МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК ТА ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНИХ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач випускової кафедри
_____ Аліна САВЧЕНКО
«___»_____2023 р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ВИПУСКНИКА ОСВІТНЬОГО СТУПЕНЯ МАГІСТР
ЗА ОСВІТНЬО-ПРОФЕСІЙНОЮ ПРОГРАМОЮ
«ІНФОРМАЦІЙНІ ТЕХНОЛОГІЇ ПРОЕКТУВАННЯ»

Тема: «Інтернет-магазин з нереляційною базою даних MongoDB»

Виконавець: Дмитро ГРОМАДСЬКИЙ

Керівник: к.т.н., доцент Олена ТОЛСТІКОВА

Нормоконтролер: к.т.н., доцент Олена ТОЛСТІКОВА

КИЇВ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет *комп'ютерних наук та технологій*
Кафедра *комп'ютерних інформаційних технологій*
Спеціальність *122 «Комп'ютерні науки»*
Освітньо-професійна програма *«Інформаційні технології проектування»*

ЗАТВЕРДЖУЮ:
завідувач кафедри КІТ
_____Аліна САВЧЕНКО
(підпис)
«____»_____2023 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Громадського Дмитра Володимировича
(ПІБ випускника)

1. Тема роботи: «Інтернет-магазин з нереляційною базою даних MongoDB» затверджена наказом ректора № 1976/ст від 29.09.2023р.
2. Термін виконання роботи: з 02 жовтня 2023 року по 31 грудня 2023 року.
3. Вихідні дані до роботи: застосунок на мові програмування JavaScript з функціоналом інтернет магазину.
4. Зміст пояснювальної записки: 1. Аналіз та поняття технології. 2. Проектування застосунку. 3. Розробка та тестування застосунку.
5. Перелік обов'язкового ілюстративного матеріалу: 1. Приклад структури ієрархічної бази даних. 2. Діаграма зв'язків сутностей. 3. Приклад структури графової бази даних. 4. Приклад структури реліаційної бази даних. 5. Приклад структури бази даних ключ-значення. 6. Приклад документної бази даних. 7. Приклад структури стовбчикової бази даних. 8. Сутність користувача. 9. Сутність продукту. 10. Сутність продукту.

6. Календарний план-графік

№ з/п	Завдання	Термін виконання	Підпис керівника
1.	Аналіз предметної області та огляд аналогів. Написання 1 розділу, аналіз та поняття технології	02.10.2023-15.10.2023	
2.	Вибір та опис використаних технологій. Написання 2 розділу, проектування застосунку	16.10.2023-28.10.2023	
3.	Написання 3 розділу, розробка та тестування застосунку	30.10.2023-15.11.2023	
4.	Загальне редагування та друк пояснювальної записки	15.11.2023-21.11.2023	
5.	Проходження нормоконтролю, перепліт пояснювальної записки.	22.11.2023-24.11.2023	
6.	Написання тексту доповіді. Оформлення графічного матеріалу для презентації	01.12.2023-21.12.2023	

7. Дата видачі завдання 02.10.2023 р.

Керівник кваліфікаційної роботи _____ Олена ТОЛСТІКОВА
(підпис керівника)

Завдання прийняв до виконання _____ Дмитро ГРОМАДСЬКИЙ
(підпис випускника)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи на тему «Інтернет-магазин з нереляційною базою даних MongoDB» містить: 92 сторінки, 32 рисунок, 15 інформаційних джерела, 1 додаток.

Об'єкт дослідження – інтернет-магазин як бізнес-модель, який використовує нереляційну базу даних MongoDB для управління своїми даними.

Предмет дослідження – процеси та методи, які використовуються для розробки та управління інтернет-магазином з використанням MongoDB.

Мета кваліфікаційної роботи – дослідження нереляційних баз даних, аналіз ефективності MongoDB, розробка функіонуючого прототипу інтернет магазину.

Методи дослідження – мова програмування JavaScript, інтегроване середовище розробки VsCode**,** система управління базами даних MongoDB.

Результати кваліфікаційної роботи рекомендується та може бути використаним як фундамент реального магазину.

ІНТЕРНЕТ САЙТ, БАЗИ ДАНИХ, ВЕЛИКІ НАВАНТАЖЕННЯ, БЕЗПЕКА , NodeJS, ПРОЕКТУВАННЯ СТРУКТУРИ БАЗИ ДАНИХ

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНІВ

SDK (Software development kit)	–	Набір для розробки програмного забезпечення
IDE (Integrated development. environment)	–	Інтегроване середовище розробки
API (Application programming interface)	–	Програмний інтерфейс
URL (Uniform Resource Locator	–	Визначник місцезнаходження сайту в мережі Інтернет

ВСТУП

У сучасному цифровому світі, де гнучкість та швидкість обробки даних є ключовими факторами успіху, вибір правильної технології для бази даних є вирішальним. Ця кваліфікаційна робота присвячена розробці інтернет-магазину, який використовує нереляційну базу даних MongoDB, відому своєю гнучкістю та масштабованістю. Вона досліджує, як така система може сприяти кращому управлінню даними, забезпечуючи водночас швидкий доступ та легке масштабування. Аналізуючи можливості MongoDB, робота розкриває, як ця технологія може вдосконалити роботу інтернет-магазину, забезпечуючи ефективність та високу продуктивність.

Першим кроком є збір і аналіз академічних та індустріальних джерел, пов'язаних з MongoDB, нереляційними базами даних, та інтернет-магазинами. Це включає вивчення теоретичних основ, існуючих рішень та найкращих практик. Визначення конкретних проблем чи викликів, на які робота буде зосереджена. Також важливо визначити мету та цілі дослідження.

Другим кроком є розробка концепції інтернет-магазину, вибір архітектури, технологій та інструментів. Створення детального плану проекту, включаючи базу даних, серверну та клієнтську частини. Кодування та налаштування інтернет-магазину з використанням MongoDB. Впровадження функціональностей, як-от каталог продуктів, система замовлень, управління користувачами, інтеграція з платіжними системами тощо.

Також проведення комплексних тестів для перевірки функціональності, продуктивності, безпеки та масштабованості інтернет-магазину. Використання як ручних, так і автоматизованих тестів. Оцінка отриманих результатів у порівнянні з поставленими цілями та завданнями. Аналіз ефективності використання MongoDB та його впливу на загальну роботу системи. Написання детального звіту про виконану роботу, включаючи теоретичні висновки, опис розробки та результати тестування. Формулювання рекомендацій для подальшого використання та розвитку системи.

Метою кваліфікаційної роботи є дослідження нереляційних баз даних: глибокий аналіз нереляційних баз даних, з особливим акцентом на MongoDB, для розуміння їхніх основних властивостей, функціоналу та переваг у контексті електронної комерції.

Для досягнення поставленої мети необхідне рішення наступних завдань:

1. Аналіз ефективності MongoDB;
2. Вивчення Можливостей для Оптимізації;
3. Розробка рекомендацій для використання MongoDB у електронній комерці;
4. Аналіз перспектив розвитку.
5. Розробка прототипу інтернет-магазину

Об'єктом досліджень є інтернет-магазин як бізнес-модель, який використовує нереляційну базу даних MongoDB для управління своїми даними.

Предметом досліджень є процеси та методи, які використовуються для розробки та управління інтернет-магазином з використанням MongoDB. Основні аспекти цього предмету включають:

1. Архітектура та Розробка MongoDB: Детальне вивчення архітектури MongoDB, способів реалізації нереляційних баз даних та їхньої інтеграції в інтернет-магазин.
2. Моделювання Даних у MongoDB: Аналіз способів структурування та зберігання даних у MongoDB, включаючи використання документів, колекцій та їхніх відносин.
3. Управління Даними: Методи управління даними, включаючи запити, оновлення, видалення та агрегацію даних в контексті інтернет-магаз

Актуальність теми кваліфікаційної роботи «Інтернет-магазин з нереляційною базою даних MongoDB» ґрунтується на тому, що з огляду на глобальний розвиток цифрових технологій та зростання онлайн-торгівлі, інтернет-магазини стають все більш актуальними. Розвиток інтернет-магазинів вимагає ефективних технологічних рішень для управління великими

обсягами даних та забезпечення високої продуктивності. В сучасному світі, де кількість даних, які необхідно обробляти, швидко зростає, MongoDB пропонує ефективні рішення для управління великими обсягами неструктурованих даних, що є критично важливим для інтернет-магазинів. Нереляційні бази даних, такі як MongoDB, набувають популярності завдяки своїй гнучкості та масштабованості. Це робить їх відмінним вибором для інтернет-магазинів, де потрібно швидко адаптуватися до змін у даних та користувацькому попиті. Використання передових технологій, як-от MongoDB, в інтернет-магазинах сприяє інноваціям та дозволяє бізнесам ефективно конкурувати на ринку. Забезпечення безпеки та високої продуктивності є ключовими вимогами для інтернет-магазинів. MongoDB надає рішення, що відповідають цим вимогам, завдяки вбудованим можливостям шифрування даних та високій продуктивності обробки запитів.

Наукова новизна роботи може бути визначена наступними пунктами:

1. Інноваційний Підхід до Моделювання Даних: Розробка нових методів моделювання даних для інтернет-магазинів з використанням можливостей нереляційних баз даних, які можуть пропонувати ефективніші способи зберігання та обробки даних порівняно з традиційними реляційними базами даних.

1. Вдосконалення Алгоритмів Обробки Запитів: Розробка або вдосконалення алгоритмів для оптимізації обробки запитів в MongoDB, що може включати кешування, паралелізацію запитів та ефективне використання індексів.

2. Аналіз Продуктивності та Масштабованості: Проведення детального аналізу продуктивності та масштабованості MongoDB у контексті інтернет-магазинів, що може виявити нові підходи до оптимізації системи для великих обсягів даних.

3. Розвиток Методів Забезпечення Безпеки: Дослідження та розробка нових стратегій та методів захисту даних в MongoDB, особливо враховуючи виклики, пов'язані з безпекою в електронній комерції.

4. Інтеграція з Сучасними Технологіями: Вивчення можливостей інтеграції MongoDB з іншими сучасними технологіями, такими як штучний інтелект, машинне навчання, або IoT, що може відкрити нові шляхи для інновацій в електронній комерції.

Проєкт має перспективу користуватись великим попитом у зв'язку з широкою доступністю, зручністю та зрозумілістю для користувача. З використанням MongoDB в інтернет-магазині, можна очікувати значного прогресу в області управління даними. MongoDB забезпечує гнучкість, швидкість обробки даних та легкість масштабування, що є критично важливим для розвитку сучасних електронних комерційних платформ. Покращення безпеки даних та надійності платформи може бути ще однією перспективою. MongoDB пропонує різні функції для забезпечення безпеки даних, що є важливим для захисту конфіденційності користувачів і бізнес-інформації. Масштабованість, яку пропонує MongoDB, дозволяє легко адаптувати інтернет-магазин до змінюваних потреб бізнесу та ринку, що є ключовим фактором для довгострокового успіху. Використання MongoDB може сприяти більш ефективному та швидкому доступу до даних, що, в свою чергу, покращує користувацький досвід при перегляді продуктів, пошуку інформації та оформленні замовлень. MongoDB може бути інтегрована з іншими технологіями, такими як штучний інтелект, машинне навчання, та IoT, що відкриває нові можливості для інновацій у сфері електронної комерції.

РОЗДІЛ 1
АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Дослідження терміну база даних

База даних – це організована колекція даних, яка зберігається та доступна електронно з комп'ютерної системи. Вона дозволяє зберігати великі обсяги інформації систематизовано, таким чином, щоб було легко знаходити, управляти та оновлювати дані.

Дані в базі даних зазвичай структуровані в таблиці, які складаються з рядків та стовпців. Кожен рядок представляє один запис, а стовпці – різні атрибути або характеристики цього запису. Бази даних забезпечують різні інструменти та функції для управління даними, включаючи створення, оновлення, видалення та запити на дані. Бази даних використовують індекси для підвищення швидкості пошуку та доступу до даних. Індекси дозволяють швидко знаходити записи без необхідності просматривати всю базу даних.

Бази даних надають механізми для забезпечення безпеки даних, включаючи контроль доступу, шифрування та забезпечення цілісності даних (запобігання втраті або пошкодженню даних).

Сучасні бази даних розроблені для забезпечення високої масштабованості та надійності, що важливо для великих систем та додатків, де обробка великих обсягів даних є критичною.

Існують різні типи баз даних, включаючи реляційні (наприклад, MySQL, PostgreSQL), нереляційні або NoSQL (наприклад, MongoDB, Cassandra), та об'єктно-орієнтовані бази даних.

Кожен тип бази даних має свої унікальні характеристики та найкраще підходить для певних видів додатків та вимог обробки даних.

1.2. Оглад існуючих типів баз даних

Плоска база даних (Flat-file database) найпростіший спосіб управління даними на комп'ютері поза додатком - це зберігання їх у базовому форматі файлу. Перші рішення для управління даними використовували цей підхід, і він досі є популярним варіантом для зберігання невеликих обсягів інформації без великих вимог. Плоскі файли баз даних представляють інформацію в регулярних структурах, які може обробити машина, всередині файлів. Дані зберігаються у вигляді простого тексту, що обмежує тип вмісту, який може бути представлений у самій базі даних. Іноді вибирається спеціальний символ або інший індикатор, який використовується як роздільник, або маркер для визначення кінця одного поля та початку наступного. Наприклад, кома використовується у файлах CSV (значення, розділені комами), тоді як двокрапки або пробіли використовуються в багатьох файлах даних в системах, схожих на Unix. Інші рази роздільник не використовується, а поля визначаються фіксованою довжиною, яку можна доповнити для коротших значень.

`/etc/passwd` **on *nix systems:**

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
syslog:x:102:106::/home/syslog:/usr/sbin/nologin
bob:x:1000:1000:Bob Smith,,,:/home/bob:/bin/bash
```

Файл */etc/passwd* в системах Unix-подібних визначає користувачів, по одному на рядок. Кожен користувач має атрибути, такі як ім'я, ідентифікатори

користувача та групи, домашній каталог та стандартний оболонку, кожен з яких відділений двокрапкою.

Плоскі файли баз даних, хоча й прості, обмежені у рівні складності, яку вони можуть обробити. Системи, що читають чи маніпулюють даними, не можуть легко встановлювати зв'язки між представленими даними. Зазвичай вони також не мають функцій конкурентності користувачів чи даних. Практично використовуються зазвичай тільки в системах із невеликими вимогами до читання чи запису. Наприклад, багато операційних систем використовують плоскі файли для зберігання конфігураційних даних. Незважаючи на ці обмеження, вони все ще широко використовуються в сценаріях, де локальні процеси потребують зберігання та організації невеликих обсягів даних. Приклади: файли /etc/passwd and /etc/fstab on Linux and Unix-like systems, CSV.

Ієрархічні бази даних використовують відносини батько-дитина для відображення даних у вигляді дерев. Ієрархічні бази даних були вперше введені в 1960-х роках. Вони використовують структуру, схожу на дерево, для організації даних, де кожен запис має один батьківський елемент і може мати декілька дочірніх елементів. Ця модель спрощує навігацію та управління даними, але може бути обмежуючою з точки зору гнучкості та взаємозв'язків між різними типами даних.

Це просте відображення зв’язків надає користувачам можливість встановлювати зв’язки між елементами в структурі дерева. Це дуже корисно для певних типів даних, але не дозволяє керувати складними зв’язками. Крім того, значення стосунків «батьки-дитина» є неявним. Один зв’язок між батьками та дітьми може бути між клієнтом та його замовленнями, тоді як інший може представляти працівника та обладнання, яке їм було призначено. Сама структура даних не розрізняє ці зв’язки.

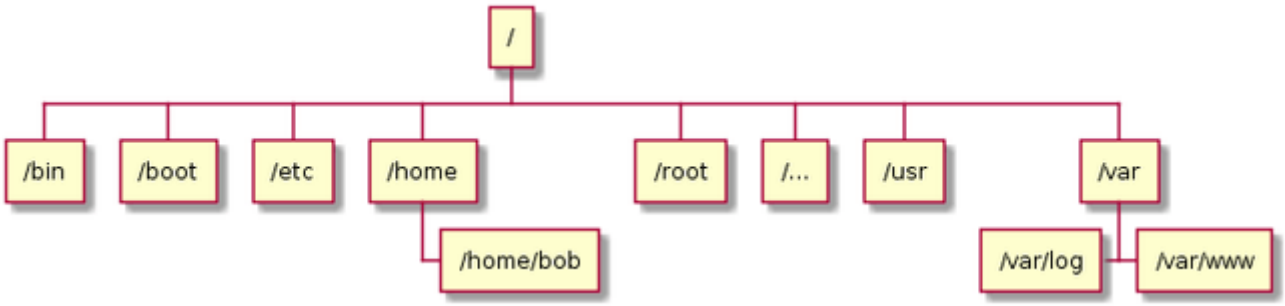

Рис. 1.1. Приклад структури ієрархічної бази даних

Ієрархічні бази даних є початком руху до мислення про управління даними в більш складних термінах. Траєкторія систем керування базами даних, які були розроблені пізніше, продовжує цю тенденцію.

Ієрархічні бази даних сьогодні не використовуються дуже часто через їхню обмежену здатність упорядковувати більшість даних і через накладні витрати на доступ до даних шляхом обходу ієрархії. Однак кілька неймовірно важливих систем можна вважати ієрархічними базами даних. Файлову систему, наприклад, можна розглядати як спеціалізовану ієрархічну базу даних, оскільки система файлів і каталогів чітко вписується в парадигму одного батька/кілька дочірніх систем. Подібним чином системи DNS і LDAP діють як бази даних для ієрархічних наборів даних.

Приклади: Файлові системи, DNS, LDAP.

Мережеві бази даних відображення більш гнучких зв’язків із неієрархічними зв’язками Мережеві бази даних були вперше введені в 1960-х роках. Мережеві бази даних, створені на основі ієрархічних баз даних, додаючи додаткову гнучкість. Замість того, щоб завжди мати одного батьківського елемента, як в ієрархічних базах даних, записи мережевої бази даних можуть мати більше ніж одного батьківського елемента, що фактично дозволяє їм моделювати складніші зв’язки. Говорячи про мережеві бази даних, важливо розуміти, що мережа використовується для позначення зв’язків між різними записами даних, а не зв’язків між різними комп’ютерами чи програмним забезпеченням.

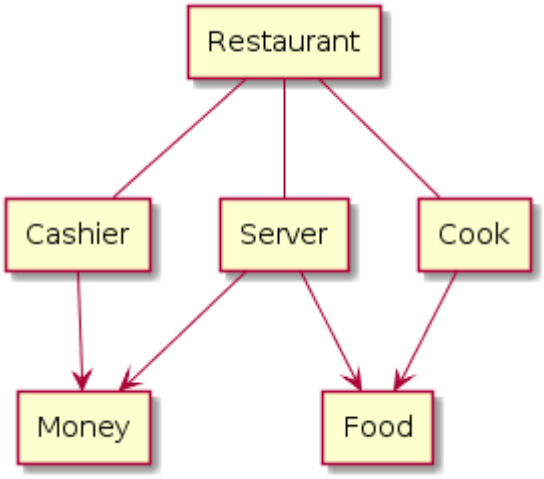

Рис. 1.2. Приклад структури мережевої бази даних

Мережеві бази даних можуть бути представлені загальним графом замість дерева. Значення графіка було визначено схемою, яка описує, що представляє кожен вузол даних і кожен зв'язок. Це надало структурі даних у спосіб, який раніше можна було досягти лише шляхом логічного висновку.

Схема бази даних — це опис логічної структури бази даних або елементів, які вона містить. Схеми часто включають оголошення для структури окремих записів, груп записів і окремих атрибутів, з яких складаються записи бази даних. Вони також можуть визначати типи даних і додаткові обмеження для керування типом даних, які можна додавати до структури.

Мережеві бази даних стали величезним кроком вперед у плані гнучкості та здатності відображати зв'язки між інформацією. Однак вони все ще були обмежені тими самими шаблонами доступу та мисленням проектування ієрархічних баз даних. Наприклад, щоб отримати доступ до даних, вам все одно потрібно було пройти мережевими шляхами до відповідного запису. Відносини «батько-нащадок», перенесені з ієрархічних баз даних, також впливали на те, як елементи могли з'єднуватися один з одним.

Важко знайти сучасні приклади мережевих систем баз даних. Налаштування мережевих баз даних і робота з ними вимагали значних навичок

і спеціальних знань у галузі. Більшість систем, які можна було апроксимувати за допомогою мережевих баз даних, знайшли кращий підхід після появи реляційних баз даних.

Приклади: IDMS.

Реляційні бази даних: робота з таблицями як стандартне рішення для організації добре структурованих даних. Була представлена у 1969 році. Реляційні бази даних є найстарішим типом бази даних загального призначення, який широко використовується й сьогодні. Насправді реляційні бази даних складають більшість баз даних, які зараз використовуються у виробництві.

Реляційні бази даних організовують дані за допомогою таблиць. Таблиці це структури, які накладають схему на записи, які вони містять. Кожен стовпець у таблиці має назву та тип даних. Кожен рядок представляє окремий запис або елемент даних у таблиці, який містить значення для кожного зі стовпців. Реляційні бази даних отримали свою назву від математичних зв’язків, які використовують кортежі (наприклад, рядки в таблиці) для представлення впорядкованих наборів даних.

Спеціальні поля в таблицях, які називаються зовнішніми ключами, можуть містити посилання на стовпці в інших таблицях. Це дозволяє базі даних з’єднати дві таблиці за запитом, щоб об’єднати різні типи даних.

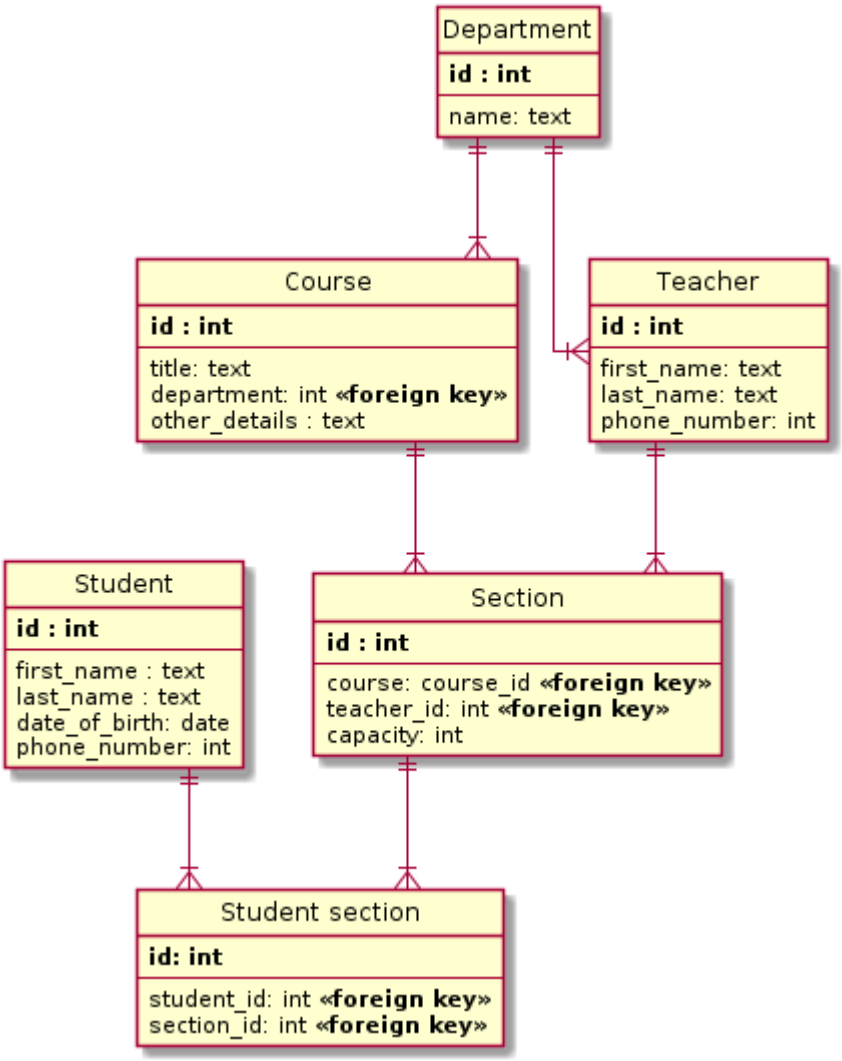

Рис. 1.3. Приклад структури реліаційної бази даних

Високоорганізована структура, яку надає жорстка структура таблиці, у поєднанні з гнучкістю, яку пропонують зв'язки між таблицями, робить реляційні бази даних дуже потужними та адаптованими до багатьох типів даних. Відповідність можна забезпечити на рівні таблиці, але операції з базою даних можуть поєднувати та маніпулювати цими даними новими способами.

Хоча мова запитів, яка називається SQL, або мова структурованих запитів, була створена для доступу до даних, що зберігаються в цьому форматі, і обробки даних, хоча це і не притаманно дизайну реляційних баз даних. Він може запитувати та об'єднувати дані з кількох таблиць в одному операторі. SQL також може фільтрувати, агрегувати, узагальнювати та обмежувати дані,

які він повертає. Отже, хоча SQL не є частиною реляційної системи, він часто є фундаментальною частиною роботи з цими базами даних.

SQL, або мова структурованих запитів, — це сімейство мов, яке використовується для запитів і обробки даних у реляційних базах даних. Він відмінно справляється з об'єднанням даних із кількох таблиць і фільтрацією на основі обмежень, що дозволяє використовувати його для вираження складних запитів. Варіанти мови були прийняті майже всіма реляційними базами даних завдяки своїй гнучкості, потужності та повсюдному поширенню.

Загалом, реляційні бази даних часто добре підходять для будь-яких даних, які є регулярними, передбачуваними та мають переваги від здатності гнучко компонувати інформацію в різних форматах. Оскільки реляційні бази даних працюють за схемою, може бути складніше змінити структуру даних після того, як вони знаходяться в системі. Однак схема також допомагає забезпечити цілісність даних, гарантуючи, що значення відповідають очікуваним форматам і включено необхідну інформацію. Загалом, реляційні бази даних є надійним вибором для багатьох програм, оскільки програми часто генерують добре впорядковані структуровані дані.

Приклади: MySQL, MariaDB, PostgreSQL, SQLite.

NoSQL бази даних: сучасні альтернативи для даних, які не відповідають реляційній парадигмі. NoSQL — це термін для різноманітної колекції сучасних типів баз даних, які пропонують підходи, що відрізняються від стандартного реляційного шаблону. Термін NoSQL є дещо неправильним, оскільки бази даних у цій категорії є скоріше реакцією на реляційний архетип, а не на мову запитів SQL. Кажуть, що NoSQL означає «не SQL» або «не тільки SQL», щоб іноді уточнити, що вони іноді дозволяють SQL-подібні запити.

Бази даних ключ-значення: простий пошук у словниковому стилі для базового зберігання та пошуку. Бази даних «ключ-значення», або сховища «ключ-значення», є одним із найпростіших типів баз даних. Сховища ключ-значення працюють, зберігаючи довільні дані, доступні через певний ключ.

Щоб зберегти дані, ви надаєте ключ і блок даних, які хочете зберегти, наприклад, об’єкт JSON, зображення або звичайний текст. Щоб отримати дані, ви надаєте ключ, а потім отримуєте блок даних назад. У більшості базових реалізацій база даних не оцінює дані, які вона зберігає, і дозволяє обмежені способи взаємодії з ними.

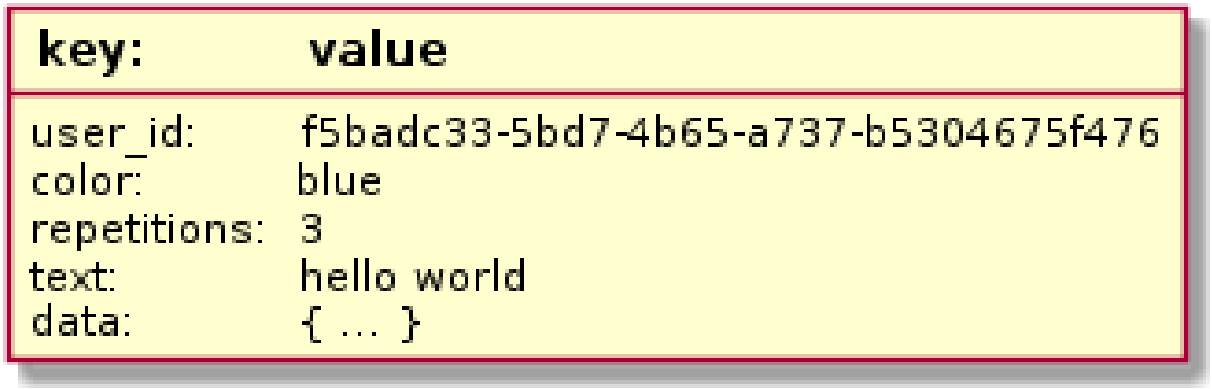

Рис. 1.4. Приклад структури бази даних ключ-значення

Якщо сховища ключ-значення здаються простими, це тому, що вони такими є. Але ця простота часто є перевагою в тих сценаріях, де вони найчастіше розгортаються. Сховища ключ-значення часто використовуються для зберігання даних конфігурації, інформації про стан і будь-яких даних, які можуть бути представлені словником або хешем у мові програмування. Сховища ключ-значення забезпечують швидкий і нескладний доступ до цього типу даних.

Деякі реалізації забезпечують більш складні дії поверх цієї основи відповідно до базового типу даних, що зберігається під кожним ключем. Наприклад, вони можуть збільшувати числові значення або виконувати зрізи чи інші операції зі списками. Оскільки багато сховищ ключ-значення завантажують усі свої набори даних у пам’ять, ці операції можна виконувати дуже ефективно.

Бази даних «ключ-значення» не призначають жодної схеми для даних, які вони зберігають, і тому часто використовуються для зберігання багатьох різних типів даних одночасно. Користувач несе відповідальність за визначення будь-якої схеми іменування для ключів, яка допоможе

ідентифікувати значення, і відповідає за те, щоб значення відповідного типу та формату. Зберігання ключів і значень є найбільш корисним як легке рішення для зберігання простих значень, якими можна керувати зовні після отримання.

Одним із найпопулярніших застосувань баз даних ключів і значень є зберігання значень конфігурації та змінних додатків і позначок для веб-сайтів і веб-додатків. Програми під час запуску можуть перевіряти конфігурацію сховища ключ-значення, що зазвичай дуже швидко. Це дозволяє вам змінювати поведінку ваших служб під час виконання, змінюючи дані в сховищі ключ-значення. Програми також можна налаштувати на періодичну повторну перевірку або перезапуск, коли вони бачать зміни. Ці конфігураційні сховища часто періодично зберігаються на диску, щоб запобігти втраті даних у разі збою системи.

Приклади: Redis, memcached.

Бази даних документів: Зберігання всіх даних елемента в гнучких структурах із самоописом**.** Набрали популярність у 2009 році.

Бази даних документів, також відомі як бази даних, орієнтовані на документи, або сховища документів, мають спільну семантику базового доступу та пошуку сховищ ключ-значення. Бази даних документів також використовують ключ для унікальної ідентифікації даних у базі даних. Насправді межа між розширеними сховищами ключ-значення та базами даних документів може бути досить нечіткою. Однак замість зберігання довільних блоків даних бази даних документів зберігають дані в структурованих форматах, які називаються документами, часто використовуючи такі формати, як JSON, BSON або XML.

Хоча дані в документах організовані в структурі, бази даних документів не передбачають жодного конкретного формату чи схеми. Кожен документ може мати різну внутрішню структуру, яку інтерпретує база даних. Отже, на

відміну від сховищ ключ-значення, вміст, що зберігається в базах даних документів, можна запитувати та аналізувати.

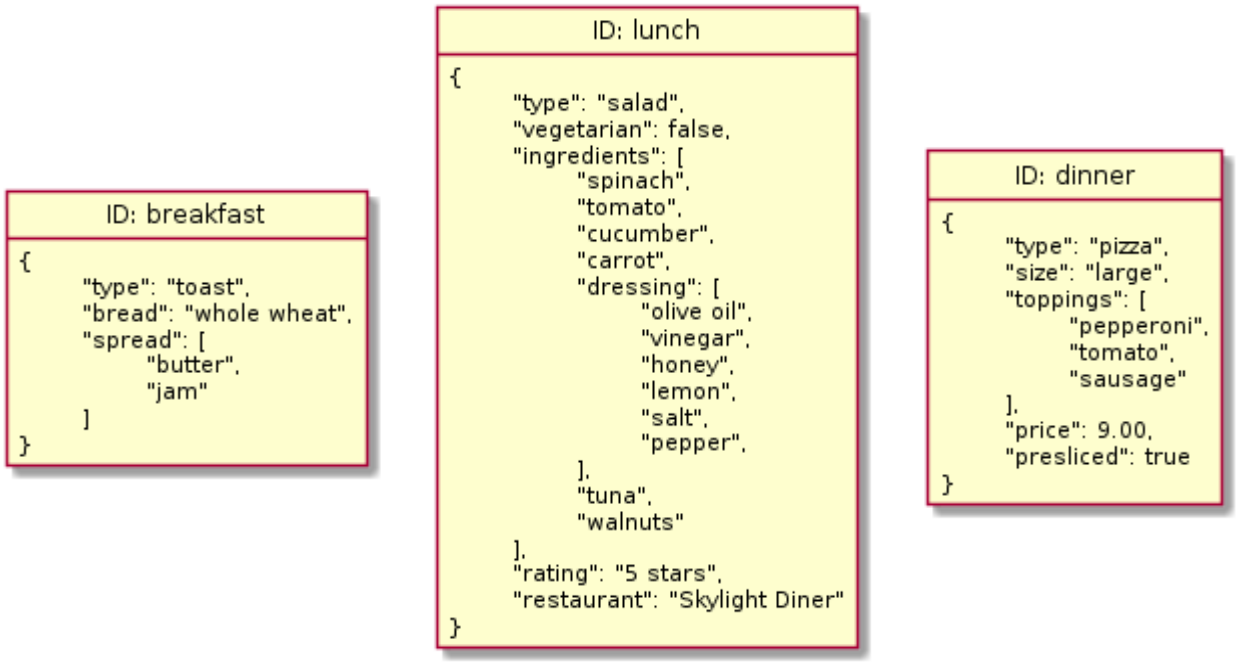

Рис. 1.5. Приклад документної бази даних

У певному сенсі бази даних документів розташовані між реляційними базами даних і сховищами ключ-значення. Вони використовують просту семантику «ключ-значення» та вільні вимоги до даних, якими відомі сховища «ключ-значення», але вони також надають можливість нав’язати структуру, яку можна використовувати для запитів і роботи з даними в майбутньому.

Однак порівняння з реляційними базами даних не слід перебільшувати. У той час як бази даних документів надають методи структурування даних в документах і роботи з наборами даних на основі цих структур, доступні гарантії, зв’язки та операції дуже відрізняються від реляційних баз даних.

Бази даних документів є гарним вибором для швидкої розробки, оскільки ви можете будь-коли змінити властивості даних, які ви хочете зберегти, не змінюючи існуючі структури чи дані. Вам потрібно лише заповнити записи, якщо ви цього хочете. Кожен документ у базі даних є самостійним із власною системою організації. Якщо ви все ще з’ясовуєте свою структуру даних і ваші

дані в основному складаються з окремих записів, які не містять багато перехресних посилань, базу даних документів може бути хорошим місцем для початку. Однак будьте обережні, оскільки додаткова гнучкість означає, що ви несете відповідальність за підтримку послідовності та структури своїх даних, що може бути надзвичайно складним завданням.

Приклади: MongoDB, RethinkDB.

Графові бази даних: відображення зв'язків, зосереджуючись на тому, наскільки зв'язки між даними мають значення. Набули популярності у 2000-х роках.

Бази даних Graph — це тип бази даних NoSQL, яка використовує інший підхід до встановлення зв'язків між даними. Замість того, щоб зіставляти зв'язки з таблицями та зовнішніми ключами, бази даних графів встановлюють зв'язки за допомогою концепцій вузлів, ребер і властивостей.

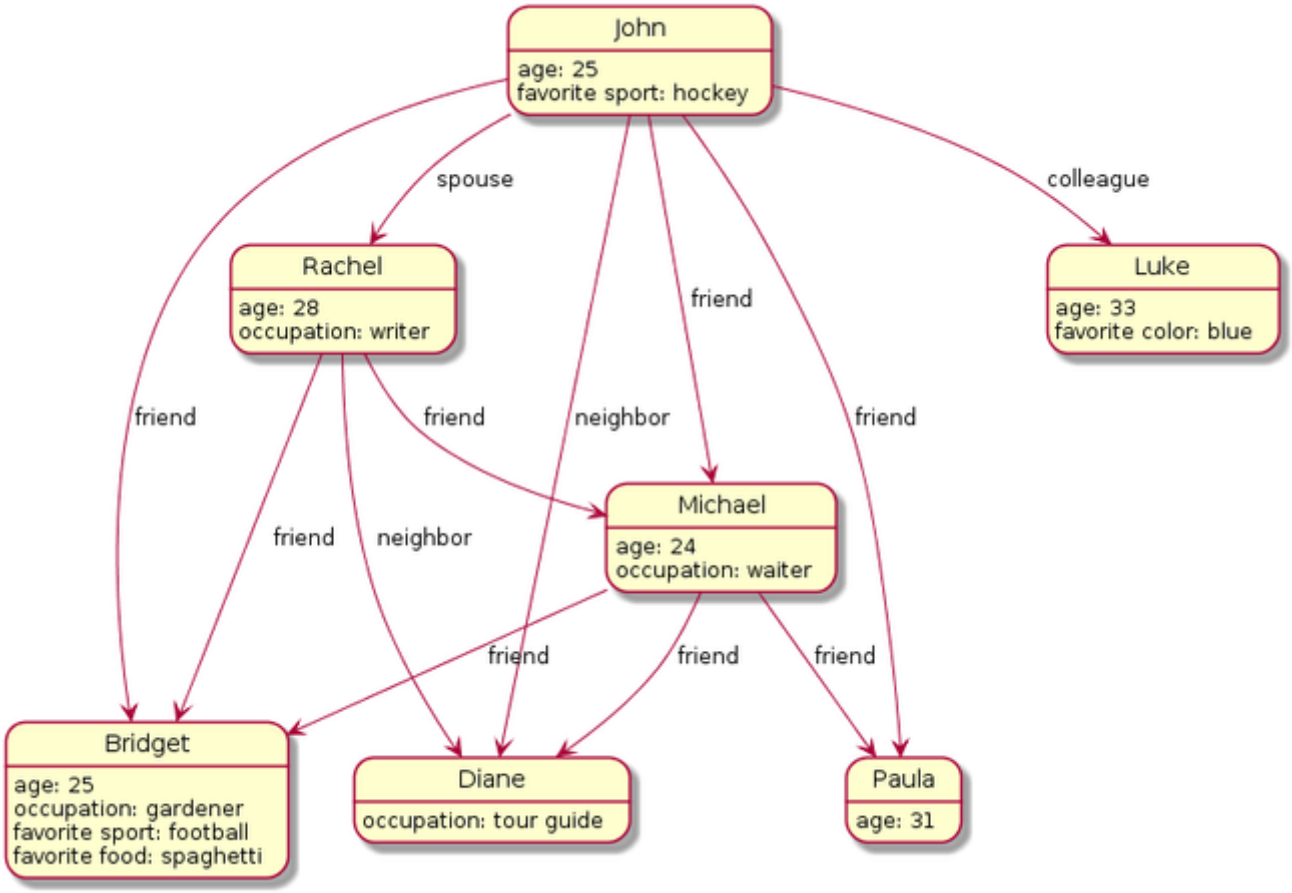

Рис. 1.6. Приклад структури графової бази даних

Графові бази даних представляють дані як окремі вузли, які можуть мати будь-яку кількість пов’язаних з ними властивостей. Між цими вузлами встановлюються ребра (також звані зв’язками), які представляють різні типи з’єднань. Таким чином база даних кодує інформацію про елементи даних у вузлах та інформацію про їхній зв’язок у краях, які з’єднують вузли.

На перший погляд бази даних графів виглядають схожими на попередні мережеві бази даних. Обидва типи зосереджені на зв’язках між елементами та дозволяють явно відображати зв’язки між різними типами даних. Проте мережеві бази даних вимагають покрокового обходу для переміщення між елементами та обмежені типами зв’язків, які вони можуть представляти.

Графові бази даних є найбільш корисними під час роботи з даними, де зв’язки чи зв’язки дуже важливі. Важливо розуміти, що, говорячи про реляційні бази даних, слово «реляційна» відноситься до здатності зв’язувати інформацію в різних таблицях. З іншого боку, основною метою баз даних графів є визначення та керування самими зв’язками.

Наприклад, запит на з’єднання між двома користувачами сайту соціальних медіа в реляційній базі даних, швидше за все, потребуватиме кількох об’єднань таблиць і, отже, буде досить ресурсоємним. Цей самий запит буде простим у графовій базі даних, яка безпосередньо відображає з’єднання. Бази даних графів спрямовані на те, щоб зробити роботу з цим типом даних інтуїтивно зрозумілою та потужною.

Приклади: Neo4j, JanusGraph, Dgraph.

Бази даних із сімейством стовпців: бази даних із гнучкими стовпцями для подолання розриву між реляційними та документними базами даних.

Бази даних із сімейством стовпців, які також називаються нереляційними сховищами стовпців, базами даних із широкими стовпцями або просто базами даних із стовпцями, є, можливо, типом NoSQL, який на перший погляд виглядає найбільш схожим на реляційні бази даних. Як і реляційні бази даних, бази даних із широкими стовпцями зберігають дані за допомогою таких понять, як рядки та стовпці. Однак у базах даних із широкими стовпцями

зв’язок між цими елементами дуже відрізняється від того, як їх використовують реляційні бази даних.

У реляційних базах даних схема визначає розташування стовпців у таблиці, вказуючи, які стовпці матиме таблиця, їхні відповідні типи даних та інші критерії. Усі рядки в таблиці повинні відповідати цій фіксованій схемі.

Замість таблиць бази даних сімейства стовпців мають структури, які називаються сімействами стовпців. Сімейства стовпців містять рядки даних, кожен з яких визначає власний формат. Рядок складається з унікального ідентифікатора рядка, який використовується для визначення місцезнаходження рядка, за яким слідують набори імен і значень стовпців.

У такому дизайні кожен рядок у сімействі стовпців визначає власну схему. Цю схему можна легко змінити, оскільки вона впливає лише на один рядок даних. Кожен рядок може мати різну кількість стовпців із різними типами даних. Іноді корисно розглядати бази даних сімейства стовпців як бази даних «ключ-значення», де кожен ключ (ідентифікатор рядка) повертає словник довільних атрибутів та їхніх значень (імен стовпців та їхніх значень).

Бази даних із сімейством стовпців добре підходять для роботи з програмами, які вимагають високої продуктивності для операцій на основі рядків і високої масштабованості. Оскільки всі дані та метадані для запису доступні за допомогою ідентифікатора одного рядка, для пошуку та отримання інформації не потрібні об’єднання, що потребують обчислень. Система бази даних також зазвичай гарантує, що всі дані в рядку розташовані на одній машині в кластері, спрощуючи шардинг даних і масштабування.

Однак бази даних родини стовпців не працюють належним чином у всіх сценаріях. Якщо у вас є високореляційні дані, які потребують об’єднань, це не правильний тип бази даних для вашої програми. Бази даних сімейства стовпців твердо орієнтовані на операції на основі рядків. Це означає, що зведені запити, як-от підсумовування, усереднення та інші процеси, орієнтовані на аналітику, можуть бути складними або неможливими. Це може мати великий вплив на те,

як ви розробляєте свої програми та які типи моделей використання ви можете використовувати.

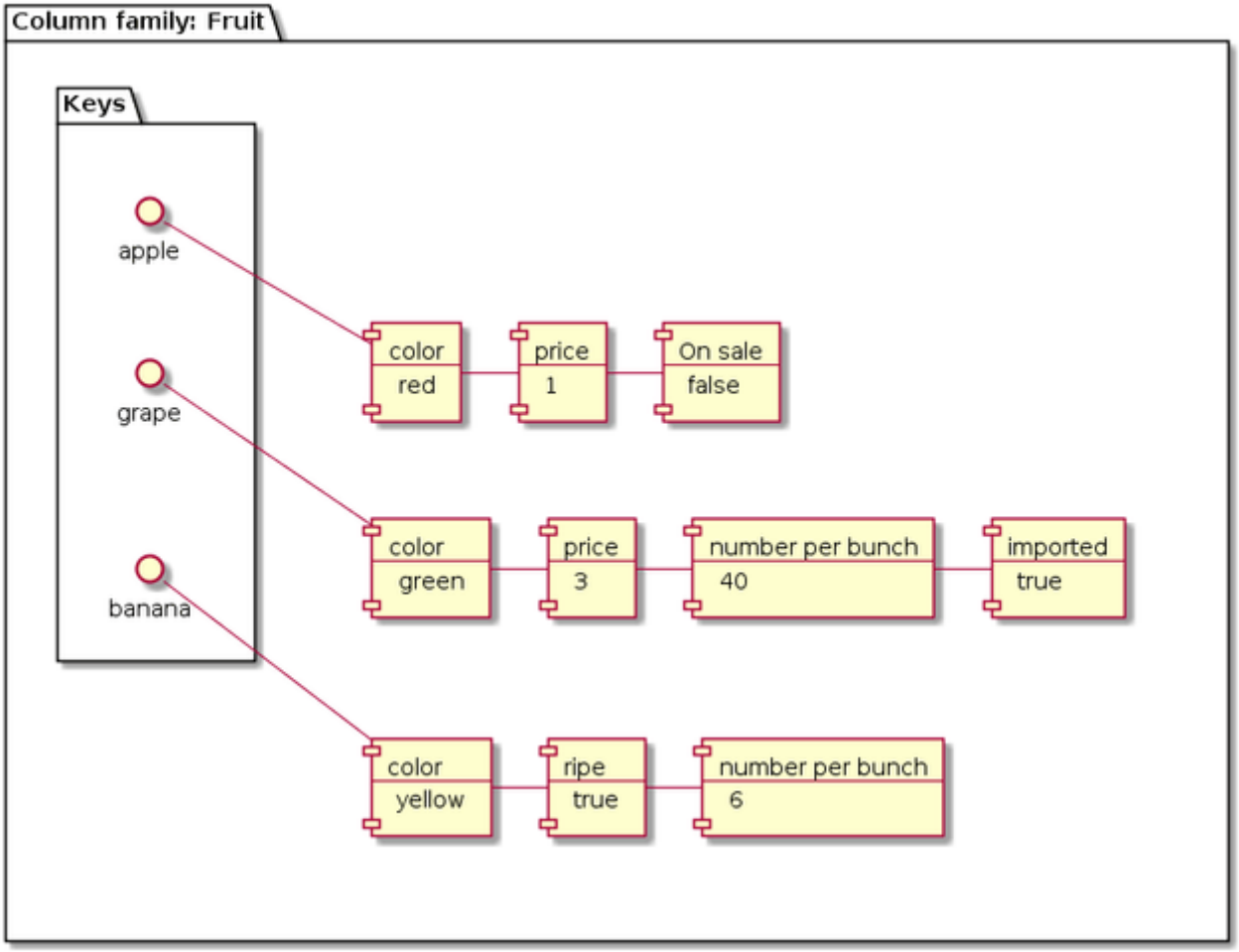

Рис. 1.7. Приклад структури стовбчикової бази даних

Приклади: Cassandra, Hbase.

Бази даних часових рядів: відстеження змін значень з часом. Бази даних часових рядів – це сховища даних, які зосереджені на зборі та управлінні значеннями, які змінюються з часом. Хоча інколи вважаються підмножиною інших типів баз даних, як-от сховища ключ-значення, бази даних часових рядів є поширеними та достатньо унікальними, щоб виправдати їх власне розглядання.

Багато баз даних часових рядів організовані в структури, які записують значення для одного елемента протягом певного часу. Наприклад, можна створити таблицю або подібну структуру для відстеження температури ЦП.

Всередині кожне значення складатиметься з позначки часу та значення температури, щоб відобразити температуру в певні моменти часу.

Time	CPU Temp
2019-10-31T03:48:05+00:00	37
2019-10-31T03:48:10+00:00	42
2019-10-31T03:48:15+00:00	33
2019-10-31T03:48:20+00:00	34
2019-10-31T03:48:25+00:00	40
2019-10-31T03:48:30+00:00	42
2019-10-31T03:48:35+00:00	41

Рис. 1.8. Приклад структури бази даних часових рядів з одним значенням

Інші реалізації використовують мітки часу як ключі для зберігання значень для кількох показників або стовпців одночасно. Наприклад, ці структури дозволять вам зберігати та отримувати значення температури процесора, завантаження системи та використання пам’яті за допомогою єдиної позначки часу.

Time	CPU Temp	System Load	Memory Usage %
2019-10-31T03:48:05+00:00	37	0.85	92
2019-10-31T03:48:10+00:00	42	0.87	90
2019-10-31T03:48:15+00:00	33	0.74	87
2019-10-31T03:48:20+00:00	34	0.72	77
2019-10-31T03:48:25+00:00	40	0.88	81
2019-10-31T03:48:30+00:00	42	0.89	82
2019-10-31T03:48:35+00:00	41	0.88	82

Рис. 1.9. Приклад структури бази даних часових рядів з баготьма значенням

З точки зору характеристик читання та запису, бази даних часових рядів сильно орієнтовані на запис. Вони призначені для обробки постійного потоку вхідних даних. Загалом бази даних часових рядів працюють із регулярними послідовними потоками даних без значних сплесків, що спрощує планування, ніж деякі інші типи даних. Продуктивність часто залежить від кількості елементів, що відстежуються, інтервалу опитування між записом нових значень і фактичного корисного навантаження даних, яке потрібно зберегти.

Бази даних часових рядів, як правило, за своєю природою лише додаються. Кожен вхідний фрагмент даних зберігається як нове значення, пов’язане з поточним моментом часу. Значення, які вже є в базі даних, зазвичай не змінюються після прийому. Оскільки найцінніші дані часто є найновішими, іноді старіші значення агрегуються, зменшуються та підсумовуються з нижчою роздільною здатністю, щоб розмір набору даних був керованим.

Бази даних часових рядів часто використовуються для зберігання інформації про моніторинг або продуктивність системи. Це робить їх ідеальним варіантом для керування інфраструктурою, особливо середовищами IoT (Інтернет речей), які генерують багато даних. Будь-яка система моніторингу або оповіщення, яку ви можете використовувати для спостереження за середовищем розгортання, швидше за все, використовуватиме певний тип бази даних часових рядів.

Приклади: OpenTSDB, Prometheus, InfluxDB, TimescaleDB.

Багатомодельні бази даних: поєднання характеристик більш ніж одного типу бази даних. Мультимодельні бази даних — це бази даних, які поєднують функціональні можливості більш ніж одного типу баз даних. Переваги цього підходу очевидні — одна й та сама система може використовувати різні представлення для різних типів даних.

Розміщення даних із кількох типів баз даних в одній системі дозволяє виконувати нові операції, які інакше були б складними або неможливими. Наприклад, багатомодельні бази даних можуть дозволяти користувачам отримувати доступ до даних, що зберігаються в різних типах баз даних, і маніпулювати ними в рамках одного запиту. Багатомодельні бази даних також допомагають підтримувати узгодженість даних, що може бути проблемою під час виконання операцій, які змінюють дані в багатьох системах одночасно.

З точки зору управління, багатомодельні бази даних допомагають полегшити робочий слід ваших систем баз даних. Наявність багатофункціональної системи дозволяє змінювати або розширювати до нових моделей у міру того, як змінюються ваші потреби, без змін базової інфраструктури чи накладних витрат на вивчення нової системи.

Важко говорити про характеристики багатомодельних баз даних як про певну категорію, оскільки вони здебільшого успадковують переваги типів баз даних, які вибирають для підтримки. Важливо мати на увазі, що ви повинні оцінити, наскільки добре окремі реалізації підтримують конкретні типи баз даних, які вам потрібні. Деякі системи можуть підтримувати кілька моделей, але з різними наборами функцій або з важливими застереженнями.

Приклади: ArangoDB, OrientDB, Couchbase.

1.3. Вибір бази даних

Після проведення детального ознайомлення з видами баз даних, мною було прийнято рішення вибрати MondgoDB.

MongoDB має декілька ключових переваг:

1. **Гнучкість Схеми:** Дозволяє зберігати документи в базі даних без попереднього визначення схеми. Це робить її ідеальною для обробки великих обсягів неструктурованих або напівструктурованих даних.

2. **Масштабованість:** Підтримує горизонтальне масштабування, що дозволяє легко розширювати базу даних за допомогою додавання додаткових серверів.

3. **Швидкість та Продуктивність:** Оптимізована для швидкого доступу до даних, особливо при роботі з великими обсягами даних.

4. **Підтримка Різноманітних Типів Даних**: Підтримує різноманітні типи даних, включаючи текст, числа, масиви та вкладені документи.

5. **Індексація для Швидкого Пошуку:** Автоматичне створення індексів для покращення продуктивності запитів.

6. **Безпека Даних**: Забезпечує механізми безпеки, включаючи аутентифікацію, авторизацію та шифрування.

1.3. Дослідження клієнтських технологій

React або «ReactJS» являє собою бібліотеку для значного спрощення побудови й маніпулювання DOM елементами. Творцем цієї бібліотеки виступає компанія Facebook. Спочатку її розробляли й використовували лише для внутрішніх потреб (тобто безпосереднього написання своєї соцмережі), й лише згодом зробили публічною, надавши до неї доступ широкому загалу програмістів.

У своєму чистому вигляді React і справді є нічим більшим, аніж звичайнісінькою бібліотекою для написання простих фронтендних додатків (при цьому, навіть не Single Page Application), оскільки не містить ані Routing, ані інших додаткових можливостей. Лише відносно нещодавно отримав офіційну можливість використовувати Context API для реалізації доволі простого state management.

Щоб користуватись React, достатньо додати посилання на бібліотеку у `index.html` сторінку. Найпростіший додаток/компонент з виводом тексту «Hello World» на React наразі виглядає наступним чином:

```
const HelloWorldComponent = () => {
  const message = "Hello world";

  return <div>{message}</div>;
};


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(AppComponent);
```

Для того, аби це запрацювало, також необхідно додати один елемент , щоб нашому застосунку було до чого підв’язатись. Він слугує кореневим елементом для зображення усього нашого застосунку.

По суті всі компоненти у React є функціями, які приймають аргументи й повертають розмітку. Також, варто відзначити, що все, що є нижче за

визначення функції AppComponent буде вказуватись лише раз — виключно у файлі «точці входу» застосунку.

Також тут є ще один цікавий момент. Якщо придивитись, функція повертає щось схоже на HTML. React використовує особливий синтаксис — JSX (Javascript Extended). JSX є синтаксичним цукром, котрий приховує набір методів і функцій, покликаних будувати репрезентацію DOM в Javascript (Virtual DOM), під виглядом максимально подібним на побудову HTML DOM дерев. Також JSX дозволяє «змішувати» HTML і Javascript. До прикладу, ось так може виглядати промальовування списку елементів:

```
<ul>
 { unorderedListItems.map((item) => <li key={ item.name }>{ item.name }</li>) }
</ul>
```

Тобто ми просто беремо і «промальовуємо» кожен елемент до <li> з item.name, як ключ і контент цього елемента.

Плюси:

1. Синтаксис є максимально зрозумілим, якщо вам знайомі Javascript і HTML.
2. Крива входу у користування React є максимально пологою.
3. Функціональний підхід до написання компонентів.
4. Основним з найважливіших підходів, які пропагує React, на мою думку, є принцип Immutability. Тобто все, що робиться в межах компонентів, в жодному випадку не має мутувати стану або даних, а лишень створювати нові екземпляри на основі вже наявних.
5. Компоненто-центристський підхід до написання застосунку (тут все є компонентами).
6. Вага бібліотеки — 2.5Kb мінімізована Gzip версія.
7. Підхід до створення екосистеми застосунку — від меншого до більшого. Тобто тут лише ви вирішуєте, що потрібно брати з додаткових бібліотек і пакетів, а що — ні.

8. Стан DOMу зберігається й обчислюється у віртуальному DOMі, що робить React дуже швидким в плані рендерінгу.

9. Юніт тестування є максимально простим, оскільки не потрібно налаштовувати жодного «додаткового» оточення і браузерів для відрендерювання компонентів.

10. Величезна кількість різноманітної інформації про помилки і основні проблеми.

11. Окрім великої кількості бібліотек, призначених спеціально для React, до нього також можна «докрутити» будь-яку Javascript-бібліотеку, було б бажання.

12. Відмінна документація, зокрема доступна й українською мовою, з хорошими прикладами й достойним описом.

13. Велике ком'юніті.

14. Велика кількість різноманітного тулінгу для полегшення процесу роботи й дебагу.

15. Широкий вибір бібліотек під будь-яку задачу і смак.

16. Повноцінна підтримка Typescript при використанні React Create App або ж інших React CLI бібліотек.

17. Регулярно оновлюється і вдосконалюється.

18. Підтримується Facebook.

Мінуси:

1. Оскільки «з коробки» React не включає практично жодних додаткових можливостей, окрім роботи з DOM, необхідно знати або ж вміти добре гуглити потрібні й надійні бібліотеки для тих чи інших завдань.

2. В простоті React криється й головний його недолік — тут можна запросто напартачити, не знаючи best practices передачі пропсів (у Angular — це Inputs і Outputs), декомпонентизації, структурування файлів React додатків, масштабування застосунку в цілому і решти нюансів. Що, з часом (й

збільшенням кількості коду), може призвести до вкрай складної структури застосунку для її підтримання й подальшого розширення.

Angular

На відміну від React, Angular, що є продуктом компанії Google, вже є не просто бібліотекою для роботи з DOM, а цілим повноцінним фреймворком, котрий керується принципами MVVM побудови застосунків, й «з коробки» містить зокрема такі можливості, як:

— CLI для генерування структури файлів різних Angular-конструкцій, структури npm бібліотек, тестування, лінтування і т.д.

— Ряд бібліотек для самих різноманітних завдань — router, бібліотеки для роботи з формами, анімацією й інші.

— При ініціалізації проєкту можна доволі тонко налаштовувати те, що з доступних бібліотек буде додано до нього.

— Засоби лінтування.

— Karma + Jasmine для Unit і End to End тестування.

Історично, Angular хоча й виник трішки пізніше за React, фактично є переписаною з «нуля» версією фреймворку AngularJS, котрий існував раніше за нього. На початку серед фронтендщиків навіть виникали деякі непорозуміння, оскільки ніхто не розумів різниці між назвами Angular і AngularJS. Як наслідок, команда розробки фреймворку почала використовувати назви Angular першої версії (для AngularJS) і Angular другої версії (для Angular), аби чіткіше їх розрізняти. Якщо порівняти Angular з React, то його структура є значно складнішою. Окрім компонентів, тут також з’являються наступні елементи, кожен зі своїм призначенням:

— Пайпи: свого роду допоміжні функції для трансформації візуального вигляду даних у HTML темплейтах. Наприклад: форматування чисел, дат, валют тощо.

— Директиви: класи, котрі використовуються для надання додаткового функціонала іншим елементам застосунку. І також поділяються на кілька видів, в залежності від форми взаємодії з елементами (цей момент я пропущу, аби не вдаватись в подробиці). В основному застосовуються у вигляді користувацьких атрибутів елементів розмітки.

— Компоненти: є елементом для візуального зображення даних. Поєднують в собі стилі, HTML-файл з розміткою, а також JS або TS з логікою компонента.

— Сервіси: для можливості централізованого зберігання даних і утилітарних методів. Можуть використовуватись Пайпами, Директивами, Компонентами й навіть іншими Сервісами, додаючи їх за допомогою Dependency Injection.

— Модулі: спосіб згрупувати всі вищеперелічені елементи. В більшості випадків, по функціоналу або ж сторінках.

Аби трішки візуально розбавити цей потік інформації, далі буде приклад того ж простого «Hello World» застосунку, що й у секції про React, але реалізованого засобами Angular. Варто відзначити, що тут простим додаванням посилання на бібліотеку в index.html не обійдеться. Прикладу передували ініціалізація структури проєкту за допомогою Angular CLI. Структура вже має в собі всі необхідні файли (зокрема кореневого компонента AppComponent, який я побудував як простий приклад) з підв'язуванням застосунку до DOM, а також налаштуваннями лінтерів, тестів, Typescript і тому подібного. Також для простоти розмітку було написано в межах властивості template, на противагу винесенню її в окремий html файл з подальшим вказанням шляху до нього у властивості templateUrl.

src/app/app.component.ts

```ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<h1>{{ message }}</h1>`
})
export class AppComponent {
  public message = 'message';
```

src/app/app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

На відміну від React JSX, який дозволяє вставляти по суті будь-який валідний JS код в розмітку напряму, Angular здійснює маніпуляції з DOM за допомогою вбудованих директив. Для прикладу, ось як виглядатиме промальовування списку елементів в HTML шаблоні Angular:

```
<ul>
 <li *ngFor="let item of unorderedListItems">{{ item.name }}</li>
</ul>
```

Тобто ми створюємо кілька <li> елементів використовуючи директиву «ngFor», що дозволяє отримати елемент, на який ми його навісили *unorderedListItems.length* кількість разів. Аналогічним чином досягається умовне зображення або ж приховування DOM елементів й інші подібні операції.

Плюси:

1. Можливість чітко групувати елементи, використовуючи модулі.
2. Є повноцінним фреймворком, що має все необхідне для написання сучасних вебзастосунків і навіть більше.
3. Підтримка Typescript «з коробки». По суті фреймворк є побудованим з прицілом на підтримку Typescript, як однієї з головних ідей.
4. Використання MVVM принципів побудови застосунків, що дає змогу чітко розмежовувати реалізацію бізнес-логіки від її представлення.

5. Завдяки наявності модулів — чудово масштабується, якщо попередньо чітко планувати структуру застосунку й слідувати їй.

6. Dependency Injection — чудовий спосіб передачі лише необхідних залежностей (у вигляді сервісів), у компоненти й інші елементи Angular. Особливо розквітає, коли починаєте писати всілякого роду тести для ваших компонентів, бо дозволяє легко підмінити будь-яку з цих залежностей необхідною вам імплементацією.

7. Документація актуальна і вичерпна, в плані доступних класів, методів і типів. Також містить масу додаткової інформації, зокрема такої, як:

8. Angular coding style guide — офіційний гайд по рекомендованому синтаксису, конвенціях, структурі застосунку.

9. Best Practices, як рекомендації з написання Безпечних і Доступних застосунків, а також підходів для розбиття коду на множину окремих бандлів Lazy-loading feature modules.

10. І багато чого ще, з повним переліком доступної інформації можна ознайомитись тут.

11. Кількість інформації, доступної в інтернеті щодо будь-якого аспекту роботи, чи певної бібліотеки.

12. Велике ком’юніті.

13. Широкий вибір бібліотек під будь-яку задачу і смак.

14. Регулярно оновлюється і вдосконалюється.

15. Дуже просунуте CLI, котре при цьому також піддається деякій кастомізації (для прикладу Blueprints, для кастомізації генерації компонентів та інших елементів і файлів).

16. Підтримується Google.

Мінуси:

1. Оскільки структура передбачає доволі великий обсяг знань, яким необхідно володіти аби почати писати на Angular, крива входу є доволі

крутою. Як наслідок, мінусом є висока складність, порівняно з іншими фреймворками й бібліотеками.

2. В документації подекуди катастрофічно бракує прикладів використання того чи іншого функціоналу, оскільки знання його сигнатури буває недостатньо.

3. Офіційна документація доступна лише сімома мовами. Українська в їх переліку відсутня.

4. Інколи низька швидкість роботи, яку дуже просто «зіпсувати». Але при цьому, якщо знати про те, як працює Angular Change Detection і як в цілому оптимізувати швидкість роботи й завантаження вебзастосунків, стає більш ніж конкурентною.

5. Також мінусом, на мою думку, є пряме мутування властивостей класу для зміни або оновлення стану компонента.

Vue

Наймолодший з цієї трійки бібліотек/ фреймворків і замикальний — Vue (читається як View). Я навмисне взяв для порівняння саме версію 2.6.x, оскільки це — основна версія, котру зараз використовують на проєктах. Також більшість нових проєктів, з якими я стикався, створювалась саме на ній, а не Vue 3 (попри те, що його реліз вже місяць чи два як відбувся, станом на той час). Аргументувалось це тим, що третя версія була вкрай сирою й за відгуками дуже нестабільною. У зв’язку з цим, оновлення виходили доволі часто й нерідко містили Breaking Changes. Vue є свого роду сумішшю Angular і React. З одного боку, у своєму чистому вигляді — це бібліотека для роботи з DOM, та з іншого боку — це фреймворк, в якого навіть є свій офіційний Vue CLI й певна екосистема бібліотек навколо нього. Vue широко використовує й наполягає на підході Single File Component, щодо написання компонентів, як основному. Його суть полягає в тому, що стилі, розмітка і логіка компонента мають знаходитись в одному файлі. Хоча вона й надає можливість і засоби для розбиття компонента на кілька різних файлів.

Плюси

1. Доволі висока швидкість збірки й роботи застосунку в цілому.
2. Можливість писати додатки, як з білд бібліотеками, так і без них.
3. Кількість тулів і бібліотек доволі обмежена, але мінімально допустима. Тому цей пункт в «плюсах», хоча й недалеко від «мінусів».
4. Малий розмір бандлів після білда.
5. Складність: щось середнє між Angular і React, але все ж значно ближче до React. Тому вивчити й розпочати щось писати на Vue мало б бути доволі просто.

Мінуси:

1. Бібліотека підтримується зусиллями команди творця бібліотеки й більшою мірою — ком'юніті. Фінансується бібліотека лише з «пожертв».
2. Наразі кожен Major реліз Vue супроводжувався змінами, несумісними з попередніми версіями.
3. Доволі мінімалістична офіційна документація з обмеженою кількістю доступних мов, хоча й більшою ніж в Angular.
4. У версії 2.6 я зіткнувся з проблемою неможливості зрозуміти деякі помилки, що виникали при розробці, або бодай нагуглити проблеми, котрі виникали по ходу розробки (а виникали вони доволі часто). Тому зазвичай це суттєво підвищувало час пошуку проблем. Тобто інколи консольні помилки не допомагали, а навпаки заважали, оскільки не відповідали причині.
5. Підтримка Typescript у версії 2.6 практично відсутня. Хоча й можлива, але з надзвичайно замороченими способами його додавання, не сумісними з нормальною комфортною розробкою. Підтримка Typescript у версії 3.x стала кращою, але за відгуками досі не дуже позитивно сприймається Vue ком'юніті (принаймні судячи з тих людей, з котрими я мав справу, й статей, які читав на тему). Хоча, можливо, «не дуже позитивна підтримка» більше стосується самого Typescript, в плані «навіщо з ним, якщо можна без нього», аніж зручністю його використання з Vue 3.

ВИСНОВКИ ДО РОЗДІЛУ 1

Було проведений детальний аналіз усіх сучасних типів баз даних, знайдено їх недоліки та обрано найбільш придатну для даного додатку базу даних. Так як різкий стрибок популярності NoSQL баз даних і пов'язані з ним історії використання нереляційних СУБД показали світу IT важливість реалістичної оцінки пріоритетів компанії. Деякі вендори успішно впровадили у себе NoSQL сховища і отримали помітне зниження збитків і підвищення якості їх додатків. Інші зазнали невдачі, пізно зрозумівши, що прийняте рішення їм не підходить. А треті просто залишилися зі своїми технологіями. Реляційні або нереляційні бази даних не єдиний вибір, який належить зробити компанії. Не менш важливим є і вибір між конкретними системами і конкретними стратегіями роботи з ними. У будь-якому випадку, NoSQL-революції не відбулося — реляційні бази даних утримують стабільно домінуючі позиції. Вони являють собою симбіоз надійності, функціональності і універсальності. При цьому багато NoSQL баз даних спрямовані на закриття абсолютно конкретних проблем SQL сховищ — у першу чергу на посилення горизонтальної масштабованості. Багато нереляційних баз даних відмінно працюють, виконуючи мету свого створення, але при цьому вони вже не є тим універсальним продуктом, яким є SQL.

Для клієнтської частини було обрано React, так як це комфортний, добре підтримуваний, простий в розумінні та написанні, й надзвичайно потужний, якщо старатись писати на ньому «правильно». Тобто, використовуючи класні тулзи, обов'язково Typescript, більш-менш «стандартизовані» й класно задокументовані бібліотеки (якщо це можна так назвати), й хороші підходи до структурування, по типу «Atomic Design by Bred Frost» розбавляючи це все юніт/e2e тестами (й обов'язково TDD, оскільки писати його з React — одне задоволення).

РОЗДІЛ 2
ПРОЕКТУВАННЯ ЗАСТОСУНКУ

2.1. Опис вимог до застосунку

Застосунок інтернет магазин з нереліаційною базою данних повиними відповідати наступним вимогам.

Функціональні Вимоги:

1. Інтерфейс користувача: чіткий та інтуїтивний.
2. Основні функції: визначені згідно з цілями застосунку.
3. Взаємодія з іншими системами: інтеграція API, баз даних тощо.
4. Обробка даних: введення, зберігання, виведення.

Нефункціональні Вимоги:

1. Продуктивність: час відгуку, обробки запитів.
2. Надійність: здатність працювати безперебійно.
3. Сумісність: працює на різних платформах/пристроях.
4. Безпека: захист даних, аутентифікація, авторизація.

Вимоги до Інтерфейсу:

1. Дизайн: привабливий та зрозумілий.
2. Адаптивність: коректно відображається на різних екранах.

Вимоги до Тестування:

1. Охоплення всіх сценаріїв використання.
2. Перевірка на відповідність вимогам.

2.2. Вибір шаблону проектування

У сфері розробки програмного забезпечення, професіонали часто використовують архітектурні шаблони для підвищення модульності та забезпечення ефективного модульного тестування коду. Цей підхід полегшує утримання та оновлення програм, а також дозволяє легше розширювати їх функціонал. Найбільш відомими та широко використовуваними в галузі є шаблони MVC (Model-View-Controller), MVP (Model-View-Presenter) і MVVM (Model-View-ViewModel), які користуються популярністю серед розробників.

Шаблон «Модель—Вид—Контролер» (MVC)

Архітектурний шаблон MVC (рис 2.1) ділить програму на три рівні: Модель, Перегляд і Контролер. Модель - це ядро програми, де зберігаються дані та обробляється бізнес-логіка, без зв'язку з інтерфейсом користувача. Вона взаємодіє з базою даних та мережею. Перегляд стосується інтерфейсу користувача, відображає дані з моделі та дозволяє користувачам взаємодіяти з програмою. Контролер зв'язує Модель і Перегляд, керуючи потоком інформації між ними та оновлюючи Модель згідно з діями користувача.

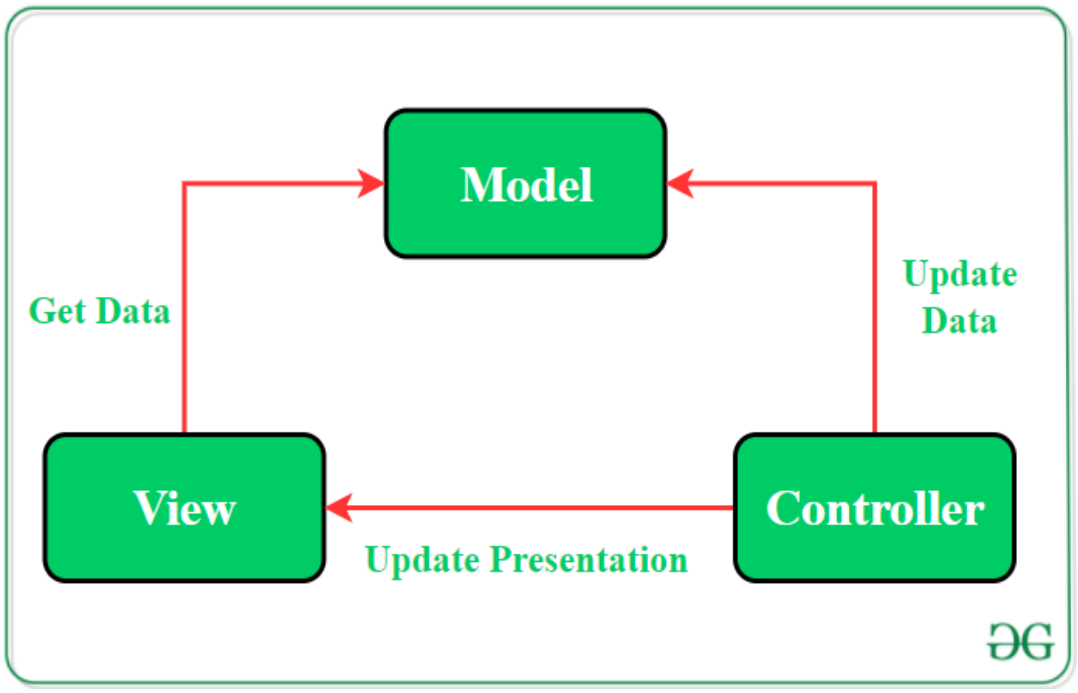

Рис. 2.1. Опис шаблону «Модель—Вид—Контролер»

Шаблон «Модель—Перегляд—Презентатор» (MVP)

Шаблон MVP (Model-View-Presenter) вирішує деякі складнощі MVC (рис. 2.2), пропонуючи спрощений метод організації коду в проектах. Він здобув популярність через свою модульність, тестованість та чистоту коду, полегшуючи підтримку. MVP включає Модель (зберігання даних і обробка бізнес-логіки), Перегляд (інтерфейс користувача і відображення даних), та Презентатор (обробляє логіку інтерфейсу та керує станом Перегляду).

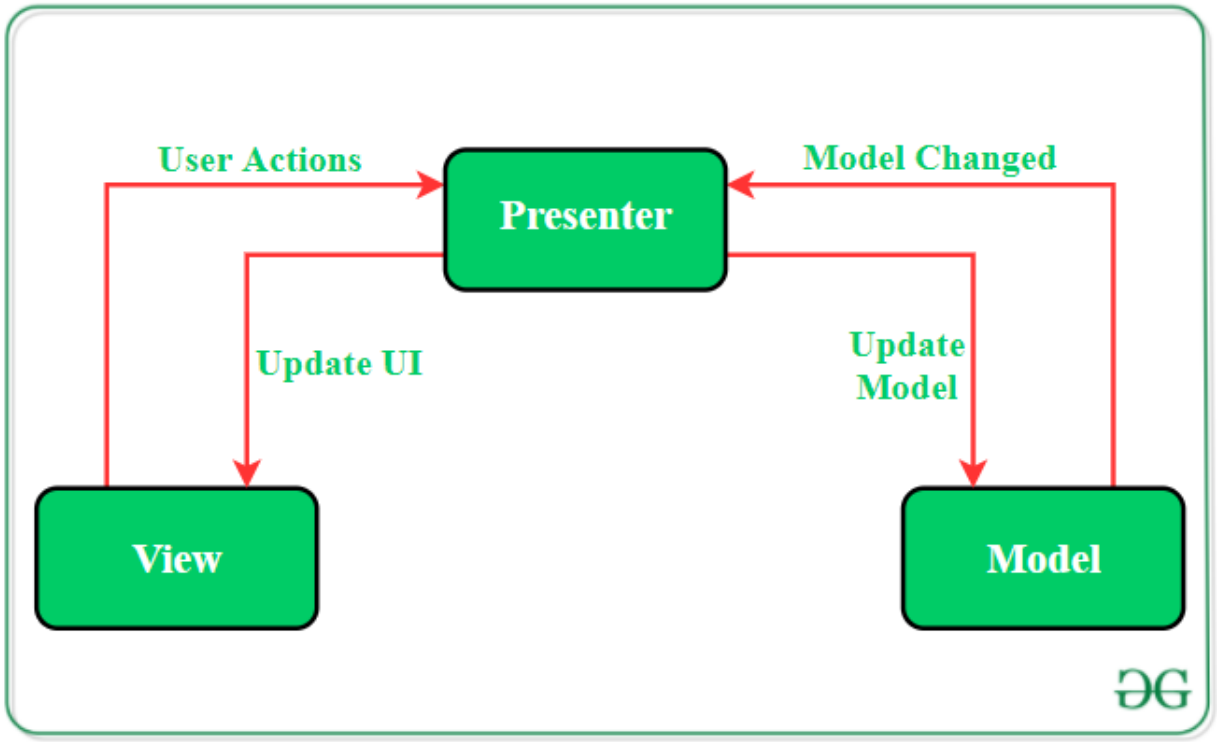

Рис. 2.2. Опис шаблону «Модель—Перегляд—Презентатор»

Шаблон «Модель — Вигляд — Модель перегляду» (MVVM)

Шаблон MVVM (Model-View-ViewModel) вдосконалює ідеї MVP (рис 2.3), перекладаючи роль Presenter на ViewModel. Він розділяє логіку відображення даних (Views або UI) від бізнес-логіки, забезпечуючи чіткіше відокремлення. MVVM включає: Модель (управління даними), Перегляд (інтерфейс, що реагує на користувацькі дії та спостерігає за ViewModel), і

ViewModel (посередник між Моделлю та Переглядом, відображає потрібні дані).

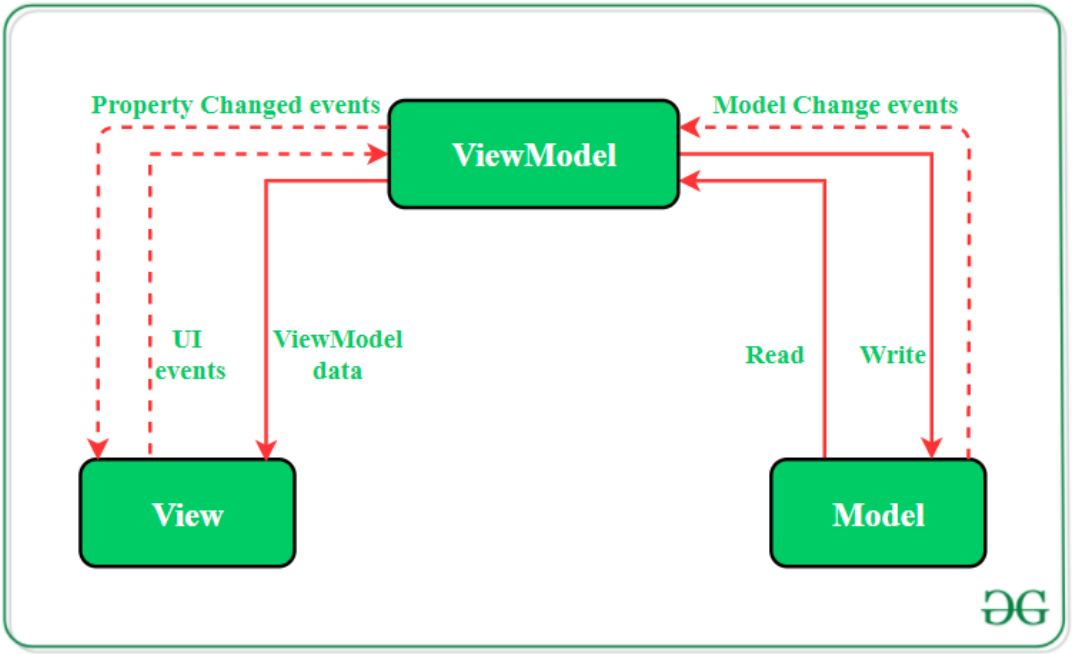

Рис. 2.3. Опис шаблону «Модель—Вигляд—Модель перегляду»

2.3. Вибір та використання серидовища розробки

Visual Studio Code (VS Code) - це безкоштовний, потужний та легкий у використанні редактор коду, розроблений Microsoft. Він підтримує багато мов програмування та має велику кількість розширень, що дозволяє налаштувати середовище розробки під конкретні потреби.

VS Code пропонує такі функції, як відладка, вбудована підтримка Git, інтелектуальне автодоповнення коду, рефакторинг та візуалізація коду. Цей редактор є популярним вибором серед розробників завдяки своїй гнучкості, продуктивності та простоті у використанні.

Рис. 2.4. Іконка застосунку VsCode

Основні особливості

Visual Studio Code — це редактор вихідного коду, який можна використовувати з різними мовами програмування, включаючи C, C#, C++, Fortran, Go, Java, JavaScript, Node.js, Python, Rust. Він базується на структурі Electron, яка використовується для розробки веб-додатків Node.js, які працюють на механізмі компонування Blink. Visual Studio Code використовує той самий компонент редактора (під кодовою назвою «Monaco»), який використовується в Azure DevOps (раніше називався Visual Studio Online і Visual Studio Team Services).

З коробки Visual Studio Code містить базову підтримку для більшості поширених мов програмування. Ця базова підтримка включає підсвічування синтаксису, зіставлення дужок, згортання коду та настроювані фрагменти. Visual Studio Code також постачається з IntelliSense для JavaScript, TypeScript, JSON, CSS і HTML, а також підтримує налагодження Node.js. Підтримка додаткових мов може бути забезпечена безкоштовними розширеннями на VS Code Marketplace.

Замість системи проектів вона дозволяє користувачам відкривати один або кілька каталогів, які потім можна зберегти в робочих областях для

подальшого використання. Це дозволяє йому працювати як мовно-агностичний редактор коду для будь-якої мови. Він підтримує багато мов програмування та набір функцій, який відрізняється для кожної мови. Небажані файли та папки можна виключити з дерева проекту за допомогою налаштувань. Багато функцій Visual Studio Code не доступні через меню чи інтерфейс користувача, але доступ до них можна отримати через палітру команд.

Код Visual Studio можна розширити за допомогою розширень, доступних через центральне сховище. Це включає доповнення до редактора та підтримку мови. Примітною функцією є можливість створювати розширення, які додають підтримку нових мов, тем, налагоджувачів, налагоджувачів подорожей у часі, виконують статичний аналіз коду та додають лінтери коду за допомогою протоколу Language Server.

Керування джерелом є вбудованою функцією Visual Studio Code. Він має спеціальну вкладку всередині панелі меню, де користувачі можуть отримати доступ до налаштувань керування версіями та переглянути зміни, внесені до поточного проекту. Щоб використовувати цю функцію, Visual Studio Code має бути зв’язано з будь-якою підтримуваною системою керування версіями (Git, Apache Subversion, Perforce тощо). Це дозволяє користувачам створювати репозиторії, а також робити запити push і pull безпосередньо з програми Visual Studio Code.

Visual Studio Code містить кілька розширень для FTP, що дозволяє використовувати програмне забезпечення як безкоштовну альтернативу для веб-розробки. Код можна синхронізувати між редактором і сервером без завантаження додаткового програмного забезпечення.

Код Visual Studio дозволяє користувачам встановлювати кодову сторінку, на якій буде збережено активний документ, символ нового рядка та мову програмування активного документа. Це дозволяє використовувати його

на будь-якій платформі, у будь-якій локальній мережі та для будь-якої заданої мови програмування.

Visual Studio Code збирає дані про використання та надсилає їх до Microsoft, хоча це можна вимкнути. Через відкритий вихідний код програми, код телеметрії доступний для громадськості, яка може бачити, що саме збирається. В опитуванні розробників Stack Overflow у 2016 році Visual Studio Code посідає 13 місце серед найпопулярніших інструментів розробки, лише 7% із 47 000 респондентів використовують його. Два роки по тому, однак, Visual Studio Code посів перше місце: 35% із 75 000 респондентів використовували його.

В опитуванні розробників у 2019 році код Visual Studio посів перше місце: 50% із 87 000 респондентів використовували його. В опитуванні розробників 2021 року Visual Studio Code продовжував займати перше місце з 74,5% із 71 000 респондентів, піднявшись до 74,48% із 71 010 відповідей в опитуванні 2022 року.

Visual Studio Code було вперше анонсовано 29 квітня 2015 року Microsoft на конференції Build 2015. Попередня збірка була випущена незабаром після цього. 18 листопада 2015 року вихідний код Visual Studio Code було випущено за ліцензією MIT і доступно на GitHub. Також було оголошено про підтримку розширення. 14 квітня 2016 року код Visual Studio закінчив етап загальнодоступної попередньої версії та був випущений в Інтернет. Корпорація Майкрософт опублікувала більшість вихідного коду Visual Studio Code на GitHub під дозвільною ліцензією MIT, тоді як випуски Microsoft є пропрієтарними безкоштовними програмами.

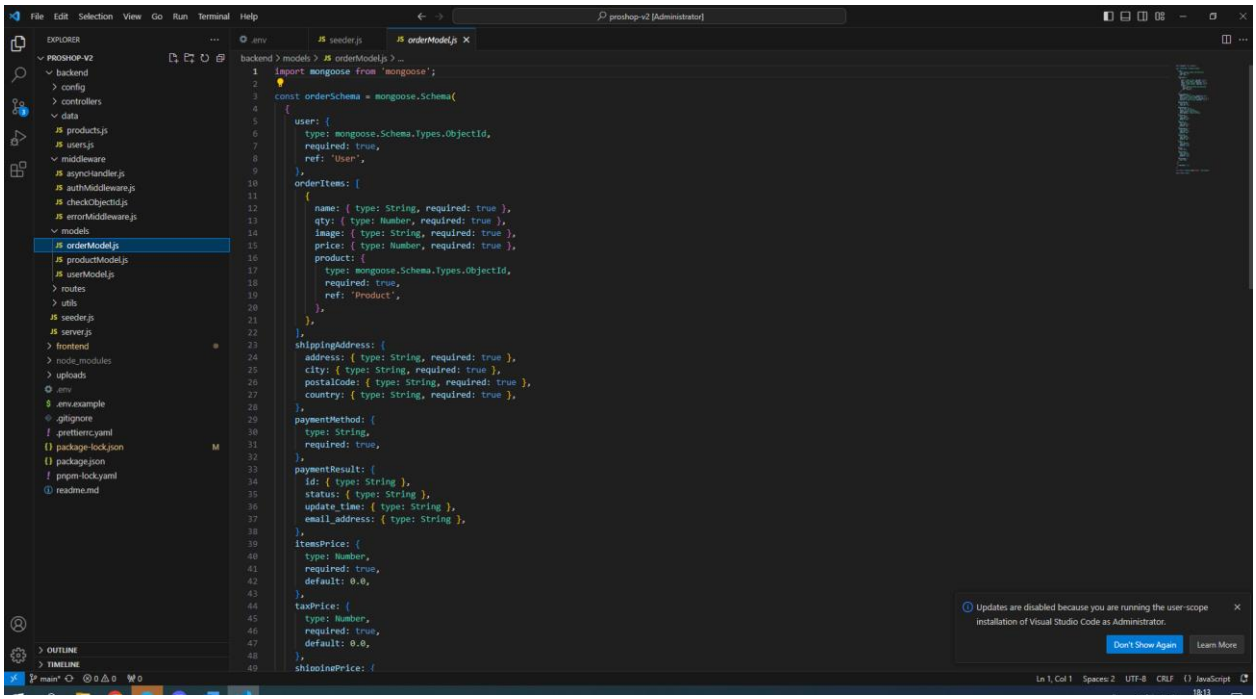

Рис. 2.5. Вид VsCode на прикладі коду мого застосунку

2.4. Використання JWT для аутентифікації

JSON Web Token (JWT, МФА: [dʒɒt], вимовляється "джот") це стандарт токена доступу на основі JSON, стандартизованого в RFC 7519. Як правило, використовується для передачі даних для аутентифікації в клієнт-серверних програмах. Токени створюються сервером, підписуються секретним ключем і передаються клієнту, який надалі використовує цей токен для підтвердження своєї особи.

Токен JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису або даних шифрування. Перші два елементи – це JSON об'єкти певної структури. Третій елемент обчислюється на основі перших і залежить від обраного алгоритму (у разі використання непідписаного JWT може бути опущений).

Заголовок

Заголовок (Header) це JSON елемент, що описує до якого типу належить даний токен і які методи шифрування використовувались.

Заголовок може наприклад виглядати так:

{

"alg": "HS256",

"typ": "JWT"

}

Вміст (англ. Payload) складається з елемента JSON який описує твердження.

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

Підпис

Структура підпису визначається стандартом JSON Web Signature (JWS, RFC 7515).

Щоб згенерувати підпис, заголовок та вміст кодуються в Base64, записуються в один рядок через крапку, а потім цей рядок хешується визначеним методом:

```
var encodedString = base64UrlEncode(header) + "." +
base64UrlEncode(payload);
var hash = HMACSHA256(encodedString, secret);
```

Кодування

Заголовок, вміст і підпис кожен кодуються в Base64 і розділяються в токені крапкою. JWT може виглядати так:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJzY290Y2guaW8iLCJleHAiOjEzMDA4MTkzdHJ1ZX0.03f329983b86f7d9a9f5fef85305880101d5e302afafa20154d094b229f75773

2.5. Серверний JavaScript (NodeJS)

Node.js — платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. Засновником платформи є Раян Дал (Ryan Dahl). Якщо раніше JavaScript застосовувався для обробки даних в браузері користувача, то node.js надав можливість виконувати JavaScript-скрипти на сервері та відправляти користувачеві результат їхнього виконання. Платформа Node.js перетворила JavaScript на мову загального використання з великою спільнотою розробників.

Node.js має наступні властивості:

— асинхронна однониткова модель виконання запитів;

— неблокуючий ввід/вивід;

— система модулів CommonJS;

— рушій JavaScript Google V8;

Для керування модулями використовується пакетний менеджер npm (node package manager).

Платформа Node.js призначена для виконання високопродуктивних мережевих застосунків, написаних мовою програмування JavaScript. Платформа окрім роботи із серверними скриптами для веб-запитів, також використовується для створення клієнтських та серверних програм.

В платформі використовується розроблений компанією Google рушій V8.

Для забезпечення обробки великої кількості паралельних запитів у Node.js використовується асинхронна модель запуску коду, заснована на обробці подій в неблокуючому режимі та визначенні обробників зворотніх викликів (callback). Як способи мультиплексування з'єднань підтримується epoll, kqueue, /dev/poll і select. Для мультиплексування з'єднань використовується бібліотека libuv, для створення пулу нитей (thread pool) задіяна бібліотека libeio, для виконання DNS-запитів у неблокуючому режимі

інтегрований c-ares. Всі системні виклики, що спричиняють блокування, виконуються всередині пулу потоків і потім, як і обробники сигналів, передають результат своєї роботи назад через неіменовані канали (pipe).

За своєю суттю Node.js схожий на фреймворки Perl AnyEvent, Ruby Event Machine і Python Twisted, але цикл обробки подій (event loop) у Node.js прихований від розробника і нагадує обробку подій у веб застосунку, що працює в браузері. При написанні програм для Node.js необхідно враховувати специфіку подієво-орієнтованого програмування, наприклад, замість виконання.

Для розширення функціональності застосунків на базі Node.js підготовлена велика колекція модулів, в якій можна знайти реалізацію HTTP, SMTP, XMPP, DNS, FTP, IMAP, POP3 серверів і клієнтів, модулі для інтеграції з різними вебфреймворками, обробники WebSocket і AJAX, конектори до СКБД (MySQL, PostgreSQL, SQLite, MongoDB), шаблонізатори, CSS-рушії, реалізації криптоалгоритмів і систем авторизації (наприклад, OAuth), XML-парсери.

Mongoose

Mongoose - це ODM (* Object Document Mapper - об'єктно-документний відображувач). Це означає, що Mongoose дозволяє вам визначати об'єкти зі строго-типізованою схемою, що відповідає документу MongoDB.

Mongoose дає величезний набір функціональних можливостей для створення та роботи зі схемами. Наразі Mongoose має вісім SchemaTypes (* типи даних схеми), котрі може мати властивість, що зберігається до MongoDB. Ці типи наступні:

1. String
2. Number
3. Date
4. Buffer
5. Boolean

6. Mixed
7. ObjectId (* унікальний ідентифікатор об'єкта, первинний ключ, _id)
8. Array

Для кожного типу даних можна:

1. зазначити значення за налаштуванням
2. зазначити функцію користувача для перевірки даних
3. зазначити, що поле необхідно заповнити
4. зазначити get-функцію (геттер), яка дозволяє вам проводити маніпуляції над даними до їх повернення у вигляді об'єкта
5. зазначити set-функцію (* сеттер), яка дозволяє вам проводити маніпуляції над даними перед їх зберіганням до бази даних
6. визначити індекси для більш швидкого отримання даних

Окрім цих спільних можливостей для деяких типів даних також можна налаштувати особливості зберігання та отримання даних із бази даних. Наприклад, для типу даних String можна зазначити наступні додаткові опції:

1. конвертація даних до нижнього регістру
2. конвертація даних до верхнього регістру
3. обрізання даних перед зберіганням
4. визначення регулярного виразу, який дозволяє в процесі перевірки даних обмежити дозволені для зберігання варіанти даних
5. визначення переліку, який дозволяє установити список припустимих рядків

Для властивостей типу Number і Date можна зазначити мінімально та максимально допустне значення. Більшість із восьми допустних типів даних мають бути вам добре знайомі. Однак, деякі (Buffer, Mixed, ObjectId та Array) можуть викликати труднощі. Тип даних Buffer дозволяє вам зберігати двійкові дані. Типовим прикладом двійкових даних може бути зображення чи закодований файл, наприклад, документ у PDF-форматі (* формат передаваного документа).

Тип даних Mixed використовується для перетворення властивості в "неперебірливе" поле (поле, в якому припустимі дані будь-якого типу). Так само, як багато розробників використовують MongoDB для різних цілей, у цьому полі можна зберігати дані різного типу, оскільки відстутня визначена структура. Обачливо використовуйте цей тип даних, оскільки він обмежує можливості, надавані Mongoose, наприклад, перевірку даних та відстежування змін сутності для автоматичного оновлення властивості при зберіганні. Тип даних ObjectId використовується звичайно для визначення посилання на інший документ у вашій базі даних. Наприклад, якщо б у вас була колекція книг та авторів, документ книги міг би містити властивість ObjectId, шо посилається на визначеного автора документа. Тип даних Array дозволяє вам зберігати JavaScript-подібні масиви. Завдяки цьому типу даних ви можете виконувати над даними типові JavaScript операції над масивами, наприклад, push, pop, shift, slice і т.д.

ВИСНОВКИ ДО РОЗДІЛУ 2

У другому розділі були описані функціональні та нефункціональні вимоги до застосунку.

Було описано принцип дії, переваги та недоліки архітектурних рішень на основі mongoDb.

Програма була розроблена з використанням сучасного стеку технологій, що використовується при створенні інтернет ресурсів з використанням сучасного середовища розробки VsCode.

Також було ознойомення з додатковим програмним забезпеченням для роботи бози даних ODM mongoose.

РОЗДІЛ 3
РОЗРОБКА ТА ТЕСТУВАННЯ ЗАСТОСУНКУ

3.1. Визначення сутностей інтернет магазину

Визначення сутностей перед розробкою інтернет-магазину є важливим, оскільки це дозволяє структурувати дані, визначення сутностей дає чітке розуміння того, як дані будуть організовані і зв'язані між собою, планувати функціональність, розуміння сутностей допомагає визначити, які функції потрібні для кожної з них, оптимізувати розробку, попереднє визначення структури даних спрощує процес розробки і запобігає потребі в значних змінах під час розробки, забезпечити масштабованість, правильно структуровані сутності полегшують масштабування і розширення магазину в майбутньому.

Першою сутністю було визначено користувачів (Users).

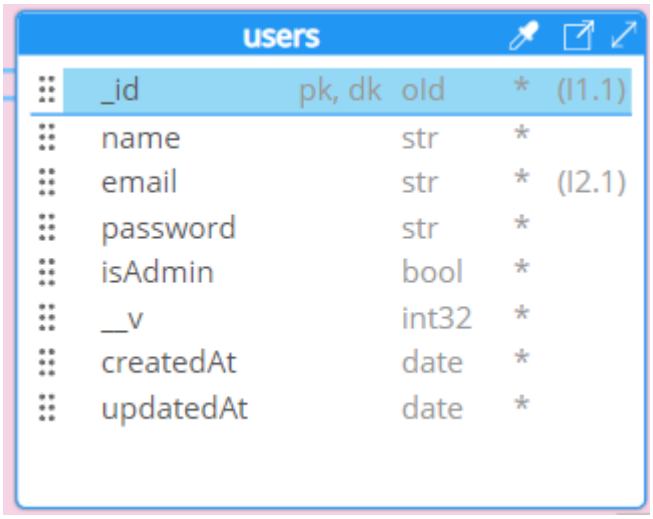

Рис. 3.1. Сутність користувача

Дана сутність має наступні властивості:

1. Ім'я (Name): Персональне ім'я користувача, яке використовується для ідентифікації та спілкування на сайті.

2. Електронна Пошта (Email): Унікальна електронна адреса користувача, що служить для входу в систему та комунікації.

3. Пароль (Password): Захищений пароль для аутентифікації користувача в системі.

4. Адміністратор (isAdmin): Логічна змінна (true/false), яка вказує, чи має користувач права адміністратора для управління магазином.

Наступною сутністю є товар (Product)

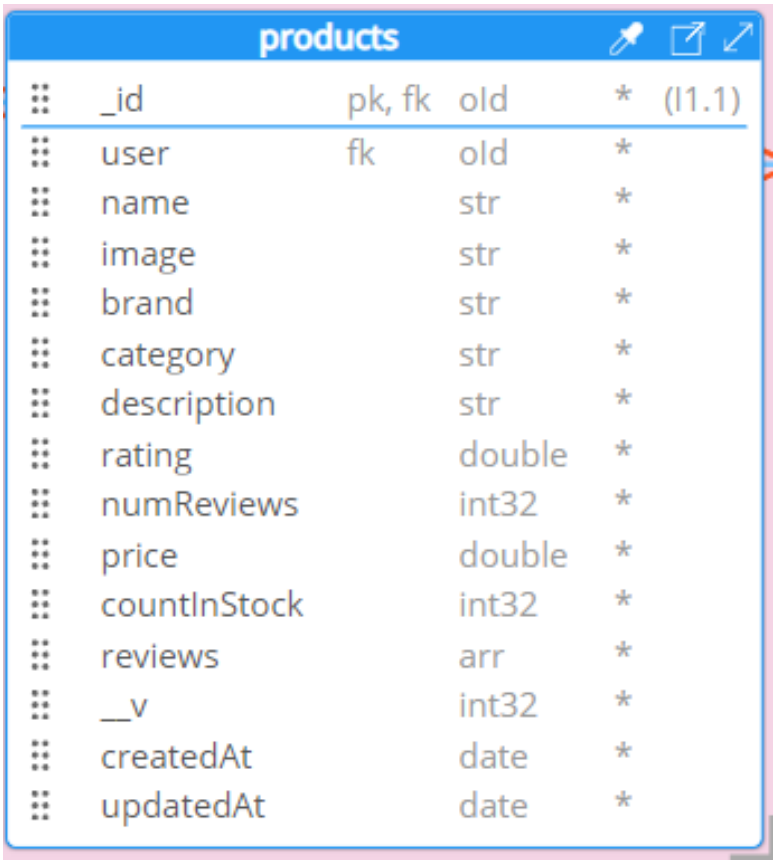

Рис. 3.2. Сутність продукту

1. _id: Унікальний ідентифікатор продукту.
2. user: Ідентифікатор користувача, який додав продукт.
3. name: Назва продукту.
4. image: Шлях до зображення продукту.
5. brand: Бренд продукту.
6. category: Категорія продукту.

7. description: Опис продукту.
8. rating: Рейтинг продукту.
9. numReviews: Кількість відгуків.
10. price: Ціна продукту.
11. countInStock: Кількість продуктів на складі.
12. reviews: Масив відгуків про продукт.
13. __v: Версія схеми в MongoDB.
14. createdAt та updatedAt: Час створення та оновлення запису.

І останньою сутністю на даному етапі буде сутність замовлення (Order):

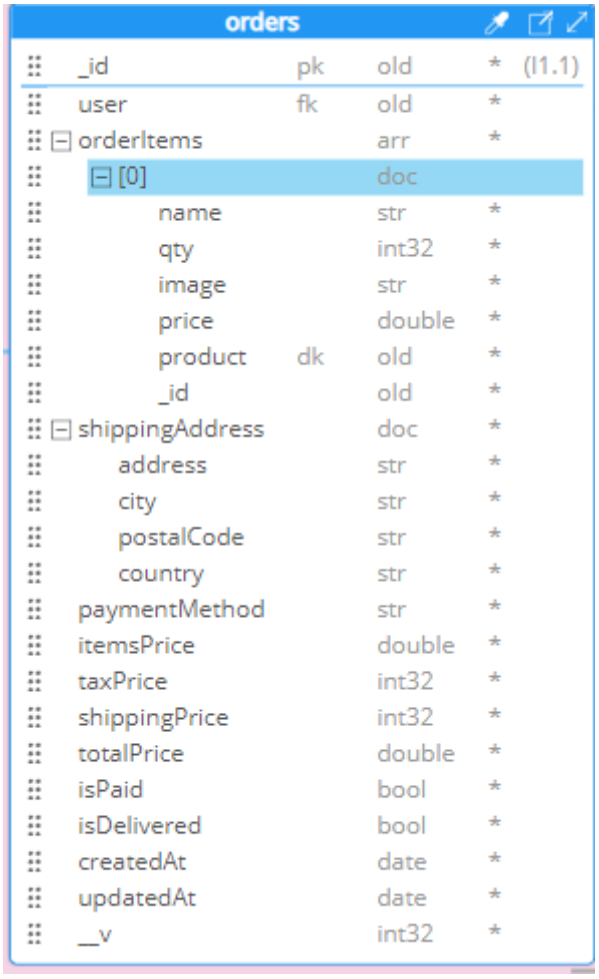

Рис. 3.3. Діаграма сутності продажу

1. _id: Унікальний ідентифікатор замовлення.
2. user: Ідентифікатор користувача, який зробив замовлення.
3. orderItems: Список товарів у замовленні, з інформацією про кожен товар, включаючи назву, кількість, зображення, ціну та ідентифікатор продукту.
4. shippingAddress: Адреса доставки замовлення.
5. paymentMethod: Спосіб оплати.
6. itemsPrice: Загальна вартість товарів.
7. taxPrice: Сума податку.
8. shippingPrice: Вартість доставки.
9. totalPrice: Загальна вартість замовлення.
10. isPaid: Чи було замовлення оплачено.
11. isDelivered: Чи було замовлення доставлено.
12. createdAt, updatedAt: Час створення та останнього оновлення замовлення.
13. __v: Версія схеми в MongoDB.

Створення зв’язків між сутностями

Хоч і в данній роботі буде використовуватися не реліаційна база даних деякі зв’язки між сутностями будуть встановлені. Єдиною різницею між реліаційними базами даних є те що контролюватися ці зв’язки будуть на стороні додатку, а не в самій базі даних, що має свії недоліки у вигляді додаткового коду який буде потрібно написати самостійно та підтримувати.

Для мінімальної роботи інтернет магазину необходно встановити зв’язок між користувачами та замовленнями, так в об’єкті замовлення буде зберігатися ідентифікатор користувача котрий оформив це замовлення, також в замовленні необхідно зберігати ідентифікатори придбаних продуктів. Щодо самих продуктів передбачено зберігати ідентифікатор

користувача з правами адміністатора котрий створив цей продукт. Нижче на рис. 3.4 можна ознайомитися з діаграмою зв’язків сутностей.

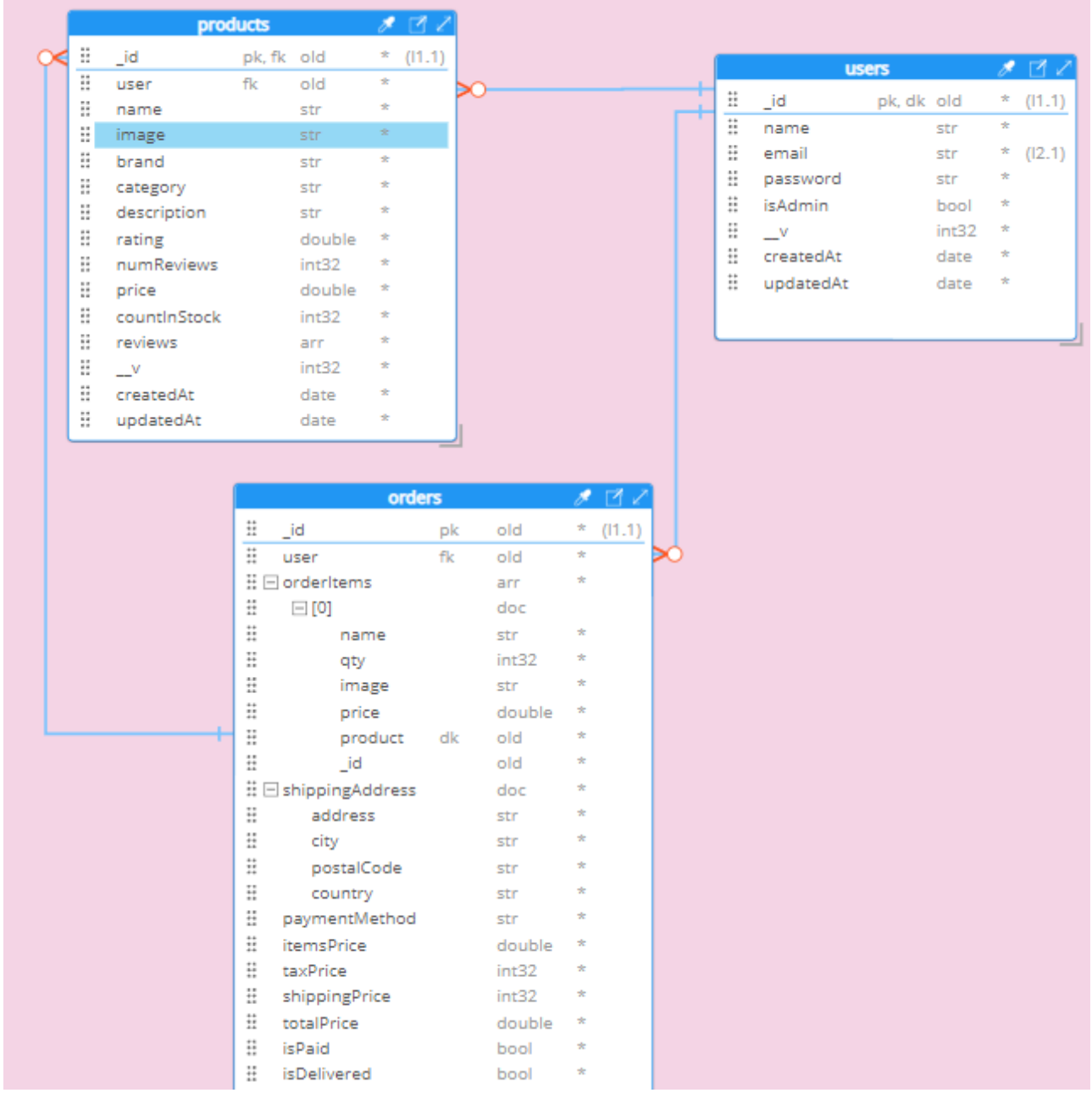

Рис. 3.4. Діаграма зв’язків сутностей

Дана структура даних цілком задовільнить потреби базового інтернет магазину. А перевагою використання не реліційної бази даних є те, що в випадку коли буде необхідність додати нові поля, ми з легкістю додамо їх.

Також це є великою перевагою, що кожен продукт може мати свій власний список полів, що в реліаційній базі даних не можливо.

Створення моделей сутностей в Mongoose

Тепер коли ми маємо діаграму бази даних, можна створити моделі для кожної сутності.

Сутність юзера:

```
const userSchema = mongoose.Schema(
  {
    name: {
      type: String,
      required: true,
    },
    email: {
      type: String,
      required: true,
      unique: true,
    },
    password: {
      type: String,
      required: true,
    },
    isAdmin: {
      type: Boolean,
      required: true,
      default: false,
    },
  },
  {
    timestamps: true,
  }
);
```

Як ви можете бачити кожен окремий атрибут має свій тип даних, котрий буде перевірений ORM, також всі поля були позначені як обов'язкові і в випадку якщо при створені користувача не буде передане якесь з цих атрибутів, ORM поверне помилку. Також окремо було активовано

функіанал автоматичного додавання часу створення, тепер не потрібно вручну передавати час, що зручно.

Сутність продукту:

```
const reviewSchema = mongoose.Schema(
  {
    name: { type: String, required: true },
    rating: { type: Number, required: true },
    comment: { type: String, required: true },
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'User',
    },
  },
  {
    timestamps: true,
  }
);

const productSchema = mongoose.Schema(
  {
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'User',
    },
    name: {
      type: String,
      required: true,
    },
    image: {
      type: String,
      required: true,
    },
    brand: {
      type: String,
      required: true,
    },
    category: {
      type: String,
```

```
      required: true,
    },
    description: {
      type: String,
      required: true,
    },
    reviews: [reviewSchema],
    rating: {
      type: Number,
      required: true,
      default: 0,
    },
    numReviews: {
      type: Number,
      required: true,
      default: 0,
    },
    price: {
      type: Number,
      required: true,
      default: 0,
    },
    countInStock: {
      type: Number,
      required: true,
      default: 0,
    },
  },
  {
    timestamps: true,
  }
);
```

З відмінностей можна побачити, що продукти мають масив з відгуками, а сам відгук має свою окрему схему.

Також тут відображається зв’язок з моделю користувача через атрибут user, що фактично зберігається як ідентифікатор юзера в самій базі даних. Пізнише ми використаємо цей зв’язок для швидкого об’єднання данних поміж двух коллекцій.

Сутність замовлення:

```
const orderSchema = mongoose.Schema(
  {
    user: {
      type: mongoose.Schema.Types.ObjectId,
      required: true,
      ref: 'User',
    },
    orderItems: [
      {
        name: { type: String, required: true },
        qty: { type: Number, required: true },
        image: { type: String, required: true },
        price: { type: Number, required: true },
        product: {
          type: mongoose.Schema.Types.ObjectId,
          required: true,
          ref: 'Product',
        },
      },
    ],
    shippingAddress: {
      address: { type: String, required: true },
      city: { type: String, required: true },
      postalCode: { type: String, required: true },
      country: { type: String, required: true },
    },
    paymentMethod: {
      type: String,
      required: true,
    },
    paymentResult: {
      id: { type: String },
      status: { type: String },
      update_time: { type: String },
      email_address: { type: String },
    },
    itemsPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
```

```
    taxPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    shippingPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    totalPrice: {
      type: Number,
      required: true,
      default: 0.0,
    },
    isPaid: {
      type: Boolean,
      required: true,
      default: false,
    },
    paidAt: {
      type: Date,
    },
    isDelivered: {
      type: Boolean,
      required: true,
      default: false,
    },
    deliveredAt: {
      type: Date,
    },
  },
  {
    timestamps: true,
  }
);
```

В цьому випадку також було використано синтаксис для очевидного позначення зв’язку з продуктами та користувачами.

3.2. Виконання правил безпеки

Паролі

Найпростіший спосіб – це запис паролів просто в базу даних у незашифрованому вигляді. Відповідно, при спробі користувача аутентифікуватися залишається тільки порівняти ланцюжок символів, що вводиться ним, з тим, що зберігається у вас в базі.

У цьому випадку є ризик того, що зловмисники зможуть у той чи інший спосіб вкрасти цю базу даних. Наприклад, використання будь-яких вразливостей у використовуваному для зберігання даних ПЗ. Інший варіант – таблицю з паролями може вкрасти недобросовісний співробітник із високим рівнем доступу. Або для крадіжки паролів можуть бути використані вкрадені або перехоплені облікові дані співробітника. Загалом, варіантів того, що може піти не так, тут багато. Головна думка: якщо зберігати якісь дані у відкритому вигляді, на них рано чи пізно хтось може накласти руки.

Найправильніше взагалі не зберігати в себе паролі. Так-так, дуже проста ідея: якщо у вас чогось немає, це в принципі не може бути вкрадено.

Але як тоді перевірити, чи правильний пароль вводить користувач, який намагається залогінитися у ваш сервіс? Для цього дуже зручно використовувати хеш-функції: спеціальні криптографічні алгоритми перетворюють будь-які дані на рядок бітів фіксованої довжини передбачуваним, але необоротним чином.

Передбачуваним тут означає, що одні й самі дані завжди будуть перетворені на той самий хеш. А незворотнім — з хеша неможливо відновити ті дані, які були захешовані.

Саме так і роблять всі онлайн-сервіси, які хоча б трохи дорожать своєю репутацією і піклуються про захист від витоку даних користувача. Коли користувач вводить пароль під час реєстрації у сервісі, у базі даних поруч із логіном записується не сам пароль, а хеш цього пароля. І,

відповідно, при наступних спробах залогінити записаний в базу хеш порівнюється з хеш комбінації символів, що вводиться користувачем. Якщо хеші збігаються — це означає, що пароль відповідає вказаному при реєстрації. У разі витоку бази в руках зловмисників виявляються не самі паролі, а їх хеші, з яких неможливо відновити оригінальні дані (необоротність, пам'ятаєте?). Це набагато менш небезпечно, ніж доступ до паролів у відкритому вигляді, але радіти поки що рано: якщо в руках у злочинців виявилися хеші, то вони можуть використовувати їх для атаки перебором. Здобувши базу, зломщики підбиратимуть комбінації символів під записані в ній хеші. Тобто вони братимуть якусь комбінацію, обчислюватимуть її хеш та шукатимуть збіги за всіма записами бази. Якщо збігів не знайдено, то брати наступну і так далі. У разі збігу виходить, що паролі, на основі яких було обчислено відповідні хеші, тепер відомі.

Ще гірше те, що насправді процес злому хешованих паролів можна суттєво прискорити: для цього використовуються так звані райдужні таблиці (rainbow tables). Це величезні масиви даних із заздалегідь обчисленими хешами для купи різноманітних комбінацій символів. Відповідно, досить просто шукати збіги між вмістом райдужної таблиці та вкраденої бази. І, звичайно ж, робиться це все не вручну, а автоматизовано, тому процес злому паролів може займати набагато менше часу, ніж усім нам хотілося б.

Для хешування паролів користувачів було використано стандартну бібліотеку bcrypt, вона є сучасною та має весь необзідний функціонал для цього.

```
userSchema.pre('save', async function (next) {
  if (!this.isModified('password')) {
    next();
  }
  const salt = await bcrypt.genSalt(10);
  this.password = await bcrypt.hash(this.password, salt);
});
```

Цей фрагмент коду використовує Mongoose middleware для хешування пароля користувача перед збереженням в базі даних. Коли запускається подія 'save' на схемі користувача, код перевіряє, чи було змінено поле 'password'. Якщо пароль не змінено, код просто переходить до наступної мідлвари. Якщо ж пароль змінено, використовується бібліотека bcrypt для створення "солі" та хешування пароля, після чого хешований пароль зберігається в базі даних.

```
userSchema.methods.matchPassword = async function
(enteredPassword) {
  return await bcrypt.compare(enteredPassword,
this.password);
};
```

Цей код представляє метод matchPassword для схеми користувача в Mongoose, який використовується для перевірки, чи введений пароль відповідає захешованому паролю в базі даних. Метод compare від bcrypt порівнює введений пароль з хешем пароля, який зберігається в об'єкті користувача (this.password), і повертає true або false залежно від результату порівняння.

Сесії користувачів

Для зберігання сесій користуачів було обрано механізм JWT, він чудово подходить для потреб цього проекту.

Почнемо написання коду авторизації додатку в окремому файлі auth.js

```
const protect = asyncHandler(async (req, res, next) => {
  let token;

  // Read JWT from the 'jwt' cookie
  token = req.cookies.jwt;

  if (token) {
    try {
      const decoded = jwt.verify(token,
process.env.JWT_SECRET);
```

```
      req.user = await
User.findById(decoded.userId).select('-password');

      next();
    } catch (error) {
      console.error(error);
      res.status(401);
      throw new Error('Not authorized, token failed');
    }
  } else {
    res.status(401);
    throw new Error('Not authorized, no token');
  }
});

// User must be an admin
const admin = (req, res, next) => {
  if (req.user && req.user.isAdmin) {
    next();
  } else {
    res.status(401);
    throw new Error('Not authorized as an admin');
  }
};
```

Цей код містить дві мідлвари для використання в Express.js додатку. Перша, protect, перевіряє наявність та валідність JWT токена, що зберігається в куках (cookies) запиту. Якщо токен є валідним, інформація про користувача дістається з бази даних та додається до об'єкта запиту. У випадку помилки або відсутності токена генерується помилка авторизації. Друга мідлвара, admin, перевіряє, чи користувач, пов'язаний із запитом, має адміністративні права. Якщо ні, генерується помилка про недостатню авторизацію.

Згодом ці методи будуть використовуватися в окремих точках доступу за потреби.

3.3. Розробка клієнтської частини

Для написаня клієнтського коду було обрано React. Вхідна точка в застосунок буде виглядати наступним чином:

```
const App = () => {
  const dispatch = useDispatch();

  useEffect(() => {
    const expirationTime =
localStorage.getItem('expirationTime');
    if (expirationTime) {
      const currentTime = new Date().getTime();

      if (currentTime > expirationTime) {
        dispatch(logout());
      }
    }
  }, [dispatch]);

  return (
    <>
      <ToastContainer />
      <Header />
      <main className='py-3'>
        <Container>
          <Outlet />
        </Container>
      </main>
      <Footer />
    </>
  );
};
```

Цей фрагмент коду визначає компонент App в React додатку. Він використовує хук useEffect для перевірки терміну дії токена, збереженого в

localStorage. Якщо час закінчився, викликається дія logout через Redux dispatch. У компоненті також рендеряться ToastContainer, Header, основний контент через Outlet, і Footer. Контент основної частини врапований у Container, що дозволяє структурувати вміст всередині.

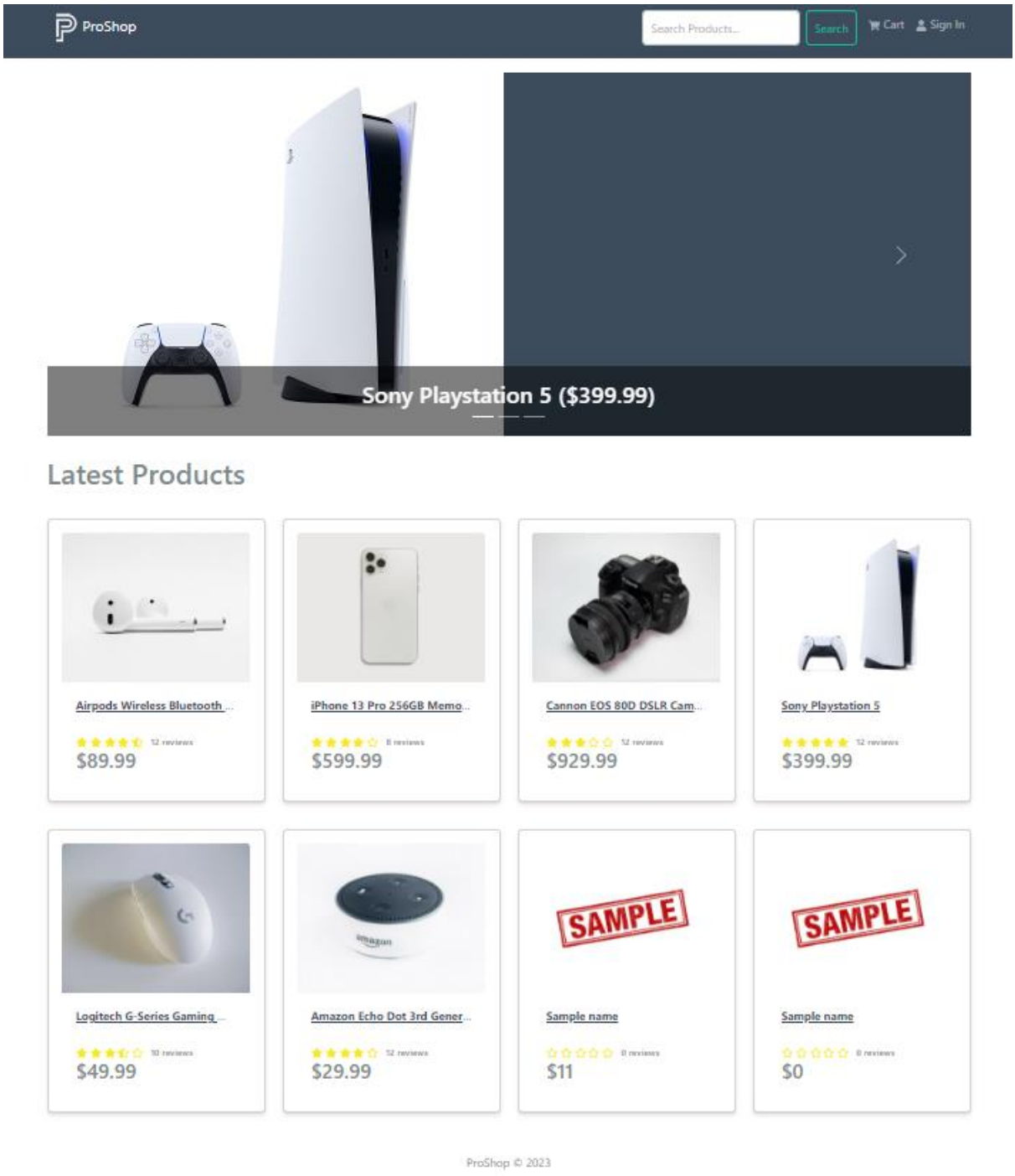

Рис. 3.3. Головна сторінка сайту

Сторінка реєстрації

Сторінка реєстрації в інтернет-сервісах має ключове значення, оскільки:

1. Дозволяє Новим Користувачам Створити Акаунт: Вона є першим кроком для користувачів до взаємодії з сервісом.

2. Збір Основної Інформації: Запитує основні дані для створення профілю та ідентифікації користувача.

3. Забезпечення Безпеки: Встановлює первинні заходи безпеки, наприклад, створення надійного пароля.

4. Відсівання Ботів та Зловмисників: Може включати механізми перевірки, які допомагають уникнути фальшивих або зловмисних реєстрацій.

```
<FormContainer>
      <h1>Register</h1>
      <Form onSubmit={submitHandler}>
        <Form.Group className='my-2' controlId='name'>
          <Form.Label>Name</Form.Label>
          <Form.Control
            type='name'
            placeholder='Enter name'
            value={name}
            onChange={(e) => setName(e.target.value)}
          ></Form.Control>
        </Form.Group>

        <Form.Group className='my-2' controlId='email'>
          <Form.Label>Email Address</Form.Label>
          <Form.Control
            type='email'
            placeholder='Enter email'
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          ></Form.Control>
        </Form.Group>
```

```
        <Form.Group className='my-2' controlId='password'>
          <Form.Label>Password</Form.Label>
          <Form.Control
            type='password'
            placeholder='Enter password'
            value={password}
            onChange={(e) => setPassword(e.target.value)}
          ></Form.Control>
        </Form.Group>
        <Form.Group className='my-2'
controlId='confirmPassword'>
          <Form.Label>Confirm Password</Form.Label>
          <Form.Control
            type='password'
            placeholder='Confirm password'
            value={confirmPassword}
            onChange={(e) =>
setConfirmPassword(e.target.value)}
          ></Form.Control>
        </Form.Group>

        <Button disabled={isLoading} type='submit'
variant='primary'>
          Register
        </Button>

        {isLoading && <Loader />}
      </Form>

      <Row className='py-3'>
        <Col>
          Already have an account?{' '}
          <Link to={redirect ?
`/login?redirect=${redirect}` : '/login'}>
            Login
          </Link>
        </Col>
      </Row>
    </FormContainer>
```

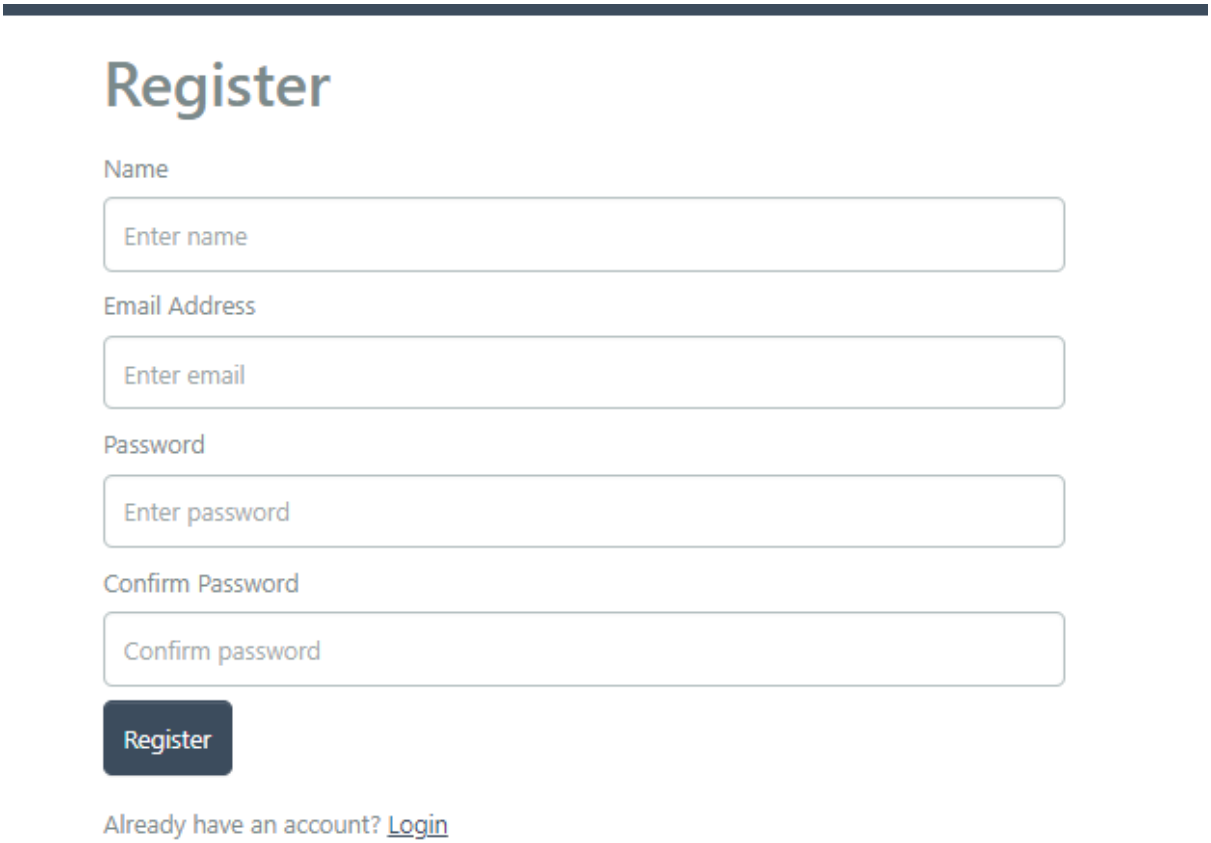

Рис. 3.4. Форма реєстрації

Сторінка авторизації

Сторінка авторизації важлива в будь-якому онлайн сервісі чи додатку з кількох причин:

1. Захист Інформації: Авторизація допомагає захистити конфіденційність та безпеку інформації користувачів, запобігаючи несанкціонованому доступу.

2. Персоналізація Досвіду: Вона дозволяє надати користувачам персоналізований досвід, пам'ятаючи їхні налаштування та історію використання.

3. Валідація Користувачів: Авторизація допомагає ідентифікувати та валідувати користувачів, що є ключовим для забезпечення відповідальності та управління доступом до ресурсів.

4. Запобігання Махінаціям: Захищає від махінацій та зловмисної діяльності, перевіряючи, що користувачі є тими, за кого вони себе видають.

```
    <FormContainer>
      <h1>Sign In</h1>

      <Form onSubmit={submitHandler}>
        <Form.Group className='my-2' controlId='email'>
          <Form.Label>Email Address</Form.Label>
          <Form.Control
            type='email'
            placeholder='Enter email'
            value={email}
            onChange={(e) => setEmail(e.target.value)}
          ></Form.Control>
        </Form.Group>

        <Form.Group className='my-2' controlId='password'>
          <Form.Label>Password</Form.Label>
          <Form.Control
            type='password'
            placeholder='Enter password'
            value={password}
            onChange={(e) => setPassword(e.target.value)}
          ></Form.Control>
        </Form.Group>

        <Button disabled={isLoading} type='submit'
variant='primary'>
          Sign In
        </Button>

        {isLoading && <Loader />}
      </Form>

      <Row className='py-3'>
        <Col>
```

```
          New Customer?{' '}
          <Link to={redirect ?
`/register?redirect=${redirect}` : '/register'}>
            Register
          </Link>
        </Col>
      </Row>
    </FormContainer>
```

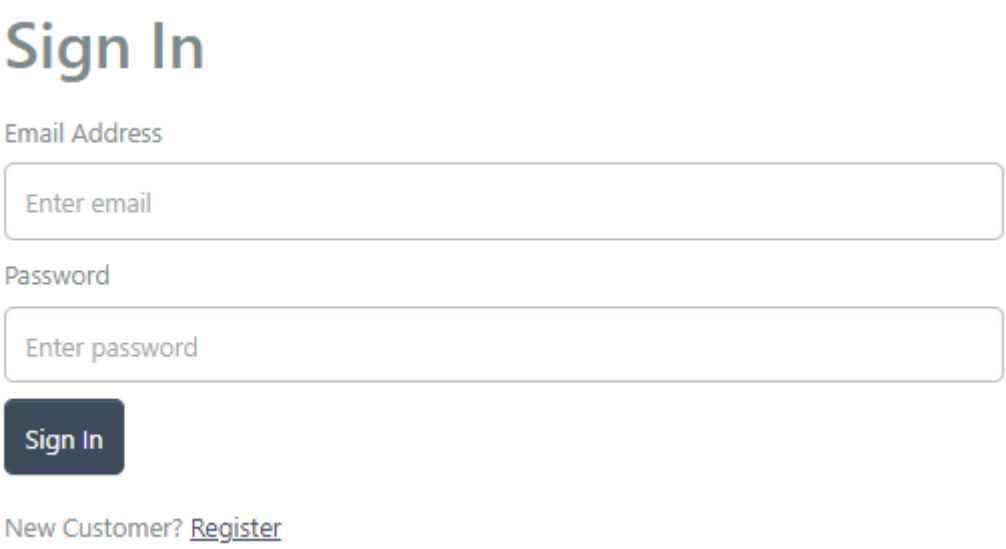

Рис. 3.5. форма авторизації

Сторінка товару

Сторінка товару в інтернет-магазині відіграє ключову роль, оскільки:

1. Демонструє Деталі Товару: Надає детальний опис, фотографії, ціну та характеристики продукту.

2. Функції Користувача: Дозволяє користувачам читати відгуки, додавати товар до кошика або списку бажань.

3. Сприяє Рішенню про Покупку: Забезпечує всю необхідну інформацію, щоб допомогти користувачам ухвалити рішення про покупку.

4. Маркетингові Можливості: Може включати рекомендації схожих товарів або спеціальні пропозиції.

```
<>
  <Link className='btn btn-light my-3' to='/'>
    Go Back
  </Link>
  {isLoading ? (
    <Loader />
  ) : error ? (
    <Message variant='danger'>
      {error?.data?.message || error.error}
    </Message>
  ) : (
    <>
      <Meta title={product.name} description={product.description} />
      <Row>
        <Col md={6}>
          <Image src={product.image} alt={product.name} fluid />
        </Col>
        <Col md={3}>
          <ListGroup variant='flush'>
            <ListGroup.Item>
              <h3>{product.name}</h3>
            </ListGroup.Item>
            <ListGroup.Item>
              <Rating
                value={product.rating}
                text={`${product.numReviews} reviews`}
              />
            </ListGroup.Item>
            <ListGroup.Item>Price: ${product.price}</ListGroup.Item>
            <ListGroup.Item>
              Description: {product.description}
            </ListGroup.Item>
          </ListGroup>
        </Col>
        <Col md={3}>
          <Card>
            <ListGroup variant='flush'>
              <ListGroup.Item>
                <Row>
                  <Col>Price:</Col>
                  <Col>
                    <strong>${product.price}</strong>
                  </Col>
                </Row>
              </ListGroup.Item>
              <ListGroup.Item>
                <Row>
                  <Col>Status:</Col>
                  <Col>
```

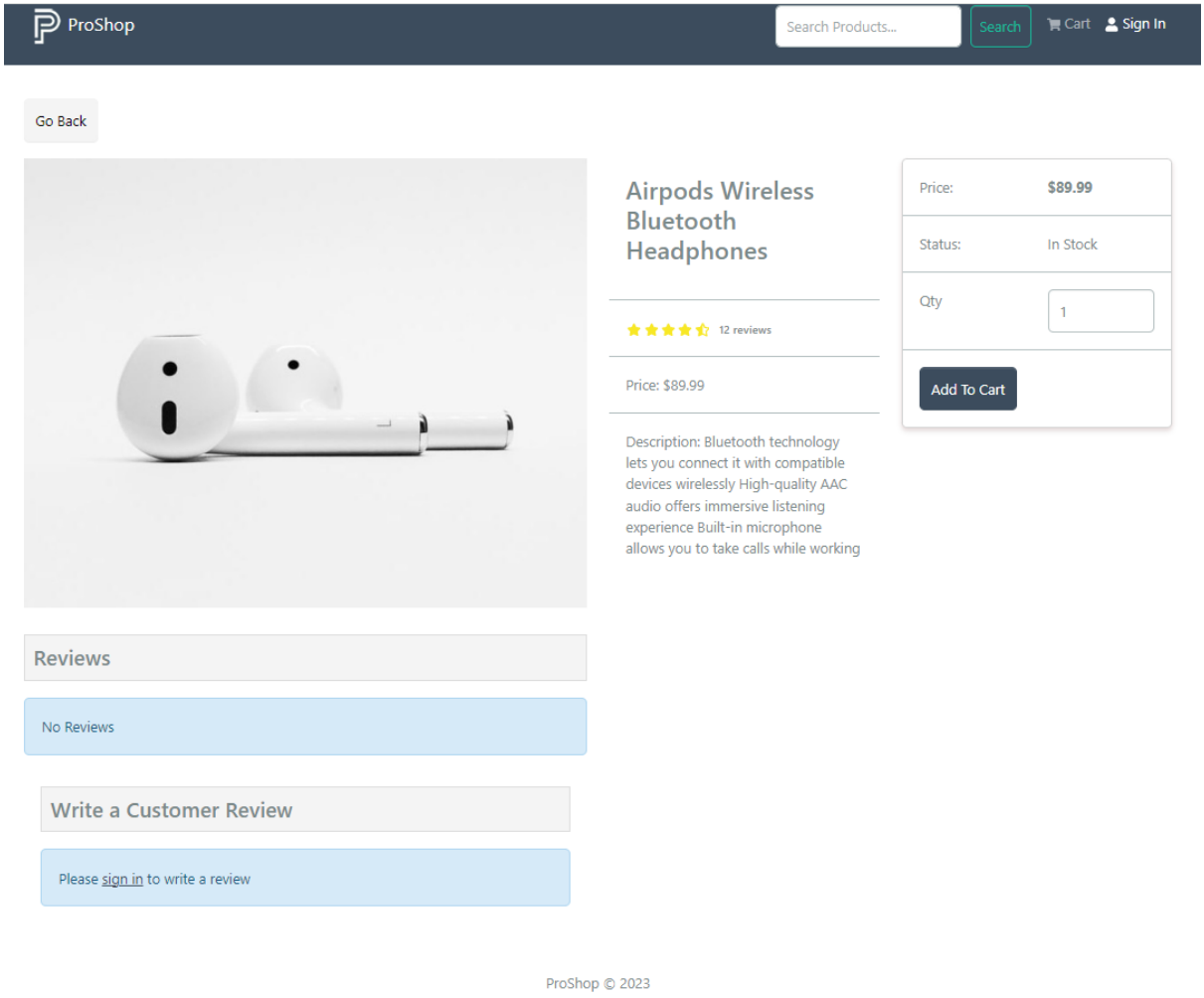

Рис. 3.6. Сторінка продукту

Сторінка створення продукту

Сторінка створення продукту в інтернет-магазині важлива для управління каталогом товарів, оскільки:

1. Дозволяє Додавати Нові Товари: Забезпечує інтерфейс для введення інформації про новий продукт.

2. Налаштування Параметрів: Можна вказати назву, опис, ціну, категорію, запаси та інші деталі.

3. Завантаження Зображень: Надає можливість додавати зображення продукту.

4. Управління Запасами: Дозволяє налаштувати кількість доступних одиниць продукту.

```
<FormContainer>
  <h1>Edit Product</h1>
  {loadingUpdate && <Loader />}
  {isLoading ? (
    <Loader />
  ) : error ? (
    <Message variant='danger'>{error.data.message}</Message>
  ) : (
    <Form onSubmit={submitHandler}>
      <Form.Group controlId='name'>
        <Form.Label>Name</Form.Label>
        <Form.Control
          type='name'
          placeholder='Enter name'
          value={name}
          onChange={(e) => setName(e.target.value)}
        ></Form.Control>
      </Form.Group>

      <Form.Group controlId='price'>
        <Form.Label>Price</Form.Label>
        <Form.Control
          type='number'
          placeholder='Enter price'
          value={price}
          onChange={(e) => setPrice(e.target.value)}
        ></Form.Control>
      </Form.Group>

      <Form.Group controlId='image'>
        <Form.Label>Image</Form.Label>
        <Form.Control
          type='text'
          placeholder='Enter image url'
          value={image}
          onChange={(e) => setImage(e.target.value)}
        ></Form.Control>
        <Form.Control
          label='Choose File'
          onChange={uploadFileHandler}
          type='file'
```

```
          ></Form.Control>
          {loadingUpload && <Loader />}
        </Form.Group>
```

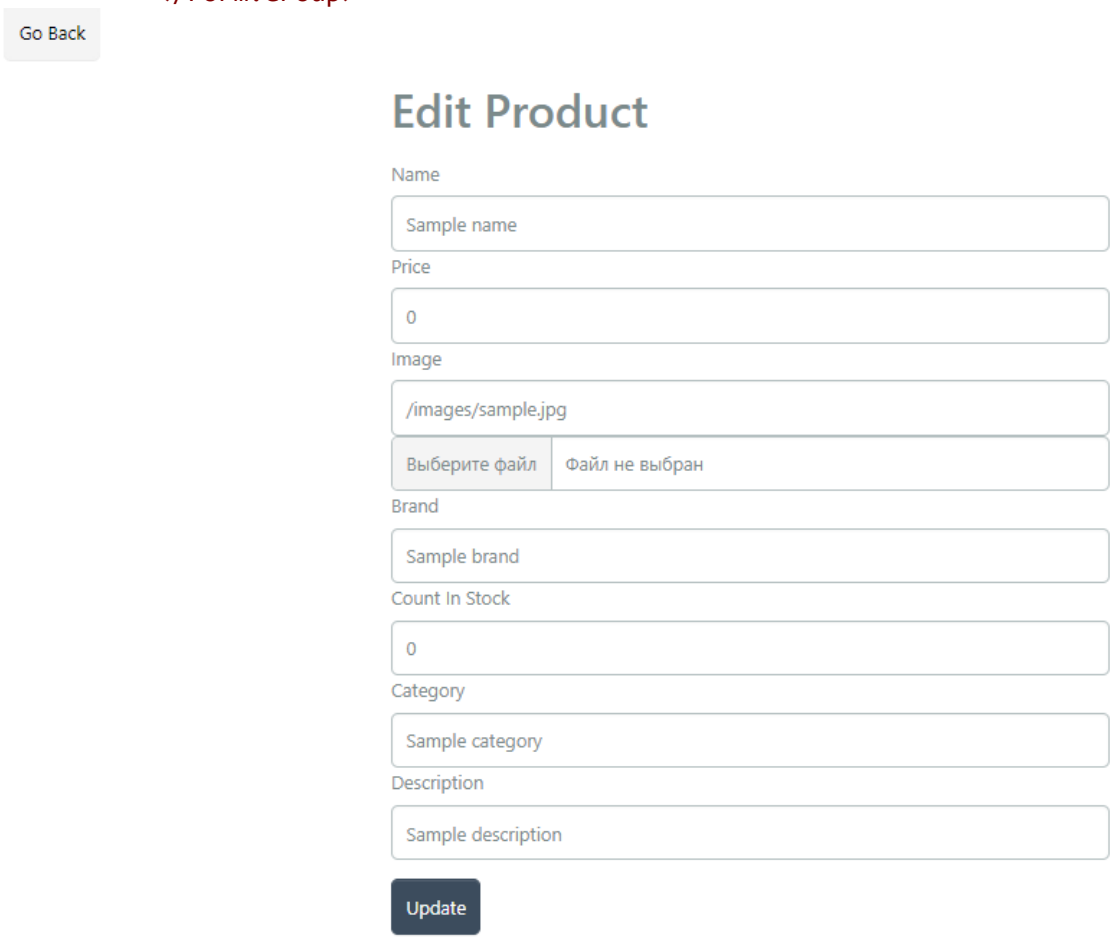

Рис. 3.7. Сторінка створення та редагування продукту

Сторінка списку продуктів для адміністратора

Дана сторінка необхідна для адекватного адміністрування магазину. Вона має відображати який товар є і скільки його лишилося, також можна видалити не потрібний товар.

```
<>
      <Row className='align-items-center'>
        <Col>
          <h1>Products</h1>
        </Col>
        <Col className='text-end'>
          <Button className='my-3' onClick={createProductHandler}>
            <FaPlus /> Create Product
          </Button>
```

```
  </Col>
</Row>

{loadingCreate && <Loader />}
{loadingDelete && <Loader />}
{isLoading ? (
  <Loader />
) : error ? (
  <Message variant='danger'>{error.data.message}</Message>
) : (
  <>
    <Table striped bordered hover responsive className='table-sm'>
      <thead>
        <tr>
          <th>ID</th>
          <th>NAME</th>
          <th>PRICE</th>
          <th>CATEGORY</th>
          <th>BRAND</th>
          <th></th>
        </tr>
      </thead>
      <tbody>
        {data.products.map((product) => (
          <tr key={product._id}>
            <td>{product._id}</td>
            <td>{product.name}</td>
            <td>${product.price}</td>
            <td>{product.category}</td>
            <td>{product.brand}</td>
            <td>
              <LinkContainer to={`/admin/product/${product._id}/edit`}>
                <Button variant='light' className='btn-sm mx-2'>
                  <FaEdit />
                </Button>
              </LinkContainer>
              <Button
                variant='danger'
                className='btn-sm'
                onClick={() => deleteHandler(product._id)}
              >
                <FaTrash style={{ color: 'white' }} />
              </Button>
            </td>
          </tr>
        ))}
      </tbody>
    </Table>
    <Paginate pages={data.pages} page={data.page} isAdmin={true} />
```

```
      </>
    )}
  </>
```

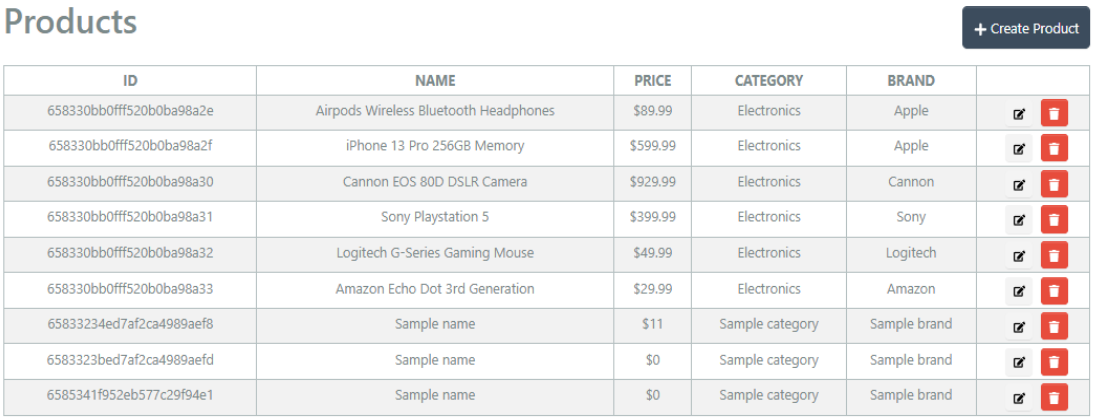

Рис. 3.8. Сторінка списку продуктів

Сторінка профілю користувача

Невід'ємна сторінка для системи з користувачами, так як тут можна переграну персональні дані та редагувати їх. Також тут представлений список минулих замовлень, та їх статус.

```
<Row>
      <Col md={3}>
        <h2>User Profile</h2>

        <Form onSubmit={submitHandler}>
          <Form.Group className='my-2' controlId='name'>
            <Form.Label>Name</Form.Label>
            <Form.Control
              type='text'
              placeholder='Enter name'
              value={name}
              onChange={(e) => setName(e.target.value)}
            ></Form.Control>
          </Form.Group>

          <Form.Group className='my-2' controlId='email'>
            <Form.Label>Email Address</Form.Label>
            <Form.Control
              type='email'
              placeholder='Enter email'
              value={email}
              onChange={(e) => setEmail(e.target.value)}
```

```
        ></Form.Control>
      </Form.Group>

      <Form.Group className='my-2' controlId='password'>
        <Form.Label>Password</Form.Label>
        <Form.Control
          type='password'
          placeholder='Enter password'
          value={password}
          onChange={(e) => setPassword(e.target.value)}
        ></Form.Control>
      </Form.Group>

      <Form.Group className='my-2' controlId='confirmPassword'>
        <Form.Label>Confirm Password</Form.Label>
        <Form.Control
          type='password'
          placeholder='Confirm password'
          value={confirmPassword}
          onChange={(e) => setConfirmPassword(e.target.value)}
        ></Form.Control>
      </Form.Group>

      <Button type='submit' variant='primary'>
        Update
      </Button>
      {loadingUpdateProfile && <Loader />}
    </Form>
  </Col>
  <Col md={9}>
    <h2>My Orders</h2>
    {isLoading ? (
      <Loader />
    ) : error ? (
      <Message variant='danger'>
        {error?.data?.message || error.error}
      </Message>
    ) : (
      <Table striped hover responsive className='table-sm'>
        <thead>
          <tr>
            <th>ID</th>
            <th>DATE</th>
            <th>TOTAL</th>
            <th>PAID</th>
            <th>DELIVERED</th>
            <th></th>
          </tr>
        </thead>
```

```
<tbody>
  {orders.map((order) => (
```

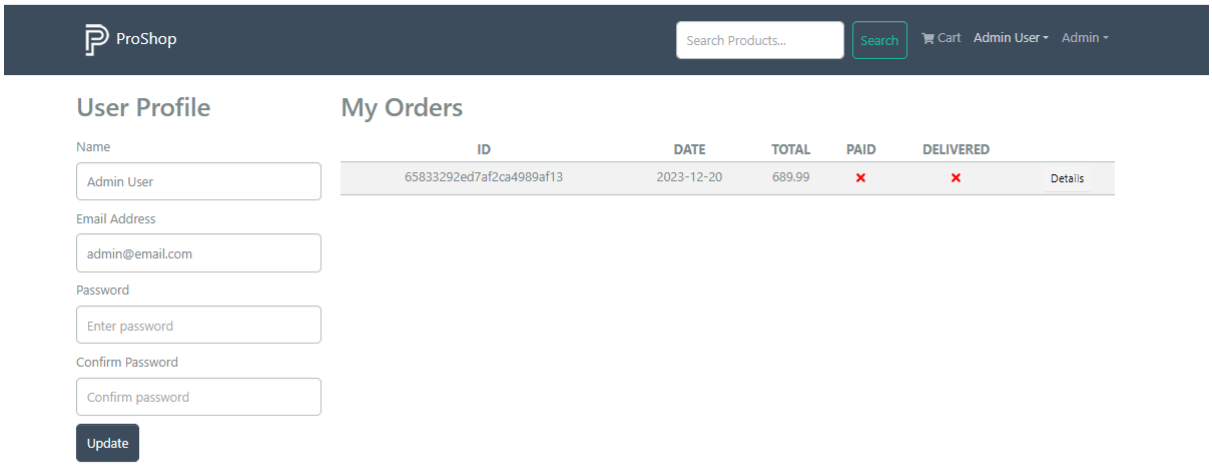

Рис. 3.9. Сторінка профілю користувача

Сторінка корзини

Сторінка корзини є ключувою в світі електроної комерції, тому до неї була окрема увага.

```
<Row>
      <Col md={8}>
        <h1 style={{ marginBottom: '20px' }}>Shopping Cart</h1>
        {cartItems.length === 0 ? (
          <Message>
            Your cart is empty <Link to='/'>Go Back</Link>
          </Message>
        ) : (
          <ListGroup variant='flush'>
            {cartItems.map((item) => (
              <ListGroup.Item key={item._id}>
                <Row>
                  <Col md={2}>
                    <Image src={item.image} alt={item.name} fluid rounded />
                  </Col>
                  <Col md={3}>
                    <Link to={`/product/${item._id}`}>{item.name}</Link>
                  </Col>
                  <Col md={2}>${item.price}</Col>
                  <Col md={2}>
                    <Form.Control
                      as='select'
```

```
                    value={item.qty}
                    onChange={(e) =>
                      addToCartHandler(item, Number(e.target.value))
                    }
                  >
                    {[...Array(item.countInStock).keys()].map((x) => (
                      <option key={x + 1} value={x + 1}>
                        {x + 1}
                      </option>
                    ))}
                  </Form.Control>
                </Col>
                <Col md={2}>
                  <Button
                    type='button'
                    variant='light'
                    onClick={() => removeFromCartHandler(item._id)}
                  >
                    <FaTrash />
                  </Button>
                </Col>
              </Row>
            </ListGroup.Item>
          ))}
        </ListGroup>
      )}
    </Col>
    <Col md={4}>
      <Card>
        <ListGroup variant='flush'>
          <ListGroup.Item>
            <h2>
              Subtotal ({cartItems.reduce((acc, item) => acc + item.qty,
0)})

              items
            </h2>
            $
            {cartItems
              .reduce((acc, item) => acc + item.qty * item.price, 0)
              .toFixed(2)}
          </ListGroup.Item>
          <ListGroup.Item>
            <Button
              type='button'
              className='btn-block'
              disabled={cartItems.length === 0}
              onClick={checkoutHandler}
            >
              Proceed To Checkout
```

```
            </Button>
          </ListGroup.Item>
```

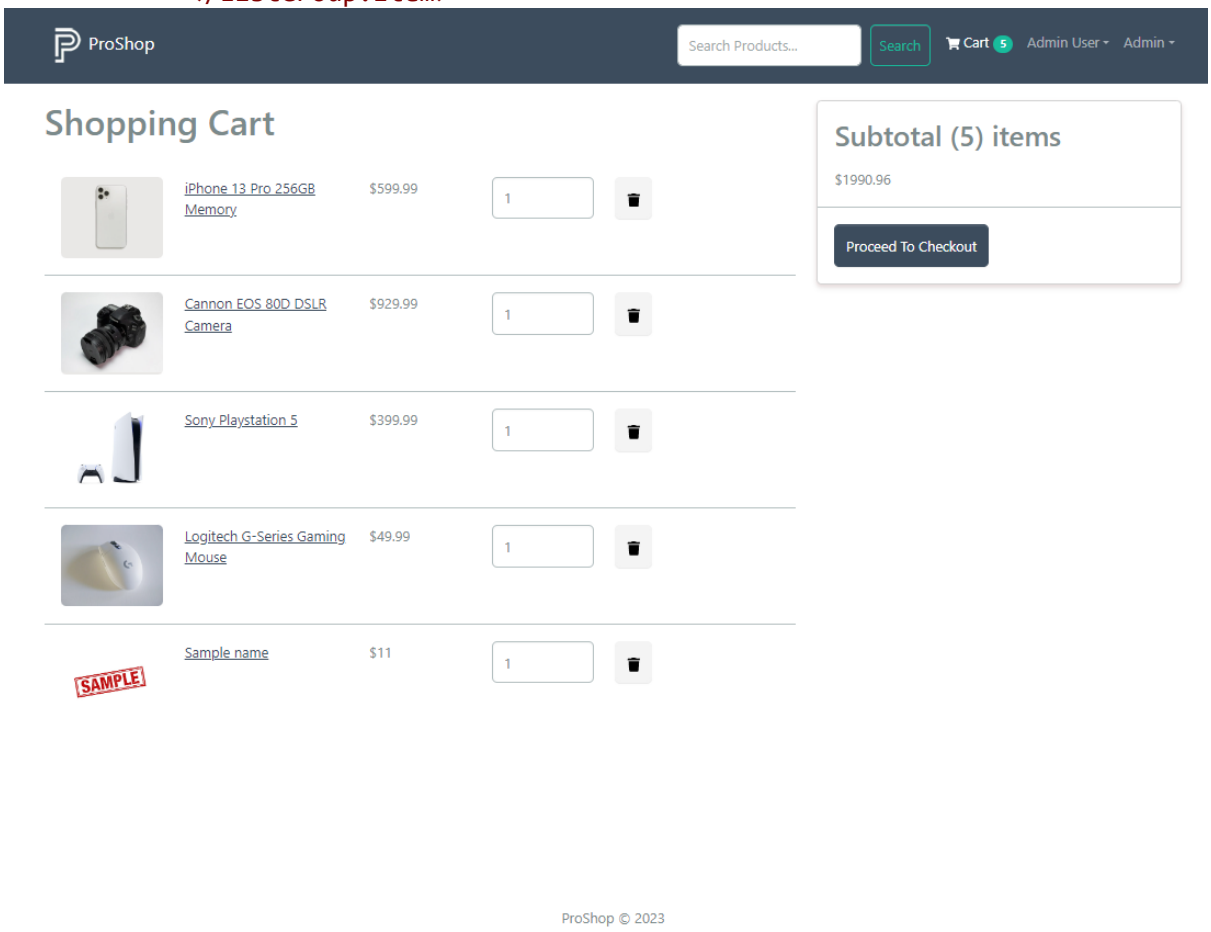

Рис. 3.10. Сторінка корзини

Сторінка оформення замовлення

```
<>
      <CheckoutSteps step1 step2 step3 step4 />
      <Row>
        <Col md={8}>
          <ListGroup variant='flush'>
            <ListGroup.Item>
              <h2>Shipping</h2>
              <p>
                <strong>Address:</strong>
                {cart.shippingAddress.address}, {cart.shippingAddress.city}{'
'}
                {cart.shippingAddress.postalCode},{' '}
                {cart.shippingAddress.country}
              </p>
            </ListGroup.Item>

            <ListGroup.Item>
              <h2>Payment Method</h2>
              <strong>Method: </strong>
```

```
            {cart.paymentMethod}
          </ListGroup.Item>

          <ListGroup.Item>
            <h2>Order Items</h2>
            {cart.cartItems.length === 0 ? (
              <Message>Your cart is empty</Message>
            ) : (
              <ListGroup variant='flush'>
                {cart.cartItems.map((item, index) => (
                  <ListGroup.Item key={index}>
                    <Row>
                      <Col md={1}>
                        <Image
                          src={item.image}
                          alt={item.name}
                          fluid
                          rounded
                        />
                      </Col>
                      <Col>
                        <Link to={`/product/${item.product}`}>
                          {item.name}
                        </Link>
                      </Col>
                      <Col md={4}>
                        {item.qty} x ${item.price} = $
                        {(item.qty * (item.price * 100)) / 100}
                      </Col>
                    </Row>
                  </ListGroup.Item>
                ))}
              </ListGroup>
            )}
          </ListGroup.Item>
        </ListGroup>
      </Col>
      <Col md={4}>
        <Card>
          <ListGroup variant='flush'>
            <ListGroup.Item>
              <h2>Order Summary</h2>
            </ListGroup.Item>
            <ListGroup.Item>
              <Row>
                <Col>Items</Col>
                <Col>${cart.itemsPrice}</Col>
              </Row>
            </ListGroup.Item>
```

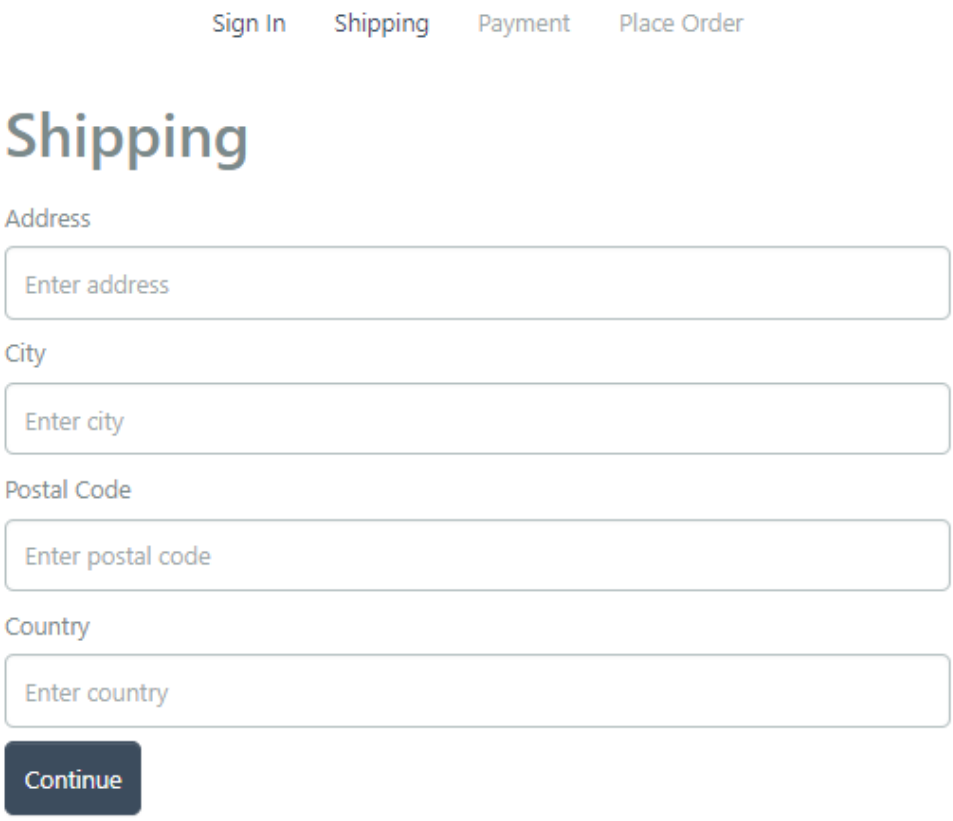

Рис. 3.11. Створення замовлення (інформацію про доставку)

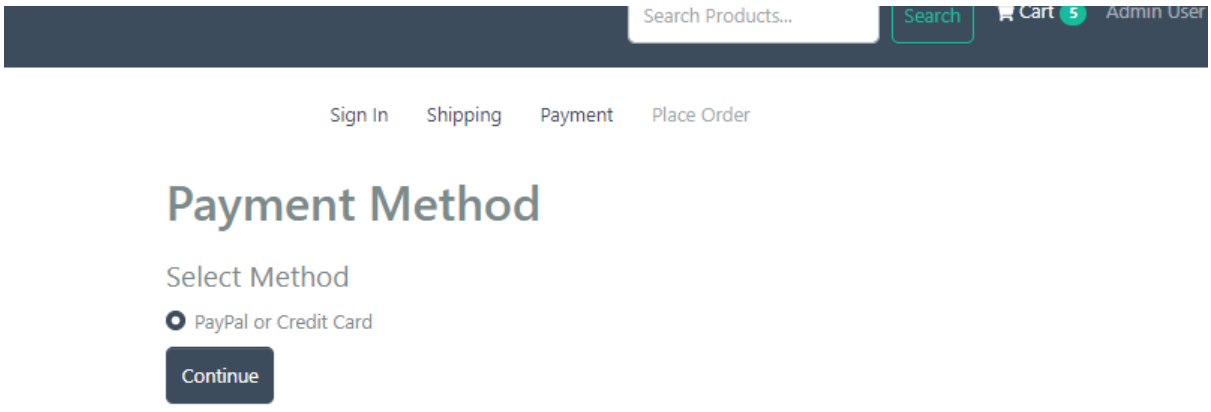

Рис. 3.12. Створення замовлення (оплата)

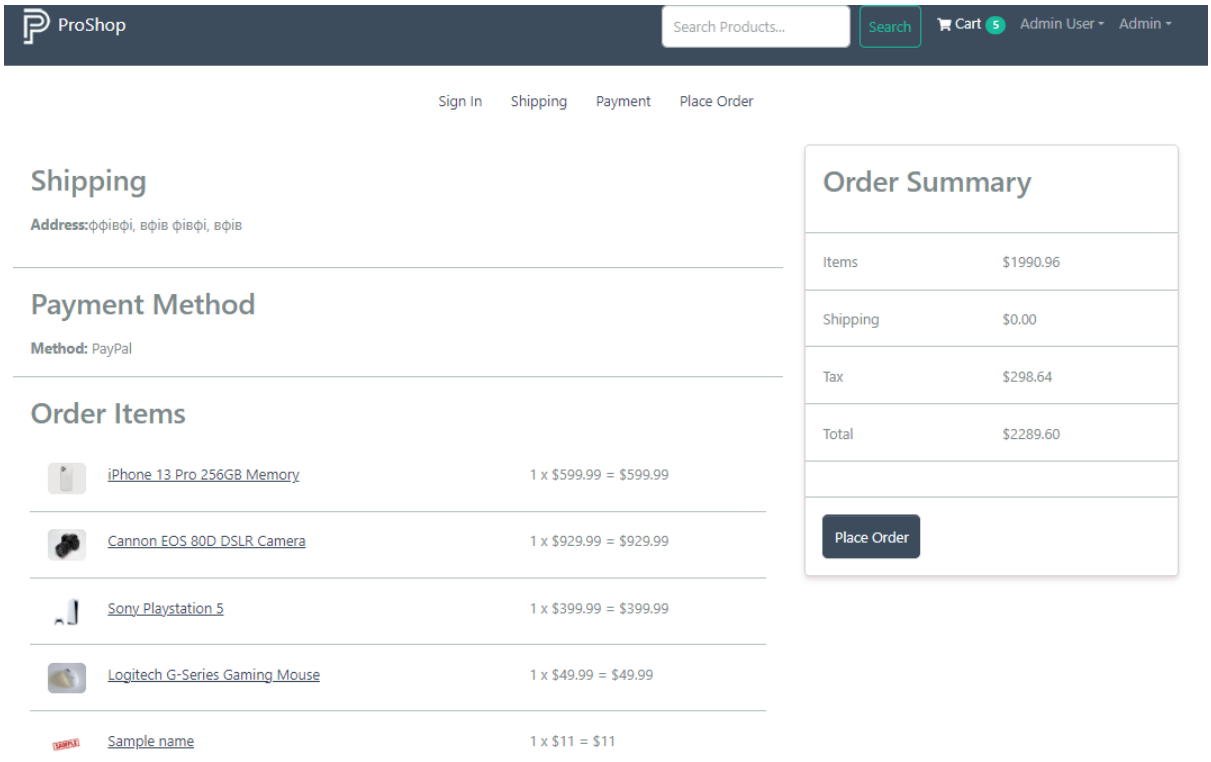

Рис. 3.13. Розміщення замовлення

Сторінка замовлення

```
<Loader />
 ) : error ? (
   <Message variant='danger'>{error.data.message}</Message>
 ) : (
   <>
     <h1>Order {order._id}</h1>
     <Row>
       <Col md={8}>
         <ListGroup variant='flush'>
           <ListGroup.Item>
             <h2>Shipping</h2>
             <p>
               <strong>Name: </strong> {order.user.name}
             </p>
             <p>
               <strong>Email: </strong>{' '}
               <a href={`mailto:${order.user.email}`}>{order.user.email}</a>
             </p>
             <p>
               <strong>Address:</strong>
```

```
                {order.shippingAddress.address},
{order.shippingAddress.city}{' '}
                {order.shippingAddress.postalCode},{' '}
                {order.shippingAddress.country}
              </p>
              {order.isDelivered ? (
                <Message variant='success'>
                  Delivered on {order.deliveredAt}
                </Message>
              ) : (
                <Message variant='danger'>Not Delivered</Message>
              )}
            </ListGroup.Item>

            <ListGroup.Item>
              <h2>Payment Method</h2>
              <p>
                <strong>Method: </strong>
                {order.paymentMethod}
              </p>
              {order.isPaid ? (
                <Message variant='success'>Paid on {order.paidAt}</Message>
              ) : (
                <Message variant='danger'>Not Paid</Message>
              )}
            </ListGroup.Item>

            <ListGroup.Item>
              <h2>Order Items</h2>
              {order.orderItems.length === 0 ? (
                <Message>Order is empty</Message>
              ) : (
                <ListGroup variant='flush'>
                  {order.orderItems.map((item, index) => (
                    <ListGroup.Item key={index}>
                      <Row>
                        <Col md={1}>
                          <Image
                            src={item.image}
                            alt={item.name}
                            fluid
                            rounded
                          />
                        </Col>
                        <Col>
                          <Link to={`/product/${item.product}`}>
                            {item.name}
                          </Link>
```

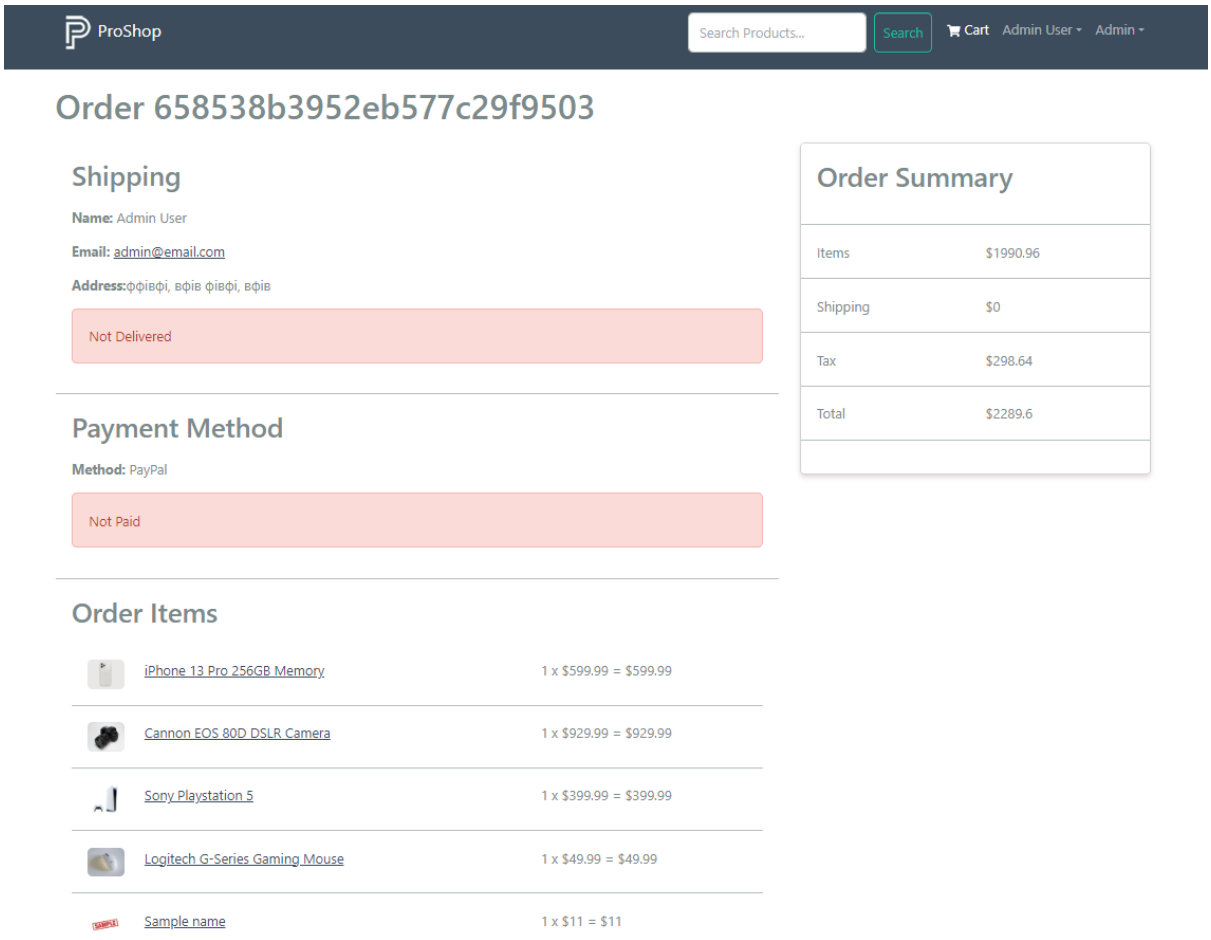

Рис. 3.14. Сторінка замовлення

Пошук товарів

Дуже важливим фактором є мати функціонал пошуку товарів, це значно підвищить комфорт використання інтренет магазину.

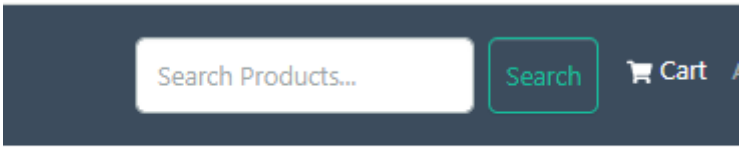

Рис. 3.15. Пошук товарів

Структура проекту

Детальна структура проекту виглядає наступним чином:

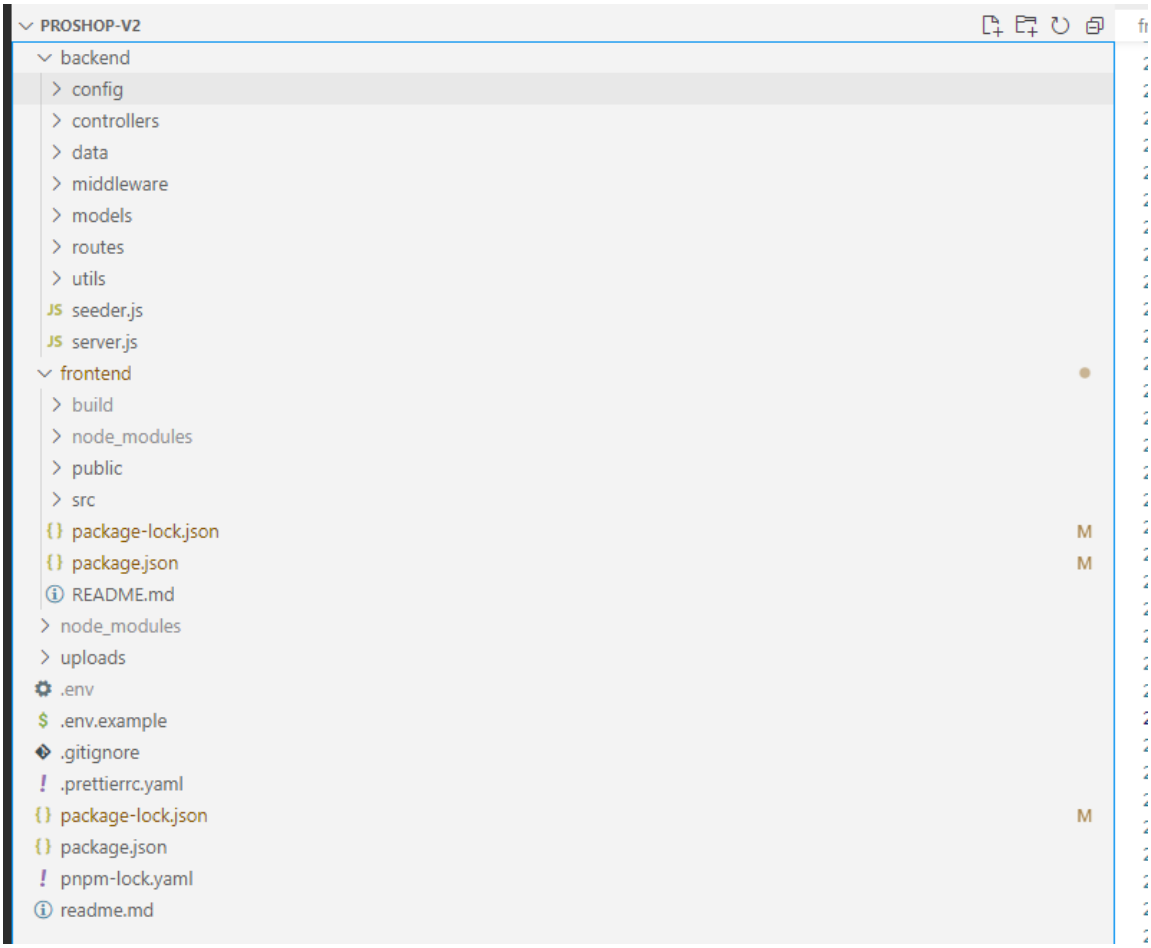

Рис. 3.16. Структура проекту

ВИСНОВКИ ДО РОЗДІЛУ 3

В даному розділі було описано структуру бази даних, стуктуру проекту, розроблено дизайн інтерфейсу.

Для розробки користувацького інтерфейсу було використано React та Redux, також CSS та HTML.

Було релізовано повноційний функціонал інтернет магазину, головна сторінка, сторінки реєстрації, товарів, створення замовлення, завантаження зображень та інше.

Також вексь функціонал був покритий юніт тестами та протестований вручну.

ВИСНОВКИ

В кваліфікаційній роботі було проведений детальний аналіз усіх сучасних типів баз даних, знайдено їх недоліки та обрано найбільш придатну для даного додатку базу даних. Так як різкий стрибок популярності NoSQL баз даних і пов'язані з ним історії використання нереляційних СУБД показали світу IT важливість реалістичної оцінки пріоритетів компанії. Деякі вендори успішно впровадили у себе NoSQL сховища і отримали помітне зниження збитків і підвищення якості їх додатків. Інші зазнали невдачі, пізно зрозумівши, що прийняте рішення їм не підходить. А треті просто залишилися зі своїми технологіями. Реляційні або нереляційні бази даних не єдиний вибір, який належить зробити компанії. Не менш важливим є і вибір між конкретними системами і конкретними стратегіями роботи з ними. У будь-якому випадку, NoSQL-революції не відбулося — реляційні бази даних утримують стабільно домінуючі позиції. Вони являють собою симбіоз надійності, функціональності і універсальності. При цьому багато NoSQL баз даних спрямовані на закриття абсолютно конкретних проблем SQL сховищ — у першу чергу на посилення горизонтальної масштабованості. Багато нереляційних баз даних відмінно працюють, виконуючи мету свого створення, але при цьому вони вже не є тим універсальним продуктом, яким є SQL.

Для клієнтської частини було обрано React, так як це комфортний, добре підтримуваний, простий в розумінні та написанні, й надзвичайно потужний, якщо старатись писати на ньому «правильно». Тобто, використовуючи класні тулзи, обов'язково Typescript, більш-менш «стандартизовані» й класно задокументовані бібліотеки (якщо це можна так назвати), й хороші підходи до структурування, по типу «Atomic Design by Bred Frost» розбавляючи це все юніт/e2e тестами (й обов'язково TDD, оскільки писати його з React — одне задоволення). Були описані функціональні та нефункціональні вимоги до застосунку.

Було описано принцип дії, переваги та недоліки архітектурних рішень на основі mongoDb.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Синявська О.О. Електронна торгівля в Україні: тенденції та перспективи розвитку. Вісник ХНУ імені В.Н. Каразіна. Серія «Міжнародні відносини. Економіка. Країнознавство. Туризм». 2019. Вип. 9. С. 126–132.

2. Startsev M.V. Electronic commerce as the way of the intensification of business processes. Соціально-економічні явища та процеси. 2011. № 5-6 (027-028). С. 212-215.

3. Zheng Qin. Introduction to E-commerce. Beijing: Tsinghua University Press, 2019. 517 c.

4. Model–view–controller [Електронний ресурс] – Режим доступу до ресурсу: https://en.wikipedia.org/wiki/Model-view-controller (дата звернення 12.05.2022)

5. eCommerce Usage Distribution in the Top 1 Million Sites [Електронний ресурс] – Режим доступу до ресурсу: https://trends.builtwith.com/shop (дата звернення 14.05.2022)

6. Adobe Commerce 2.4 Developer Guide. Architecture Guide. Magento architectural diagrams [Електронний ресурс] – Режим доступу до ресурсу: https://devdocs.magento.com/guides/v2.4/architecture/archi_perspectives/arch_diagrams.html (дата звернення 10.05.2022)

7. Shopify. For developers. Theme architecture. [Електронний ресурс] – Режим доступу до ресурсу: https://shopify.dev/themes/architecture (дата звернення 12.05.2022)

8. Keystone 6. [Електронний ресурс] – Режим доступу до ресурсу: https://keystonejs.com/ (дата звернення 11.05.2022)

9. Hiren Dhaduk. How to Build a Node.js Ecommerce App? [Електронний ресурс] – Режим доступу до ресурсу:

https://www.simform.com/blog/buildecommerce-web-application-node-js/ (дата звернення 14.05.2022)

12. Yarn package overview [Електронний ресурс] – Режим доступу до ресурсу: https://yarnpkg.com (дата звернення 11.05.2022)

13. Maciej Krawczyk. Build Your Own E-Commerce Keystone.js-Based System

— Environment Setup and Base Models [Електронний ресурс] – Режим доступу до ресурсу: https://betterprogramming.pub/build-your-own-ecommerce-keystone-js-based-system-environment-setup-and-base-models2c02a3e3a70b (дата звернення 18.05.2022)

14. Maciej Krawczyk. Building Your Own E-Commerce Keystone.js-Based System — Access Control [Електронний ресурс] – Режим доступу до ресурсу: https://betterprogramming.pub/building-your-own-e-commerce-keystone-jsbased-system-access-control-1a366ed7e064 (дата звернення 17.05.2022)

15. Спірінцев В.В.Web-технології та Web-дизайн: HTML, CSS [Текст]: навч. посіб. / В.В.Спірінцев, В.В.Гнатушенко, О.С.Волковський//Дніпро: «Ліра», 2017.-163с.

ДОДАТОК А

«API ДОКУМЕНТАЦІЯ»

```
import asyncHandler from '../middleware/asyncHandler.js';
import Order from '../models/orderModel.js';
import Product from '../models/productModel.js';
import { calcPrices } from '../utils/calcPrices.js';
import { verifyPayPalPayment, checkIfNewTransaction } from '../utils/paypal.js';

// @desc    Create new order
// @route   POST /api/orders
// @access  Private
const addOrderItems = asyncHandler(async (req, res) => {
  const { orderItems, shippingAddress, paymentMethod } = req.body;

  if (orderItems && orderItems.length === 0) {
    res.status(400);
    throw new Error('No order items');
  } else {
    // NOTE: here we must assume that the prices from our client are incorrect.
    // We must only trust the price of the item as it exists in
    // our DB. This prevents a user paying whatever they want by hacking our client
    // side code -
https://gist.github.com/bushblade/725780e6043eaf59415fbaf6ca7376ff

    // get the ordered items from our database
    const itemsFromDB = await Product.find({
      _id: { $in: orderItems.map((x) => x._id) },
    });

    // map over the order items and use the price from our items from database
    const dbOrderItems = orderItems.map((itemFromClient) => {
      const matchingItemFromDB = itemsFromDB.find(
        (itemFromDB) => itemFromDB._id.toString() === itemFromClient._id
      );
      return {
        ...itemFromClient,
        product: itemFromClient._id,
        price: matchingItemFromDB.price,
        _id: undefined,
      };
    });

    // calculate prices
    const { itemsPrice, taxPrice, shippingPrice, totalPrice } =
      calcPrices(dbOrderItems);

    const order = new Order({
      orderItems: dbOrderItems,
      user: req.user._id,
```

```
      shippingAddress,
      paymentMethod,
      itemsPrice,
      taxPrice,
      shippingPrice,
      totalPrice,
    });

    const createdOrder = await order.save();

    res.status(201).json(createdOrder);
  }
});

// @desc    Get logged in user orders
// @route   GET /api/orders/myorders
// @access  Private
const getMyOrders = asyncHandler(async (req, res) => {
  const orders = await Order.find({ user: req.user._id });
  res.json(orders);
});

// @desc    Get order by ID
// @route   GET /api/orders/:id
// @access  Private
const getOrderById = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id).populate(
    'user',
    'name email'
  );

  if (order) {
    res.json(order);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

// @desc    Update order to paid
// @route   PUT /api/orders/:id/pay
// @access  Private
const updateOrderToPaid = asyncHandler(async (req, res) => {
  // NOTE: here we need to verify the payment was made to PayPal before marking
  // the order as paid
  const { verified, value } = await verifyPayPalPayment(req.body.id);
  if (!verified) throw new Error('Payment not verified');

  // check if this transaction has been used before
  const isNewTransaction = await checkIfNewTransaction(Order, req.body.id);
```

```
  if (!isNewTransaction) throw new Error('Transaction has been used before');

  const order = await Order.findById(req.params.id);

  if (order) {
    // check the correct amount was paid
    const paidCorrectAmount = order.totalPrice.toString() === value;
    if (!paidCorrectAmount) throw new Error('Incorrect amount paid');

    order.isPaid = true;
    order.paidAt = Date.now();
    order.paymentResult = {
      id: req.body.id,
      status: req.body.status,
      update_time: req.body.update_time,
      email_address: req.body.payer.email_address,
    };

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

// @desc    Update order to delivered
// @route   GET /api/orders/:id/deliver
// @access  Private/Admin
const updateOrderToDelivered = asyncHandler(async (req, res) => {
  const order = await Order.findById(req.params.id);

  if (order) {
    order.isDelivered = true;
    order.deliveredAt = Date.now();

    const updatedOrder = await order.save();

    res.json(updatedOrder);
  } else {
    res.status(404);
    throw new Error('Order not found');
  }
});

// @desc    Get all orders
// @route   GET /api/orders
// @access  Private/Admin
const getOrders = asyncHandler(async (req, res) => {
```

```
  const orders = await Order.find({}).populate('user', 'id name');
  res.json(orders);
});

export {
  addOrderItems,
  getMyOrders,
  getOrderById,
  updateOrderToPaid,
  updateOrderToDelivered,
  getOrders,
};


import asyncHandler from '../middleware/asyncHandler.js';
import Product from '../models/productModel.js';

// @desc    Fetch all products
// @route   GET /api/products
// @access  Public
const getProducts = asyncHandler(async (req, res) => {
  const pageSize = process.env.PAGINATION_LIMIT;
  const page = Number(req.query.pageNumber) || 1;

  const keyword = req.query.keyword
    ? {
        name: {
          $regex: req.query.keyword,
          $options: 'i',
        },
      }
    : {};

  const count = await Product.countDocuments({ ...keyword });
  const products = await Product.find({ ...keyword })
    .limit(pageSize)
    .skip(pageSize * (page - 1));

  res.json({ products, page, pages: Math.ceil(count / pageSize) });
});

// @desc    Fetch single product
// @route   GET /api/products/:id
// @access  Public
const getProductById = asyncHandler(async (req, res) => {
  // NOTE: checking for valid ObjectId to prevent CastError moved to separate
  // middleware. See README for more info.

  const product = await Product.findById(req.params.id);
  if (product) {
```

```
    return res.json(product);
  } else {
    // NOTE: this will run if a valid ObjectId but no product was found
    // i.e. product may be null
    res.status(404);
    throw new Error('Product not found');
  }
});

// @desc    Create a product
// @route   POST /api/products
// @access  Private/Admin
const createProduct = asyncHandler(async (req, res) => {
  const product = new Product({
    name: 'Sample name',
    price: 0,
    user: req.user._id,
    image: '/images/sample.jpg',
    brand: 'Sample brand',
    category: 'Sample category',
    countInStock: 0,
    numReviews: 0,
    description: 'Sample description',
  });

  const createdProduct = await product.save();
  res.status(201).json(createdProduct);
});

// @desc    Update a product
// @route   PUT /api/products/:id
// @access  Private/Admin
const updateProduct = asyncHandler(async (req, res) => {
  const { name, price, description, image, brand, category, countInStock } =
    req.body;

  const product = await Product.findById(req.params.id);

  if (product) {
    product.name = name;
    product.price = price;
    product.description = description;
    product.image = image;
    product.brand = brand;
    product.category = category;
    product.countInStock = countInStock;

    const updatedProduct = await product.save();
    res.json(updatedProduct);
  } else {
```

```
    res.status(404);
    throw new Error('Product not found');
  }
});

// @desc    Delete a product
// @route   DELETE /api/products/:id
// @access  Private/Admin
const deleteProduct = asyncHandler(async (req, res) => {
  const product = await Product.findById(req.params.id);

  if (product) {
    await Product.deleteOne({ _id: product._id });
    res.json({ message: 'Product removed' });
  } else {
    res.status(404);
    throw new Error('Product not found');
  }
});

// @desc    Create new review
// @route   POST /api/products/:id/reviews
// @access  Private
const createProductReview = asyncHandler(async (req, res) => {
  const { rating, comment } = req.body;

  const product = await Product.findById(req.params.id);

  if (product) {
    const alreadyReviewed = product.reviews.find(
      (r) => r.user.toString() === req.user._id.toString()
    );

    if (alreadyReviewed) {
      res.status(400);
      throw new Error('Product already reviewed');
    }

    const review = {
      name: req.user.name,
      rating: Number(rating),
      comment,
      user: req.user._id,
    };

    product.reviews.push(review);

    product.numReviews = product.reviews.length;

    product.rating =
```

```
      product.reviews.reduce((acc, item) => item.rating + acc, 0) /
      product.reviews.length;

    await product.save();
    res.status(201).json({ message: 'Review added' });
  } else {
    res.status(404);
    throw new Error('Product not found');
  }
});

// @desc    Get top rated products
// @route   GET /api/products/top
// @access  Public
const getTopProducts = asyncHandler(async (req, res) => {
  const products = await Product.find({}).sort({ rating: -1 }).limit(3);

  res.json(products);
});

export {
  getProducts,
  getProductById,
  createProduct,
  updateProduct,
  deleteProduct,
  createProductReview,
  getTopProducts,
};


import asyncHandler from '../middleware/asyncHandler.js';
import generateToken from '../utils/generateToken.js';
import User from '../models/userModel.js';

// @desc    Auth user & get token
// @route   POST /api/users/auth
// @access  Public
const authUser = asyncHandler(async (req, res) => {
  const { email, password } = req.body;

  const user = await User.findOne({ email });

  if (user && (await user.matchPassword(password))) {
    generateToken(res, user._id);

    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
```

```
      isAdmin: user.isAdmin,
    });
  } else {
    res.status(401);
    throw new Error('Invalid email or password');
  }
});

// @desc    Register a new user
// @route   POST /api/users
// @access  Public
const registerUser = asyncHandler(async (req, res) => {
  const { name, email, password } = req.body;

  const userExists = await User.findOne({ email });

  if (userExists) {
    res.status(400);
    throw new Error('User already exists');
  }

  const user = await User.create({
    name,
    email,
    password,
  });

  if (user) {
    generateToken(res, user._id);

    res.status(201).json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
    });
  } else {
    res.status(400);
    throw new Error('Invalid user data');
  }
});

// @desc    Logout user / clear cookie
// @route   POST /api/users/logout
// @access  Public
const logoutUser = (req, res) => {
  res.cookie('jwt', '', {
    httpOnly: true,
    expires: new Date(0),
  });
```

```
  res.status(200).json({ message: 'Logged out successfully' });
};

// @desc    Get user profile
// @route   GET /api/users/profile
// @access  Private
const getUserProfile = asyncHandler(async (req, res) => {
  const user = await User.findById(req.user._id);

  if (user) {
    res.json({
      _id: user._id,
      name: user.name,
      email: user.email,
      isAdmin: user.isAdmin,
    });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});

// @desc    Update user profile
// @route   PUT /api/users/profile
// @access  Private
const updateUserProfile = asyncHandler(async (req, res) => {
  const user = await User.findById(req.user._id);

  if (user) {
    user.name = req.body.name || user.name;
    user.email = req.body.email || user.email;

    if (req.body.password) {
      user.password = req.body.password;
    }

    const updatedUser = await user.save();

    res.json({
      _id: updatedUser._id,
      name: updatedUser.name,
      email: updatedUser.email,
      isAdmin: updatedUser.isAdmin,
    });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});
```

```
// @desc    Get all users
// @route   GET /api/users
// @access  Private/Admin
const getUsers = asyncHandler(async (req, res) => {
  const users = await User.find({});
  res.json(users);
});

// @desc    Delete user
// @route   DELETE /api/users/:id
// @access  Private/Admin
const deleteUser = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id);

  if (user) {
    if (user.isAdmin) {
      res.status(400);
      throw new Error('Can not delete admin user');
    }
    await User.deleteOne({ _id: user._id });
    res.json({ message: 'User removed' });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});

// @desc    Get user by ID
// @route   GET /api/users/:id
// @access  Private/Admin
const getUserById = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id).select('-password');

  if (user) {
    res.json(user);
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});
// @desc    Update user
// @route   PUT /api/users/:id
// @access  Private/Admin
const updateUser = asyncHandler(async (req, res) => {
  const user = await User.findById(req.params.id);

  if (user) {
    user.name = req.body.name || user.name;
    user.email = req.body.email || user.email;
    user.isAdmin = Boolean(req.body.isAdmin);
```

```
    const updatedUser = await user.save();

    res.json({
      _id: updatedUser._id,
      name: updatedUser.name,
      email: updatedUser.email,
      isAdmin: updatedUser.isAdmin,
    });
  } else {
    res.status(404);
    throw new Error('User not found');
  }
});

export {
  authUser,
  registerUser,
  logoutUser,
  getUserProfile,
  updateUserProfile,
  getUsers,
  deleteUser,
  getUserById,
  updateUser,
};
```