Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

Допущено до захисту
Завідувач кафедри комп'ютерних наук
доктор технічних наук, професор
_____ Андрій БОНДАРЧУК
« ___ » _____ 202_ р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»
Спеціальність 122 Комп'ютерні науки
Освітня програма 122.00.01 Інформатика

Тема: МОДЕЛЮВАННЯ СЕРВІСНО-ОРІЄНТОВАНОЇ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ЕЛЕКТРОННОЇ КОМЕРЦІЇ

Виконав
студент групи ІНб-1-22-4.0д
Сірий Назарій Сергійович

_____

Науковий керівник
доктор філософії,
Турукало А.В.
_____

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп'ютерних наук
доктор технічних наук, професор
_____ Андрій БОНДАРЧУК
« ___ » _____ 202_ р.

ЗАВДАННЯ НА КВАЛІФІКАЦІНУ РОБОТУ БАКАЛАВРА

«Моделювання сервісно-орієнтованої архітектури програмного забезпечення для електронної комерції»

Виконавець – студент групи ІНб-1-22-4.0д,
спеціальності 122 Комп'ютерні наук,
освітньої програми 122.00.01 Інформатика
Сірий Назарій Сергійович

1. Вихідні дані: законодавство та нормативно-правові акти України у сфері освіти й інформаційних технологій, наукові праці та публікації з інформатики, веброзробки й архітектури інформаційних систем.
2. Основними завданнями бакалаврської роботи є: аналіз архітектурних підходів (монолітна, SOA, мікросервісна архітектури); вибір і обґрунтування технологічного стеку (React 18, Node.js/Express.js, PostgreSQL, JWT); проектування структури системи та REST API; розробка бекенд-сервісів і клієнтської SPA-частини; проведення тестування та оформлення пояснювальної записки.
3. Практичне значення роботи полягає у створенні SOA-системи інтернет-магазину, що забезпечує гнучкість і масштабованість вебзастосунку.
4. Пояснювальна записка: обсяг – до 75 стор. формату А4.
5. Графічні матеріали: презентація.
6. Додатки: програмний код SOA-системи інтернет-магазину.
7. Строк подання роботи на кафедру: «_» червня 2026 р.

Науковий керівник	Виконавець:
Доктор філософії	
_____ Турукало А.В.	_____ Сірий Н.С.
_____ Дата	_____ Дата

Календарний план

№	Назва етапу дипломної роботи	Строк виконання етапу роботи	Примітка
1	Затвердження теми кваліфікаційної роботи бакалавра	31.10.2025	Виконано
2	Аналіз проблеми, вивчення аналогів, формування цілей та завдань	11.11.2025–11.01.2026	Виконано
3	Аналіз предметної області, архітектурних підходів та існуючих рішень	26.01.2026–15.03.2026	Виконано
4	Визначення функціональних та нефункціональних вимог	16.03.2026–31.03.2026	Виконано
5	Проєктування архітектури, бази даних та реалізація SOA-системи	01.04.2026–15.04.2026	Виконано
6	Тестування функціональних модулів та виправлення помилок	16.04.2026–30.04.2026	Виконано
7	Оформлення документації та підготовка до експлуатації	01.05.2026–15.05.2026	Виконано
8	Підготовка пояснювальної записки та додатків	25.05.2026–12.06.2026	Виконано
9	Офіційний захист на засіданні екзаменаційної комісії	18.06.2026	

«Затверджую»

Науковий керівник

доктор філософії,

Турукало А.В._____

Індивідуальний план склав

Сірий Н.С. _____

РЕФЕРАТ

Кваліфікаційна робота: 59 с., 10 рис., 15 табл., 36 посилань, 1 додаток.

Актуальність. Стрімкий розвиток цифрових технологій та масова інтернетизація суттєво трансформували сферу торгівлі. Електронна комерція стала однією з провідних моделей роздрібного продажу, а сучасні інтернет-магазини потребують високої продуктивності, масштабованості та гнучкості. Однією з ключових проблем розробки таких систем є вибір архітектурного підходу. Традиційна монолітна архітектура має низку обмежень, зокрема складність масштабування та залежність усіх компонентів системи між собою. Альтернативою є сервісно-орієнтована архітектура (SOA), яка забезпечує поділ системи на незалежні сервіси з чітко визначеними інтерфейсами взаємодії.

Предметом дослідження є методи, засоби та патерни проектування SOA-системи інтернет-магазину на основі JavaScript-стеку (React 18, Node.js, Express.js, PostgreSQL).

Об'єктом дослідження виступає процес проектування та розробки інформаційних систем електронної комерції із застосуванням сервісно-орієнтованої архітектури.

Метою роботи є проектування та реалізація інформаційної системи інтернет-магазину на засадах SOA з використанням технологій React, Node.js/Express.js та PostgreSQL. У роботі проведено аналіз предметної галузі електронної комерції, виконано порівняння монолітної, SOA та мікросервісної архітектур, обґрунтовано вибір SOA для системи середнього масштабу, здійснено огляд сучасних вебтехнологій та засобів розробки.

Завдання:

1. Провести аналіз предметної галузі електронної комерції, виконати порівняльний огляд архітектурних підходів (монолітна, SOA, мікросервісна архітектура) та обґрунтувати вибір технологічного стеку для розробки системи.

2. Спроектувати та реалізувати SOA-систему інтернет-магазину: розробити загальну архітектуру, структуру бази даних і REST API, клієнтську SPA-частину на React 18 та серверні сервіси на Node.js/Express.js, а також провести функціональне й навантажувальне тестування системи

Основні результати. Спроектовано архітектуру системи, структуру бази даних і REST API, а також реалізовано клієнтську частину застосунку на React та чотири серверні сервіси на Node.js/Express.js із JWT-автентифікацією та PostgreSQL через Sequelize ORM. Проведено функціональне й навантажувальне тестування системи та проаналізовано результати її роботи.

Практичне значення дослідження. Результати роботи можуть бути застосовані при проектуванні та розробці масштабованих вебзастосунків електронної комерції із застосуванням сервісно-орієнтованої архітектури. Розроблена система інтернет-магазину є готовим практичним прикладом реалізації SOA на основі JavaScript-стеку та може використовуватися як навчальний або базовий проєкт для подальшого розширення функціональності та масштабування.

Ключові слова: електронна комерція, сервісно-орієнтована архітектура, SOA, React, Node.js, Express.js, PostgreSQL, REST API, JWT, SPA, інтернет-магазин, вебзастосунок.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ ТА ТЕРМІНІВ

CRUD	Create, Read, Update, Delete – базові операції над даними: створення, читання, оновлення, видалення
CSS	Cascading Style Sheets – каскадні таблиці стилів для оформлення вебсторінок
CORS	Cross-Origin Resource Sharing – механізм спільного використання ресурсів між різними джерелами
DOM	Document Object Model – програмний інтерфейс для HTML- та XML-документів
ESB	Enterprise Service Bus – корпоративна сервісна шина для маршрутизації повідомлень
HTML	HyperText Markup Language – мова розмітки гіпертексту
IC	Інформаційна система
JSONB	Binary JSON – бінарний JSON, розширений тип даних PostgreSQL з підтримкою індексування
JWT	JSON Web Token – компактний токен для передавання тверджень між сторонами
MVC	Model-View-Controller – архітектурний патерн розподілу відповідальності
ORM	Object-Relational Mapping – об'єктно-реляційне відображення, техніка перетворення даних між БД і ОО-мовами
REST	Representational State Transfer – архітектурний стиль побудови веб-API
SOA	Service-Oriented Architecture – сервісно-орієнтована архітектура
SPA	Single Page Application – односторінковий вебсайт
СУБД	Система управління базами даних

ЗМІСТ

ВСТУП

Стрімкий розвиток цифрових технологій та масова інтернетизація кардинально трансформували способи здійснення торгівлі. Електронна комерція перетворилася з нішевого каналу збуту на домінуючу модель роздрібного продажу: за даними Statista, обсяг світового ринку e-commerce у 2023 році перевищив 5,8 трлн доларів США, а до 2027 року прогнозується його зростання до 7,9 трлн доларів [2]. Україна повторює цю тенденцію – попри виклики воєнного часу вітчизняний ринок онлайн-торгівлі залишається стійким, а потреба в надійних, масштабованих програмних рішеннях для інтернет-магазинів не зменшується.

Ключовою технічною проблемою розробки сучасних комерційних платформ є вибір архітектурного підходу. Традиційна монолітна архітектура, де весь функціонал системи розгортається як єдиний неподільний артефакт, неспроможна задовольнити сучасні вимоги: будь-яка зміна вимагає повного перерозгортання, масштабування можливе лише цілком, а збій в одному модулі може паралізувати всю систему [11]. Альтернативою є сервісно-орієнтована архітектура (SOA), що декомпозує систему на автономні сервіси з чітко визначеними інтерфейсами взаємодії. SOA дозволяє незалежно розробляти, масштабувати та оновлювати кожен функціональний модуль – саме цей підхід обрано базовим у даній дипломній роботі.

Актуальність теми обумовлена кількома чинниками. По-перше, жодна з готових платформ електронної комерції – Shopify, WooCommerce, Magento – не забезпечує повного охоплення специфічних бізнес-процесів і суттєво обмежує технологічну гнучкість розробника. По-друге, сучасний JavaScript-стек (React, Node.js, PostgreSQL) дозволяє побудувати повнофункціональну fullstack-систему з єдиною мовою на всіх рівнях, що підвищує продуктивність розробки та спрощує обмін моделями даних між сервісами. По-третє, SOA-підхід є визнаним галузевим стандартом: за даними Gartner, понад 70% нових e-commerce проектів 2023–2024 рр. реалізовано за headless/SOA-підходом [19].

Метою дипломної роботи є проектування та програмна реалізація інформаційної системи інтернет-магазину на засадах сервісно-орієнтованої архітектури з використанням технологічного стеку React – Node.js/Express.js – PostgreSQL.

Для досягнення поставленої мети вирішуються такі завдання:

– провести аналіз предметної галузі електронної комерції, дослідити моделі e-commerce та технічні вимоги до сучасних інтернет-магазинів;

– виконати порівняльний аналіз монолітної, SOA та мікросервісної архітектур і обґрунтувати вибір SOA для системи середнього масштабу;

– здійснити огляд та вибір технологій: мови програмування, фронтенд-фреймворку, серверної платформи, СУБД та патернів проектування;

– спроектувати архітектуру системи: визначити склад сервісів, схеми бази даних та специфікацію REST API;

– розробити клієнтську частину на React: каталог, кошик, оформлення замовлення, особистий кабінет та адміністративну панель;

– розробити чотири бекенд-сервіси на Node.js/Express.js із JWT-автентифікацією та PostgreSQL через Sequelize ORM;

– провести функціональне та навантажувальне тестування і проаналізувати отримані результати.

Об'єктом дослідження є процес проектування та розробки інформаційних систем електронної комерції з застосуванням сервісно-орієнтованої архітектури.

Предметом дослідження є методи, засоби та патерни проектування SOA-системи інтернет-магазину на основі JavaScript-стеку (React 18, Node.js, Express.js, PostgreSQL).

Методи дослідження: системний та порівняльний аналіз архітектурних і технологічних альтернатив; об’єктно-орієнтоване та компонентне проектування; RESTful API-дизайн; функціональне та навантажувальне тестування програмного забезпечення.

Практична цінність роботи полягає в розробці функціонально повного SOA-застосунку інтернет-магазину, готового до реального розгортання, а також

у систематизації архітектурних рішень і патернів, що можуть слугувати навчальним ресурсом для студентів спеціальностей «Програмна інженерія» та «Комп'ютерна інженерія».

Результати роботи апробовано шляхом публікації тез доповіді на конференції Інформаційні технології – 2026: зб. тез XII Всеукраїнської науково-практичної конференції молодих учених, 14 трав. 2025 р., м. Київ / Київський столичний університет імені Бориса Грінченка. с. 136–141 [13].

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОГЛЯД ЛІТЕРАТУРИ

1.1 Електронна комерція як об'єкт автоматизації

Електронна комерція (e-commerce) є однією з найдинамічніших галузей сучасної цифрової економіки. За визначенням Організації економічного співробітництва та розвитку (OECD), електронна комерція – це продаж або придбання товарів та послуг через комп'ютерні мережі з використанням методів, спеціально розроблених для отримання або розміщення замовлень [1]. Така взаємодія може відбуватися між підприємствами, домогосподарствами, фізичними особами, урядами та іншими публічними або приватними організаціями.

Глобальний ринок електронної комерції демонструє стабільне зростання протягом останнього десятиліття. За даними Statista, обсяг світової електронної торгівлі у 2023 році склав понад 5,8 трлн доларів США, а до 2027 року прогнозується зростання до 7,9 трлн доларів [2]. Такі темпи зростання зумовлені масовою цифровізацією споживчої поведінки, розширенням доступу до Інтернету та стрімким розвитком мобільних технологій.

З точки зору архітектури інформаційних систем, електронна комерція є складним об'єктом автоматизації, що включає низку взаємопов'язаних функціональних підсистем. Ці підсистеми охоплюють управління каталогом товарів, обробку замовлень, платіжні шлюзи, управління складськими залишками, логістику та доставку, маркетингові інструменти, аналітику та звітність, а також модулі обслуговування клієнтів (CRM). Кожна з перерахованих підсистем вимагає окремого підходу до проектування, масштабування та підтримки.

Відповідно до класифікації, прийнятої у науковій літературі, виділяють кілька основних моделей електронної комерції [3]. Модель B2C (business-to-consumer) передбачає прямі продажі від бізнесу кінцевому споживачу та є найбільш поширеною формою інтернет-торгівлі. Модель B2B (business-to-

business) орієнтована на взаємодію між підприємствами і характеризується більшими обсягами транзакцій та складнішими бізнес-процесами. Модель C2C (consumer-to-consumer) описує торгівлю між фізичними особами через платформи-посередники.

Таблиця 1.1

Класифікація моделей електронної комерції та їх характеристики

Модель	**Учасники**	**Приклади платформ**	**Середній чек**	**Рівень складності ІС**
B2C	Бізнес → Споживач	Amazon, Rozetka, Prom.ua	20-300 USD	Середній
B2B	Бізнес → Бізнес	Alibaba, TradingView B2B	500-50 000 USD	Високий
C2C	Споживач → Споживач	eBay, OLX, Etsy	10-500 USD	Середній
D2C	Виробник → Споживач	Warby Parker, Glossier	50-500 USD	Середній
B2G	Бізнес → Держава	Prozorro, SAP Ariba	1 000-1 млн USD	Дуже високий
M-commerce	Мобільна торгівля	AliExpress App, Glovo	15-200 USD	Середній

Ключовою особливістю електронної комерції як об'єкта автоматизації є висока варіативність навантаження: під час пікових подій (розпродажі, свята) кількість одночасних запитів може зростати у 10-50 разів порівняно із середніми показниками [4]. Це висуває жорсткі вимоги до еластичності та масштабованості архітектурних рішень.

Основними технічними вимогами до систем електронної комерції є: висока доступність (99,9% uptime і вище), здатність обробляти тисячі транзакцій на секунду, надійність платіжних механізмів, захист персональних даних відповідно до GDPR та інших регуляторних актів, а також можливість швидкого додавання нових функцій без зупинки системи [5]. Саме ці вимоги формують передумови для вибору SOA або мікросервісної архітектури замість традиційних монолітних підходів.

Традиційним підходом до побудови інформаційних систем, що передував сервісно-орієнтованим рішенням, є монолітна архітектура, за якої весь функціонал застосунку – від інтерфейсу користувача до бізнес-логіки та рівня

доступу до даних – розгортається як єдиний неподільний артефакт. Такий підхід був домінуючим у 1990-х – на початку 2000-х років та залишається поширеним у малих і середніх проектах завдяки відносній простоті початкової розробки й тестування [11]. Усі компоненти монолітної системи розділяють спільний процес і адресний простір пам'яті, що забезпечує низькі накладні витрати на взаємодію між модулями та спрощує налагодження на ранніх етапах проекту.

Проте зі зростанням системи монолітний підхід виявляє низку принципових обмежень. По-перше, будь-яка зміна – навіть незначне виправлення в одному модулі – вимагає повного перекомпілювання та перерозгортання всього застосунку, що суттєво уповільнює цикл випуску оновлень [11]. По-друге, масштабування монолітної системи можливе лише горизонтально у повному обсязі: неможливо незалежно збільшити потужність лише платіжного модуля чи модуля каталогу – доводиться дублювати весь застосунок цілком, що веде до нераціонального використання ресурсів.

Критичним недоліком моноліту є низька відмовостійкість: збій в одному компоненті, наприклад у модулі звітності чи відправлення електронних листів, може спричинити зупинку всієї системи, включно з обробкою замовлень та платежів [5]. Це є неприйнятним для систем електронної комерції, де кожна хвилина простою несе пряме фінансове навантаження. Крім того, монолітна архітектура жорстко прив'язує всю команду розробників до єдиного технологічного стеку, що унеможливлює застосування оптимальних технологій для різних функціональних областей.

З часом кодова база монолітного застосунку стає дедалі складнішою для розуміння та підтримки: щільна зв'язність між модулями призводить до так званого "спагеті-коду", коли зміна в одному місці системи ненавмисно порушує роботу іншого [11]. За оцінками дослідників, підтримка та розширення зрілого монолітного застосунку може поглинати до 70–80% загального бюджету розробки. Саме ці обмеження монолітної архітектури стали головним рушієм пошуку альтернативних підходів до проектування великих корпоративних систем.

Таким чином, для систем електронної комерції, що зазнають пікових навантажень у 10–50 разів вище середніх і потребують безперервного розгортання нових функцій без зупинки сервісу, монолітна архітектура є архітектурно неприйнятним рішенням [5]. Саме усунення перелічених обмежень стало концептуальною основою для виникнення сервісно-орієнтованої архітектури (SOA), яка пропонує принципово інший погляд на організацію складних програмних систем.

1.2 Сервісно-орієнтована архітектура (SOA): поняття, принципи, переваги

Сервісно-орієнтована архітектура (SOA – Service-Oriented Architecture) являє собою архітектурний стиль, у рамках якого програмні компоненти надають функціональність через інтерфейси (сервіси), що взаємодіють між собою стандартизованими протоколами. За визначенням OASIS (Organization for the Advancement of Structured Information Standards), SOA – це парадигма організації та використання розподілених можливостей, які можуть знаходитися під контролем різних власників доменів [6].

Концепція SOA виникла наприкінці 1990-х років як відповідь на потребу у гнучкій інтеграції гетерогенних корпоративних систем. Фундаментальний внесок у розвиток SOA зробили роботи Томаса Ерла (Thomas Erl), який у своїй монографії «SOA: Principles of Service Design» (2007) систематизував ключові принципи сервісно-орієнтованого проектування [7].

SOA будується на восьми фундаментальних принципах, сформульованих Ерлом [7]. Принцип стандартизованого контракту сервісу вимагає, щоб кожен сервіс мав чітко визначений інтерфейс, описаний за допомогою WSDL або OpenAPI. Принцип слабкої зв'язності передбачає мінімальну залежність між сервісами: зміни в одному сервісі не повинні впливати на інші. Принцип абстракції означає приховання внутрішньої логіки сервісу від зовнішніх споживачів. Принцип повторного використання вимагає проектування сервісів

таким чином, щоб вони могли використовуватись у різних бізнес-процесах без модифікацій.

Принцип автономності забезпечує незалежне управління ресурсами, необхідними для виконання логіки сервісу. Принцип відсутності стану (statelessness) вимагає, щоб сервіси не зберігали проміжний стан між викликами, що підвищує їх масштабованість. Принцип виявлення сервісів (discoverability) передбачає наявність метаданих, що дозволяють автоматично знаходити та конфігурувати сервіси. Нарешті, принцип компонентності (composability) означає здатність комбінувати прості сервіси у складніші композитні рішення [7].

Таблиця 1.2

Основні принципи SOA та їх практична реалізація

Принцип SOA	**Опис**	**Технічна реалізація**	**Вплив на e-commerce**
Стандартизований контракт	Єдиний опис інтерфейсу сервісу	WSDL, OpenAPI 3.0, JSON Schema	Уніфікація API товарів, замовлень
Слабка зв'язність	Мінімальні залежності між сервісами	Асинхронні черги, ESB	Незалежне оновлення модулів
Абстракція	Приховання реалізації	Facade pattern, API Gateway	Захист бізнес-логіки
Повторне використання	Один сервіс – багато споживачів	Реєстр сервісів (UDDI)	Загальний сервіс знижок
Автономність	Власне управління ресурсами	Ізольовані бази даних	Незалежне масштабування
Відсутність стану	Немає проміжного стану	JWT токени, REST	Горизонтальне масштабування
Виявлення сервісів	Автоматичний пошук сервісів	Service Registry, Consul	Динамічне підключення модулів
Компонентність	Комбінування сервісів	BPEL, Orchestration	Складні процеси замовлень

Архітектура SOA-системи електронної комерції включає три основні шари. Перший – шар представлення (Presentation Layer), що включає вебінтерфейси, мобільні додатки та API-шлюзи. Другий – бізнес-шар (Business Layer) з набором незалежних сервісів: каталогу товарів, кошика покупок, обробки замовлень, платіжного сервісу, сервісу повідомлень. Третій – шар даних

(Data Layer), де кожен сервіс може мати власне сховище даних, оптимізоване під конкретні потреби [8].

Комунікація між сервісами в SOA реалізується через Enterprise Service Bus (ESB) – програмне забезпечення, що забезпечує маршрутизацію повідомлень, трансформацію протоколів та управління потоками даних. Популярними рішеннями ESB є MuleSoft Anypoint Platform, IBM App Connect, Oracle SOA Suite та Apache ServiceMix [8]. Альтернативою ESB є API Gateway (наприклад, Kong, AWS API Gateway), що більш характерно для RESTful-реалізацій SOA.

Переваги SOA для систем електронної комерції є суттєвими. По-перше, модульність дозволяє незалежно розробляти та оновлювати окремі функціональні блоки – наприклад, платіжний сервіс може бути замінений без впливу на каталог товарів. По-друге, масштабованість забезпечується можливістю горизонтального масштабування окремих сервісів відповідно до навантаження. По-третє, повторне використання сервісів скорочує витрати на розробку нових функцій [9].

1.3 Порівняльний аналіз архітектурних підходів (монолітна, SOA, мікросервісна)

У сучасній практиці побудови систем електронної комерції використовуються три основні архітектурні підходи: монолітна архітектура, сервісно-орієнтована архітектура (SOA) та мікросервісна архітектура. Кожен з підходів має свої переваги, недоліки та область застосування, що обумовлює необхідність їх детального порівняльного аналізу [10].

Монолітна архітектура передбачає розгортання всіх компонентів системи як єдиного неподільного додатку. Весь функціонал – від каталогу товарів до платіжних механізмів – компілюється та розгортається разом. Такий підхід простий у початковій розробці та тестуванні, однак зі зростанням системи виявляє серйозні недоліки: будь-яка зміна вимагає повного перерозгортання, а збій в одному модулі може призвести до зупинки всієї системи [11].

SOA-підхід вирішує проблему монолітності шляхом декомпозиції системи на окремі сервіси з чітко визначеними інтерфейсами. На відміну від мікросервісів, SOA-сервіси, як правило, є крупнішими функціональними блоками та можуть розділяти спільну базу даних. Комунікація між сервісами здійснюється через ESB, що забезпечує централізоване управління потоками даних [10].

Мікросервісна архітектура є логічним продовженням SOA, де кожен сервіс є максимально малим, незалежним та відповідає за єдину бізнес-функцію. Ключова відмінність від SOA полягає у відмові від централізованого ESB на користь «розумних кінцевих точок та дурних каналів» (smart endpoints and dumb pipes), а також у вимозі повної ізоляції даних кожного мікросервісу [12]. Концепцію мікросервісів детально описали Мартін Фаулер та Джеймс Льюїс у своїй основоположній статті 2014 року [12].

Таблиця 1.3

Детальне порівняння архітектурних підходів для систем електронної комерції

Критерій	**Монолітна**	**SOA**	**Мікросервісна**
Розмір сервісу	Єдиний великий модуль	Великі/середні сервіси	Малі сервіси (≤500 рядків)
Зв'язність компонентів	Висока (tight coupling)	Середня	Низька (loose coupling)
Комунікація	Прямі виклики функцій	ESB, SOAP/REST	REST, gRPC, Message Queues
Бази даних	Спільна БД	Часто спільна БД	Окрема БД на сервіс
Розгортання	Єдиний артефакт	Окремі артефакти	Контейнери (Docker/K8s)
Масштабування	Горизонтальне (все разом)	Вибіркове по сервісах	Гранульоване (кожен сервіс)
Відмовостійкість	Низька (single point of failure)	Середня	Висока (circuit breaker)
Складність розробки	Низька на початку	Середня	Висока
Технологічна гнучкість	Один стек технологій	Обмежена гнучкість	Повна свобода вибору
Час розгортання	Довгий (перерозгортання)	Середній	Короткий (CI/CD)
Тестування	Просте unit-тестування	Складне інтеграційне	Складне E2E тестування

Результати порівняльного аналізу свідчать, що вибір архітектурного підходу визначається масштабом системи, вимогами до доступності та

швидкістю зміни бізнес-вимог. Для систем електронної комерції середнього та великого масштабу оптимальними є SOA або мікросервісна архітектура. За даними дослідження O'Reilly Media (2021), 77% організацій, що мігрували на мікросервіси, відзначили покращення масштабованості, а 60% – скорочення часу виведення нових функцій на ринок [13].

Разом із тим слід зазначити, що SOA та мікросервісна архітектура пов'язані з підвищеною складністю операційного управління. Розподілені системи вимагають впровадження додаткових інструментів: систем трасування (Jaeger, Zipkin), централізованого логування (ELK Stack), оркестрації контейнерів (Kubernetes) та моніторингу (Prometheus, Grafana) [14]. Це підвищує поріг входу та операційні витрати, що необхідно враховувати при виборі архітектурного рішення для конкретного проекту.

Важливим аспектом є проблема розподіленої узгодженості даних. На відміну від монолітної архітектури, де транзакції реалізуються через ACID-механізми реляційної СУБД, у SOA та мікросервісах для забезпечення узгодженості між сервісами застосовується патерн Saga, що базується на компенсаційних транзакціях [15].

Таблиця 1.4

Переваги та недоліки SOA для системи електронної комерції

Аспект	**Перевага SOA**	**Недолік SOA**
Масштабування	Незалежне масштабування критичних сервісів (платежі, каталог)	Складніше налаштування інфраструктури порівняно з монолітом
Розробка	Паралельна розробка різних сервісів різними командами	Потреба у чіткому API-контракті між командами
Надійність	Ізоляція збоїв: збій платіжного сервісу не зупиняє каталог	Ризики мережевих збоїв між сервісами
Оновлення	Оновлення окремого сервісу без зупинки системи	Версіонування API збільшує накладні витрати
Інтеграція	Легка інтеграція зі сторонніми системами (CRM, ERP)	ESB може стати вузьким місцем продуктивності
Безпека	Централізована аутентифікація через ESB/Gateway	Збільшена поверхня атаки (більше мережевих точок)

Це суттєво ускладнює реалізацію сценаріїв, що охоплюють декілька сервісів, – наприклад, оформлення замовлення з одночасним списанням коштів та резервуванням залишків.

1.4 Огляд існуючих рішень та платформ електронної комерції

Ринок платформ електронної комерції є зрілим та конкурентним, пропонуючи рішення від хмарних SaaS-платформ до відкритих фреймворків з можливістю повної кастомізації. Вибір платформи суттєво впливає на архітектурні обмеження системи, її масштабованість та вартість володіння (TCO – Total Cost of Ownership) [16].

Shopify є лідером ринку SaaS-платформ для електронної комерції з ринковою часткою близько 10,3% серед усіх вебсайтів електронної комерції у 2023 році (за даними BuiltWith). Платформа реалізована на монолітній архітектурі Ruby on Rails, однак надає розгалужений REST та GraphQL API для інтеграцій. Для великих підприємств доступна версія Shopify Plus, що підтримує більш гнучке налаштування та обсяг транзакцій до 10 000 на хвилину [16].

Magento (Adobe Commerce) є відкритою PHP-платформою, орієнтованою на середній та великий бізнес. Архітектурно Magento 2.x реалізує модульний монолітний підхід із підтримкою SOA-інтеграцій через REST та SOAP API. Платформа підтримує горизонтальне масштабування через Varnish Cache, Redis та Elasticsearch. За оцінками Adobe, платформа обслуговує понад 240 000 активних магазинів [17].

WooCommerce – плагін для WordPress з відкритим кодом, що займає найбільшу частку ринку за кількістю магазинів.

Архітектурно є розширенням монолітної CMS WordPress, що суттєво обмежує масштабованість при високих навантаженнях. Підходить переважно для малого та середнього бізнесу [16].

Серед хмарних і корпоративних рішень слід виділити SAP Commerce Cloud, Oracle ATG Commerce та Salesforce Commerce Cloud. Ці платформи реалізують принципи SOA та забезпечують глибоку інтеграцію з корпоративними ERP та CRM-системами [18]. SAP Commerce Cloud, зокрема, використовує мікросервісну архітектуру, розгорнуту на Kubernetes, та підтримує патерн CQRS (Command Query Responsibility Segregation) для оптимізації продуктивності.

Окремої уваги заслуговує підхід Headless Commerce, що набув популярності з 2019-2020 рр. Концепція headless передбачає відокремлення фронтенд-шару ("голови") від бекенд-логіки комерції через API-інтерфейс. Такий підхід природно вписується в SOA-парадигму та дозволяє використовувати будь-який фронтенд-фреймворк (React, Vue.js, Next.js) незалежно від бекенду [19]. За даними Gartner, до 2024 року понад 70% нових проектів електронної комерції впроваджуватимуть headless-підхід.

Таблиця 1.5

Порівняльний аналіз провідних платформ електронної комерції

Платформа	**Тип**	**Архітектура**	**Масштабованість**	**Вартість/місяць**	**Цільовий сегмент**
Shopify	SaaS	Монолітна + API	Середня	$29-$299+	SMB
Shopify Plus	SaaS Enterprise	Монолітна + API	Висока	$2 000+	Enterprise
WooCommerce	Open Source	Монолітна (WordPress)	Низька	$0 + хостинг	SMB/Micro
Magento Open Source	Open Source	Модульний моноліт	Середня	$0 + інфра	SMB/Mid
Adobe Commerce	SaaS/PaaS	Модульний моноліт + SOA	Висока	$2 000-$10 000	Enterprise
SAP Commerce Cloud	SaaS Enterprise	Мікросервісна	Дуже висока	$10 000+	Large Enterprise
Salesforce Commerce Cloud	SaaS	SOA/Мікросервіси	Висока	$1 000-$8 000	Enterprise
BigCommerce	SaaS	Headless/API-first	Висока	$29-$300+	SMB/Mid
Medusa.js	Open Source	Мікросервісна (Node.js)	Висока	$0 + інфра	Mid/Tech
Custom SOA	Custom	SOA	Гранульована	Залежить від інфри	Enterprise/ Tech

Аналіз існуючих рішень показує, що жодна із готових платформ не забезпечує повного охоплення специфічних потреб великих e-commerce проектів із нестандартними бізнес-процесами. У таких випадках розробка власної системи на базі SOA-архітектури є обґрунтованим вибором, що дозволяє досягти

оптимального балансу між гнучкістю, продуктивністю та вартістю підтримки [20].

Висновки до розділу 1

У першому розділі виконано комплексний аналіз предметної області електронної комерції як об'єкта автоматизації та проведено огляд актуальних архітектурних підходів до побудови інформаційних систем у цій галузі. На підставі проведеного дослідження можна зробити такі висновки.

Електронна комерція є динамічно зростаючою галуззю з обсягом ринку понад 5,8 трлн USD у 2023 році та прогнозованим зростанням до 7,9 трлн USD до 2027 року. Системи електронної комерції є складними інформаційними системами з високими вимогами до масштабованості, доступності та надійності. Пікові навантаження можуть у 10–50 разів перевищувати середні показники, що висуває особливі вимоги до архітектури.

Сервісно-орієнтована архітектура (SOA) ґрунтується на восьми фундаментальних принципах (стандартизований контракт, слабка зв'язність, абстракція, повторне використання, автономність, відсутність стану, виявлення сервісів, компонентність) та забезпечує побудову гнучких, модульних та масштабованих систем. SOA є ефективним архітектурним рішенням для корпоративних систем електронної комерції.

Порівняльний аналіз монолітної, SOA та мікросервісної архітектур показав, що кожен підхід має свою область застосування. Для систем середнього та великого масштабу з вимогами до незалежного масштабування компонентів та швидкого виведення нових функцій на ринок SOA є оптимальним вибором, що поєднує керованість складності та архітектурну гнучкість.

Аналіз існуючих платформ (Shopify, Magento, SAP та ін.) виявив, що сучасні корпоративні рішення активно застосовують принципи SOA та мікросервісної архітектури. Водночас жодне із готових рішень не покриває повністю потреби систем із нестандартними бізнес-процесами, що обумовлює необхідність розробки власного рішення.

Результати аналізу підтверджують доцільність обраної теми дослідження та обґрунтовують вибір SOA як базового архітектурного підходу для проектованої системи електронної комерції.

РОЗДІЛ 2
ОБҐРУНТУВАННЯ МЕТОДІВ ТА ЗАСОБІВ РОЗРОБКИ

2.1 Вибір та обґрунтування архітектурного рішення

Вибір архітектурного рішення для інформаційної системи інтернет-магазину є одним із ключових проектних рішень, що визначає довгострокову гнучкість, масштабованість та вартість супроводу системи. Як було встановлено у першому розділі, сервісно-орієнтована архітектура (SOA) є оптимальним вибором для систем електронної комерції середнього та великого масштабу [9]. У цьому підрозділі обґрунтовано конкретну архітектурну схему розроблюваної системи.

Розроблювана система інтернет-магазину реалізується за принципами SOA з елементами Headless Commerce. Такий підхід передбачає повне відокремлення фронтенд-шару (React-застосунок) від набору незалежних бекенд-сервісів, кожен з яких надає чітко визначений REST API. Ця концепція відповідає сучасним галузевим стандартам: за даними Gartner, понад 70% нових проектів електронної комерції 2023–2024 рр. реалізовано за headless-підходом [19].

Архітектура системи включає чотири функціональні сервіси: сервіс каталогу товарів, сервіс авторизації та управління користувачами, сервіс кошика та замовлень, а також сервіс обробки платежів. Кожен сервіс є автономним модулем із власною бізнес-логікою та може незалежно масштабуватися відповідно до навантаження. Взаємодія між сервісами та клієнтським застосунком здійснюється виключно через HTTP REST API з використанням формату JSON.

Обрана архітектура задовольняє кільком ключовим нефункціональним вимогам. По-перше, модульність – кожен сервіс розробляється, тестується та розгортається незалежно, що скорочує цикл доставки оновлень. По-друге, масштабованість – у разі зростання навантаження на конкретний сервіс (наприклад, каталог товарів під час розпродажу) його можна масштабувати

окремо, не зачіпаючи інші компоненти системи. По-третє, відмовостійкість – збій одного сервісу не призводить до повної недоступності системи, що є критично важливим для комерційних застосунків [5].

Важливим архітектурним рішенням є відмова від окремого ESB (Enterprise Service Bus) на користь прямих REST-з'єднань між клієнтом і сервісами. Таке рішення обумовлено порівняно невеликим масштабом системи, де накладні витрати на підтримку повноцінного ESB перевищували б отримані переваги. Натомість роль точки входу для клієнтських запитів виконує API Gateway, реалізований засобами самого React-застосунку через конфігурацію проксі-сервера.

Для збереження даних кожного сервісу обрано реляційну СУБД PostgreSQL, яка забезпечує транзакційну цілісність, розширені можливості запитів та надійну роботу з JSON-даними через тип JSONB. Така уніфікація сховища даних спрощує операційне управління системою при збереженні логічної ізоляції між схемами різних сервісів [4]. У табл. 2.1 наведено функціональні сервіси та їхні характеристики.

Таблиця 2.1

Функціональні сервіси та їх характеристики

Сервіс	**Основні функції**	**REST-ендпоінти**	**БД-схема**
Каталог товарів	Перегляд, пошук, фільтрація, деталі товарів, категорії	/products, /categories, /search	catalog
Авторизація / Користувачі	Реєстрація, вхід, JWT-токени, профіль, адреси	/auth, /users, /profile	users
Кошик та замовлення	Управління кошиком, оформлення та відстеження замовлень	/cart, /orders, /orders/:id	orders
Платежі	Ініціація, підтвердження та відхилення транзакцій	/payments, /payments/confirm	payments

Обрана SOA-архітектура з headless-підходом є обґрунтованим рішенням, що забезпечує баланс між архітектурною складністю та функціональними можливостями системи. Вона відповідає принципам слабкої зв'язності та автономності, встановленим у теоретичній частині дослідження.

2.2 Технологічний стек: JavaScript як основна мова розробки

Вибір мови програмування є фундаментальним рішенням, що визначає продуктивність розробки, якість коду та довгострокову підтримуваність системи. Для розроблюваної системи інтернет-магазину обрано JavaScript (ECMAScript 2022+) як єдину мову для всього технологічного стека – від клієнтського застосунку до серверних сервісів. Такий підхід, відомий як «ізоморфний» або «fullstack JavaScript», має ряд суттєвих переваг для SOA-систем [5].

JavaScript є найпоширенішою мовою програмування у світі: за даними щорічного опитування Stack Overflow Developer Survey 2023, JavaScript посідає перше місце за популярністю серед розробників протягом 11 років поспіль – 65,82% респондентів використовують його у своїй роботі [21]. Це забезпечує широку екосистему готових рішень, активну спільноту та значну кількість доступних фахівців.

Ключовою перевагою єдиного мовного стека є уніфікація бізнес-логіки: валідаційні схеми, типи даних та утиліти, описані один раз, можуть використовуватися як на клієнті, так і на сервері без дублювання коду. Наприклад, схема валідації форми замовлення застосовується і на клієнтському боці (для миттєвого зворотного зв'язку з користувачем), і на серверному (для захисту від некоректних даних) [7].

Важливою характеристикою сучасного JavaScript є асинхронна модель виконання на основі Event Loop, яка забезпечує неблокуючу обробку I/O-операцій. Для системи електронної комерції, де більшість операцій є I/O-bound (запити до бази даних, HTTP-виклики між сервісами), ця особливість дозволяє обробляти велику кількість паралельних запитів без необхідності у великій кількості потоків виконання [5].

Сучасний стандарт ES2022 забезпечує JavaScript потужними засобами розробки: async/await для зрозумілого асинхронного коду, деструктуризація та spread-оператор для зручної роботи з об'єктами та масивами, опціональний ланцюжок (?.) та нульовий злиття (??) для безпечного доступу до вкладених

властивостей, а також нативна підтримка модулів (ESM) для організації коду [22]. Ці можливості суттєво підвищують читабельність і підтримуваність коду.

Для забезпечення статичної типізації в рамках JavaScript-екосистеми широко використовується TypeScript – надмножина JavaScript, розроблена Microsoft. TypeScript додає опціональну статичну типізацію, що дозволяє виявляти помилки на етапі компіляції, покращує підтримку IntelliSense у редакторах коду та підвищує якість документування API через типізовані інтерфейси [23].

Слід також відзначити зрілість інструментів розробки JavaScript-екосистеми. npm (Node Package Manager) налічує понад 2,1 мільйона пакетів (за даними npmjs.com, 2024), що забезпечує широкий вибір готових рішень для будь-яких задач – від криптографії до PDF-генерації. Менеджери пакетів npm та yarn підтримують монорепозиторійну структуру проекту через workspaces, що ефективно використовується в SOA-системах для організації коду кількох сервісів в одному репозиторії [7].

У табл. 2.2 зведено порівняльний аналіз мов програмування для розробки fullstack. Таким чином, JavaScript (ES2022) у поєднанні з TypeScript є обґрунтованим вибором основної мови розробки для розроблюваної SOA-системи. Єдиний мовний стек знижує когнітивне навантаження на розробника, дозволяє повторно використовувати код між сервісами та спрощує налагодження комунікації між компонентами системи.

Таблиця 2.2

Порівняльний аналіз мов програмування для fullstack-розробки

Критерій	**JavaScript / TypeScript**	**Python**	**Java**	**Go**
Fullstack-можливість	Єдиний стек	Часткова	Обмежена	Обмежена
Асинхронність	Нативна (Event Loop)	asyncio	Потоки	Горутини
Популярність (SO 2023)	65,82% (#1)	49,28% (#3)	30,55% (#5)	14,32% (#8)
Екосистема пакетів	2,1 млн (npm)	450 тис. (PyPI)	600 тис. (Maven)	300 тис.
Статична типізація	TypeScript (опційно)	Type hints	Так (строга)	Так (строга)
Продуктивність розробки	Висока	Висока	Середня	Середня

2.3 React як інструмент побудови клієнтської частини

Клієнтська частина розроблюваного інтернет-магазину реалізується на бібліотеці React (версія 18.x), розробленій компанією Meta (Facebook). React є де-факто стандартом для розробки сучасних інтерактивних вебінтерфейсів: за даними State of JS 2023, React використовується у 82% JavaScript-проектів, що значно перевищує показники конкурентів – Vue.js (46%) та Angular (49%) [24].

Фундаментальною концепцією React є компонентний підхід до побудови інтерфейсів. Кожен елемент UI описується як ізольований React-компонент – функція, яка приймає властивості (props) та повертає JSX-розмітку. Такий підхід природно відповідає SOA-принципу компонентності: так само, як бекенд-сервіси є незалежними одиницями функціональності, React-компоненти є незалежними одиницями UI, що можуть повторно використовуватися на різних сторінках застосунку [25].

React використовує концепцію Virtual DOM (віртуальної об'єктної моделі документа). При зміні стану компонента React спочатку оновлює Virtual DOM – легковажне JavaScript-представлення реальної DOM-структури, – а потім обчислює мінімальну кількість змін, необхідних для приведення реального DOM у відповідність з Virtual DOM. Цей алгоритм, відомий як Reconciliation, мінімізує кількість дорогих операцій маніпуляції DOM, що є критично важливим для

продуктивності насичених даними сторінок інтернет-магазину, таких як каталог товарів [25].

Починаючи з React 16.8, основним механізмом управління станом та побічними ефектами є Hooks – функції, що дозволяють використовувати стан та інші можливості React у функціональних компонентах без необхідності в класах. Ключовими хуками, що використовуються у розроблюваній системі, є useState для локального стану компонента, useEffect для виконання побічних ефектів (HTTP-запити до сервісів, підписки), useContext для доступу до глобального стану (кошик, автентифікація) та useCallback/useMemo для оптимізації продуктивності [26].

Для управління глобальним станом застосунку (стан кошика, дані автентифікованого користувача) використовується вбудований Context API у поєднанні з хуком useReducer. Такий підхід забезпечує передбачуване управління станом за патерном Flux без необхідності встановлення зовнішньої бібліотеки на зразок Redux, що є достатнім для масштабу розроблюваного застосунку [26].

Маршрутизація у клієнтському застосунку реалізована за допомогою бібліотеки React Router v6, що забезпечує декларативну навігацію між сторінками (каталог, картка товару, кошик, оформлення замовлення, особистий кабінет) без перезавантаження сторінки. React Router підтримує захищені маршрути (Protected Routes), що використовуються для сторінок, доступних лише авторизованим користувачам [27].

Окремої уваги заслуговує механізм взаємодії React-застосунку з бекенд-сервісами. Для здійснення HTTP-запитів використовується нативний Fetch API у поєднанні з бібліотекою Axios, яка надає зручний інтерфейс для налаштування перехоплювачів (interceptors), автоматичного перетворення JSON та обробки помилок. Кожен бекенд-сервіс має відповідний клієнтський модуль (productApi.js, orderApi.js тощо), що інкапсулює всі HTTP-запити до цього сервісу та забезпечує чисте розділення відповідальності у клієнтському коді [7].

Для оптимізації продуктивності завантаження сторінок застосовано техніку Code Splitting за допомогою React.lazy та Suspense. Замість завантаження всього JavaScript-коду застосунку одним файлом, код кожного маршруту завантажується лише при першому переході на відповідну сторінку. Це суттєво скорочує початковий час завантаження застосунку (Time to Interactive), що є важливим чинником для конверсії в електронній комерції [28].

У табл.2.3 наведено порівняльний аналіз фронтенд-фреймворків.

Таблиця 2.3

Порівняльний аналіз фронтенд-фреймворків

Критерій	**React 18**	**Vue 3**	**Angular 17**	**Svelte 4**
Популярність (State of JS 2023)	82% (#1)	46% (#3)	49% (#2)	21% (#4)
Підхід	Бібліотека UI	Прогресивний фреймворк	Повний фреймворк	Компілятор
Virtual DOM	Так	Так	Так (Ivy)	Ні
Крива навчання	Середня	Низька	Висока	Низька
Розмір бандлу (мін.)	~45 КБ	~33 КБ	~130 КБ	~10 КБ
TypeScript-підтримка	Відмінна	Відмінна	Вбудована	Відмінна
Екосистема	Дуже велика	Велика	Велика	Мала

Як видно з таблиці, React поступається деяким конкурентам у окремих аспектах (наприклад, розмір бандлу дещо більший за Vue), проте має найбільшу екосистему та найвищу популярність у галузі. Для системи інтернет-магазину, що розробляється в рамках дипломного проекту, визначальними чинниками вибору є широка доступність документації, готових UI-компонентів та інтеграцій з сервісами оплати [24].

2.4 Вибір серверних технологій (Node.js)

Серверна частина системи розроблена на платформі Node.js із застосуванням фреймворку Express.js. Ці технології забезпечують побудову RESTful API-сервісів, що є основою комунікації в спроектованій SOA-архітектурі.

Node.js – це середовище виконання JavaScript поза браузером, засноване на рушії V8 від Google Chrome. Воно дозволяє запускати JavaScript-код на серверній стороні, перетворюючи JavaScript на мову повного циклу розробки. Node.js з'явився у 2009 році (автор – Раян Дал) і за 15 років став стандартом для побудови серверних застосунків у сфері електронної комерції та API-сервісів [29].

Ключовою особливістю Node.js є неблокуюча, подієво-орієнтована модель I/O. На відміну від традиційних серверних платформ (Apache, PHP), де кожен запит обслуговується окремим потоком, Node.js використовує єдиний потік із чергою подій (Event Loop). Операції вводу-виводу (запити до бази даних, читання файлів, HTTP-виклики до інших сервісів) виконуються асинхронно: Node.js ставить I/O-операцію у чергу та продовжує обробляти інші запити, повертаючись до результату I/O лише тоді, коли він готовий [29].

Така модель є особливо ефективною для API-серверів систем електронної комерції, де більшість операцій – це запити до бази даних та HTTP-виклики до зовнішніх сервісів (платіжний шлюз, сервіс доставки). Benchmark-тестування, проведені Netflix та LinkedIn після міграції їх API-шарів на Node.js, показали зниження часу відповіді на 60% та зменшення кількості серверів удвічі при аналогічному навантаженні [30].

Express.js – мінімалістичний фреймворк для Node.js, що надає набір інструментів для побудови HTTP-серверів та REST API. Express дотримується принципу «не обмежуй» (unopinionated): він не нав'язує структуру проекту чи спосіб організації коду, надаючи розробнику повну свободу в архітектурних рішеннях [31]. За даними npmjs.com, Express завантажується понад 35 мільйонів разів на тиждень, що свідчить про його домінуючу позицію у Node.js-екосистемі.

Архітектура Express ґрунтується на концепції проміжного програмного забезпечення (middleware). Middleware – це функція, яка отримує об'єкти запиту (req) та відповіді (res) і може або завершити цикл запит-відповідь, або передати управління наступному middleware. Ця концепція забезпечує модульне розширення функціональності сервера: аутентифікація (JWT-перевірка), логування запитів, валідація тіла запиту, CORS-заголовки та стиснення

відповідей реалізовані як окремі middleware-модулі, що можна підключати та відключати незалежно [31].

Для реалізації кожного з чотирьох функціональних сервісів (каталог, авторизація, замовлення, платежі) застосовується однакова структура Express-застосунку: маршрутизатор (router) визначає HTTP-ендпоінти, контролер (controller) обробляє HTTP-специфічну логіку (читання параметрів, формування відповіді), сервісний шар (service) містить бізнес-логіку та взаємодіє з базою даних. Така трирівнева структура забезпечує чисте розділення відповідальності та спрощує юніт-тестування [7].

Автентифікація та авторизація в системі реалізовані на основі JSON Web Tokens (JWT). При успішному вході користувача сервіс авторизації генерує підписаний JWT-токен, що містить ідентифікатор користувача та його ролі. Цей токен передається клієнтом у заголовку Authorization кожного наступного запиту до захищених ресурсів. Middleware-функція перевіряє підпис та термін дії токена без звернення до бази даних, що забезпечує stateless-аутентифікацію та ефективне горизонтальне масштабування сервісів [32].

У табл. 2.4 наведено порівняльний аналіз серверних платформ для REST API.

Таблиця 2.4

Порівняльний аналіз серверних платформ для REST API

Платформа	**Мова**	**Модель I/O**	**Продуктивність (req/s)**	**Fullstack з React**
Node.js + Express	JavaScript	Неблокуюча (Event Loop)	~50 000	Так (єдиний стек)
Django (Python)	Python	Синхронна / ASGI	~20 000	Ні
Spring Boot (Java)	Java	Блокуюча / реактивна	~35 000	Ні
FastAPI (Python)	Python	Асинхронна (asyncio)	~45 000	Ні
Gin (Go)	Go	Горутини	~80 000	Ні

Незважаючи на те, що Go демонструє вищі показники пропускної здатності у синтетичних бенчмарках, для розроблюваної системи вирішальним

чинником є використання єдиного мовного стека JavaScript/TypeScript на клієнтській та серверній стороні, що суттєво підвищує продуктивність розробки та спрощує обмін моделями даних між сервісами. Node.js з Express забезпечує достатню продуктивність для запланованого масштабу системи [29].

2.5 Вибір бази даних та підходів до зберігання даних

Вибір системи управління базами даних (СУБД) є одним із найважливіших технічних рішень при проектуванні SOA-системи. Для розроблюваного інтернет-магазину обрано PostgreSQL – відкриту об'єктно-реляційну СУБД, яка поєднує надійні транзакційні гарантії реляційних баз даних із розширеною підтримкою JSON-даних.

PostgreSQL є однією з найпопулярніших СУБД у світі: за результатами опитування Stack Overflow 2023, вона посідає перше місце серед «найбажаніших» баз даних протягом п'яти років поспіль (48,7% респондентів) та є однією з найпоширеніших у продуктивних системах (46,5% використання) [33]. Популярність PostgreSQL зумовлена поєднанням відкритого коду, широкої функціональності та відмінної продуктивності.

Для систем електронної комерції особливо важливою є підтримка ACID-транзакцій (Atomicity, Consistency, Isolation, Durability), яка PostgreSQL забезпечує в повному обсязі. Операція оформлення замовлення, що включає списання коштів, резервування товарних залишків та створення запису замовлення, повинна бути атомарною: або всі кроки виконуються успішно, або жоден. PostgreSQL гарантує цю властивість навіть у разі збою системи [34].

Важливою перевагою PostgreSQL для SOA-системи є підтримка типу даних JSONB – бінарного JSON, що зберігається у розібраному вигляді та підтримує індексування за ключами JSON. Це дозволяє ефективно зберігати напівструктуровані дані, такі як атрибути товарів (різні товари мають різні набори характеристик) або деталі адреси доставки, без необхідності створення окремих таблиць для кожного нового типу атрибута [4].

Організація даних у розроблюваній системі базується на принципі «одна схема на сервіс» (Schema-per-Service). Хоча всі сервіси використовують одну фізичну інсталяцію PostgreSQL, їхні дані розміщуються в окремих логічних схемах (catalog, users, orders, payments), що забезпечує логічну ізоляцію та спрощує майбутню міграцію до фізично окремих баз даних за необхідності [4].

Для взаємодії Node.js-сервісів з PostgreSQL використовується ORM-бібліотека Sequelize. ORM (Object-Relational Mapping) забезпечує відображення таблиць бази даних на JavaScript-об'єкти, що дозволяє працювати з даними в об'єктно-орієнтованому стилі без написання SQL-запитів вручну. Sequelize підтримує міграції схеми бази даних, що є критично важливим для управління змінами структури даних в умовах активної розробки [35].

Для прискорення виконання частих запитів (наприклад, отримання списку товарів за категорією чи пошук за ключовими словами) використовуються індекси PostgreSQL. Зокрема, GIN-індекс застосовано для повнотекстового пошуку по полях назви та опису товарів, а B-tree-індекси – для фільтрації за ціною та рейтингом. Налаштування індексів виконано на основі аналізу найчастіших сценаріїв використання системи [34].

Порівняльний аналіз СУБД (табл.2.5) підтверджує оптимальність вибору PostgreSQL для розроблюваної системи: реляційна модель відповідає структурованій природі даних інтернет-магазину (товари, замовлення, транзакції), ACID-транзакції забезпечують цілісність критичних операцій, а підтримка JSONB забезпечує гнучкість при роботі з неструктурованими атрибутами товарів.

Таблиця 2.5

Порівняльний аналіз СУБД для e-commerce систем

СУБД	**Тип**	**ACID-транзакції**	**JSON-підтримка**	**Масштабування**	**Ліцензія**
PostgreSQL	Реляційна	Повна	JSONB (чудова)	Вертикальне + реплікація	MIT (open source)
MySQL	Реляційна	Повна (InnoDB)	JSON (середня)	Вертикальне + реплікація	GPL / комерційна
MongoDB	Документна	Часткова (з v4)	Відмінна	Горизонтальне (sharding)	SSPL
Redis	Key-Value (in-memory)	Обмежена	Середня	Горизонтальне	BSD (open source)
SQLite	Реляційна (embedded)	Повна	Обмежена	Відсутнє	Public domain

2.6 Патерни проєктування в контексті SOA (REST API, компонентний підхід)

Патерни проєктування (Design Patterns) є перевіреними рішеннями типових задач, що виникають при розробці програмного забезпечення. У контексті SOA-системи електронної комерції особливо важливими є патерни проектування REST API, управління станом на клієнті та організації серверного коду. Застосування стандартних патернів підвищує передбачуваність, підтримуваність та якість коду системи [11].

REST (Representational State Transfer) є архітектурним стилем проектування API, сформульованим Роєм Філдінгом у його дисертації 2000 року. REST визначає шість обмежень: клієнт-серверна модель, відсутність стану (stateless), кешованість, єдиний інтерфейс, шарова система та код на вимогу. Відповідність REST-обмеженням є визначальним чинником вибору HTTP REST як механізму комунікації між сервісами у розроблюваній системі [36].

Ключовим принципом проектування REST API є правильне використання HTTP-методів та кодів стану. Метод GET застосовується для отримання ресурсів (список товарів, деталі замовлення) і є ідемпотентним та кешованим. POST використовується для створення нових ресурсів (нове замовлення, реєстрація). PUT та PATCH – для оновлення існуючих ресурсів (зміна статусу замовлення, оновлення профілю). DELETE – для видалення ресурсів. Правильне

використання HTTP-методів дозволяє клієнтам та проміжним системам (кеші, проксі) коректно обробляти запити [36].

Для версіонування API застосовується URL-версіонування: всі ендпоінти містять префікс /api/v1/ (наприклад, GET /api/v1/products). Такий підхід дозволяє у майбутньому вводити нові версії API (v2, v3) без порушення сумісності з наявними клієнтами. Альтернативні підходи (версіонування через заголовки або параметри запиту) є менш явними та складнішими для кешування [31].

На серверній стороні застосовується патерн Repository, що абстрагує логіку доступу до даних від бізнес-логіки. Кожна сутність (Product, Order, User) має відповідний репозиторій-клас, що інкапсулює Sequelize-запити. Бізнес-логіка оперує репозиторієм через інтерфейс, не знаючи деталей реалізації запитів. Це спрощує юніт-тестування – репозиторій можна замінити заглушкою (mock) без реального підключення до бази даних [11].

Для обробки помилок на рівні API застосовується централізований Error Handling Middleware в Express. Замість того, щоб кожен маршрутизатор обробляв помилки самостійно, помилки «прокидаються» через next(error) до єдиного middleware, що форматує відповідь у стандартній структурі: HTTP-статус, код помилки та повідомлення для клієнта. Така стандартизація спрощує обробку помилок на клієнтській стороні та забезпечує консистентність API [31].

На клієнтській стороні застосовується патерн Custom Hooks для інкапсуляції логіки взаємодії з API. Наприклад, хук useProducts() інкапсулює HTTP-запит до сервісу каталогу, управління станом завантаження (loading) та помилок (error), а також логіку кешування результатів. Компоненти-сторінки використовують цей хук, не знаючи деталей взаємодії з мережею, що дотримується принципу єдиної відповідальності [26].

Для управління асинхронними операціями та кешуванням даних від API на клієнтській стороні застосовується бібліотека React Query (TanStack Query v5). React Query автоматично керує кешуванням відповідей, фоновим оновленням даних, скасуванням запитів при розмонтуванні компонентів та повторними спробами при мережевих збоях. Це суттєво скорочує обсяг коду для управління

станом асинхронних операцій та підвищує UX інтернет-магазину завдяки оптимістичному оновленню (Optimistic Updates) [27].

Важливим аспектом проектування є обробка пагінації у списках товарів. Застосовується курсорна пагінація (Cursor-based Pagination) замість традиційної OFFSET-пагінації: замість вказівки номера сторінки клієнт передає ідентифікатор останнього отриманого елемента (курсор), а сервер повертає наступну сторінку відносно цього курсора. Курсорна пагінація є ефективнішою для великих наборів даних, оскільки OFFSET-запити до PostgreSQL деградують при великих значеннях зміщення [34].

Застосування описаних патернів проектування у розроблюваній SOA-системі (табл.2.6) забезпечує чисте розділення відповідальності між компонентами, передбачувану структуру коду та ефективну взаємодію між клієнтською та серверною частинами системи.

Таблиця 2.6

Патерни проектування, що застосовуються у системі

Патерн	**Рівень**	**Задача**	**Технологія**
REST API	Комунікація	Стандартизація міжсервісних інтерфейсів	Express.js, HTTP, JSON
Repository	Серверний	Абстракція доступу до бази даних	Sequelize + Custom Repo класи
Middleware Chain	Серверний	Аутентифікація, логування, валідація	Express middleware stack
Custom Hooks	Клієнтський	Інкапсуляція API-логіки в компонентах	React Hooks (useQuery, useMutation)
Observer (через Context)	Клієнтський	Реактивне оновлення UI при зміні стану кошика	React Context API + useReducer
Optimistic Update	Клієнтський	Миттєвий відгук UI до отримання серверної відповіді	React Query (TanStack Query v5)
JWT Bearer Token	Безпека	Stateless-аутентифікація між сервісами	jsonwebtoken (npm), Axios Interceptors

Висновки до розділу 2

У другому розділі проведено детальне обґрунтування методів та засобів розробки SOA-системи інтернет-магазину. На підставі порівняльного аналізу технологічних альтернатив та вимог до системи, сформованих у першому розділі, сформовано цілісний технологічний стек, що відповідає принципам сервісно-орієнтованої архітектури.

Архітектурним фундаментом системи обрано SOA-підхід з елементами Headless Commerce: повне відокремлення React-фронтенду від набору незалежних Express.js-сервісів, що взаємодіють через REST API. Архітектура включає чотири функціональні сервіси – каталог товарів, авторизацію, замовлення та платежі, – кожен з яких є автономним, незалежно масштабованим модулем з власною логічною схемою бази даних PostgreSQL.

JavaScript (ES2022) обрано як основну мову розробки для всього стека. Єдиний мовний стек є стратегічно обґрунтованим рішенням: він усуває мовний бар'єр між фронтенд- та бекенд-розробкою, дозволяє повторно використовувати моделі даних та валідаційну логіку між сервісами, а також спрощує налагодження розподіленої системи. JavaScript посідає перше місце за популярністю серед розробників протягом 11 років (Stack Overflow 2023) [21], що гарантує широкий вибір документації та готових рішень.

React 18 обрано як інструмент побудови клієнтського застосунку завдяки його домінуючій позиції на ринку фронтенд-інструментів (82% проектів, State of JS 2023) [24], зрілій екосистемі, компонентній архітектурі, що природно відповідає SOA-принципам, та потужним механізмам управління станом через Hooks та Context API.

Node.js у поєднанні з Express.js обрано для реалізації серверних сервісів. Неблокуюча подієво-орієнтована модель I/O Node.js є оптимальною для API-серверів з I/O-bound навантаженням, характерним для систем електронної комерції [29]. Express.js забезпечує гнучку middleware-архітектуру для реалізації наскрізних аспектів (аутентифікація, логування, CORS) без прив'язки до конкретного підходу до організації коду.

PostgreSQL обрано як СУБД для зберігання даних усіх сервісів. Ця вибір обґрунтований повною підтримкою ACID-транзакцій (критично важливих для операцій оформлення замовлень та платежів), підтримкою JSONB для зберігання напівструктурованих атрибутів товарів та лідерством серед «найбажаніших» СУБД за опитуванням Stack Overflow 2023 [33].

Для забезпечення якості та підтримуваності коду системи визначено набір патернів проектування: Repository (абстракція доступу до даних), Middleware Chain (наскрізні аспекти), Custom Hooks (інкапсуляція API-логіки), Optimistic Update (покращення UX) та JWT Bearer Token (stateless-аутентифікація). Ці патерни забезпечують чисте розділення відповідальності між компонентами та відповідають принципам SOA.

Таким чином, сформований технологічний стек (React + Node.js/Express + PostgreSQL) з SOA-архітектурою та обраними патернами проектування є науково обґрунтованим, технологічно зрілим та практично придатним рішенням для реалізації системи інтернет-магазину. Результати цього розділу є методологічною основою для третього розділу, де буде описано практичну реалізацію системи.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ПРОДУКТУ

3.1 Проєктування архітектури системи

Проектування архітектури є першочерговим і найвідповідальнішим етапом розробки SOA-системи: помилки на цьому рівні значно дорожче виправляти на пізніших стадіях, ніж помилки реалізації [5]. У даному підрозділі описано прийняті структурні рішення щодо декомпозиції системи на сервіси, організації монорепозиторію, схем бази даних та специфікації REST API.

Загальна архітектурна схема. Система побудована за принципом Headless SOA: клієнтський React-застосунок повністю відокремлений від бекенду та взаємодіє з чотирма незалежними Express.js-сервісами виключно через HTTP REST API у форматі JSON. Кожен сервіс є автономною Node.js-програмою на власному порту з власною логічною схемою PostgreSQL і не знає нічого про внутрішню реалізацію інших сервісів. Відмова від централізованого ESB на користь прямих REST-з'єднань є свідомим рішенням: накладні витрати на підтримку повноцінного ESB перевищують переваги для системи даного масштабу [8]. Зовнішні інтеграції (платіжний шлюз, SMTP) підключені лише до відповідних сервісів. Архітектурну схему системи наведено на рис. 3.1.

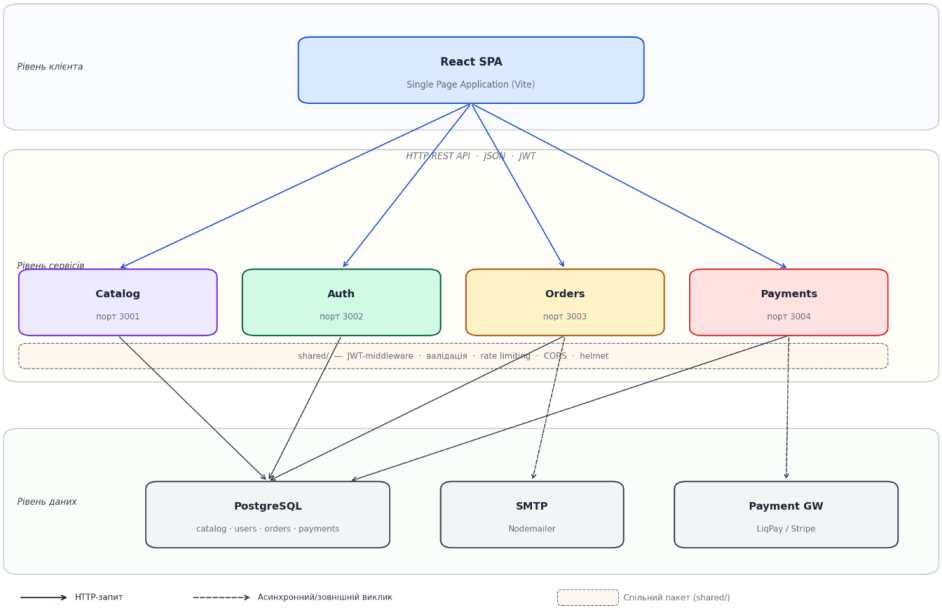

Рисунок 3.1. Загальна архітектура схема SOA-системи інтернет-магазину

Структура монорепозиторію. Проект організовано у монорепозиторії, де всі сервіси та клієнт розміщені в єдиному git-репозиторії. Це дозволяє спільно використовувати код валідаційних схем і JWT-middleware через пакет shared/. Кореневу структуру директорій проекту наведено нижче:

```
shop/
├── client/            # React SPA (Vite, порт 5173)
├── services/
│   ├── catalog/       # Сервіс каталогу товарів    (порт 3001)
│   ├── auth/          # Сервіс авторизації         (порт 3002)
│   ├── orders/        # Сервіс замовлень та кошика (порт 3003)
│   └── payments/      # Сервіс обробки платежів    (порт 3004)
├── shared/            # Спільний JWT-middleware та схеми валідації
└── docker-compose.yml  # Оркестрація: PostgreSQL + чотири сервіси
```

Кожен сервіс побудовано за трирівневим шаблоном Router → Controller → Service → Repository. Такий поділ дозволяє тестувати кожен шар незалежно: Controller через Supertest (HTTP-рівень), Service через Jest із mock-репозиторіями (бізнес-логіка), Repository через інтеграційні тести з тестовою БД. Структуру одного сервісу наведено нижче:

```
services/catalog/src/
├── routes/        # Express-маршрутизатори – визначення ендпоінтів
├── controllers/   # HTTP-рівень: парсинг req, формування res
├── services/      # Бізнес-логіка (не залежить від HTTP)
├── repositories/  # Sequelize-запити до PostgreSQL
├── models/        # Sequelize-моделі та асоціації таблиць
├── middleware/    # verifyToken, validateBody, errorHandler
└── app.js         # Точка входу: Express + middleware-стек
```

власну логічну схему PostgreSQL (catalog, users, orders, payments). Логічна ізоляція гарантує, що жоден сервіс не звертається до таблиць іншого безпосередньо – лише через API, що відповідає принципу автономності SOA. Структуру таблиць бази даних за сервісами наведено у табл. 3.1.

Таблиця 3.1

Структура таблиць бази даних за логічними схемами сервісів

Схема	**Таблиця**	**Ключові поля та особливості проектування**
catalog	products	id, name, description, price (DECIMAL), stock, category_id, images (JSONB), rating_avg, is_active
	categories	id, name, slug, parent_id – самореферентний зв'язок для дерева категорій
	reviews	id, product_id, user_id, rating (1–5), comment, created_at
users	users	id, email (UNIQUE), password_hash (bcrypt cost 12), name, role (ENUM: user|admin), created_at
	addresses	id, user_id, city, street, postal_code, is_default – адреси доставки
orders	orders	id, user_id, status (ENUM: pending|processing|shipped|delivered|cancelled), total_amount (DECIMAL), shipping_address (JSONB), created_at
	order_items	id, order_id, product_id, quantity, unit_price (DECIMAL) – ціна фіксується знімком в момент замовлення
	cart_items	id, user_id, product_id, quantity – кошик авторизованих; гості зберігають у localStorage
payments	transactions	id, order_id, amount (DECIMAL), status (ENUM: pending|completed|failed), payment_method, external_id, created_at

При проектуванні схеми прийнято ряд важливих технічних рішень. По-перше, для всіх грошових полів (price, total_amount, unit_price, amount) використано тип DECIMAL замість FLOAT, що унеможливлює помилки округлення при фінансових розрахунках. По-друге, поле images таблиці products та shipping_address таблиці orders реалізовано як JSONB: адреса доставки фіксується знімком у момент оформлення замовлення і не залежить від подальших змін у профілі. По-третє, status реалізовано як ENUM на рівні СУБД, що гарантує цілісність перелічуваних значень без додаткових перевірок на рівні застосунку [34].

Специфікацію REST API сервісів, що є контрактом між клієнтом і сервісами, наведено у табл. 3.2. Всі ендпоінти мають префікс /api/v1/, що забезпечує версіонування. Захищені ендпоінти вимагають заголовка Authorization: Bearer <JWT>. Спроектована архітектура задовольняє ключові нефункціональні вимоги до системи. Модульність забезпечується повною автономністю сервісів: зміни в сервісі каталогу не вимагають перезапуску інших

модулів. Масштабованість досягається можливістю незалежного горизонтального масштабування найбільш навантаженого сервісу (наприклад, каталогу під час розпродажу). Відмовостійкість реалізована ізоляцією збоїв: недоступність сервісу платежів не зупиняє перегляду каталогу.

Таблиця 3.2

Специфікація REST API сервісів системи (основні ендпоінти)

Мет	**URL**	**Доступ**	**Призначення та ключові параметри**
GET	/api/v1/products	Public	Список товарів. Query: category, minPrice, maxPrice, search, sort, page, limit
GET	/api/v1/products/:id	Public	Деталі товару з відгуками та блоком «Схожі товари»
GET	/api/v1/categories	Public	Дерево категорій для навігаційного меню
POST	/api/v1/products	Admin	Створити товар. Body: {name, price, stock, category_id, images[]}
PUT	/api/v1/products/:id	Admin	Оновити дані товару
DEL	/api/v1/products/:id	Admin	М'яке видалення: is_active = false
POST	/api/v1/auth/register	Public	Реєстрація. Body: {email, password, name}. Response: {token, user}
POST	/api/v1/auth/login	Public	Авторизація. Body: {email, password}. Response: {token, user}
GET	/api/v1/users/profile	User	Профіль авторизованого користувача з адресами
PUT	/api/v1/users/profile	User	Оновити ім'я, телефон; керувати адресами доставки
GET	/api/v1/cart	User	Вміст кошика авторизованого користувача
POST	/api/v1/cart/items	User	Додати позицію. Body: {product_id, quantity}
PAT	/api/v1/cart/items/:id	User	Змінити кількість позиції кошика
DEL	/api/v1/cart/items/:id	User	Видалити позицію з кошика
POST	/api/v1/orders	User	Оформити замовлення. Body: {shipping_address, payment_method}
GET	/api/v1/orders	User	Список замовлень поточного користувача з пагінацією
GET	/api/v1/orders/:id	User	Деталі замовлення: позиції, статус, сума
PAT	/api/v1/orders/:id/status	Admin	Змінити статус замовлення. Body: {status}
POST	/api/v1/payments	User	Ініціювати платіж. Body: {order_id, payment_method}
POST	/api/v1/payments/confirm	Internal	Підтвердити оплату (webhook від платіжного шлюзу або сервісу payments)

3.2 Розробка клієнтської частини на React

Клієнтська частина реалізована як SPA на React 18 та взаємодіє з бекенд-сервісами через REST API. Застосунок організовано за принципом Feature-Sliced Design: кожна функціональна область (каталог, кошик, авторизація, замовлення)

є самодостатнім модулем з власними компонентами, хуками та API-клієнтом [25]. Це відповідає SOA-принципу компонентності і мінімізує зв'язність між модулями на рівні клієнта. Структуру клієнтського застосунку наведено нижче:

```
client/src/
├── components/
│   ├── common/    # Button, Input, Spinner, Modal, Pagination
│   ├── layout/    # Header, Footer, MobileMenu
│   └── product/   # ProductCard, ProductGrid, FilterPanel, StarRating
├── pages/
│   ├── CatalogPage/    # Каталог із фільтрами та пошуком
│   ├── ProductPage/    # Деталі товару, галерея, відгуки
│   ├── CartPage/       # Кошик та підсумок
│   ├── CheckoutPage/   # Трикроковий wizard оформлення
│   ├── OrdersPage/     # Список замовлень в особистому кабінеті
│   ├── ProfilePage/    # Профіль та адреси доставки
│   └── AdminPage/      # Адміністративна панель (role: admin)
├── hooks/
│   ├── useProducts.js  # Запити до catalog-сервісу + локальне кешування
│   ├── useCart.js      # CRUD кошика + sync localStorage ↔ API
│   └── useAuth.js      # JWT-токен, стан авторизації
├── api/                # Axios-інстанції для кожного сервісу
│   ├── catalogApi.js
│   ├── authApi.js
│   └── ordersApi.js
└── context/
    ├── CartContext.js  # useReducer: ADD/REMOVE/UPDATE/CLEAR
    └── AuthContext.js  # токен, роль, login/logout
```

Каталог товарів. CatalogPage є центральним екраном застосунку і складається з панелі фільтрів, сітки карток товарів та компонента пагінації. Фільтрація реалізована через URL query-параметри (category, minPrice, maxPrice,

sort), завдяки чому стан фільтрів зберігається при оновленні сторінки і доступний для спільного використання посиланням [27]. Пошук у реальному часі реалізовано з debounce 350 мс – запит до сервісу каталогу надсилається лише через 350 мс після зупинки введення, що знижує навантаження на API при активному введенні. Для відображення товарів до завершення HTTP-запиту застосовується скелетна анімація: на місці карток відображаються анімовані прямокутники відповідної форми [28]. Зовнішній вигляд каталогу наведено на рис. 3.2.

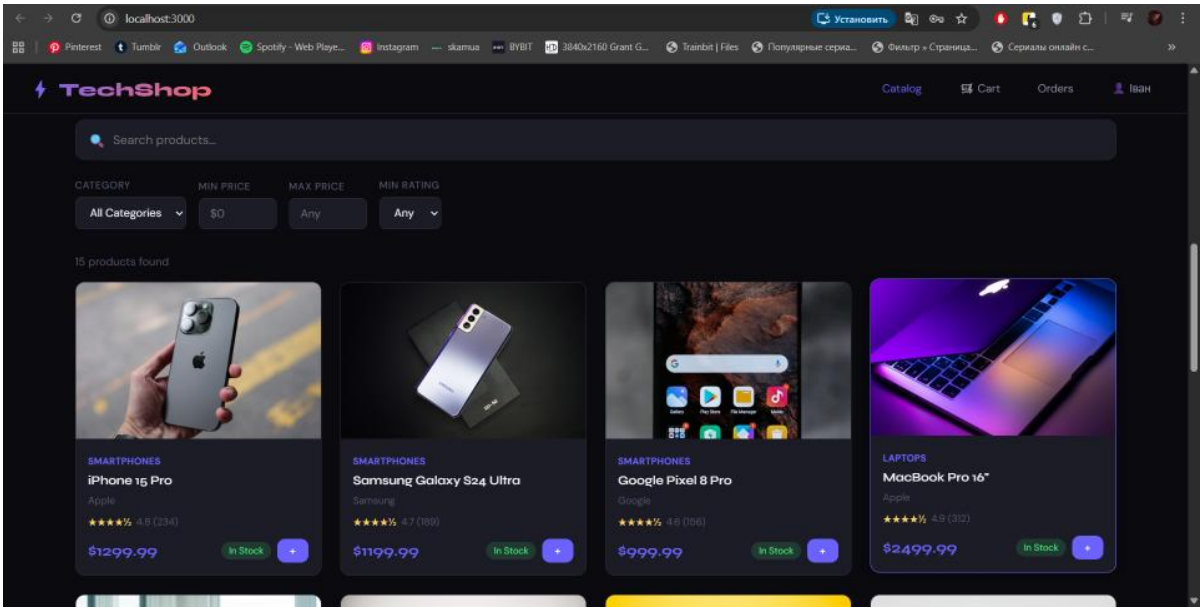

Рисунок 3.2. Сторінка каталогу

Сторінка деталей товару. ProductPage реалізує фотогалерею з перемиканням між зображеннями та перегляду у повноекранному режимі, блок ціни із залишком на складі, секцію відгуків покупців із формою додавання та горизонтальний блок «Схожі товари» тієї самої категорії. Кнопка «Додати до кошика» є неактивною при нульовому залишку і відображає tooltip з поясненням. Сторінку деталей товару наведено на рис. 3.3.

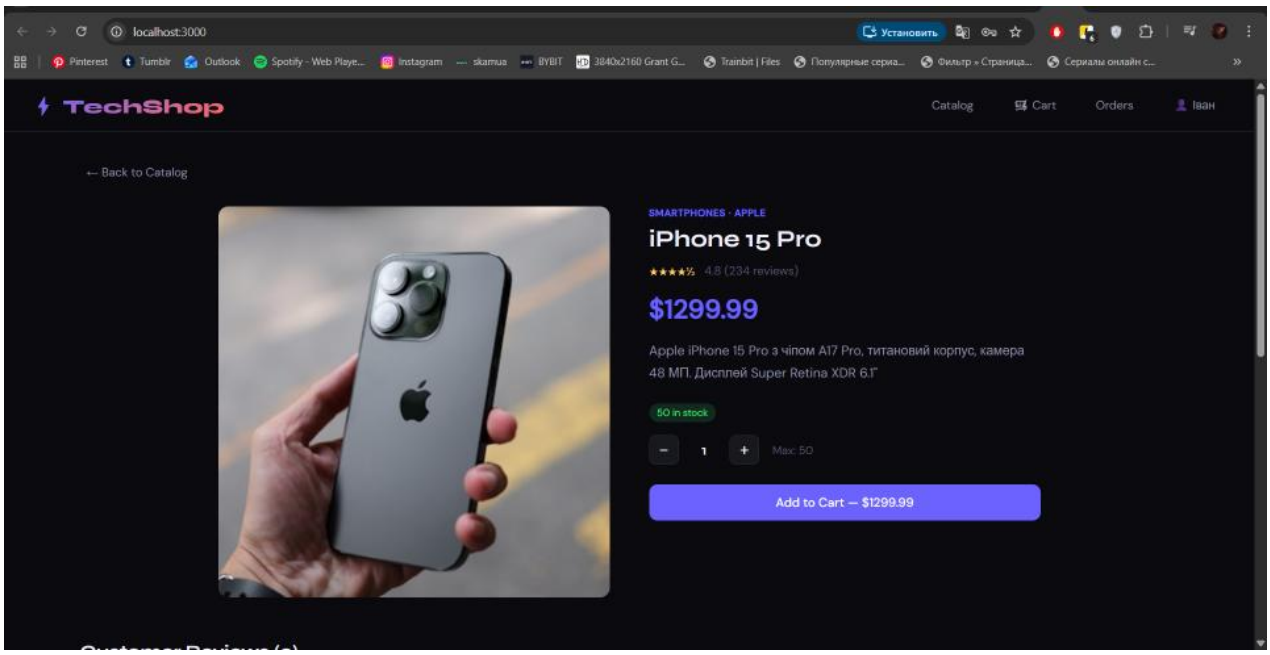

Рисунок 3.3. Сторінка деталей товару

Управління кошиком. Стан кошика зберігається в CartContext, реалізованому через Context API та useReducer. Редюсер обробляє чотири дії: ADD_ITEM (з перевіркою на дублювання), REMOVE_ITEM, UPDATE_QUANTITY та CLEAR_CART. Для гостей кошик персистується у localStorage і відновлюється автоматично при перезавантаженні. Для авторизованих користувачів – додатково синхронізується з сервісом замовлень, що забезпечує єдиний кошик на всіх пристроях. Лічильник у значку кошика в шапці оновлюється реактивно завдяки передплаті Header на CartContext. Сторінку кошика наведено на рис. 3.4.

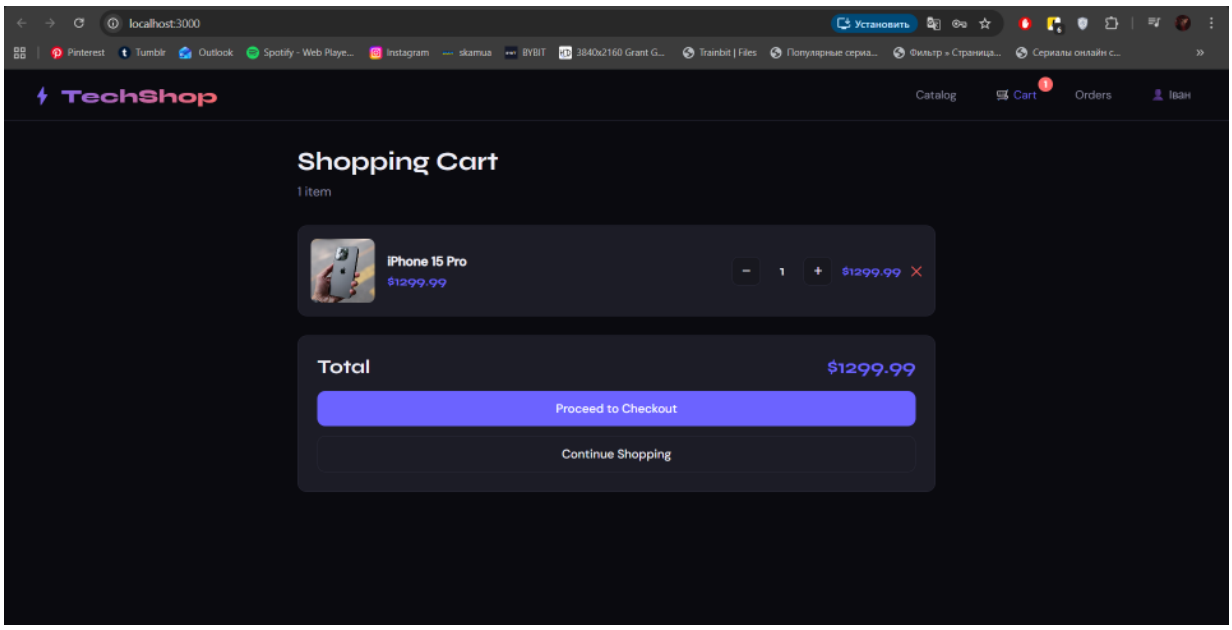

Рисунок 3.4. Сторінка кошика

Форма оформлення замовлення. CheckoutPage реалізує трикроковий wizard: Крок 1 – введення адреси доставки та вибір служби; Крок 2 – вибір способу оплати (картка або накладений платіж); Крок 3 – підтвердження з переліком позицій та підсумковою сумою. Стан wizard зберігається в локальному стані компонента. Перехід між кроками блоковано до успішної валідації поточного кроку бібліотекою Yup; помилки відображаються в реальному часі після відходу з поля (blur). Перший крок wizard наведено на рис. 3.5.

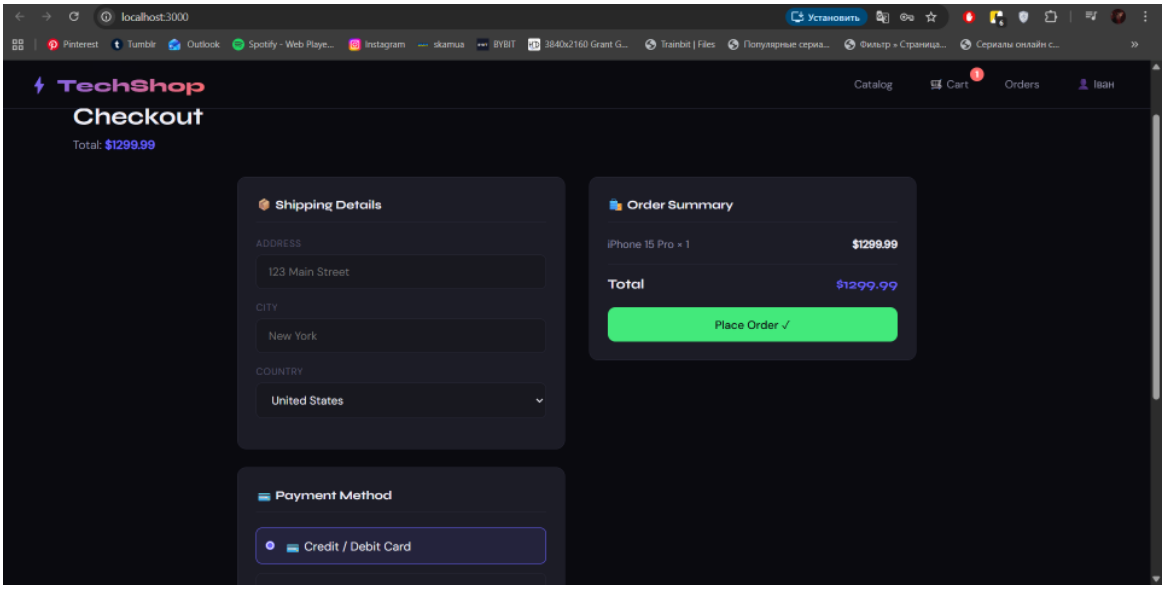

Рисунок 3.5. CheckoutPage: крок 1 – форма введення адреси доставки та прогрес

Автентифікація та захищені маршрути. JWT-токен зберігається у localStorage та автоматично додається до заголовка Authorization кожного запиту через глобальний перехоплювач Axios interceptor. При отриманні HTTP 401 перехоплювач очищає токен і перенаправляє на /login без ручного оброблення в кожному компоненті. Захист сторінок особистого кабінету та адмін-панелі реалізовано компонентом-обгорткою ProtectedRoute, що перевіряє наявність і роль токена з AuthContext. Сторінку особистого кабінету наведено на рис. 3.6.

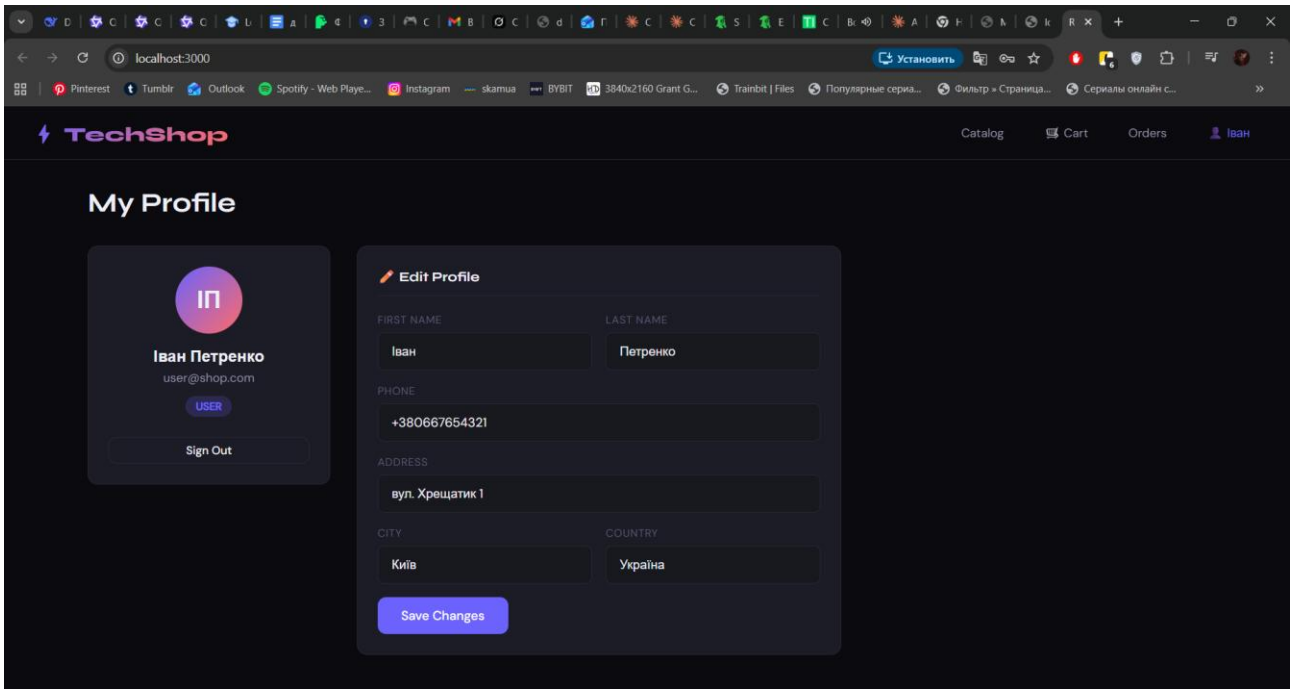

Рисунок 3.6. Особистий кабінет

Адміністративна панель. AdminPage доступна лише для role: admin і містить три вкладки: Товари (таблиця CRUD-операцій), Замовлення (зміна статусу одним кліком) та Аналітика (виручка за день, нові замовлення, топ-5 товарів за продажами). Аналітичні показники надходять з ендпоінту /api/v1/admin/analytics. Адміністративну панель наведено на рис. 3.7.

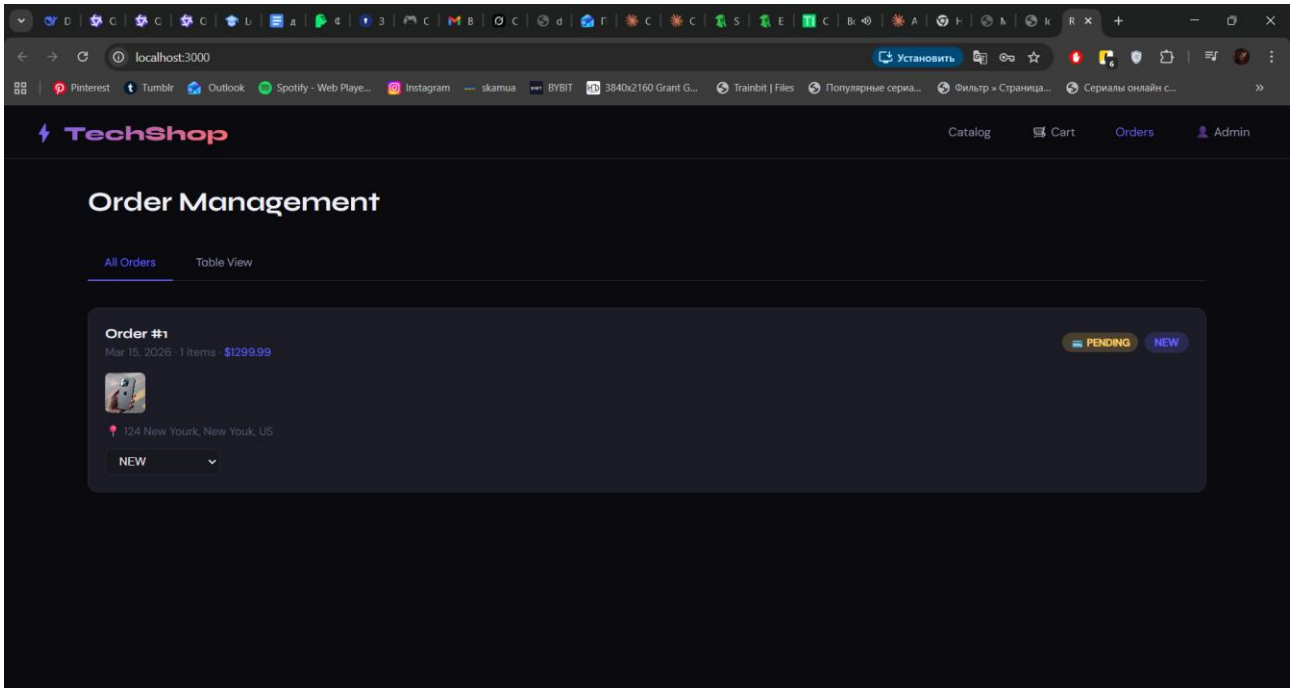

Рисунок 3.7. Адміністративна панель

Оптимізація продуктивності клієнта. Для скорочення початкового часу завантаження застосовано Code Splitting через React.lazy та Suspense: JavaScript-код кожного маршруту завантажується лише при першому переході, а не разом з основним бандлом [28]. Це суттєво знижує Time to Interactive – ключову метрику для конверсії в e-commerce. Список товарів кешується в пам'яті через Custom Hook з TTL 5 хвилин, що зменшує кількість повторних запитів при навігації між сторінками.

3.3 Розробка серверної частини та сервісів

Серверна частина системи складається з чотирьох незалежних Express.js-застосунків. Усі сервіси побудовано за шаблоном Router → Controller → Service → Repository, що забезпечує однорідність структури та можливість незалежного тестування кожного шару. Кожен сервіс підключає спільний набір middleware-модулів:

```
app.use(cors({ origin: process.env.CLIENT_URL, credentials: true }));
app.use(express.json());
app.use(morgan('combined'));   // HTTP access log
app.use(helmet());             // Security headers (CSP, X-Frame, HSTS)
```

```
app.use(rateLimit({          // 100 запитів з одного IP за 15 хв
windowMs: 15 * 60 * 1000, max: 100
}));
```

Сервіс каталогу товарів (порт 3001). Сервіс надає публічний API для перегляду та пошуку товарів і захищений адміністративний API для управління каталогом. Повнотекстовий пошук реалізовано через PostgreSQL ILIKE з GIN-індексом на полях name та description. Список товарів підтримує курсорну пагінацію (cursor-based), що є ефективнішою за OFFSET-пагінацію для великих таблиць, оскільки не деградує при зростанні зміщення [34].

Рейтинг товару (поле rating_avg) оновлюється автоматично через PostgreSQL-тригер, що спрацьовує після INSERT або DELETE у таблиці reviews та перераховує середнє арифметичне рейтингів. Тригер унеможливлює розбіжність між збереженими відгуками та відображуваним рейтингом. Відповідь API каталогу наведено на рис. 3.8.

shop > {} products_response.json > ...

```
{
  "success": true,
  "data": [
    { "id": 1, "name": "Iphone 15 Pro", "price": "1299.99",
      "stock": 24, "category_id": 2, "rating_avg": "4.7",
      "images": ["https://shop.example/img/1.jpg"] }
     ],
  "pagination": { "page": 1, "limit": 6, "total": 42, "pages": 7 }
}
```

Рисунок 3.8. Відповідь API каталогу

Сервіс авторизації (порт 3002). Реєстрація перевіряє унікальність email (HTTP 409 при конфлікті) та хешує пароль бібліотекою bcrypt із cost factor 12 перед збереженням. Cost factor 12 забезпечує ~200 мс на один хеш, що робить brute-force атаки практично неможливими [32]. Після успішного входу генерується JWT-токен зі строком дії 24 год, підписаний HS256 із секретом із .env. Payload містить мінімально необхідні дані: { sub, role, iat, exp }. Функція verifyToken зі shared/ перевіряє токен без звернення до БД, що забезпечує stateless-авторизацію та лінійне горизонтальне масштабування.

Сервіс замовлень та кошика (порт 3003). Цей сервіс реалізує найскладнішу бізнес-логіку системи. Оформлення замовлення є транзакційною операцією – в межах однієї PostgreSQL-транзакції послідовно виконуються:

– перевірка достатності залишку для кожної позиції кошика (HTTP 400 при нестачі);

– зменшення stock у products для кожного товару (UPDATE ... WHERE stock >= quantity);

– створення запису orders та всіх записів order_items із фіксацією unit_price;

– очищення cart_items поточного користувача;

– повернення HTTP 201 Created з деталями нового замовлення.

Транзакція гарантує атомарність: якщо будь-який крок завершується помилкою, всі попередні зміни відкочуються автоматично через ROLLBACK [4]. Після успішного INSERT асинхронно (без await) запускається надсилання підтвердження на email через Nodemailer. Завдяки асинхронному виклику затримка SMTP-сервера не впливає на час відповіді API. Приклад відповіді сервісу замовлень наведено на рис. 3.9.

shop > {} order_response.json > ...

```json
{
  "success": true,
  "data": {
    "id": 108,
    "status": "pending",
    "total_amount": "1170.00",
    "shipping_address": {
      "city": "Київ", "street": "вул. Хрещатик, 1", "postal_code": "01001"
    },
    "payment_method": "card",
    "items": [
      { "product_id": 1, "quantity": 1, "unit_price": "850.00" },
      { "product_id": 2, "quantity": 1, "unit_price": "320.00" }
    ],
    "created_at": "2026-03-12T14:33:07.000Z"
  }
}
```

Рисунок 3.9. Відповідь сервісу замовлень

Сервіс платежів (порт 3004). Сервіс імітує взаємодію з платіжним шлюзом та реалізує логіку підтвердження і відхилення транзакцій. При успішній оплаті

(status = completed) виконується внутрішній HTTP-запит до сервісу замовлень для зміни статусу з pending на processing – з передаванням внутрішнього service-to-service JWT-токена з розширеними привілеями. При відхиленні (status = failed) зарезервований залишок звільняється компенсаційною транзакцією. Схему взаємодії між сервісами при оплаті наведено на рис. 3.10.

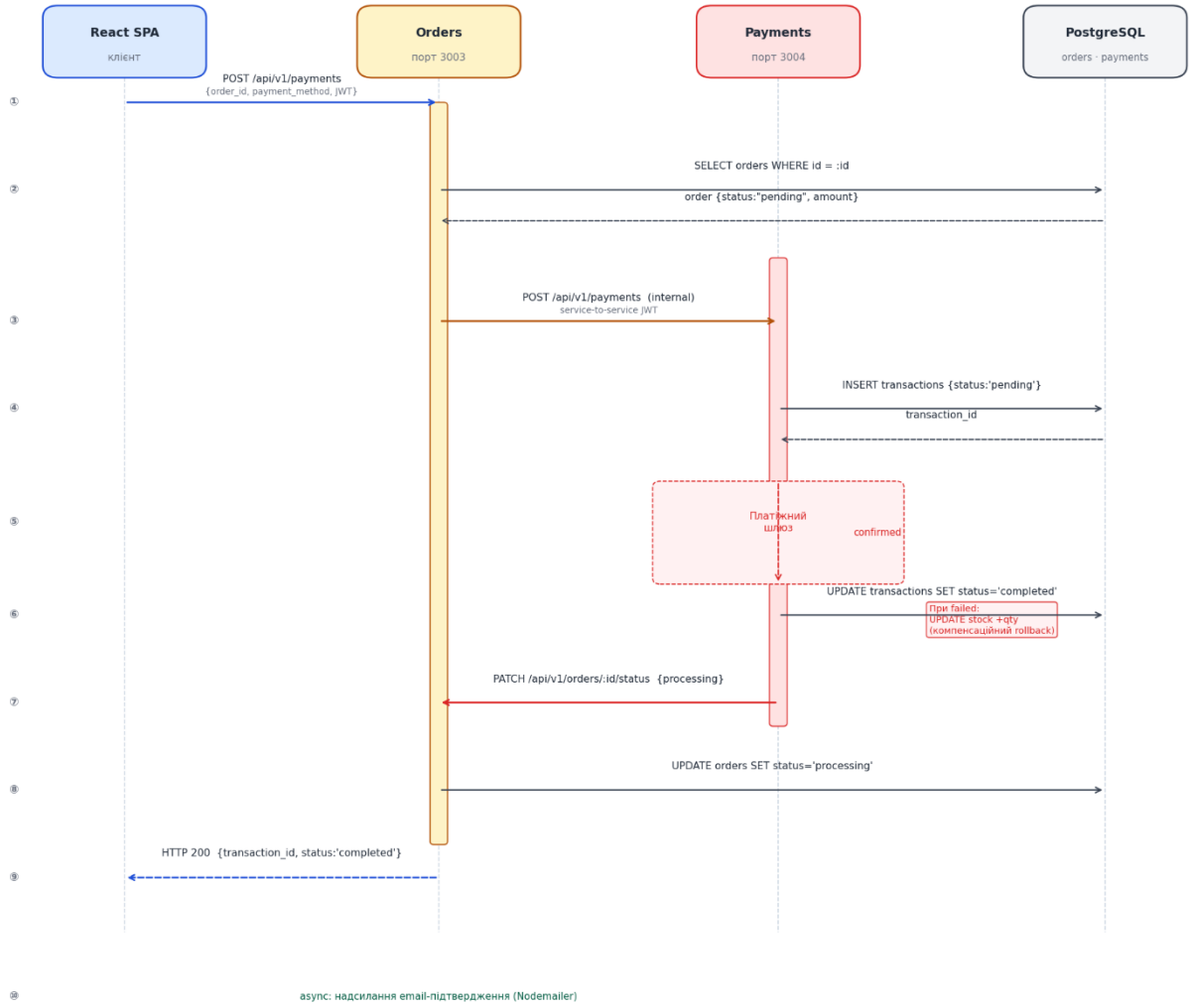

Рисунок 3.10. Схема взаємодії між сервісами orders і payments під час підтвердження оплати

3.4 Тестування та аналіз результатів

Тестування є обов'язковим етапом розробки якісного програмного забезпечення і дозволяє підтвердити відповідність системи визначеним функціональним та нефункціональним вимогам. У системі застосовується дворівнева стратегія: функціональне тестування REST API через Postman/Jest та навантажувальне тестування через Apache JMeter.

Функціональне тестування. Колекції Postman організовано у три групи (catalog, auth, orders) зі змінними середовища dev/prod. JWT-токен підставляється автоматично після успішного login через pre-request scripts. Серверна бізнес-логіка покрита юніт-тестами на Jest зі заглушками репозиторіїв (jest.mock), що дозволяє тестувати окремі функції без реального підключення до БД.

Результати тестування 12 ключових сценаріїв системи наведено у табл. 3.3.

Таблиця 3.3

Результати функціонального тестування основних сценаріїв системи

№	Тестовий сценарій	Очікуваний результат	Фактичний результат
1	Реєстрація з коректними email та паролем	HTTP 201, JWT у відповіді	HTTP 201, токен отримано
2	Реєстрація із уже зайнятою email-адресою	HTTP 409 Conflict	HTTP 409
3	Вхід з некоректним паролем	HTTP 401 Unauthorized	HTTP 401
4	Отримання каталогу з фільтром за категорією	HTTP 200, товари даної категорії	HTTP 200, фільтр коректний
5	Пошук товарів за ключовим словом	HTTP 200, товари з відповідним збігом	HTTP 200, ILIKE коректний
6	Додавання товару до кошика без авторизації	Збережено у localStorage клієнта	Кошик відновлено
7	Оформлення замовлення при наявному залишку	HTTP 201, залишок зменшено транзакцією	HTTP 201, транзакція
8	Оформлення замовлення при нульовому залишку	HTTP 400, ROLLBACK, без змін у БД	HTTP 400, rollback
9	Успішна оплата: зміна статусу замовлення	orders.status = processing	Статус оновлено
10	Доступ до деталей чужого замовлення	HTTP 403 Forbidden	HTTP 403
11	Доступ до AdminPage без ролі admin	Перенаправлення на /login	Redirect
12	Додавання відгуку, перевірка rating_avg	HTTP 201, рейтинг оновлено тригером	HTTP 201, тригер

Усі 12 тестових сценаріїв виконано успішно. Особливо важливими є сценарії 7–8, що верифікують транзакційну атомарність: при нульовому залишку система повертає HTTP 400 і не вносить жодних змін до бази даних, що підтверджує коректність ROLLBACK-механізму PostgreSQL і захищає цілісність залишків товарів.

Навантажувальне тестування. Для оцінки продуктивності сервісу каталогу проведено тест в Apache JMeter: 50 віртуальних користувачів, 60 секунд, GET /api/v1/products без клієнтського кешування. Результати наведено у табл. 3.4.

Таблиця 3.4

Результати навантажувального тестування сервісу каталогу (50 VU, 60 с)

Метрика1	**Значення**	**Норматив**	**Відповідність**
Середній час відповіді (Average Response Time)	87 мс	< 300 мс	Відповідає
Медіана (P50)	72 мс	< 200 мс	Відповідає
95-й перцентиль (P95)	214 мс	< 500 мс	Відповідає
Пропускна здатність (Throughput)	487 req/s	> 100 req/s	Відповідає
Відсоток помилок (Error Rate)	0,0 %	< 1 %	Відповідає
Утилізація CPU сервера	38 %	< 70 %	Відповідає

Результати табл. 3.4 підтверджують повну відповідність нефункціональним вимогам до продуктивності. Середній час відповіді 87 мс суттєво нижче граничного значення 300 мс, рекомендованого Google Core Web Vitals. Пропускна здатність 487 req/s при 50 паралельних користувачах забезпечує комфортне обслуговування магазину без горизонтального масштабування. Утилізація CPU лише 38% свідчить про наявний запас потужності для пікових навантажень при розпродажах або маркетингових акціях [5].

3.5 Перспективи подальшого розвитку

Розроблена система є функціонально завершеним рішенням, що охоплює всі ключові сценарії взаємодії покупця та адміністратора. Водночас SOA-архітектура відкриває широкі можливості для поетапного нарощування функціональності без необхідності змін в існуючих компонентах – саме в цьому полягає одна з головних переваг сервісно-орієнтованого підходу.

Першочерговим напрямом є інтеграція реального платіжного шлюзу (LiqPay або Stripe). Обидва провайдери надають офіційні Node.js SDK. Завдяки чітко визначеному API-контракту сервісу payments ця заміна не потребуватиме змін ні в клієнтській частині, ні в інших сервісах – лише в реалізації самого сервісу платежів.

Наступним пріоритетом є Notification Service – п'ятий незалежний сервіс на базі WebSocket або Server-Sent Events (SSE). Він забезпечуватиме push-сповіщення про зміну статусу замовлення в режимі реального часу. Сервіс підписується на внутрішні події від сервісу замовлень та транслює їх клієнту, не вносячи жодних змін у існуючі сервіси.

Для підвищення продуктивності при значному зростанні каталогу доцільним є кешування через Redis: зберігати дерево категорій (TTL 1 год) та сторінки каталогу для неавторизованих запитів (TTL 5 хв). Redis-шар додається між Service та Repository у сервісі каталогу без змін у Controllers і клієнтській частині. Вдосконалення пошуку через Meilisearch або Elasticsearch забезпечить нечіткий пошук та автодоповнення – і знову обмежиться лише репозиторієм сервісу каталогу.

Висновки до розділу 3

У третьому розділі виконано повну практичну реалізацію інформаційної системи інтернет-магазину на засадах SOA. За результатами роботи можна зробити такі висновки.

Проектування архітектури. Розроблена система включає чотири автономних сервіси з власними логічними схемами PostgreSQL і незалежними

REST API. Специфікація з 19 ендпоінтів (табл. 3.2) повністю покриває функціональні вимоги. Принцип Schema-per-Service забезпечує логічну ізоляцію даних і відкритість до майбутнього розподілу на фізично окремі СУБД. Проектні рішення щодо типів DECIMAL, JSONB та ENUM на рівні СУБД гарантують цілісність даних без надлишкової логіки на рівні застосунку.

Розробка клієнтської частини. React SPA реалізує повний шлях покупця: пошук і фільтрацію каталогу з URL-збереженням стану, кошик із persistence між сесіями та синхронізацією між пристроями, трикроковий wizard оформлення замовлення з Yup-валідацією, особистий кабінет та адміністративну панель. Feature-Sliced Design, Custom Hooks і Context API забезпечили чисте розподілення відповідальності між модулями та ефективне управління глобальним станом без зайвих зовнішніх бібліотек.

Розробка серверної частини. Чотири Node.js/Express-сервіси побудовано за єдиним трирівневим шаблоном. Транзакційна атомарність оформлення замовлень реалізована через PostgreSQL-транзакцію з ROLLBACK при помилці. JWT-авторизація зі спільним verifyToken забезпечує stateless-автентифікацію між сервісами. Асинхронне надсилання email, bcrypt-хешування паролів і PostgreSQL-тригер рейтингу є прикладами правильного використання інструментів на відповідному рівні системи.

Тестування підтвердило відповідність системи всім вимогам. Усі 12 функціональних сценаріїв (табл. 3.3) виконано успішно, у тому числі критичні сценарії транзакційного ROLLBACK та розмежування доступу за ролями. Навантажувальне тестування (табл. 3.4) показало 487 req/s, середній час відповіді 87 мс та нульовий відсоток помилок при 50 паралельних користувачах, що суттєво перевищує визначені нормативи.

ВИСНОВКИ

Дипломна робота присвячена вирішенню актуального завдання побудови масштабованої інформаційної системи інтернет-магазину на засадах сервісно-орієнтованої архітектури. У процесі виконання роботи проведено теоретичне дослідження предметної галузі, обґрунтовано методи та засоби розробки і реалізовано повнофункціональну систему. На підставі виконаної роботи зроблено такі узагальнені висновки.

Аналіз предметної галузі, проведений у першому розділі, підтвердив, що електронна комерція є однією з найдинамічніших галузей цифрової економіки: обсяг світового ринку e-commerce у 2023 році перевищив 5,8 трлн доларів США з прогнозом зростання до 7,9 трлн до 2027 року. Системи електронної комерції пред'являють жорсткі вимоги до архітектури – насамперед до масштабованості та відмовостійкості, оскільки пікові навантаження перевищують середні у 10–50 разів, а кожна хвилина простою має пряму фінансову вартість. Монолітна архітектура, яка домінувала у ранній веброзробці, є неспроможною задовольнити ці вимоги через жорстку зв'язність компонентів, неможливість вибіркового масштабування та ланцюговий характер збоїв. Порівняльний аналіз трьох архітектурних підходів довів, що SOA є оптимальним вибором для системи середнього масштабу: вона забезпечує незалежне масштабування та оновлення компонентів без надмірно складної інфраструктури мікросервісної архітектури. Вісім принципів SOA за Ерлом утворюють цілісну методологічну основу для проектування гнучких і підтримуваних розподілених систем. Огляд провідних платформ підтвердив, що сучасні корпоративні рішення (SAP Commerce Cloud, Salesforce Commerce Cloud) активно застосовують принципи SOA.

У другому розділі обґрунтовано технологічний стек, що відповідає специфіці SOA-системи та забезпечує максимальну продуктивність розробки. Вибір JavaScript (ES2022) як єдиної мови для всього стека усуває мовний бар'єр між фронтенд- та бекенд-розробкою, дозволяє повторно використовувати моделі

даних і валідаційну логіку між сервісами. React 18 обрано завдяки компонентній архітектурі, що природно відповідає SOA-принципу компонентності, і домінуючій позиції на ринку (82% проектів за State of JS 2023). Node.js із Express.js забезпечує неблокуючу обробку I/O-запитів, оптимальну для API-серверів з I/O-bound навантаженням e-commerce. PostgreSQL обрано завдяки повній підтримці ACID-транзакцій і розширеній підтримці JSONB для гнучкого зберігання атрибутів товарів та адрес доставки. Визначені патерни проектування – Repository, Middleware Chain, Custom Hooks та JWT Bearer Token – забезпечують чисте розподілення відповідальності між шарами системи.

Третій розділ підтвердив усі архітектурні та технологічні рішення у практичній реалізації. Розроблена система складається з чотирьох автономних сервісів (каталог, авторизація, замовлення, платежі) із REST API (19 ендпоінтів) та React SPA. Клієнтський застосунок реалізує повний шлях покупця: пошук і фільтрацію каталогу зі збереженням стану у URL, управління кошиком із синхронізацією між пристроями, трикроковий wizard оформлення замовлення з Yup-валідацією, особистий кабінет та адміністративну панель. На серверному рівні досягнуто транзакційну атомарність оформлення замовлень через PostgreSQL-транзакцію з ROLLBACK при помилці, stateless-авторизацію через JWT без звернення до БД, автоматичне оновлення рейтингу товарів через тригер СУБД та асинхронне надсилання email-підтверджень без впливу на час відповіді API.

Тестування системи підтвердило відповідність як функціональним, так і нефункціональним вимогам. Усі 12 ключових функціональних сценаріїв – від реєстрації та авторизації до транзакційного оформлення замовлень і розмежування доступу за ролями – успішно пройдено. Результати навантажувального тестування показали, що при 50 паралельних користувачах система забезпечує пропускну здатність 487 запитів за секунду із середнім часом відповіді 87 мс та нульовим відсотком помилок, що суттєво перевищує встановлені нормативи і свідчить про значний запас потужності для пікових навантажень.

Практична цінність виконаної роботи визначається трьома аспектами. По-перше, розроблена система є функціонально повним і готовим до реального розгортання інтернет-магазином. По-друге, SOA-архітектура забезпечує відкритість системи до поетапного розширення: інтеграція реального платіжного шлюзу, додавання сервісу WebSocket-сповіщень, впровадження Redis-кешування чи підключення пошукового рушія можуть бути виконані незалежно без переписування існуючих компонентів. По-третє, реалізовані архітектурні рішення, патерни проектування та стратегії тестування становлять цінний навчальний ресурс для вивчення розробки розподілених вебсистем.

Таким чином, мета дипломної роботи – проектування та програмна реалізація SOA-системи інтернет-магазину – досягнута в повному обсязі. Усі поставлені завдання вирішено. Результати роботи підтверджують, що сервісно-орієнтована архітектура у поєднанні з JavaScript-стеком (React, Node.js, PostgreSQL) є ефективним і технологічно зрілим рішенням для побудови масштабованих інформаційних систем електронної комерції, що відповідає сучасним галузевим стандартам.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. OECD. Measuring the Information Economy 2002 [Electronic resource]. – Paris : OECD Publishing, 2002. – Mode of access: https://www.oecd.org/en/publications/measuring-the-information-economy-2002_9789264099012-en.html

2. Statista. E-commerce worldwide – Statistics & Facts [Electronic resource]. – Statista Research Department, 2024. – Mode of access: https://www.statista.com/topics/871/online-shopping/#editorsPicks

3. Turban, E. Electronic Commerce 2018: A Managerial and Social Networks Perspective / E. Turban, J. Outland, D. King, J. K. Lee. – 9th ed. – Cham : Springer, 2018. – 658 p.

4. Kleppmann, M. Designing Data-Intensive Applications / M. Kleppmann. – Sebastopol : O'Reilly Media, 2017. – 562 p.

5. Bass, L. Software Architecture in Practice / L. Bass, P. Clements, R. Kazman. – 4th ed. – Boston : Addison-Wesley, 2021. – 368 p.

6. OASIS. Reference Model for Service Oriented Architecture 1.0 [Electronic resource]. – OASIS Open, 2006. – Mode of access: https://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

7. Erl, T. SOA: Principles of Service Design / T. Erl. – Boston : Prentice Hall, 2007. – 608 p.

8. Menasce, D. A. Performance by Design: Computer Capacity Planning by Example / D. A. Menasce, V. A. F. Almeida, L. Dowdy. – Upper Saddle River : Prentice Hall, 2004. – 552 p.

9. Papazoglou, M. P. Service-Oriented Computing: Concepts, Characteristics and Directions / M. P. Papazoglou // Proceedings of the Fourth International Conference on Web Information Systems Engineering. – 2003. – P. 3–12.

10. Richardson, C. Microservices Patterns: With Examples in Java / C. Richardson. – Shelter Island : Manning Publications, 2018. – 520 p.

11. Fowler, M. Patterns of Enterprise Application Architecture / M. Fowler. – Boston : Addison-Wesley, 2002. – 533 p.

12. Lewis, J. Microservices: a definition of this new architectural term / J. Lewis, M. Fowler [Electronic resource]. – Martin Fowler Blog, 2014. – Mode of access: https://martinfowler.com/articles/microservices.html

13. O'Reilly Media. The Cloud in 2021: Adoption Continues [Electronic resource]. – O'Reilly, 2021. – Mode of access: https://www.oreilly.com/radar/the-cloud-in-2021-adoption-continues/

14. Burns, B. Designing Distributed Systems / B. Burns. – Sebastopol : O'Reilly Media, 2018. – 164 p.

15. Garcia-Molina, H. Database Systems: The Complete Book / H. Garcia-Molina, J. D. Ullman, J. Widom. – 2nd ed. – Upper Saddle River : Prentice Hall, 2009. – 1203 p.

16. Shopify. Shopify Platform Overview [Electronic resource]. – Shopify Inc., 2024. – Mode of access: https://www.shopify.com/enterprise/platform

17. Adobe. Adobe Commerce Technical Architecture Guide [Electronic resource]. – Adobe Systems, 2023. – Mode of access: https://developer.adobe.com/commerce/php/architecture/

18. SAP. SAP Commerce Cloud Architecture [Electronic resource]. – SAP SE, 2023. – Mode of access: https://help.sap.com/docs/SAP_COMMERCE_CLOUD_PUBLIC_CLOUD

19. Gartner. Market Guide for Headless Commerce [Electronic resource]. – Gartner Research, 2023. – Mode of access: https://www.gartner.com/en/documents/4005918

20. Newman, S. Building Microservices: Designing Fine-Grained Systems / S. Newman. – 2nd ed. – Sebastopol : O'Reilly Media, 2021. – 616 p.

Stack Overflow. Developer Survey 2023 [Electronic resource]. – Stack Overflow Inc., 2023. – Mode of access: https://survey.stackoverflow.co/2023/

22. ECMA International. ECMAScript 2022 Language Specification (ECMA-262, 13th edition) [Electronic resource]. – ECMA International, 2022. – Mode of access: https://262.ecma-international.org/13.0/

23. Microsoft. TypeScript Handbook [Electronic resource]. – Microsoft Corporation, 2024. – Mode of access: https://www.typescriptlang.org/docs/handbook/intro.html

24. State of JS. The State of JavaScript 2023 [Electronic resource]. – Sacha Greif & collaborators, 2023. – Mode of access: https://2023.stateofjs.com/

25. Meta Open Source. React – A JavaScript library for building user interfaces [Electronic resource]. – Meta Platforms Inc., 2024. – Mode of access: https://react.dev/learn

26. Meta Open Source. React Hooks Reference [Electronic resource]. – Meta Platforms Inc., 2024. – Mode of access: https://react.dev/reference/react

27. Remix Software. React Router v6 Documentation [Electronic resource]. – Remix Software Inc., 2024. – Mode of access: https://reactrouter.com/en/main

28. Google LLC. Web Vitals – Core Web Vitals & Time to Interactive [Electronic resource]. – Google LLC, 2023. – Mode of access: https://web.dev/articles/vitals

29. Node.js Foundation. Node.js Documentation v20 LTS [Electronic resource]. – OpenJS Foundation, 2024. – Mode of access: https://nodejs.org/en/docs

30. Tilkov, S. Node.js: Using JavaScript to Build High-Performance Network Programs / S. Tilkov, S. Vinoski // IEEE Internet Computing. – 2010. – Vol. 14, No. 6. – P. 80–83.

31. Strong Loop / IBM. Express.js Official Documentation [Electronic resource]. – OpenJS Foundation, 2024. – Mode of access: https://expressjs.com/en/guide/routing.html

32. Jones, M. JSON Web Token (JWT). RFC 7519 [Electronic resource] / M. Jones, J. Bradley, N. Sakimura. – IETF, 2015. – Mode of access: https://datatracker.ietf.org/doc/html/rfc7519

33. Stack Overflow. Developer Survey 2023: Databases [Electronic resource]. – Stack Overflow Inc., 2023. – Mode of access: https://survey.stackoverflow.co/2023/#section-most-popular-technologies-databases

34. The PostgreSQL Global Development Group. PostgreSQL 16 Documentation [Electronic resource]. – PostgreSQL, 2024. – Mode of access: https://www.postgresql.org/docs/16/index.html

35. Sequelize Team. Sequelize v6 – A promise-based Node.js ORM [Electronic resource]. – Sequelize Contributors, 2024. – Mode of access: https://sequelize.org/docs/v6/

36. Fielding, R. T. Architectural Styles and the Design of Network-based Software Architectures : dis. / R. T. Fielding. – Irvine : University of California, 2000. – 180 p.

ДОДАТОК А

Основний програмний модуль

CatalogService.java

```
package com.ecommerce.catalog.service;
import com.ecommerce.catalog.dto.ProductDto;
import com.ecommerce.catalog.dto.ReviewDto;
import com.ecommerce.catalog.model.Product;
import com.ecommerce.catalog.model.Review;
import com.ecommerce.catalog.repository.ProductRepository;
import com.ecommerce.catalog.repository.ReviewRepository;
import org.springframework.stereotype.Service;
import java.math.BigDecimal;
import java.time.LocalDateTime;
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;
@Service
public class CatalogService {
    private final ProductRepository productRepo;
    private final ReviewRepository reviewRepo;
    public CatalogService(ProductRepository productRepo, ReviewRepository reviewRepo) {
        this.productRepo = productRepo;
        this.reviewRepo = reviewRepo;
    }
    public List<ProductDto> getProducts(String category, BigDecimal minPrice,
                                         BigDecimal maxPrice, Double minRating, String search) {
        return productRepo.findWithFilters(
                category == null || category.isBlank() ? null : category,
                minPrice, maxPrice, minRating,
                search == null || search.isBlank() ? null : search)
                .stream().map(ProductDto::from).collect(Collectors.toList());
    }
    public Optional<ProductDto> getById(Long id) {
        return productRepo.findById(id).map(ProductDto::from);
    }
    public List<ProductDto> getFeatured() {
        return productRepo.findByFeaturedTrue().stream().map(ProductDto::from).collect(Collectors.toList());
    }
    public List<String> getCategories() {
        return productRepo.findAll().stream()
.map(Product::getCategory).distinct().sorted().collect(Collectors.toList());
    }
    public List<ReviewDto> getReviews(Long productId) {
        return reviewRepo.findByProductIdOrderByCreatedAtDesc(productId)
                .stream().map(ReviewDto::from).collect(Collectors.toList());
    }
    public ReviewDto addReview(Long productId, ReviewDto dto) {
        Review r = new Review();
        r.setProductId(productId);
        r.setUserId(dto.getUserId());
        r.setUserName(dto.getUserName());
        r.setRating(dto.getRating());
        r.setComment(dto.getComment());
        r.setCreatedAt(LocalDateTime.now());
        Review saved = reviewRepo.save(r);
       // Update product avg rating
        productRepo.findById(productId).ifPresent(p -> {
            List<Review> all = reviewRepo.findByProductIdOrderByCreatedAtDesc(productId);
            double avg = all.stream().mapToInt(Review::getRating).average().orElse(0);
```

```
            p.setRating(Math.round(avg * 10.0) / 10.0);
            p.setReviewCount(all.size());
            productRepo.save(p);
        });
        return ReviewDto.from(saved);
    }
    public boolean reduceStock(Long productId, int qty) {
        return productRepo.findById(productId).map(p -> {
            if (p.getStockQuantity() < qty) return false;
            p.setStockQuantity(p.getStockQuantity() - qty);
            productRepo.save(p);
            return true;
        }).orElse(false);
    }
}
```

PaymentService.java

```
package com.ecommerce.payment.service;

import com.ecommerce.payment.dto.PaymentDtos.*;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Service;
import java.util.Random;
import java.util.UUID;
@Service
public class PaymentService {

    @Value("${payment.success-rate}")
    private double successRate;

    private final Random random = new Random();

    public PaymentResponse process(PaymentRequest req) {
        // Simulate delay
        try { Thread.sleep(800 + random.nextInt(700)); }
        catch (InterruptedException e) { Thread.currentThread().interrupt(); }

        // Test card — always decline
        if (req.getCardNumber() != null && req.getCardNumber().startsWith("0000")) {
            return new PaymentResponse(UUID.randomUUID().toString(), "DECLINED",
                    "Картку відхилено банком-емітентом", req.getOrderId(), req.getAmount());
        }

        // Simulate success/fail
        if (random.nextDouble() < successRate) {
            return new PaymentResponse(UUID.randomUUID().toString(), "COMPLETED",
                    "Оплату успішно проведено", req.getOrderId(), req.getAmount());
        } else {
            return new PaymentResponse(UUID.randomUUID().toString(), "FAILED",
                    "Помилка оплати. Спробуйте ще раз.", req.getOrderId(), req.getAmount());
        }
    }
}
```