

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до кваліфікаційної роботи освітнього ступеня «бакалавр»
за спеціальністю 121 «Інженерія програмного забезпечення»
(освітня програма «Інженерія програмного забезпечення»)

на тему:

«Інтернет-магазин одягу»

Виконав студент групи ЗПЗ-21-1
КОРОТКИЙ Микита Олександрович

Керівник роботи:
ЧИЖМОТРЯ Олексій Володимирович

Рецензент:
БАКАЛЮК Тетяна Анатоліївна

Житомир – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДЕРЖАВНИЙ УНІВЕРСИТЕТ «ЖИТОМИРСЬКА ПОЛІТЕХНІКА»
ФАКУЛЬТЕТ ІНФОРМАЦІЙНО-КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

«ЗАТВЕРДЖУЮ»

Зав. кафедри інженерії
програмного забезпечення

Тетяна ВАКАЛЮК

«14» лютого 2025 р.

ЗАВДАННЯ
на кваліфікаційну роботу

Здобувач вищої освіти: **КОРОТКИЙ Микита Олександрович**

Керівник роботи: **ЧИЖМОТРИЯ Олексій Володимирович**

Тема роботи: **«Інтернет-магазин одягу»**,

затверджена Наказом закладу вищої освіти від **«14» лютого 2025 р., №62/с**

Вихідні дані для роботи: предметом дослідження є програмне забезпечення інтернет-магазину одягу, його функціональна архітектура, структура бази даних, взаємодія користувача з інтерфейсом, а також процеси, що забезпечують повний цикл купівлі товару; об'єктом дослідження є процес реалізації електронної комерції в сфері роздрібного продажу одягу.

Консультанти з бакалаврської кваліфікаційної роботи із зазначенням розділів, що їх стосуються:

Розділ	Консультант	Завдання видав	Завдання прийняв
1	Чижмотря О. В.	14.02.2025р.	14.02.2025р.
2	Чижмотря О. В.	14.02.2025р.	14.02.2025р.
3	Чижмотря О. В.	14.02.2025р.	14.02.2025р.

РЕФЕРАТ

Випускна кваліфікаційна робота на тему «Інтернет-магазин одягу» містить 89 сторінок, 36 ілюстрацій, 9 таблиць, додаток, 31 використане літературне джерело.

Метою роботи є розробка сучасного вебзастосунку для продажу одягу онлайн з використанням сучасних вебтехнологій.

У процесі виконання роботи використано фреймворк Next.js для побудови інтерфейсу користувача та серверної логіки, а також базу даних для зберігання інформації про товари, замовлення та користувачів. Основні методи, які застосовувались у роботі — аналіз потреб цільової аудиторії, проєктування архітектури вебзастосунку, реалізація функціоналу та тестування.

Структура роботи складається зі вступу, чотирьох розділів основної частини, висновків, списку використаних джерел та додатків. У першому розділі подано аналіз предметної області та технічних рішень. У другому — описано вибір технологій і проєктування структури застосунку. У третьому — представлено реалізацію інтерфейсу, серверної логіки та бази даних. У четвертому — проведено тестування та наведено рекомендації щодо використання системи.

У результаті було створено функціональний прототип інтернет-магазину, який забезпечує реєстрацію користувачів, перегляд каталогу товарів, додавання товарів у кошик, оформлення замовлень тощо.

Результати роботи можуть бути впроваджені у малого або середнього бізнесу для автоматизації онлайн-продажів. Сайт протестований та готовий до подальшого розширення.

Ключові слова: БАЗА ДАНИХ, ІНТЕРНЕТ-МАГАЗИН, NEXT.JS, КОРИСТУВАЧ

					ІПЗ.КР.Б – 121 – 25 – ПЗ			
Зм.	Лист	№ докум.	Підпис	Дата				
Розроб.		Короткий М.О.			Інтернет-магазин одягу Пояснювальна записка	Літ.	Аркуш	Аркушів
Керівник		Чижмоторя О.В.					3	89
						Державний університет «Житомирська політехніка» група ЗІПЗ-21-1		
Рецензент		Вакалюк Т.А.						
Зав. каф.		Вакалюк Т.А.						

ABSTRACT

The bachelor's thesis titled "Online Clothing Store" consists of 89 pages, 36 illustrations, 9 tables, appendice, and 31 sources.

The purpose of this work is to develop a modern web application for online clothing sales using up-to-date web technologies.

The project uses the Next.js framework for building both the user interface and server-side logic, as well as a database to store product, order, and user information. The main methods applied include audience needs analysis, web app architecture design, implementation of functionality, and testing.

The structure of the consists of an introduction, four chapters of the main part, conclusions, list of references and appendices. The first section provides an analysis of the subject area and technical solutions. The second section describes the choice of technologies and design of the application structure. The third section presents the implementation of the interface, server logic, and database. The fourth section tests and provides recommendations on how to use the system.

As a result, a functional prototype of an online store was developed, providing user registration, product catalog viewing, shopping cart functionality, order processing, and more.

The results of the study can be applied in small or medium-sized businesses to automate online sales. The site has been tested and is ready for further scaling.

Keywords: DATABASE, E-STORE, NEXT.JS, USER

ЗМІСТ

РЕФЕРАТ	3
ABSTRACT	4
ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ.....	6
ВСТУП	7
РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ІНТЕРНЕТ-МАГАЗИН ОДЯГУ	9
1.1. Постановка задачі	9
1.2. Аналіз аналогів програмного продукту.....	10
1.3. Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення	14
Висновки до першого розділу	20
РОЗДІЛ 2. ПРОЄКТУВАННЯ ІНТЕРНЕТ-МАГАЗИН ОДЯГУ	22
2.1. Визначення варіантів використання та об'єктно-орієнтованої структури системи.....	22
2.2. Розробка бази даних системи	25
2.3. Проєктування та реалізація алгоритмів роботи системи.....	34
2.4. Реалізація веб-сайту інтернет-магазину одягу	41
Висновки до другого розділу.....	54
РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ВЕБ-ДОДАТКОМ ІНТЕРНЕТ-МАГАЗИНОМ ОДЯГУ	55
3.1. Порядок налаштування параметрів системи.....	55
3.2. Структура веб-додатку інтернет-магазин одягу	57
3.3. Тестування роботи програмного комплексу	67
Висновки до третього розділу	72
ВИСНОВКИ.....	73
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ	75
ДОДАТКИ.....	78

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

СКБД - Система керування базами даних

RHF – react-hook-form

ORM - Object-Relational Mapping

UI – User Interface

UX – User Experience

API – Application Programming Interface

SPA – Single Page Application

CSR – Client-Side Rendering

SSR – Server-Side Rendering

CRUD – Create, Read, Update, Delete

DB –база даних

TS – TypeScript

JS – JavaScript

CSS – Cascading Style Sheets

HTML – HyperText Markup Language

REST – Representational State Transfer

ВСТУП

У сучасних умовах динамічного розвитку цифрових технологій електронна комерція стрімко набирає популярності у всьому світі. Онлайн-платформи для продажу товарів та послуг стали невід’ємною частиною повсякденного життя людей. Особливо актуальною є розробка спеціалізованих інтернет-магазинів, які надають можливість користувачам швидко, зручно та безпечно здійснювати покупки з будь-якої точки світу.

Актуальність теми випускної роботи обумовлена зростаючим попитом на онлайн-платформи для роздрібної торгівлі, зокрема в індустрії моди. В умовах високої конкуренції на ринку одягу компанії активно шукають ефективні способи взаємодії з клієнтами через цифрові канали. Інтернет-магазини, стали основним інструментом для продажу товарів, просування бренду та забезпечення зворотного зв’язку з покупцем. Незважаючи на велику кількість існуючих платформ, багато бізнесів потребують індивідуальних рішень, адаптованих до своїх потреб, стилю та цільової аудиторії. Саме тому створення інтернет-магазину з використанням сучасних технологій, таких як Next.js [1] та інтеграція з базами даних, є актуальним завданням прикладної інформатики.

Наукова новизна роботи полягає в поєднанні швидкого серверного рендерингу, зручної маршрутизації та масштабованої архітектури на базі Next.js для створення оптимізованого користувацького досвіду. Аналіз тенденцій у веброботці показує, що використання фреймворку Next.js стає все популярнішим серед розробників завдяки його універсальності та підтримці SSR (Server Side Rendering), що позитивно впливає на SEO [29] та швидкість завантаження сторінок.

Метою роботи є створення сучасного вебзастосунку — інтернет-магазину одягу, який дозволяє користувачам переглядати каталог товарів, додавати їх до кошика, оформляти замовлення та взаємодіяти з інтерфейсом у зручний та інтуїтивно зрозумілий спосіб.

Для досягнення цієї мети необхідно виконати наступні завдання:

1. Проаналізувати існуючі рішення інтернет-магазинів та визначити їхні сильні й слабкі сторони;
2. Спроекувати архітектуру вебзастосунку з урахуванням вимог зручності, безпеки та масштабованості;
3. Реалізувати функціонал інтернет-магазину з використанням фреймворку Next.js;
4. Організувати взаємодію з базою даних для збереження інформації про товари, користувачів та замовлення;
5. Протестувати працездатність системи та провести її оптимізацію;
6. Оцінити можливості масштабування та перспективи впровадження у реальні бізнес-процеси.

Об'єктом дослідження є процес реалізації електронної комерції в сфері роздрібного продажу одягу.

Предметом дослідження виступає програмне забезпечення інтернет-магазину одягу, його функціональна архітектура, структура бази даних, взаємодія користувача з інтерфейсом, а також процеси, що забезпечують повний цикл купівлі товару.

Методи дослідження, що були використані в роботі, включають: аналіз і синтез інформаційних джерел, порівняльний аналіз існуючих вебтехнологій, проєктування інтерфейсу користувача, побудова архітектурної моделі програмного забезпечення, реалізація прототипу за допомогою фреймворку Next.js, тестування застосунку та моделювання поведінки користувача в системі.

Інформаційну базу дослідження склали наукові статті, онлайн-курси, технічна документація до Next.js, сучасні рекомендації з UX/UI дизайну [28].

Практичне значення отриманих результатів полягає у створенні готового до впровадження інтернет-магазину, який може бути адаптований для реальних бізнес-потреб. Система може бути використана як стартап-рішення для малого бізнесу або в ролі навчального проєкту з вивчення сучасної веброзробки.

РОЗДІЛ 1. АНАЛІЗ НАПРЯМКІВ ЗАСТОСУВАННЯ ІНТЕРНЕТ-МАГАЗИН ОДЯГУ

1.1. Постановка задачі

Розробка веборієнтованих застосунків, зокрема інтернет-магазинів, вимагає комплексного підходу, що охоплює як технічні, так і бізнесові аспекти. Стрімкий розвиток електронної комерції змушує розробників впроваджувати нові рішення, які не лише відповідають сучасним вимогам, а й забезпечують користувацький комфорт, безпеку та стабільність роботи ресурсу. У межах цієї кваліфікаційної роботи ставиться завдання створити інтернет-магазин одягу з повноцінною системою управління товарами, замовленнями та користувачами. Сайт має відповідати основним вимогам до сучасних e-commerce платформ і демонструвати застосування актуальних вебтехнологій.

Передусім необхідно провести глибокий аналіз предметної області, зокрема структури та логіки роботи подібних систем. Аналіз дозволяє виявити типові функціональні модулі, зокрема систему авторизації, каталог продукції, фільтрацію, кошик, оформлення замовлення та сторінку оплати. З метою точного формулювання вимог до майбутнього продукту, було вивчено кілька реальних прикладів інтернет-магазинів (як локальних, так і міжнародних), а також враховано очікування кінцевих користувачів щодо зручності навігації, швидкодії, адаптивності інтерфейсу та естетичного дизайну.

Технічне завдання передбачає створення повнофункціонального вебзастосунку, розробленого з використанням сучасного фреймворку Next.js, який поєднує переваги серверного рендерингу (SSR) і клієнтської SPA-архітектури. В якості сховища даних буде використано реляційну базу даних, яка зберігатиме відомості про користувачів, товари, категорії, замовлення, а також додаткові атрибути, необхідні для ефективної роботи системи.

Особливу увагу буде приділено проєктуванню архітектури майбутнього застосунку. Потрібно чітко визначити логіку взаємодії між клієнтською та серверною частинами, структуру бази даних, а також забезпечити ефективний обмін інформацією через API [2]. У системі повинна бути реалізована

авторизація користувачів, яка дозволяє здійснювати захищений вхід до особистого кабінету, оформлення замовлень та відстеження їх статусу.

Інтерфейс користувача повинен бути зручним, логічним, адаптивним до різних типів пристроїв, а також естетично привабливим. Він має включати головну сторінку, каталог з можливістю пошуку та фільтрації, сторінку товару з докладним описом і галереєю зображень, кошик та форму оформлення замовлення. Усі компоненти мають працювати узгоджено, забезпечуючи швидкий та інтуїтивно зрозумілий користувацький досвід.

Реалізація серверної логіки включає створення API-інтерфейсів [19] для взаємодії з базою даних. Це дозволить виконувати операції додавання, редагування, видалення та перегляду даних, що зберігаються у системі. Також необхідно впровадити механізми захисту від типових загроз (ін'єкції, міжсайтові скрипти, CSRF), забезпечити валідацію даних та надійну обробку помилок.

Завершальними етапами роботи є тестування, налагодження та підготовка системи до розгортання. Тестування охоплює як окремі компоненти, так і систему загалом. Воно повинно гарантувати правильність функціонування, відповідність усім заявленим вимогам та стабільність роботи при реальному навантаженні. Після завершення тестування система готується до розгортання на віддаленому сервері або спеціалізованій платформі на кшталт Vercel, що забезпечить доступність інтернет-магазину для широкої аудиторії.

Таким чином, у межах кваліфікаційної роботи буде розв'язано низку взаємопов'язаних задач, що охоплюють аналіз, проєктування, розробку, тестування та впровадження вебзастосунку, який може бути використаний як основа для комерційного проєкту у сфері онлайн-торгівлі.

1.2. Аналіз аналогів програмного продукту

Сфера електронної комерції в Україні та світі активно розвивається, і ринок інтернет-магазинів одягу є одним із найбільш динамічних. З метою створення конкурентоспроможного продукту було проведено аналіз популярних рішень, доступних як для кінцевих користувачів, так і для власників бізнесу.

Аналіз охоплює як міжнародні бренди, так і локальні платформи, що дозволяє отримати повну картину поточного стану галузі.

Серед глобальних лідерів у сфері онлайн-торгівлі одягом варто виділити Zalando, ASOS, H&M, а також всесвітньо відому платформу Shopify, яка використовується для створення індивідуальних онлайн-магазинів. В українському контексті особливої уваги заслуговують маркетплейс Prom.ua та спеціалізовані проєкти на кшталт Kasta [23].

Zalando [22] вирізняється широкою товарною базою та високим рівнем персоналізації (рис. 1.1.). Користувачі отримують індивідуальні рекомендації на основі попередніх пошуків і покупок. Інтерфейс сайту оптимізований для зручної навігації, пропонуються численні фільтри за стилем, брендом, кольором, матеріалом, сезоном тощо.

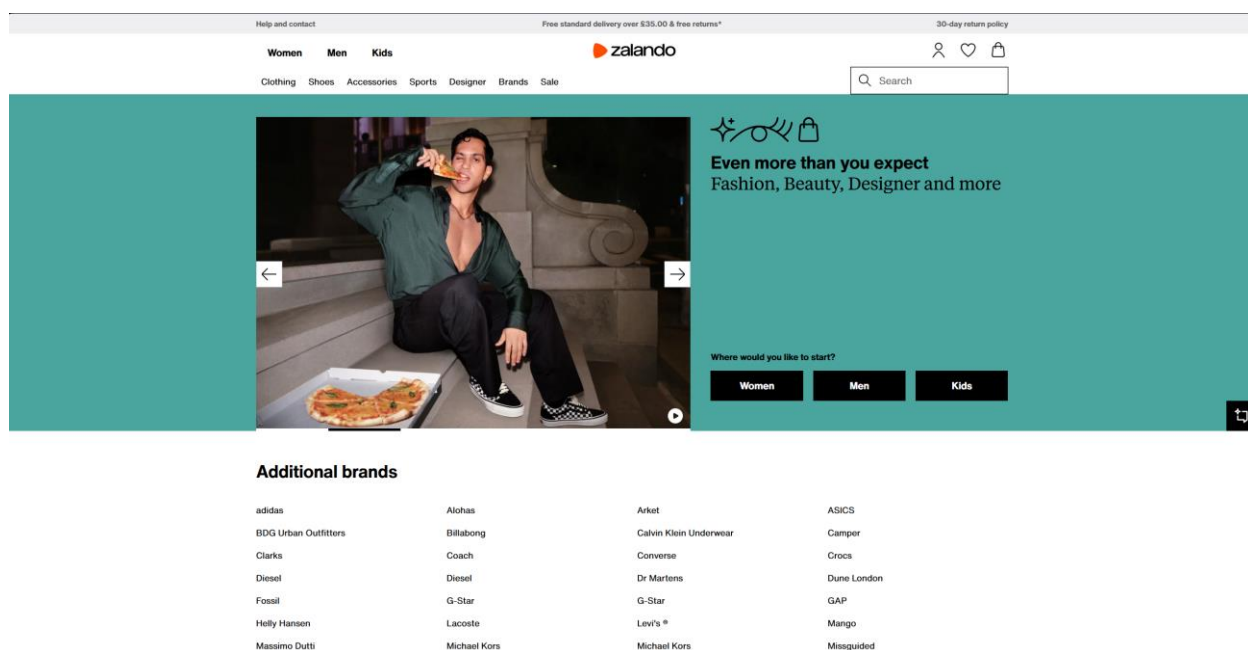


Рисунок 1.1 – Інтерфейс www.zalando.co.uk

ASOS [24] активно використовує сучасні технології — зокрема відеоогляди товарів та функцію віртуальної примірки (рис. 1.2.). Це дозволяє користувачам краще оцінити вигляд товару в динаміці. Платформа також підтримує гнучку систему повернення товарів, що є критично важливим у сфері продажу одягу онлайн. Крім того, ASOS використовує алгоритми машинного

навчання для рекомендацій товарів, з урахуванням індивідуальних параметрів користувача.

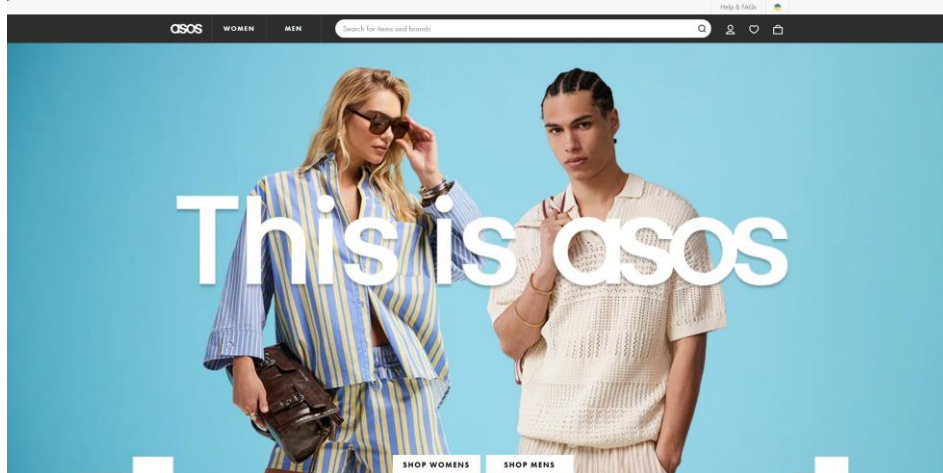


Рисунок 1.2 – Інтерфейс asos.com

H&M [25], окрім онлайн-магазину, пропонує зручну інтеграцію з мережею фізичних магазинів (рис. 1.3.). Користувач може перевірити наявність товару в конкретному торговому центрі, що покращує користувацький досвід. Дизайн сайту витриманий у мінімалістичному стилі, а швидкість завантаження оптимізована навіть для слабких підключень.

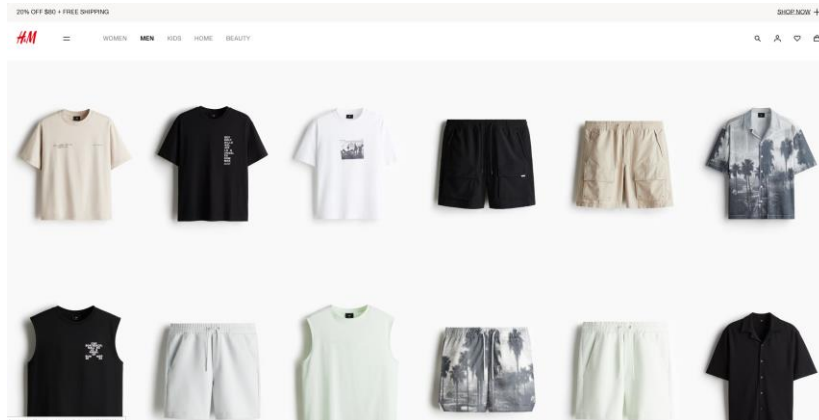


Рисунок 1.3 – Інтерфейс hm.com

Окремо слід розглянути Shopify — платформу, яка дозволяє створювати кастомізовані інтернет-магазини без необхідності глибоких знань у програмуванні. Shopify надає широкий набір шаблонів, інтеграцію з платіжними системами, складським обліком, звітністю та аналітикою. Проте використання Shopify передбачає щомісячну абонплату, а функціональність розширюється лише за допомогою додаткових платних модулів. Для бізнесів, що прагнуть повного контролю над логікою сайту, це може бути обмеженням.

Kasta є прикладом платформи, яка поєднує локалізацію, акції, розпродажі, оплату після примірки та оперативну доставку по Україні (рис. 1.4.). Сервіс адаптований під українського користувача, має інтеграцію з популярними поштовими службами та можливість замовлення без попередньої реєстрації.

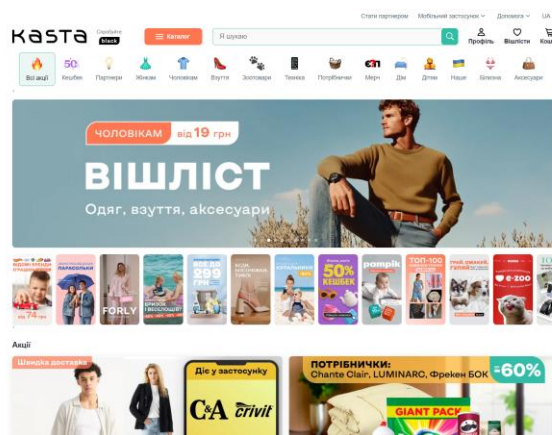


Рисунок 1.4 – Інтерфейс kasta.ua

Таблиця 1.1.

Таблиця аналогів

Платформа	Варіант и товару	Розширена фільтрація	Drawer- корзина	SSR	Кастомне формування замовлення	Масшта- бованість	Гнучка кастоміза- ція
Власна розробка	+	+	+	+	+	+	+
Shopify	+	—	—	+	—	+	—
Zalando	+	+	—	+	—	+	—
Kasta	+	+	—	+	—	+	—
ASOS	+	+	+	+	—	+	—

Результати аналізу свідчать, що сучасні інтернет-магазини одягу мають високий рівень функціональної складності, що забезпечує зручність для кінцевого користувача, але водночас створює труднощі в адаптації та розширенні для малого або середнього бізнесу. Більшість платформ побудовані як SaaS-рішення або великі корпоративні системи. У цьому контексті перевагою власної розробки є можливість гнучко адаптувати систему під специфічні потреби бізнесу, не обмежуючись типовими шаблонами чи комерційними модулями.

Розроблений сайт вирізнятиметься своєю технічною легкістю, швидкістю завантаження, сучасним технологічним стеком і локальною адаптацією, що робить його ефективним інструментом як для кінцевого користувача, так і для адміністрування бізнесу.

1.3. Обґрунтування вибору інструментальних засобів та вимоги до апаратного забезпечення

Проект інтернет-магазину одягу вимагає вибору оптимальної архітектури веб-додатку [26], враховуючи потреби у продуктивності, SEO та масштабованості. Буде розглянуто два основних підходи: монолітний додаток або мікросервісна архітектура, а також стратегії рендерингу інтерфейсу — серверний рендеринг (SSR) [31] та підхід односторінкового застосунку (SPA) з клієнтським рендерингом.

Монолітна архітектура передбачає, що всі компоненти системи (фронтенд, бекенд, база даних) розгортаються як єдиний цілісний застосунок. Переваги моноліту — спрощена розробка і підтримка, адже весь код зосереджений в одному місці, що полегшує координацію роботи і управління версіями. Також розгортання такого рішення є відносно простим, оскільки немає потреби узгоджувати роботу декількох окремих сервісів. Недоліком є обмежена масштабованість: складно масштабувати окремі модулі незалежно, що може стати проблемою при значному рості навантаження.

Мікросервісна архітектура, навпаки, розділяє систему на набір незалежних сервісів, кожен з власним призначенням та API. Це підвищує гнучкість і масштабованість, проте додає складності у розробці, налагодженні взаємодії сервісів та оркестрації деплою. У підсумку, вибір між монолітною та мікросервісною архітектурою залежить від специфіки проєкту, його розміру та вимог до масштабування. Для невеликих та середніх проєктів монолітна архітектура часто є оптимальним вибором завдяки простоті та швидкості розробки, оскільки спрощує реалізацію і не вимагає зайвих витрат на підтримку розподіленої системи. Стратегія рендерингу інтерфейсу теж суттєво впливає на вибір технологій.

Односторінкові додатки (SPA) виконують рендеринг на стороні клієнта (в браузері), що зменшує навантаження на сервер, проте перше завантаження сторінки може бути повільнішим, а контент менш доступний для пошукових роботів до повного виконання JavaScript. Натомість серверний рендеринг (SSR) генерує HTML-сторінки на сервері на кожен запит. Це забезпечує швидке відображення контенту для користувача з самого початку та краще індексування сторінок пошуковими системами, адже пошуковикам не потрібно виконувати JavaScript для отримання змісту сторінки. SSR також покращує доступність контенту, оскільки вся інформація одразу присутня в HTML і не залежить від роботи скриптів; це особливо важливо для користувачів з обмеженими можливостями (екранних читалок тощо). Недоліком SSR є збільшене навантаження на сервер та складніша архітектура застосунку, що має працювати і на сервері, і в браузері. Сучасні фреймворки дозволяють реалізувати «гібридний» підхід, поєднуючи плюси SSR і SPA: початковий контент рендериться на сервері для швидкого старту, а подальша взаємодія виконується як SPA на клієнті. Такий ізоморфний рендеринг забезпечує і швидкий перший відгук, і динамічність подальшої роботи застосунку. Зважаючи на характер проєкту (каталог товарів з можливістю онлайн-замовлення) та обмежений масштаб розробки, було вирішено використати монолітну архітектуру з ізоморфним рендерингом на базі фреймворку Next.js. Це дозволило реалізувати

веб-додаток, що поєднує серверну логіку та клієнтський інтерфейс в одному репозиторії, спрощує розгортання та забезпечує оптимальний баланс між швидкістю роботи і простотою підтримки.

Основою клієнтської частини та веб-інтерфейсу став фреймворк Next.js версії 14, побудований на бібліотеці React 18 [3]. Next.js – це відкритий фреймворк від компанії Vercel, що розширює можливості React для розробки високопродуктивних веб-застосунків із підтримкою серверного рендерингу. Використання Next.js обумовлене його перевагами: він забезпечує швидку роботу та оптимізоване завантаження сторінок, маючи з коробки механізми SSR та гібридного рендерингу. Фреймворк автоматично здійснює розподіл коду між сторінками (code splitting), що прискорює завантаження інтерфейсу. Крім того, Next.js пропонує зручну систему маршрутизації на основі файлової структури проєкту, спрощуючи організацію сторінок, та підтримує попередню генерацію статичних сторінок (SSG) для контенту, який рідко змінюється. Таким чином, Next.js поєднує переваги статичного й динамічного підходів: коли користувач заходить на сайт, фреймворк може або віддати заздалегідь згенеровану HTML-сторінку, або згенерувати контент на сервері динамічно – це гарантує швидку видачу актуальної інформації і є ефективним з точки зору SEO. Важливим фактором вибору Next.js стала його повноцінна інтеграція з бекендом: фреймворк дозволяє реалізувати серверну логіку через вбудовані API-роути. Тобто у тому самому проєкті можна визначати серверні функції (наприклад, для обробки замовлень або інтеграції з платіжною системою) без потреби розгортати окремий бекенд-сервіс. Такий підхід відповідає обраній монолітній архітектурі і спрощує розробку та деплоймент. Крім того, Next.js підтримує оптимізацію зображень та SEO-налаштувань (мета-теги, OpenGraph) на рівні сторінок, що є суттєвим для інтернет-магазину (наприклад, сторінки товарів мають коректні заголовки і метадані для пошукових систем, а зображення автоматично стискаються і підлаштовуються під різні роздільності екрану).

React, на якому базується Next.js, обрано як бібліотеку користувацького інтерфейсу завдяки її популярності та ефективності у побудові складних

інтерактивних UI. React забезпечує компонентно-орієнтований підхід: інтерфейс розбитий на незалежні компоненти, що спрощує розвиток і підтримку коду. Версія React 18 привнесла покращення в продуктивність рендерингу (зокрема, завдяки новим можливостям Concurrent Mode та автоматичному пакетуванню оновлень стану), що позитивно впливає на відгук інтерфейсу. Завдяки використанню React, розробники мають доступ до екосистеми численних готових бібліотек і можуть повторно використовувати UI-компоненти. У поєднанні з Next.js, React 18 дозволяє створити сучасний інтерфейс, який швидко реагує на дії користувача та одночасно забезпечує серверне формування початкового вмісту сторінок.

У розробці проєкту використано мову TypeScript [5] – надмножину JavaScript із підтримкою статичної типізації. Головною перевагою TypeScript є те, що статично типізований код легше підтримувати та масштабувати у довгостроковій перспективі. Завдяки суворій типізації типові помилки виявляються ще на етапі компіляції, що підвищує надійність програмного забезпечення та полегшує налагодження. Для командної розробки це особливо цінно: нові учасники команди можуть швидше зрозуміти структуру програми, маючи чіткі визначення типів і інтерфейсів, які документують контракт між компонентами системи. TypeScript також забезпечує кращу інтеграцію з інструментами розробки (IDE) – автодоповнення, рефакторинг та навігація по коду працюють ефективніше, що прискорює процес створення функціоналу. Оскільки проєкт базується на сучасному стеку (React 18, Next.js), TypeScript є природним вибором: він повністю сумісний з цими технологіями і підтримується на рівні фреймворку Next.js без додаткових налаштувань. Таким чином, використання TypeScript дозволить підвищити якість коду інтернет-магазину та убезпечити його від багатьох типових помилок, які в JavaScript проявляються лише під час виконання.

Для реалізації оформлення інтерфейсу було обрано фреймворк Tailwind [4] CSS, який використовує підхід utility-first. Tailwind CSS надає набір класів-утиліт, що інкапсулюють базові CSS-властивості (розміри, відступи, кольори,

шрифти тощо). Використання цих утилітарних класів безпосередньо в HTML/JCX розмітці дозволяє швидко створювати сучасні, адаптивні та узгоджені дизайни без написання CSS-коду з нуля. Фреймворк істотно прискорює процес верстки: розробник може одразу застосовувати готові стилі до елементів, уникаючи постійного переключення між розміткою і таблицями стилів. Важливою перевагою Tailwind є підтримка консистентності дизайну. У проєкті визначено єдину палітру кольорів, типографіку та систему відступів згідно з конфігурацією Tailwind – це гарантує уніфікований стиль усіх сторінок. Tailwind фактично примушує використовувати стандартну шкалу значень для відступів, кольорів і розмірів шрифту, що запобігає появі «випадкових» нестандартних стилів та спрощує підтримку коду. Крім того, Tailwind CSS автоматично видаляє невикористані стилі під час збірки (purge), мінімізуючи розмір фінального CSS і покращуючи продуктивність. Завдяки Tailwind стало можливим швидко зверстати інтерфейс інтернет-магазину, дотримуючись принципів сучасного дизайну і не жертвуючи зручністю подальшої підтримки.

Окрім основного стеку, в проєкті використано декілька допоміжних бібліотек для оптимізації роботи фронтенду. Для керування глобальним станом застосунку обрано легку бібліотеку Zustand [10] – вона надає простий API для створення сховища стану React без зайвого шаблонного коду, що вигідно відрізняє її від більш складних рішень (на кшталт Redux). Використання Zustand обґрунтоване потребою зберігати певні дані (наприклад, вміст кошика покупок чи статус автентифікації) доступними в різних компонентах, забезпечуючи при цьому високу продуктивність і невеликий розмір бібліотеки.

Для зберігання даних інтернет-магазину (асортимент товарів, інформація про користувачів, замовлення тощо) обрано реляційну систему управління базами даних PostgreSQL [7]. PostgreSQL – одна з найпотужніших відкритих СКБД, що здобула високу репутацію завдяки надійності та цілісності даних, які вона забезпечує. Як відкритий продукт, PostgreSQL не потребує ліцензійних витрат, має широку спільноту підтримки і працює на всіх популярних платформах – це зручно як для розробки, так і для розгортання. Для доступу до

бази даних у коді застосунку використовується ORM Prisma [6]. Prisma – сучасний інструмент об’єктно-реляційного відображення для Node.js та TypeScript, який забезпечує розробникам зручний і безпечний інтерфейс до бази даних. Замість написання сирих SQL-запитів Prisma дозволяє оперувати даними через методи та об’єкти, що відповідають схемі БД, причому всі типи даних строго визначені на рівні TypeScript. Це означає, що під час компіляції програми перевіряється коректність звертань до бази (імен полів, типів значень тощо), що запобігає багатьом помилкам. За допомогою Prisma розробник описує модель даних у єдиному файлі схеми (schema.prisma), на основі якої ORM генерує типізований клієнт для доступу до таблиць. Такий підхід прискорює розробку і підвищує впевненість у правильності коду, оскільки Prisma автоматично виконує необхідні міграції бази даних та надає автодоповнення під час написання запитів

У результаті зв’язка PostgreSQL + Prisma [18] забезпечить надійний механізм збереження даних із зручним інструментарієм для взаємодії: це спростить реалізацію функціоналу роботи з товарами та замовленнями у застосунку.

Для впровадження онлайн-оплати замовлень у проєкті буде інтегровано платіжний сервіс LiqPay [8]. LiqPay — це електронна платіжна система інтернет-еквайрингу від ПриватБанку, найбільшого банку України. Вона дозволяє покупцям безпечно розраховуватися банківськими картками Visa/MasterCard просто на сайті, а кошти зараховуються на рахунок магазину практично миттєво. Вибір LiqPay обумовлений її популярністю на локальному ринку та наявністю зручного API для інтеграції: сервіс надає готові інструменти (SDK, REST API) для проведення платежів і перевірки їх статусу. До того ж, використання LiqPay знімає з розробників необхідність обробляти та зберігати конфіденційні платіжні дані клієнтів (номери карток тощо), оскільки усі такі операції виконуються на захищених серверах LiqPay. Це підвищує безпеку рішення і спрощує відповідність стандартам (PCI DSS) при прийомі онлайн-оплат.

Для хостингу та деплою веб-застосунку було обрано платформу Vercel [21]. Vercel – це хмарний сервіс, створений тією ж командою, що

розробляє Next.js, і оптимізований для запуску Next-додатків. Найпростіший спосіб розгорнути Next.js-додаток у продакшн – використати Vercel, адже платформа автоматично визначає потрібні налаштування збірки та надає інфраструктуру для масштабування без додаткової конфігурації. Vercel працює за принципом serverless: додаток розгортається у вигляді статичних файлів та хмарних функцій (аналог AWS Lambda) для серверних API-роутів з axios [12], які автоматично масштабуються під навантаження. Статичні ресурси (HTML, CSS, зображення) обслуговуються через глобальну мережу CDN, що забезпечує швидку доставку контенту користувачам по всьому світу. Процес CI/CD [14] (безперервної інтеграції та розгортання) на Vercel максимально спрощений: достатньо підключити репозиторій проєкту з GitHub [20], і при кожному оновленні основної гілки код автоматично збирається та публікується нова версія застосунку. Це прискорює цикл розробки та усуває людський фактор при деплоїменті. Vercel має безкоштовний тарифний план, достатній для потреб невеликого інтернет-магазину на етапі розробки та тестування, з можливістю подальшого масштабування до платних планів у разі зростання аудиторії. Використання Vercel також усуває необхідність керувати власними серверами або контейнерами: команда може зосередитися на коді, а питання інфраструктури (балансування навантаження, SSL-сертифікати, моніторинг, CDN) вирішує платформа. Такий підхід відповідає вимогам проєкту щодо швидкого та надійного запуску сервісу в мережі.

Висновки до першого розділу

У розділі першому було сформульовано основну мету створення інтернет-магазину одягу, визначено його цільову аудиторію та опрацьовано необхідний функціонал.

У рамках аналізу аналогів досліджено провідні інтернет-магазини одягу — Zara, H&M, Kasta, Asos та інші. Виявлено їхні сильні сторони, зокрема інтуїтивно зрозумілий інтерфейс, широкий асортимент товарів, зручне фільтрування і сортування та інтеграцію з популярними платіжними та логістичними сервісами.

На підставі проведеного аналізу виокремлено ключові особливості, які слід врахувати при розробці власної системи — розширений функціонал пошуку і фільтрації, ефективна система рекомендацій, орієнтація на локальний ринок та надійні механізми захисту даних. У рамках аналізу аналогів виявлено їхні сильні сторони, зокрема інтуїтивно зрозумілий інтерфейс, широкий асортимент товарів, зручне фільтрування і сортування, адаптивний дизайн та інтеграцію з популярними платіжними та логістичними сервісами.

РОЗДІЛ 2. ПРОЄКТУВАННЯ ІНТЕРНЕТ-МАГАЗИН ОДЯГУ

2.1. Визначення варіантів використання та об'єктно-орієнтованої структури системи

У проєктованій системі інтернет-магазину одягу передбачено дві базові ролі користувачів: USER (звичайний користувач) і ADMIN (адміністратор). До платформи застосовуються типові сценарії роботи електронної комерції. Зокрема, користувачі можуть реєструватися та проходити авторизацію в системі, проте навіть без цього їм доступні основні функції покупця. Це перегляд каталогу товарів з можливістю фільтрації за різними параметрами (наприклад, за категоріями, ціною чи розмірами), детальний перегляд кожного товару, додавання товарів до кошика, оформлення замовлення та здійснення оплати. Адміністратор володіє всіма функціями звичайного користувача, а також матиме розширені права на управління системою: створення, редагування та видалення товарів, категорій та підкатегорій. Такий розподіл ролей забезпечує розмежування прав доступу, при якому роль ADMIN відповідає за контент каталогу, а звичайний користувач може повноцінно переглядати товари та проводити покупки. Система базується на реляційній СКБД PostgreSQL, що є «хребтом» зберігання структурованих даних – саме цією СКБД зручно користуватися для складних запитів і транзакцій в e-commerce-проєктах.

Для опису об'єктно-орієнтованої структури даних обрано ORM Prisma у зв'язці з PostgreSQL. У проєкті всі сутності (моделі) та зв'язки між ними визначаються у файлі `schema.prisma`, що є основним конфігураційним файлом Prisma. На його підставі Prisma CLI автоматично генерує Prisma Client – типобезпечний API для взаємодії з базою даних. Це означає, що кожна модель описується у формі Prisma-моделі, що значно знижує ризик помилок під час розробки. З огляду на еволюцію системи, Prisma надає зручний механізм міграцій для оновлення схеми бази даних без втрати даних або функціональності. Взаємодія з БД відбуватиметься через згенеровані методи Prisma Client

(наприклад, `prisma.user.findMany()`), що дає можливість ефективно виконувати CRUD-операції без написання сирого SQL.

У системі визначено наступні сутності предметної області:

User – модель користувача. Зберігає інформацію про зареєстрованого користувача: унікальний ідентифікатор, ім'я чи прізвище, email, захешований пароль, роль (USER або ADMIN), а також часові мітки створення та оновлення запису. Кожен користувач може мати один пов'язаний об'єкт Cart (кошик) та може мати багато пов'язаних записів замовлень (Order). Зв'язки визначаються як «один-до-одного» між User і Cart (один користувач – один кошик) і «один-до-багатьох» між User і Order (один користувач – декілька замовлень). Ці зв'язки будуть відображені у `schema.prisma` через `@relation`, що автоматично встановлюється по зовнішніх ключах `userId` у моделях Cart та Order.

Category і Subcategory – моделі, що описують ієрархію каталогів товарів. Сутність Category відповідає за велику категорію товарів (наприклад, «Чоловічий одяг», «Жіночий одяг» тощо), а Subcategory – за підкатегорію в межах цієї категорії (наприклад, «Футболки», «Сукні» тощо). Кожна категорія може містити декілька підкатегорій (зв'язок «один-до-багатьох» у бік Subcategory), а кожна підкатегорія належить до однієї категорії (зв'язок «багато-до-одного» у бік Category). Така структура дозволяє організувати каталог і забезпечити зручну фільтрацію товарів за підрівнями категорій. У схемі даних це реалізується через поле `categoryId` у моделі Subcategory з вказівкою `@relation` на відповідну Category. Приклад зв'язку: Category → Subcategory[], Subcategory → Category. Моделі можуть містити назву та опціонально опис категорії.

Item – модель товару (однієї позиції одягу). Вона містить унікальний ідентифікатор, назву товару, опис, ціну, можливі атрибути (наприклад, колір чи розмір), а також поле `subcategoryId`, яке вказує, до якої підкатегорії (і відповідно до якої категорії) належить цей товар. Зв'язки: кожен Item належить до однієї Subcategory (та опосередковано до однієї Category). Також Item може бути пов'язаний з багатьма CartItem і OrderItem – це відображає факт того, що різні користувачі можуть додавати цей товар у свої кошики або замовлення. Ці зв'язки

реалізуються через відношення «один-до-багатьох» ($\text{Item} \rightarrow \text{CartItem}[]$) та ($\text{Item} \rightarrow \text{OrderItem}[]$), а в моделях Prisma будуть відповідні `@relation` на поля `itemId`.

`Cart` – модель кошика користувача. Зазвичай кожен користувач має один активний кошик, де тимчасово збираються товари перед оформленням замовлення. `Cart` містить свій унікальний ідентифікатор та посилання `userId` на власника (`User`), а також зв'язок з багатьма `CartItem`. Тут реалізовано відношення «один-до-одного» між `User` і `Cart` ($\text{User} \rightarrow \text{Cart}$), а також «один-до-багатьох» між `Cart` і `CartItem` ($\text{Cart} \rightarrow \text{CartItem}[]$). Поле `userId` у `Cart` пов'язує кошик з конкретним користувачем через Prisma `@relation`.

`CartItem` – модель позиції в кошику. Реалізує проміжну таблицю для зв'язку «багато-до-багатьох» між `Cart` і `Item`. Сутність `CartItem` містить власний ідентифікатор, кількість одиниць товару, а також зовнішні ключі `cartId` і `itemId`. Через поле `cartId` (із `@relation` на `Cart`) і поле `itemId` (із `@relation` на `Item`) здійснюється відношення «багато `CartItem` – до одного `Cart`» та «багато `CartItem` – до одного `Item`». Таким чином описуються всі товари, що знаходяться в кошику користувача з вказанням їх кількості.

`Order` – модель замовлення. Містить інформацію про оформлене замовлення: унікальний ідентифікатор, дату і час створення, статус замовлення (наприклад, «PENDING», «SUCCEEDED» тощо), загальну суму та `userId` користувача, який зробив замовлення. Зв'язок «багато-до-одного» між `Order` і `User` (через `userId`) вказує, що кожне замовлення належить конкретному користувачеві. Кожне замовлення містить також набір позицій замовлення, тому в систему вводиться проміжна сутність `OrderItem`.

`OrderItem` – модель позиції в замовленні, аналогічна `CartItem`. Вона зберігає для кожного конкретного товару в замовленні інформацію про кількість, ціну на момент замовлення тощо. Поля `orderId` і `itemId` зв'язують `OrderItem` із відповідним `Order` і `Item` через Prisma `@relation`. Таким чином встановлюється зв'язок «один `Order` – багато `OrderItem`» і «один `Item` – багато `OrderItem`». Ця структура дозволяє фіксувати в замовленні деталі кожного товару (назва, ціну, кількість), навіть якщо пізніше в каталозі ціна товару зміниться.

Усі вищезазначені моделі – User, Category, Subcategory, Item, Cart, CartItem, Order, OrderItem – детально описуються у файлі `schema.prisma`. Наприклад, для кожної моделі задаються поля із вказівкою типів (Int, String, DateTime тощо) та ключових атрибутів (@id, @unique, @relation, @default тощо). Система Prisma автоматично створить відповідні таблиці й зв'язки в PostgreSQL відповідно до цієї схеми. В результаті кожній моделі Prisma Client генерує набір методів для доступу до бази: пошук записів (findMany, findUnique), створення (create), оновлення (update) тощо. Наявність строго типізованого клієнта Prisma забезпечує безпомилкове виконання CRUD-операцій над цими сутностями на етапі компіляції коду.

Узагальнюючи, проєктована система інтернет-магазину одягу підтримує повний цикл обробки замовлення — від реєстрації користувача й перегляду каталогу до оформлення та оплати замовлення. Ролі USER і ADMIN розмежовують права доступу, а ORM Prisma з PostgreSQL надійно реалізують об'єктно-орієнтовану модель даних. Опис сутностей і їхніх взаємозв'язків у `schema.prisma` гарантує, що зміни структури (наприклад, додавання нових атрибутів або зв'язків) будуть прозоро конвертуватися в схему бази і валідуватися на етапі побудови клієнта. Такий підхід спрощує подальшу розробку та обслуговування системи, оскільки всі бізнес-об'єкти чітко відображені у схемі даних і взаємодіють через згенерований Prisma Client

2.2. Розробка бази даних системи

Нормалізація даних є ключовою практикою при проєктуванні реляційних баз даних. Вона полягає в організації таблиць таким чином, щоб зменшити надлишковість і уникнути аномалій при вставці чи оновленні даних. У контексті інтернет-магазину одягу це означає виділення окремих таблиць для користувачів, товарів, категорій, кошика та налагодження чітких зв'язків між ними. Реляційна модель забезпечує цілісність даних через обмеження зовнішніх ключів і гарантує підтримку транзакцій за властивостями ACID, що критично

важливо при обробці замовлень і платежів. Вибір реляційної СКБД з потужною реалізацією транзакцій (таких як PostgreSQL) сприяє надійності системи.

Для зберігання та керування даними обрано систему PostgreSQL – сучасну, відкриту реляційну СКБД з багатим функціоналом. PostgreSQL підтримує розширені типи даних та складну транзакційну обробку, що робить його придатним для великих e-commerce платформ з високими навантаженнями. Він відомий широким набором можливостей (наприклад, продуктивними індексами, засобами безпеки, розширеннями для користувацьких типів). Особливо важливою є вбудована підтримка роботи з JSON-структурами (JSONB), що дозволяє зберігати у базі як реляційні, так і неструктуровані дані. Це буде використано в моделі Order для збереження структури товарів замовлення у полі JSON. Крім того, PostgreSQL має велику ком'юніті і розвинуті можливості резервного копіювання та масштабування, що позитивно впливає на надійність і підтримку проєкту.

У ролі ORM для взаємодії з базою даних обрано Prisma. Prisma – інноваційний ORM-фреймворк, що надає єдиний інтерфейс для різних СКБД і генерує оптимізовані SQL-запити з урахуванням особливостей конкретної СКБД. Як відзначено в літературі, Prisma «адаптується до різних БД, зміщуючи синтаксис SQL залежно від їхніх особливостей». Використання Prisma забезпечує типобезпечний доступ до бази та пришвидшує розробку: моделі бази даних описуються у файлі `schema.prisma`, а згенерований Prisma Client автоматично надає зручні методи для виконання запитів і маніпуляцій з даними. Підтримка різних СКБД (у тому числі PostgreSQL) робить Prisma універсальним інструментом для проєктування додатка. Загалом, комбінація PostgreSQL та Prisma поєднує високу надійність і гнучкість СКБД з ефективною ORM-надбудовою, що полегшує розробку та експлуатацію системи.

Проєктування схеми бази даних здійснювалося з використанням Prisma. Основний файл `schema.prisma` містить оголошення джерела даних та генератора клієнта (рис. 2.1.). У секції `datasource db { provider = "postgresql" url = env("DATABASE_URL") }` вказано тип СКБД (PostgreSQL) та рядок

підключення до неї. Секція `generator client { provider = "prisma-client-js" }` задає генерацію Prisma Client – автогенерованого клієнта для роботи з базою. Далі описуються моделі, де кожна модель відповідає таблиці в БД з набором полів (із зазначенням типів) та атрибутами (наприклад, `@id`, `@unique`, `@default`).

Після визначення моделей за допомогою команди `prx prisma generate` формується Prisma Client – клієнтський бібліотечний код із типами, що відповідають схемі БД. Prisma Client є «автогенерованим і типобезпечним конструктором запитів, налаштованим під вашу схему даних». Завдяки цьому можна виконувати операції над базою (створення, читання, оновлення, видалення) через методи Prisma (наприклад, `prisma.user.create`), отримуючи автодоповнення та перевірку типів на етапі компіляції.

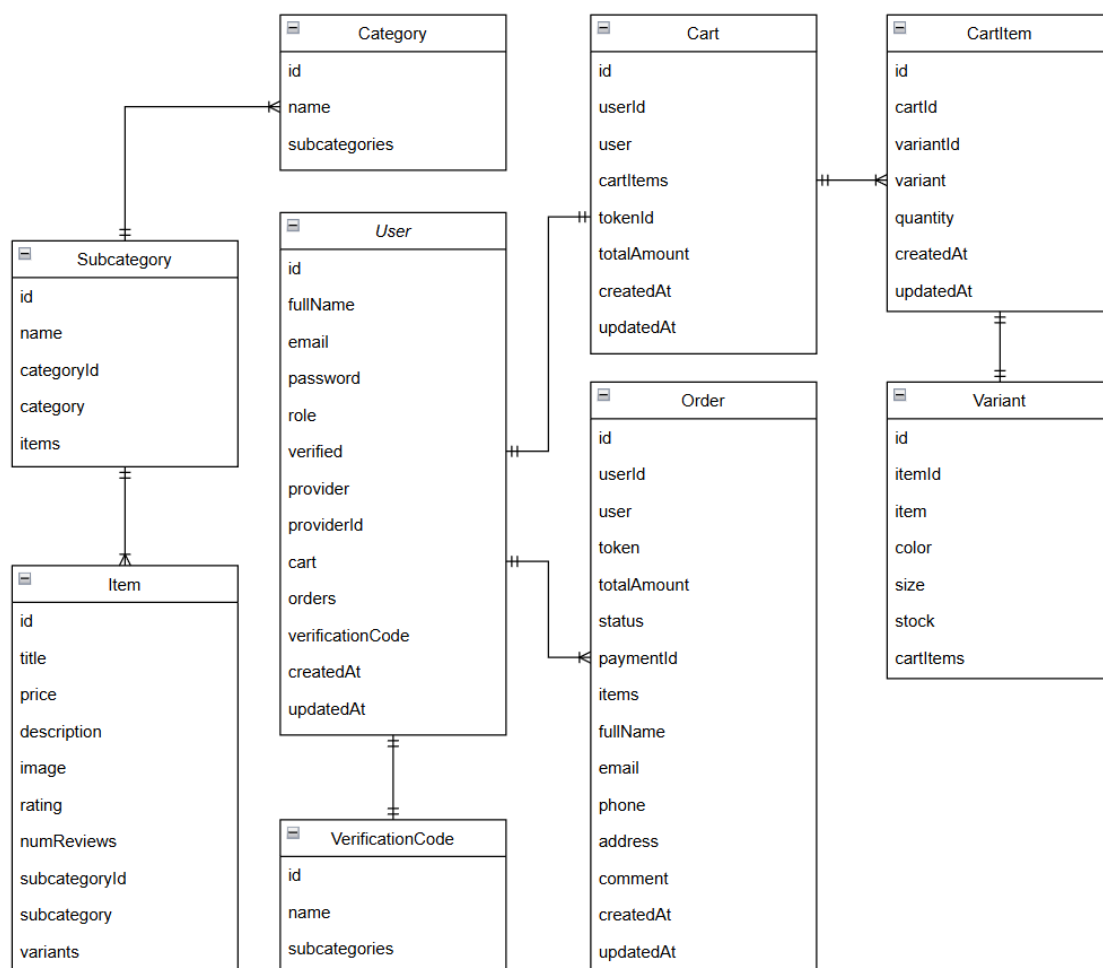


Рисунок 2.1 – Діаграма таблиць бази даних

Зміни у структурі БД відстежуються за допомогою інструмента Prisma Migrate. Кожне оновлення файлу `schema.prisma` фіксується в новій міграції –

SQL-скрипті, що виконує відповідне DDL-онлайн (створення/зміна таблиці тощо). Prisma Migrate генерує історію .sql файлів міграцій для синхронізації реальної схеми БД з моделями застосунку. Використовуючи команди `prisma migrate dev` або `prisma migrate deploy`, розробники можуть застосувати зміни до БД без втрати даних. Такий підхід спрощує еволюцію схеми: файл `schema.prisma` описує бажаний стан схеми декларативно, а Prisma Migrate автоматично створює та застосовує необхідні запити. Усі сутності зв'язків між таблицями (один-до-одного, один-до-багатьох, багато-до-одного) вказуються в описі моделей за допомогою ключових слів `@relation`. Це забезпечує Prisma інформацією для збереження цілісності даних на рівні ORM – наприклад, при видаленні користувача можна вказати каскадне видалення його кошика або кодів верифікації. Таким чином, у підході проєктування використовуються можливості Prisma для управління схемою та підтримки зв'язків між таблицями, що підвищує надійність і зручність роботи з БД.

Модель User відповідає сутності користувача системи. Вона містить стандартні поля: `id` (цілочисельний первинний ключ з автоматичним нарощуванням), `createdAt` (дата реєстрації), `email` (адреса електронної пошти, помічена атрибутом `@unique` для забезпечення унікальності), `password` (хеш пароля), `name` (ім'я користувача) та `role` (роль користувача, тип `UserRole`). Поле `role` реалізовано як перелічуваний тип (enum) `UserRole`, який може містити, наприклад, значення `USER` або `ADMIN`. Зв'язки: кожен користувач має асоційований один кошик (`cart Cart? @relation(...)`), багато замовлень (`orders Order[]`), а також може мати один активний код верифікації (`verificationCode VerificationCode?`). Таким чином, зв'язки: один-до-одного з `Cart` (через поле `cart`) та з `VerificationCode`, а один-до-багатьох з `Order`.

Таблиця 2.1.

Опис таблиці User

Поле	Тип	Призначення
1.	2.	3.
<code>id</code>	<code>Int</code>	Унікальний ідентифікатор користувача
<code>fullName</code>	<code>String</code>	Повне ім'я користувача
<code>email</code>	<code>String</code>	Email-адреса (унікальна)

1.	2.	3.
password	String	Пароль (захешований)
role	UserRole	Роль користувача (USER / ADMIN)
provider	String?	Зовнішній провайдер (опціонально)
providerId	String?	Ідентифікатор у провайдера (опціонально)
verified	DateTime?	Час підтвердження пошти
createdAt	DateTime	Дата створення
updatedAt	DateTime	Дата останнього оновлення

Модель Cart (кошик) містить поля id (PK) та userId (FK на користувача). Вона має зв'язок user User @relation(fields: [userId], references: [id]), що забезпечує один-до-одного відношення з User. У карті зберігаються товари в моделі проміжної таблиці CartItem, для чого в моделі Cart визначено зв'язок items CartItem[] (кошик може містити багато позицій). Модель CartItem є проміжною між кошиком і товаром. Поля: id (PK), cartId (FK на кошик), variantId (FK на конкретний варіант товару), quantity (кількість одиниць). Зв'язки: cart Cart @relation(fields: [cartId], references: [id]) і variant Variant @relation(fields: [variantId], references: [id]). Для уникнення дублювання одного й того ж варіанта в кошику на одну позицію встановлено унікальне обмеження @@unique([cartId, variantId]), тобто один варіант товару може бути вказаний лише один раз у конкретному кошику. (При необхідності користувач може оновити quantity, але не додавати дублікати.)

Таблиця 2.2.

Опис таблиці Cart

Поле	Тип	Призначення
id	Int	Ідентифікатор кошика
userId	Int?	Посилання на користувача (опціонально)
tokenId	String?	Унікальний токен сесії
totalAmount	Int	Загальна сума
createdAt	DateTime	Дата створення
updatedAt	DateTime	Дата оновлення

Модель Order описує замовлення користувача. Поля включають: id (PK), userId (FK на User), createdAt, updatedAt. Додатково є поля для адреси доставки (city, street, zipcode – приклад), status (тип OrderStatus – enum зі значеннями,

наприклад, PENDING, SUCCEEDED, CANCELLED), а також paymentId (рядок ID платежу від платіжної системи) та token (рядок для токена підтвердження замовлення, якщо необхідно). Поле products або items реалізовано як тип Json, у якому зберігається структура придбаних товарів (назви та кількості) на момент оформлення – завдяки підтримці PostgreSQL JSONB така інформація може бути збережена без нормалізації в окрему таблицю. Зв'язок: user User @relation(fields: [userId], references: [id]), що реалізує один-до-багатьох з User (один користувач може мати багато замовлень). Поле status використовує перелічуваний тип OrderStatus, а paymentId і token – прості рядкові поля для інтеграції з платіжними API.

Таблиця 2.3.

Опис таблиці Order

Поле	Тип	Призначення
id	Int	Ідентифікатор замовлення
userId	Int?	Посилання на користувача
items	Json?	JSON зі списком товарів
status	OrderStatus	Статус замовлення
totalAmount	Int	Загальна сума
paymentId	String?	ID платежу
fullName	String	Ім'я одержувача
address	String	Адреса доставки
email	String	Email клієнта
phone	String	Телефон
comment	String?	Коментар до замовлення (опц.)
token	String	Токен замовлення
createdAt	DateTime	Дата створення
updatedAt	DateTime	Дата оновлення

Модель Item описує товар (одну одиницю асортименту). Зазвичай містить поля id, name (назва товару), description (опис), price (ціна), imageUrl (посилання на зображення). Кожен Item належить до певної підкатегорії або категорії, тому є поле subcategoryId (FK на Subcategory). Зв'язок: subcategory Subcategory @relation(fields: [subcategoryId], references: [id]). З одного боку, Item є «багато»-стороною зв'язку один-до-багатьох з моделлю Subcategory (одна підкатегорія може містити багато товарів), з іншого – сам Item має зв'язок один-до-багатьох з моделлю Variant.

Таблиця 2.4.

Опис таблиці Item

Поле	Тип	Призначення
id	Int	Ідентифікатор товару
title	String	Назва товару
price	Int	Ціна
description	String	Опис
image	String	URL зображення
rating	Float	Рейтинг
numReviews	Int	Кількість відгуків
subcategoryId	Int	Посилання на підкатегорію

Модель Variant відповідає конкретному варіанту товару (наприклад, конкретний колір і розмір у продажі). Поля: id (PK), itemId (FK на Item), color (тип Color – enum зі значеннями кольорів, наприклад RED, BLUE тощо), size (тип Size – enum зі значеннями розмірів, наприклад S, M, L), stock (кількість на складі). Зв'язок: item Item @relation(fields: [itemId], references: [id]), що реалізує зв'язок багато-до-одного – кожен варіант належить до одного товару. Поля color і size використовують відповідні enum-типи, обмежуючи допустимі значення (наприклад, для кольорів та розмірів) згідно з бізнес-логікою.

Таблиця 2.5.

Опис таблиці Variant

Поле	Тип	Призначення
id	Int	Ідентифікатор варіанта товару
itemId	Int	Посилання на товар
color	Color	Колір
size	Size?	Розмір (опціонально)
stock	Int	Кількість на складі

Модель Category описує категорію товарів (наприклад, «Чоловічий одяг»). Містить поля id, name. Зв'язок: subcategories Subcategory[] – категорія може включати багато підкатегорій.

Таблиця 2.6.

Опис таблиці Category

Поле	Тип	Призначення
id	Int	Ідентифікатор варіанта товару
name	String	Назва підкатегорії

Модель Subcategory (підкатегорія) має поля id, name, categoryId (FK на Category). Зв'язок: category Category @relation(fields: [categoryId], references: [id]) реалізує відношення багато-до-одного (кожна підкатегорія належить одній категорії), а items Item[] – одна підкатегорія може містити багато товарів. Таким чином, відношення Category–Subcategory є один-до-багатьох.

Таблиця 2.7.

Опис таблиці Subcategory

Поле	Тип	Призначення
id	Int	Ідентифікатор варіанта товару
name	String	Назва підкатегорії
categoryId	Int	Посилання на категорію

Модель VerificationCode відповідає одноразовим кодам підтвердження (наприклад, для авторизації чи підтвердження email). Поля: id, userId (FK на User), code (сам код, рядок), createdAt (дата генерації). Оскільки один користувач може мати тільки один актуальний код одночасно, для поля userId встановлено унікальне обмеження (@unique), що реалізує зв'язок один-до-одного з User. Зв'язок: user User @relation(fields: [userId], references: [id]). Це гарантує, що при генерації нового коду старий запис замінюється новим або видаляється – забезпечуючи унікальність і цілісність.

Таблиця 2.8.

Опис таблиці Subcategory

Поле	Тип	Призначення
id	Int	Ідентифікатор
userId	Int	Посилання на користувача
code	String	Код підтвердження
expiresAt	DateTime	Час завершення дії
createdAt	DateTime	Дата створення
updatedAt	DateTime	Дата оновлення

Перелічувані типи (enum) застосовуються так: OrderStatus використовується в полі status моделі Order для позначення стадії обробки замовлення (наприклад, PENDING, SUCCEEDED); UserRole – в полі role моделі User (наприклад, USER, ADMIN); Color і Size – в моделі Variant для полів color і size відповідно. Це обмежує можливі значення цих полів згідно з предметною областю (кольори і розміри одягу).

У схемі реалізовано такі типи зв'язків між таблицями бази даних. Один-до-одного: між моделями User та Cart (кожен користувач має власний кошик) та між User та VerificationCode (для кожного користувача може бути не більше одного активного коду). Один-до-багатьох: між User та Order (один користувач може мати багато замовлень), між Category та Subcategory (категорія містить багато підкатегорій), та між Item та Variant (товар має багато варіантів). Багато-до-одного: це зворотні сторони попередніх зв'язків – кожна підкатегорія належить одній категорії (Subcategory→Category), а кожен варіант належить одному товару (Variant→Item). Для забезпечення цілісності даних у Prisma застосовано обмеження @relation (для зв'язків) і @unique (унікальних ключів). Зокрема, у моделі CartItem визначено @@unique([cartId, variantId]), що унеможливило дублювання одного й того ж варіанта в рамках одного кошика. Аналогічні унікальні обмеження застосовуються для email в User та userId в VerificationCode (щоб один користувач не міг мати два активних коди).

Усі перераховані моделі та зв'язки описані декларативно в schema.prisma, і Prisma піклується про створення відповідних SQL-структур і зв'язків на рівні PostgreSQL. Така схема реалізації гарантує чітку інтеграцію між моделями та БД: розробник працює з абстракцією моделей, а Prisma автоматично створює необхідні зовнішні ключі, індекси і обмеження на стороні СУБД. Висновок. Розроблена база даних відповідає вимогам предметної області інтернет-магазину одягу. Використання реляційної моделі з PostgreSQL забезпечує надійне зберігання критичних даних (асортименту, замовлень, даних користувачів), а ORM Prisma підвищує продуктивність розробки та читаємість коду. Всі сутності системи (користувач, кошик, товар, замовлення тощо) формалізовані як таблиці з продуманими зв'язками один-до-одного та один-до-багатьох, що забезпечує цілісність і коректність даних. Наявні унікальні обмеження гарантують правильність операцій над даними, а підтримка перелічуваних типів і JSON-структур дозволяє точно відобразити специфіку категорій, кольорів, розмірів та інших атрибутів товарів. Така архітектура БД через Prisma сприяє гнучкості системи та полегшує її майбутнє масштабування.

2.3. Проектування та реалізація алгоритмів роботи системи

Реєстрація та авторизація користувача

Для організації реєстрації (рис. 2.3.) та авторизації (рис. 2.2.) застосовано NextAuth.js [12], що забезпечує підтримку авторизації через зовнішні провайдери (GitHub, Google) і через email+пароль. Використання вбудованих OAuth-провайдерів означає, що користувач може зайти через GitHub або Google без необхідності створювати окремий обліковий запис (NextAuth має готові провайдери для GitHub, Google та інших сервісів). Крім того, реалізовано реєстрацію по email та паролю за допомогою Credentials Provider у NextAuth. На етапі реєстрації генерується унікальний код підтвердження (verification code), який надсилається на вказану користувачем електронну адресу. Цей процес полягає у створенні випадкового хешу, збереженні його у БД разом з email-ом і відправці листа з посиланням, що містить код підтвердження. При переході користувача за цим посиланням виконується перевірка токена (через окремий API-ендпоінт) і, у разі відповідності, активується акаунт. Для форм авторизації та реєстрації застосовано Zod [9] – бібліотеку для декларативної валідації даних. Zod дозволяє описувати схеми даних і автоматично перевіряти, що всі поля введені у коректному форматі; це підвищує цілісність даних і покращує досвід користувача, оскільки помилки введення виявляються ще на клієнті.

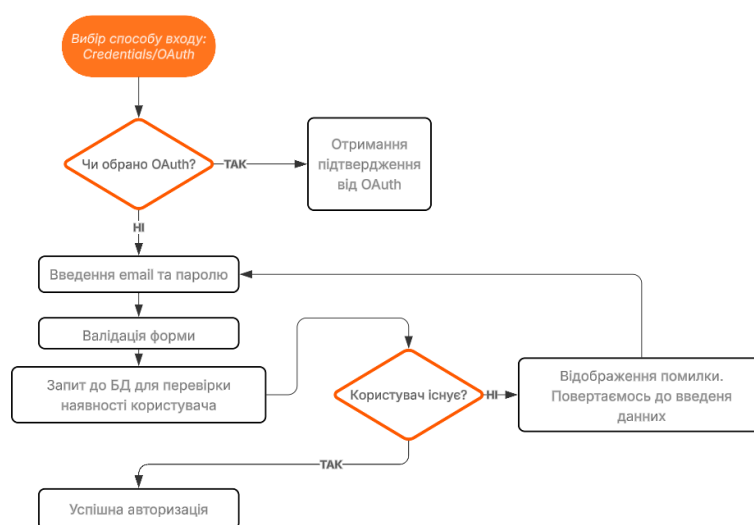


Рисунок 2.2 – Діаграма активності для авторизації

Права доступу організовані таким чином: у неавторизованого користувача відсутня кнопка переходу до профілю, натомість відображається кнопка «Sign in» для входу. Якщо ж неавторизований користувач намагається перейти на адресу профілю або іншу приватну сторінку, система перехоплює цей запит і віддає користувачу кастомізовану помилку (404 зі зверненням про відмову у доступі).

Для перевірки стану сесії на клієнті використовується хук `useSession()` [15] з NextAuth, який повертає об'єкт сесії та статус (`loading`, `authenticated`, `unauthenticated`). Якщо статус не є `authenticated`, доступ до приватних компонентів заборонено і користувач може бути перенаправлений або бачити повідомлення про необхідність авторизації.

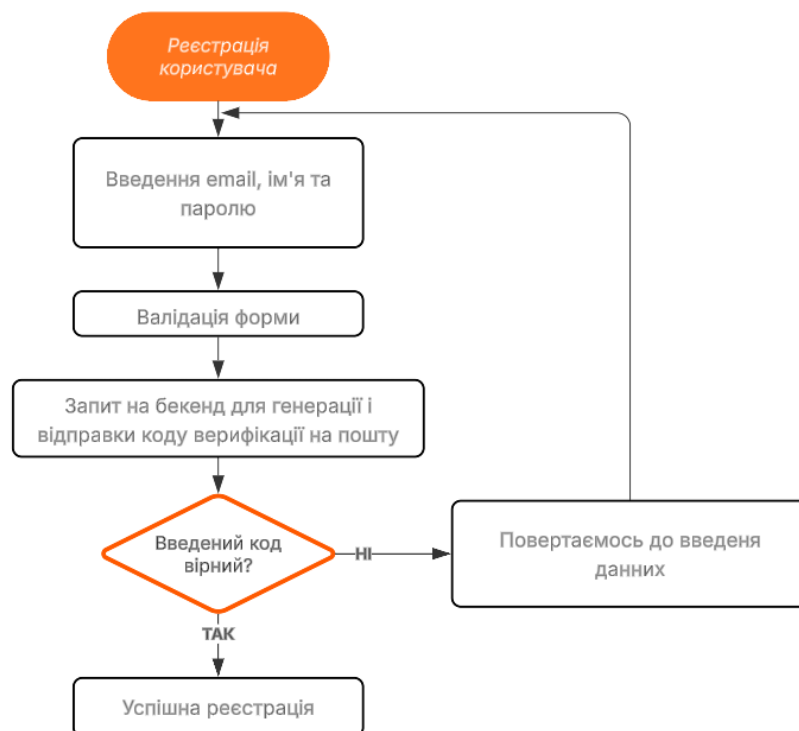


Рисунок 2.3 – Діаграма активності для реєстрації

Отримання та відображення товарів

Дані про товари завантажуються з бази даних через серверні API-ендпоінти Next.js (файли в папці `/api`). В основі запитів лежить Prisma ORM, що забезпечує простий і типізований доступ до БД. Модель даних передбачає зв'язки «один-до-багатьох» між категоріями, підкатегоріями і товарами. Prisma

дозволяє одночасно отримувати пов'язані дані в одному запиті, наприклад завантажувати список товарів разом з їхньою категорією і підкатегоріями (використовуючи `include` у `PrismaClient`). Це означає, що API може повертати ієрархічний набір даних для відображення списку товарів за категоріями: спочатку витягуються категорії з БД, а потім – товари, що їм відповідають. Такий підхід гарантує цілісність даних і зручність формування списків товарів. Зокрема, використання `Prisma` у поєднанні з `Next.js` є потужним поєднанням для створення сучасних серверно-клієнтних веб-додатків, що полегшує реалізацію API для торговельного майданчику. Таким чином, кожний API-ендпоінт, наприклад `/api/items`, може отримувати параметри запиту (категорію, підкатегорію), витягувати товари з `PrismaClient` і повертати їх у JSON-форматі для відображення на клієнті.

Фільтрація товарів

Реалізація фільтрації товарів заснована на передачі параметрів у рядок запиту (`query string`) браузера. Наприклад, для встановлення фільтрів розміру, кольору або підкатегорії використовується URL виду `...?sizes=S&colors=WHITE&subcategories=1`. На клієнті застосовуються стандартні хуки `Next.js` – `useRouter` та `useSearchParams` (`App Router`), які дозволяють читати і змінювати параметри пошуку в URL. Коли користувач змінює фільтри, ці параметри оновлюються в адресному рядку, що одразу відображає поточний стан фільтрації. Відомо, що використання `URL-params` для керування станом пошуку дає низку переваг: сама адреса веб-сторінки відображає параметри фільтрів, а після перезавантаження чи копіювання URL користувач отримає той самий набір фільтрованих даних. На сервері ці параметри приймаються у відповідному API-ендпоінті і використовуються в `Prisma`-запитах для відбору товарів. При цьому сама сторінка перевантажується і повторно рендериться з урахуванням нових даних – процес дуже схожий на реалізацію пошуку у `Next.js`, де після зміни параметру в URL таблиця даних перерендерюється вже з відфільтрованим результатом. Такий підхід також дозволяє зробити фільтрацію

«спільною» і «відслідковуваною»: по посиланню з усіма параметрами можна по вернутися до тих самих фільтрів або поділитися ними з іншим користувачем.

Кошик

Кошик користувача реалізований як окремий модуль, що спирається на базу даних для збереження елементів. Ідентифікація кошика здійснюється через унікальний токен, збережений у cookie браузера [30]. Коли користувач вперше заходить на сайт, система перевіряє наявність такого токена в cookie. Якщо токена немає (наприклад, це новий або неавторизований користувач), генерується новий випадковий ідентифікатор (наприклад, GUID) і записується в cookie. Цей токен потім використовується при всіх операціях з кошиком (додання товару, оновлення кількості, видалення тощо). Такий підхід дозволяє навіть неавторизованому користувачеві додавати товари в кошик і повертатися до нього пізніше – він практично ідентифікується приватним токеном в cookie.

На клієнті використовується Zustand – легка бібліотека для глобального управління станом на базі React-хуків. Завдяки Zustand створено «store», в якому визначені функції для роботи з кошиком: `fetchCartItems` (отримання поточних товарів з БД), `addCartItem` (додавання товару), `updateItemQuantity` (зміна кількості) та `removeCartItem` (видалення товару). Наприклад, метод `removeFromCart` всередині стору Zustand прибирає товар з масиву кошика в стейті і одразу оновлює цей стан. Алгоритм дій наступний: при будь-якій зміні (додавання/видалення/оновлення кількості) спочатку оновлюється локальний стан (таким чином взаємодія з користувачем відбувається миттєво), а потім відправляється відповідний API-запит на бекенд для синхронізації. Це забезпечує оптимістичний UX: користувач одразу бачить зміни, а за ним відпрацюють серверні операції. У процесі завантаження даних або очікування відповіді API показується індикатор (loader), завдяки чому користувач бачить, що дані оновлюються. Комбінація Next.js і Zustand показує хороший баланс: Zustand зручний для реактивного зберігання стану (наприклад, кількості товарів), а серверний API на Next.js через Prisma гарантує консистентність даних у БД.

Оформлення замовлення

Процес оформлення замовлення починається з показу користувачу форми, де він вводить необхідні реквізити (контактні дані, адреса доставки, вибір способу оплати тощо). Валідація даних форми здійснюється за допомогою Zod: для кожного поля описується схема (наприклад, якщо поле email – то воно має бути у форматі електронної пошти, а ім'я – непорожнім рядком). Ця перевірка дозволяє гарантовано відсіяти некоректні чи неповні дані перед їхньою обробкою (таким чином «відловлювати помилки на ранніх стадіях»). Після того, як користувач підтверджує оформлення, серверна частина створює новий запис (замовлення) у базі даних – тут зазвичай записуються товари, їх кількість, фінальна сума, інформація про покупця тощо. Після успішного створення замовлення система перенаправляє користувача на платіжний шлюз LiqPay для здійснення оплати. Тобто замість відображення успішного повідомлення на сайті відбувається HTTP-редирект на сторінку оплати LiqPay, куди передаються необхідні параметри транзакції (наприклад, сума, ID замовлення, ключі).

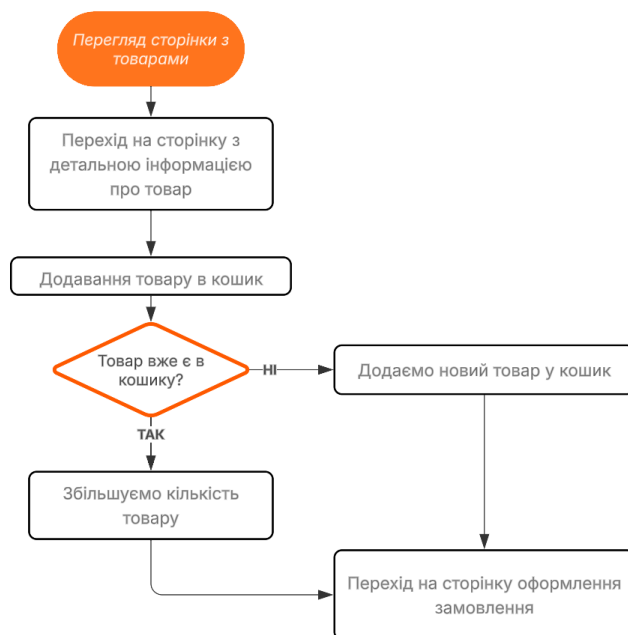


Рисунок 2.4 – Діаграма активностей для оформлення замовлення

Оплата

Інтеграція з платіжним сервісом LiqPay здійснюється за допомогою їхнього API. Після того, як користувач оплачує замовлення на стороні LiqPay, сервіс надсилає на зазначений серверний webhook (callback) HTTP POST із

JSON-повідомленням про результат транзакції. Це повідомлення містить інформацію про статус платежу (success, failure тощо), унікальний ID транзакції і статус обробки. Серверна частина додатку приймає цей callback і першочергово перевіряє цифровий підпис (signature) даних, що приходять, щоб упевнитися, що це справді повідомлення від LiqPay і що дані не було підроблено. Лише після успішної верифікації ми оновлюємо статус замовлення в базі: якщо оплата вдалася, записується статус SUCCEEDED; у разі відмови – CANCELLED або PENDING. У випадках невдалого платежу користувачу, як правило, надсилається автоматичне нагадування на електронну адресу з листом, у якому пропонується повторити платіж (цей механізм повідомлення реалізовано за допомогою RESEND). Важливо, що обробка callback відбувається негайно після отримання від LiqPay: сервер читає JSON-посилання, перевіряє його за схемою (типи полів, наявність статусу) і підтверджує отримання (HTTP 200), щоб LiqPay не надсилала повідомлення повторно. Таким чином, процес оплати повністю контролюється з боку сервісу: він сигналізує про успіх чи невдачу, а додаток оновлює запис замовлення відповідним статусом і виконує додаткові дії (наприклад, відправку email) на основі результатів транзакції.

Профіль

У розділі «Profile» користувач бачить інформацію про свій обліковий запис та історію замовлень. Зокрема, відображаються дані профілю (ім'я, email тощо) із можливістю змінити пароль через спеціальну форму. Зміна пароля здійснюється також із клієнтською валідацією через Zod, яка перевіряє, що новий пароль задовольняє політиці безпеки. Крім того, у кабінеті виводиться історія замовлень: для кожного замовлення показуються список товарів, їхні кількості, ціни й статус виконання (PENDING, SUCCEEDED тощо). Доступ до цієї інформації закритий для неавторизованих користувачів: при завантаженні сторінки кабінету система спочатку виконує перевірку сесії. Як зазначено в документації NextAuth, хук useSession() повертає об'єкт session, і якщо status === "unauthenticated", блокується відображення приватних даних. Це забезпечує, що гості не побачать нічого з особистої інформації або історії замовлень. У разі

спроби зайти до кабінету без авторизації користувач може бути перенаправлений на сторінку входу або побачити повідомлення про необхідність увійти в систему.

Повідомлення

Для покращення користувацького досвіду (UX) у додатку застосовано бібліотеку `react-hot-toast`. Ця легка бібліотека дозволяє просто викликати спливаючі повідомлення (тости) при різних подіях: успіх чи помилка операції, нагадування тощо. Наприклад, після того, як користувач додає товар до кошика, одразу викликається `toast.success('Товар додано до кошика')`, що виводить на екран маленьке віконце з повідомленням. Або після успішної оплати з'явиться повідомлення про вдало завершену транзакцію. Використання `toast` ("Повідомлення") дозволяє викликати нотифікацію з будь-якого місця в коді. Таким чином, замість звичайних `alert` чи звичайного тексту на сторінці ми отримуємо приємні анімовані сповіщення. В документації зазначено, що з допомогою `react-hot-toast` можна легко викликати повідомлення будь-де у застосунку, викликавши функцію `toast(...)`. Це додає системі інтуїтивно-зрозумілі підказки для користувача (наприклад, при успішному додаванні в кошик чи повідомленні про помилку).

Безпека та middleware

Багато аспектів безпеки реалізовано через проміжний рівень (middleware) та налаштування `NextAuth/Next.js`. Зокрема, при кожному запиті до критичних API-ендпоінтів (наприклад, пов'язаних з кошиком) система перевіряє наявність cookie-токену. Якщо токен відсутній, міграція проходить у такий спосіб: на сервері генерується новий унікальний ідентифікатор (за потреби, використовуючи криптографічно сильний генератор GUID), і він записується у cookie відповіді. Таким чином забезпечується, що кожен користувач, навіть якщо він не увійшов в акаунт, має прив'язаний до сесії ідентифікатор кошика. Для таких ідентифікаторів краще використовувати GUID, щоб унеможливити пересилки або підбирання відомих числових ID. Аналогічний механізм відпрацьовується і при спробі аутентифікації: якщо гість намагається перейти на захищену сторінку, перевірка сесії `NextAuth` виявляють, що `status ===`

"unauthenticated", і припиняють завантаження контенту, видаючи помилку чи перенаправляючи на сторінку входу. В системі використовуються також перевірки на сервері: у будь-якому API-ендоїнті можна додатково перевіряти getSession() з NextAuth і відкидати запити, якщо сесія відсутня або вказаний токен недійсний. Також обмежено права дій: лише аутентифікований користувач може, наприклад, переглядати сторінку замовлення або історію покупок. Таким чином, поєднання перевірки cookie-токену, механізмів NextAuth і власних middleware забезпечує, що кожен запит обробляється з урахуванням безпеки і правильності авторизації.

У результаті описані вище модулі взаємодіють наступним чином: аутентифікація користувача гарантує, що приватні дані будуть доступні тільки перевіреним користувачам; API-ендпоінти на базі Prisma надійно отримують і фільтрують дані з БД (товари, кошик, замовлення); підсистема кошика з cookie-токеном дозволяє зберігати товари для будь-якого користувача; а інтеграція з LiqPay і використання системи повідомлень роблять процес покупки зручним і безпечним. Кожен блок реалізовано з урахуванням найкращих практик валідації, обробки помилок та індикації стану, що забезпечує стабільну роботу системи в цілому. Усі компоненти описаної архітектури об'єднані єдиним фреймворком Next.js, який забезпечує узгоджену роботу серверних та клієнтських алгоритмів, дозволяючи реалізувати складні бізнес-процеси інтернет-магазину одягу з високим рівнем надійності.

2.4. Реалізація веб-сайту інтернет-магазину одягу

Головна сторінка інтернет-магазину відображає список категорій одягу з відповідними товарами. Для цього у компоненті Home виконується запит до бази даних за допомогою Prisma. Зокрема, підключено Prisma Client і виконується prisma.category.findMany({ include: { products: { include: { variants: true } } } }), щоб отримати всі категорії разом з їх товарами та варіантами (кольори, розміри). У результаті запит повертає вкладену структуру: кожна категорія містить масив товарів, а кожен товар – масив своїх варіантів (variants).

```

await prisma.category.findMany({
  ...(subcategoryIds.length > 0 && {
    where: {
      subcategories: {
        some: { id: { in: subcategoryIds } },
      },
    },
  })),
include: {
  subcategories: {
    ...(subcategoryIds.length > 0 && {
      where: { id: { in: subcategoryIds } },
    }),
    include: {
      items: {
        where: {
          price: { gte: minPrice, lte: maxPrice },
          ...(colorFilters.length > 0 || sizeFilters.length > 0
            ? {
                variants: {
                  some: {
                    ...(colorFilters.length > 0 && { color: { in:
colorFilters } })),
                    ...(sizeFilters.length > 0 && { size: { in:
sizeFilters } })),
                  },
                },
              : {})),
        },
      },
    },
  },
},
include: { variants: true },
},
},
},
})

```

Для зручності фільтрації використовується функція `findItemsFlat`, яка перетворює цю вкладену структуру в плоский список товарів. Ця функція проходить по всіх категоріях і товарах, формуючи єдиний масив товарів, де кожен об'єкт містить конкретний варіант (комбінацію кольору та розміру) і основні дані товару (назву, ціну тощо). У новій структурі поля `color` і `size` винесені на рівень товару, що значно спрощує їхню обробку. Такий підхід можливий тому, що після виконання запиту Prisma маємо оригінальні вкладені дані, а на рівні самого ORM немає вбудованої підтримки «сплющення» результатів. Відповідно, використовується рішення: отримавши від сервера

вкладену відповідь, вручну «розгортаємо» її – створюємо окремий запис товару для кожного варіанта. Власне, як зауважують розробники Prisma, прямого методу для отримання плоского списку з вкладених реляцій немає, тому виникла необхідність реалізувати це самостійно після вибірки.

На головній сторінці передбачено фільтрацію товарів за категоріями (підкатегоріями), кольорами, розмірами та ціновим діапазоном. При зміні будь-якого фільтра відповідні параметри додаються до URL у вигляді query-параметрів. Наприклад, після вибору розміру «S», кольору «WHITE» та певної підкатегорії адресний рядок може приймати вигляд `?sizes=S&colors=WHITE&subcategories=1` (рис 2.5.).



Рисунок 2.5 – Приклад query параметрів фільтрації в URL

Такий підхід означає, що стан фільтрації відображено в URL сторінки – це дозволяє зробити посилання закладкою з поточним станом і легко поділитися ним. Крім того, передавання параметрів у URL вигідне з точки зору серверної логіки: пошук включає ці параметри одразу в запит до сервера, що спрощує отримання саме тих даних, які повинні бути відображені при первинному рендерингу.

Інтерфейс вибору фільтрів реалізовано в окремому клієнтському компоненті `Filters`. У ньому є поля введення для цінового діапазону (наприклад, два числові поля) та набори чекбоксів для категорій, кольорів і розмірів. Коли користувач взаємодіє з цими елементами (відмічає галочку чи змінює ціну), відпрацьовують обробники, які формують новий рядок запиту та оновлюють адресу сторінки без перезавантаження. Для цього використовується хук `useSearchParams` – він повертає об'єкт `URLSearchParams` із поточними параметрами. Частина компоненту `<Filters/>`:

```
<div className=" mt-5 border-y border-y-neutral-100 py-6 pb-7">
  <p className="mb-3 font-bold">Price range:</p>
  <div className="flex mb-5 gap-3">
    <Input
```

```

        type="number"
        placeholder="0"
        min={0}
        max={3000}
        onChange={(e) => filters.setPrices("priceFrom",
Number(e.target.value))}
        value={String(filters.prices.priceFrom || 0)}
      />
      [...]
    </div>
    [...]
  </div>
  <CheckboxFiltersGroup
    name="itemsSubcategories"
    className="mb-5 mt-5"
    title="Subcategory"
    items={defaultSubcategories}
    defaultItems={defaultSubcategories.slice(0, 3)}
    loading={loading}
    onClickCheckbox={filters.toggleSubcategory}
    selectedIds={filters.selectedSubcategories}
  />

```

Хук `useRouter` дає можливість програмно змінити адресу: можна, наприклад, викликати `router.push(pathname + '?' + newQuery)`, щоб додати або змінити потрібні параметри. Після оновлення URL `Next.js` перерендерює сторінку: компонент `Home` отримує нові параметри запиту й відображує вже відфільтровані товари.

Товари на сторінці групуються і виводяться за допомогою компонента `ItemsGroupList`. Цей компонент приймає плоский список товарів (отриманий із `findItemsFlat`, після застосування фільтрів) і групує їх за категоріями. Внутрішньо він проходить по кожній категорії, що залишилася, та виводить рядок товарів у вигляді сітки) для цієї категорії. При цьому на сторінці виводяться лише ті категорії, в яких є хоч один товар після фільтрації; якщо фільтр повернув порожній набір для певної категорії, `ItemsGroupList` просто пропускає цю групу. Таким чином користувач бачить на сторінці тільки релевантні категорії з доступними товарами, а не порожні розділи.

Важливою частиною реалізації є використання механізму `React Suspense` для клієнтського компонента `Filters`. Оскільки `Filters` може потребувати завантаження додаткових даних (наприклад, списку усіх підкатегорій чи значень

фільтрів), ми обгортаємо його в `<Suspense>` з резервним відображенням (з індикатором завантаження). `Suspense` дозволяє «призупинити» рендеринг дочірнього компоненту, що виконує асинхронну операцію, та показати заданий `fallback` (завантажувач) доки дані не надійдуть. Після завершення завантаження `Filters` «розмахується» у зворотному порядку – замість фейкового UI з’являються реальні елементи фільтрації. Це підвищує UX: користувач бачить “loader” під час очікування даних.

Для пришвидшення початкового рендеру та покращення SEO застосовано серверне передрендерення (SSR). Компонент `Home` реалізовано як асинхронний серверний компонент, і `Next.js` виконує його на сервері до відправки HTML клієнту. Це означає, що всі необхідні дані (категорії, товари) завантажуються ще на сервері, і браузер одразу отримує готову HTML-сторінку з вмістом. Такий підхід значно зменшує час, потрібний для відображення першого контенту, – користувач побачить наповнену сторінку значно швидше. Крім того, оскільки сторінка приходить вже у вигляді повного HTML без необхідності виконувати додатковий JavaScript, це позитивно впливає на SEO. Серверне рендерення забезпечує наявність контенту в HTML на момент завантаження і таким чином робить сторінку «читабельною» для пошукових систем. Таким чином використання SSR дозволяє поєднувати швидкий перший рендер та оптимізований індексування сайту.

У результаті поєднання цих компонентів і механізмів забезпечує повноцінну роботу головної сторінки інтернет-магазину одягу. Вихідні дані витягуються за допомогою `Prisma`, формуються у зручну плоску структуру, після чого застосовуються гнучкі фільтри, синхронізовані з URL. Інтерфейс фільтрації виконано через `Filters`, який оновлює URL та забезпечує налаштування без перезавантаження. Компонент `ItemsGroupList` відповідає за виведення згрупованих за категоріями товарів, а `Suspense` і `SSR` гарантують, що користувач відразу побачить актуальний контент і що сайт добре індексується пошуковими системами.

На головній сторінці магазину всі доступні товари відображаються за допомогою компонента `ItemsGroupList`. Цей компонент отримує масив товарів і використовує метод `map` для створення карток товарів – компонентів `ItemCard` – для кожного елемента списку. Кожна `ItemCard` містить зображення товару, його назву, ціну та перелік доступних кольорів (якщо їх більше одного), а також кнопку «Add». Такий підхід з компонентами дозволяє гнучко формувати інтерфейс: змінюючи структуру `ItemCard`, ми автоматично міняємо відображення усіх товарів. Наприклад, для відображення списку ми можемо використати `items.map(item => <ItemCard key={item.id} item={item} />)`, де `ItemCard` отримує необхідні дані (зображення, назву, ціну, кольори). Верстка компоненту `ItemsGroupList`:

```
<div ref={intersectionRef} className={cn("mb-14", className)}
id={`_${title}`}>
  <Title text={title ?? "Категорія"} size="lg" className="font-
extrabold mb-5" />
  <div className={cn("grid grid-cols-3 gap-[50px]",
listClassName)}>
    {filteredItems.map((item) => (
      <ItemCard
        key={item.id}
        id={item.id}
        title={item.title}
        image={item.image}
        price={item.price}
        description={item.description}
        rating={item.rating}
        numReviews={item.numReviews}
        subcategoryId={item.subcategoryId}
        size={item.sizes}
        color={item.colors}
      />
    ))}
  </div>
</div>
```

У `ItemCard` вся логіка цілком клієнтська – це звичайний React-компонент, який приймає властивості товару (`image`, `name`, `price`, `colors`) та рендерить картинку, текст і кнопку. Зображення і назва товару поміщені у компонент `<Link>` з посиланням на сторінку товару `/item/[id]`. Таке посилання побудоване за допомогою динамічного маршруту Next.js: файл `pages/item/[id].js` означає, що сегмент `id` буде підставлений із URL при переході. Використання компонента

Link з Next.js дозволяє здійснювати переходи клієнтською стороною без повного оновлення сторінки, подібно до односторінкового додатка. Таким чином, при кліку на картинку чи назву товару браузер швидко навігує на сторінку `/item/123`, не перезавантажуючи весь сайт.

Сторінка конкретного товару `/item/[id]` рендериться на сервері при кожному запиті. Для цього у Next.js експортується асинхронна функція `getServerSideProps`, яка витягує необхідні дані перед рендерингом. Так, Next.js за допомогою `getServerSideProps` забезпечує `server-side rendering` і відображає сторінку на підставі даних з бекенду. У цій функції ми отримуємо переданий параметр `id` (через `context.params.id`) і звертаємося до бази даних. У нашому випадку ми використовуємо Prisma Client для доступу до бази. Запит на базу має вигляд як виклик `prisma.item.findUnique({ where: { id }, include: { variants: true, subcategory: true, category: true } })`. Тут ключовим є опція `include`: вона дозволяє витягнути пов'язані записи (родинні таблиці) одним запитом. Як зазначено в документації, ключова особливість Prisma – це можливість отримувати пов'язані дані, наприклад товари разом із їхніми підкатегоріями чи варіантами, використовуючи `include`. Таким чином, ми одночасно отримуємо з бази сам товар (його назву, опис тощо), усі варіанти (`variants`), а також пов'язані сутності `subcategory` та `category`.

Отримавши масив варіантів товару, далі ми формуємо списки унікальних доступних кольорів та розмірів. Якщо кожен варіант має поля `color` та `size` і властивість наявності (`quantity > 0`), то пройдемося по списку варіантів і відфільтруємо унікальні значення кольорів і розмірів за допомогою структури `Set` або методу `filter/map`. При цьому ми відкидаємо ті варіанти, у яких немає наявності (`quantity = 0`). З результату дістається набір кольорів і розмірів, що реально існують у товара. Це гарантує, що на сторінці товару користувач побачить тільки ті комбінації кольору і розміру, які є в наявності, а не всі гіпотетичні. Отримані дані передаються у `props` компоненту сторінки товару.

На сторінці товару відображається компонент `ItemForm`, який дає змогу користувачу обрати параметри (колір і розмір) товару. Це можуть бути селектори

або кнопки для вибору кольору та розміру. Вибрана комбінація одразу зберігається у локальному стані компонента (це зроблено за допомогою React-хука `useState`). Створено два стейти `const [selectedColor, setSelectedColor] = useState(defaultColor)` та `const [selectedSize, setSelectedSize] = useState(defaultSize)`, де `defaultColor` і `defaultSize` – перші доступні опції з отриманих масивів. Використання `useState` дає змогу зберігати і змінювати обрану комбінацію всередині компонента. Валідація окремо не потрібна, бо за замовчуванням завжди обрані дефолтні (існуючі) варіанти, і користувач не може обрати недоступну комбінацію. Коли користувач змінює вибір (наприклад, клікає на інший колір), відповідна функція `setSelectedColor` оновлює стан, і компонент повторно рендериться з новим вибором. Такий підхід (локальний state) гарантує, що інформація про обрані параметри завжди актуальна всередині компонента `ItemForm` та не потребує кожного разу звернення до сервера.

Після того, як користувач вибрав параметри, він натискає кнопку «Add» на картці товару або формі. При цьому викликається метод `addCartItem()` із глобального сховища стану кошика, реалізованого через бібліотеку `Zustand`. `Zustand` – це хук-бейзована бібліотека для управління глобальним станом у `React`. Вона дозволяє створити глобальне сховище, яке можна використовувати у будь-яких компонентах без використання контекстних провайдерів. Стор створюється за допомогою функції `create`, в якій описуються початкові поля стану (`items`, `loading`, `error`, `totalAmount` тощо) і методи (`actions`) для зміни цього стану. У сховищі можна визначено функції `addCartItem`, `removeCartItem`, `updateItemQuantity`. При цьому у коді компонентів ми використовуємо хук `useCartStore`, який повертає необхідні дані або методи зі сховища.

Метод `addCartItem` у сховищі працює наступним чином: він отримує дані про товар (`id`, колір, розмір тощо) і перевіряє, чи вже існує такий товар з тими ж параметрами у масиві `items`. Якщо такий товар вже є, він просто збільшує його поле `quantity` на одиницю, а якщо ні – додає новий об'єкт товару у масив `items`. Оскільки `Zustand` за своєю природою оновлює стан з іммубельними оновленнями, то він використовує функцію `set` для модифікації стану. Згідно з

документацією, `set` приймає функцію оновлення, яка отримує старий стан і повертає об'єкт з оновленими полями. Наприклад, якщо у шаблоні сховища є поле `bears`, то можна було б зробити `increasePopulation: () => set(state => ({ bears: state.bears + 1 })))`. Аналогічно тут `addCartItem` використовує `set` для зміни `items`, `totalAmount` та інших полів залежно від логіки (перераховує суму чи кількість товарів).

Після того, як елемент успішно додано в кошик (або його кількість збільшено), на сторінці показується тост-повідомлення про успішне додавання. Таке повідомлення (`toast`) – це невеличке спливаюче віконце в верхній частині екрану, яке коротко інформує користувача. Воно не блокує інтерфейс і швидко зникає через кілька секунд. Як відзначають у документації, тости використовуються для того, щоб повідомити користувачу про результат якоїсь дії (успіх чи помилку), не перериваючи роботу з додатком. В нашому випадку це повідомлення «Товар додано у кошик», яке відображається після натискання «Add».

Весь стан кошика у додатку є глобальним і централізованим в одному місці – у Zustand-сховищі. Приклад структури `useCartStore` може включати такі поля: `items` – масив об'єктів товарів у кошику, `loading` – прапорець стану завантаження (наприклад, під час синхронізації з сервером), `error` – рядок або код помилки, та `totalAmount` – загальна сума вартості усіх товарів у кошику. Крім полів, визначені методи: `fetchCartItems` (що може запитувати початковий стан кошика з API чи cookies), `addCartItem`, `removeCartItem`, `updateItemQuantity` – типові дії для редагування вмісту кошика. У нашій логіці, наприклад, `fetchCartItems` може викликатися при першому завантаженні сторінки, щоб ініціалізувати стан із збережених даних (може брати дані з cookies, якщо користувач не авторизований).

Важливо, що для роботи з кошиком не потрібно проходити авторизацію: при додаванні в корзину формується якась ідентифікація сесії на рівні браузера (через cookies), і завдяки цьому гість може додавати товари без входу в акаунт. Перезавантаження чи закриття вкладки не зітре стан кошика доти, поки дані

зберігаються у cookie. Таким чином, навіть незареєстрований користувач має «сесію» завдяки механізму збереження стану на клієнті.

Оскільки ми оновлюємо кошик локально, відбувається плавний UX: сторінка не перезавантажується при кожній дії, а зміни відображаються миттєво. Next.js забезпечує клієнтську навігацію між сторінками і можливість виклику `setState` без фетчингу з API щоразу. Сам компонент `ItemCard` або форма просто викликають методи глобального сховища (наприклад, `addCartItem`), а React перерендерує тільки ті компоненти, які підписані на відповідні дані (наприклад, індикатор кількості товарів у хедері). У результаті робота з кошиком здається швидкою і односторінковою, хоча на сервері дані зберігаються завдяки Prisma. Цей підхід (керування всім станом через глобальний сховищний шар) дозволяє уникнути зайвих запитів до сервера при кожному кліку — зміни йдуть через Zustand і відображаються миттєво.

Компонент `CartDrawer` відповідає за відображення кошика товарів у вигляді бічної панелі, що висувається справа по кліку користувача. Для побудови інтерфейсу використовуються UI-компоненти бібліотеки `shadcn` [16] – зокрема `Sheet`, `SheetTrigger`, `SheetContent`, `SheetHeader`, `SheetFooter` та інші. Компонент `Sheet` є надбудовою над діалоговим вікном Radix UI [27] і призначений для висування додаткового вмісту на екран. Кнопка кошика, обгорнута в `<SheetTrigger>`, відкриває цю панель, а сам список товарів відображається всередині `<SheetContent>`.

Після відкриття `CartDrawer` користувач бачить заголовок (через `<SheetHeader>` і `<SheetTitle>`), де може бути вказана назва кошика або інший текст. Під ним розташовано список усіх товарів, доданих до кошика. Кожен товар відображається окремим блоком (компонент `CartDrawerItem`): у ньому показані назва, мініатюрне зображення товару, обрані параметри (колір і розмір), ціна та кількість одиниць. Іншими словами, у блоці кожного товару показані всі необхідні деталі: назва, зображення, кількість, ціна і обрані опції (колір/розмір). Користувач може змінювати кількість кожного товару або видаляти його з кошика.

Кнопки «+» і «-» у блоці товару відповідають за збільшення або зменшення його кількості. При натисканні на ці кнопки викликається обробник `onClickCountButton`, який через функцію `updateItemQuantity` у сховищі `Zustand` оновлює відповідну кількість товару. При цьому загальна сума кошика перераховується автоматично. Зокрема, якщо кількість товару більше одиниці, при зменшенні вона просто зменшується на 1, але не може стати менше одиниці – кнопка блокується.

Для видалення товару передбачена окрема кнопка «видалити» (у вигляді іконки смітника) у блоці товару. Натискання її викликає метод `removeCartItem`, який прибирає цей товар із глобального стану кошика. Наприклад, якщо товар був в єдиному екземплярі в кошику, функція повністю видаляє його з масиву товарів. Після цього підсумкова сума і вміст кошика оновлюються одразу — користувач відразу бачить зменшену суму і відсутність товару.

У футері (`<SheetFooter>`) `CartDrawer` відображається підсумкова сума `totalAmount` — загальна вартість усіх товарів з урахуванням їхніх кількостей. Її обчислюємо як суму `item.price * item.quantity` по всіх товарах. Поруч із сумою знаходиться кнопка оформлення замовлення (з текстом «Make an order»), яка веде користувача на сторінку `/checkout`. При натисканні на цю кнопку може з'являтися індикатор завантаження, що повідомляє про перенаправлення.

Якщо кошик порожній (немає жодного товару), `CartDrawer` показує запасний контент замість списку – іконка пустого кошика [17], повідомлення «Cart is empty» та кнопка переходу до каталогу.

У рамках розробки інтернет-магазину одягу було виділено окремий маршрут для сторінки оформлення замовлення із шляхом `/checkout`, використовуючи можливості `Next.js route groups`. За допомогою назви папки в дужках (`app/(checkout)/checkout/page.tsx`) створюється спеціальна група маршруту, яка не додає додаткового сегменту до URL, але дозволяє застосувати власний макет (`layout`) саме для оформлення замовлення. Таким чином, при переході на `/checkout` відображається контент із папки (`checkout`) із заданим

layout, що дозволяє ізолювати оформлення замовлення від інших розділів сайту і, наприклад, вимкнути загальну навігаційну панель чи змінити стилі. На сторінці /checkout розміщено форму для введення персональних даних покупця: ім'я, прізвище, email, телефон, адресу та вільний коментар. Для керування станом цієї форми використовується бібліотека React Hook Form, яка є легкою та ефективною для валідації форм у React. З її допомогою поля реєструються через функцію <Controller>, а відправка обробляється через метод handleSubmit. Щоб покращити реюзабельність і передати можливість використання методів форми у вкладених компонентах, над формою обгортається FormProvider з усіма методами useForm – це дозволяє дочірнім елементам отримати доступ до контексту форми. Валідація полів здійснюється за допомогою бібліотеки Zod, що дозволяє декларативно описати схеми для даних форми і забезпечує типобезпечну перевірку. Zod характеризують як “TypeScript-first validation library”, що дозволяє визначати схеми для валідації будь-яких структур даних, від простих рядків до вкладених об'єктів. У поєднанні з React Hook Form та використанням zodResolver, це дає можливість легко перевіряти введені значення на відповідність заданим правилам. Комбінація Zod і React Hook Form робить реалізацію форм керованою та масштабованою: Zod задає жорсткі правила для даних, а RHF [11] обробляє управління станом і відображає помилки. Таким чином, для поля «email» задаємо схему z.string().email(), а для «телефону» – патерн перевірки, і під час спроби відправки React Hook Form використовуватиме результати перевірки Zod для відображення повідомлень про помилки. Якщо користувач введе недійсні дані, об'єкт errors з React Hook Form надасть відповідні повідомлення, які виведуться поруч із полями. Якщо покупець вже авторизований у системі, при завантаженні сторінки виконується запит getMe(), який отримує збережені дані користувача (зокрема email, ім'я та прізвище). Ці значення передаються в React Hook Form як defaultValues або через метод reset, щоб попередньо заповнити форму. Бібліотека RHF підтримує задання початкових значень навіть у асинхронному режимі, тому можна отримати дані за API і одразу передати їх у параметр defaultValues хука useForm.

Таким чином ім'я, прізвище та email авторизованого покупця автоматично з'являються в полях форми, що полегшує і прискорює процес оформлення замовлення для користувача. Поруч із формою розташований компонент CheckoutSidebar, який відображає підсумкову суму до сплати. У цьому блоці виводиться загальна вартість товарів у кошику плюс фіксована вартість доставки (150 грн). Нижче відображається кнопка «Continue to Payment» для переходу до оформлення оплати. При натисканні на цю кнопку виконується метод handleSubmit, який передає зібрані дані до функції createOrder(). У цій функції дані замовлення (введені поля форми, сума та інші деталі) надсилаються на сервер чи зберігаються у базу даних: створюється новий запис у таблиці Order зі статусом PENDING, куди записуються всі поля замовлення. Після успішного створення такого запису очищується вміст кошика користувача, аби повторно не оплатити те саме замовлення. Одночасно з цим відправляється електронний лист з деталями замовлення на пошту покупця, що підтверджує факт оформлення. Далі на клієнтській стороні отримується відповідь сервера з посиланням paymentUrl (через LiqPay API для оплати), і відбувається редирект користувача на цей URL для здійснення платіжної транзакції. Якщо під час оформлення замовлення сталася помилка (наприклад, проблема з мережею чи внутрішня помилка серверу), користувачу на екрані показується сповіщення (toast) з повідомленням про невдачу, щоб він міг зрозуміти, що треба спробувати ще раз. Для покращення користувацького досвіду (UX) на сторінці передбачені додаткові візуальні підказки та обмеження. Зокрема, поки йде відправка форми, кнопка «Continue to Payment» блокується, щоб уникнути подвійного надсилання даних. За найкращою практикою, при сабміті відключаємо кнопку та показувати індикатор зайнятості, щоб користувач зрозумів, що триває обробка запиту. У разі помилки валідації відповідних полів поруч із полем виводяться чіткі повідомлення (які формуються завдяки Zod і RHF), а при збої оформлення сам процес супроводжується повідомленням у форматі toast. Таким чином, користувач бачить підказки про те, які поля потрібно виправити, і відразу бачить, коли замовлення успішно відправлено чи сталася проблема. Загалом, описана

реалізація сторінки оформлення замовлення враховує окрему маршрутизацію та макет Next.js, надійну клієнтську валідацію з React Hook Form і Zod, а також забезпечує плавний UX за рахунок відображення статусу відправки, блокування кнопок та повідомлень про помилки. Такий підхід спрощує підтримку коду і робить оформлення замовлення зручним для користувача.

Висновки до другого розділу

Таким чином, архітектура сторінки товару побудована на компонентах React і Next.js: головна сторінка показує список ItemCard з товарами, клік веде на динамічний маршрут /item/[id]. На сторінці товару дані завантажуються серверно через Prisma (отримуємо товар, його варіанти, під- та категорію), а вибір кольорів та розмірів відображається в ItemForm. Обрана комбінація зберігається локально (useState), і при натисканні «Add» викликається addCartItem() із Zustand. Сховище Zustand тримає масив товарів, загальну суму, стан завантаження та помилки і надає методи для оновлення. Завдяки цьому UX залишається плавним: додавання товару відбувається без перезавантаження сторінки (Next.js Link робить клієнтську навігацію), а користувачу миттєво показується підтвердження у вигляді тосту.

З такою системою навіть неавторизований гість може додавати товари до кошика без проблем, бо стан підтримується на рівні клієнта (через cookies чи localStorage), а не вимагає логіну. Всі ці рішення забезпечують швидку та зручну роботу користувача з інтернет-магазином.

РОЗДІЛ 3. ІНТЕРФЕЙС ТА ПОРЯДОК РОБОТИ З ВЕБ-ДОДАТКОМ ІНТЕРНЕТ-МАГАЗИНОМ ОДЯГУ

3.1. Порядок налаштування параметрів системи

Перед початком експлуатації інтернет-магазину необхідно виконати ряд підготовчих налаштувань, які забезпечать коректну роботу серверної та клієнтської частини, а також інтеграцію з підсистемою аутентифікації та платіжним шлюзом. У першу чергу слід сконфігурувати середовище виконання Node.js: встановити версію не нижче 18 LTS, а також пакетний менеджер npm або yarn. Після цього у корені проєкту розміщується файл .env, у якому зберігаються всі критичні параметри середовища. Серед них — DATABASE_URL, який вказує на підключення до PostgreSQL (наприклад, postgresql://user:password@host:port/dbname), та POSTGRES_URL_NON_POOLING для виконання операцій без пулінгу. Саме з цих змінних Prisma зчитує налаштування бази даних, а їхня правильність напряду впливає на те, чи зможе клієнт працювати з БД: для початку потрібно запустити команду npm run prisma migrate deploy, яка виконає всі накопичені міграції схеми, і потім перевірити, що таблиці User, Category, Item та інші створені без помилок.

Далі необхідно розгорнути схему автентифікації. У файлі next-auth.d.ts або у секції конфігурації NextAuth у authOptions задаються провайдери (GitHub, Google), а також Credentials Provider для email-пароллю. У .env слід прописати NEXTAUTH_URL=https://your-domain.com, а також секретну фразу NEXTAUTH_SECRET для підпису сесій. Після збереження цих значень сервіс NextAuth автоматично підхоплює їх при старті.

Особливу увагу слід приділити інтеграції з платіжним шлюзом LiqPay. У .env додаються поля LIQPAY_PUBLIC_KEY, LIQPAY_PRIVATE_KEY, які використовуються як у клієнтському запиті на оплату, так і при генерації хешів на сервері. Для безпеки краще зберігати ключі виключно в середовищі, не передаючи їх у фронтенд. У коді функції createOrder вони зчитуються з

процесного середовища (`process.env.LIQPAY_PUBLIC_KEY`) і передаються LiqPay API. Щоб перевіряти `callback`, слід додатково вказати `LIQPAY_CALLBACK_URL` у налаштуваннях сервісу.

Після налаштування `.env` виконується встановлення залежностей командою `npm install` або `yarn`. Потім слід запустити локальний сервер розробки через `npm run dev`. У консолі повинні з'явитися повідомлення про успішне підключення до бази даних і прослуховування HTTP-порту (за замовчуванням 3000). Переходом на `http://localhost:3000` перевіряють, чи коректно відображаються головна сторінка, логін та фільтрація товарів.

Налаштування фронтенд-частини включає збирання TailwindCSS. У проєкті присутній файл `tailwind.config.js`, у якому визначені шляхи до всіх папок з JSX/TSX-файлами. Щоб побачити зміни стилів, достатньо відредагувати CSS та перезапустити збірку. Для оптимізації клієнтської частини на продакшн необхідно виконати `npm run build`, що запустить Next.js із Turbopack і створить оптимізовані збірки JavaScript і CSS.

Після локального тестування систему розгортаємо на хостингу (наприклад, Vercel). Перед деплоєм у веб-інтерфейсі Vercel задаються всі ті ж змінні середовища (`DATABASE_URL`, `NEXTAUTH_SECRET` тощо). Vercel автоматично виконає міграції Prisma та запустить сервер з SSR. Для забезпечення безпеки рекомендується встановити режим HTTPS та перевірити, що сторінка входу/реєстрації, оплати та `callback` LiqPay працюють через захищений протокол.

Отже, послідовність налаштувань виглядає так: підготовка `.env` з усіма ключами та URL, застосування Prisma-міграцій, встановлення залежностей, налаштування NextAuth, інтеграція LiqPay, збірка TailwindCSS, локальне тестування, деплой на продакшн і визначення прав доступу. Дотримання цього порядку гарантує, що інтернет-магазин запрацює без помилок, а всі підсистеми — від БД до платіжного шлюзу та авторизації — будуть коректно конфігуровані.

3.2. Структура веб-додатку інтернет-магазин одягу

Розглянемо інтерфейс розробленого інтернет-магазину. При відкритті веб-сайту користувач спочатку бачить головну сторінку (див. рис. 3.1.), на якій відображаються всі товари. У верхній частині сторінки розміщується навігаційне меню: поле для пошуку товару, кнопка авторизації/переходу на сторінку профілю користувача і меню корзини, що відображає основну інформацію по загальній сумі корзини і кількості елементів в корзині.

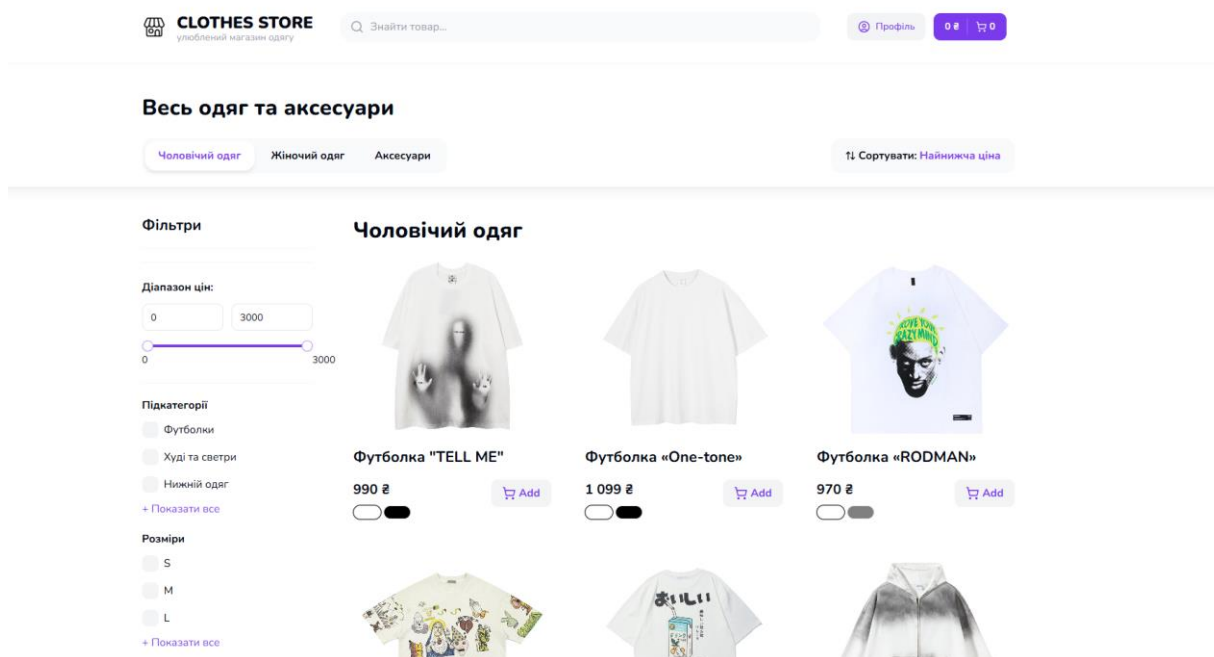


Рисунок 3.1 – Головна сторінка, верхня частина сайту

Під навігаційним меню представлено компонент, який було названо ТорВаг — він відповідає за швидкий доступ до підрозділів каталогу та керування порядком відображення товарів (див. рис. 3.2.). Горизонтально розташований ряд із трьох кнопок-категорій, що дозволяє одним клацанням перемикатися між чоловічим, жіночим та аксесуарним асортиментом. Активна категорія підсвічується фіолетовим кольором, а сама панель «плаває» над контентом, залишаючись завжди доступною під час прокрутки списку товарів.

Праворуч від кнопок категорій знаходиться кнопка сортування. Вона представлена іконкою зі стрілкою і текстом «Сортувати», що одразу повідомляє про поточний порядок упорядкування. При натисканні відкривається невеликий спадний попап (див. рис. 3.3.), у якому користувач може обрати інші варіанти

сортування (наприклад, за рейтингом або за зростанням/спаданням ціни). Після вибору нового критерію попап ховається, а список товарів автоматично ререндериться у відповідності з обраним порядком. У такий спосіб ТорВаг забезпечує зручну навігацію по каталогах і швидке сортування, зберігаючи компактність і мінімалізм інтерфейсу.

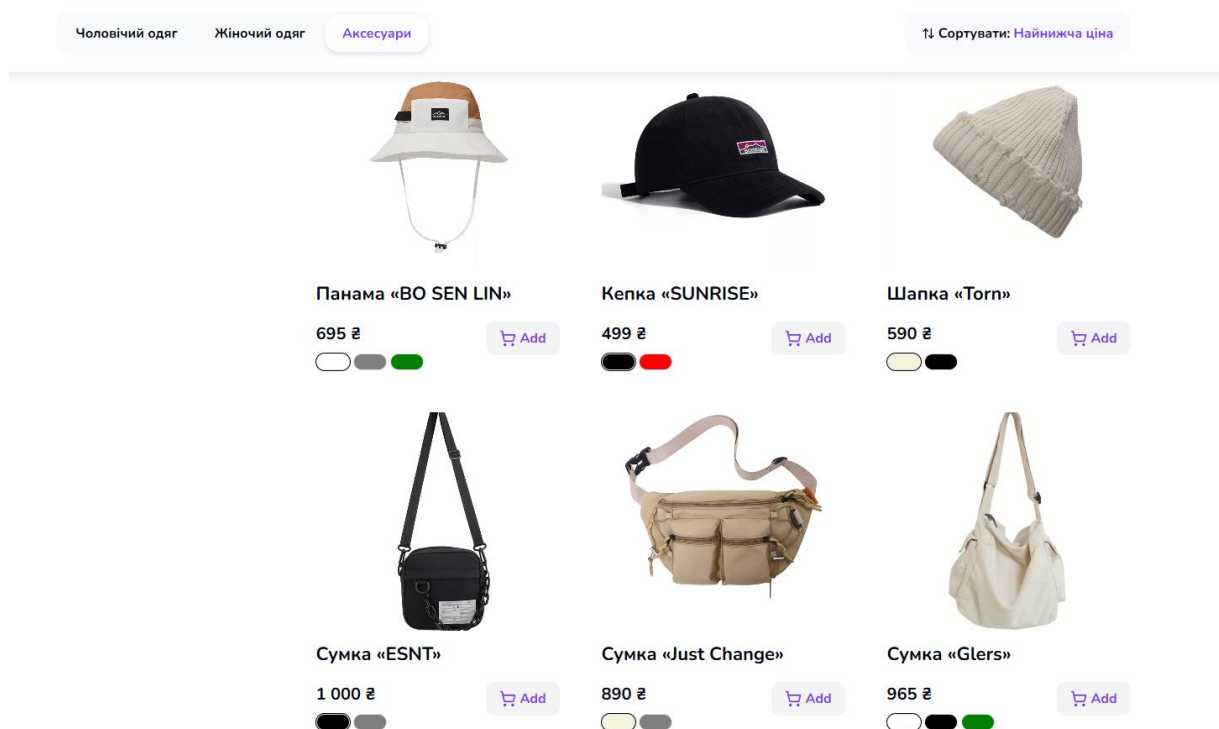


Рисунок 3.2 – Компонент ТорВаг з застосованою категорією і обраним режимом сортування

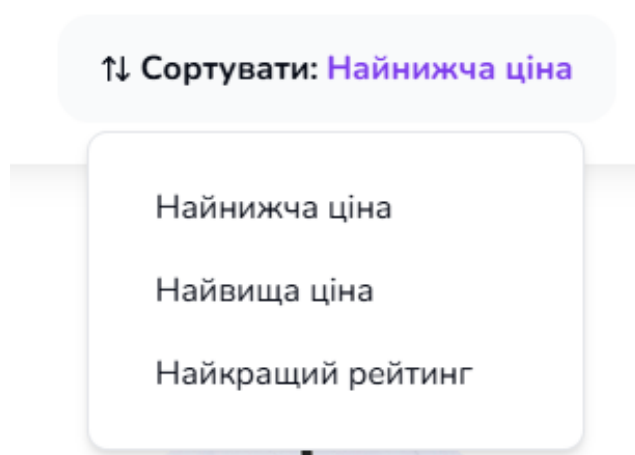


Рисунок 3.3 – Попап сортування

Зліва від основного блоку товарів розташовано панель фільтрів, яка дозволяє користувачеві швидко звузити асортимент за потрібними параметрами. Найвище стоїть заголовок «Фільтри». Першим елементом є діапазон цін: два числові поля, у які можна ввести мінімальне та максимальне значення, а під ними розміщено інтерактивний слайдер із двома повзунками (див. рис. 3.4.).

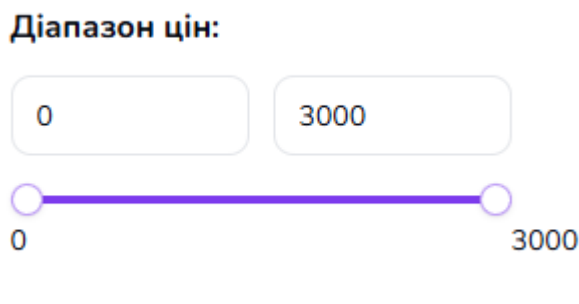


Рисунок 3.4 – Блок для фільтрації по ціні з інтерактивним слайдером

При перетягуванні повзунків або зміні чисел у полях підсумкове значення діапазону миттєво відображається як підказка під слайдером. Далі йде розділ «Підкатегорії» (див. рис. 3.5.), де спочатку показано кілька груп товарів — «Футболки», «Худі та светри», «Нижній одяг» — кожна з яких має власну чекбокс-позначку.

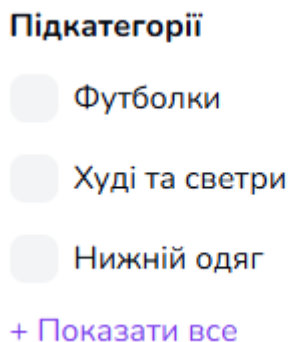


Рисунок 3.5 – Блок для фільтрації за підкатегоріями

Якщо користувачу потрібно розгорнути повний перелік підкатегорій, достатньо натиснути на посилання «Показати все» (див. рис. 3.6.). При розгортанні блоку користувач отримує доступ до всіх елементів, за якими можна здійснити фільтрацію, має можливість скористатись полем для пошуку всередині кожного фільтра і згорнути блок назад у вихідне положення. Аналогічним чином

реалізовано розділ «Розміри» (див. рис. 3.7.): за замовчуванням видно найпоширеніші S, M, L, а для повного огляду доступних розмірів служить кнопка «Показати все».

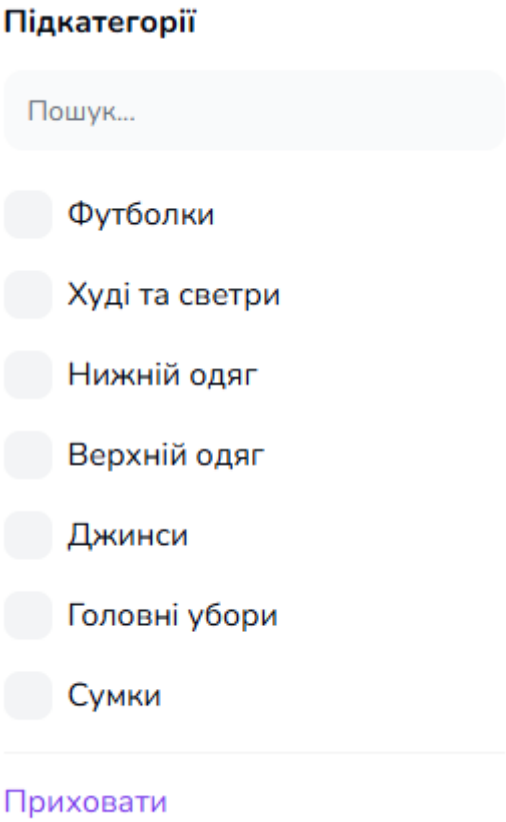


Рисунок 3.6 – Розгорнутий блок фільтрації за підкатегоріями

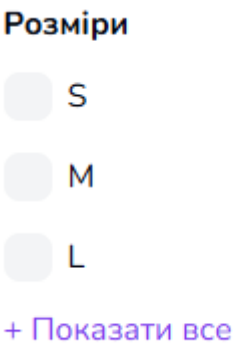


Рисунок 3.7 – Блок фільтрації за розмірами

У нижній частині панелі знаходиться блок «Кольори» (див. рис. 3.8.). На екрані відразу видно базові кольори: білий, чорний, сірий. При розгортанні блоку так само є можливість скористатись пошуком, переглянути всі доступні розміри або приховати блок.

Кольори

☐ WHITE

☐ BLACK

☐ GREY

[+ Показати все](#)

Рисунок 3.8 – Блок фільтрації за кольорами

Після вибору параметрів або переміщення повзунків фільтрація відбувається автоматично, оновлюючи список товарів без перезавантаження сторінки. Такий інструментальний набір дозволяє користувачу гнучко й швидко знаходити саме ті товари, які йому потрібні, одночасно даючи можливість детального налаштування під будь-які критерії пошуку.

При натисканні на кнопку «Увійти» у верхній частині сайту з’являється модальна форма (див. рис. 3.9.), яка динамічно перемикається між режимами «Увійти» та «Зареєструватися» (див. рис. 3.10.), не вимагаючи переходу на окрему сторінку.

Рисунок 3.9 – Форма авторизації

Рисунок 3.10 – Форма реєстрації

У режимі входу два великі поля з фіолетовою рамкою приймають адресу електронної пошти та пароль. Під ними розташована широка фіолетова кнопка «Увійти», натискання якої відправляє форму через NextAuth. Далі — дві сірі кнопки з іконками GitHub та Google, які дозволяють авторизуватися через зовнішні провайдери. Внизу — лінк «Зареєструватися»: клік по ньому перетягне користувача у режим реєстрації, зберігаючи ту саму модальну форму.

У режимі реєстрації заголовок змінюється на «Зареєструватися», а форма розширюється: першим полем іде email, другим — повне ім'я, третім і четвертим — пароль і його підтвердження. Кнопка «Зареєструватися» виконана тим самим фіолетовим кольором, що робить інтерфейс цілісним. Під нею знову йдуть кнопки GitHub та Google, щоб можна було зареєструватися через ці сервіси, а зовсім унизу — лінк «Увійти», що повертає до форми входу.

Усі поля марковані зірочкою «*», а невірно введені дані підсвічуються червоним контуром із короткою підказкою-помилкою. Таким чином, користувач має під рукою всю необхідну функціональність для входу або створення облікового запису в одному компактному й зрозумілому інтерфейсі.

CLOTHES STORE

улюблений магазин одягу

🔍

Знайти товар...

👤 Профіль

0 ₴

🛒 0

Особисті дані | Korotkyi Nikita

Email *

zip211_kmo@student.ztu.edu.ua

×

Повне ім'я *

Korotkyi Nikita

×

Новий пароль *

Повторіть пароль *

Зберегти

Вихід

Замовлення 21

1 червня 2025 р. о 20:20

Статус оплати: Succeeded

Товари:

Zip-худи «SOUL QUALITY»
Колір: WHITE, Розмір: XS

2 × 1 750 ₴
3 500 ₴

Футболка «One-tone»
Колір: BLACK, Розмір: L

3 × 1 099 ₴
3 297 ₴

Загальна сума: 6 797 ₴

Замовлення 19

24 травня 2025 р. о 11:01

Статус оплати: Pending

Товари:

Футболка "TELL ME"
Колір: BLACK, Розмір: XXL

5 × 990 ₴
4 950 ₴

Футболка «One-tone»
Колір: BLACK, Розмір: S

1 × 1 099 ₴
1 099 ₴

Загальна сума: 6 049 ₴

Замовлення 18

19 травня 2025 р. о 22:09

Статус оплати: Cancelled

Товари:

Джинси «PROFOUND»
Колір: BLACK, Розмір: S

1 × 1 599 ₴
1 599 ₴

Загальна сума: 1 599 ₴

Рисунок 3.11 – Профіль користувача

Сторінка особистого кабінету (див. рис. 3.11.) користувача виконує подвійну роль: вона не лише дозволяє оновити власні дані, а й надає повний огляд історії покупок. У верхній частині відображається назва розділу з іменем користувача. Одразу під заголовком знаходиться форма, в якій автоматично підставлені email та ПІБ, отримані з бази даних. Поле email видно в активному стані з можливістю швидкого очищення, поруч із ним розташовано поле «Повне ім'я». Користувач може змінити ці дані просто увівши нові значення й натиснувши кнопку «Зберегти», яка, у свою чергу, проводить валідацію через налаштовані правила Zod і надсилає оновлені дані на сервер. Нижче цієї пари полів розміщені два поля для введення нового пароля та його підтвердження, що

					ІПЗ.КР.Б – 121 – 25 – ПЗ	63
--	--	--	--	--	--------------------------	----

дозволяють водночас оновити облікові дані й гарантовано уникнути помилки при наборі. Після успішного збереження змін у формі з'являється невелике повідомлення про успіх, а у разі помилки — коротка підказка, що вказує, яке поле вимагає виправлення. Поруч із кнопкою «Зберегти» стоїть сірий елемент «Вихід», що дозволяє завершити сесію й повернутися на головну сторінку магазину.

Праворуч від форми розташований хронологічний перелік минулих замовлень. Кожен запис починається з номера та дати замовлення, після чого вказано поточний статус оплати («Succeeded», «Pending» або «Cancelled»), що дозволяє розібратися замовлення успішно оплачено, чекає на оплату або скасовано. Далі слідує детальний опис товарів: назва продукту, обрана колірна та розмірна конфігурація, кількість одиниць та розрахункова сума за позицію, яка формується множенням ціни на кількість. Наприкінці кожної картки виноситься загальний підсумок замовлення великим жирним шрифтом. У випадку, якщо користувач ще не здійснював жодної покупки, цей блок просто залишається пустим, і сторінка фокусується лише на формі особистих даних. Завдяки тому, що весь інтерфейс працює в рамках одного React-сервісу, оновлення і збереження інформації відбуваються миттєво без перезавантаження, а взаємодія з базою даних через Prisma гарантує цілісність і своєчасність змін.

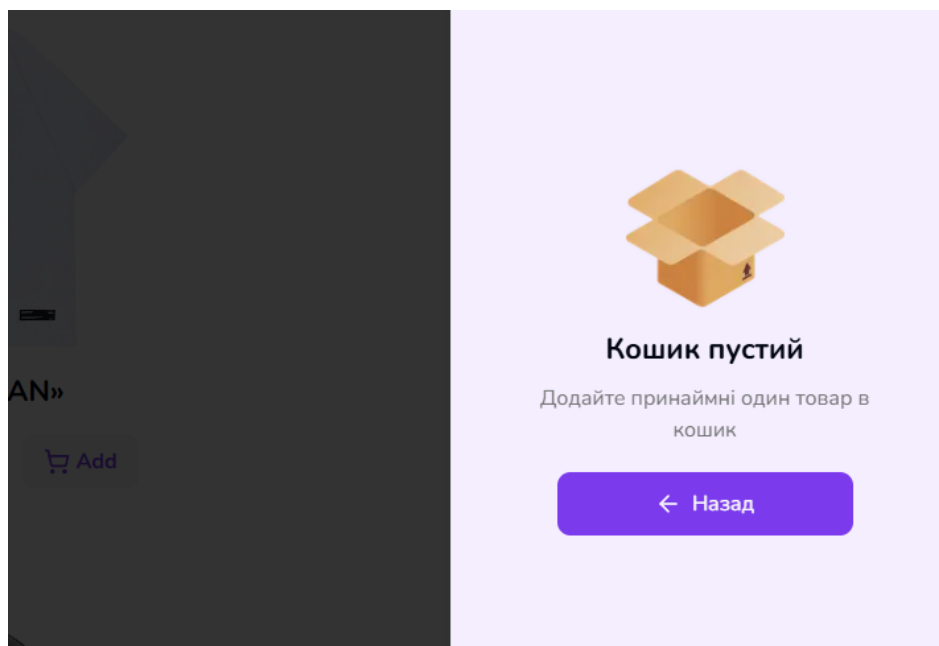


Рисунок 3.12 – Drawer-меню кошика

У верхній частині сайту є кнопки профілю та кошика (див. рис. 3.13.). При натисканні на кошик в правій частині сторінки розкривається меню кошика (див. рис. 3.12.).



Рисунок 3.13 – Кнопки профілю та кошика

Сторінка товару (див. рис. 3.14.) виконує роль інтерфейсу для ознайомлення з конкретною позицією одягу та її додавання до кошика. Вона надає користувачу основні функціональні можливості для персоналізації покупки: вибір варіанта кольору і розміру. Поряд виводиться актуальна кількість одиниць товару на складі для вибраної комбінації кольору та розміру, що допомагає зорієнтуватися, наскільки терміновим є рішення про покупку.

Клік по кнопці «Додати в корзину» додає товар до кошика, враховуючи обрані параметри (колір, розмір), після чого користувач може продовжити перегляд або перейти до оформлення замовлення. Вся логіка побудована так, щоб мінімізувати кількість дій і зробити шлях до покупки максимально швидким та інтуїтивним.

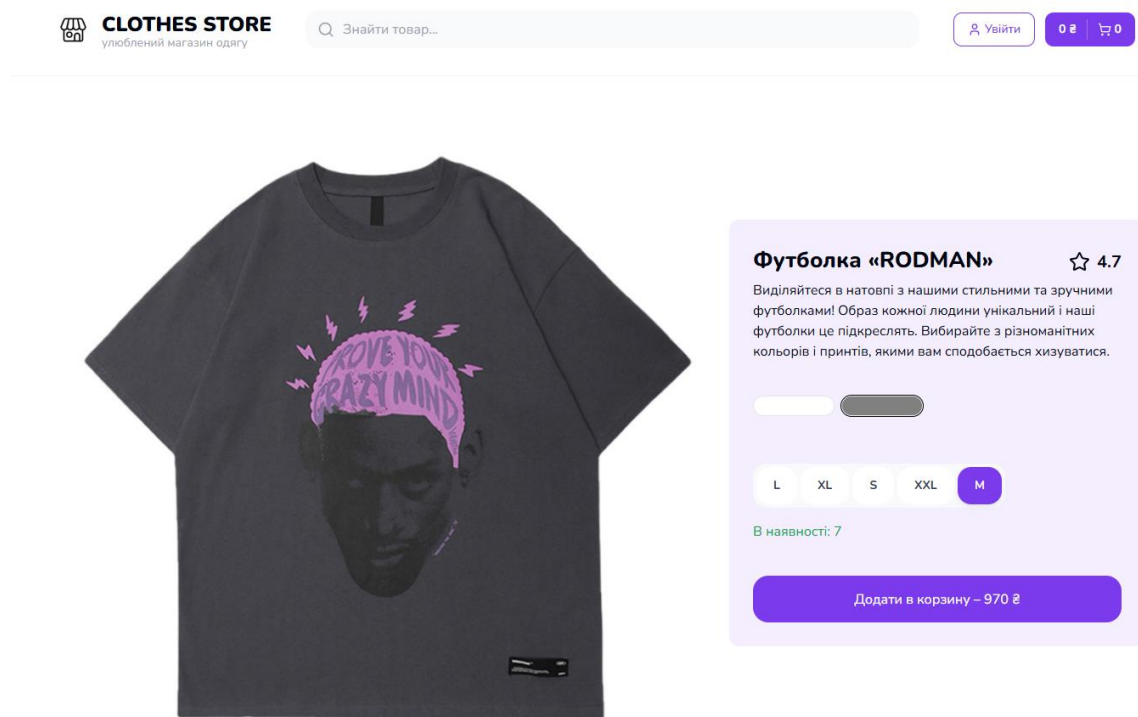


Рисунок 3.14 – Сторінка товару

Після додавання товарів у козину користувач може перейти на сторінку оформлення замовлення натиснувши відповідну кнопку в корзині (див. рис. 3.15.).

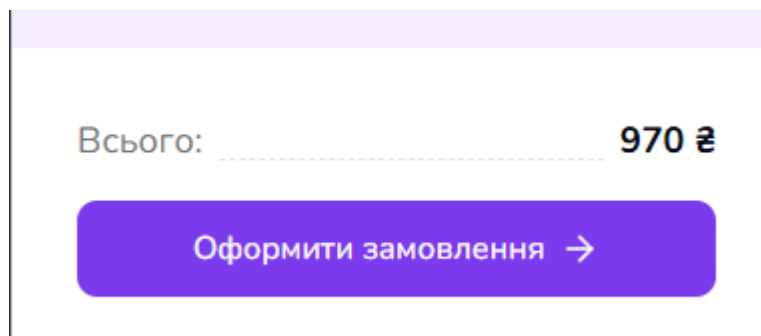


Рисунок 3.15 – Кнопка оформлення замовлення

На сторінці оформлення замовлення користувачу відображаються елементи обраної ним корзини, дві форми для особистих даних отримувача замовлення і блок з усіма сумами та кнопкою переходу на оплату замовлення (див. рис. 3.16.).

Рисунок 3.16 – Сторінка оформлення замовлення

3.3. Тестування роботи програмного комплексу

Проведемо функціональне тестування розробленого інтернет-магазину. Функціональне тестування спрямоване на те, щоб кожна функція веб-сайту працювала відповідно до специфікації вимог. Тестування функціоналу веб-сайту показує "Що робить система".

Для прикладу застосуємо фільтр за підкатегорією «Сумки» і сортуванням за найвищою ціною (див. рис. 3.17.).

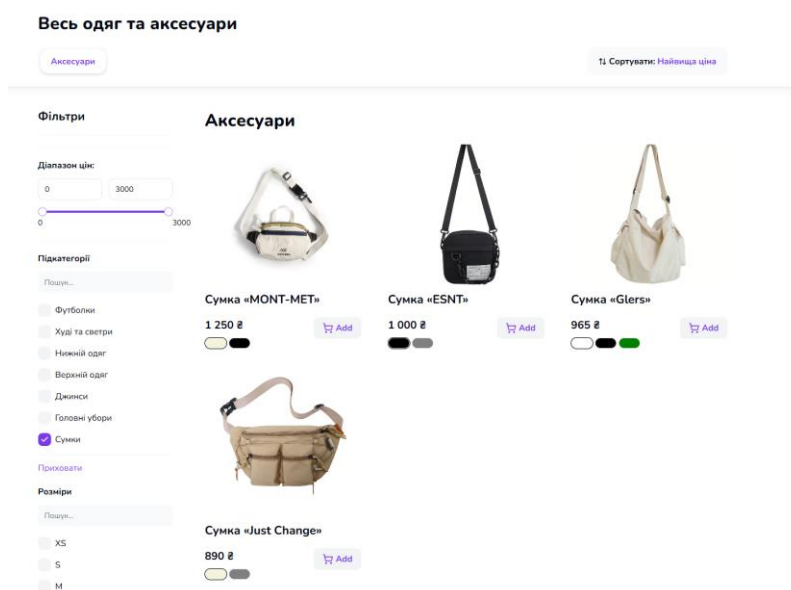


Рисунок 3.17 – Застосування фільтрації і сортування одночасно

Додаємо фільтри за кольором і змінюємо сортування, у результаті отримуємо коректний результат (див. рис. 3.18.)

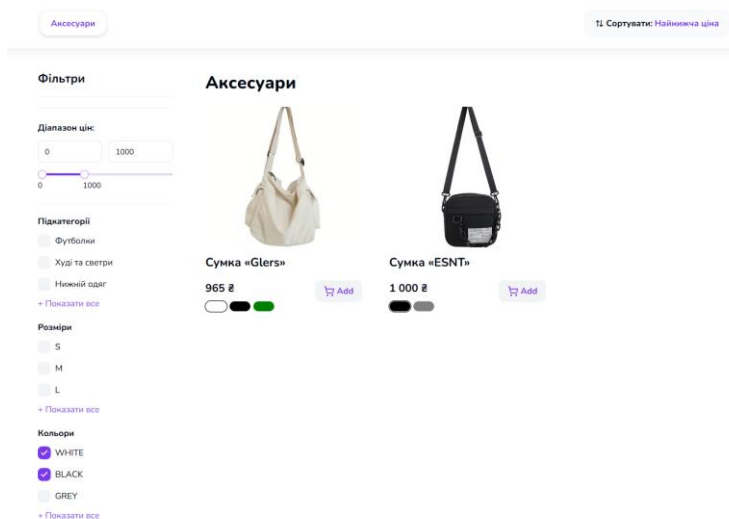
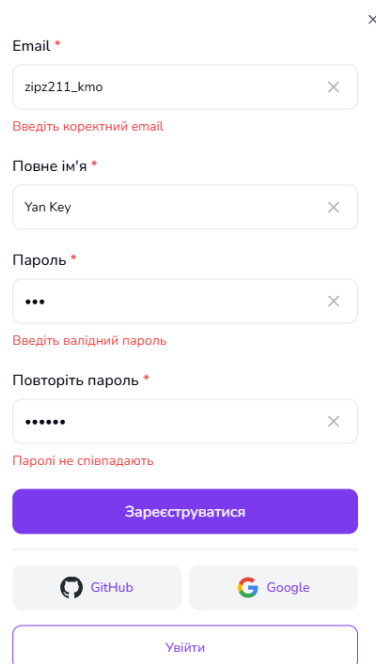


Рисунок 3.18 – Застосування фільтрації і сортування одночасно

При введенні у форму реєстрації (див. рис. 3.19.) некоректних даних стикаємось з наявністю валідації полів. Є перевірка вводу коректного формату пошти, для паролів передбачена перевірка на довжину паролю та на відповідність двох введених паролів.



Registration form with the following fields and errors:

- Email ***: Input field contains "zipz211_kmo". Error message: "Введіть коректний email".
- Повне ім'я ***: Input field contains "Yan Key".
- Пароль ***: Input field contains "...". Error message: "Введіть валідний пароль".
- Повторіть пароль ***: Input field contains ".....". Error message: "Паролі не співпадають".

Buttons: "Зареєструватися" (purple), "GitHub", "Google", "Увійти" (outlined).

Рисунок 3.19 – Валідація форми реєстрації

Розглянемо два стани (див. рис. 3.20., 3.21.): до додавання товару в кошик та після додавання товару в кошик.

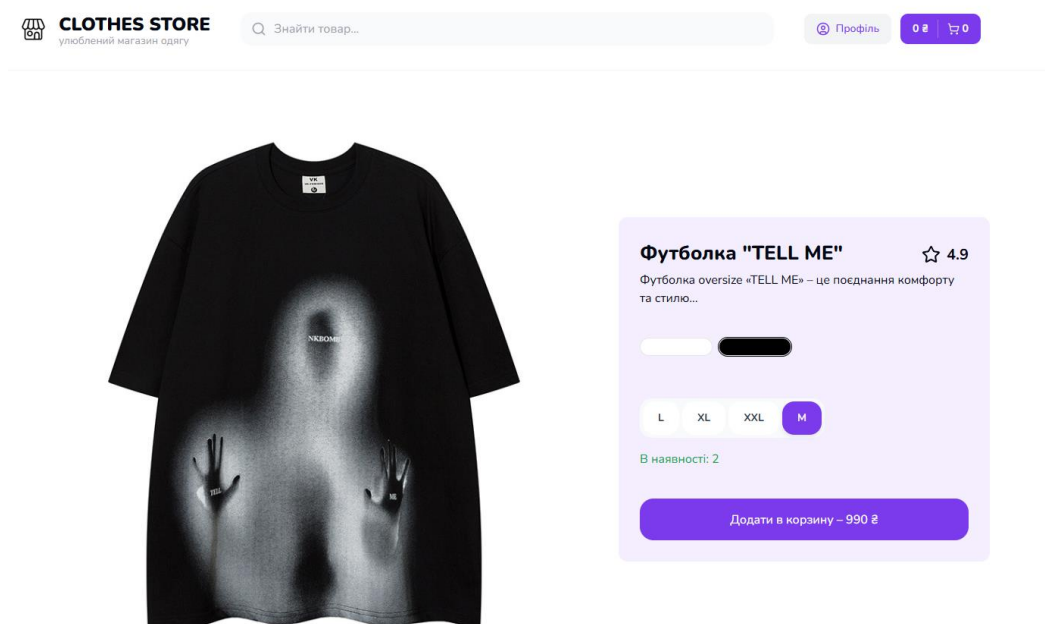


Рисунок 3.20 – Сторінка товару до його додавання у кошик

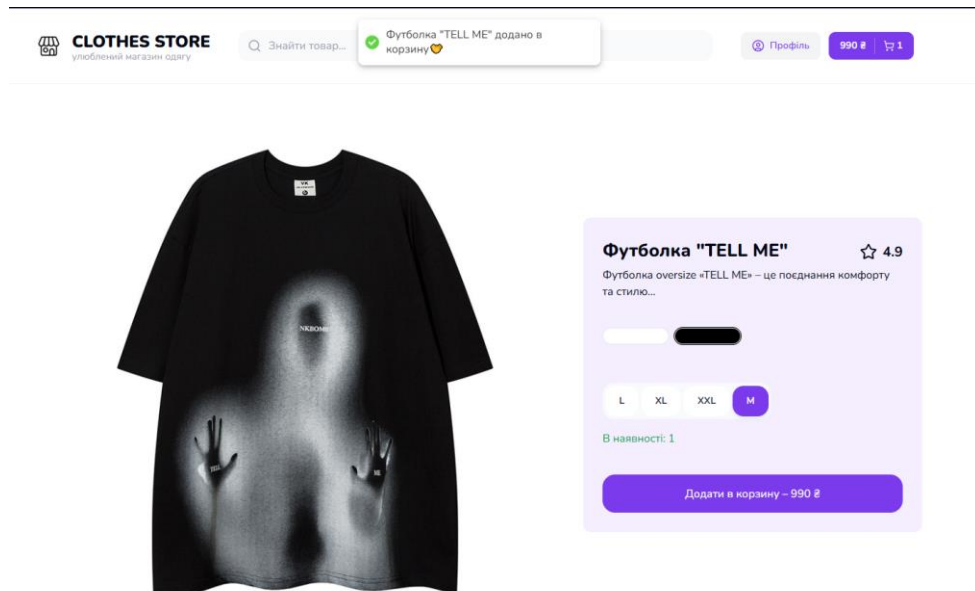


Рисунок 3.21 – Сторінка товару після його додавання у кошик

Після успішного додавання товару у кошик з'являється повідомлення для користувача, що повідомляє про успішне завершення дії (див. рис. 3.21.). На кнопці кошику з'явилися числові значення, що означають суму товарів і кількість товарів в кошику. Також зменшився лічильник «В наявності» – при додаванні товару в кошику в резервується для користувача, щоб інші користувачі не змогли цей же момент викупити останній товар.

Якщо додати останній товар в кошик – кнопка додавання в кошик стає неактивною, кнопка збільшення кількості одиниць конкретної конфігурації товару в кошику також стає неактивною (див. рис. 3.22.).

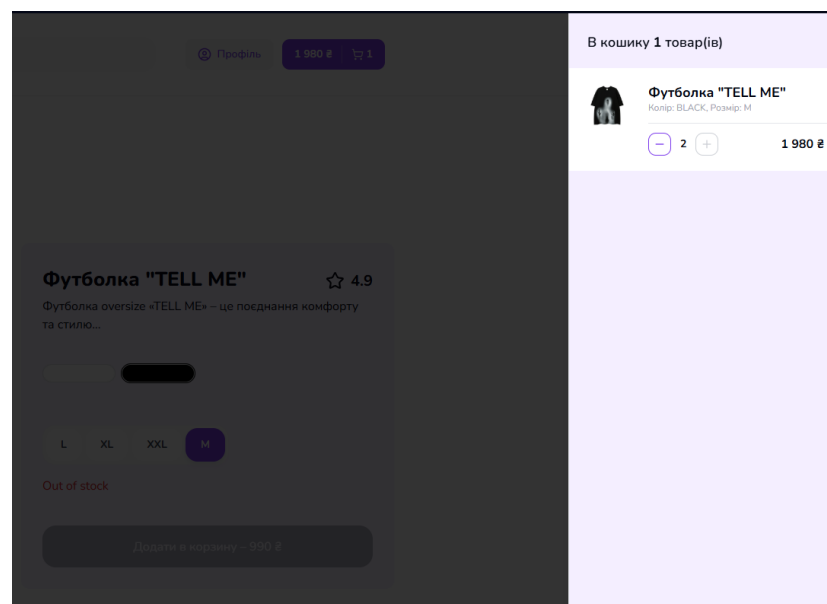


Рисунок 3.22 – Сторінка товару після його додавання у кошик

При заповненні кошику товарами правильно підраховуються суми на всіх блоках. Правильно працює розділення товарів: при додаванні одного товару з одним кольором, але іншим розміром товар додається в корзину як новий елемент (див. рис. 3.23.).

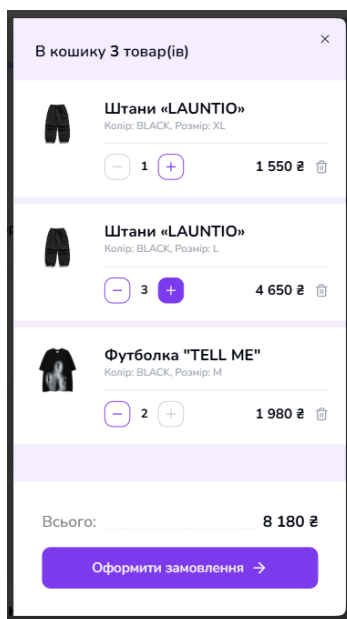


Рисунок 3.23 – Меню кошика з різними товарами

На сторінці оформлення замовлення відображаються всі елементи корзини, що були обрані в момент оформлення замовлення, є можливість змінити кількість елементів. Також працює валідація всіх полів, дані з яких необхідні для оформлення замовлення (див. рис. 3.24.).

Оформлення замовлення

1. Корзина

	Штани «LAUNTIO» Колір: BLACK, Розмір: XL	1 550 ₴	- 1 +	×
	Штани «LAUNTIO» Колір: BLACK, Розмір: L	4 650 ₴	- 3 +	×
	Футболка «TELL ME» Колір: BLACK, Розмір: M	1 980 ₴	- 2 +	×

До сплати:
8330 ₴

Сума замовлення: 8180 ₴

Доставка: 150 ₴

Перейти до оплати →

2. Дані отримувача

Ім'я
 Прізвище

Ім'я повинно містити щонайменше 2 букини
Прізвище повинно містити щонайменше 2 букини

Email
 Номер телефону

Введіть правильний email
Введіть дійсний номер телефону

3. Адреса отримувача

 Оберіть адресу доставки

Вкажіть правильну адресу

 Коментар до замовлення

Рисунок 3.24 – Валідація форм на сторінці оформлення

Після оформлення замовлення відбувається редірект на сторінку оплати Лідрау, перед цим для юзера відображається повідомлення про успішне оформлення замовлення (див. рис. 3.25.).

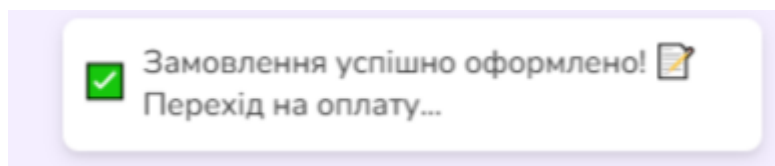


Рисунок 3.25 – Повідомлення про успішне оформлення замовлення

QR-код для оплати

Використовуйте Privat24

TESTOVYI ROK

LIQPAY

Дані про оплату

Оплата замовлення #22

До оплати: 8180.00 UAH

24 Pay

G Pay

4987

або

Регістр

Приват24

Розрахунок

Номер картки

4000 0000 0000 3055

VISA

Термін дії

03/43

CV2

...

Ім'я власника картки*

Nikita

Прізвище власника картки*

Korotkyi

☐ Відправити квитанцію на e-mail

Натискаючи на кнопку «Оплатити», ви підтверджуєте що ознайомлені з переліком інформації про послугу та приймаєте умови публічного договору

Оплатити 8180.00 UAH

Скесувати оплату

Рисунок 3.26 – Сторінка оплати замовлення

Незалежно від результату оплати замовлення, воно з'явиться в профілі користувача, якщо він авторизований (див. рис. 3.27.).

CLOTHES STORE

улюблений магазин одягу

🔍 Знайти товар...

Профіль

08 🛒

Особисті дані | Korotkyi Nikita

Email *

zip211_kmo@student.ztu.edu.ua ✕

Повне ім'я *

Korotkyi Nikita ✕

Новий пароль *

Повторіть пароль *

Зберегти

Вихід

Замовлення 22

12 червня 2025 р. о 20:30

Статус оплати: Cancelled

Товари:

Штани «LAUNTIO»
Колір: BLACK, Розмір: XL
1 × 1 550 ₴
1 550 ₴

Штани «LAUNTIO»
Колір: BLACK, Розмір: L
3 × 1 550 ₴
4 650 ₴

Футболка "TELL ME"
Колір: BLACK, Розмір: M
2 × 990 ₴
1 980 ₴

Загальна сума: 8 180 ₴

Замовлення 21

1 червня 2025 р. о 20:20

Статус оплати: Succeded

Товари:

Зір-худи <SOUL QUALITY>
Колір: WHITE, Розмір: XS
2 × 1 750 ₴
3 500 ₴

Рисунок 3.27 – Профіль користувача

Висновки до третього розділу

У третьому розділі кваліфікаційної роботи бакалавра детально розглянуто функціональні можливості та логіку роботи інтерфейсу розробленого вебзастосунку — інтернет-магазину одягу. Представлено опис основних екранів системи, таких як сторінка профілю користувача, інтерфейс перегляду замовлень та сторінка окремого товару, а також проаналізовано їхню поведінку з точки зору зручності користування.

Особливу увагу приділено інтерактивним елементам, які підвищують юзабіліті системи. Зокрема, на сторінці товару реалізовано зміну варіанта кольору, динамічний вибір розміру зі зворотним зв'язком про наявність обраної комбінації на складі, а також функцію додавання товару до кошика з урахуванням усіх заданих параметрів. Такий підхід дозволяє користувачеві швидко і без зайвих кроків сформулювати своє замовлення.

У профілі авторизованого користувача передбачено можливість редагування особистих даних (email, ім'я, пароль) та перегляду історії замовлень із відповідними статусами (Succeeded, Pending, Cancelled), детальним списком товарів у кожному замовленні та розрахунком загальної суми. Це забезпечує прозору взаємодію з клієнтом і дозволяє відслідковувати виконання замовлень.

Також у розділі наведено приклад роботи системи з демонстрацією ключових дій користувача та описано послідовність налаштування і запуску застосунку. Проведено базове тестування для перевірки коректності виконання основних сценаріїв. Усі ці аспекти підтверджують стабільність, функціональність та готовність системи до практичного використання.

ВИСНОВКИ

У процесі виконання кваліфікаційної роботи було повністю реалізовано інтернет-магазин одягу, який дозволяє користувачам зручно переглядати товари, обирати розміри та кольори, додавати їх до кошика, оформлювати замовлення та відслідковувати історію своїх покупок. Основною метою роботи було створення сучасного, функціонального та зручного вебзастосунку, і ця мета була досягнута. Система розроблена з урахуванням реальних потреб користувачів і включає всі базові функції, необхідні для повноцінного онлайн-шопінгу. Користувач може пройти реєстрацію або авторизуватися, змінити свої персональні дані, при потребі — змінити пароль. У профілі також реалізовано розділ з історією замовлень, де користувач бачить статус оплати (успішна, очікує, скасована), дату, склад кожного замовлення та загальну суму до сплати. Каталог товарів реалізовано з акцентом на зручність вибору. Користувач може переглядати товари за зображенням, обирати доступні кольори та розміри, бачити актуальну кількість одиниць товару на складі та одразу додавати обрані позиції в кошик. Кожен товар має короткий опис, рейтинг, що дозволяє сформуванню уявлення про нього ще до покупки. Також система забезпечує збереження та відображення замовлень з детальною інформацією по кожній позиції. Це дає можливість користувачам у будь-який момент повернутись до попередніх замовлень, переглянути їхній вміст і статус. Навіть при скасованих або незавершених покупках інформація не втрачається — усе зберігається в особистому кабінеті.

Тестування системи показало, що вона стабільно працює, коректно обробляє введені дані та запити користувачів. Було перевірено функціональність форм, коректність переходів між сторінками, точність обчислення сум замовлення та відповідність відображуваних даних фактичному стану системи. Жодних критичних помилок не виявлено.

Серед реалізованих функцій можна відзначити:

- створення особистого кабінету користувача з можливістю редагування особистих даних;

- додавання товарів до кошика з урахуванням обраного розміру та кольору;
- перегляд залишку товару на складі;
- оформлення та збереження замовлень із фіксацією часу, статусу та вартості;
- перегляд історії покупок.

Всі ці функції відповідають початковим вимогам, поставленим перед початком розробки, і підтверджують, що система повністю реалізована згідно з технічним завданням. Її можна вважати завершеною з точки зору базової функціональності.

Практична цінність отриманого результату полягає в тому, що система може використовуватися як основа для комерційного онлайн-магазину. Вона вже має набір основних функцій, і за потреби може бути масштабована — додані нові категорії товарів, розділи для акцій, система бонусів, інтеграція платіжних сервісів тощо.

У майбутньому систему доцільно розширити адміністративною панеллю, яка дозволить додавати нові товари, редагувати їх, керувати замовленнями, відслідковувати статистику покупок. Також варто розглянути можливість створення мобільної версії або адаптивного інтерфейсу для зручного користування на телефонах і планшетах.

Таким чином, виконана робота підтвердила можливість реалізації повноцінного інтернет-магазину з використанням сучасних технологій, який має практичне значення та потенціал для подальшого розвитку.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Next.js Documentation [Електронний ресурс] – Режим доступу: <https://nextjs.org/docs>
2. REST API Documentation [Електронний ресурс] – Режим доступу: <https://docs.github.com/en/rest/about-the-rest-api>
3. React – A JavaScript library for building user interfaces [Електронний ресурс] – Режим доступу: <https://react.dev>
4. Tailwind CSS – Documentation [Електронний ресурс] – Режим доступу: <https://tailwindcss.com/docs>
5. TypeScript Documentation [Електронний ресурс] – Режим доступу: <https://www.typescriptlang.org/docs>
6. Prisma ORM – Documentation [Електронний ресурс] – Режим доступу: <https://www.prisma.io/docs>
7. PostgreSQL – The World's Most Advanced Open Source Relational Database [Електронний ресурс] – Режим доступу: <https://www.postgresql.org>
8. LiqPay API Documentation [Електронний ресурс] – Режим доступу: <https://www.liqpay.ua/documentation/en>
9. Zod – TypeScript-first schema validation with static type inference [Електронний ресурс] – Режим доступу: <https://zod.dev>
10. Zustand – A small, fast and scalable bearbones state-management solution [Електронний ресурс] – Режим доступу: <https://github.com/pmndrs/zustand>
11. react-hook-form – Performant, flexible and extensible forms [Електронний ресурс] – Режим доступу: <https://react-hook-form.com>
12. NextAuth.js – Authentication for Next.js [Електронний ресурс] – Режим доступу: <https://next-auth.js.org>
13. Axios – Promise based HTTP client for the browser and node.js [Електронний ресурс] – Режим доступу: <https://axios-http.com>
14. CI/CD documentation [Електронний ресурс] – Режим доступу: <https://docs.gitlab.com/ci/>

15. Хук useSession [Електронний ресурс] – Режим доступу: <https://next-auth.js.org/getting-started/client>
16. shadcn/ui – Customizable components built on Radix UI and Tailwind CSS [Електронний ресурс] – Режим доступу: <https://ui.shadcn.com>
17. Lucide Icons – Beautiful & consistent icon toolkit [Електронний ресурс] – Режим доступу: <https://lucide.dev>
18. PostgreSQL with Prisma [Електронний ресурс] – Режим доступу: <https://www.prisma.io/docs/concepts/database-connectors/postgresql>
19. REST API Design – Best Practices [Електронний ресурс] – Режим доступу: <https://restfulapi.net>
20. GitHub – Code hosting platform for version control and collaboration [Електронний ресурс] – Режим доступу: <https://github.com>
21. Vercel – Deployment and Hosting for Next.js apps [Електронний ресурс] – Режим доступу: <https://vercel.com>
22. Zalando – Інтернет-магазин одягу та взуття [Електронний ресурс] – Режим доступу: <https://www.zalando.co.uk>
23. Kasta – Інтернет-магазин модного одягу [Електронний ресурс] – Режим доступу: <https://kasta.ua>
24. ASOS – Інтернет-магазин модного одягу [Електронний ресурс] – Режим доступу: <https://www.asos.com/>
25. H&M – Інтернет-магазин одягу [Електронний ресурс] – Режим доступу: <https://www.hm.com>
26. How to choose the best architecture for your Web application [Електронний ресурс] – Режим доступу: <https://medium.com/crowdbotics/how-to-choose-the-best-architecture-for-your-web-application-67eb47886021>
27. Radix UI – Primitives for building accessible UI components [Електронний ресурс] – Режим доступу: <https://www.radix-ui.com>
28. UX Planet – Інтерфейсні практики та досвід користувача [Електронний ресурс] – Режим доступу: <https://uxplanet.org>

29. Web.dev – Керівництво з продуктивності, доступності та SEO вебсайтів [Електронний ресурс] – Режим доступу: <https://web.dev>
30. HTTP Cookies – MDN Web Docs [Електронний ресурс] – Режим доступу: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>
31. Server Actions – Next.js Documentation [Електронний ресурс] – Режим доступу: <https://nextjs.org/docs/app/building-your-application/data-fetching/server-actions>

ДОДАТКИ

					ПЗ.КР.Б – 121 – 25 – ПЗ	
						78

Лістинг коду файлу `schema.prisma`, що відповідає за схему БД у вигляді коду, яку Prisma використовує для генерації запитів і взаємодії з базою

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

datasource db {
  provider = "postgresql"
  url      = env("POSTGRES_PRISMA_URL")
  directUrl = env("POSTGRES_URL_NON_POOLING")
}

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

model Category {
  id      Int      @id @default(autoincrement())
  name    String
  subcategories Subcategory[]
}

model Subcategory {
  id      Int      @id @default(autoincrement())
  name    String
  categoryId Int
  items    Item[]
  category Category @relation(fields: [categoryId], references: [id])
}

model Item {
  id      Int      @id @default(autoincrement())
  title    String
  price    Int
  description String
  image    String
  rating   Float
  numReviews Int
  subcategoryId Int
  subcategory Subcategory @relation(fields: [subcategoryId], references: [id])
  variants  Variant[]
}

model Variant {
```

```

id      Int      @id @default(autoincrement())
itemId  Int
color   Color
size    Size?
stock   Int
cartItems CartItem[]
item     Item     @relation(fields: [itemId], references: [id])
}

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

model CartItem {
  id      Int      @id @default(autoincrement())
  cartId   Int
  variantId Int
  quantity Int
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  cart     Cart     @relation(fields: [cartId], references: [id])
  variant  Variant  @relation(fields: [variantId], references: [id])

  @@unique([cartId, variantId])
}

model Order {
  id      Int      @id @default(autoincrement())
  userId  Int?
  items    Json?
  status   OrderStatus @default(PENDING)
  totalAmount Int
  paymentId String?
  fullName String
  address   String
  email     String
  phone     String
  comment   String?
  createdAt DateTime @default(now())
  updatedAt DateTime @updatedAt
  token     String
  user      User?    @relation(fields: [userId], references: [id])
}

model VerificationCode {
  id      Int      @id @default(autoincrement())
  userId  Int      @unique
  code    String
  expiresAt DateTime @default(now())
  createdAt DateTime @default(now())

```

```

    updatedAt DateTime @updatedAt
    user User @relation(fields: [userId], references: [id])

    @@unique([userId, code])
  }
  enum OrderStatus {
    PENDING
    SUCCEEDED
    CANCELLED
  }
  enum UserRole {
    USER
    ADMIN
  }
  enum Color {
    WHITE
    BLACK
    GREY
    BEIGE
    RED
    BLUE
    GREEN
  }
  enum Size {
    XS
    S
    M
    L
    XL
    XXL
  }

```

Лістинг коду файлу find-items.ts, логіка в якому повертає список товарів і категорій згідно з фільтрами

```

export interface GetSearchParams {
  query?: string
  sortBy?: SortKey
  sizes?: string
  colors?: string
  subcategories?: string
  priceFrom?: string
  priceTo?: string
}

const DEFAULT_MIN_PRICE = 0
const DEFAULT_MAX_PRICE = 3000

export async function findItemsFlat(params: GetSearchParams): Promise<CategoryWithItems[]> {
  const subcategoryIds: number[] = params.subcategories
    ? params.subcategories.split(",").map((v) => Number(v))
    : []

```

```
const colorFilters: Color[] = params.colors ? params.colors.split(",").map((c) => c as Color) : []
```

```
const sizeFilters: Size[] = params.sizes ? params.sizes.split(",").map((s) => s as Size) : []
```

```
const minPrice = Number(params.priceFrom) || DEFAULT_MIN_PRICE
```

```
const maxPrice = Number(params.priceTo) || DEFAULT_MAX_PRICE
```

```
const categories = await prisma.category.findMany({
  ...(subcategoryIds.length > 0 && {
    where: {
      subcategories: {
        some: { id: { in: subcategoryIds } },
      },
    },
  }),
  include: {
    subcategories: {
      ...(subcategoryIds.length > 0 && {
        where: { id: { in: subcategoryIds } },
      }),
    },
    include: {
      items: {
        where: {
          price: { gte: minPrice, lte: maxPrice },
          ...(colorFilters.length > 0 || sizeFilters.length > 0
            ? {
                variants: {
                  some: {
                    ...(colorFilters.length > 0 && { color: { in: colorFilters } }),
                    ...(sizeFilters.length > 0 && { size: { in: sizeFilters } }),
                  },
                },
              }
            : {}),
        },
        include: { variants: true },
      },
    },
  },
})
```

```
function sortItems(items: GroupedItem[], sortBy?: SortKey): GroupedItem[] {
  if (!sortBy) return items
```

```
  const sorted = [...items]
  if (sortBy === "priceAsc") {
    sorted.sort((a, b) => a.price - b.price)
  } else if (sortBy === "priceDesc") {
    sorted.sort((a, b) => b.price - a.price)
  } else if (sortBy === "ratingDesc") {
    sorted.sort((a, b) => b.rating - a.rating)
  }
}
```

```

    return sorted
  }

  return categories.map((cat) => {
    const allItems: GroupedItem[] = cat.subcategories.flatMap((sub) =>
      sub.items.map((item) => {
        const colors = Array.from(new Set(item.variants.map((v) => v.color)))
        const sizes = Array.from(
          new Set(item.variants.map((v) => v.size).filter((s): s is Size => s != null))
        )
        const stockValue: number = (item as any).stock ?? 0

        const gi: GroupedItem = {
          id: item.id,
          title: item.title,
          price: item.price,
          description: item.description,
          image: item.image,
          rating: item.rating,
          numReviews: item.numReviews,
          subcategoryId: item.subcategoryId,
          colors,
          sizes,
          stock: stockValue,
        }
        return gi
      })
  })
)

const sortedItems = sortItems(allItems, params.sortBy)

return {
  id: cat.id,
  name: cat.name,
  items: sortedItems,
}
})
}

```

Лістинг коду файлу create-payment.ts логіка якого відповідає за створення посилання для оплати замовлення

```

import crypto from "crypto"

export type LiqpayItem = {
  id: number
  title: string
  color: string
  price: number
  size: string | null
  quantity: number
}

```

```

interface LiqpayCheckoutOptions {
  items?: LiqpayItem[]
  amount: number | string
  order_id: number
  description: string
  result_url?: string
  server_url?: string
}

export function generateLiqpayCheckoutUrl({
  items,

  amount,
  order_id,
  description,
  result_url = `${process.env.NEXT_APP_URL}`,
  server_url = `${process.env.NEXT_APP_URL}/api/checkout/callback`,
}: LiqpayCheckoutOptions): string {
  const publicKey = process.env.LIQPAY_PUBLIC_KEY!
  const privateKey = process.env.LIQPAY_PRIVATE_KEY!

  const expiresAt = new Date()
  expiresAt.setMinutes(expiresAt.getMinutes() + 1)
  const expired_date = formatUTC(expiresAt)

  const params = {
    public_key: publicKey,
    version: "3",
    action: "pay",
    amount: String(amount),
    currency: "UAH",
    description,
    order_id,
    result_url,
    server_url,
    expired_date,
    items,
  }

  const data = Buffer.from(JSON.stringify(params)).toString("base64")

  const signature = crypto
    .createHash("sha1")
    .update(privateKey + data + privateKey)
    .digest("base64")

  return
  `https://www.liqpay.ua/api/3/checkout?data=${encodeURIComponent(data)}&signature=${encodeURIComponent(signature)}`
}

function formatUTC(date: Date): string {

```

```

const YYYY = date.getUTCFullYear()
const MM = String(date.getUTCMonth() + 1).padStart(2, "0")
const DD = String(date.getUTCDate()).padStart(2, "0")
const hh = String(date.getUTCHours()).padStart(2, "0")
const mm = String(date.getUTCMinutes()).padStart(2, "0")
const ss = String(date.getUTCSeconds()).padStart(2, "0")
return `${YYYY}-${MM}-${DD} ${hh}:${mm}:${ss}`
}

```

Лістинг коду файлу url-filter.ts, логика в якому відповідає за сортування

```

import { prisma } from "@prisma/prisma-client"

export interface GetSearchParams {
  query?: string
  sizes?: string
  colors?: string
  subcategories?: string
  priceFrom?: string
  priceTo?: string
  sortBy?: "priceAsc" | "priceDesc" | "titleAsc" | "titleDesc"
}

export interface FilteredItem {
  id: number
  title: string
  description: string
  price: number
  image: string
  rating: number
  numReviews: number
  subcategoryId: number
  sizes: Array<string | null>
  colors: Array<string>
}

export interface CategoryGroup {
  id: number
  name: string
  items: FilteredItem[]
}

export async function filterItems({
  query: searchQuery,
  sizes,
  colors,
  subcategories,
  priceFrom,
  priceTo,
  sortBy,
}: GetSearchParams): Promise<CategoryGroup[]> {

```

```

const sizeIds = sizes
  ?.split(",")
  .map(Number)
  .filter((n) => !isNaN(n))
const colorIds = colors
  ?.split(",")
  .map(Number)
  .filter((n) => !isNaN(n))
const subcatIds = subcategories
  ?.split(",")
  .map(Number)
  .filter((n) => !isNaN(n))

const minPrice = priceFrom ? parseInt(priceFrom, 10) : 0
const maxPrice = priceTo ? parseInt(priceTo, 10) : Number.MAX_SAFE_INTEGER

const itemWhere: any = {
  price: { gte: minPrice, lte: maxPrice },
}
if (searchQuery) {
  itemWhere.OR = [
    { title: { contains: searchQuery, mode: "insensitive" } },
    { description: { contains: searchQuery, mode: "insensitive" } },
  ]
}
if (subcatIds?.length) {
  itemWhere.subcategoryId = { in: subcatIds }
}

const variantWhere: any = {}
if (sizeIds?.length) variantWhere.size = { in: sizeIds }
if (colorIds?.length) variantWhere.color = { in: colorIds }

const orderBy: any = {}
switch (sortBy) {
  case "priceAsc":
    orderBy.price = "asc"
    break
  case "priceDesc":
    orderBy.price = "desc"
    break
  case "titleDesc":
    orderBy.title = "desc"
    break
  default:
    orderBy.title = "asc"
    break
}

const categories = await prisma.category.findMany({
  where: subcatIds?.length ? { subcategories: { some: { id: { in: subcatIds } } } } : {},
  include: {
    subcategories: {

```

```

include: {
  items: {
    where: {
      ...itemWhere,
      ...(variantWhere.size || variantWhere.color ? { variants: { some: variantWhere } } : {}),
    },
    include: {
      variants: true,
    },
    orderBy: [orderBy],
  },
},
},
},
},
})

const result: CategoryGroup[] = categories.map((cat) => ({
  id: cat.id,
  name: cat.name,
  items: cat.subcategories.flatMap((subcat) =>
    subcat.items.map((item) => ({
      id: item.id,
      title: item.title,
      description: item.description,
      price: item.price,
      image: item.image,
      rating: item.rating,
      numReviews: item.numReviews,
      subcategoryId: item.subcategoryId,

      sizes: Array.from(new Set(item.variants.map((v) => v.size))),
      colors: Array.from(new Set(item.variants.map((v) => v.color))),
    })))
  ),
})))

return result
}

```

Лістинг коду файлу useCart.ts, що містить в собі логіку роботи з корзиною за допомогою Zustand

```

import { create } from "zustand"
import { Api } from "../services/api-client"
import { getCartDetails, CartStateItem } from "../lib/get-cart-details"

import { CreateCartItemValues } from "../services/dto/cart.dto"

export interface CartState {
  loading: boolean
  error: boolean
  totalAmount: number
  items: CartStateItem[]

```

```

fetchCartItems: () => Promise<void>

updateItemQuantity: (id: number, quantity: number) => Promise<void>

addCartItem: (values: CreateCartItemValues) => Promise<void>

removeCartItem: (id: number) => Promise<void>
}

export const useCartStore = create<CartState>((set, get) => ({
  items: [],
  error: false,
  loading: true,
  totalAmount: 0,

  fetchCartItems: async () => {
    try {
      set({ loading: true, error: false })
      const data = await Api.cart.getCart()
      set(getCartDetails(data))
    } catch (error) {
      console.error(error)
      set({ error: true })
    } finally {
      set({ loading: false })
    }
  },

  updateItemQuantity: async (id: number, quantity: number) => {
    try {
      set((state) => ({
        loading: true,
        error: false,
        items: state.items.map((item) => (item.id === id ? { ...item, disabled: true } : item)),
      }))
      const data = await Api.cart.updateItemQuantity(id, quantity)
      set(getCartDetails(data))
    } catch (error) {
      console.error(error)
      set({ error: true })
    } finally {
      set((state) => ({
        loading: false,
        items: state.items.map((item) => ({ ...item, disabled: false })),
      }))
    }
  },

  removeCartItem: async (id: number) => {
    try {
      set((state) => ({
        loading: true,

```

```

        error: false,
        items: state.items.map((item) => (item.id === id ? { ...item, disabled: true } : item)),
    )))

    const data = await Api.cart.updateItemQuantity(id, 0)

    set(getCartDetails(data))
  } catch (error) {
    console.error(error)
    set({ error: true })
  } finally {
    set((state) => ({
      loading: false,
      items: state.items.map((item) => ({ ...item, disabled: false })),
    })))
  }
},
addCartItem: async (values: CreateCartItemValues) => {
  try {
    set({ loading: true, error: false })
    const data = await Api.cart.addCartItem(values)
    set(getCartDetails(data))
  } catch (error) {
    console.error(error)
    set({ error: true })
  } finally {
    set({ loading: false })
  }
},
)))

```