore модуль, що автоматично синхронізує стан між сервером та локальними змінними.

Його переваги:

- Глобальний стан даних на клієнті;
- Реактивне (динамічне) оновлення даних во всіх компонентах;
- Синхронізація з back-end сервером;

- Спрощення взаємодії між клієнтом та сервером;
- Перевикористовування коду;
- Робить кодову базу чистим та зрозумілим.

На рис. 2.11 зазначено усі змінні AppStore та методи. У форматі (параметри) => повернуті значення.

Page

Сторінки в цьому проєкті, як сказано раніше, генеруються по файлам **page.tsx.** Їхня головна мета – це синхронізація між appStore даних та передача цих даних іншим компонентам.

Сторінка може використовувати директиви “use client” та “use server” (за замовченням). Перша директива позначає, що сторінка буде компілюватися на стороні клієнта. А друга – на стороні сервера. Гарною практикою є рендер на стороні сервера верхнього блоку Page, а далі передача динамічних даних та елементів що потребують взаємодії з клієнтом на стороні клієнта.

Component

Реакт компоненти – це функції, що повертають JSX .

JSX – JavaScript syntax extention. Це свого роду частинка в html->body->…, що має дуже схожий синтаксис з html, але розширений можливістю додавати прямо в html JavaScript код. Це дозволяє в удобному та швидкому форматі рендерити масиви даних використовуючи **.map** оператор.

Приклад JSX:

```
    return <div className="grid grid-cols-2 gap-4 py-6 text-[13px] min-[800px]:grid-cols-3 lg:grid-cols-4">
      {filteredProducts?.map((product) => (
        <Product
          productState={product}
          likeAction={ActionLikeProduct}
          key={product._id}
        />
      ))}
      {!filteredProducts?.length && <h2>Product not found</h2>}
    </div>
  </section>
</div>
)
}
```

Рисунок 2.12 – компонент

В даному прикладі (рис. 2.12) функція повертає контейнер div з динамічно відмальованими картками продуктів. Використовуючи {js код тут} дужки можемо писати JavaScript код прямо в html документах.

Також бачимо на рисунку змінну **filteredProducts** – синхронізований стан бекенду та пошуку певних товарів. метод **.map** дозволяє використати колл-бек функцію в котрій аргумент – це данні про певний товар. Далі викликається ініціалізація наступного компонента **<Product/>** в якості аргумента передаємо данні з **product**.

У випадку, якщо список з продуктами пустий, пишемо повідомлення: “Product not found”.

2.7.3 Повна версія архітектурного шаблону проєкта

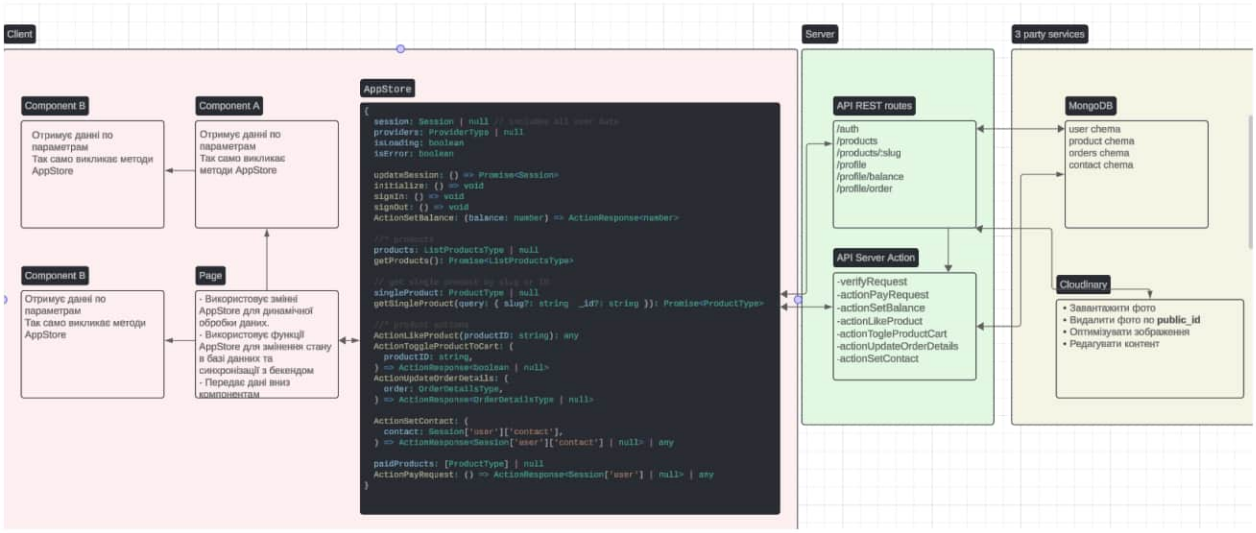


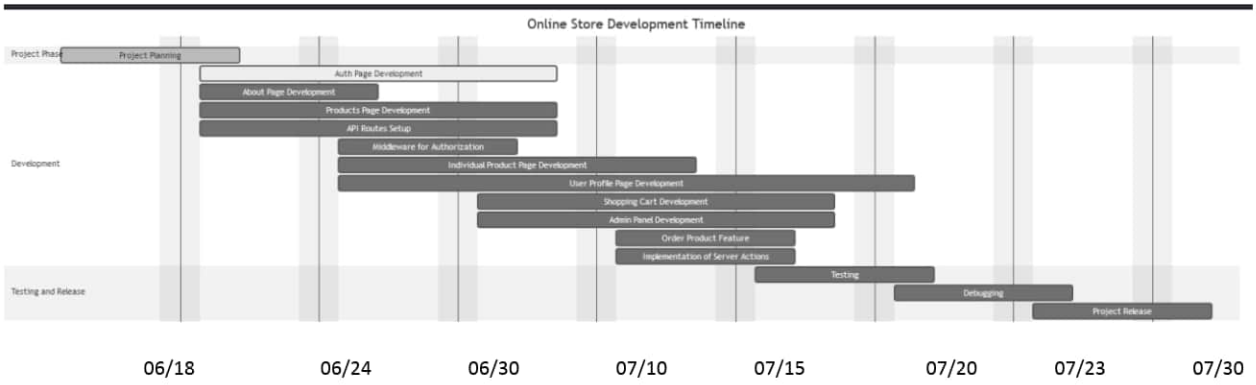
Рисунок 2.13 – загальна картина

На рис. 2.13 можемо побачити “шаблонну архітектуру” проєкту, по цьому принципу була побудована демо версія проєкту.

Дана модель дозволяє виробити патерн розробки в команді розробників та чітку структуру взаємодії між клієнтом та сервером.

2.8 Розробка діаграми Ганта

Розробили детальний план проєкту розробки інтернет-магазину, представлений у вигляді діаграми Ганта. Ця діаграма відображає завдання, їх тривалість, дати початку і завершення, а також залежності між завданнями.



- **Планування та моніторинг:** Діаграма Ганта допомагає планувати і стежити за виконанням проєкту, показуючи, коли повинні починатися і закінчуватися завдання.
- **Визначення залежностей:** Вона візуально показує залежності між завданнями, що дозволяє краще розуміти взаємозв'язки між різними етапами проєкту.
- **Управління часом:** Діаграма допомагає краще розподіляти час і ресурси, щоб уникнути затримок.
- **Комунікація:** Діаграма Ганта є корисним інструментом для комунікації всередині команди, забезпечуючи загальне розуміння прогресу проєкту.

2.9 Висновки до Розділу 2

У цьому розділі було здійснено порівняння популярних платформ для розробки інтернет-магазинів (SaaS, open source, кастомна розробка), що дозволило визначити ключові переваги індивідуального підходу з використанням сучасних фреймворків. На основі критеріїв гнучкості, масштабованості та можливості кастомізації обґрунтовано вибір Next.js як основного інструменту розробки. Також детально розглянуто архітектурні

рішення, структуру бази даних, побудову взаємодії між клієнтською і серверною частинами, а також інтеграцію з зовнішніми сервісами. Це сформувало чітке бачення архітектури майбутнього веб-додатку.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ПРОЄКТУ

3.1 Типи даних

Використовуючи переваги Next.js та TypeScript ми можемо задекларувати один формати даних, що будуть використовуватися як на фронтенді, так і серверній частині.

Задані типи або (Інтерфейси) обов’язково повинні відповідати полям в базі даних для уникнення конфліктів.

Також додали опціональні поля, що можуть бути або на серверній частині, або на стороні клієнта. Такі поля позначаються як “:?”.

3.1.1 стан користувача та авторизація

В даному проєкті увесь стан користувача зберігається в єдиному об’єкті: <**Session**> . Також в цьому файлі оголошені додаткові інтерфейси для схем:

OrderContact – схема контактів, OrderDetailsType – схема замовлення.

Об’єкт **Session** являє собою ключовим елементом, який повертається з бекенду при авторизації.

Також оголошені допоміжні об’єкт JWT для збереження id користувача в JWT токені.

```ts
// types/next-auth.d.ts

import { Session as NextAuthSession, User as NextAuthUser } from 'next-auth'
import { DefaultJWT, JWT } from 'next-auth/jwt'
import { ProductType } from 'product-types'

declare module 'next-auth' {
  type OrderContact = {
    fullName: string
    email: string
    address: string
    phone: string
  }

  type OrderDetailsType = {
    // order ID
    _id: string
    product: ProductType

    details: {
      contact: OrderContact
      quantity: number
      isPaid: boolean
    }
    // time snap
    createdAt: any
    updatedAt: any
  }

  interface Session extends NextAuthSession {
    user: {
      _id: string
      access_token?: JWT | string
      username?: string | null
      email?: string | null
      image?: string | null
      liked?: [string]

      balance: number | 0
      // array with products in cart
      cart: [OrderDetailsType]

      //
      order?: [OrderDetailsType] | []
```

```
      contact?: OrderContact
    }
    isAuth: boolean | undefined | null
  }

declare module 'next-auth/jwt' {
  interface JWT extends DefaultJWT {
    userID?: string
  }
}
```

3.1.2 Задекларовані типи продуктів

ProductType – ключовий тип , схеми Product chema, який включає в себе GalleryType (фотографії з Cloudinary).

Цей тип даних використовується практично усюди: в списку продуктів, в замовленні, в адмін панелі.

Також містить опціональні поля, що позначені “?”

```
declare module 'product-types' {
  type GalleryType = {
    href: string
    options?: UploadApiOptions
    public_id?: string
    file?: file
    _id?: string
  }

  type ProductType = {
    _id: string
    slug: string
    name: string
    price: number
```

```
    quantity: number

    // href with cloudinary imgs
    gallery: GalleryType[]
    description: string
    isLiked?: boolean
    isInCart?: boolean
    features: [
      {
        title: string
        value: string
      },
    ]
  }
  type ListProductsType = Array<ProductType>
}
```

3.2 Безпека програми

Даний проєкт містить публічні та приватні дані, тому для авторизації було використано Google Oauth, що дозволяє поєднати процес реєстрації, якщо немає гугл акаунту; якщо є, слугує для дуже зручного та швидкого входу через сервіс Google.

Інтерфейс сторінки авторизації подано на рис. 3.1.

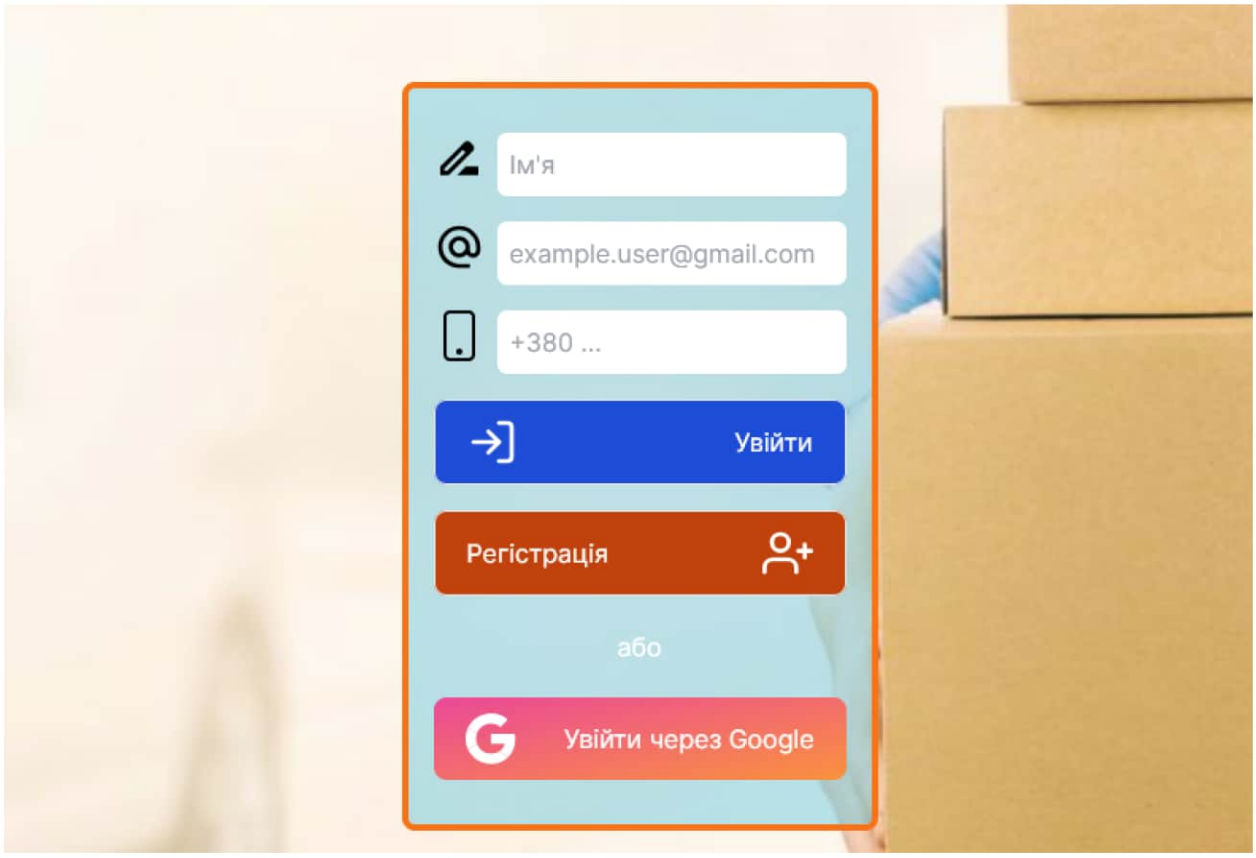


Рисунок 3.1 – сторінка /auth

При натисканні кнопки входу користувач потрапляє на сторінку вибору Google акаунту (рис. 3.2), що забезпечує швидкий та безпечний вхід до системи.

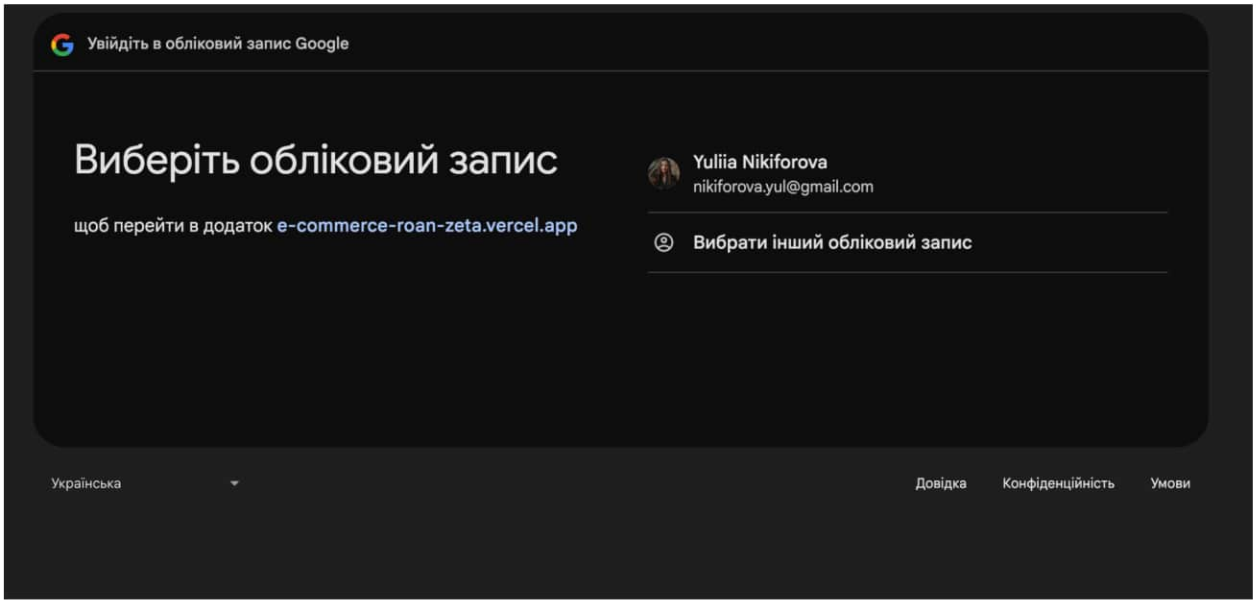


Рисунок 3.2 – вибір акаунту google

Після успішного входу користувач автоматично перенаправляється на профіль, як показано на рис. 3.3.

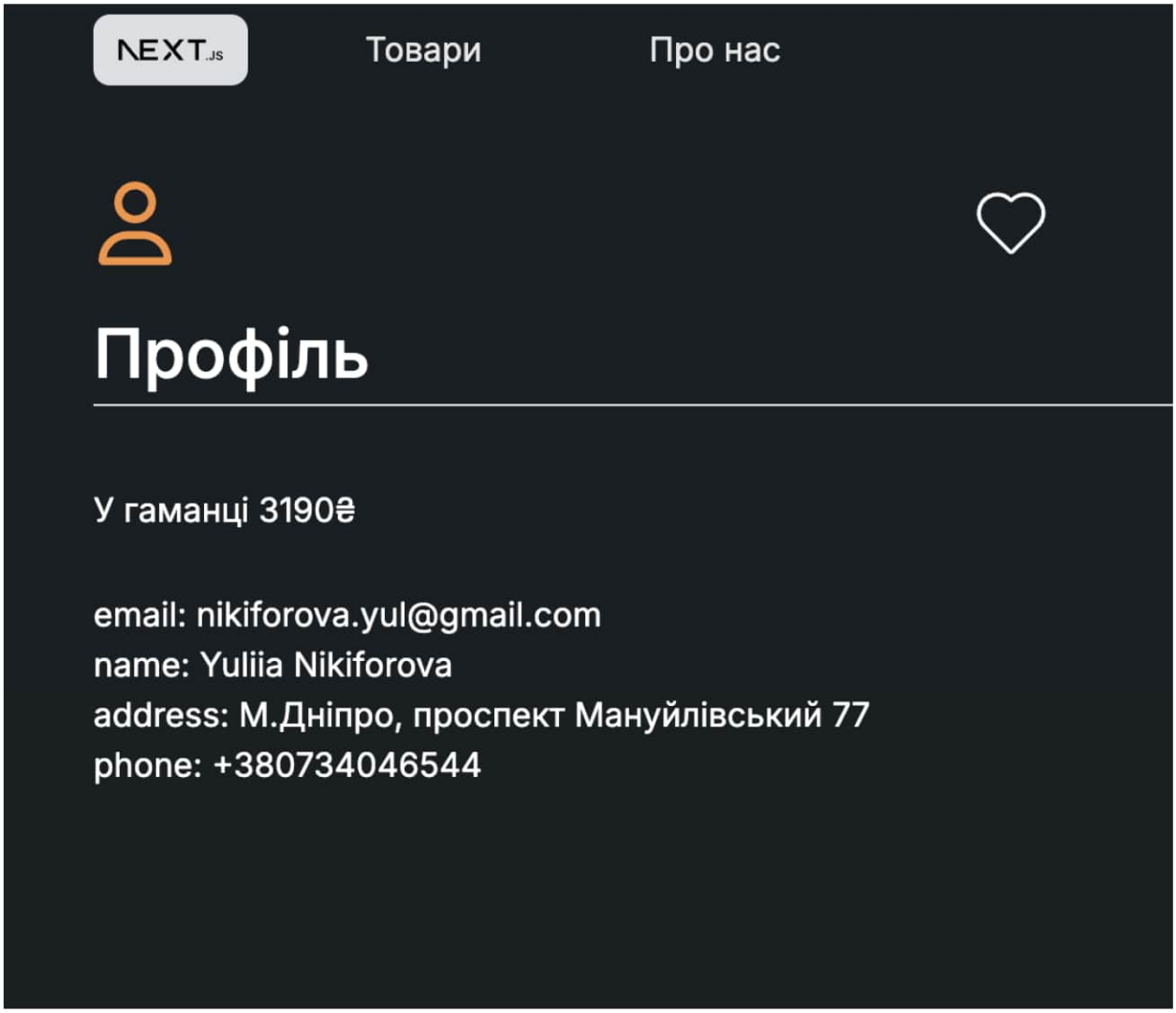


Рисунок 3.3 – переадресація при успішному вході на сторінку /profile

Ключовими файлами тут є /api/auth та middleware.

Авторизація відбувається завдяки бібліотеці https://next-auth.js.org/

Ісходний код роуту **/api/auth**:

```
// auth route with lib: nextauth

import GoogleProvider from 'next-auth/providers/google'
import NextAuth, { NextAuthOptions, Session } from 'next-auth'
```

```
import { connectDB } from '@/utils/server/database'
import UserModel from '@/models/user'

const options: NextAuthOptions = {
  providers: [
    GoogleProvider({
      clientId: process.env.GOOGLE_ID as string,
      clientSecret: process.env.GOOGLE_CLIENT_SECRET as string,
    }),
  ],
  secret: process.env.NEXTAUTH_SECRET,
  session: {
    strategy: 'jwt', // Use JWT for session
    maxAge: 60 * 60 * 24 * 30,
  },

  pages: { signIn: '/auth/' },
  events: {
    signOut() {},
    async signIn({ user }) {},
  },
  callbacks: {

    async session({ session, token, user }) {
      let userFromDatabase: Session | any
      // if (!session?.user?.email) return session
      try {
        await connectDB('ProductsDB')

        userFromDatabase = await UserModel.findOne({
          email: session.user?.email,
        })
          .populate(['cart.product', 'order.product'])
          .maxTimeMS(5000)
          .exec()
      } catch (err) {
        console.log('err on findOne user:\n', err)
        console.log('returned session:', { session })

        return session
      }

      if (session.user && userFromDatabase) {
        session.user = userFromDatabase._doc
        session.isAuth = true
      } else {
```

```
        session.isAuth = false
      }

      return session
    },

    async signIn({ profile }) {
      try {
        await connectDB('ProductsDB')
      } catch (err) {
        console.error('Err in connectDB:' + err)
        return false
      }

      const isUserExists = await UserModel.findOne({
        email: profile?.email,
      }).maxTimeMS(1000)

      if (!isUserExists) {
        await UserModel.create({
          email: profile?.email,
          username: profile?.name,
          image: profile?.image,
        })
      }

      // Return true to allow sign in
      return true
    },
  },
}

const handler = NextAuth(options)

export { handler as GET, handler as POST }
```

Даний блок коду створює роут на авторизацію через гугл. Спочатку він перевіряє чи існує користувач в базі даних, якщо ні, то створює його, якщо так, то повертає його.

Бібліотека next-auth автоматично створює зашифрований JWT токен, що містить данні про користувача.

Також JWT-токен автоматично додається до cookies клієнта і додається при кожному http запиті на сервер.

При кожному запиті по захищеному роуті перевіряється цей JWT-токен та оброблюються помилки при його не валідності чи відсутності.

Щоб розшифрувати JWT-токен потрібен лише ключ, котрий зберігається лише на стороні серверу в **.env** файлі та не повинен міститися у відкритому доступі.

Middleware

Middleware – це проміжуточна функція, котра перехоплює кожен запит користувача та додає кастомну логіку, наприклад перевірку авторизованості користувача.

Тому більше не має потреби окремо захищати кожен приватний роут, достатньо додати його шлях в об’єкт **matcher,** та він буде автоматично захищеним.

```
import { NextRequest, NextResponse } from 'next/server'
import { VerifyRequest } from './app/actions/verify'

export const config = {
  matcher: [
    '/',
    '/products',
    '/profile',
    '/profile/(.*)',
```

```
    '/admin',
    '/admin/(.*)',
    '/api/',
    '/api/products',
    '/api/products/(.*)',
  ],
}
const protectedRoutes = [
  '/profile',
  '/profile/balance',
  '/profile/order',
  '/admin',
  '/admin/edit',
]

function isProtectedRoute(pathname: string) {
  // nextUrl.pathname
  return protectedRoutes.some((route) => route === pathname)
}

export async function middleware(request: NextRequest) {
  const { method, url, nextUrl } = request
  if (method !== 'GET' || isProtectedRoute(nextUrl.pathname)) {
    try {
      const decodedToken = await VerifyRequest()

      if (!decodedToken) return redirectResponse(url, 'unauth')
    } catch (err) {
      return redirectResponse(url, 'Token verification failed:' + err)
    }
  }
  return NextResponse.next()
}

export function redirectResponse(url: string, log?: any) {
  console.log(log)
  return NextResponse.redirect(new URL('/auth/', url), {
    status: 302,
    statusText: log,
  })
}
```

Верифікація запиту відбувається завдяки Server Action – VerifyRequest функції , про яку було зазначено в Розділі 2.

Перевірка на веріфікацію проходить автоматично для всіх запитів, що не є GET, та для всіх захищених роутів зазначених в **protectedRoutes**.

3.3 Робота на стороні клієнта

Як зазначено в Розділі 2, “шаблонний архітектурний патерн”, маємо сторінку, яка взаємодіє з глобальним станом AppStore. Та передає отримані дані вниз по дереву компонентів.

3.3.1 Приклад реалізації сторінки:

Спочатку імпортуємо необхідні компоненти: реакт хуки, та типи:

```
'use client'
import Product from '@/components/Product'
import { Pagination, Navigation } from 'swiper/modules'
import { Swiper, SwiperSlide } from 'swiper/react'
import { AppStore } from '@/store/appStore'

import 'swiper/css'
import 'swiper/css/pagination'
import 'swiper/css/navigation'
import { useEffect, useState } from 'react'
import { SearchInput } from '@/components/SearchInput'
```

Далі екпортуємо функцію за замовченням та в верхній частині функції пишемо головну логіку при ініціалізації компоненти, а знизу повернутий JSX компонент:

```
export default function ProductsPage() {
  // деструктуризація об'єктів з глобального стану
  const { products, getProducts, ActionLikeProduct } = AppStore()
  // окремий стан для фільтрування продуктів
  const [filteredProducts, setFilteredProducts] = useState(products)
  // useEffect викликає колл-бек функцію, коли сторінка перщий раз завантажується
  // та коли змінюється масив залежностей [...depth]
  useEffect(() => {
    const update = async () => {
      const products = await getProducts()
      setFilteredProducts(products)
    }
    update()
  }, [getProducts])
  // оброблюємо вивід повідомлення залежно від стану
  if (products === null) return <div>Loading...</div>
  if (!products?.length) return <h2>no products</h2>
```

Коли данні перешли перевірку, повертаємо повну сторінку з товарами:

```
  return (
    <div className="">
      <section>
        <h2 className="pb-8">
          <b>Аксесуари</b> на будь-який смак
        </h2> // etc
```

3.3.2 Використання компонентів

Як сторінки, так і компоненти технічно являються React Components, але було розділено їх так, щоб була видна ієрархічної система.

Якщо сторнінка синхронізує дані з сервером та передає їх компонентам, то компоненти отримують ці діні та вимальовують їх, тим самим вони

містять свою логіку, свій JSX та скривають велику складність під собою (Інкапсуляція), що робить їх дуже простими у використанні.

Нижче наведено приклад компоненту картки товару:

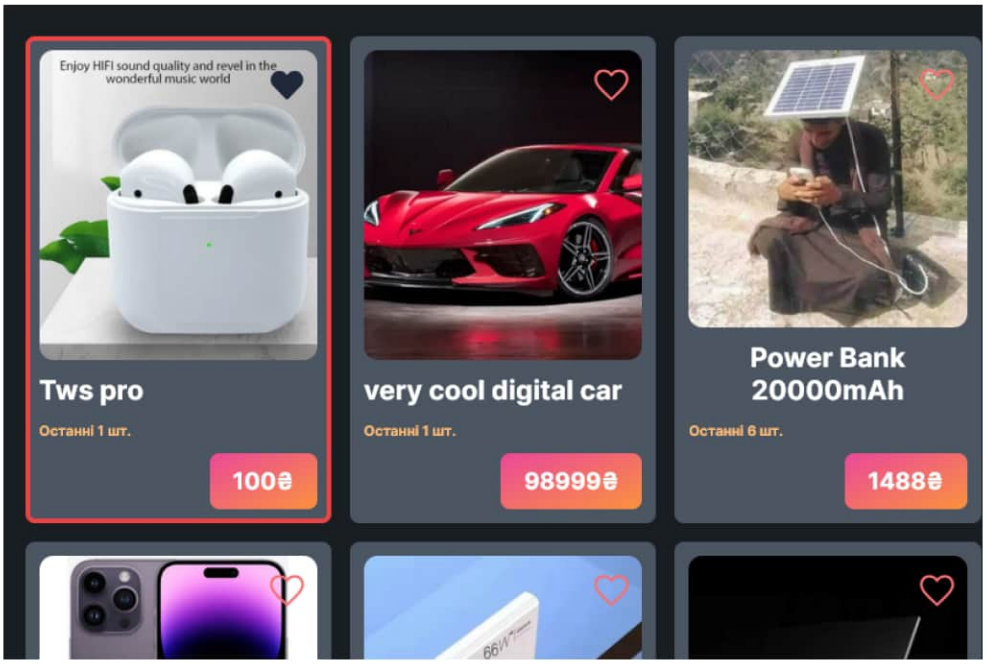


Рисунок 3.4 – список з картками /products

```
'use client'
import { ProductType } from 'product-types'
import Link from 'next/link'
import { AppStoreType } from '@/store/appStore'
import LikeButton from './LikeButton'
import { CldImage } from 'next-cloudinary'

type ProductProps = {
  productState: ProductType
  likeAction: AppStoreType['ActionLikeProduct']
}

export default function Product({ productState, likeAction }: ProductProps) {
  const { _id, name, price, slug, isLiked, quantity, gallery } = productState
```

```
  return (
    <div // далі великий шматок JSX
```

З даного уривку кода можемо побачити просту передачу по параметрам даних, та подальшу деструктуризацію об’єктів через оператор {} .

На сторінкі /products отримали з бекенду **products** та рендеримо його через .map метод JSX компоненти. Таким чином, із масива з типом даних **ProductType** ми створили UI елементи на рис. 3.4.

```
          {products?.map((product) => (
            <SwiperSlide
              key={product.name}
              className={`min-h-full md:max-w-[350px]`}
            >
              <Product
                productState={{ ...product }}
                likeAction={ActionLikeProduct}
              />
            </SwiperSlide>
          ))}
```

3.4 Використання REST API routes

Як було сказано в Розділі 2, в Next.js відбувається автоматичне створення роутів для файлів з назвою route.ts. Приклад роуту для отримання списку з продуктами:

```
// Marking the route as dynamic
export const dynamic = 'force-dynamic'

import { VerifyRequest } from '@/app/actions/verify'
import Product from '@/models/product'
import User from '@/models/user'
import { connectDB } from '@/utils/server/database'
import { mapPersonalProducts } from '@/utils/server/mapUserProducts'
import { Session } from 'next-auth'
```

```
import { NextRequest, NextResponse } from 'next/server'
import { GalleryType, ProductType } from 'product-types'

await connectDB('ProductsDB')

// /products GET
export async function GET(request: NextRequest) {
  const jwt = await VerifyRequest()
  let user: Session['user'] | null, products: ProductType[]

  try {
    products = await Product.find().lean()
    user = jwt?.email ? await User.findOne({ email: jwt?.email }).lean() : null
    if (user) products = mapPersonalProducts(products, user)

    return new Response(JSON.stringify(products))
  } catch (error) {
    console.error('Error fetching products:', error)
    return new Response('Error fetching products', { status: 500 })
  }
}
```

Даний route виконує наступні дії:

- Отримує дані з JWT токену;
- Якщо не валідні, то повертає загальний публічний список;
- Якщо валідні, то повертає список з продуктами згідно стану користувача.

Тобто, відображає усі лайкнуті товари, усі товари що в корзині і тд.

Функція

```
mapPersonalProducts(products, user)
```

це кастомна утилітна функція, що міняє список з продуктами, згідно дій користувача.

```
import { Session } from 'next-auth'
import { ProductType } from 'product-types'

//! helper function
// map for user his products
export const mapPersonalProducts = (
  products: ProductType[],
  user?: Session['user'],
) => {
  if (!user) return products

  if (products.length === 0) return []

  const { liked, cart } = user
  const cartIDs =
    cart?.map(({ product }) => product._id.toString()) ?? []

  const mapped: ProductType[] = [...products].map((product) => ({
    ...product,
    isLiked: liked?.includes(product._id.toString()), // Ensure the product ID is
a string
    isInCart: cartIDs?.includes(product._id.toString()),
  }))
  return mapped
}
```

Завдяки цій функції по запиту GET /products авторизовані користувачі одразу отримають персональні продукти та інтерактивні дії (товари лайкнуті та в корзині).

3.5 Використання AppStore модуля

AppStore містить як динамічні змінні, так і методи для відправки на сервер. Вона спрощує використання API з бекендом.

Для змінення інтерфейсу AppStore достатньо змінити оголошення типу AppStore:

```
export type AppStoreType = {
  session: Session | null // includes all user data
  providers: ProviderType | null
  isLoading: boolean
```

```
  isError: boolean
  //* products
  products: ListProductsType | null
  getProducts(): Promise<ListProductsType> ... інші типи
```

У даному прикладі маємо глобальну зміну products та метод getProducts.

Об’єкт create <AppStoreType> – це Zustand стан типу **AppStoreType**. Цей тип зобов’язує заповнити по замовченню значення змінних залежно від їх типу, та написати методи, які відповідають їхнім типам в AppStore.

Використовуючи метод set – ми можемо змінити поля стану AppStore.
Використовуючи метод get – ми можемо отримати змінні чи методи.

```
export const AppStore = create<AppStoreType>((set, get) => ({
products: null,
  async getProducts() {
                // змінюємо стан на завантаження
                set({ isLoading: true })
                // відправляємо запит на сервер
                const products = await fetcher('/api/products')
                // беремо метод updateSession
                const { updateSession } = get()
                //викликаємо його
                await updateSession()
                // змінну products міняємо на отриманий список з продуктами
                set({ products, isLoading: false })

                return products
              },
 ... інший код
}
```

Таким чином, достатньо викликати метод getProducts та отримаємо свіжу версію продуктів з сервера, та автоматично в реальному часі отримаємо стан завантаження.

ВИСНОВКИ

Отже, у ході виконання проєктно-технологічної роботи продемонстрували, як сучасні технології можуть бути ефективно використані для створення функціонального та надійного інтернет-магазину. Застосування Next.js, TypeScript та MongoDB дозволило досягти високої продуктивності та зручності для користувачів, а також забезпечити гнучкість і масштабованість системи. Розроблений інтернет-магазин все має деякі технічні обмеження, але як демо версія готовий до використання та може бути легко розширена у майбутньому для задоволення нових потреб і вимог бізнесу.

Функціональні можливості

- Реалізована **сторінка авторизації через Google OAuth** забезпечила зручність і безпеку доступу користувачів до системи.
- **Сторінка профіля користувача** дозволила зібрати всі необхідні дані про клієнтів, включаючи список лайкнутих товарів, що підвищує зручність користувачів, корзина та замовлення користувача.
- **Список замовлень** відображає детальну інформацію про кожне замовлення, забезпечуючи користувачам повний контроль над своїми покупками.
- **Корзина користувача** з можливістю редагування кількості товарів, видалення та введення контактних даних дозволяє легко оформлювати замовлення.
- **Публічна сторінка з товарами** з можливістю пошуку та лайків підвищує інтерактивність та зручність користувачів.

- **Сторінка певного товару** з детальною інформацією, описом, ціною та можливістю додавання в корзину дозволяє користувачам отримувати всю необхідну інформацію про продукти.
- **Адаптивний дизайн** був реалізований за допомогою TailwindCss, що дозволило написати за методологією mobile-first застосунок , також із підтримкою планшет та комп'ютерних дисплеїв.

Адміністрування

Реалізована **адмін панель** дозволяє ефективно керувати продуктами, виконувати CRUD операції, а також інтеграцію з сервісом Cloudinary для зберігання зображень.

Безпека

Використання **JWT-токенів** для аутентифікації забезпечило високий рівень безпеки користувацьких даних і захист від несанкціонованого доступу.

Процес Розробки

- Проєкт був розроблений відповідно до графіку, використовуючи **діаграму Ганта** для планування та моніторингу прогресу. Це дозволило ефективно координувати роботу команди і вчасно завершити всі етапи проєкту.
- Тестування і налагодження були проведені для забезпечення стабільної і безпомилкової роботи системи.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Звіт про стан електронної комерції в Україні. [Електронний ресурс]. URL: https://example.com/ecommerce-report (дата звернення 17.04.2025).
2. Тенденції розвитку інтернет-магазинів. [Електронний ресурс]. URL: https://example.com/ecommerce-trends (дата звернення 17.04.2025).
3. Next.js Documentation. [Електронний ресурс]. URL: https://nextjs.org/ (дата звернення 20.04.2025).
4. Світова статистика електронної комерції. [Електронний ресурс]. URL: https://example.com/global-ecommerce-stats (дата звернення 17.04.2025).
5. Порівняння SaaS платформ для e-commerce. [Електронний ресурс]. URL: https://example.com/saas-comparison (дата звернення 17.04.2025).
6. Огляд Open Source рішень для інтернет-магазинів. [Електронний ресурс]. URL: https://example.com/opensource-ecommerce (дата звернення 25.04.2025).
7. Переваги індивідуальної розробки інтернет-магазину. [Електронний ресурс]. URL: https://example.com/custom-ecommerce (дата звернення 17.04.2025).
8. Прогресивні веб-додатки (PWA). [Електронний ресурс]. URL: https://example.com/pwa-info (дата звернення 20.04.2025).
9. Архітектура мікросервісів. [Електронний ресурс]. URL: https://example.com/microservices-architecture (дата звернення 30.04.2025).
10. Що таке JAMstack. [Електронний ресурс]. URL: https://example.com/jamstack-explained (дата звернення 11.05.2025).
11. Headless Commerce: переваги та недоліки. [Електронний ресурс]. URL: https://example.com/headless-commerce (дата звернення 11.05.2025).
12. Роль API в сучасній розробці. [Електронний ресурс]. URL:

https://example.com/api-development (дата звернення 11.05.2025).

13. MongoDB Documentation. [Електронний ресурс]. URL: https://www.mongodb.com/products/platform/atlas-database (дата звернення 11.05.2025).

14. Tailwind CSS Documentation. [Електронний ресурс]. URL: https://tailwindcss.com/ (дата звернення 11.05.2025).

15. Cloudinary Documentation. [Електронний ресурс]. URL: https://cloudinary.com/ (дата звернення 25.05.2025).

16. Next-auth Documentation. [Електронний ресурс]. URL: https://next-auth.js.org/ (дата звернення 29.05.2025).

17. Zustand Documentation. [Електронний ресурс]. URL: https://zustand.pmnd.rs/ (дата звернення 29.05.2025).

ДОДАТОК А

Лістинг файлу package.json

```
{
     "name": "e-commerse",
     "version": "0.1.0",
     "private": true,
     "scripts": {
          "dev": "next dev -p 4555",
          "build": "next build",
          "start": "next start",
          "lint": "next lint"
     },
     "dependencies": {
          "@types/mongoose": "^5.11.97",
          "autoprefixer": "^10.4.19",
          "bcrypt": "^5.1.1",
          "cloudinary": "^2.3.0",
          "cookie": "^0.6.0",
          "cookies-next": "^4.2.1",
          "cors": "^2.8.5",
          "fs": "^0.0.1-security",
          "google-auth-library": "^9.11.0",
          "jose": "^5.6.2",
          "jsonwebtoken": "^9.0.2",
          "mongodb": "^6.7.0",
          "mongoose": "^8.4.4",
          "next": "14.2.4",
          "next-auth": "^4.24.7",
          "next-cloudinary": "^6.6.2",
          "next-cors": "^1.0.0",
          "nextjs-cors": "^2.2.0",
          "postcss-import": "^16.1.0",
          "react": "^18",
```

```
            "react-dom": "^18",
            "react-icons": "^5.2.1",
            "react-toastify": "^10.0.5",
            "swiper": "^11.1.4",
            "swr": "^2.2.5",
            "universal-cookie": "^7.1.4",
            "uuid": "^10.0.0",
            "zustand": "^4.5.4"
      },
      "devDependencies": {
            "@types/cors": "^2.8.17",
            "@types/jsonwebtoken": "^9.0.6",
            "@types/next-auth": "^3.15.0",
            "@types/node": "^20",
            "@types/react": "^18",
            "@types/react-dom": "^18",
            "@types/uuid": "^10.0.0",
            "eslint": "^8",
            "eslint-config-next": "14.2.4",
            "postcss": "^8",
            "prettier": "^3.3.2",
            "prettier-plugin-tailwindcss": "^0.6.5",
            "tailwindcss": "^3.4.4",
            "typescript": "^5"
      }
}
```

ДОДАТОК Б

Лістинг файлу tsconfig.json

```
{
     "compilerOptions": {
    "target": "es2017",
          "lib": ["dom", "dom.iterable", "esnext"],
          "allowJs": true,
          "skipLibCheck": true,
          "strict": true,
          "noEmit": true,
          "esModuleInterop": true,
          "module": "esnext",
          "moduleResolution": "bundler",
          "resolveJsonModule": true,
          "isolatedModules": true,
          "jsx": "preserve",
          "incremental": true,
          "plugins": [
               {
                    "name": "next"
               }
          ],
          "paths": {
               "@/*": ["./*"]
          },
          "typeRoots": ["./types", "./node_modules/@types"]
     },
     "include":     ["next-env.d.ts",     "**/*.ts",     "**/*.tsx",
".next/types/**/*.ts"],
     "exclude": ["node_modules"]
}
```

ДОДАТОК В

Лістинг файлу middleware.ts

```
import { NextRequest, NextResponse } from 'next/server'
import { VerifyRequest } from './app/actions/verify'

export const config = {
  matcher: [
    '/',
    '/products',
    '/profile',
    '/profile/(.*)',
    '/admin',
    '/admin/(.*)',
    '/api/',
    '/api/products',
    '/api/products/(.*)',
  ],
}
const protectedRoutes = [
  '/profile',
  '/profile/balance',
  '/profile/order',
  '/admin',
  '/admin/edit',
]

function isProtectedRoute(pathname: string) {
  // nextUrl.pathname
  return protectedRoutes.some((route) => route === pathname)
}

export async function middleware(request: NextRequest) {
```

```
  const { method, url, nextUrl } = request

  if (method !== 'GET' || isProtectedRoute(nextUrl.pathname)) {
    try {
      const decodedToken = await VerifyRequest()

      if (!decodedToken) return redirectResponse(url, 'unauth')
    } catch (err) {
        return redirectResponse(url, 'Token verification failed:' +
err)
    }
  }
  return NextResponse.next()
}

export function redirectResponse(url: string, log?: any) {
  console.log(log)
  return NextResponse.redirect(new URL('/auth/', url), {
    status: 302,
    statusText: log,
  })
}
```

ДОДАТОК Г

Лістинг файлу next.config.mjs

```
/** @type {import('next').NextConfig} */
const nextConfig = {
  experimental: {
    serverComponentsExternalPackages: ['mongoose'],
  },
  images: {
    domains: ['lh3.googleusercontent.com'],
  },
  webpack(config) {
    config.experiments = {
      ...config.experiments,
      topLevelAwait: true,
    }
    return config
  },
  reactStrictMode: false,
  async headers() {
    return [
      {
        // Apply these headers to all routes in your application.
        source: '/(.*)',
        headers: [
          { key: 'Access-Control-Allow-Credentials', value: 'true' },
                      {  key:  'Access-Control-Allow-Origin',  value:
'http://localdev.com' },
          {
            key: 'Access-Control-Allow-Methods',
            value: 'GET,OPTIONS,PATCH,DELETE,POST,PUT',
          },
          {
```

```
            key: 'Access-Control-Allow-Headers',
            value:
                          'X-CSRF-Token, X-Requested-With, Accept,
Accept-Version, Content-Length, Content-MD5, Content-Type, Date,
X-Api-Version',
          },
        ],
      },
    ]
  },
}

export default nextConfig
```

ДОДАТОК Д

Лістинг типів (/types/next-auth.d.ts, /types/product-types.d.ts)

```
import { Session as NextAuthSession, User as NextAuthUser } from
'next-auth'
import { DefaultJWT, JWT } from 'next-auth/jwt'
import { ProductType } from 'product-types'

declare module 'next-auth' {
  type OrderContact = {
    fullName: string
    email: string
    address: string
    phone: string
  }

  type OrderDetailsType = {
    _id: string
    product: ProductType

    details: {
      contact: OrderContact
      quantity: number
      isPaid: boolean
    }
    createdAt: any
    updatedAt: any
  }

  interface Session extends NextAuthSession {
    user: {
      _id: string
      access_token?: JWT | string
```

```
      username?: string | null
      email?: string | null
      image?: string | null
      liked?: [string]

      balance: number | 0
      cart: [OrderDetailsType]

      order?: [OrderDetailsType] | []
      contact?: OrderContact
    }
    isAuth: boolean | undefined | null
  }

  interface User extends NextAuthUser {
    id: string
  }
}

declare module 'next-auth/jwt' {
  interface JWT extends DefaultJWT {
    userID?: string
  }
}
```

```
declare module 'product-types' {
  type GalleryType = {
    href: string
    options?: UploadApiOptions
    public_id?: string
    file?: file
    _id?: string
  }
```

```
  type ProductType = {
    _id: string
    slug: string
    name: string
    price: number
    quantity: number

    gallery: GalleryType[]
    description: string
    isLiked?: boolean
    isInCart?: boolean
    features: [
      {
        title: string
        value: string
      },
    ]
  }
  type ListProductsType = Array<ProductType>
}
```

ДОДАТОК Е

Лістинг моделей (/models/orders.ts, /models/product.ts, models/user.ts)

```
import mongoose, { models, model, Schema } from 'mongoose'

export const ContactChema = new Schema({
  fullName: { type: String, required: true },
  email: { type: String, required: true },
  phone: { type: String, required: true },
  address: { type: String, required: true },
})

// Define the schema for the product
export const OrderSchema = new Schema(
  {
    product: {
      type: Schema.Types.ObjectId,
      ref: 'Product',
      required: true,
    },
    details: {
      contact: { type: ContactChema },
      quantity: { type: Number, required: true, default: 1 },
      isPaid: { type: Boolean, required: true, default: false },
    },
  },

  { timestamps: true },
)

// Create the model based on the schema
const Orders = models.Orders || model('Orders', OrderSchema)
```

```
export default Orders
```

```
import { ProductType } from 'product-types'
import mongoose, { models, model } from 'mongoose'

// Extend the Mongoose Document interface with your custom fields
export interface ProductDocument extends ProductType, Document {}
// Define the schema for the product
const productSchema = new mongoose.Schema<ProductDocument>({
  name: { type: String, required: true },
  price: { type: Number, required: true },
  quantity: { type: Number, required: true, min: 0 },
  slug: { type: String, required: true, unique: true },
  description: { type: String, required: true, default: 'Опис' },

  gallery: {
    type: [
      {
        href: String,
        public_id: String,
      },
    ],
    default: [],
  },

  features: [
    {
      title: String,
      value: String,
    },
  ],
})
```

```
// Create the model based on the schema
const Product =
  models.Product || model<ProductDocument>('Product', productSchema)

export default Product
```

```
import { Schema, model, models } from 'mongoose'
import { ContactChema, OrderSchema } from './orders'

const UserSchema = new Schema({
  email: {
    type: String,
    unique: [true, 'Email already exists!'],
    required: [true, 'Email is required!'],
  },
  username: {
    type: String,
    required: [true, 'Username is required!'],
  },
  image: { type: String },
  liked: [{ type: String, default: [] }],

  cart: [OrderSchema],
  order: [OrderSchema],
  contact: ContactChema,

  balance: { type: Number, default: 0 },
})

const User = models.User || model('User', UserSchema)

export default User
```

ДОДАТОК Є

Лістинг utils/server папки

database.ts file

```
'use server'

import mongoose from 'mongoose'

export const connectDB = async (DBname: 'ProductsDB') => {
  if (mongoose.connection.readyState === 1) return

  mongoose.set('strictQuery', true)

  try {
    await mongoose.connect(process.env.MONGODB_URI as string, {
      dbName: DBname,
    })
    console.log(`✅🚀 ${DBname} initial connect ✅🚀`)
  } catch (err) {
    console.log(`\n${err}`)
  }
}
```

errResponse.ts

```
export const errorResponse = ({
  state,
  message,
  redirectedURL,
  success,
}: {
  state?: any | null
```

```
  message?: string
  redirectedURL?: string
  success?: boolean
}) => {
  console.log({ message, redirectedURL })
  return {
    success: success ?? false,
    state: state,
    message,
    redirectedURL,
  }
}
```

mapUserProducts.ts

```
import { Session } from 'next-auth'
import { ProductType } from 'product-types'

//! helper function
// map for user his products
export const mapPersonalProducts = (
  products: ProductType[],
  user?: Session['user'],
) => {
  if (!user) return products

  if (products.length === 0) return []

  const { liked, cart } = user
  const cartIDs =
    cart?.map(({ product }) => product._id.toString()) ?? []

  const mapped: ProductType[] = [...products].map((product) => ({
    ...product,
```

```
     isLiked: liked?.includes(product._id.toString()), // Ensure the
product ID is a string
    isInCart: cartIDs?.includes(product._id.toString()),
  }))
  return mapped
}
```

ДОДАТОК Ж

Лістинг actions обробників

/actions/orderActions.ts

```
'use server'
import { connectDB } from '@/utils/server/database'
import { ActionResponse } from './actionTypes'
import User from '@/models/user'
import { errorResponse } from '@/utils/server/errResponse'
import { OrderDetailsType, Session } from 'next-auth'
import { VerifyRequest } from './verify'
import { ProductType } from 'product-types'

type actionSetBalanceProps = {
  userID: string
  balance: number
}

export async function actionSetBalance({
  userID,
  balance,
}: actionSetBalanceProps): ActionResponse<number> {
  try {
    if (!userID || !balance) throw new Error('Bad request')

    await connectDB('ProductsDB')
    let user = await User.findOne({ _id: userID })
    user.balance += balance
    await user.save()

    return { state: balance, success: true }
  } catch (error) {
```

```
    console.log({ error })

    return { state: 0, success: false, message: error + '' }
  }
}

export async function actionPayRequest(): ActionResponse<
  Session['user'] | null
> {
  try {
    const dec = await VerifyRequest()

    if (!dec || !dec.email) throw new Error('Unauth req')

    await connectDB('ProductsDB')

    const user = await User.findOne({ email: dec.email })
      .populate(['cart.product', 'order.product'])
      .exec()

    if (!user) throw new Error('user not found')

    let finalPrice = 0

    const newProductsDB: ProductType[] = []
    const newOrderDB: OrderDetailsType[] = user?.order ?? []

    // validate each order
    user.cart = user.cart?.filter((order: OrderDetailsType) => {
      let { details, product } = order

      if (details.quantity === 0) return true

       if (details.quantity > product.quantity || details.quantity <=
```

```
0)
        throw Error('Error: invalid order quantity')

      finalPrice += details.quantity * product.price

      product.quantity -= details.quantity

      details.isPaid = true
      details.contact = user?.contact

      newOrderDB.push(order)

      newProductsDB.push(product)

      return false
    })
    if (finalPrice > user.balance) {
      return errorResponse({
        message: `not enough money. To pay: ${finalPrice} On balance:
${user.balance} `,
      })
    }
    user.balance -= finalPrice
    user.order = newOrderDB

    await user.save()
    newProductsDB.forEach(async (productDB: any) => {
      await productDB.save()
    })

    return { state: null, success: true }
  } catch (error) {
    return errorResponse({ message: error + '', state: null })
  }
```

```
}
```

/actions/productActions.ts

```
'use server'

import User from '@/models/user'
import { connectDB } from '@/utils/server/database'

import { ActionResponse } from './actionTypes'
import { OrderDetailsType, Session } from 'next-auth'
import { errorResponse } from '@/utils/server/errResponse'
import Product from '@/models/product'

export interface ActionLikeProductProps {
  productID: string
  userID?: string
}

export async function actionLikeProduct({
  productID,
  userID,
}: ActionLikeProductProps): ActionResponse<boolean | null> {
  console.log('POST ACTION like')

  if (!productID || !userID)
    return errorResponse({ state: false, message: 'bad request' })

  await connectDB('ProductsDB')
  const user = await User.findById(userID).maxTimeMS(1000)

  if (!user)
     return errorResponse({ message: 'user not found', redirectedURL:
'/auth/' })
```

```
  // //! toggle db state

  // find product by id
  const isLikedProduct: boolean = user.liked.includes(productID)

  // add or remove product based on isLikedProduct
  if (isLikedProduct)
        user.liked = user.liked.filter((elID: string) => elID !=
productID)
  else user.liked.push(productID)

  await user.save()

  return { success: true, state: !isLikedProduct }
}

export interface ActionAddCartProps {
  productID: string
  userID?: string
}

// add or delete user cart obj based on state
export async function actionToggleProductCart({
  productID,
  userID,
}: ActionAddCartProps): ActionResponse<boolean | null> {
  if (!productID || !userID)
      return errorResponse({ message: 'bad request', redirectedURL:
'/auth/' })
  await connectDB('ProductsDB')

  const user = await User.findOne({ _id: userID })
    .populate('cart.product')
```

```
    .exec()

  if (!user)
    return errorResponse({ message: 'user not found', redirectedURL:
'/auth/' })

  const isInCart = user?.cart?.some(
      (order: OrderDetailsType) => order.product._id.toString() ===
productID,
  )

  console.log({ isInCart })
  if (isInCart) {
    user.cart = user.cart.filter(
      (order: OrderDetailsType) => order.product._id != productID,
    )
  } else {
    user.cart.push({
      product: productID,
      details: {
        quantity: 1,
        isPaid: false,
      },
    })
  }

  await user.save()

  return { state: !isInCart, success: true }
}

export async function actionUpdateOrderDetails({
  order,
  userID,
```

```
}: {
  order: OrderDetailsType
  userID: string
  contact?: Session['user']['contact']
}): ActionResponse<OrderDetailsType | null> {
  const { _id, details } = order

  await connectDB('ProductsDB')

  const userDB = await User.findById(userID)
  const productDB = await Product.findById(order.product._id)

  const isValidOrder =
    details.quantity >= 0 && details.quantity <= productDB.quantity

  if (isValidOrder) {
     const updatedOrders = userDB.cart.map((order: OrderDetailsType)
=> {
      return order._id.toString() === _id
        ? { ...order, details: details }
        : order
    }) as [OrderDetailsType]

    userDB.cart = updatedOrders

    await userDB.save()
  }

  return isValidOrder
    ? { success: true, state: order }
       : { success: false, state: null, message: 'invalid order
quantity' }
}
```

```
interface ActionSetContact {
  userID: string
  contact: Session['user']['contact']
}

export async function actionSetContact({
  userID,
  contact,
}:  ActionSetContact):  ActionResponse<Session['user']['contact']  |
null> {
  if (!contact || !userID) return errorResponse({ message: 'bad req'
})
  try {
    const { address, email, fullName, phone } = contact

    const user = await User.findById(userID)
    user['contact'] = contact
    await user.save()
  } catch (error) {
    return errorResponse({ message: 'bad contact request' })
  }

  return { state: contact, success: true }
}
```