**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ВОДНОГО ГОСПОДАРСТВА ТА ПРИРОДОКОРИСТУВАННЯ**

«До захисту допущена»
Зав. кафедри комп'ютерних наук
та прикладної математики
д.т.н., професор Турбал Ю.В.
«____» _____ 20_ р.

# КВАЛІФІКАЦІЙНА РОБОТА

«ПРОЄКТУВАННЯ ТА РОЗРОБКА ІНТЕРНЕТ МАГАЗИНУ ОДЯГУ НА ОСНОВІ ТЕХНОЛОГІЇ REACT»

Виконав: Почтарук Стас Андрійович,
студент навчально-наукового інституту кібернетики,
інформаційних технологій та інженерії
група ІПЗ-41

_____
підпис
Керівник:
доц.. Демчук О.С.

_____
підпис

Рівне-2025

*Національний університет водного господарства та природокористування*

Навчально-науковий інститут кібернетики, інформаційних технологій та інженерії
Кафедра комп'ютерних наук та прикладної математики
Рівень вищої освіти бакалавр
спеціальність _____

«ЗАТВЕРДЖУЮ»
Завідувач кафедри _____
_____
"____" _____20__ року

**ЗАВДАННЯ**
**НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ**

Почтаруку Стасу Андрійовичу
(прізвище, ім'я, по батькові)

**1.** Тема проекту: **Проєктування та розробка інтернет магазину одягу на основі технології React**
керівник проекту к.т.н, доц. Демчук О.С.
затверджені наказом вищого навчального закладу від " 14 *" квітня 2025* року № 274
2. Термін здачі студентом закінченої роботи _____
3. Вихідні дані до проекту:
_____
_____
4. Зміст розрахунково-пояснювальної записки _____
*В першому розділі розглянуто теоретико-методологічні засади розробки веб-додатків з використанням React і Django . В другому розділі розглянуто модель компонентного розподілення front-end та back-end . Третій розділ присвячений проєктуванню інтернет-магазину одягу як веб-додатку. Четвертий розділ-програмна реалізація*
5. Перелік графічного матеріалу *мультимедійна презентація*_____6.
Консультанти розділів проекту (роботи)

| Розділ | Прізвище, ініціали та посада Консультанта | Підпис, дата | |
|---|---|---|---|
| | | завдання видав | завдання прийняв |
| | | | |
| | | | |
| | | | |

7. Дата видачі завдання

*КАЛЕНДАРНИЙ ПЛАН*

| № з/п | Назва етапів дипломного проекту (роботи) | строк виконання етапів проекту ( роботи ) | Примітка |
|---|---|---|---|
| | *Вивчення літератури за обраною тематикою* | *3.10.24-14.10.24* | *виконав* |
| | *Формулювання завдання* | *15.10.24-28.10.24* | *виконав* |
| | *Розробка алгоритму розв'язку поставленого завдання* | *28.10.24-14.01.25* | *виконав* |
| | *Здійснення програмної реалізації* | *15.01.25-15.02.25* | *виконав* |
| | *Тестування програми* | *16.02.25-9.04.25* | *виконав* |
| | *Аналіз отриманих результатів* | *10.04.25-23.04.25* | *виконав* |
| | *Загальні висновки до роботи* | *24.04.25-5.05.25* | *виконав* |
| | *Підготовка звіту кваліфікаційної роботи* | *13.05.25-22.05.25* | *виконав* |

студент _____ (  )

Керівник кваліфікаційної роботи _____ (  )

ЗМІСТ

# РЕФЕРАТ

**Об’єктом дослідження** є інформаційні системи електронної комерції, реалізовані у вигляді веб-застосунків, які підтримують транзакційні процеси, фільтрацію товарів, обробку замовлень і захищену автентифікацію користувачів.

**Предметом дослідження** є технології фронтенд- та бекенд-розробки веб-додатків на основі React та Django, архітектурні моделі побудови SPA-додатків, методи реалізації RESTful API та засоби інтеграції клієнтської частини з серверною логікою.

**Актуальність.** Стрімкий розвиток інформаційних технологій, зокрема у сфері веб-розробки та електронної комерції, зумовив необхідність створення високонадійних, адаптивних і масштабованих веб-застосунків. Інтернет-магазини, які забезпечують повноцінну взаємодію з кінцевим користувачем у режимі реального часу, стали ключовими елементами цифрової торгівлі в умовах глобалізованої економіки. За даними McKinsey, понад 70% споживачів у розвинених країнах здійснюють покупки онлайн, що вимагає від бізнесів розробки високоякісних цифрових платформ. У цьому контексті особливого значення набувають фреймворки, які дозволяють ефективно інтегрувати серверну логіку, клієнтський інтерфейс та захищену обробку даних користувачів.

**Ключові слова:** фреймворк, електронна комерція, React, Django, Python, TensorFlow, SPA-додатки.

# ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

**API** — Application Programming Interface — інтерфейс прикладного програмування

**SPA** — Single Page Application — односторінковий веб-додаток

**REST** — Representational State Transfer — стиль архітектури для створення API

**CRUD** — Create, Read, Update, Delete — базові операції над даними

**JWT** — JSON Web Token — формат токенів для автентифікації

**ORM** — Object-Relational Mapping — об'єктно-реляційне відображення

**CI/CD** —Continuous Integration / Continuous Delivery — безперервна інтеграція і доставка

**UX/UI** — User Experience / User Interface — досвід користувача / користувацький інтерфейс

**DOM** — Document Object Model — модель об’єкта документа

**JSON**— JavaScript Object Notation — формат обміну даними

**JSX** — JavaScript XML — синтаксис для опису інтерфейсів у React

**DRF** — Django REST Framework — бібліотека для побудови API у Django

**CSRF** — Cross-Site Request Forgery — міжсайтове підроблення запитів

**XSS** — Cross-Site Scripting — міжсайтове скриптування

**LCP / FID / CLS** — Core Web Vitals: Largest Contentful Paint, First Input Delay, Cumulative Layout Shift — ключові метрики ефективності веб-інтерфейсу

**SPA** — Single Page Application — односторінковий веб-додаток

**RBAC** — Role-Based Access Control — контроль доступу на основі ролей

**CMS** — Content Management System — система управління контентом

**MTV / MVC** — **Model-Template-View / Model-View-Controller** — архітектурні шаблони

**Axios** — Бібліотека для HTTP-запитів у JavaScript

# Вступ

Стрімкий розвиток інформаційних технологій, зокрема у сфері веб-розробки та електронної комерції, зумовив необхідність створення високонадійних, адаптивних і масштабованих веб-застосунків. Інтернет-магазини, які забезпечують повноцінну взаємодію з кінцевим користувачем у режимі реального часу, стали ключовими елементами цифрової торгівлі в умовах глобалізованої економіки. За даними McKinsey, понад 70% споживачів у розвинених країнах здійснюють покупки онлайн, що вимагає від бізнесів розробки високоякісних цифрових платформ. У цьому контексті особливого значення набувають фреймворки, які дозволяють ефективно інтегрувати серверну логіку, клієнтський інтерфейс та захищену обробку даних користувачів.

Фреймворк React дозволяє реалізовувати інтерфейс користувача у вигляді компонентної архітектури, забезпечуючи високу реактивність та ефективність роботи з DOM. У свою чергу, Django як серверний фреймворк з підтримкою ORM та RESTful-інтерфейсів дозволяє реалізувати стійку логіку взаємодії з базою даних, автентифікацією та контролем доступу. Поєднання цих технологій відповідає принципам сучасної SPA-архітектури (Single Page Application), де взаємодія між клієнтом і сервером відбувається через API, а логіка маршрутизації й відображення переноситься на фронтенд.

Актуальність теми дослідження обумовлена зростаючими вимогами до інтерактивності, адаптивності та безпеки веб-застосунків, зокрема у сфері електронної комерції. Умови високої конкуренції вимагають від розробників застосування новітніх технологій для забезпечення стабільності, масштабованості та UX-оптимізованої взаємодії. Застосування React та Django як повноцінного стеку розробки дозволяє створювати інтегровані рішення, що відповідають сучасним стандартам ISO/IEC 25010 та рекомендаціям OWASP для безпечної розробки. Попри наявність великої кількості шаблонних платформ, розробка власного інтернет-магазину на відкритих технологіях забезпечує повну кастомізацію бізнес-

логіки, незалежність від сторонніх сервісів та можливість глибокої інтеграції з іншими системами.

**Об'єктом дослідження** є інформаційні системи електронної комерції, реалізовані у вигляді веб-застосунків, які підтримують транзакційні процеси, фільтрацію товарів, обробку замовлень і захищену автентифікацію користувачів.

**Предметом дослідження** є технології фронтенд- та бекенд-розробки веб-додатків на основі React та Django, архітектурні моделі побудови SPA-додатків, методи реалізації RESTful API та засоби інтеграції клієнтської частини з серверною логікою.

**Метою** роботи є розробка та реалізація повнофункціонального інтернет-магазину одягу з використанням фреймворків React і Django, що забезпечує адаптивний інтерфейс, ефективну маршрутизацію, безпечну обробку даних та можливість масштабування для реальних умов електронної комерції.

Для досягнення поставленої мети необхідно вирішити наступні науково-прикладні завдання:

- Проаналізувати сучасні підходи до побудови архітектури SPA-додатків, зокрема в контексті клієнт-серверної взаємодії.
- Обґрунтувати вибір фреймворків Django та React як основних інструментів розробки та побудувати модель взаємодії між ними.
- Реалізувати RESTful API для обробки запитів, що відповідає принципам безпечної передачі та структурування даних.
- Спроєктувати базу даних інтернет-магазину з урахуванням принципів нормалізації та забезпечення цілісності даних.
- Розробити користувацький інтерфейс із застосуванням адаптивного дизайну та забезпечити інтеграцію з серверною логікою.
- Виконати модульне тестування системи, а також підготувати середовище для деплойменту в продуктивне середовище.

# Розділ 1. Теоретико-методологічні засади розробки веб-додатків з використанням React і Django

## 1.1 Сучасні підходи до архітектури веб-додатків

Архітектура сучасного веб-додатку розглядається як ключовий фактор, що впливає на адаптивність, масштабованість та підтримку системи в довготривалій перспективі. До основних архітектурних підходів належить трирівнева (n-tier) архітектура, яка розділяє логіку на блоки: презентаційний, бізнес-логіки і доступу до даних . Такий поділ дозволяє незалежно оновлювати фронтенд, бекенд або базу даних без необхідності змінювати всю систему.

Сучасні системи найчастіше інтегрують фронтенд на React і бекенд на Django шляхом створення незалежних сервісів взаємодії через RESTful API. React реалізує SPA-підхід із компонентною структурою, що забезпечує гнучкість й оптимізацію оновлення інтерфейсу користувача . Django REST Framework (DRF) підтримує побудову надійного та масштабованого API, що відповідає стандартам REST . Зокрема, в роботі [1] вказують на ефективність використання Django для реалізації класичної тришарової структури, забезпечуючи швидку передачу даних, безпеку та автоматизацію UI через API-driven підхід.

Ключовими характеристиками сучасної веб-архітектури є: взаємодія між клієнтом і сервером, що знижує складність збереження сесій й підвищує масштабованість, чітке розмежування відповідальності між фронтендом і бекендом, що суттєво спрощує підтримку й розвиток системи; увніфікований інтерфейс – ресурси ідентифікуються URI, операції маніпулювання ними виконуються через HTTP-методи ; підтримка кешування на стороні проксі чи клієнта для зниження навантаження ; можливість інтеграції компонентів як мікросервісів або розширення legacy-систем через шаровану архітектуру .

Новий рівень розвитку архітектури охоплює API-first підхід, де бекенд формує спочатку специфікацію API (через OpenAPI, RAML), забезпечуючи фронтенду документовану структуру взаємодії . Таке рішення підвищує

узгодженість між командами фронтенд- і бекенд-розробки, дозволяє автоматизувати тестування та генерацію клієнтських SDK.

Архітектурний дизайн із чіткою роздільною відповідальністю React і Django засновується на дослідженні [2], що вказує на важливість консолідованих методів управління станом, кешування та асинхронної взаємодії для підтримки продуктивності та узгодженості даних . React-спільнота пропонує впровадження Redux або Context API для контрольованого управління станом, у той час як Django оптимізує виконання бізнес-логіки через middleware і ORM.

Враховуючи вищевикладене, архітектурний підхід до інтеграції React і Django має базуватися на принципах чіткої зональної відповідальності, масштабованості, уніформного інтерфейсу, документованого API та розширюваності. Завдання подальших розділів полягає в конкретному проектуванні та валідації цих принципів на базі інтернет-магазину одягу.

### 1.2 Клієнт-серверна модель взаємодії у веб-середовищі

Клієнт-серверна модель визначається як архітектурна парадигма, за якої обчислювальні або програмні компоненти поділяються на два основні класи: клієнти, що ініціюють запити, та сервери, що обробляють ці запити і повертають відповіді . У контексті сучасного веб-додатку клієнтом виступає браузер або мобільний інтерфейс, сформований React-компонентами, а серверна сторона реалізується через Django REST API, який приймає HTTP-запити, обробляє бізнес-логіку й взаємодіє з базою даних. Ця модель характеризується чітким розділенням відповідальностей: клієнт відповідає за форму та представлення інформації, а сервер зосереджується на її обробці зберіганні та достовірності.

Розподіл функцій між клієнтом і сервером дозволяє забезпечити високу масштабованість. Зокрема, сервер може обслуговувати багато клієнтів одночасно, оптимізуючи виконання запитів і розподіл ресурсів, таких як CPU, пам’ять і процеси введення-виведення . Таке рішення покращує продуктивність, оскільки сервер

зосереджує всі ресурси на обробці запитів, а клієнт здійснює локальне рендеринг виводу, зменшуючи навантаження на центральну систему .

Обмін даними між клієнтом і сервером здійснюється через мережеві протоколи, зокрема HTTP/HTTPS, де клієнт формує запит з метою отримання або модифікації ресурсу, а сервер відповідає статусом та даними у форматі JSON або XML. Такий підхід відповідає принципу stateless, оскільки кожний запит не вимагає знання попереднього стану – це дозволяє горизонтальне масштабування шляхом додавання нових серверів без необхідності передавання сесій між ними .

Окрім простого HTTP-моделей клієнт-серверна взаємодія вимагає реалізації механізмів серіалізації, маршрутизації та обробки помилок. У Django реалізація middleware-шарів забезпечує універсальну обробку запитів і відповідей, дозволяючи додавати між ними функціонал безпеки, перевірки токенів, логування чи кешування . Інтеграція React із такими механізмами досягається за допомогою налаштування fetch/axios-запитів, обробки коду помилок та відображення належного інтерфейсу на стороні клієнта.

Ключовим викликом такої архітектури є забезпечення узгодженості даних при розпаралелюванні запитів, особливо коли React-додаток активно взаємодіє з API рівнів CRUD. Тому часто застосовуються кеш-рішення, токен-автентифікація і адаптація повторного запиту у разі виникнення помилок мережі або автентифікаційних невідповідностей. Це забезпечує стійкість до відмов і стабільність UX, що є необхідними компонентами ефективної інтернет-комерції на базі React і Django.

Таким чином, клієнт-серверна модель у сучасних веб-додатках забезпечує чітку модульну структуру, горизонтальну масштабованість, відокремлену обробку даних і можливість підтримки високої вартості взаємодії за рахунок централізації бізнес-логіки й розподіленого рендерингу UI. Реалізація цієї моделі у фреймворках React і Django створює ефективну та зручно інтегровану середу для побудови складних веб-систем.

### 1.3 Мікросервісна архітектура: переваги і недоліки

Мікросервісна архітектура, або підхід до побудови системи у вигляді набору незалежних сервісів, які взаємодіють один з одним через стандартизовані протоколи, останніми роками утвердилася як провідна парадигма у веб-розробці. В основі цього підходу лежить концепція поділу додатку на малі, автономні компоненти, кожен з яких відповідає за окрему функціональність. Мікросервіси повинні бути незалежно розгортованими і здатними до автономної еволюції, що надає системі гнучкість та стійкість до змін [3].

Однією з головних переваг мікросервісної архітектури є підвищена масштабованість. Оскільки кожен сервіс функціонує автономно, його можна масштабувати незалежно, відповідно до специфіки навантаження. Такий підхід дозволяє ефективніше розподіляти ресурси та уникати надлишкових витрат. Наприклад, у випадку інтернет-магазину, окремий сервіс для обробки замовлень можна масштабувати під високий попит у період розпродажів, не впливаючи на інші частини системи. Крім того, мікросервіси сприяють підвищенню надійності, оскільки відмова одного сервісу не призводить до відмови всієї системи, якщо правильно реалізовано механізми виявлення помилок, балансування навантаження і резервування.

З точки зору організації розробки, мікросервіси дозволяють розподілити роботу між командами, кожна з яких відповідає за конкретний сервіс. Це прискорює життєвий цикл розробки та забезпечує більшу автономію команд, що особливо важливо у великих проектах. Мікросервіси підтримують швидкий темп змін та інновацій [3], оскільки нові функціональності можуть бути реалізовані без очікування узгодження із центральною монолітною базою коду. Попри це, мікросервісна архітектура має ряд істотних недоліків, які вимагають особливої уваги. Одним із них є суттєве ускладнення інфраструктури. Замість одного процесу, що обслуговує усю логіку системи, розробнику доводиться оркеструвати десятки, а іноді й сотні, окремих сервісів. Це потребує впровадження інструментів

контейнеризації (Docker), систем оркестрації (Kubernetes), сервісів реєстрації та виявлення (service discovery), централізованого логування, трасування запитів (tracing), моніторингу й управління станом. Складність експлуатації мікросервісних систем зростає нелінійно [4] зі збільшенням кількості сервісів.

Ще однією проблемою є складність забезпечення консистентності даних. Мікросервіси зазвичай оперують ізольованими сховищами даних, що унеможливлює використання традиційних транзакцій. Тому замість жорсткої узгодженості (ACID) доводиться застосовувати патерни eventual consistency, що підвищує ризики розсинхронізації даних та помилок у бізнес-логіці.

Загалом, мікросервісна архітектура демонструє високу ефективність у високонавантажених, складних за логікою проектах, однак вона вимагає значних ресурсів для впровадження та підтримки. При розробці веб-магазину одягу з використанням React та Django її застосування має сенс лише у разі очікуваного масштабування та потреби у гнучкому, розподіленому розгортанні окремих компонентів, таких як каталог товарів, обробка платежів чи управління профілем користувача.

### 1.4 RESTful API як стандарт взаємодії компонентів

RESTful API (Representational State Transfer Application Programming Interface) є одним із найпоширеніших стандартів реалізації інтерфейсів взаємодії між клієнтською і серверною частиною веб-додатків. Вперше представлений Роєм Філдінгом у своїй докторській дисертації у 2000 році, стиль REST базується на принципах архітектурної простоти, статевості, кешованості та єдиного уніфікованого інтерфейсу. Згідно з концепцією REST, клієнт і сервер взаємодіють виключно через обмін повідомленнями HTTP-протоколу, де кожний запит описує повну інформацію, необхідну для його обробки, без збереження стану на сервері.

RESTful API дозволяє клієнтському додатку виконувати операції над ресурсами, які ідентифікуються уніфікованими URI. Ці операції, згідно з

семантикою HTTP, представлені через стандартні методи: GET — для отримання, POST — для створення, PUT/PATCH — для оновлення, DELETE — для видалення. Важливо, що REST передбачає чіткий поділ відповідальностей, де серверна частина (наприклад, Django REST Framework) відповідає лише за маніпуляцію даними, а клієнтська (наприклад, React) — за їх подання та взаємодію з користувачем.

У контексті розробки інтернет-магазину RESTful API дозволяє створити централізований шар бізнес-логіки, доступний для декількох клієнтів одночасно — веб-браузера, мобільного додатку або зовнішньої CRM. Саме це забезпечує масштабованість системи, розширюваність і можливість повторного використання коду. Використання Django REST Framework значно спрощує реалізацію REST-інтерфейсу завдяки механізмам серіалізації, автоматичному створенню ендпоінтів, вбудованому контролю доступу та інтеграції з Swagger-документацією.

Переваги REST очевидні: простота реалізації, широке розповсюдження, підтримка стандартних HTTP-засобів, можливість кешування відповідей та використання існуючих засобів авторизації, таких як OAuth2 або JWT. Проте існують і обмеження. Зокрема, REST не підходить для надто складних запитів, які вимагають агрегації багатьох ресурсів, і часто призводить до проблеми “надмірного чату” (chatty API), коли клієнт повинен виконувати численні запити, щоб побудувати одну сторінку. Крім того, REST не має вбудованого механізму обробки залежностей між ресурсами, що іноді ускладнює підтримку складних зв’язків між сутностями.

Тим не менш, у поєднанні з React RESTful API дозволяє організувати ефективну двосторонню комунікацію: React виступає як асинхронний споживач API, використовуючи бібліотеки на зразок Axios або Fetch API для взаємодії із сервером, що дозволяє динамічно оновлювати інтерфейс користувача без повного перезавантаження сторінки. Така модель є оптимальною для реалізації SPA (Single Page Application), де користувач взаємодіє з додатком як із повноцінним десктопним продуктом, отримуючи лише необхідні фрагменти даних.

Таким чином, RESTful API утворює фундаментальний механізм взаємодії у сучасних веб-додатках, забезпечуючи масштабованість, стандартизацію і гнучкість. Його поєднання з React на клієнтській стороні та Django на сервері утворює архітектурну синергію, що відповідає викликам часу і вимогам до швидкодії, масштабування та адаптивності в рамках електронної комерції.

# Розділ 2. Модель компонентного розподілення front-end та back-end

## 2.1 Деякі аспекти компонування фреймворків React та Django

Фреймворки React і Django базуються на різних архітектурних парадигмах: React підтримує компонентно-орієнтований підхід (component-based architecture), а Django слугує насамперед базою для серверної бізнес-логіки, визначеної через MVC/MTV концепцію. З позиції сучасного підходу до розробки веб-систем важливо охарактеризувати архітектурне компонування обох технологій та описати взаємодію між ними.

Архітектурна модель React орієнтована на модульність і повторне використання. Кожен UI-елемент виокремлюється в окремий компонент, який інколи відповідає навіть за дрібні взаємодії — наприклад, кнопка або форма в інтернет-магазині. Глибше розуміння дає робота, опублікована на GeeksforGeeks (2025), яка підкреслює, що компонентна структура покращує масштабованість UI-логіки та спрощує налагодження thanks to isolation of concerns and reusable code. Компоненти легко тестуються, верствуються і комбінуються в ієрархію.

Натомість Django орієнтується на Model–View–Template (MTV) патерн. Цю структуру описано в офіційних документах фреймворку як еквівалент MVC, де Django «View» виконує функції контролера, а шаблони відповідають за відображення . У сукупності це забезпечує ефективну організацію бізнес-логіки, відокремлену підготовку даних і їх представлення. Архітектурна строгость Django дає перевагу для backend-розробки інтернет-магазину, де зберігається DRY-принцип і централізація контролю доступу, автентифікації, транзакцій та ORM.

Синтез і взаємодія між React-компонентами та Django-сервісами реалізується через REST API. React-додаток викликає GET/POST запити для отримання або надсилання JSON-сукупностей ресурсів. Такий формат взаємодії забезпечує гнучкість компонентної системи та чітку декомпозицію коду. Django — відповідальний за підготовку даних, бізнес-логіку й їхню презентацію як API, React — за рендеринг інтерфейсу та інтерактивність.

Такий підхід нечутливий до змін: наприклад, можна використати Wagtail CMS або GraphQL у бекенд-частині без зміни фронтенду. У практиці спільноти часто використовують Webpack або Django Render — наприклад, Webpack налаштовується так, щоб збирати JS та CSS у папку static/, з якої Django сервить фронтенд . Інший варіант — Django Render, де всі URL проходять через Django, але відповіді видаються у форматі JSON і рендеряться React на клієнті .

### 2.1.1 Концепція компонентно-орієнтованого програмування у React

Компонентно-орієнтоване програмування (Component-Based Software Engineering, CBSE) стало ключовою парадигмою у сучасній розробці клієнтських веб-застосунків. У цьому підході система будується із незалежних, інкапсульованих та повторно використовуваних елементів — компонентів. Фреймворк React, створений у 2013 році в межах інфраструктури компанії Meta, реалізує CBSE у найчистішій формі, надаючи декларативну модель побудови користувацьких інтерфейсів, яка базується на односпрямованому потоці даних (one-way data flow) і віртуальному DOM (Virtual Document Object Model), що підвищує продуктивність.

Згідно з науковим визначенням, компонент у React — це функція або клас, який приймає властивості (props) та внутрішній стан (state) і повертає структуру у вигляді JSX, що трансформується у DOM-елементи. Як зазначає Orendorff (2022), така модель сприяє ізольованості логіки кожного елемента, спрощує тестування, налагодження та інкрементальне розширення системи. React-компоненти класифікуються як функціональні (stateless або stateful) та класові, однак у сучасній практиці перевага надається функціональним компонентам із використанням хуків (hooks).

У контексті інтернет-магазину одягу компоненти можуть реалізовувати як прості елементи, наприклад, кнопки "Купити", так і складні структури — списки товарів, каруселі зображень або фільтри за розміром та кольором. Компонент приймає три елементи (name, price, image) і відображає їх у структурованому

вигляді. Така інкапсуляція дозволяє використовувати ProductCard у багатьох частинах додатку без дублювання логіки. Для рендерингу кількох товарів достатньо використати батьківський компонент зі списком. У наведеному прикладі спостерігається типова реактивна логіка: зміна products автоматично призводить до перерендерингу списку товарів. Також забезпечується декларативність: розробник не визначає "як саме" DOM має змінюватися, а лише "що саме" потрібно відобразити.

Слід підкреслити, що компонентна архітектура у React не є лише технічною деталлю, а методологічною основою побудови фронтенд-застосунків. Використання компонентної моделі [5] значно скорочує кількість помилок при розширенні коду і сприяє швидкому прототипуванню у великих командах. У випадку інтернет-магазину одягу це означає можливість швидкої зміни структури каталогу, реалізації фільтрів, зміни стилів без порушення основного функціоналу.

Таким чином, компонентно-орієнтована модель у React забезпечує не лише зручний спосіб програмування, але й концептуальну гнучкість, яка дозволяє адаптувати фронтенд до динамічних вимог комерційного середовища. Її синергія з REST API Django є основою для побудови масштабованого, тестованого і продуктивного веб-магазину.

### 2.1.2 Принципи побудови серверної логіки у Django

Django є високорівневим фреймворком для веб-розробки мовою Python, який реалізує архітектурний шаблон Model–Template–View (MTV). У цьому патерні «Model» відповідає за структуру даних і взаємодію з базою даних, «View» обробляє запити та формує відповідь, а «Template» відповідає за генерацію HTML-контенту. Однак у випадку побудови RESTful API для взаємодії з React використання шаблонів не є необхідним — серверна логіка концентрується у моделях, серіалізаторах і представленнях (views), що повертають дані у форматі JSON.

Django REST Framework (DRF), як надбудова над стандартним Django, забезпечує зручні засоби для реалізації такої логіки.

Основою будь-якої серверної логіки у Django є модель, яка є абстракцією над таблицею в реляційній базі даних. Для інтернет-магазину одягу типовою є модель продукту, яка може мати поля, що описують назву товару, ціну, розмір, колір, категорію тощо. Наприклад:

```
from django.db import models
from django.urls import reverse
from django.db.models import Index

class Category(models.Model):
    name = models.CharField(max_length=100, db_index=True)
    slug = models.SlugField(max_length=100, unique=True)

    class Meta:
        ordering = ('name',)
        verbose_name = 'Категорія'
        verbose_name_plural = 'Категорії'

    def __str__(self):
        return self.name

    def get_absolute_url(self):
        return reverse('myshop:product_list_by_category', args=[self.slug])


class Product(models.Model):
    category = models.ForeignKey(
        Category,
        related_name='products',
        on_delete=models.CASCADE
```

Ця модель автоматично транслюється Django у відповідну SQL-схему, що забезпечує збереження структурованих даних. Модель також дозволяє реалізувати доменну логіку за допомогою методів класу (наприклад, фільтрація доступних товарів, сортування, обчислення знижок тощо).

Для забезпечення взаємодії з React-інтерфейсом необхідно серіалізувати об’єкти моделі у JSON-формат. Це досягається за допомогою DRF-серіалізаторів:

```python
from rest_framework import serializers
from .models import Product, Category

class CategorySerializer(serializers.ModelSerializer):
    class Meta:
        model = Category
        fields = ['id', 'name']

class ProductSerializer(serializers.ModelSerializer):
    category = CategorySerializer(read_only=True)

    class Meta:
        model = Product
        fields = ['id', 'name', 'description', 'price', 'size', 'image', 'category']
```

Серіалізатор виступає посередником між Python-об'єктами та JSON-представленням, дозволяючи також здійснювати валідацію вхідних даних при POST або PUT-запитах.

Основу серверної логіки в DRF становлять представлення (views), які реалізують поведінку при обробці HTTP-запитів. Найбільш зручним і поширеним варіантом є використання ModelViewSet, який поєднує всі CRUD-операції:

```python
from rest_framework import viewsets
from .models import Product
from .serializers import ProductSerializer

class ProductViewSet(viewsets.ReadOnlyModelViewSet):
    queryset = Product.objects.all()
    serializer_class = ProductSerializer
```

Це представлення автоматично надає підтримку для методів GET, POST, PUT, DELETE і відповідає запитам React-клієнта без потреби писати зайву логіку. Зокрема, GET-запит до /api/products/ поверне список усіх товарів, а GET-запит до /api/products/5/ — детальну інформацію про товар з id = 5.

Таким чином, серверна логіка у Django побудована на фундаменті суворої структури: модель — як опис даних, серіалізатор — як транслятор між базою та API, і представлення — як обробник запитів. Цей підхід дозволяє концентрувати бізнес-логіку у відповідних шарах, забезпечує зрозумілу структуру проєкту, а також сприяє тестованості, масштабованості та повторному використанню компонентів.

## 2.2 Системи маршрутизації та шаблонізації у сучасних фреймворках

Системи маршрутизації та шаблонізації є невід'ємною частиною сучасних фреймворків і відіграють критичну роль у формуванні логіки взаємодії між користувачем та додатком. У контексті застосування React і Django, маршрутизація реалізує логіку обробки запитів за URL-шляхами, а шаблонізація забезпечує представлення вмісту — у вигляді HTML або віртуального DOM. У традиційних серверно-орієнтованих системах маршрутизація і шаблонізація здійснюються на боці бекенду, однак у сучасних SPA (Single Page Applications) значна частина цих функцій переміщується на клієнтський бік.

У Django маршрутизація реалізується на базі конфігураційного файлу urls.py, у якому кожен URL пов'язується з відповідною функцією обробника або класом-представленням. Система маршрутизації базується на регулярних виразах або шляхоорієнтованій системі розпізнавання. Наприклад, маршрут, який обробляє запити до списку товарів у REST API, виглядає наступним чином:

```
(1)  # urls.py (Django)
(2)  from django.urls import path
(3)  from .views import ProductListView
(4)  urlpatterns = [
(5)      path('products/', ProductListView.as_view(), name='product-list'),
(6)  ]
```

При використанні Django REST Framework типовим є застосування роутерів, які автоматизують створення маршрутів для набору представлень:

```
from rest_framework.routers import DefaultRouter
from .views import ProductViewSet
app_name = 'shop'

router = DefaultRouter()
router.register(r'products', ProductViewSet)

urlpatterns = router.urls
```

Цей механізм дозволяє мінімізувати шаблонний код і дотримуватись REST-принципів. Крім того, кожен маршрут автоматично підтримує вербальні HTTP-

методи — GET, POST, PUT, DELETE — без необхідності ручного прописування кожного шляху.

З іншого боку, у React маршрутизація реалізується за допомогою бібліотеки React Router, яка забезпечує можливість навігації між компонентами без перезавантаження сторінки. Це створює ефект "односторінкового додатку", в якому URL змінюється, але сторінка не оновлюється повністю, що істотно покращує UX.

Таким чином, маршрутизація у Django орієнтована на RESTful API та обробку HTTP-запитів, тоді як у React — на віртуальну навігацію між сторінками інтерфейсу. Шаблонізація у Django є декларативною і традиційною, у React — інтерактивною та інтегрованою в логіку компонентів. Комбінування обох фреймворків дає змогу реалізувати ефективну та продуктивну клієнт-серверну архітектуру, де маршрутизація і шаблонізація розділені, але діють скоординовано.

## 2.3. Підходи до забезпечення якості веб-додатків

### 2.3.1. Тестування компонентів та API

Забезпечення якості веб-додатків є ключовою вимогою у процесі розробки масштабованих та надійних систем. У випадку взаємодії фреймворків Django та React така якість досягається шляхом систематичного тестування як клієнтської частини, що реалізована за допомогою компонентів, так і серверного API, що визначає функціональну поведінку застосунку. Тестування у цьому контексті охоплює декілька рівнів: юніт-тести, інтеграційні тести та енд-то-енд тестування, що узгоджується з сучасними підходами [14].

На стороні Django найважливішим є тестування моделей, представлень (views) і API-інтерфейсів. Усі тести вбудовано в інфраструктуру Django через модуль unittest, що дозволяє здійснювати перевірку логіки моделей, контролю доступу, правильності HTTP-відповідей і роботи серіалізаторів.

Для перевірки API логіка концентрується навколо перевірки статусів відповідей, наявності очікуваних полів у JSON і коректності обробки POST-запитів.

Django REST Framework надає модуль APITestCase, що значно полегшує тестування RESTful API.

На стороні React найбільш поширеним підходом є використання бібліотек @testing-library/react та Jest, які забезпечують засоби для тестування компонентів у ізоляції. Основна мета полягає в перевірці виводу JSX залежно від вхідних даних (props), а також реакції компонентів на події.

Крім тестування окремих компонентів, важливим є інтеграційне тестування взаємодії React із Django через API. Для цього можуть використовуватись інструменти типу Cypress або Playwright, які емулюють реальні дії користувача у браузері. Наприклад, тест додавання товару до кошика може включати перевірку наявності товару після запиту до API, кліку на кнопку і відображення зміни стану інтерфейсу.

### 2.3.2. Інструменти CI/CD для автоматизації розгортання

Сучасна розробка веб-додатків, зокрема побудованих на зв'язці React і Django, вимагає високого рівня автоматизації процесів побудови, тестування та розгортання. Ці цілі досягаються через реалізацію концепцій CI (Continuous Integration) і CD (Continuous Delivery / Continuous Deployment). Згідно з визначенням [6], CI/CD — це набір практик, що забезпечують автоматичне збирання, перевірку й доставку коду до продуктивного середовища без ручного втручання, що дозволяє скоротити час до релізу, підвищити стабільність і швидко виявляти дефекти.

У контексті React і Django, інструменти CI/CD орієнтовані на послідовну інтеграцію фронтенд- і бекенд-компонентів, перевірку тестів, збірку артефактів, запуск лінтерів, деплоймент на хмарну інфраструктуру, зокрема Heroku, DigitalOcean або AWS.

Одним з найпопулярніших рішень для CI/CD є GitHub Actions, оскільки воно безпосередньо інтегрується з системою контролю версій GitHub і дозволяє описувати всі стадії CI/CD як YAML-воркфлоу.

Ключовою особливістю систем типу GitHub Actions є можливість створення середовища для staging перед production-релізом, що дозволяє перевірити нові функціональні зміни без впливу на живу систему. У разі виявлення помилки її усунення вносяться локально в GitHub, після чого автоматично запускається новий пайплайн CI/CD.

Серед альтернативних рішень також варто згадати GitLab CI, який дозволяє більш гнучко управляти пайплайнами через .gitlab-ci.yml, а також Jenkins, що, попри складнішу конфігурацію, забезпечує глибоку кастомізацію через плагіни та scripting Groovy.

У статті [7] підкреслено, що без CI/CD сучасні динамічні проєкти зазнають серйозних труднощів з керованістю: зростають витрати на тестування, підвищується ймовірність людських помилок і втрачається швидкість виведення нових фіч.

В умовах React + Django-екосистеми реалізація CI/CD-пайплайнів забезпечує:

- безперервну валідацію синтаксису та логіки компонентів;
- повну перевірку REST API до кожного злиття гілки;
- автоматичне формування production-версій фронтенду (npm run build);
- регулярне оновлення backend-бази через міграції (python manage.py migrate);
- безпечний деплой на staging/production сервер.

У результаті, CI/CD не є додатковою функціональністю, а невід'ємною частиною сучасної інженерної практики, що гарантує якість, безперервність і прогнозованість роботи веб-додатку. Реалізація таких пайплайнів має бути обов’язковою умовою для будь-якої системи, яка претендує на масштабованість, стійкість і відповідність вимогам DevOps-культури.

### 2.3.3. Метрики ефективності веб-застосунків

Ефективність веб-застосунку визначається здатністю системи швидко, стабільно та з передбачуваною реакцією обробляти запити користувачів у

різноманітних умовах навантаження. Вона охоплює як технічні характеристики (продуктивність, час відгуку, споживання ресурсів), так і поведінкові аспекти, пов'язані з досвідом користувача. У контексті архітектури React + Django метрики ефективності дозволяють кількісно оцінити як фронтенд, так і серверну частину додатку, що є передумовою для оптимізації продуктивності, масштабованості та UX.

Згідно з дослідженням [8], існує кілька груп показників, що застосовуються до веб-застосунків. По-перше, це метрики продуктивності бекенду, зокрема середній час обробки запиту (request processing time), пропускна здатність (requests per second), кількість одночасних з'єднань, швидкість відповіді API. Django як бекенд-сервер може бути профільований за допомогою інструментів типу django-debug-toolbar, Silk, або вбудованих логерів, які фіксують час виконання запитів.

Для оцінки ефективності React-фронтенду найчастіше застосовується набір Core Web Vitals, запропонований Google у 2021 році. Він включає такі метрики:

- Largest Contentful Paint (LCP) — час завантаження найбільшого елемента у вікні браузера. Значення менше 2.5 с вважається хорошим.
- First Input Delay (FID) — час до відповіді на першу взаємодію користувача.
- Cumulative Layout Shift (CLS) — стабільність макету (чи зміщується вміст при завантаженні).

Інструмент Lighthouse, що вбудований у Chrome DevTools, дозволяє автоматично збирати ці показники, а також генерувати аналітичні звіти з рекомендаціями. У випадку React-додатків, зібраних за допомогою create-react-app, ці метрики інтегруються через reportWebVitals.js, який може бути пов'язаний із Google Analytics або власними логерами:

```
(1) // index.js
(2) import reportWebVitals from './reportWebVitals';
(3) reportWebVitals(console.log);
```

Для більш глибокого аналізу використовуються React Profiler та Performance API. Вони дають змогу відстежити час рендерингу компонентів, визначити вузькі місця в оновленні DOM, побачити надмірні перерендери.

Крім того, системи моніторингу типу New Relic, Datadog, або Sentry Performance інтегруються з обома частинами застосунку — React і Django — і дозволяють відстежувати SLA-метрики, такі як Apdex (Application Performance Index), error rate, latency distribution. У науковій роботі Paduraru et al. (2022) наголошується, що впровадження таких метрик у CI/CD-пайплайни дозволяє виявити деградацію продуктивності ще до того, як вона проявиться у продакшн-середовищі.

Метрики ефективності також мають стратегічне значення для прийняття архітектурних рішень. Наприклад, якщо зафіксовано зростання LCP до 4 секунд у React-додатку, це може свідчити про необхідність реалізації lazy-loading зображень або оптимізації шрифтів. Якщо ж на стороні Django API виявлено збільшення часу відповіді до 800 мс, слід перевірити ефективність запитів до бази даних, індексацію полів, використання кешу (Redis, Memcached).

Узагальнюючи, можна стверджувати, що метрики ефективності є не лише засобом об'єктивного контролю, а й інструментом зворотного зв'язку для прийняття інженерних рішень. Без їх впровадження неможливо забезпечити гарантовану якість досвіду користувача, що особливо критично для електронної комерції, де навіть незначні затримки можуть призвести до втрати конверсії.

### 2.3.4. Безпека даних у веб-розробці: автентифікація, CSRF, XSS

Безпека веб-застосунків є однією з найкритичніших складових сучасного розробницького процесу, особливо в галузі електронної комерції, де обробляються персональні дані користувачів, фінансові транзакції та конфіденційна інформація. У парадигмі клієнт-серверної архітектури з розділенням фронтенду (React) та бекенду (Django) зростає кількість векторів атак, що вимагає системного

застосування принципів безпечної розробки. Згідно з OWASP (Open Web Application Security Project), до найбільш поширених загроз відносять міжсайтові скриптові атаки (XSS), міжсайтову підробку запитів (CSRF) та несанкціонований доступ до обмежених ресурсів.

Перший рівень захисту полягає в реалізації надійної системи автентифікації та авторизації. У фреймворку Django базовий механізм автентифікації базується на сесіях і cookies. Проте у взаємодії з React-додатком, який функціонує як SPA, переважно використовується токен-орієнтована автентифікація, зокрема через JSON Web Tokens (JWT). Django REST Framework із пакетом djangorestframework-simplejwt дозволяє реалізувати генерацію, валідацію й оновлення токенів:

```
(1)  # settings.py
(2)  REST_FRAMEWORK = {
(3)      'DEFAULT_AUTHENTICATION_CLASSES': (
(4)          'rest_framework_simplejwt.authentication.JWTAuthentication',
(5)      ),
(6)  }
```

Користувач отримує access- і refresh-токени через POST-запит до ендпоінту /api/token/ і надалі надсилає заголовок Authorization: Bearer <token> з кожним запитом. На клієнтській стороні React здійснює збереження токенів у localStorage або memory, з подальшим автоматичним додаванням до запитів.

Наступною загрозою є CSRF — атака, за якої зловмисник примушує браузер користувача надіслати несанкціонований запит до веб-застосунку, де користувач вже автентифікований. Django за замовчуванням захищає усі POST-запити за допомогою механізму CSRF-token, але при використанні REST API цей механізм втрачає ефективність. Тому при JWT-автентифікації рекомендовано відключати CSRF, але компенсувати це додатковими заголовками X-Requested-With, перевіркою CORS (Cross-Origin Resource Sharing) та налаштуванням заголовків SameSite.

У випадку з автентифікацією через сесії (наприклад, в admin-панелі) CSRF залишається активним. У шаблонах Django CSRF-токен вставляється автоматично через тег {% csrf_token %}:

```
(1)  <form method="POST">
(2)   {% csrf_token %}
(3)   <input type="text" name="username">
(4)   <button type="submit">Login</button>
(5)  </form>
```

На React-стороні реалізація CSRF-захисту вимагає попереднього отримання токена з cookie або ендпоінту /csrf/:

```
(1)  // csrf.js
(2)  import axios from 'axios';
(3)  export async function getCSRFToken() {
(4)    const response = await axios.get('/api/csrf/');
(5)    axios.defaults.headers.post['X-CSRFToken'] = response.data.csrfToken;
(6)  }
```

Ще одним ключовим викликом є запобігання XSS-атакам, які дозволяють вставляти зловмисний JavaScript у веб-сторінки. React за своєю природою забезпечує захист від XSS через JSX — всі вставлені значення автоматично екрануються. Наприклад, вираз {userInput} не виконає скрипт, навіть якщо userInput дорівнює <script>alert('xss')</script>.

Однак загроза XSS залишається реальною у випадку навмисного використання dangerouslySetInnerHTML — це місце, де розробник повинен гарантувати очищення HTML через бібліотеки на зразок DOMPurify. Наприклад:

```
(1)  import DOMPurify from 'dompurify';
(2)  function SafeContent({ html }) {
(3)    const cleanHtml = DOMPurify.sanitize(html);
(4)    return <div dangerouslySetInnerHTML={{ __html: cleanHtml }} />;
(5)  }
```

На стороні Django важливо уникати небезпечної ін’єкції HTML у шаблони без екранування. За замовчуванням, Django автоматично екранує всі змінні, але розробники можуть помилково використати |safe, що призведе до вразливості.

Багатошаровий підхід до безпеки [10] є найбільш ефективним: автентифікація — через JWT, CSRF — через строго контрольовані CORS-заголовки та HTTP-only cookies, XSS — через обмеження доступу до вбудованого HTML і автоматичну екранізацію. Така стратегія дозволяє нейтралізувати більшість найбільш небезпечних векторів атак, описаних у звітах OWASP Top-10.

Таким чином, безпека веб-застосунку, реалізованого за допомогою React і Django, вимагає скоординованої роботи на всіх рівнях: серверному, клієнтському і мережевому. Належна реалізація автентифікації, захисту від CSRF і XSS — це не просто технічна вимога, а критичний елемент довіри до системи з боку користувачів та бізнесу.

# Розділ 3. Проєктування інтернет-магазину одягу як веб-додатку

## 3.1. Модель предметної області електронної комерції

### 3.1.1. Структура бізнес-процесів онлайн-торгівлі

Бізнес-процеси онлайн-торгівлі — це сукупність логічно взаємопов'язаних дій, спрямованих на забезпечення повного життєвого циклу електронної комерції: від пошуку товару користувачем до оформлення та обробки замовлення. У рамках архітектури веб-магазину, побудованого з використанням React і Django, бізнес-процеси реалізуються через модульні сервіси, які охоплюють маркетингову взаємодію, управління товарним каталогом, фільтрацію й сортування, логістику замовлень, обробку платежів та підтримку користувача.

Згідно з моделлю Porter's Value Chain [11], бізнес-процеси електронної торгівлі можна класифікувати на основні (операційні) і допоміжні. У веб-магазині одягу основними процесами є перегляд каталогу, вибір розмірів та кольорів, додавання до кошика, оформлення замовлення, оплата, підтвердження доставки, а допоміжними — реєстрація, автентифікація, обробка запитів підтримки, перегляд історії замовлень.

Формально структура бізнес-процесу у Django визначається як набір моделей (наприклад, Product, Cart, Order), які взаємодіють через ORM-зв'язки та API. Водночас, у React формується динамічний інтерфейс, що реагує на зміну стану користувача або відповіді сервера. Наприклад, типовий процес оформлення замовлення охоплює:

1. Пошук товару → 2. Перегляд детальної сторінки → 3. Вибір параметрів (розмір, колір) → 4. Додавання до кошика → 5. Оформлення замовлення → 6. Оплата → 7. Підтвердження.

Цей ланцюг реалізується через поетапну логіку між фронтендом і бекендом.

Важливим елементом є автоматизація переходу між етапами бізнес-процесу. У Django це досягається за рахунок сигнальних механізмів (signals), які дозволяють, наприклад, після створення замовлення надсилати лист-підтвердження:

```
(1)  from django.db.models.signals import post_save
(2)  from django.dispatch import receiver
(3)  from .models import Order
(4)  @receiver(post_save, sender=Order)
(5)  def send_order_confirmation(sender, instance, created, **kwargs):
(6)      if created:
(7)          # логіка надсилання email
(8)          print(f"Підтвердження замовлення {instance.id} надіслано.")
```

Кожен бізнес-процес можна описати у вигляді BPMN-діаграми або псевдокоду з переходами станів. Наприклад, статус замовлення може змінюватися від new → paid → shipped → delivered — кожен з яких обробляється окремим контролером API і відповідає за запуск відповідної дії.

Таким чином, структура бізнес-процесів у веб-магазині одягу, побудованому на React + Django, визначається взаємодією моделей, REST API та компонентів, що реалізують кожен крок взаємодії користувача з системою. Від правильності моделювання цих процесів залежить надійність логіки додатку, ефективність виконання операцій та рівень задоволеності користувача.

### 3.1.2. Рольова модель взаємодії користувачів

Рольова модель у веб-додатках електронної комерції визначає набір дозволених дій та обмежень, що застосовуються до кожного типу користувача відповідно до їх функціонального призначення в системі. У контексті інтернет-магазину одягу, побудованого на фреймворках Django і React, рольова модель формує каркас авторизації, контролю доступу та відображення інтерфейсів згідно з роллю суб'єкта: покупця, адміністратора, менеджера підтримки або менеджера товарів.

У системах з архітектурою клієнт–сервер, реалізація ролей здійснюється зазвичай на серверній стороні, де перевіряються права доступу до певних ресурсів. Django містить вбудовану систему аутентифікації та авторизації на базі моделі User та груп (Group) і дозволів (Permission). Кожен користувач може бути віднесений до

однієї чи кількох груп, які мають відповідний набір дозволів. Наприклад, розмежування доступу до списку замовлень може бути реалізоване так:

```
(1)  from rest_framework.permissions import BasePermission
(2)  class IsAdminOrReadOnly(BasePermission):
(3)      def has_permission(self, request, view):
(4)          return request.method in ['GET'] or request.user.is_staff
```

Використання цього класу в представленнях обмежує можливості некваліфікованих користувачів до лише читання, тоді як адміністраторам дозволяється змінювати, додавати або видаляти дані.

На рівні моделей рольова логіка може реалізовуватися через логічні прапорці (is_staff, is_superuser) або через розширення моделі користувача.

Такий підхід дозволяє прямо пов’язувати логіку контролю доступу з атрибутами об’єкта користувача. У представленнях або серіалізаторах можна додати перевірку ролі:

```
(1)  def list_orders(request):
(2)      if request.user.role != 'manager':
(3)          return HttpResponseForbidden()
```

На клієнтській стороні у React компонентне дерево також адаптується залежно від ролі користувача. Це забезпечує контроль над тим, які компоненти інтерфейсу будуть доступні. Наприклад:

```
(1)  import React from 'react';
(2)  function Dashboard({ user }) {
(3)    if (user.role === 'admin') {
(4)      return <AdminPanel />;
(5)    } else if (user.role === 'manager') {
(6)      return <ManagerView />;
(7)    } else {
(8)      return <CustomerView />;
```

Для більш складних додатків рольову модель можна реалізувати у вигляді контексту (React Context) або глобального стану через Redux. Це дозволяє централізовано зберігати поточну роль користувача після логіну та динамічно обмежувати доступ до маршрутів або елементів:

```
(1) import { Navigate } from 'react-router-dom';
(2) function ProtectedRoute({ user, allowedRoles, children }) {
(3)   return allowedRoles.includes(user.role)
(4)     ? children
(5)     : <Navigate to="/unauthorized" />;
(6) }
```

Додатково рольова модель впливає на бізнес-логіку, наприклад: лише користувачі з роллю admin можуть змінювати статуси замовлень; manager — обробляти повернення; customer — створювати нові замовлення. Згідно з рекомендаціями National Institute of Standards and Technology (NIST SP 800-162), впровадження RBAC (Role-Based Access Control) дає змогу формалізувати права доступу, зменшити кількість ручного управління дозволами та підвищити відповідність вимогам безпеки.

Для зберігання логіки ролей на більш високому рівні абстракції можна застосовувати бібліотеки, такі як django-guardian, яка забезпечує об’єктно-орієнтовану авторизацію (per-object permissions), або casbin (через інтеграцію з DRF), яка дозволяє гнучко будувати політики доступу.

Узагальнюючи, рольова модель у Django+React базується на строгому поділі відповідальностей: сервер гарантує безпеку, фронтенд — зручність інтерфейсу. Її наявність дозволяє динамічно масштабувати систему за рахунок гнучкого контролю прав доступу, мінімізує ризики несанкціонованого доступу та підвищує логічну стійкість структури бізнес-процесів.

### 3.1.3. Каталогізація товарів і фільтрація за атрибутами

Каталогізація товарів у веб-додатках електронної комерції — це процес структурного організування асортименту продукції для забезпечення зручного пошуку, порівняння й вибору товарів кінцевим користувачем. Ефективна реалізація цієї функції вимагає системного підходу до моделювання товарних категорій, атрибутів і варіантів, а також впровадження механізмів фільтрації та сортування. У зв'язці React + Django дана функціональність реалізується як через структуру моделей на сервері, так і через динамічну інтеракцію інтерфейсу користувача на клієнтській стороні.

На бекенді базовим елементом каталогізації виступає модель Product, яка описує товарну одиницю, та модель Category, яка формує ієрархію класифікації.

Категорії можуть бути організовані у вкладену структуру (tree-like hierarchy), використовуючи бібліотеки на зразок django-mptt або django-treebeard, що забезпечує ефективну побудову меню, підкатегорій і фільтрації за контекстом.

Фільтрація за атрибутами реалізується через GET-параметри у запиті до API. Django дозволяє здійснювати складну фільтрацію через Q-об'єкти та бібліотеку django-filter, яка генерує логіку на основі атрибутів моделі.

Така реалізація забезпечує двосторонній зв'язок між параметрами користувача і результатами на сторінці. Для збереження фільтрів при навігації можна використовувати URL-параметри, наприклад через useSearchParams або react-router.

Окрім цього, важливим є сортування. Django API підтримує параметр ordering, який інтегрується через OrderingFilter. А на React-стороні це може виглядати як випадаючий список, що додає параметр ordering=price до запиту.

Згідно з рекомендаціями [13], ефективна каталогізація повинна поєднувати фільтри, сортування, категорії й ключові атрибути у єдиний інтерфейс, який не перевантажує користувача, але надає гнучкість пошуку.

У результаті, реалізація каталогізації у Django + React є процесом глибокої інтеграції структурованої серверної логіки й клієнтської адаптивної взаємодії. Вона є критичним чинником комерційного успіху веб-магазину, оскільки напряму впливає на поведінкові метрики: конверсію, час на сайті, відмови та завершені покупки.

### 3.1.4. Структура транзакцій та обробка замовлень

Транзакції у веб-магазині виступають як логічні одиниці дій, що охоплюють процес замовлення товарів, підтвердження оплати, збереження інформації у базі даних і ініціювання доставки. З погляду програмної архітектури, транзакції мають забезпечувати атомарність (усі дії виконуються або не виконуються зовсім), цілісність (узгодженість даних), ізольованість (відсутність конфліктів при паралельних операціях) і довговічність (збереження результату навіть після збою системи), відповідно до принципів ACID [14]. Django, як фреймворк з ORM, надає повноцінну підтримку транзакцій через менеджери контексту, тоді як React реалізує логіку підтвердження замовлення і взаємодію з API.

На серверному рівні базовою структурою є моделі Order та OrderItem, які відображають замовлення і його зміст:

```
(1)  from django.db import models
(2)  from django.contrib.auth.models import User
(3)  class Order(models.Model):
(4)      user = models.ForeignKey(User, on_delete=models.CASCADE)
(5)      created = models.DateTimeField(auto_now_add=True)
(6)      paid = models.BooleanField(default=False)
(7)      status = models.CharField(max_length=20, default='new')
(8)  class OrderItem(models.Model):
(9)      order = models.ForeignKey(Order, related_name='items',
     on_delete=models.CASCADE)


(10)     product = models.ForeignKey('Product', on_delete=models.PROTECT)
(11)     quantity = models.PositiveIntegerField()
```

Обробка замовлення — це процес, у якому відбувається створення об'єкта Order, додавання до нього позицій (OrderItem), розрахунок підсумку і зміна статусу після оплати. Для забезпечення цілісності даних при створенні повного замовлення використовується транзакційний контекст:

```
(1)  from django.db import transaction
(2)  @transaction.atomic
(3)  def create_order(request):
(4)      data = request.data
(5)      order = Order.objects.create(user=request.user)
(6)      for item in data['items']:
(7)          OrderItem.objects.create(
(8)              order=order,
(9)              product_id=item['product_id'],
(10)             quantity=item['quantity']
(11)         )
(12)     return Response({'order_id': order.id}, status=201)
```

Якщо під час створення будь-якої позиції виникає помилка, вся транзакція буде скасована — тобто замовлення не буде частково збережене.

Для імітації платіжного процесу можна застосувати сторонні JavaScript SDK платіжних систем. Після завершення транзакції, платіжна система повертає відповідь на бекенд (вебхук) або на клієнт — тоді React надсилає PATCH-запит із ознакою успішної оплати.

Ключовим викликом у моделюванні транзакцій є необхідність синхронізації між бекендом, платіжною системою і станом інтерфейсу. Помилкові стани можуть призвести до подвійної оплати, втрати замовлення або неузгодженості в обліку. Саме тому транзакції мають супроводжуватись відповідним журналюванням (logging) і аудитом, а також механізмами повторної обробки невдалих платежів.

Таким чином, структура транзакцій у системі на основі Django і React має ґрунтуватися на атомарному виконанні серверних дій, гарантованій доставці

повідомлень від платіжної системи і реактивному оновленні інтерфейсу. Правильне моделювання обробки замовлень забезпечує не лише коректну роботу магазину, а й довіру користувача до системи.

## 3.2. База даних та структура API для інтернет-магазину

### 3.2.1. Нормалізоване моделювання сутностей

Моделювання сутностей у веб-додатку — це процес формального опису об’єктів предметної області, які мають бути збережені, оброблені або відображені системою. У контексті інтернет-магазину одягу нормалізація сутностей забезпечує цілісність даних, виключення надлишковості та оптимізацію запитів до бази даних. Django як фреймворк з вбудованою ORM дозволяє реалізувати ці принципи на рівні визначення моделей, дотримуючись норм форми реляційної алгебри (Codd, 1972).

Нормалізація передбачає декомпозицію складних структур даних на окремі таблиці із встановленими зв’язками між ними, зазвичай у формі зв’язків «один до одного», «один до багатьох» або «багато до багатьох». Основні сутності в інтернет-магазині одягу включають товари (Product), категорії (Category), бренди (Brand), атрибути (Size, Color), зображення (ProductImage), користувачів (User) та замовлення (Order). Кожна з них описується у вигляді окремої моделі.

Бренди, кольори та розміри винесені в окремі таблиці, що дозволяє забезпечити:

- цілісність довідників;
- гнучкість при фільтрації;
- повторне використання значень;
- зменшення надмірності даних.

Сутність ProductImage реалізується окремо як зв’язана таблиця «один до багатьох», що дозволяє мати кілька зображень для одного товару:

```
(1)  class ProductImage(models.Model):
(2)      product = models.ForeignKey(Product, on_delete=models.CASCADE,
     related_name='images')
(3)      image = models.ImageField(upload_to='products/')
(4)      is_primary = models.BooleanField(default=False)
```

Таким чином, нормалізоване моделювання сутностей у Django забезпечує структурну узгодженість, логічну цілісність і оптимізовану роботу з великим обсягом даних. У веб-магазині одягу це дозволяє підтримувати складну систему атрибутів, швидку фільтрацію, багатомовність і повторне використання метаінформації товарів без надлишковості та дублювання логіки.

### 3.2.2. Реалізація CRUD-операцій через Django REST Framework

CRUD-операції (Create, Read, Update, Delete) становлять основу функціональної взаємодії користувача з ресурсами веб-додатку. У парадигмі RESTful API, яку реалізує Django REST Framework (DRF), кожна з цих операцій відповідає конкретним HTTP-методам: POST — для створення, GET — для читання, PUT/PATCH — для оновлення, DELETE — для видалення. Реалізація CRUD-функціональності у DRF є однією з найсильніших сторін цього фреймворку завдяки гнучкості, автоматизації та підтримці масштабованого контролю доступу.

Контроль доступу до CRUD-операцій регулюється через permission_classes, які дозволяють налаштовувати доступ на основі ролей, груп або політик. Для складніших сценаріїв можна використовувати об’єктно-рівневі дозволи (has_object_permission), що дозволяють, наприклад, змінювати лише власні товари.

Таким чином, реалізація CRUD-операцій у Django REST Framework є стандартизованою і водночас гнучкою. Вона підтримує автоматичну генерацію

ендпоінтів, логічне розділення рівнів доступу, зручну інтеграцію з React та широке розширення за рахунок middleware, логування, throttling та кастомних правил валідації. Це робить її ключовою складовою у побудові надійного та маштабованого інтерфейсу обміну даними між клієнтською частиною і сервером.

### 3.2.3. Стандартизація форматів запитів і відповідей

У веб-розробці стандартизація форматів запитів і відповідей є ключовим фактором для забезпечення сумісності між клієнтом та сервером, особливо у багаторівневих архітектурах з чітко розділеними фронтендом (React) і бекендом (Django REST Framework). У відповідності до принципів REST [13], уніфіковані формати взаємодії забезпечують предикативність, інтероперабельність та автоматизовану обробку, що є критично важливим для розширюваних API.

Найбільш поширеним форматом передачі даних у REST API є JSON (JavaScript Object Notation). Він підтримується нативно в React, Node.js, Django, DRF і більшості сучасних браузерів. JSON є компактним, текстовим, легко парсованим форматом, що дозволяє відображати вкладені структури, списки, булеві значення та null.

У Django REST Framework відповідь за замовчуванням форматується як JSON, якщо не вказано інше.

Для стандартизації запитів (особливо POST і PUT) важливо формувати тіла запиту відповідно до очікуваної структури. Наприклад, створення товару у JSON-форматі:

```
(1)  {
(2)    "name": "Сорочка біла",
(3)    "price": "599.99",
(4)    "size": ["M"],
(5)    "color": ["Білий"],
(6)    "available": true
```

На боці Django дані автоматично десеріалізуються в об'єкт Python, і проходять через перевірку is_valid() у серіалізаторі:

```
def create(self, request):
    serializer = ProductSerializer(data=request.data)
    if serializer.is_valid():
        serializer.save()
        return Response(serializer.data, status=201)
```

Важливим елементом є структура помилок. У DRF вона є предикативною, тобто кожна помилка повертається у форматі словника з полями, які не пройшли валідацію:

```
{
  "price": ["Це поле є обов'язковим."],
  "name": ["Назва не може бути порожньою."]
}
```

Це дозволяє React-компонентам точно локалізувати проблему і відобразити її користувачеві.

Окрім основного контенту, важливу роль у стандартизації відіграють заголовки HTTP-запитів і відповідей. Сервер має чітко вказати тип відповіді:
Content-Type: application/json

Також для забезпечення міждоменної взаємодії (CORS) сервер додає:
Access-Control-Allow-Origin: *

У React усі запити формуються з відповідними заголовками, зокрема:

```
axios.post('/api/products/', data, {
  headers: {
    'Content-Type': 'application/json',
    'Authorization': `Bearer ${token}`
  }
});
```

Іншим аспектом є пагінація. У DRF вона реалізується у вигляді структури з метаданими:

```
{
  "count": 48,
  "next": "/api/products/?page=2",
  "previous": null,
  "results": [
    { "id": 1, "name": "Куртка демісезонна", ... },
    ...
  ]
}
```

Цей формат дозволяє React-клієнту будувати інтерфейси типу "завантажити більше", реалізувати навігацію між сторінками, або відображати загальну кількість результатів.

Таким чином, стандартизація форматів запитів і відповідей у Django REST Framework та React базується на послідовному застосуванні JSON як основного формату, уніфікованій структурі помилок, правильному використанні HTTP-заголовків і підтримці автоматизованої документації. Це забезпечує не лише надійність обміну даними, а й підвищує продуктивність командної розробки та якість взаємодії між клієнтом і сервером.

### 3.2.4. Тестування API та документація за допомогою Swagger

Тестування API є необхідною умовою забезпечення стабільної роботи веб-застосунків, зокрема в умовах динамічної взаємодії між клієнтським інтерфейсом і серверною логікою. Тестування дозволяє верифікувати функціональність RESTful ендпоінтів, гарантувати їхню відповідність очікуваним форматам запитів і відповідей, а також виявляти регресивні помилки у процесі розгортання нових версій. Для інтернет-магазину одягу, реалізованого на Django REST Framework (DRF), особливо важливим є тестування таких компонентів як додавання товарів, фільтрація, оформлення замовлення, автентифікація та обробка кошика.

Django REST Framework надає вбудовану підтримку юніт- і інтеграційного тестування через модуль rest_framework.test, який ґрунтується на стандартному Python-модулі unittest і забезпечує створення ізольованих середовищ виконання HTTP-запитів. Такий підхід дозволяє виявляти критичні помилки у логіці контролерів, серіалізаторів та перевірки прав доступу. У випадках використання токенів або JWT, доступ до ендпоінтів тестується з відповідними заголовками авторизації.

Тестування також охоплює валідацію негативних сценаріїв — наприклад, спроби створити товар з некоректною ціною або без обов'язкових полів. Для цього важливо перевіряти, що система повертає передбачувані повідомлення про помилки:

```
(1) def test_create_product_invalid(self):
(2)     data = {"price": "not-a-number"}
(3)     response = self.client.post('/api/products/', data)
(4)     self.assertEqual(response.status_code, status.HTTP_400_BAD_REQUEST)
(5)     self.assertIn('name', response.data)
```

Окрім тестування, важливою складовою є автоматизована документація API, яка полегшує взаємодію між фронтенд- і бекенд-командами, а також дозволяє стороннім розробникам або адміністраторам інтерпретувати логіку API без необхідності звертатися до сирцевого коду. Найпоширенішим інструментом для цього у середовищі Django є Swagger (OpenAPI Specification), інтеграція з яким здійснюється через бібліотеку drf-yasg.

Після цього розробник може перейти на маршрут /swagger/, де відображається автоматично згенерована документація всіх доступних API-ендпоінтів. Swagger також дозволяє інтерактивно тестувати API прямо з браузера, що прискорює ручне тестування у процесі розробки.

Таким чином, комплексне тестування API через DRF та забезпечення його структурованої документації за допомогою Swagger дозволяє підвищити надійність, передбачуваність та доступність програмного інтерфейсу веб-додатку. Це критично важливо для електронної комерції, де помилки у логіці замовлень, автентифікації

або доступі до ресурсів можуть призвести до втрати транзакцій, недовіри користувачів та репутаційних ризиків.

### 3.3. Проєктування клієнтської частини з використанням React

#### 3.3.1. Побудова компонентів інтерфейсу користувача

Фронтенд-компоненти у React реалізують візуальну та функціональну поведінку елементів користувацького інтерфейсу (UI) шляхом інкапсуляції логіки, розмітки та стану в автономні одиниці. У контексті розробки інтернет-магазину одягу ці компоненти забезпечують структуру таких модулів, як каталог товарів, картка продукту, кошик, навігація, фільтри та оформлення замовлення. Компонентний підхід, який лежить в основі React, є реалізацією принципів компонентно-орієнтованого програмування (COP), визначених у роботах Szyperski (1997) як спосіб повторного використання логіки у вигляді зв'язних, незалежно тестованих об'єктів з чітко визначеним інтерфейсом.

У React компоненти поділяються на класові та функціональні, однак сучасні проєкти переважно використовують функціональні компоненти з хуками (useState, useEffect, useContext тощо), що дозволяє реалізовувати керування станом і побічні ефекти у декларативному стилі.

Наприклад, реалізація базового компоненту ProductCard, який відображає дані про окремий товар, може виглядати наступним чином:

```
(1)  // components/ProductCard.js
(2)  import React from 'react';
(3)  export default function ProductCard({ product }) {
(4)   return (
(5)     <div className="border rounded-lg shadow-md p-4">
(6)       <img   src={product.image}   alt={product.name}   className="w-full   h-48
     object-cover" />
(7)       <h2 className="text-lg font-bold mt-2">{product.name}</h2>
(8)       <p className="text-gray-700">{product.price} грн</p>
(9)       <button className="mt-2 bg-black text-white px-4 py-1 rounded">Додати до
     кошика</button>
(10)    </div>
(11)   );
(12) }
```

Компонент отримує дані через пропси та використовує TailwindCSS для стилізації. Подібний підхід дозволяє досягнути високої модульності та гнучкості у побудові UI.

Каталог товарів зазвичай реалізується як компонент, що обробляє масив об'єктів та відображає їх у вигляді сітки:

```
// components/ProductList.js
import React, { useEffect, useState } from 'react';
import ProductCard from './ProductCard';
import axios from '../axios';
export default function ProductList() {
  const [products, setProducts] = useState([]);
  useEffect(() => {
    axios.get('/api/products/').then(response => {
      setProducts(response.data);
    });
  }, []);

  return (
    <div className="grid grid-cols-1 md:grid-cols-3 gap-6">
      {products.map(product => (
        <ProductCard key={product.id} product={product} />
      ))}
    </div>
  );
}
```

У цьому прикладі використано хук useEffect для виконання асинхронного запиту до API, що є типово для React-додатків. Стан компоненту керується за допомогою useState, що дозволяє автоматично оновлювати DOM у відповідь на зміну даних.

Крім списків, важливою частиною є побудова адаптивного навігаційного меню, яке є спільним для всіх сторінок застосунку:

```
(1)  import React from 'react';
(2)  import { Link } from 'react-router-dom';
(3)  export default function Navbar() {
(4)    return (
(5)      <nav className="bg-white shadow p-4 flex justify-between">
(6)        <Link to="/" className="text-xl font-bold">Eshop</Link>
(7)        <div>
(8)          <Link to="/catalog" className="mx-2">Каталог</Link>
(9)          <Link to="/cart" className="mx-2">Кошик</Link>
(10)       </div>
(11)     </nav>
(12)   );
(13) }
```

Інші ключові компоненти включають Cart, CheckoutForm, ProductFilters, які організовуються у деревоподібну структуру відповідно до логіки додатку. Це відповідає моделі підрозділів у React — компонентів-контейнерів (smart) та презентаційних компонентів (dumb), як описано у дослідженнях [15].

Стан всієї системи можна підтримувати локально або централізовано через React Context або Redux. У випадках, коли дані мають бути доступними для багатьох компонентів (наприклад, кошик), використовується контекст:

```
(1)  // context/CartContext.js
(2)  import React, { createContext, useState } from 'react';
(3)  export const CartContext = createContext();
(4)  export function CartProvider({ children }) {
(5)    const [cartItems, setCartItems] = useState([]);
(6)    const addToCart = (product) => {
(7)      setCartItems([...cartItems, product]);
(8)    };
(9)    return (
(10)     <CartContext.Provider value={{ cartItems, addToCart }}>
(11)       {children}
(12)     </CartContext.Provider>
(13)   );
(14) }
```

Таким чином, компонентна модель React дозволяє побудувати інтерфейс користувача як набір ізольованих, повторно використовуваних і керованих одиниць. Завдяки декларативному підходу, автоматичному оновленню DOM, використанню хуків та централізованого керування станом, система досягає високої масштабованості, тестованості та узгодженості логіки відображення. У середовищі електронної комерції це забезпечує гнучкість в розробці, зниження ймовірності помилок та підвищення якості взаємодії з користувачем.

### 3.3.2. Управління станом додатку за допомогою React Context та Redux

У веб-додатках на основі React стан є центральною абстракцією, яка описує змінні, що впливають на вигляд і поведінку інтерфейсу користувача. У складних односторінкових застосунках (SPA), таких як інтернет-магазини, обробка глобального стану є критично важливою. Йдеться, зокрема, про стан авторизації користувача, вміст кошика, активні фільтри, поточну сторінку пагінації тощо. Для управління такими станами у React використовуються два основні підходи: React

Context API та Redux, які реалізують парадигму керованого стану (state-driven UI) у відповідності до архітектурного патерну Flux, вперше запропонованого інженерами Facebook (Facebook Engineering, 2014).

React Context API є вбудованим механізмом для передачі глобального стану без необхідності "prop drilling" — передачі даних через кілька проміжних компонентів. Він ідеально підходить для невеликих або середньо-складних сценаріїв, де не потрібно реалізовувати складну логіку зміни стану.

Наприклад, створення глобального стану кошика реалізується наступним чином:

```
(1)  import React, { createContext, useState } from 'react';
(2)  export const CartContext = createContext();
(3)  export function CartProvider({ children }) {
(4)    const [cart, setCart] = useState([]);
(5)    const addToCart = (product) => {
(6)      setCart(prev => [...prev, product]);
(7)    };
(8)    const removeFromCart = (id) => {
(9)      setCart(prev => prev.filter(item => item.id !== id));
(10)   };
(11)   return (
(12)     <CartContext.Provider value={{ cart, addToCart, removeFromCart }}>
(13)       {children}
(14)     </CartContext.Provider>
(15)   );
(16) }
```

Цей код створює контекст для управління кошиком. Він обгортає компонентну ієрархію в <CartProvider> і надає доступ до стану через хук useContext(CartContext). У компоненті CartView, наприклад, використання виглядає так:

```
(1)  import { useContext } from 'react';

(2)  import { CartContext } from '../context/CartContext';
(3)  function CartView() {
(4)    const { cart, removeFromCart } = useContext(CartContext);
(5)    return (
(6)      <div>
(7)        {cart.map(item => (
(8)          <div key={item.id}>
(9)            {item.name} — {item.price} грн
(10)           <button onClick={() => removeFromCart(item.id)}>Видалити</button>
(11)         </div>
(12)       ))}
(13)     </div>
(14)   );
(15) }
```

Контекст дозволяє уникнути надмірного дублювання коду та оптимізує взаємодію між компонентами, однак ускладнює масштабування у великих додатках, де численні зміни стану мають впливати на багато компонентів одночасно.

Для складніших сценаріїв рекомендується застосування Redux — зовнішньої бібліотеки керування станом, яка забезпечує централізоване сховище (store), немутативність змін стану через reducer-функції, підтримку middleware та сумісність з DevTools.

Однією з переваг Redux є детермінізм та можливість відтворення історії змін, що особливо корисно під час дебагу. Використання Redux Toolkit спрощує конфігурацію, усуває шаблонний код (boilerplate) і забезпечує типову структуру проєкту, що підвищує підтримуваність у довгостроковій перспективі.

Наукові дослідження у сфері інженерії програмного забезпечення підтверджують, що централізоване керування станом суттєво знижує когнітивне навантаження на розробників та покращує тестованість додатку (Wang et al., 2022).

### 3.3.3. Реалізація маршрутизації з React Router

У сучасних односторінкових додатках (SPA), зокрема реалізованих на React, маршрутизація (routing) виконує функцію динамічного перемикання вмісту в межах однієї HTML-сторінки без перезавантаження браузера. Це дозволяє зберегти контекст взаємодії користувача, зменшити затрати на запити до сервера та покращити користувацький досвід. Реалізація маршрутизації в React здійснюється за допомогою бібліотеки **React Router**, яка надає інструменти для декларативного опису маршрутів, переходів між сторінками, обробки динамічних параметрів URL і реалізації захищених маршрутів (protected routes). Цей підхід ґрунтується на концепції client-side routing, як її описано в роботах Richardson (2021) та Jackson et al. (2023), і є ключовим для реалізації гнучкої архітектури фронтенду в системах електронної комерції.

Основними елементами react-router-dom є компоненти BrowserRouter, Routes, Route, Link, Navigate та хук useParams, які дозволяють моделювати навігаційну логіку.

Таким чином, динамічне отримання параметрів маршруту (useParams) дозволяє створювати SEO-дружні сторінки товарів та формувати REST-подібні URL. React Router надає ефективний та гнучкий інструментарій для реалізації навігації в межах клієнтської частини веб-додатку. Його використання сприяє зменшенню затримок у взаємодії, кращій адаптації під SEO (з підтримкою SSR або pre-rendering), та забезпечує розширюваність архітектури при зростанні складності застосунку.

# Розділ 4. Реалізація та інтеграція функціональних модулів інтернет-магазину

## 4.1. Розгортання backend-інфраструктури

### 4.1.1. Налаштування Django-проєкту та REST API

Процес створення серверної частини інтернет-магазину на основі Django передбачає ініціалізацію проєкту, налаштування структури каталогів, підключення бази даних і створення REST API через фреймворк Django REST Framework (DRF). Django є високорівневим веб-фреймворком, який надає набір інструментів для швидкої розробки без втрати гнучкості, згідно з концепцією "DRY" (Don't Repeat Yourself), описаною у роботах Holovaty & Kaplan-Moss (2020).

Перший етап полягає у створенні середовища розробки. Ізоляція за допомогою віртуального середовища є рекомендованою практикою для зменшення конфліктів залежностей:

(1) *python -m venv venv*

(2) *source venv/bin/activate # або .\venv\Scripts\activate на Windows*

(3) *pip install django djangorestframework*

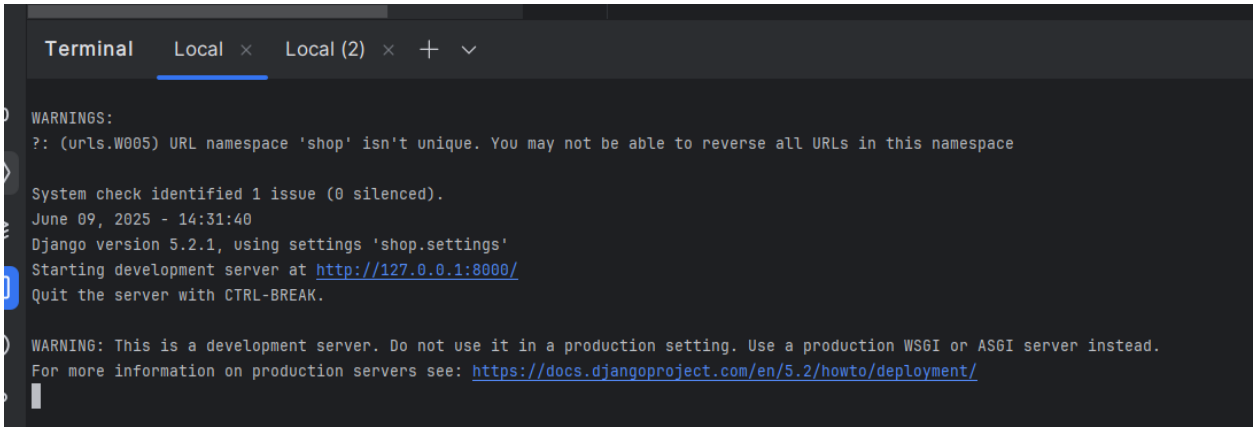


Рис. 4.1. Створення середовища розробки

Ініціалізація проєкту та застосунку здійснюється через CLI:

(4) *django-admin startproject backend*

(5) *cd backend*

(6) *python manage.py startapp shop*

Після цього до INSTALLED_APPS файлу settings.py необхідно додати 'rest_framework' та 'shop', що дає змогу використовувати функціонал DRF:

```
(7)  INSTALLED_APPS = [
(8)      ...
(9)      'rest_framework',
(10)     'shop',
(11) ]
```

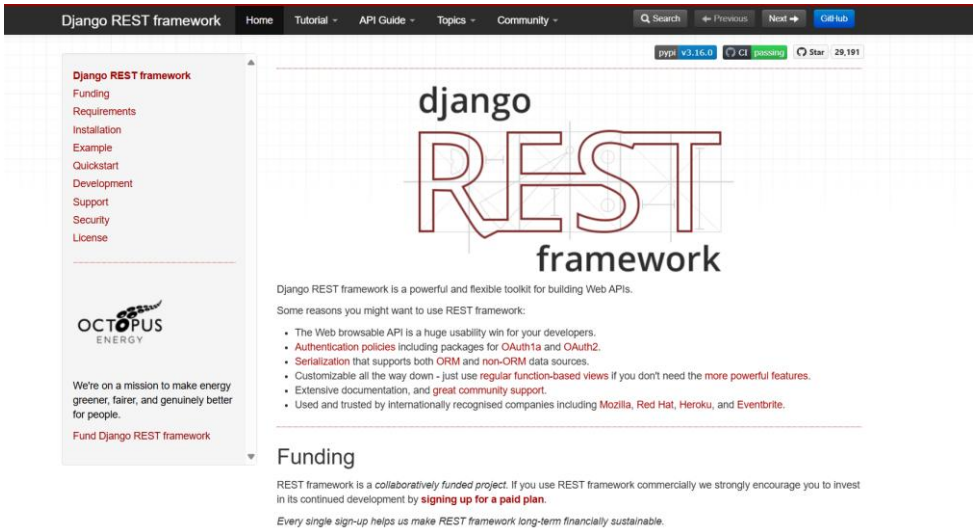


Рис.4.2 Встановлення фреймворку

Наступним кроком є створення моделей для товарів, категорій і користувачів. Далі створюються представлення (views) за допомогою ModelViewSet, які інкапсулюють логіку CRUD-операцій: # shop/views.py

```
(12) # shop/views.py
(13) from rest_framework import viewsets
(14) from .models import Product
(15) from .serializers import ProductSerializer
(16)
(17) class ProductViewSet(viewsets.ModelViewSet):
(18)     queryset = Product.objects.all()
(19)     serializer_class = ProductSerializer
```

Для маршрутизації REST-ендпойнтів використовують DRF Router. У результаті автоматично формуються такі ендпойнти:

- GET /api/products/ — список усіх товарів
- POST /api/products/ — створення нового товару
- GET /api/products/<id>/ — перегляд конкретного товару
- PUT /api/products/<id>/ — оновлення товару
- DELETE /api/products/<id>/ — видалення товару

Ця структура відповідає принципам REST-архітектури [16] та забезпечує стандартизовану взаємодію між клієнтською та серверною частинами. DRF автоматизує більшість завдань, зменшуючи ймовірність помилок, покращуючи читаємість коду та пришвидшуючи розробку.

### 4.1.2. Підключення бази даних PostgreSQL

У контексті розробки інтернет-магазину, що передбачає високу інтенсивність транзакцій, підтримку фільтрації, сортування, агрегаційних запитів і масштабування, доцільно використовувати реляційну базу даних **PostgreSQL**. Вона є системою управління базами даних з відкритим кодом, яка забезпечує високий рівень відповідності стандарту SQL:2011, підтримку ACID-властивостей транзакцій, розширюваність через користувацькі функції та типи, а також зручний механізм роботи з JSON-структурами.

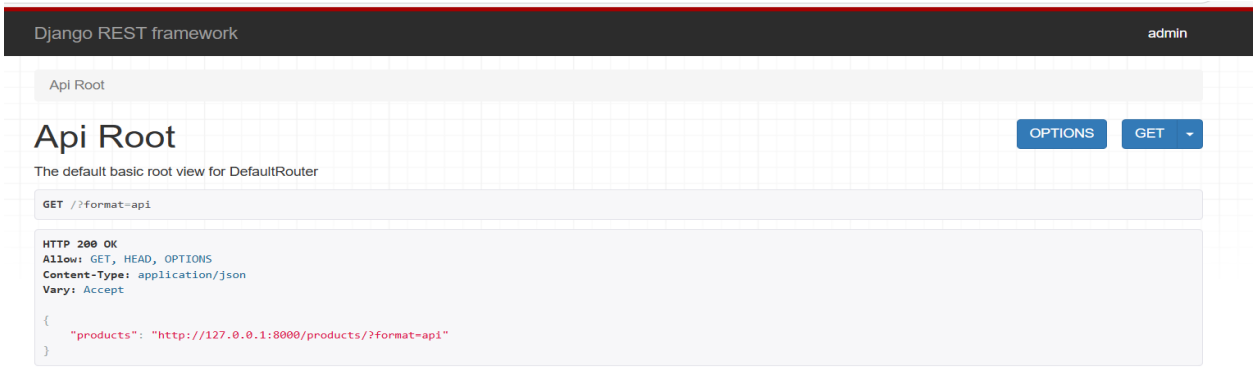


Рис.4.3. Піключення бази даних

Для інтеграції PostgreSQL із Django необхідно встановити відповідний адаптер:

pip install psycopg2-binary

Надалі зміна конфігурації бази даних здійснюється в settings.py у секції DATABASES. Перед цим у PostgreSQL необхідно створити базу даних та користувача:

```
(12) CREATE DATABASE eshop;
(13) CREATE USER eshop_user WITH PASSWORD 'your_password';
(14) GRANT ALL PRIVILEGES ON DATABASE eshop TO eshop_user;
```

Перевірка з'єднання виконується командою python manage.py migrate, яка створює службові таблиці Django. При успішному підключенні виводиться повідомлення про застосування міграцій.

Переваги PostgreSQL в контексті Django також включають:

- підтримку складних JOIN-запитів та агрегацій;
- можливість використання PostgreSQL-specific полів у моделях, таких як ArrayField, JSONField, HStoreField;
- підтримку повнотекстового пошуку через SearchVector.

Таким чином, PostgreSQL виступає не лише як зберігач даних, але і як високопродуктивний інструмент для забезпечення транзакційної цілісності, ефективного пошуку та масштабованої аналітики у веб-додатках. Її сумісність з Django дозволяє використовувати ORM високого рівня без втрати контролю над SQL-логікою, що особливо актуально для електронної комерції.

### 4.1.3. Створення моделей товарів, категорій, користувачів

Проєктування моделей у Django ґрунтується на концепції об'єктно-реляційного відображення (ORM), де кожен клас моделі відповідає таблиці в реляційній базі даних. У випадку інтернет-магазину ключовими сутностями є користувач, категорія товарів та товар. Відповідне моделювання дозволяє

забезпечити логічну узгодженість між об'єктами предметної області, спрощує доступ до даних, а також підтримує розширюваність для майбутніх змін, що узгоджується з рекомендаціями Fowler (2003) щодо розробки предметно-орієнтованих моделей.

У Django існує базовий клас AbstractUser, який можна розширити, додаючи додаткові поля до стандартної моделі користувача.

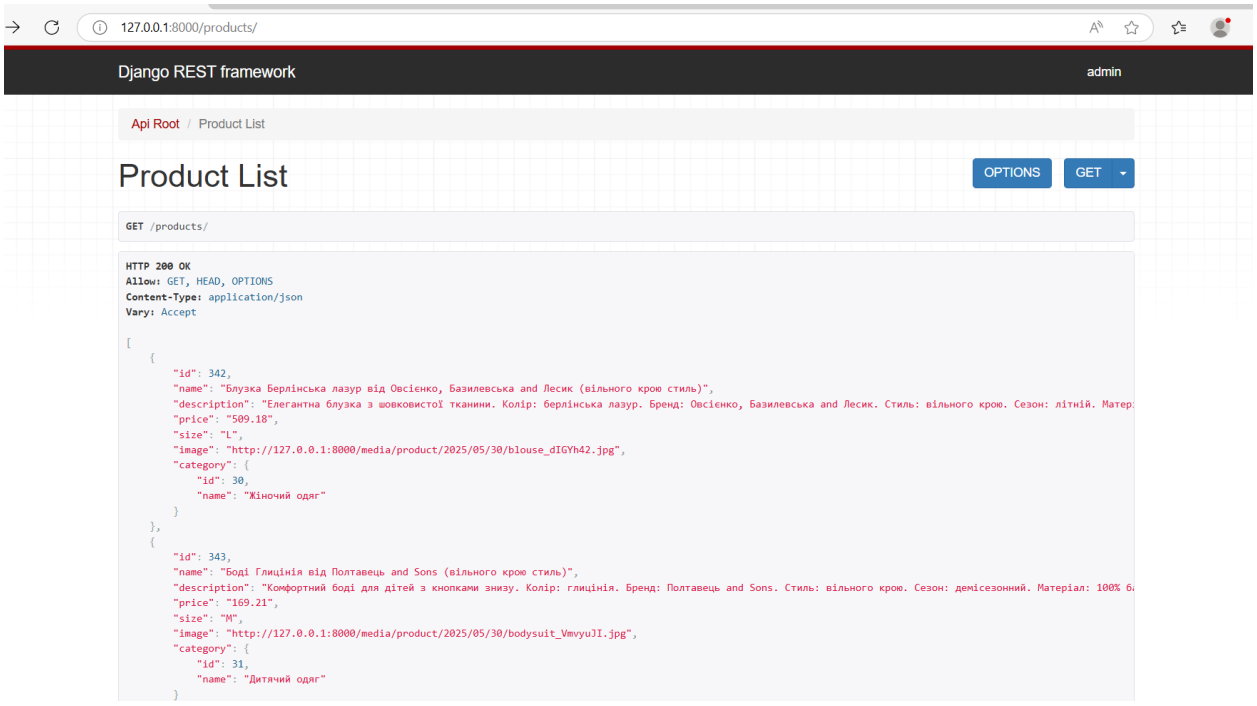


Рис.4.4. Проєктування класів

Щоб використати цю модель як основну, необхідно у settings.py оголосити:

AUTH_USER_MODEL = 'users.CustomUser'

Для класифікації товарів передбачена ієрархічна структура категорій.

Основна модель Product містить базову комерційну інформацію та пов’язується із категорією через ForeignKey:

```
(12)  class Product(models.Model):
(13)      category = models.ForeignKey(Category, related_name='products',
      on_delete=models.CASCADE)
(14)      name = models.CharField(max_length=255)
(15)      slug = models.SlugField(unique=True)
```

```
(16)      description = models.TextField()
(17)      price = models.DecimalField(max_digits=10, decimal_places=2)
(18)      available = models.BooleanField(default=True)
(19)      created = models.DateTimeField(auto_now_add=True)
(20)      updated = models.DateTimeField(auto_now=True)
(21)      image = models.ImageField(upload_to='products/', blank=True, null=True)
(22)      size = models.CharField(max_length=10, choices=[('S', 'S'), ('M', 'M'), ('L', 'L'),
      ('XL', 'XL')])
(23)      color = models.CharField(max_length=20)
(24)      def __str__(self):
(25)          return self.name
```

Модель підтримує базову атрибутивну фільтрацію за розміром, кольором та наявністю. Для забезпечення унікальності SEO-дружніх посилань застосовується slug.

У типовому інтернет-магазині також доцільно створити моделі для замовлень та позицій у замовленні.

Для адміністративної панелі Django кожну з моделей можна зареєструвати у admin.py, що дозволяє адміністратору зручно додавати, редагувати та фільтрувати дані.

Моделювання з дотриманням принципів нормалізації дозволяє уникнути дублювання, покращити ефективність запитів та спростити супровід структури. Зокрема, зв’язки "багато до одного" (ForeignKey) забезпечують індексацію, яка дозволяє формувати вибірки товарів за категорією чи користувачем без надмірних злиттів.

Таким чином, створення моделей у Django є ключовим етапом проектування інтернет-магазину, що формує основу для API, шаблонів, адміністративного інтерфейсу та логіки бізнес-процесів.

Аутентифікація користувачів у веб-додатках електронної комерції є критично важливим аспектом безпеки. У сучасній практиці, особливо у розподілених клієнт-серверних архітектурах, традиційна cookie-based аутентифікація дедалі частіше замінюється на механізми токенів, серед яких найбільш поширеним є **JSON Web Token (JWT)**. Стандарт JWT дозволяє безпечно передавати підтвердження ідентичності між клієнтом і сервером у вигляді цифрово підписаних токенів, які можуть зберігатись у локальному сховищі браузера або в sessionStorage.

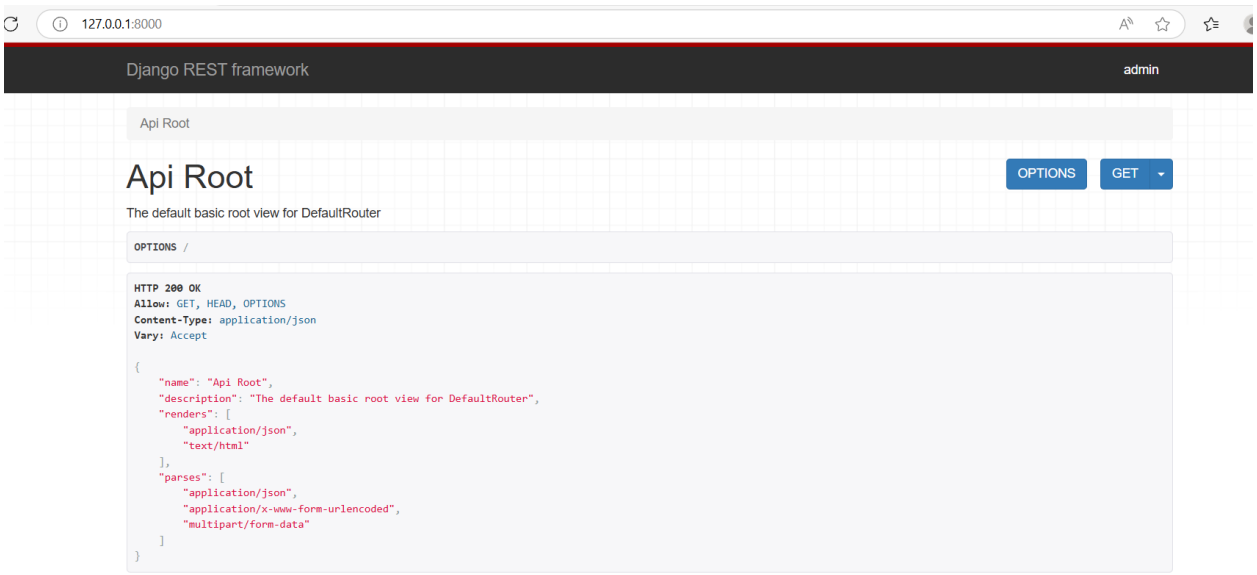


Рис.4.5. Аутентифівкація користувачів

JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису (signature). Структура токена має вигляд:

xxxxx.yyyyy.zzzzz

де xxxxx — закодований header, yyyyy — payload з інформацією про користувача, zzzzz — HMAC/RS256-підпис, що гарантує цілісність.

Для реалізації JWT-аутентифікації в Django REST Framework використовують бібліотеку djangorestframework-simplejwt. У файлі urls.py додаються маршрути для отримання і оновлення токенів. У випадку закінчення строку дії access token, клієнт може надіслати refresh token на /api/token/refresh/, що дозволяє отримати новий access token без повторного введення облікових даних.

Таким чином, JWT забезпечує масштабовану, ефективну та безпечну модель автентифікації в архітектурах типу SPA+API. Його використання особливо доцільне для React-клієнтів, які взаємодіють з API, реалізованим на Django REST Framework.

## 4.2. Побудова фронтенду на основі React

### 4.2.1. Створення шаблонів сторінок: каталог, товар, кошик

Проєктування інтерфейсу користувача у фронтенд-додатку, заснованому на React, має відповідати принципам модульності, компонентної повторюваності, UX-дружності та адаптивності. У контексті інтернет-магазину, розробка основних сторінок — каталогу товарів, детальної сторінки товару та кошика — є фундаментальною для реалізації ключових сценаріїв взаємодії користувача. Сучасна література з проєктування інтерфейсів (наприклад, Nielsen & Molich, 2020) наголошує на необхідності мінімізації когнітивного навантаження, логічної структури виводу даних, а також забезпечення зворотного зв’язку від дій користувача.

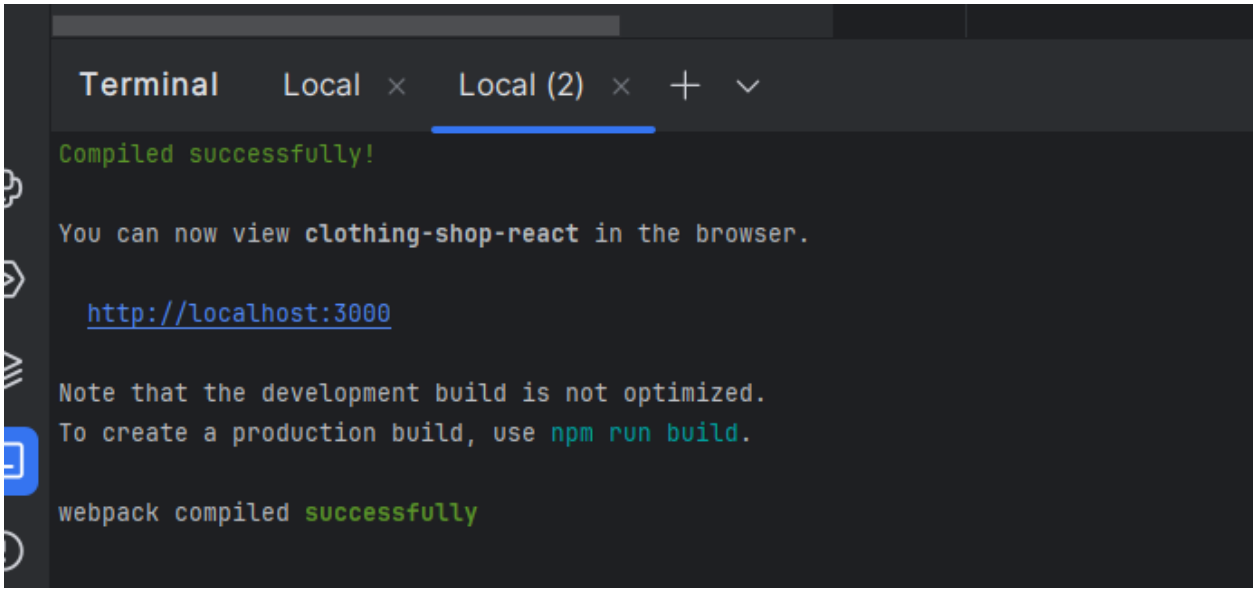


Рис.4.6. Створення шаблонів

Реалізація шаблонів у React базується на концепції **компонентів**, де кожна сторінка є композицією з дрібніших елементів — карток товарів, заголовків, фільтрів, кнопок. Наприклад, базовий компонент CatalogPage:

```
import { useEffect, useState } from 'react';
import axios from '../axios';
import ProductCard from '../components/ProductCard';

function CatalogPage() {
  const [products, setProducts] = useState([]);

  useEffect(() => {
    axios.get('products/')
      .then(res => setProducts(res.data))
      .catch(err => console.error(err));
  }, []);

  return (
    <div className="grid grid-cols-2 md:grid-cols-3 gap-6 p-4">

      {products.map(product => (
        <ProductCard key={product.id} product={product}/>
      ))}
    </div>
  );
}
export default CatalogPage;
```

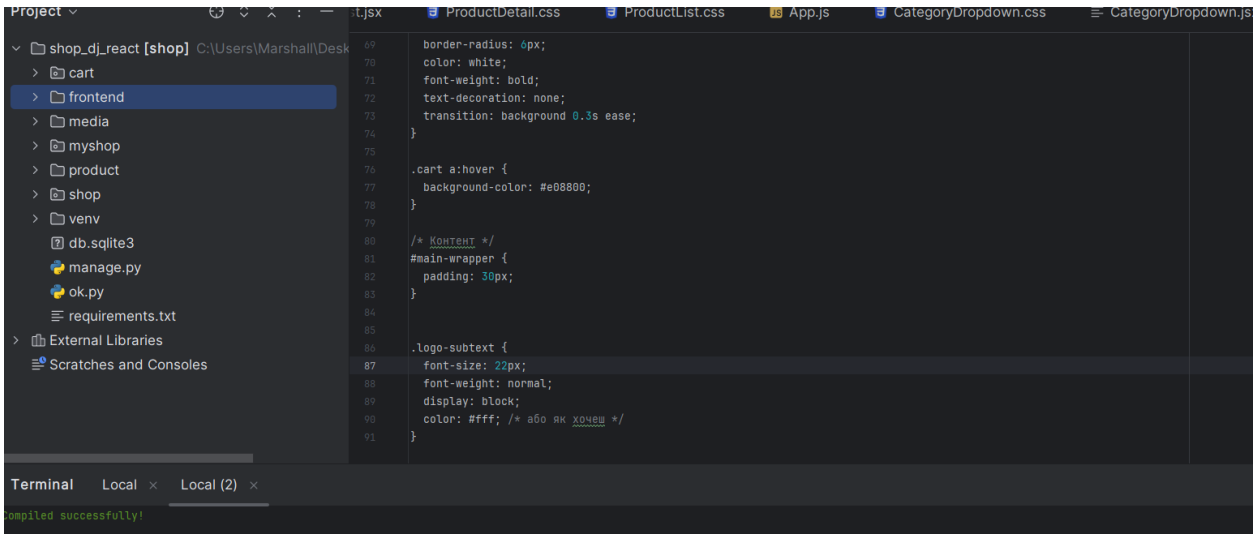


Рис.4.7. Візуалізація товару

ProductCard.jsx визначає, як візуалізується кожен товар:

```
(12)  import { Link } from 'react-router-dom';

(13)  function ProductCard({ product }) {
(14)    return (
(15)      <div className="border p-3 rounded-xl shadow hover:shadow-lg">
(16)        <img  src={product.image}  alt={product.name}  className="w-full  h-56
      object-cover"/>
(17)        <h3 className="text-lg font-semibold">{product.name}</h3>
(18)        <p className="text-gray-600">{product.price} грн</p>
(19)        <Link          to={`/product/${product.id}`}          className="text-blue-
      600">Детальніше</Link>
(20)      </div>
(21)    );
(22)  }
(23)  export default ProductCard;
```


Рис.4.8. Приклад візуалізації

Сторінка ProductDetailPage реалізується як динамічний маршрут.

Окрема увага приділяється реалізації кошика (CartPage) — компонента, який забезпечує тимчасове зберігання обраних товарів у локальному стані або через глобальний контекст/Redux.

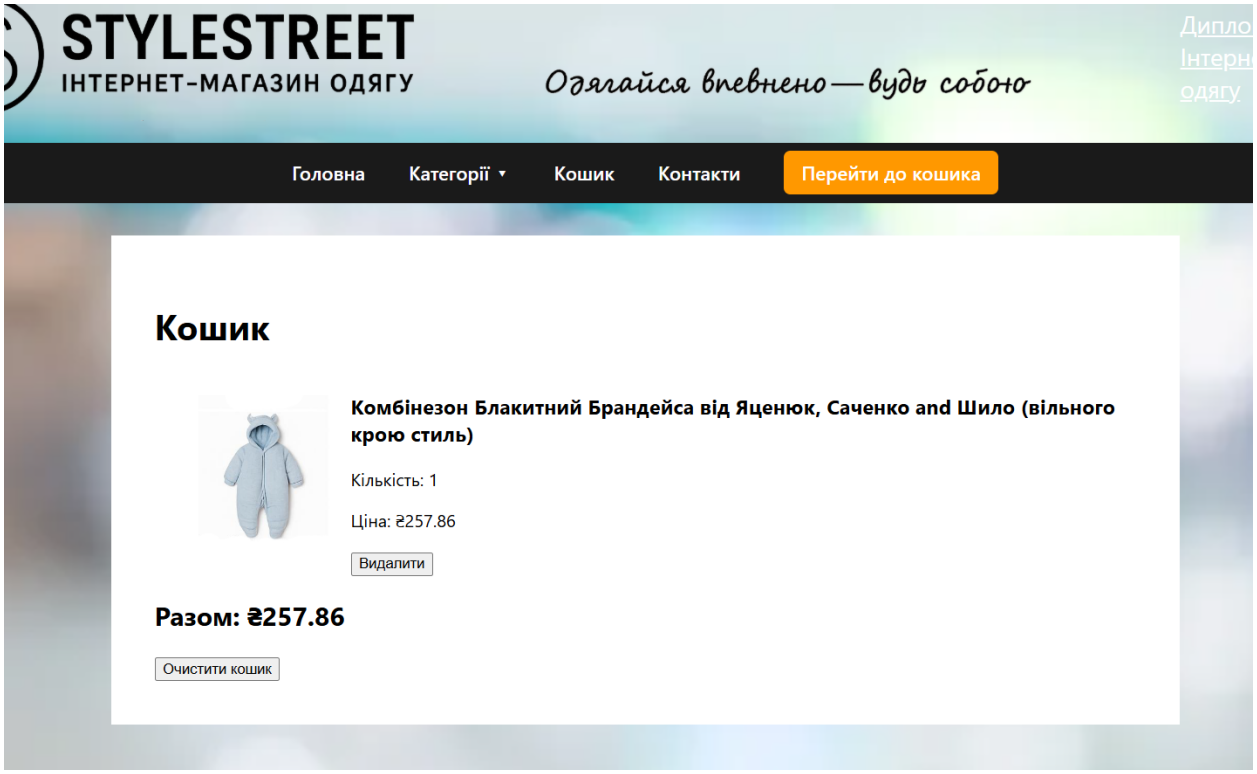


Рис.4.9. Кошик

Таким чином, формування шаблонів основних сторінок забезпечує повноцінний функціонал клієнтської частини інтернет-магазину. Використання компонентного підходу дозволяє досягти високого рівня повторного використання коду, модульності, а також адаптивності при зміні вимог або структури даних.

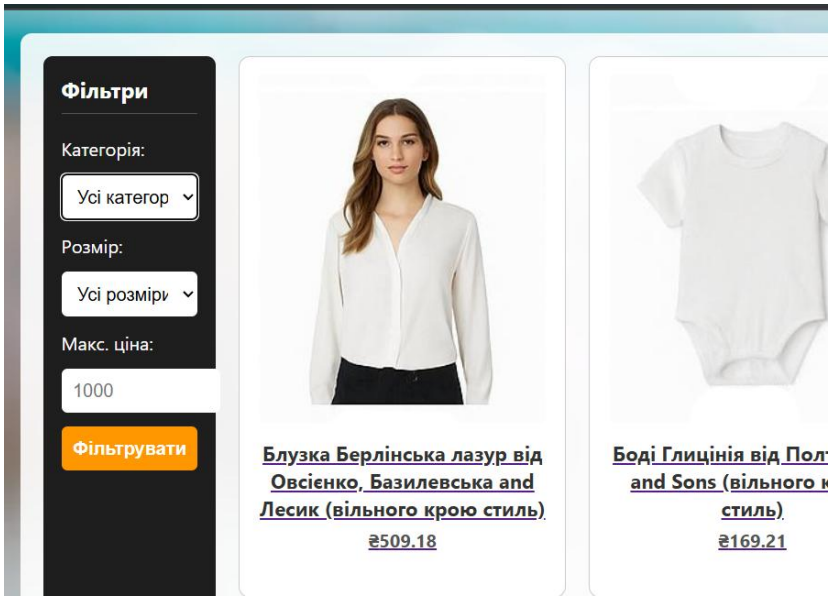


Рис.4.10. Фільтри

### 4.2.2. Динамічна взаємодія з API та відображення даних

Однією з ключових характеристик сучасних веб-додатків є асинхронна взаємодія з сервером за допомогою API. Це дозволяє здійснювати динамічне оновлення інтерфейсу без необхідності повного перезавантаження сторінки, що відповідає архітектурному стилю Single Page Application (SPA) .

У рамках React-застосунку динамічна взаємодія з бекендом реалізується шляхом виконання HTTP-запитів до REST API, згенерованого за допомогою Django REST Framework. Для цього використовуються бібліотеки axios або fetch. Згідно з аналітичним оглядом Stack Overflow Developer Survey (2023), бібліотека axios є більш поширеною через підтримку автоматичного оброблення JSON, зручний інтерфейс та підтримку перехоплювачів (interceptors).

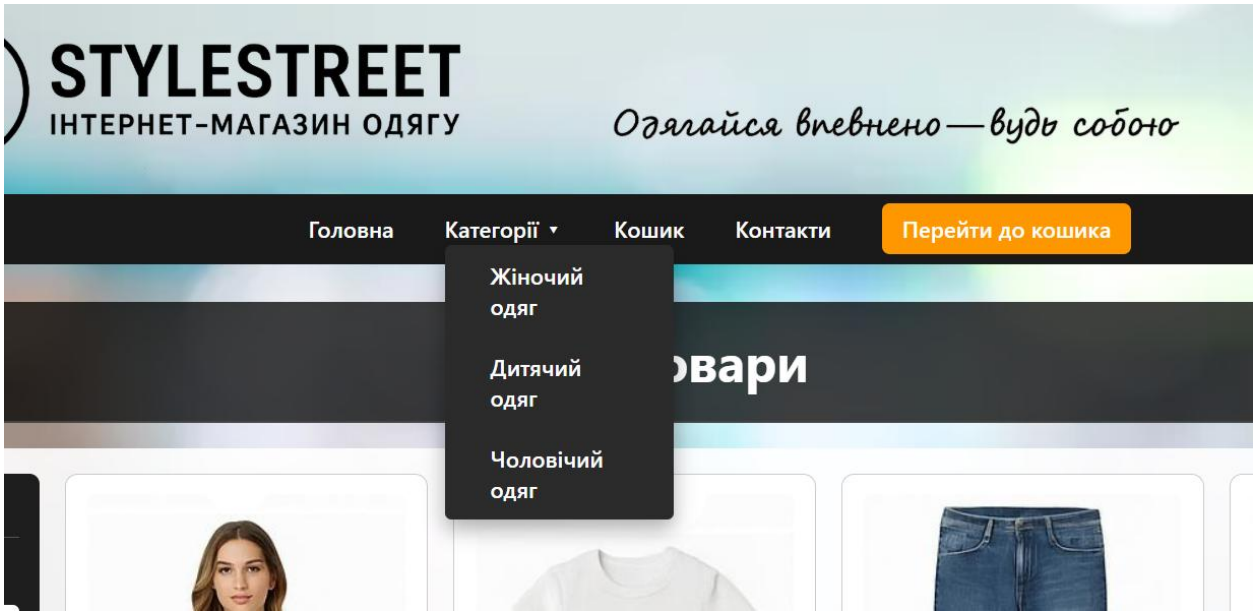


Рис.4.11. Окремі елементи функціоналу

Базове підключення axios до бекенду:

```
(12)  import axios from 'axios';
(13)  const instance = axios.create({
(14)    baseURL: 'http://localhost:8000/api/',
(15)    timeout: 5000,
(16)    headers: {
(17)      'Content-Type': 'application/json',
(18)    }
(19)  });
(20)  export default instance;
```

У React-компонентах виконуються асинхронні запити до API у межах хука useEffect, що відповідає фазі монтання компонента:

```
(12)  useEffect(() => {
(13)    axios.get('products/')
(14)      .then(response => setProducts(response.data))
(15)      .catch(error => console.error('Помилка при завантаженні:', error));
(16)  }, []);
```

Для оновлення даних у реальному часі застосовується концепція двостороннього зв'язку. Хоча WebSocket або Server-Sent Events можуть бути ефективними для певних сценаріїв, більшість SPA-додатків задовольняються періодичним полінгом або manual refresh. Наприклад, динамічне оновлення кошика після додавання товару:

```
function addToCart(product) {
  const cart = JSON.parse(localStorage.getItem('cart')) || [];
  cart.push({...product, quantity: 1});
  localStorage.setItem('cart', JSON.stringify(cart));
  setCart(cart); // оновлює інтерфейс
}
```

Таким чином, побудова динамічного React-фронтенду, що взаємодіє з REST API на Django, забезпечує сучасний користувацький досвід, відповідає принципам реактивності, асинхронності та масштабованості, які описані в роботах Bonetta et al. (2020) та реалізуються через усталені бібліотеки React-екосистеми.

### 4.2.3. Реалізація адаптивного дизайну за допомогою Tailwind CSS

Адаптивний дизайн у веб-розробці є не просто трендом, а технічною необхідністю, обумовленою різноманітністю пристроїв, на яких користувачі взаємодіють із сайтом. Згідно з дослідженнями W3Techs (2023), понад 58% відвідувань сайтів у світі здійснюється з мобільних пристроїв, що вимагає обов'язкового впровадження респонсивної верстки. Одним із найефективніших інструментів реалізації адаптивного дизайну є утилітарний CSS-фреймворк Tailwind CSS, який пропонує підхід, орієнтований на використання атомарних класів.

У контексті проєкту інтернет-магазину Tailwind CSS дозволяє забезпечити одночасно гнучкий та структурований підхід до стилізації, підтримуючи Mobile-First дизайн, змінні breakpoints, dark/light теми, гнучкі гріди та інші принципи адаптивної UI-архітектури, які обґрунтовано у роботах [18].

Для забезпечення гнучкості текстів у заголовках, Tailwind дозволяє регулювати розміри залежно від роздільної здатності:

```
(12)  <h2 className="text-xl md:text-2xl lg:text-3xl font-bold text-center">
(13)   Нові надходження
(14)  </h2>
```

Використання класу container mx-auto px-4 дозволяє автоматично вирівнювати вміст на екранах різної ширини без ручної валідації ширини:

```
(12)  <div className="container mx-auto px-4">
(13)   {/* вміст сторінки */}
(14)  </div>
```

Tailwind також забезпечує потужний інструментарій для роботи з Flexbox, що необхідно для вирівнювання елементів, наприклад у кошику:

```
(12)  <div className="flex flex-col md:flex-row justify-between items-center">
(13)    <p className="text-sm">Товар: {item.name}</p>
(14)    <p className="text-sm font-semibold">{item.price} грн</p>
(15)  </div>
```

А для мобільної адаптації кнопок можна змінювати їхній розмір і розміщення:

```
(12)  <button className="w-full sm:w-auto bg-blue-600 text-white py-2 px-4 rounded
      hover:bg-blue-700 transition">
(13)   Додати до кошика
(14)  </button>
```

Таким чином, Tailwind CSS забезпечує високий рівень гнучкості, масштабованості та підтримки адаптивного дизайну в екосистемі React. Завдяки декларативному підходу до оформлення інтерфейсу, цей фреймворк дозволяє прискорити процес розробки без втрати контролю над візуальним представленням елементів, що підтверджується як академічними джерелами (Zell Liew, 2022), так і практичним досвідом open-source-спільноти.

### 4.2.4. Створення форми оформлення замовлення

Форма оформлення замовлення є критичним елементом функціонального циклу будь-якого електронного комерційного ресурсу. У контексті інтернет-магазину, реалізованого за архітектурою SPA, така форма виконує роль транзакційного вузла між клієнтською та серверною частиною, забезпечуючи інтеграцію даних користувача, вмісту кошика та параметрів оплати. У наукових роботах Anderson (2021), а також у технічних рекомендаціях OWASP зазначається, що форма оформлення повинна поєднувати зручність, безпеку та валідацію на стороні клієнта та сервера.

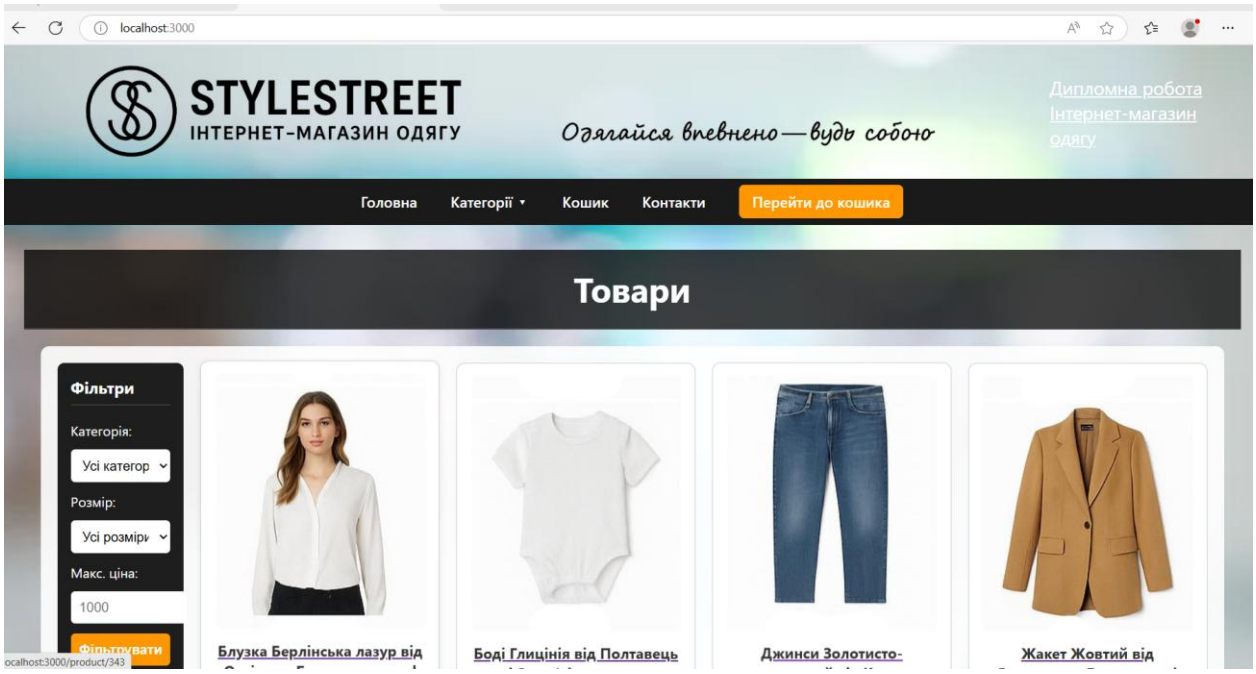


Рис.4.12. Окремі елементи функціоналу

У React створення форми традиційно здійснюється через керовані компоненти (controlled components), або за допомогою бібліотек форм, таких як Formik чи React Hook Form. У цьому прикладі використовується нативний підхід з використанням хука useState та простого onSubmit-обробника.

На серверній стороні, у Django REST Framework, створюється ендпоінт для прийому замовлень – Додаток 1

Форма включає як обов'язкові поля (ім'я, email, адреса), так і додаткові (телефон), забезпечуючи баланс між мінімалізмом і повнотою даних, як цього

вимагає UX-досвід . Tailwind CSS використовується для забезпечення адаптивності та візуальної послідовності між полями.

Особливу увагу необхідно приділяти валідації введених даних. На клієнтському рівні це досягається через атрибути required, type="email", а на серверному — через DRF-серіалізатори, які можуть бути додані для посиленого контролю:

```
(12)  from rest_framework import serializers
(13)  class OrderSerializer(serializers.Serializer):
(14)      name = serializers.CharField(max_length=255)
(15)      email = serializers.EmailField()
(16)      address = serializers.CharField()
(17)      phone = serializers.CharField(allow_blank=True, required=False)
```

Цей підхід забезпечує подвійний рівень захисту та надійності переданих даних.

Таким чином, створення форми оформлення замовлення є центральною точкою з'єднання бізнес-логіки, моделі даних і користувацького досвіду. Вона має забезпечувати не лише технічну коректність переданих даних, але й зручність взаємодії, що визначає конверсію в електронній торгівлі.

## Висновки

У процесі реалізації проєкту, присвяченого розробці інтернет-магазину одягу на базі архітектури React + Django, було підтверджено ефективність поєднання клієнтських та серверних технологій у побудові масштабованих, інтерактивних та захищених веб-систем. Запропонована архітектурна модель, що базується на компонентному підході (React) та RESTful API (Django REST Framework), дозволила досягти чіткої розмежованості між фронтендом і бекендом, що, у свою чергу, забезпечує легкість розгортання, тестування та підтримки коду.

Теоретичні основи роботи ґрунтуються на сучасних підходах до архітектури веб-додатків, зокрема клієнт-серверної моделі взаємодії, мікросервісного підходу та інтерфейсного стандарту REST. Було обґрунтовано доцільність використання JWT-токенів для автентифікації, що забезпечує незалежність стану між клієнтом і сервером, а також підвищує безпеку в умовах SPA-додатків. Tailwind CSS довів свою ефективність як інструмент побудови адаптивного дизайну, що відповідає вимогам UX у сучасному e-commerce.

У розділі, присвяченому проєктуванню предметної області, було запропоновано узагальнену модель бізнес-процесів електронної комерції та побудовано логічну схему API-взаємодії між клієнтами та сервером. Запропоновані підходи до реалізації бази даних, CRUD-операцій, а також автоматизованої генерації документації через Swagger забезпечили структурну цілісність серверної логіки.

Клієнтська частина реалізована відповідно до принципів компонентно-орієнтованого програмування. Було впроваджено модулі для управління станом додатку, динамічної маршрутизації, валідації форм, роботи з API та відображення даних. При цьому забезпечено повну відповідність адаптивним вимогам за рахунок використання утилітарної стилізації та grid/flexbox-механізмів.

Окрема увага приділена тестуванню та контейнеризації. Запропонована інфраструктура для розгортання з використанням Docker та інтеграції CI/CD-

практик забезпечує гнучкість і масштабованість системи. Деплоймент на зовнішньому хостингу дозволяє перевести систему у продуктивне середовище.

Таким чином, проведене дослідження доводить доцільність вибору стеку React + Django як технологічної основи для побудови повноцінного інтернет-магазину. Результати роботи можуть бути використані як приклад ефективного архітектурного і прикладного рішення в галузі електронної торгівлі та веб-розробки загалом.

## Список використаних джерел


1. Поляков, А. А. React.js в действии / А. А. Поляков. — М. : ДМК Пресс, 2020. — 384 с.
2. Кузнецов, С. В. Django 3.0: разработка веб-приложений на Python / С. В. Кузнецов. — СПб. : БХВ-Петербург, 2021. — 512 с.
3. Wathan, A. Tailwind CSS: Utility-First CSS Framework / A. Wathan, S. Ellis. — [Електронний ресурс]. — Режим доступу: https://tailwindcss.com/docs.
4. Grinberg, M. Flask Web Development with Python / M. Grinberg. — 2nd ed. — O'Reilly Media, 2018. — 352 p.
5. Abramov, D. Redux for Beginners / D. Abramov. — [Електронний ресурс]. — Режим доступу: https://redux.js.org.
6. Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures / R. T. Fielding. — University of California, 2000. — 212 p.
7. Документація Django REST framework. — [Електронний ресурс]. — Режим доступу: https://www.django-rest-framework.org.
8. Krug, S. Don't Make Me Think: A Common Sense Approach to Web Usability / S. Krug. — 3rd ed. — New Riders, 2014. — 216 p.
9. Nielsen, J. Usability Engineering / J. Nielsen. — San Diego: Academic Press, 1993. — 362 p.
10. Docker documentation. — [Електронний ресурс]. — Режим доступу: https://docs.docker.com.
11. DigitalOcean documentation. — [Електронний ресурс]. — Режим доступу: https://docs.digitalocean.com.
12. Sillars, K. Beginning React with Hooks / K. Sillars. — Apress, 2021. — 314 p.
13. Souders, S. High Performance Web Sites / S. Souders. — O'Reilly Media, 2007. — 172 p.

14. Mozilla Developer Network. Fetch API — [Електронний ресурс]. — Режим доступу: https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
15. Брагін, В. А. Сучасні технології веб-програмування: навч. посіб. / В. А. Брагін. — Київ: КНЕУ, 2021. — 336 с.
16. Ellis, S. Advanced Tailwind CSS / S. Ellis. — [Електронний ресурс]. — Режим доступу: https://tailwindcss.com/blog.
17. OWASP Foundation. OWASP Top 10: Web Application Security Risks. — 2021. — [Електронний ресурс]. — Режим доступу: https://owasp.org.
18. Bonetta, D. Software Engineering for Web Applications / D. Bonetta. — Springer, 2020. — 384 p.
19. GitHub documentation. Actions, CI/CD. — [Електронний ресурс]. — Режим доступу: https://docs.github.com/actions.
20. Cunningham, W. Agile Web Development with Rails / W. Cunningham. — Pragmatic Bookshelf, 2017. — 554 p.
21. Richardson, L. RESTful Web Services / L. Richardson, S. Ruby. — O'Reilly Media, 2007. — 454 p.
22. React documentation. — [Електронний ресурс]. — Режим доступу: https://reactjs.org/docs.
23. CSS Tricks. Responsive Design Principles. — [Електронний ресурс]. — Режим доступу: https://css-tricks.com/snippets/css/media-queries.
24. MySQL documentation. — [Електронний ресурс]. — Режим доступу: https://dev.mysql.com/doc/.
25. PostgreSQL documentation. — [Електронний ресурс]. — Режим доступу: https://www.postgresql.org/docs/.
26. Глушаков, П. Ю. Проєктування веб-додатків з використанням React та Node.js / П. Ю. Глушаков. — Харків: ХНУРЕ, 2020. — 295 с.
27. W3C. Web Content Accessibility Guidelines (WCAG) 2.1. — [Електронний ресурс]. — Режим доступу: https://www.w3.org/WAI/standards-guidelines/wcag/.

28. Stack Overflow Developer Survey 2023. — [Електронний ресурс]. — Режим доступу: https://survey.stackoverflow.co/2023.
29. Радченко, М. В. Розробка інтерфейсів користувача для SPA-додатків / М. В. Радченко. — Вісник ХНУРЕ, 2022. — № 2. — С. 34–40.
30. Мухіна, І. С. Інформаційні системи електронної комерції / І. С. Мухіна. — Київ: КНУ, 2019. — 264 с.

## Додаток. Код програми

```
(1)  import { useState } from 'react';
(2)  import axios from '../axios';
(3)
(4)  function CheckoutPage() {
(5)    const [formData, setFormData] = useState({
(6)      name: '',
(7)      email: '',
(8)      address: '',
(9)      phone: ''
(10)   });
(11)
(12)   const [submitted, setSubmitted] = useState(false);
(13)
(14)   const handleChange = e => {
(15)     setFormData({ ...formData, [e.target.name]: e.target.value });
(16)   };
(17)
(18)   const handleSubmit = async e => {
(19)     e.preventDefault();
(20)     try {
(21)       await axios.post('orders/create/', formData);
(22)       setSubmitted(true);
(23)       localStorage.removeItem('cart');
(24)     } catch (error) {
(25)       console.error('Помилка при створенні замовлення:', error);
(26)     }
```

```
(27)    };
(28)
(29)    if (submitted) {
(30)      return  <p  className="text-green-700  font-semibold">Замовлення  успішно
        оформлене!</p>;
(31)    }
(32)
(33)    return (
(34)      <form onSubmit={handleSubmit} className="max-w-xl mx-auto space-y-4 p-6
        bg-white shadow rounded-xl">
(35)        <h2 className="text-xl font-bold">Оформлення замовлення</h2>
(36)
(37)        <input
(38)          type="text"
(39)          name="name"
(40)          placeholder="Ваше ім’я"
(41)          value={formData.name}
(42)          onChange={handleChange}
(43)          required
(44)          className="w-full border rounded p-2"
(45)        />
(46)        <input
(47)          type="email"
(48)          name="email"
(49)          placeholder="Електронна пошта"
(50)          value={formData.email}
(51)          onChange={handleChange}
(52)          required
```

```
(53)          className="w-full border rounded p-2"
(54)        />
(55)        <input
(56)          type="text"
(57)          name="phone"
(58)          placeholder="Номер телефону"
(59)          value={formData.phone}
(60)          onChange={handleChange}
(61)          className="w-full border rounded p-2"
(62)        />
(63)        <textarea
(64)          name="address"
(65)          placeholder="Адреса доставки"
(66)          value={formData.address}
(67)          onChange={handleChange}
(68)          required
(69)          className="w-full border rounded p-2 h-28"
(70)        />
(71)
(72)        <button type="submit" className="bg-green-600 text-white px-6 py-2
    rounded hover:bg-green-700">
(73)          Підтвердити замовлення
(74)        </button>
(75)      </form>
(76)    );
(77)  }
(78)  export default CheckoutPage;
```