МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ВЕБСАЙТУ ПІЦЕРІЇ З ВИКОРИСТАННЯМ REACT ТА NODE.JS»**

Виконав: студент 4 курсу, групи 6.1211-1пі
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми програмна інженерія
(назва освітньої програми)
Я.В. Фоменко
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, PhD Столярова А.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент доцент кафедри комп'ютерних наук, доцент, к.т.н. Решевська К.С.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти бакалавр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма програмна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Фоменку Ярославу Володимировичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка вебсайту піцерії з використанням React та Node.js

керівник роботи Столярова Анастасія Валеріївна, PhD
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 20 » грудня 2024 року № 2001-с

2. Строк подання студентом роботи 02.06.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі та аналіз вимог.
2. Проєктування вебзастосунку.
3. Реалізація вебзастосунку.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 26.12.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	13.01.2025	
2.	Збір вихідних даних.	10.02.2025	
3.	Обробка методичних та теоретичних	03.03.2025	
	джерел.		
4.	Розробка першого та другого розділу.	14.04.2025	
5.	Розробка третього розділу.	19.05.2025	
6.	Оформлення та нормоконтроль	26.05.2025	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	17.06.2025	

Студент _____ Я.В. Фоменко
(підпис) (ініціали та прізвище)

Керівник роботи _____ А.В. Столярова
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота бакалавра «Розробка вебсайту піцерії з використанням React та Node.js»: 51 с., 27 рис., 10 джерел.

БАЗА ДАНИХ, ІНТЕРФЕЙС КОРИСТУВАЧА, КОРЗИНА ЗАМОВЛЕНЬ, ПІЦЕРІЯ, NODE.JS, POSTGRESQL, REACT COMPONENT.

Об’єкт дослідження – процес створення сучасних вебзастосунків для закладів громадського харчування.

Предмет дослідження – технології розробки вебсервісів із застосуванням бібліотеки React для клієнтської частини та середовища Node.js для серверної частини.

Мета роботи – створення функціонального вебзастосунку для обслуговування замовлень у сфері громадського харчування, з використанням сучасних вебтехнологій клієнтської та серверної розробки.

Методи дослідження – моделювання, проєктування програмного забезпечення, порівняльний аналіз технологій.

З огляду на постійне зростання популярності онлайн-замовлень їжі, створення адаптивного та функціонального вебсайту стає актуальним завданням для бізнесу у сфері громадського харчування. У кваліфікаційній роботі представлено процес розробки вебсайту для піцерії з акцентом на зручність користувача та ефективність обробки замовлень.

Вебсервіс розроблено з використанням бібліотеки React для клієнтської частини та Node.js для серверної логіки. Для зберігання інформації про замовлення та меню використано систему керування базами даних PostgreSQL.

SUMMARY

Bachelor’s qualifying paper “Development of a Pizzeria Website Using React and Node.js”: 51 pages, 27 figures, 10 references.

DATABASE, USER INTERFACE, ORDER CART, PIZZERIA, NODE.JS, POSTGRESQL, REACT COMPONENT.

The object of the study is the process of developing modern web applications for food service establishments.

The subject of the study is the use of technologies for building web services with the React library on the client side and the Node.js runtime on the server side.

The aim of the study is to develop a functional web application for managing orders in the food service industry using modern web technologies for both client-side and server-side development.

The research methods include modeling, software design, and comparative analysis of technologies.

Given the growing popularity of online food ordering, creating an adaptive and functional website has become a critical task for businesses in the food service sector. This qualification paper presents the process of developing a website for a pizzeria, with a focus on user convenience and efficient order processing.

The web service was developed using the React library for the frontend and Node.js for the backend logic. The PostgreSQL database management system was used to store data related to orders and the menu.

ЗМІСТ

Завдання на кваліфікаційну роботу……..2
Реферат 4
Summary 5
Вступ..... 7
1 Аналіз вимог та планування проєкту 9
1.1 Аналіз вимог замовника 9
1.2 Визначення функціональних та нефункціональних вимог..... 10
1.3 Бібліотека React та її можливості 12
1.4 Середовище Node.js і каркас Express.js та їхні можливості 15
1.5 Аналіз існуючих альтернативних рішень на ринку..... 17
1.6 Висновки до розділу 1 19
2 Проєктування вебзастосунку 21
2.1 Проєктування архітектури вебзастосунку..... 21
2.2 Розробка користувацького інтерфейсу 26
2.3 Налаштування серверної частини вебзастосунку..... 29
2.4 Проєктування структури БД з використанням ER-діаграми..... 30
2.5 Висновки до розділу 2 33
3 Реалізація вебзастосунка 34
3.1 Розробка клієнтської частини з використанням бібліотеки React 34
3.2 Розробка серверної частини з використанням Node.js..... 41
3.3 Тестування вебзастосунку..... 45
3.4 Висновки до розділу 3 49
Висновки 50
Перелік посилань..... 51

ВСТУП

Розвиток інформаційних технологій значно впливає на бізнес, особливо у сфері обслуговування та громадського харчування. З кожним роком популярність онлайн-замовлень їжі в закладах швидкого харчування стрімко зростає завдяки зручності, швидкості та доступності для кінцевого користувача. Раніше більшість замовлень здійснювалися переважно за телефоном або безпосередньо у закладі, що вимагало значних часових витрат як від персоналу, так і від клієнтів. Нині ж вебзастосунки дозволяють значно спростити цей процес, автоматизувати прийом замовлень, покращити взаємодію з клієнтами та оптимізувати роботу закладів харчування, що й обумовлює **актуальність** проєкту.

Мета роботи – створення функціонального вебзастосунку для обслуговування замовлень у сфері громадського харчування, з використанням сучасних вебтехнологій клієнтської та серверної розробки.

Досягнення поставленої мети передбачає реалізацію наступних завдань:

- проаналізувати вимоги до сучасних вебсайтів у сфері громадського харчування;
- дослідити особливості використання React та Node.js для створення вебзастосунків;
- розробити архітектуру та дизайн вебсайту піцерії;
- спроєктувати та реалізувати клієнтську та серверну частини вебзастосунку;
- протестувати розроблений вебсайт на предмет відповідності поставленим вимогам та ефективності його роботи.

Об’єкт дослідження – процес створення сучасних вебзастосунків для закладів громадського харчування.

Предмет дослідження – технології розробки вебсервісів із застосуванням бібліотеки React для клієнтської частини та середовища Node.js для серверної частини.

Методи дослідження – моделювання, проєктування програмного забезпечення, порівняльний аналіз технологій.

Кваліфікаційна робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. У першому розділі розглянуто основні сучасні технології розробки вебсайтів, зокрема бібліотеку React для створення клієнтської частини та серверне середовище Node.js для реалізації бекенду. Другий розділ присвячено проєктуванню архітектури вебсайту піцерії, опису його основних компонентів, структури бази даних PostgreSQL та взаємодії між клієнтською і серверною частинами. У третьому розділі наведено опис практичної реалізації функціоналу вебсайту та тестування.

1 АНАЛІЗ ВИМОГ ТА ПЛАНУВАННЯ ПРОЄКТУ

1.1 Аналіз вимог замовника

Замовник висловив потребу у створенні вебдодатку для закладу громадського харчування. Метою проєкту є розробка зручного та функціонального вебзастосунку піцерії, що відповідає очікуванням кінцевих користувачів та забезпечує комфортну взаємодію з системою. У рамках реалізації передбачається створення наступних функціональних сторінок:

- **сторінки реєстрації та автентифікації**: можливість реєстрації та входу користувачів;
- **головна сторінка**: повний перелік доступних піц із можливістю фільтрації;
- **кошик замовлень**: перегляд обраних позицій, редагування кількості товарів, оформлення замовлення;
- **профіль користувача**: перегляд історії замовлень, та можливість коригування контактних даних.

Крім загальних вимог, було сформульовано і низку конкретних технічних характеристик окремих компонентів системи:

- **кошик замовлень**: можливість динамічно змінювати кількість піц, автоматичний перерахунок загальної вартості, зручний перехід до етапу оформлення;
- **оформлення замовлення**: валідація всіх обов’язкових полів, збереження історії замовлень для авторизованих користувачів;
- **реєстрація користувача**: перевірка правильності введених даних, шифрування пароля перед збереженням у базі даних.

Виявлені побажання щодо функціональності, зручності використання та технічних характеристик системи лягли в основу структурування вимог, що визначають, які саме можливості має реалізовувати система, а також якісні

показники, яким вона повинна відповідати. Деталізований опис цих вимог представлено у наступному розділі.

1.2 Визначення функціональних та нефункціональних вимог

Усі вимоги до програмного продукту умовно можна поділити на дві категорії: функціональні та нефункціональні. Функціональні вимоги визначають конкретні можливості, які система повинна реалізовувати, відповідаючи на питання «що саме має робити застосунок?». Нефункціональні вимоги, у свою чергу, визначають якісні характеристики програмного забезпечення, описують рівень безпеки, продуктивності та інші особливості, відповідаючи на питання «як має працювати застосунок?».

Функціональні вимоги формулюють конкретні функції та можливості системи, які необхідно реалізувати для досягнення поставленої мети. Вони описують, як система повинна поводитися в різних сценаріях використання, які дії повинна підтримувати, і як вона має взаємодіяти з користувачами та іншими компонентами.

Основні характеристики функціональних вимог:

- точність: функціональні вимоги повинні бути конкретними та однозначними, без зайвої інтерпретації;
- гнучкість: з урахуванням змін в процесі розробки, вимоги можуть уточнюватися та доповнюватися;
- орієнтація на користувача: вимоги мають відображати реальні потреби та очікування кінцевих користувачів;
- первірюваність: виконання кожної функціональної вимоги повинно бути можливо підтвердити через тестування.

Отже, виокремимо основні **функціональні вимоги** до розроблюваного вебзастосунку:

- реєстрація користувачів із валідацією пароля за певними правилами;
- реєстрація та автентифікація користувачів;

– створення та обробка замовлень через кошик із можливістю редагування вмісту;
– можливість фільтрації та пошуку піц за категоріями;
– ведення історії замовлень для зареєстрованих користувачів.

Функціональні вимоги створюють основу для технічної реалізації проєкту і є безпосереднім орієнтиром для команди розробників.

Нефункціональні вимоги визначають якісні характеристики вебзастосунку, що впливають на його стабільність, безпеку, продуктивність і подальшу масштабованість. На відміну від функціональних вимог, які легко виражаються через функції або процеси, нефункціональні вимоги відображають загальні стандарти роботи системи.

Основні характеристики нефункціональних вимог:

– стабільність: залишаються відносно незмінними протягом життєвого циклу проєкту;
– глобальність: стосуються усієї системи загалом, а не окремих функцій;
– якісна орієнтація: фокусуються на характеристиках системи, а не на її поведінці;
– можливість перевірки: деякі з них важче виміряти, але їх відповідність можна протестувати за допомогою спеціальних методик.

Отже, виокремимо основні **нефункціональні вимоги** до розроблюваного вебзастосунку:

– продуктивність: час завантаження будь-якої сторінки сайту не повинен перевищувати 2-3 секунди навіть при великому навантаженні одночасних користувачів;
– адаптивність: коректне відображення сайту на різних пристроях – смартфонах, планшетах, настільних комп’ютерах;
– масштабованість: можливість розширення системи для обслуговування збільшеної кількості користувачів без необхідності перепроєктування архітектури;

– надійність: система повинна бути стійкою до відмов.

Таким чином, правильне визначення функціональних та нефункціональних вимог має вирішальне значення для успішної розробки вебзастосунку. Функціональні вимоги гарантують відповідність очікуванням користувачів, а нефункціональні – якість роботи, безпеку та надійність системи в умовах реального використання. Їх чітке формулювання на етапі планування дозволяє мінімізувати ризики, полегшити процес розробки.

З урахуванням сформульованих функціональних та нефункціональних вимог, наступним етапом є вибір оптимальних інструментів і технологій, що дозволять ефективно реалізувати поставлені завдання.

1.3 Бібліотека React та її можливості

React – це бібліотека JavaScript, створена та підтримувана компанією Facebook (нині Meta), яка вперше була представлена у 2013 році. Основною метою її створення було спрощення розробки інтерфейсів, які швидко реагують на зміни даних та дозволяють ефективно управляти станом додатка. На відміну від багатьох інших рішень, які пропонували повноцінні фреймворки, React сконцентрувався виключно на рівні представлення – створенні компонентів, що відповідають за вигляд та поведінку інтерфейсу.

Однією з ключових концепцій React є використання компонентного підходу. Замість того, щоб розробляти монолітний інтерфейс, у React інтерфейс розбивається на невеликі ізольовані частини – компоненти [4]. Кожен компонент управляє власним станом та логікою і може бути легко повторно використаний у різних частинах програми. Компоненти у React можуть бути як простими функціями, що повертають опис візуальної частини (JSX-розмітку), так і більш складними класами, які володіють додатковими методами життєвого циклу.

Ще однією важливою особливістю є використання віртуального DOM (Virtual DOM) [9]. У традиційній веброзробці будь-яка зміна в інтерфейсі

вимагала прямого маніпулювання реальним DOM-деревом браузера, що було повільним і ресурсозатратним процесом. React вирішив цю проблему шляхом створення копії DOM в оперативній пам’яті, яка оновлюється швидше та мінімізує кількість реальних змін у браузері. Коли дані змінюються, React оновлює спочатку віртуальний DOM, порівнює його зі старим варіантом і визначає найменший набір змін, які потрібно внести до реального DOM. Такий підхід значно підвищує продуктивність додатків і забезпечує плавний досвід взаємодії для користувачів. На рисунку 1.1 можна спостерігати схематичну роботу віртуального DOM.

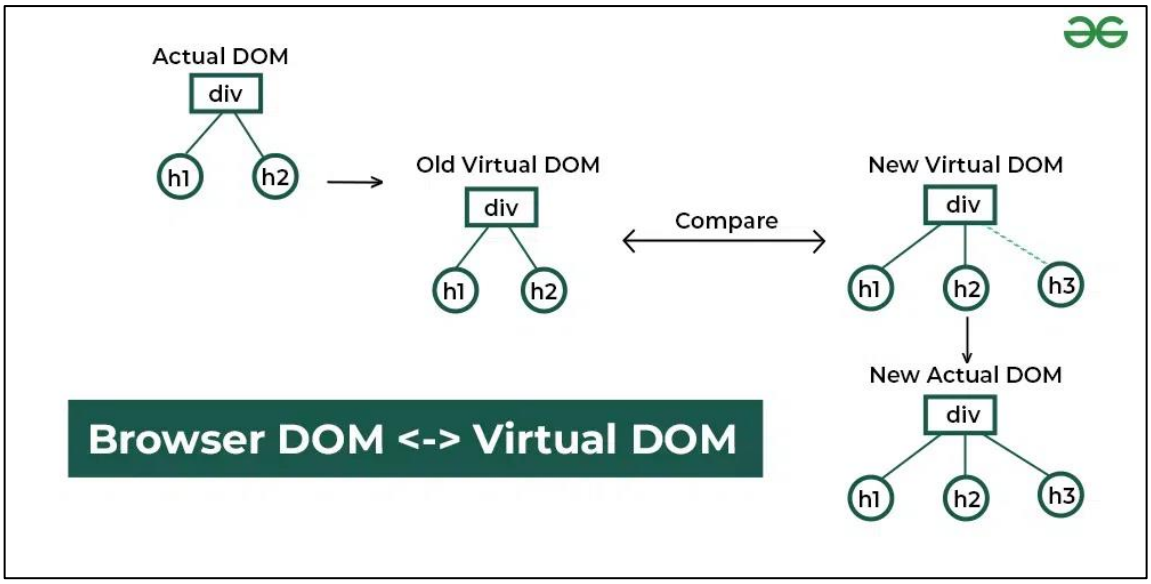


Рисунок 1.1 – Схематичне представлення роботи віртуального DOM [8]

React також активно використовує концепцію одностороннього потоку даних (one-way data flow). Це означає, що дані передаються від батьківських компонентів до дочірніх через властивості (props) [7], що дозволяє краще контролювати зміну стану програми та уникати неочікуваних ефектів. Для управління складними станами більших додатків часто використовують спеціальні бібліотеки на кшталт Redux, MobX або Context API, що дозволяють централізовано зберігати та змінювати стан.

Що стосується можливостей, React відкриває широкий спектр функціоналу для розробників. Серед них – можливість легко створювати

універсальні додатки, що працюють як на клієнтській, так і на серверній стороні. Завдяки бібліотеці Next.js [10], що базується на React, стало можливим серверний рендеринг (SSR) – процес створення HTML-контенту на сервері для подальшого його відправлення на клієнт. Це сприяє покращенню SEO-оптимізації додатків та прискорює перше завантаження сторінок. Також React має потужну екосистему додаткових інструментів і бібліотек. Наприклад, для побудови складної маршрутизації використовується React Router. Для стилізації компонентів пропонуються рішення типу styled-components або Emotion, які дозволяють писати CSS безпосередньо у коді JavaScript. Для тестування існують спеціалізовані інструменти, такі як Jest та React Testing Library, що спрощують процес створення надійних тестів.

Ще одна особливість, що заслуговує на увагу – це hooks (хуки) [3]. Запроваджені у React 16.8, хуки дозволяють використовувати стан та інші можливості React у функціональних компонентах, без необхідності створювати класи. Наприклад, за допомогою хука useState можна легко додати внутрішній стан у компонент, а useEffect дозволяє виконувати побічні ефекти, такі як отримання даних з сервера або підписка на події. Хуки сприяли спрощенню архітектури додатків, зробивши код більш чистим і зрозумілим.

Особливу увагу варто приділити і підходу до розробки у React, який стимулює створення декларативних інтерфейсів. Замість того, щоб явно описувати послідовність змін, які потрібно внести у DOM, розробник описує бажаний результат у термінах стану додатка, а React самостійно вирішує, як його досягти. Такий підхід полегшує розробку і підтримку великих проєктів, зменшує кількість помилок і підвищує передбачуваність роботи додатків.

Щодо гнучкості, React легко інтегрується з іншими бібліотеками чи навіть з наявним кодом. Це дозволяє поступово впроваджувати React у вже існуючі проєкти без необхідності повної переробки. Крім того, React не диктує суворих правил щодо структури проєкту або організації коду, надаючи розробникам свободу у виборі архітектурних рішень відповідно до потреб проєкту.

React також активно розвивається та підтримується спільнотою. Постійні оновлення бібліотеки, випуск нових функціональних можливостей і розширення офіційної документації свідчать про високий рівень підтримки та актуальності цього інструменту у сучасній веброзробці. Крім Meta, над розвитком екосистеми React працюють тисячі ентузіастів і компаній з усього світу.

Таким чином, бібліотека React є потужним і гнучким інструментом для створення сучасних, масштабованих і ефективних вебдодатків. Її філософія компонентного підходу, використання віртуального DOM, односторонній потік даних, а також підтримка декларативного стилю програмування роблять її одним із найкращих виборів для розробників, що прагнуть створювати якісні, підтримувані і продуктивні інтерфейси.

1.4 Середовище Node.js і каркас Express.js та їхні можливості

Node.js є відкритим середовищем виконання JavaScript-коду на стороні сервера, яке дозволяє створювати масштабовані, високопродуктивні та ефективні серверні додатки. На відміну від традиційних середовищ, таких як PHP або Java EE, де кожен запит обробляється окремим потоком, Node.js використовує неблокуючу модель введення-виведення, що дозволяє обробляти тисячі одночасних з’єднань без створення великої кількості потоків. Це стало можливим завдяки використанню подієво-орієнтованої архітектури та механізму асинхронного виконання. Node.js базується на рушії V8 від Google, який компілює JavaScript у високоефективний машинний код. Завдяки цьому додатки, написані на Node.js, мають високу швидкодію. Крім того, Node.js має вбудований менеджер пакетів – npm (Node Package Manager), що дозволяє розробникам легко встановлювати, оновлювати та управляти зовнішніми бібліотеками і модулями. Спільнота розробників для Node.js є однією з найбільших у світі, що сприяє постійному розвитку середовища та великій кількості доступних пакетів для найрізноманітніших завдань.

Особливістю Node.js є концепція однониткового циклу подій (event loop). У той час як класичні сервери створюють новий потік для кожного підключення, Node.js використовує один потік для обробки всіх запитів, перемикаючись між завданнями у відповідь на події. Завдяки цьому споживання ресурсів суттєво зменшується, що дозволяє запускати масштабовані системи навіть на обмежених обчислювальних потужностях. Проте для обчислювально важких задач, які можуть блокувати головний потік, Node.js надає можливість використання воркерів або розподілення навантаження через кластери.

Node.js особливо добре підходить для розробки серверів реального часу, таких як чати, онлайн-ігри або платформи для потокової передачі даних. Його також часто використовують для створення RESTful API, мікросервісних архітектур та серверних частин вебдодатків, що працюють із великим навантаженням на обробку запитів. Однак Node.js сам по собі надає досить низькорівневий інтерфейс для роботи із запитами та відповідями HTTP, що іноді ускладнює створення повноцінних серверних додатків. Для полегшення цього процесу був розроблений програмний каркас Express.js.

Express.js є мінімалістичним, гнучким і потужним каркасом для Node.js, який суттєво спрощує розробку серверних додатків і API. Він надає набір інструментів для роботи з маршрутами, обробкою запитів та відповідей, підтримкою шаблонізаторів, а також для реалізації проміжного програмного забезпечення (middleware).

Однією з ключових концепцій Express.js є використання проміжних обробників – функцій, які мають доступ до об’єктів запиту, відповіді та наступної функції в циклі обробки запитів. Це дозволяє легко впроваджувати автентифікацію, логування, обробку помилок та інші аспекти без дублювання коду. Також він надає організовувати маршрутизацію, дозволяючи об’єднувати обробники за шляхами URL і HTTP-методами. Це значно спрощує створення API, особливо у великих проєктах, де структура запитів може бути складною і багаторівневою. Для спрощення роботи з даними,

отриманими від клієнта, Express.js підтримує інтеграцію з різними парсерами, такими як body-parser, що дозволяє обробляти JSON- та URL-encoded дані. Ще однією важливою можливістю Express.js є підтримка шаблонізаторів для динамічного формування HTML-сторінок на сервері. Завдяки цьому можна розділяти бізнес-логіку та представлення, що підвищує підтримуваність коду. Серед популярних шаблонізаторів, які часто використовуються разом з Express.js, варто згадати EJS, Pug та Handlebars. Express.js також чудово інтегрується з іншими технологіями, такими як бази даних MongoDB, PostgreSQL або Redis, системи автентифікації типу OAuth2 або JWT, а також бібліотеки для обробки вебсокетів, як-от socket.io.

Завдяки своїй легкості, невеликій кількості залежностей і високій швидкодії Express.js забезпечує швидкий старт розробки, при цьому не обмежуючи можливості розширення проєкту у майбутньому. Багатий вибір додаткових бібліотек, що розширюють функціональність Express.js, дозволяє розробникам легко інтегрувати необхідні компоненти.

Отже, комбінація Node.js та Express.js є потужним інструментом для створення сучасних серверних додатків. Node.js забезпечує основу у вигляді високопродуктивного та масштабованого середовища виконання, а Express.js надає зручний інструментарій для організації маршрутизації, обробки запитів і побудови структурованих вебдодатків. Завдяки широким можливостям інтеграції, підтримці масштабування та величезній спільноті, ці технології стали основою для розробки тисяч комерційних, наукових та соціальних проєктів у всьому світі.

1.5 Аналіз існуючих альтернативних рішень на ринку

Розглядаючи альтернативи до зв’язки React, Node.js та Express.js, варто зазначити, що на ринку існує багато інших рішень, кожне з яких має свої переваги, недоліки і специфіку використання. Вибір технологічного стека

часто залежить від завдань проєкту, вимог до продуктивності, масштабованості, безпеки, а також від кваліфікації команди розробників.

Одним з конкурентів для React є бібліотека Vue.js. Vue – це прогресивний JavaScript-фреймворк, що орієнтований на спрощення створення інтерфейсів користувача. Він був розроблений таким чином, щоб забезпечити гнучкість і простоту інтеграції у вже існуючі проєкти. Завдяки своїй легкості та мінімалістичності, Vue швидко завоював популярність. Його екосистема, яка включає Vue Router для маршрутизації та Vuex для управління станом, робить його повноцінною альтернативою React для багатьох типів додатків.

Ще однією конкуруючою технологією є Angular – фронтенд-фреймворк, який підтримується компанією Google. Angular забезпечує комплексний підхід до розробки вебдодатків: від обробки форм і маршрутизації до тестування і генерації коду. На відміну від React, який є лише бібліотекою для створення інтерфейсів, Angular пропонує повноцінне середовище для розробки SPA (Single Page Application). Його основними перевагами є сувора структура проєкту, використання TypeScript, що покращує типізацію і зменшує кількість помилок, а також наявність інструментів для тестування.

Якщо говорити про альтернативи Node.js на бекенді, варто згадати середовище Deno. Deno працює за замовчуванням із сучасними стандартами JavaScript і TypeScript, має вбудовані механізми безпеки і орієнтується на спрощену розробку без потреби у зовнішніх пакетних менеджерах. Хоча Deno ще не має такої великої спільноти, як Node.js, він вважається перспективною альтернативою для розробки сучасних бекенд-додатків.

Серед фреймворків, що можуть виступати конкурентами Express.js, варто виділити NestJS. Це прогресивний Node.js-фреймворк для створення ефективних і масштабованих серверних додатків. NestJS базується на ідеях об’єктно-орієнтованого програмування і використовує TypeScript. Фреймворк надає розробникам можливість будувати складні серверні архітектури, при цьому підтримуючи модульність, залежності, а також інші патерни

проєктування. Порівняно з Express.js, NestJS більше структурований і краще підходить для великих проєктів, де важлива чітка архітектура і масштабованість.

Ще одним варіантом заміни Express.js є Koa.js – мінімалістичний фреймворк, розроблений творцями Express.js. Koa прагне бути ще більш гнучким і легким, залишаючи розробнику максимум свободи у виборі структури додатку. На відміну від Express, де багато речей йдуть «із коробки», у Koa більшість функцій потрібно підключати самостійно, що дозволяє будувати легкі і дуже швидкі додатки, мінімізуючи залежності.

Окремо варто розглянути новітні підходи до розробки повноцінних вебдодатків, серед яких Next.js та Nuxt.js. Next.js є розширенням React, яке надає можливість рендерингу сторінок на сервері (SSR – Server-Side Rendering) та генерації статичних сайтів (SSG – Static Site Generation). Це значно покращує продуктивність вебдодатків, а також сприяє SEO-оптимізації. Подібним чином Nuxt.js виконує ту ж роль для екосистеми Vue.js, пропонуючи зручне середовище для створення універсальних додатків. Крім того, для розробки великих, комплексних систем набуває поширення концепція мікрофронтендів – архітектурний підхід, який дозволяє різним командам розробляти та розгортати частини одного вебдодатку незалежно одна від одної. У цьому контексті такі рішення, як Module Federation у Webpack, стали інструментом для побудови модульних вебархітектур.

1.6 Висновки до розділу 1

Аналіз вимог до системи показав, що вебзастосунок має забезпечувати високу продуктивність, особливо під час обробки запитів, пов’язаних із формуванням замовлень та взаємодією з базою даних. Одним із головних пріоритетів розробки є створення інтуїтивно зрозумілого та швидкодіючого користувацького інтерфейсу. Все це зумовило вибір інструментарію, який

дасть змогу ефективно реалізовувати функціонал, оперативно виявляти й усувати помилки та покращувати загальну якість програмного продукту:

- **React**: забезпечує високу продуктивність, ефективну роботу з DOM та швидкий рендеринг інтерфейсу, що позитивно впливає на зручність користування;
- **Node.js**: дає змогу будувати масштабовані серверні додатки з підтримкою великої кількості одночасних підключень завдяки асинхронній природі виконання;
- **PostgreSQL**: потужна та стабільна реляційна система управління базами даних, яка забезпечує надійне зберігання структурованих даних;
- **Express.js**: мінімалістичний фреймворк для реалізації RESTful API, що полегшує інтеграцію між клієнтською та серверною логікою;
- **ESLint**: інструмент для контролю якості коду, який забезпечує дотримання єдиної стилістики та своєчасне виявлення помилок у процесі розробки.

Враховуючи підготовлену теоретичну базу, можна переходити до етапу проєктування вебзастосунку, що передбачає деталізацію його структури, взаємодію компонентів і підготовку до практичної реалізації.

2 ПРОЄКТУВАННЯ ВЕБЗАСТОСУНКУ

2.1 Проєктування архітектури вебзастосунку

Процес розробки будь-якого сучасного вебзастосунку неможливо уявити без чіткого архітектурного планування. Це стосується як великих корпоративних систем, так і локальних бізнес-проєктів. Архітектура відіграє роль своєрідного «скелета» системи, на якому базується її подальша функціональність, масштабованість, простота обслуговування та потенціал для розширення. Від того, наскільки добре реалізована архітектура, залежить стабільність роботи, швидкість розгортання нових функцій і якість взаємодії користувачів з додатком. Успішно спроєктована архітектура дозволяє мінімізувати ризики виникнення технічних боргів, спрощує виявлення та усунення помилок, а також сприяє впровадженню сучасних технологічних рішень.

Формування архітектури починається з чіткого поділу вебдодатку на два ключові логічні рівні – клієнтський і серверний. Така модель відповідає принципам багаторівневої архітектури [1] та забезпечує розподіл обов’язків між окремими частинами системи. Це призводить до низького рівня взаємозв’язку між модулями, що полегшує модифікацію та підтримку проєкту в довгостроковій перспективі. Клієнтська частина відповідає за візуальну складову – побудову та динамічне оновлення інтерфейсу, в той час як серверна частина забезпечує логічну обробку, управління користувачами, обробку запитів до бази даних та контроль доступу. Крім того, розділення обов’язків дозволяє незалежно тестувати, масштабувати та оновлювати окремі частини застосунку без порушення загальної цілісності системи.

Загальна схема логічного поділу розроблюваної системи на компоненти представлена на рисунку 2.1 у вигляді діаграми компонентів, яка ілюструє взаємодію між основними елементами вебзастосунку.

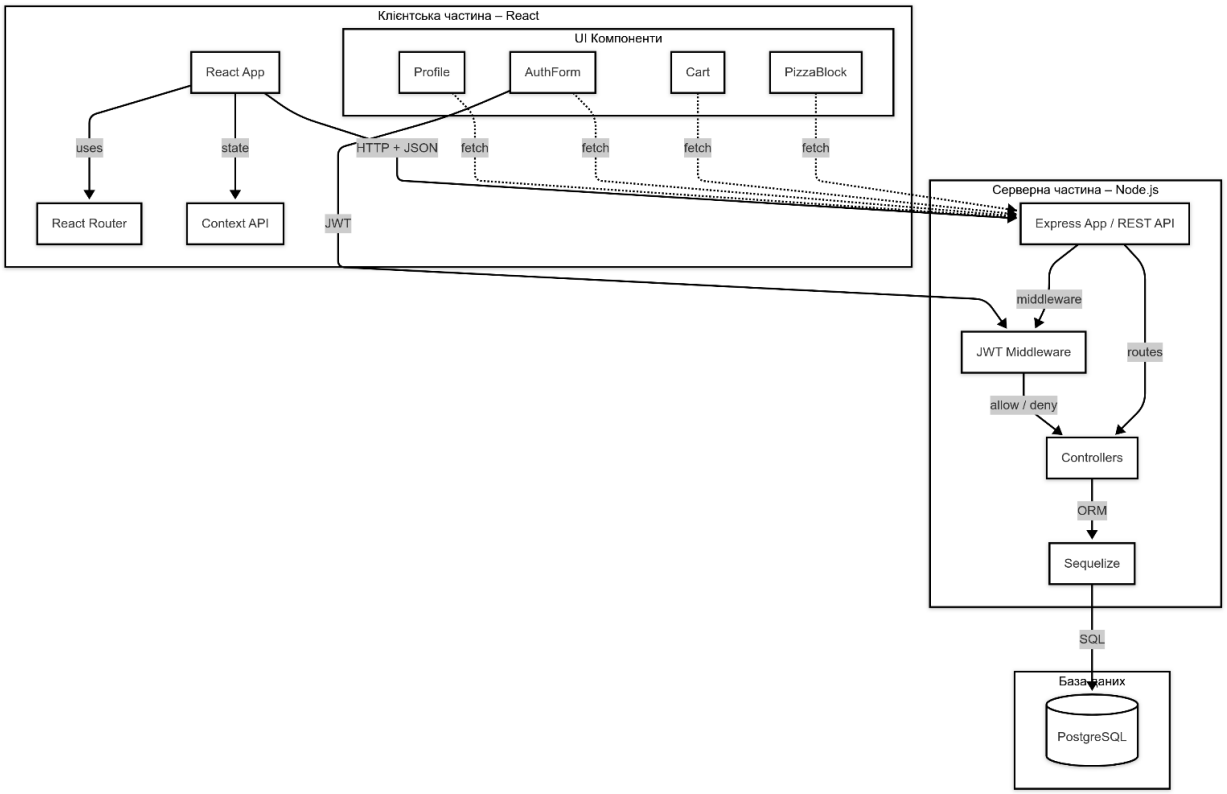


Рисунок 2.1 – Діаграма компонентів вебзастосунку

Окрім логічної структури, важливо враховувати фізичне розміщення компонентів системи на конкретних вузлах інфраструктури. На рисунку 2.2 зображено діаграму розгортання вебзастосунку, яка демонструє, як клієнтська частина взаємодіє з сервером та базою даних.

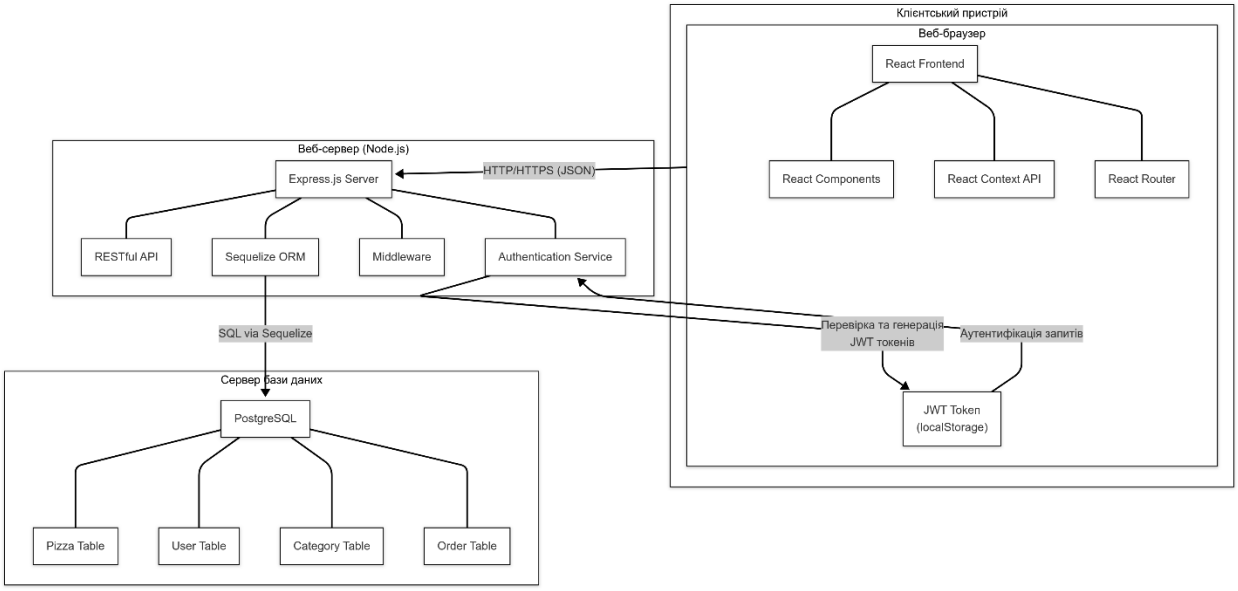


Рисунок 2.2 – Діаграма розгортання вебзастосунку

Клієнтська частина розробляється з використанням бібліотеки React, яка надає потужні можливості для створення динамічного, компонентно-орієнтованого інтерфейсу. Кожен елемент інтерфейсу – від списку піц до кошика або форми реєстрації – реалізований як окремий компонент, який можна повторно використовувати або адаптувати до змін. Така структурна організація зменшує залежність між частинами інтерфейсу та спрощує його тестування і модифікацію.

Взаємодія між компонентами організовується через React Context API, який дозволяє централізовано керувати глобальним станом, наприклад, відстежувати вміст кошика або статус авторизації користувача. Це значно спрощує побудову логіки в складних інтерфейсах і зменшує дублювання коду, що особливо важливо в умовах активного розширення функціоналу.

Для маршрутизації на клієнті буде використовуватись бібліотека React Router, яка забезпечує плавну навігацію між сторінками без необхідності перезавантаження. Реалізація умовних маршрутів дозволяє, зокрема, обмежити доступ до профілю лише авторизованим користувачам. Крім того, для покращення користувацького досвіду буде реалізовано скелетний завантажувач головної сторінки, який візуально інформує користувача про процес завантаження списку піц.

Серверна частина розробляється на Node.js з використанням фреймворку Express.js, який надає простий та розширюваний інструментарій для створення RESTful API. Кожен ресурс вебдодатку – користувачі, замовлення, піци, категорії – буде мати власний маршрут з методами обробки CRUD-операцій.

Express.js дозволяє швидко підключати проміжне програмне забезпечення для логування, обробки помилок, перевірки авторизації та інших сервісних завдань. Така модульність буде спрощувати налагодження та супровід коду. Для реалізації механізму аутентифікації сервер буде використовувати JWT (JSON Web Token). На рисунку 2.3 наведено структуру JWT-токена.

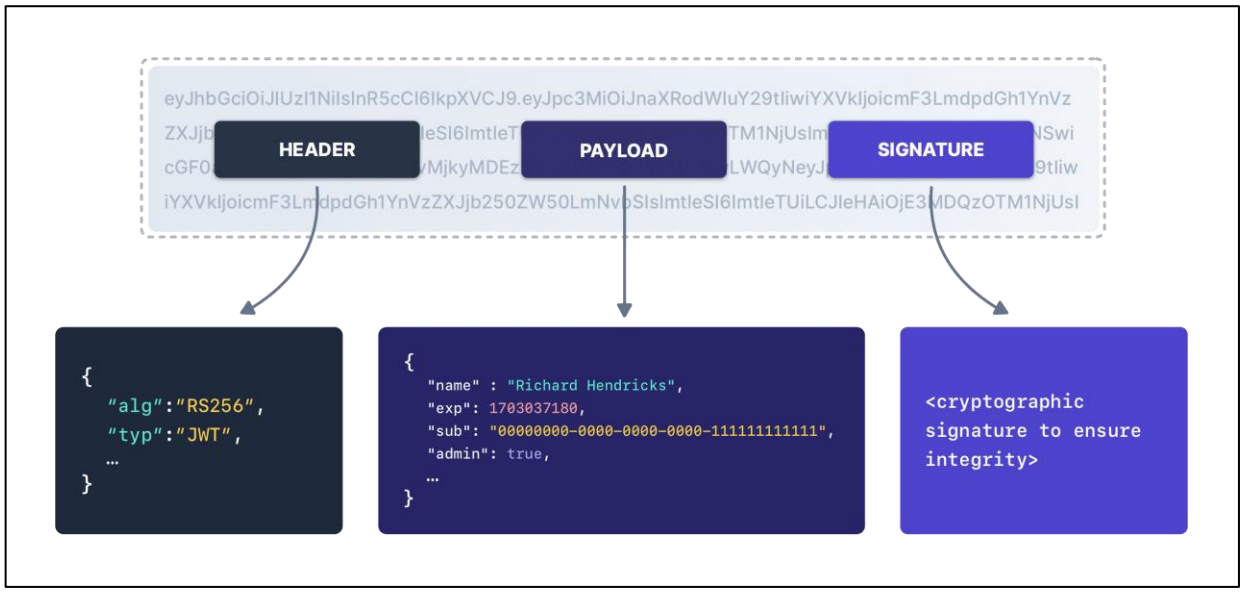


Рисунок 2.3 – Структура JWT-токена [5]

Після успішної реєстрації або входу в систему сервер буде генерувати токен, який зберігається на клієнті (зазвичай в localStorage) і додається до наступних запитів для безпечних маршрутів. Це гарантує, що сесія користувача може підтримуватися без необхідності зберігати стан на сервері, а також дозволяє централізовано контролювати доступ до конфіденційних ресурсів. У разі недійсного або простроченого токена система автоматично буде обмежувати доступ і перенаправляти користувача на сторінку входу в систему.

Логіка сервера буде працювати з базою даних PostgreSQL, взаємодія з якою реалізується за допомогою ORM-бібліотеки Sequelize. Це дозволяє описувати структуру таблиць і зв’язки між ними на рівні JavaScript-об’єктів, що робить код більш наочним і керованим.

Моделі, які відповідають за структуру даних (наприклад, моделі користувача, піци, категорії), включають атрибути, типи полів, правила валідації та асоціації. Sequelize також підтримує систему міграції, яка забезпечує контрольовані зміни схеми бази даних без втрати даних і полегшує розгортання в різних середовищах.

На рисунку 2.4 наведено приклад створення таблиці “Pizza” за допомогою бібліотеки Sequelize.

```
// Основна модель піци
const Pizza = sequelize.define('Pizza', {
  id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
  imageUrl: { type: DataTypes.STRING, allowNull: false },
  title: { type: DataTypes.STRING, allowNull: false, unique: true },
  description: { type: DataTypes.STRING, allowNull: false },
  price: { type: DataTypes.INTEGER, allowNull: false },
  popularity: { type: DataTypes.INTEGER, allowNull: false },
});

// Відношення
Pizza.belongsTo(Category);
Category.hasMany(Pizza);
Pizza.belongsToMany(DoughType, { through: 'PizzaDoughTypes' });
DoughType.belongsToMany(Pizza, { through: 'PizzaDoughTypes' });
Pizza.belongsToMany(Size, { through: 'PizzaSizes' });
Size.belongsToMany(Pizza, { through: 'PizzaSizes' });
```

Рисунок 2.4 – Приклад створення таблиці “Pizza”

Взаємодія між клієнтом і сервером реалізується у вигляді HTTP-запитів через функцію fetch, з обробкою в форматі JSON. Це дозволить зручно передавати дані між фронтендом і бекендом. Всі запити відповідають стандартам REST: GET – для отримання інформації, POST – для створення нових записів, PUT – для оновлення існуючих, DELETE – для їх видалення..

Безпека взаємодії між клієнтом і сервером є одним з пріоритетів архітектури. Кожен запит до захищених маршрутів буде супроводжуватися перевіркою автентичності, в ході якої сервер буде розпізнавати наданий JWT-токен, перевіряючи його дійсність і визначати, чи дозволений доступ до ресурсу. Таким чином, запобігається несанкціонований доступ до персональних або критично важливих даних, а порушення в логіці авторизації негайно блокуються системою.

В результаті архітектура вебзастосунку для піцерії побудована як злагоджений набір взаємодіючих компонентів. Клієнтська частина на основі React буде забезпечувати інтуїтивно зрозумілий і динамічний інтерфейс, тоді як серверна частина на основі Node.js і Express.js відповідати за бізнес-логіку, безпеку і стабільну взаємодію з базою даних. Такий поділ дозволяє гнучко масштабувати систему та швидко реагувати на мінливі вимоги.

2.2 Розробка користувацького інтерфейсу

У сучасних вебзастосунках користувацький інтерфейс відіграє вирішальну роль, оскільки є точкою безпосередньої взаємодії між користувачем і системою. Візуальна частина вебсайту визначає не тільки перше враження про сервіс, але й загальну зручність використання, швидкість навігації та рівень задоволеності відвідувачів. Основна мета при реалізації інтерфейсу – забезпечити максимальну простоту використання, незалежно від рівня технічної підготовки користувача. Інтерфейс також повинен бути адаптивним, що дозволить йому однаково добре працювати як на мобільних пристроях, так і на екранах персональних комп’ютерів. Такий підхід спрямований на охоплення якомога ширшої аудиторії, включаючи власників смартфонів, планшетів та ноутбуків.

Проєктування інтерфейсу передбачало визначення ключових сценаріїв взаємодії користувача з сайтом. Основні серед них можна спостерігати на рисунку 2.5.

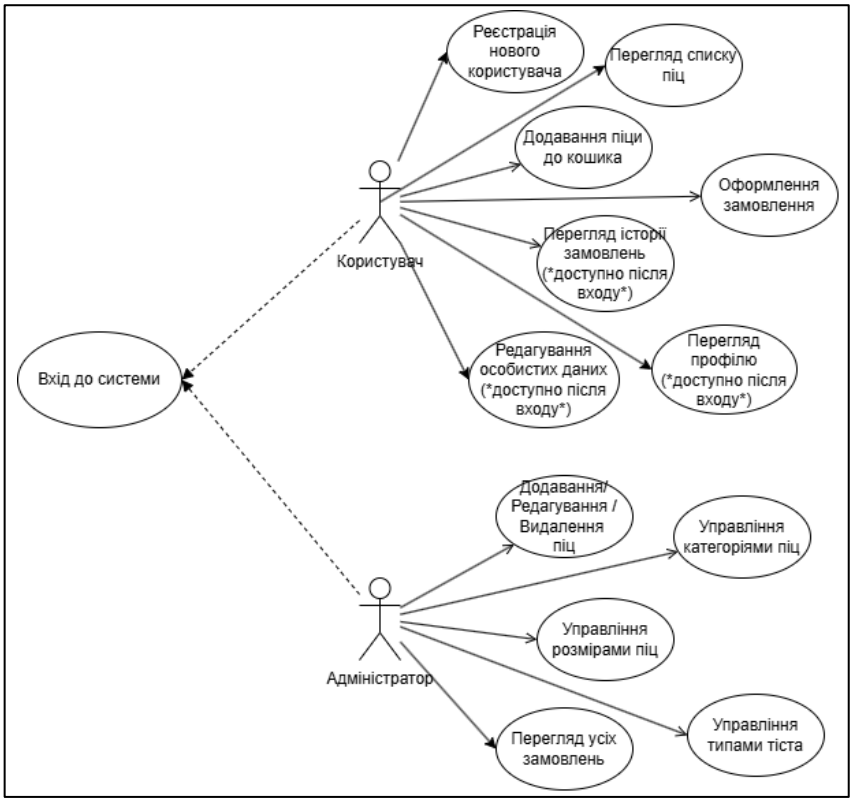


Рисунок 2.5 – Діаграма прецедентів для вебдодатку піцерії

Інтерфейс **головної сторінки** реалізується як відправна точка взаємодії з вебдодатком і містить базову навігацію та елементи презентації асортименту.

Для кращого сприйняття дизайну головної сторінки на рисунку 2.6 продемонстровано її дизайн-макет, що ілюструє просторове розміщення ключових елементів інтерфейсу.

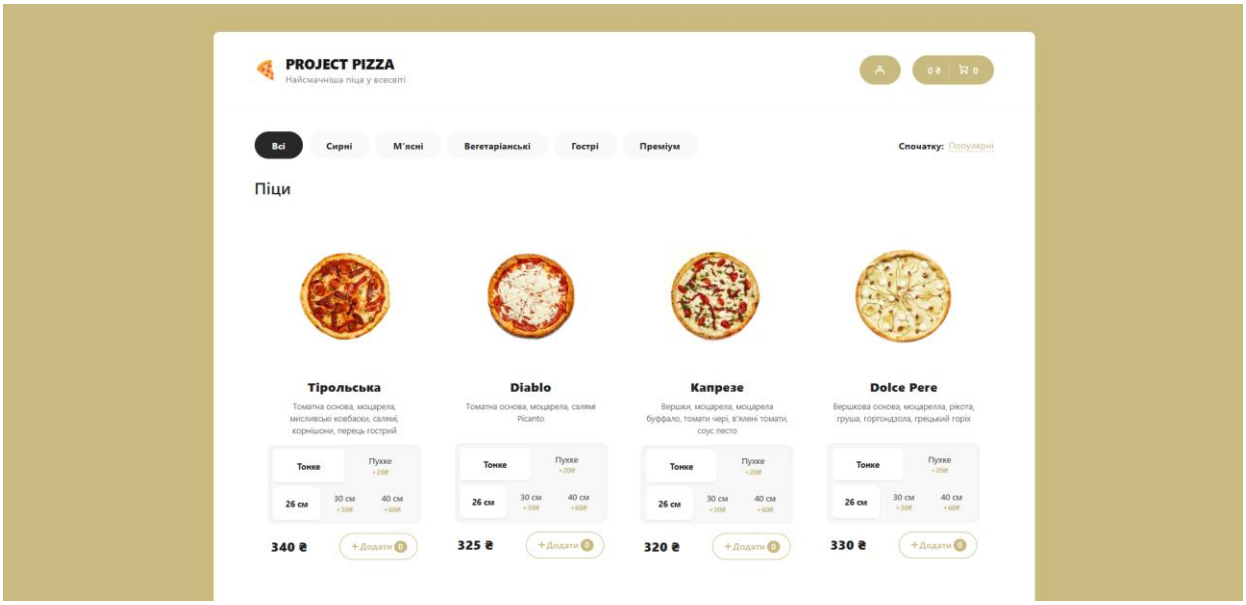


Рисунок 2.6 – Дизайн-макет головної сторінки вебдодатка

У верхній частині буде розміщено логотип компанії, який також слугуватиме посиланням на головну сторінку.

Праворуч можуть розташовуватися інтерактивні іконки профілю користувача та кошика, в якому буде відображатися поточна кількість обраних товарів.

Нижче розташовуватиметься блок категорій у вигляді горизонтального списку кнопок – активна категорія візуально підсвічуватиметься, що забезпечить зручну фільтрацію товарів.

Меню сортування буде розташоване праворуч від блоку категорій і дозволятиме змінити порядок відображення піц – за популярністю або за ціною.

Центральну частину сторінки займатиме список доступних піц. Кожна позиція реалізовуватиметься у вигляді окремого блоку із зображенням,

назвою, описом інгредієнтів, опціями вибору розміру і типу тіста, а також кнопкою «Додати». Для покращення сприйняття під час завантаження буде реалізовано скелет-заповнювач – тимчасова візуалізація, яка зникає після отримання останніх даних з сервера.

Інтерфейси **сторінок реєстрації** та **входу** реалізовуватимуться в єдиному стилістичному оформленні. Вони будуть містити поля для введення імені, прізвища, електронної пошти, пароля та адреси. Всі поля матимуть підказки, які полегшать їх заповнення. Якщо виникне помилка, система одразу ж виведе повідомлення з поясненням, наприклад, якщо неправильно вказано формат електронної пошти або пароль занадто короткий. Після успішного входу інтерфейс автоматично перенаправить користувача на сторінку профілю.

Інтерфейс **сторінки профілю** дозволить користувачеві переглядати та змінювати особисту інформацію, включаючи ім’я, прізвище, адресу доставки та номер телефону. Інтерактивні поля для редагування інформації супроводжуватимуться валідацією та повідомленнями про результат – успішне збереження або помилку. Окремий розділ сторінки профілю присвячуватиметься історії замовлень: тут будуть відображатися всі попередні покупки з детальною інформацією про вміст замовлення, дату оформлення та загальну суму.

Інтерфейс **кошика** буде реалізований у вигляді окремої сторінки, яка відкривається при натисканні на відповідну іконку в шапці сайту. На цій сторінці користувач бачитиме всі додані до кошика позиції, кожну з яких можна відредагувати: змінити кількість, видалити повністю або повернутися до каталогу товарів.

У нижній частині інтерфейсу кошика відображатиметься загальна сума замовлення. Кнопка оформлення замовлення відкриватиме форму з полями для введення контактної інформації та адреси доставки. Якщо користувач авторизований, деякі поля заповнюватимуться автоматично на основі його профілю, що скорочує час оформлення замовлення. Після підтвердження

замовлення буде з’являтися повідомлення про успішне завершення транзакції, і користувач перенаправлятиметься в особистий кабінет, де нове замовлення автоматично додаватиметься в історію.

2.3 Налаштування серверної частини вебзастосунку

Налаштування серверної частини вебдодатку має бути одним з ключових кроків у процесі створення надійної, масштабованої та безпечної системи, яка зможе ефективно обробляти запити користувачів, взаємодіяти з базою даних та відповідати як функціональним, так і нефункціональним вимогам.

На початковому етапі розробки планується створити базову структуру серверного додатку. Архітектура проєкту передбачатиме поділ на окремі директорії для маршрутів, контролерів, моделей та конфігураційних файлів. У кореневому каталозі міститиметься основний файл запуску index.js, який відповідатиме за ініціалізацію сервера, підключення до бази даних та реєстрацію проміжного програмного забезпечення.

Для безпечного зберігання конфіденційної інформації планується використовувати файл .env, в якому будуть зберігатися змінні оточення, включаючи параметри підключення до бази даних PostgreSQL, порт запуску, секретні ключі для генерації токенів та інші критичні налаштування.

Для реалізації RESTful API планується реалізувати систему маршрутизації, засновану на принципі логічного розподілу обов’язків. Кожен маршрут буде оброблятися окремим контролером, який буде містити функції для взаємодії з відповідними моделями. HTTP-запити оброблятимуться за допомогою проміжного програмного забезпечення, яке відповідатиме за логування, обробку помилок, аутентифікацію та валідацію запитів.

Підключення до бази даних PostgreSQL буде здійснюватися за допомогою ORM-бібліотеки Sequelize, що забезпечить зручну роботу з

таблицями бази даних у вигляді об’єктів. Параметри конфігурації з’єднання будуть розміщені в окремому файлі, звідки будуть зчитуватися дані з .env.

Що стосується аутентифікації, то на етапі проєктування передбачено використання методу токенізації за допомогою JWT (JSON Web Token). Після завершення процедури авторизації сервер повинен згенерувати унікальний токен, який буде надісланий клієнту та збережений у пам’яті браузера. Під час наступних запитів клієнт надсилатиме цей токен у заголовку авторизації, після чого сервер перевірятиме його дійсність. У разі успішної перевірки буде надано доступ до захищених маршрутів. Така система дозволить підтримувати аутентифікацію без необхідності зберігати сесії на сервері.

Безпечні маршрути, які потребують перевірки прав доступу, будуть захищені спеціальними функціями проміжного програмного забезпечення. У разі відсутності або недійсності токену запит буде автоматично відхилено з відповідним повідомленням, а користувач буде перенаправлений на форму входу в систему.

Таким чином, конфігурація серверної частини вебдодатку передбачається як багаторівнева система, що включатиме налаштування середовища, організацію файлової структури, побудову REST API, захист маршрутів, підключення до бази даних та реалізацію аутентифікації..

2.4 Проєктування структури БД з використанням ER-діаграми

Одним з найважливіших етапів в процесі розробки вебдодатку є створення структури бази даних, яка зберігає всі необхідні дані для функціонування системи. Щоб правильно організувати дані, уникнути дублювання та забезпечити цілісність інформації, створено ER-діаграму (Entity-Relationship diagram), яка дозволяє візуально моделювати зв’язки між сутностями, визначати їх атрибути, типи зв’язків та обмеження. В контексті розробки сайту піцерії ця діаграма стане основою для побудови логічної

моделі даних, яка в подальшому реалізовуватиметься в PostgreSQL за допомогою ORM-бібліотеки Sequelize.

ER-діаграма являє собою набір таблиць (сутностей), кожна з яких виконує окрему функцію в процесі зберігання та обробки даних. Зв’язки між таблицями моделюють дії користувачів на сайті, наприклад: додавання піци в кошик, вибір розміру, типу тіста, оформлення замовлення тощо. Це дозволяє створити систему, яка не лише ефективно виконує свої функції, а й залишається гнучкою та придатною до подальших змін у бізнес-логіці чи користувацьких сценаріях.

На рисунку 2.7 представлено ER-діаграму бази даних вебзастосунку. Опишемо таблиці, які були представлені на ER-діаграмі.

Центральною сутністю є таблиця **“Users”**, яка містить дані про зареєстрованих користувачів, включаючи їх ім’я, прізвище, електронну пошту, пароль, номер телефону, адресу доставки та роль в системі. Кожен користувач може створювати замовлення та додавати товари до кошика. Модель реалізує зв’язок «один-до-багатьох» між користувачами та замовленнями, а також між користувачами та кошиками.

Сутність **“Pizza”** містить інформацію про асортимент: назва піци, опис інгредієнтів, ціна, популярність, категорія, зображення. Кожна піца належить до певної категорії, наприклад, «м’ясна», «вегетаріанська», «гостра» тощо. Відповідно, таблиця **“Categories”** має зв’язок «один до багатьох» з піцами, тобто одна категорія може містити декілька піц, але кожна піца належить лише до однієї категорії.

Сутність **“Orders”** відповідає за зберігання інформації про всі здійснені покупки. Замовлення містить дані про користувача, дату створення, загальну суму і статус виконання.

Кожне замовлення складається з одного або декількох елементів, які зберігаються в таблиці **“OrderItems”**. У цій таблиці зберігаються посилання на конкретну піцу, обраний розмір і тип тіста, кількість одиниць і загальна вартість.

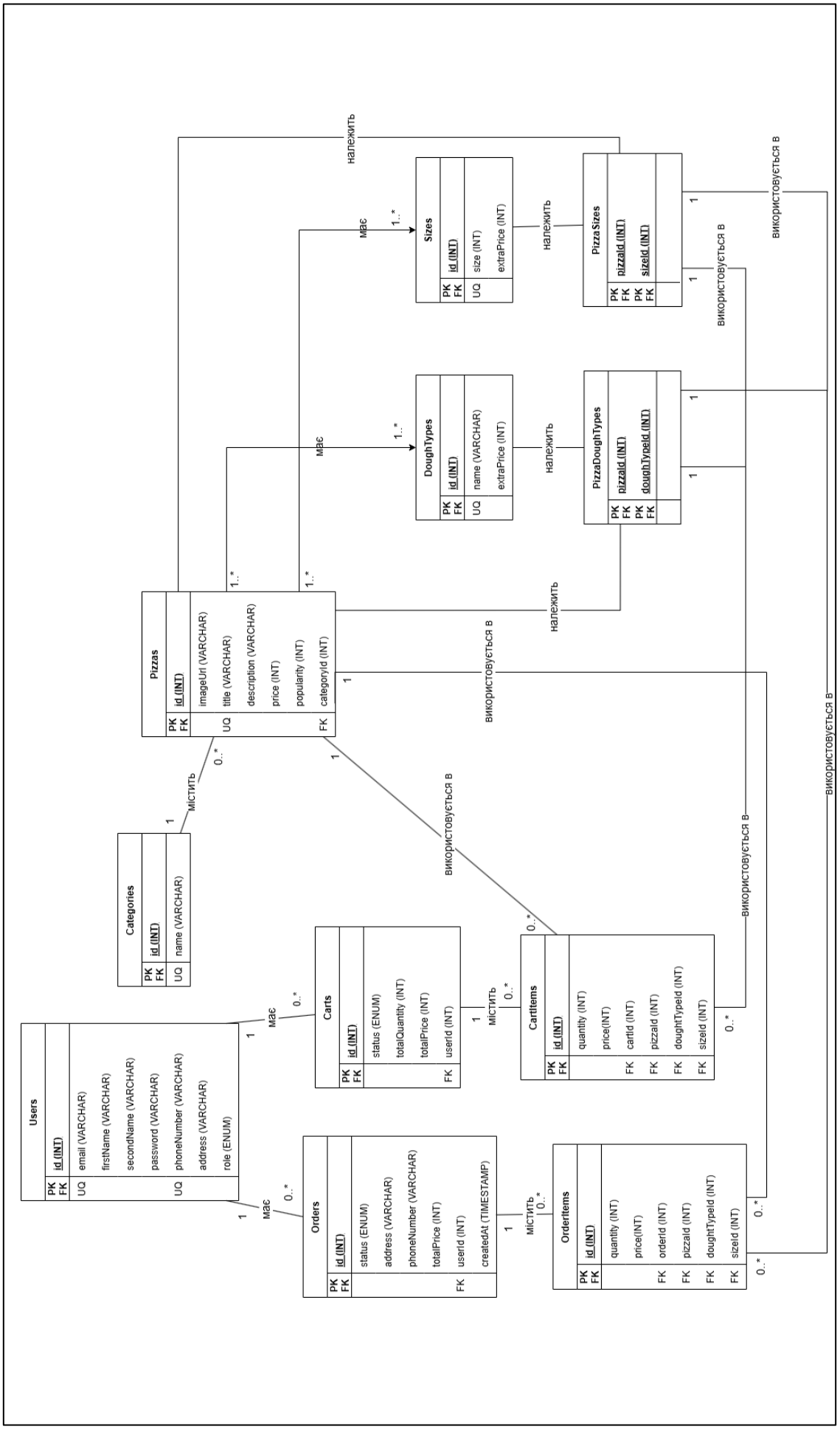


Рисунок 2.7 – ER-діаграма бази даних

Для збереження поточного вмісту кошика реалізована окрема таблиця **“Cards”**, яка має зв’язок з користувачем «один до одного». Кожен кошик складається з окремих позицій – вони описані в таблиці **“CartItems”**. Як і у випадку з замовленнями, кожна позиція містить інформацію про піцу, її розмір, тип тіста, кількість та ціну. Завдяки такій структурі зручно реалізувати функціонал додавання позицій до кошика та редагування їх параметрів до моменту підтвердження замовлення.

Також на діаграмі показані сутності **“Sizes”** і **“DoughType”**, які відповідають за варіативність піци за параметрами. Вони не пов’язані безпосередньо з таблицею “Pizzas”, а з’єднані через проміжні таблиці **“PizzaSizes”** та **“PizzaDoughTypes”**, які реалізують зв’язок «багато до багатьох». Таким чином, одна піца може бути доступна в декількох розмірах або з різними типами тіста, і навпаки – один тип тіста може бути доступний для декількох піц.

2.5 Висновки до розділу 2

Отже, в процесі проєктування вебдодатку для піцерії було сформовано архітектурну модель, що поєднує клієнтську та серверну частини, описано принципи взаємодії між ними, структуру користувацького інтерфейсу, механізми безпеки та логіку зберігання даних. Були визначені програмні засоби, побудована структура бази даних, описані основні сценарії взаємодії користувача з системою. Всі ці елементи заклали основу для подальшої реалізації вебдодатку. Тому наступним логічним кроком є перехід до етапу безпосередньої розробки та впровадження функціональних модулів системи.

3 РЕАЛІЗАЦІЯ ВЕБЗАСТОСУНКА

3.1 Розробка клієнтської частини з використанням бібліотеки React

Клієнтська частина вебдодатку піцерії реалізована з використанням сучасної бібліотеки React, яка надає зручний користувацький інтерфейс на основі компонентного підходу. Такий підхід спрощує підтримку коду, полегшує впровадження нових функцій та сприяє високій гнучкості при масштабуванні. Крім того, React має розгалужену екосистему бібліотек, які дозволяють розширювати функціональність, працювати зі станом додатку, підвищувати продуктивність та легкість розробки. Також дозволяє розділити інтерфейс на незалежні, багаторазово використовувані частини, кожна з яких відповідає за певну функціональність або зовнішній вигляд.

Сторінка “Home” – це основна точка входу користувача на сайт піцерії. Вона виступає центральним елементом клієнтського інтерфейсу, за допомогою якого користувач взаємодіє з асортиментом продукції, фільтрує товари, обирає бажані опції та додає товари до кошика.

Реалізація цієї сторінки демонструє всі ключові можливості бібліотеки React – від управління станами до побудови динамічних компонентів, які реагують на дії користувача та оновлення даних. Сторінка знаходиться в каталозі src/pages/Home.jsx і відповідає за завантаження, відображення та фільтрацію асортименту піц. Вона використовує React-хуки useState та useEffect [3] для зберігання стану та виконання побічних ефектів, включаючи API-запити.

Основна логіка полягає в завантаженні категорій і списку піц з сервера на основі обраних параметрів фільтрації. У початковому стані додаток показує всі доступні піци, але після вибору категорії або зміни параметра сортування результат динамічно оновлюється без перезавантаження сторінки. Структуру сторінки “Home” наведено на рисунку 3.1.

```
const Home = () => {
  const [pizzas, setPizzas] = React.useState([]);
  const [categories, setCategories] = React.useState([]);
  const [isLoading, setIsLoading] = React.useState(true);
  const [hasError, setHasError] = React.useState(false);
  const [categoryId, setCategoryId] = React.useState(0);
  const [sortType, setSortType] = React.useState(0);

  React.useEffect(() => {
    fetch('http://localhost:5000/api/categories')
      .then((res) => {
        if (!res.ok) throw new Error('Failed to fetch categories');
        return res.json();
      })
      .then((data) => setCategories(data))
      .catch((err) => { console.error(err); setHasError(true); });
  }, []);

  React.useEffect(() => {
    setIsLoading(true);
    setHasError(false);
    fetch(
      `http://localhost:5000/api/pizzas?categoryId=${categoryId}&sortBy=${sortType}`
    )
      .then((res) => {
        if (!res.ok) throw new Error('Failed to fetch pizzas');
        return res.json();
      })
      .then((data) => { setPizzas(data); setIsLoading(false); })
      .catch((err) => { console.error(err); setIsLoading(false); setHasError(true); });
  }, [categoryId, sortType]);

  if (hasError) {
    return (
      <div className="container">
        <h2 className="content__title">Щось пішло не так...</h2>
        <p>Не вдалося завантажити дані з сервера. Спробуйте пізніше.</p>
      </div>
    );
  }

  return (
    <div className="container">
      <div className="content__top">
        <Categories
          items={[{ id: 0, name: 'Всі' }, ...categories]}
          value={categoryId}
          onClickCategory={setCategoryId}
        />
        <Sort value={sortType} onClickSort={setSortType} />
      </div>
      <h2 className="content__title">Піци</h2>
      <div className="content__items">
        {isLoading
          ? [...new Array(8)].map((_, idx) => <Skeleton key={idx} />)
          : pizzas.map((pizza) => (
              <PizzaBlock
                key={pizza.id} id={pizza.id} title={pizza.title} description={pizza.description}
                imageUrl={pizza.imageUrl} price={pizza.price} types={pizza.types} sizes={pizza.sizes}
                />
            ))}
      </div>
    </div>
  );
};

export default Home;
```

Рисунок 3.1 – Структура сторінки “Home”

У сторінці оголошуються стани: “pizzas”, “categories”, “isLoading”, “hasError”, “categoryId” та “sortType”. Вони використовуються для зберігання отриманих даних, контролю процесу завантаження та реагування на зміну

параметрів фільтрації. Після монтування сторінки запускаються два “useEffects” – один завантажує список категорій з API “/api/categories”, інший завантажує піци з API “/api/pizzas” з урахуванням поточних “lcategoryId” і “sortType”. Якщо при отриманні даних виникає помилка (наприклад, якщо сервер недоступний), сторінка виводить відповідне повідомлення користувачеві. Повідомлення про помилку при неможливості завантаження піц наведено на рисунку 3.2.

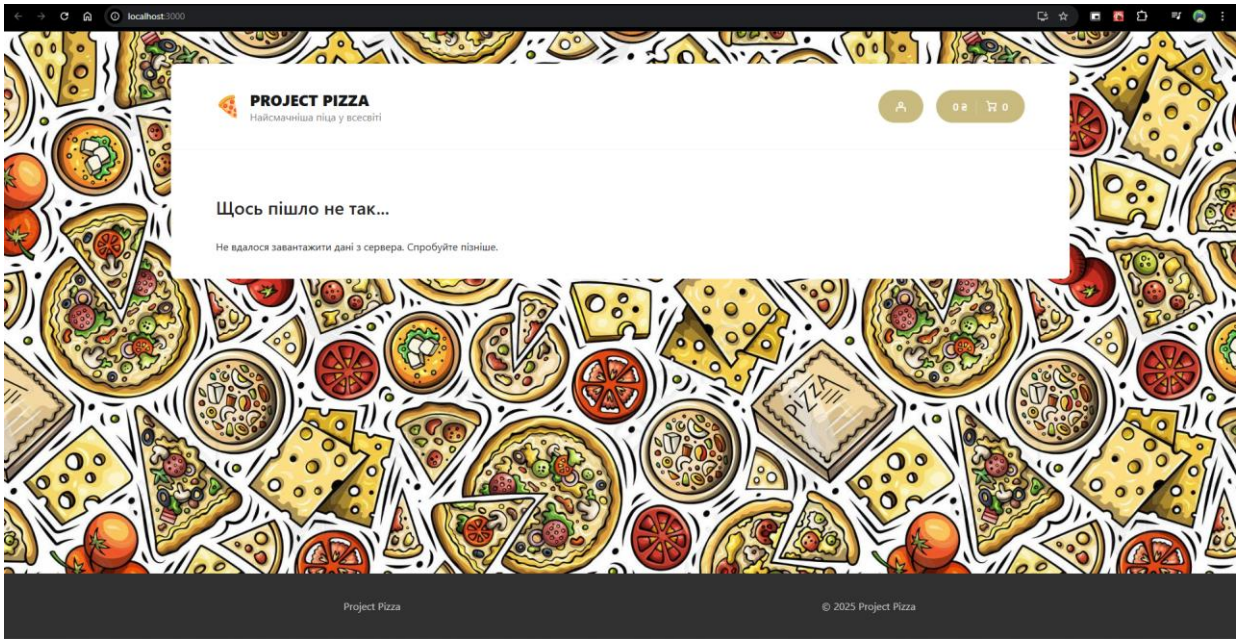


Рисунок 3.2 – Повідомлення про помилку при завантаженні піц

Компонент “Categories” відповідає за відображення категорій піц (наприклад, «М’ясні», «Сирні» тощо) у вигляді горизонтального списку кнопок. Кожен пункт є клікабельним елементом, який при натисканні оновлює стан “categoryId” на сторінці “Home”.

Компонент отримує “items” (масив категорій), поточне значення “value” (ідентифікатор обраної категорії) і функцію “onClickCategory”, яка оновлює стан в “Home”. Активна категорія візуально підсвічується, що покращує інтуїтивність взаємодії з інтерфейсом.

Візуальне відображення компонента “Categories” представлено на рисунку 3.3, а його структуру – на рисунку 3.4.

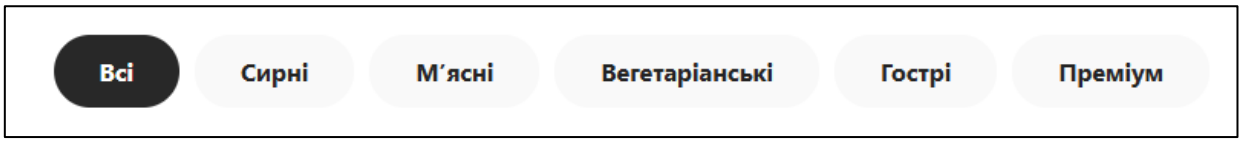


Рисунок 3.3 – Візуальне відображення компонента “Categories”

```
import React from 'react';

const Categories = ({ items, value, onClickCategory }) => {
  return (
    <div className="categories">
      <ul>
        {items.map((item) => (
          <li
            key={item.id}
            onClick={() => onClickCategory(item.id)}
            className={value === item.id ? 'active' : ''}
          >
            {item.name}
          </li>
        ))}
      </ul>
    </div>
  );
};

export default Categories;
```

Рисунок 3.4 – Структура компонента “Categories”

Компонент “Sort” відповідає за фільтрацію піц за різними критеріями: популярністю та ціною. Він відображає кнопку з поточним параметром сортування та список доступних варіантів, який відкривається при натисканні. Компонент підтримує локальний стан “isVisible”, який керує відображенням випадаючого списку. Вибрана опція передається батьківському компоненту за допомогою “onClickSort”. Візуальне відображення компонента наведено на рисунку 3.5, а його структуру – на рисунку 3.6.

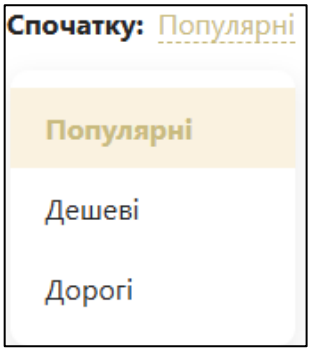


Рисунок 3.5 – Візуальне відображення компонента “Sort”

```
import React from 'react';

function Sort({ value, onClickSort }) {
  const [isVisible, setIsVisible] = React.useState(false);
  const list = ['Популярні', 'Дешеві', 'Дорогі'];
  const sortName = list[value];

  const handleSelect = (i) => {
    onClickSort(i); // повідомляємо Home
    setIsVisible(false);
  };

  return (
    <div className="sort">
      <div className="sort__label">
        {/* стрілочка */}
        <b>Спочатку:</b>
        <span onClick={() => setIsVisible(!isVisible)}>{sortName}</span>
      </div>
      {isVisible && (
        <div className="sort__popup">
          <ul>
            {list.map((name, i) => (
              <li
                key={i}
                onClick={() => handleSelect(i)}
                className={value === i ? 'active' : ''}
              >
                {name}
              </li>
            ))}
          </ul>
        </div>
      )}
    </div>
  );
}

export default Sort;
```

Рисунок 3.6 – Структура компонента “Sort”

Після завантаження даних на головній сторінці відображаються піци у вигляді окремих блоків. Коли “isLoading” має значення “true”, замість контенту відображаються скелетони – компонент “Skeleton”.

Компонент, побудований на основі бібліотеки “react-content-loader”, імітує форму блоку піци під час очікування відповіді сервера. У його шаблоні передбачено місце для зображення, назви піци, опису, селекторів типу тіста, розміру, ціни та кнопки «Додати». Це покращує користувацький досвід, оскільки скелетони запобігають появі порожніх екранів і сигналізують користувачеві, що система активно обробляє запит, навіть якщо є затримки з API.

Візуальне відображення скелетонів продемонстровано на рисунку 3.7, а його структуру – на рисунку 3.8.



Рисунок 3.7 – Візуальне відображення компонента “Skeleton”

```
import React from 'react';
import ContentLoader from 'react-content-loader';

function PizzaBlockSkeleton() {
  return (
    <ContentLoader
      className="pizza-block"
      speed={2}
      width={280}
      height={500}
      viewBox="0 0 280 500"
      backgroundColor="#f3f3f3"
      foregroundColor="#ecebeb"
    >
      {/* Зображення піци */}
      <circle cx="140" cy="125" r="100" />
      {/* Заголовок (назва піци) */}
      <rect x="35" y="270" rx="5" ry="5" width="210" height="30" />
      {/* Опис (інгредієнти) */}
      <rect x="20" y="315" rx="5" ry="5" width="240" height="15" />
      <rect x="50" y="335" rx="5" ry="5" width="180" height="15" />
      {/* Селектор типу тіста (блок) */}
      <rect x="25" y="370" rx="10" ry="10" width="230" height="45" />
      {/* Опції тіста */}
      <rect x="30" y="375" rx="8" ry="8" width="105" height="35" />{' '}
      {/* Тонке */}
      <rect x="145" y="375" rx="8" ry="8" width="105" height="35" />{' '}
      {/* Пухке */}
      {/* Селектор розміру піци (блок) */}
      <rect x="25" y="425" rx="10" ry="10" width="230" height="45" />
      {/* Опції розміру */}
      <rect x="30" y="430" rx="8" ry="8" width="65" height="35" /> {/* 26 см */}
      <rect x="107" y="430" rx="8" ry="8" width="65" height="35" />{' '}
      {/* 30 см */}
      <rect x="185" y="430" rx="8" ry="8" width="65" height="35" />{' '}
      {/* 40 см */}
      {/* Нижній блок */}
      <rect x="25" y="485" rx="5" ry="5" width="90" height="30" /> {/* Ціна */}
      <rect x="145" y="480" rx="20" ry="20" width="130" height="40" />{' '}
      {/* Кнопка "Додати" */}
    </ContentLoader>
  );
}

export default PizzaBlockSkeleton;
```

Рисунок 3.8 – Структура компонента “Skeleton”

Компонент “PizzaBlock” – це візуальний блок, який відображає інформацію про одну піцу. Він отримує такі пропси як “id”, “title”, “imageUrl”, “description”, “price”, “types”, “sizes” і формує з них повний вміст блоку.

Всередині компонента логіка вибору типу тіста та розміру піци реалізована за допомогою “useState”. Залежно від вибору розраховується остаточна ціна замовлення. Кожен варіант типу тіста та розміру має додаткову вартість. Коли користувач обирає певну комбінацію параметрів, ціна компонента оновлюється відповідно до цих додаткових витрат.

Компонент також використовує контекст “useCart”, який відповідає за глобальне зберігання інформації про додані продукти. Коли користувач натискає кнопку «Додати», функція “addItem” передає дані обраної піци в загальний стан. Після цього вона визначає, скільки одиниць поточної піци вже додано, і це число відображається на кнопці поруч з написом «Додати». Візуальне відображення компонента наведено на рисунку 3.9.

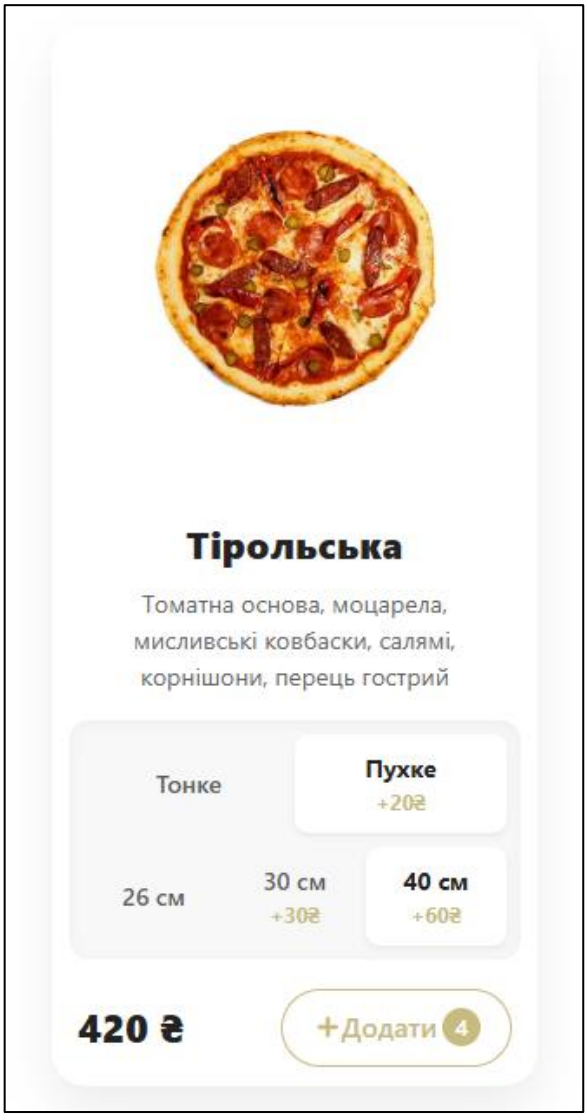


Рисунок 3.9 – Візуальне відображення компонента “PizzaBlock”

Отже, клієнтська частина React-додатку реалізована у вигляді набору взаємодіючих компонентів, кожен з яких виконує окрему роль. Сторінка “Home” забезпечує логіку завантаження, обробки та відображення даних, тоді як компоненти “Categories”, “Sort”, “PizzaBlock” та “Skeleton” відповідають за окремі функціональні частини. Завдяки використанню “React Context” забезпечено централізоване управління кошиком.

3.2 Розробка серверної частини з використанням Node.js

Розробка серверної частини вебдодатку для онлайн-піцерії здійснюється за допомогою середовища виконання JavaScript Node.js. В якості основного фреймворку для побудови HTTP-сервера використовується Express.js, що дозволяє організувати обробку маршрутів, підключення проміжного програмного забезпечення, взаємодію з базою даних та реалізацію REST API. Структура серверного додатку передбачає чіткий розподіл обов’язків: маршрутизація, контролери, моделі даних, аутентифікація та інші логічні блоки виділені в окремі модулі.

Для обміну даними між фронтенд- і бекенд-частинами реалізовано RESTful API, який дозволяє виконувати операції над сутностями, що використовуються в додатку: користувачами, піцами, категоріями, типами тіста, розмірами, візком тощо. Кожна кінцева точка обробляється відповідним маршрутом і контролером. Наприклад, маршрут “GET /api/pizzas” повертає список піц з фільтрацією за категоріями та сортуванням, а маршрут “POST /api/auth/register” обробляє запит на реєстрацію користувача.

Одним з ключових елементів функціоналу є відображення списку піц. Контролер “pizzaController” реалізує логіку завантаження піц з бази даних з урахуванням фільтрації за категоріями, сортування за ціною або популярністю, а також з урахуванням пов’язаних сутностей – категорій, типів тіста, розмірів (див. рис. 3.10).

```
const pizzas = await Pizza.findAll({
  where: catId > 0 ? { CategoryId: catId } : undefined,
  include: [
    {
      model: Category,
      attributes: ['id', 'name'],
    },
    {
      model: DoughType,
      attributes: ['id', 'name', 'extraPrice'],
      through: { attributes: [] },
    },
    {
      model: Size,
      attributes: ['id', 'value', 'extraPrice'],
      through: { attributes: [] },
    },
  ],
  order: [
    primaryOrder,
    ['id', 'ASC'],
    [Category, 'id', 'ASC'],
    [DoughType, 'id', 'ASC'],
    [Size, 'id', 'ASC'],
  ],
});

const result = pizzas.map((p) => {
  const d = p.toJSON();
  return {
    id: d.id,
    imageUrl: d.imageUrl,
    title: d.title,
    description: d.description,
    price: d.price,
    popularity: d.popularity,
    category: {
      id: d.Category.id,
      name: d.Category.name,
    },
    types: d.DoughTypes.map((dt) => ({
      id: dt.id,
      name: dt.name,
      extraPrice: dt.extraPrice,
    })),
    sizes: d.Sizes.map((sz) => ({
      id: sz.id,
      value: sz.value,
      extraPrice: sz.extraPrice,
    })),
  };
});
```

Рисунок 3.10 – Отримання потрібних даних для піц

В результаті клієнт отримує структуровані дані з усіма необхідними параметрами для відображення піци в інтерфейсі.

Для забезпечення безпечного доступу до персональних даних, таких як сторінка профілю користувача та історії замовлень, впроваджено функцію автентифікації за допомогою JSON Web Token (JWT). Коли користувач реєструється, відбувається верифікація введених даних, перевірка електронної пошти та номера телефону на унікальність, хешування пароля за допомогою

бібліотеки “bcrypt” і генерація JWT-токену. Код методу “register” демонструє етапи створення користувача, перевірки даних та створення токена (див. рис. 3.11).

```
  const { first_name, second_name, email, password, phone_number, address } =
    req.body;
  try {
    // перевіряємо, чи вже існує такий email або телефон
    const exists = await User.findOne({
      where: {
        [Op.or]: [{ email }, { phone_number }],
      },
    });
    if (exists) {
      // визначимо, що саме зайнято
      const msgs = {};
      if (exists.email === email) msgs.email = 'Email вже використовується';
      if (exists.phone_number === phone_number)
        msgs.phone_number = 'Номер телефону вже використовується';
      return res.status(400).json({ errors: msgs });
    }

    // хешуємо пароль
    const hash = await bcrypt.hash(password, 10);

    // створюємо користувача
    const user = await User.create({
      first_name,
      second_name,
      email,
      password: hash,
      phone_number,
      address,
    });

    // Перевірка наявності JWT_SECRET перед створенням токена
    if (!process.env.JWT_SECRET) {
      console.error('JWT_SECRET не встановлено! Токен не буде згенеровано.');
      return res
        .status(500)
        .json({ message: 'Помилка конфігурації сервера' });
    }

    // створюємо JWT для авторизації користувача
    const payload = { id: user.id, email: user.email, role: user.role };
    const token = jwt.sign(payload, process.env.JWT_SECRET, {
      expiresIn: '7d',
    });

    // повертаємо minimal user + token
    return res.status(201).json({
      user: { id: user.id, email: user.email, first_name, second_name },
      token,
    });
  } catch (err) {
    console.error('Register error:', err);
    return res
      .status(500)
      .json({ message: 'Помилка сервера при реєстрації' });
  }
},
```

Рисунок 3.11 – Реєстрація користувача

Згенерований токен передається у відповіді на клієнтську частину, де зберігається у локальному сховищі браузера або HTTP cookie. У подальшому цей токен автоматично додається до кожного запиту до серверу в заголовку “Authorization” у форматі “Bearer”, що дозволяє серверу швидко ідентифікувати користувача без повторного входу в систему.

Авторизація (метод “login”) включає перевірку електронної пошти та пароля, а також повернення токену в разі успішної аутентифікації. У разі неправильних облікових даних користувач отримує відповідне повідомлення про помилку (див. рис. 3.12).

```
  const { email, password } = req.body;
  try {
    const user = await User.findOne({ where: { email } });
    if (!user) {
      return res.status(401).json({ message: 'Невірний email або пароль' });
    }
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      return res.status(401).json({ message: 'Невірний email або пароль' });
    }

    // Підготовка payload та генерація токена
    const payload = { id: user.id, email: user.email, role: user.role };
    const token = jwt.sign(payload, process.env.JWT_SECRET, {
      expiresIn: '7d',
    });

    return res.json({
      user: {
        id: user.id,
        email: user.email,
        first_name: user.first_name,
        second_name: user.second_name,
      },
      token,
    });
  } catch (err) {
    console.error('Login error:', err);
    return res.status(500).json({ message: 'Помилка сервера при вході' });
  }
},
```

Рисунок 3.12 – Вхід користувача

Якщо дані успішно підтверджені, сервер генерує токен JWT, який містить закодовану інформацію про користувача, включаючи його “id”, “email” та “role”. Токен дійсний протягом обмеженого періоду часу, після чого необхідно повторно увійти в систему.

Отже, побудована архітектура дозволяє ефективно поєднувати клієнтську і серверну частини, забезпечуючи зручний інтерфейс для користувачів і надійну серверну логіку для обробки запитів, зберігання даних та захисту конфіденційної інформації.

3.3 Тестування вебзастосунку

Для перевірки працездатності та цілісності ключових функціональних модулів було обрано підхід інтеграційного тестування. Цей підхід дозволяє оцінити не тільки окремі компоненти, а й спосіб їх взаємодії – від відправки HTTP-запиту до обробки маршрутом, виконання логіки контролера, взаємодії з ORM і базою даних, і закінчуючи генерацією відповіді клієнту.

Для реалізації тестування було використано дві ключові бібліотеки. Jest [2] відповідає за структурування тестових сценаріїв, перевірку очікуваних результатів та фіксацію успішності виконання тестів, тоді як Supertest [6] надає можливість емулювати HTTP-запити без необхідності запуску браузера або зовнішнього клієнта.

Всі тести виконуються в ізольованому середовищі, спеціально налаштованому за допомогою конфігураційного файлу “.env.test”, в якому задаються параметри підключення до окремої тестової бази даних. Це забезпечує повну автономність процесу тестування, запобігає випадковому втручанню в робоче середовище і дозволяє щоразу розгортати базу з нуля, гарантуючи однакові умови запуску.

Перед кожною серією тестів база даних ініціалізується заново, а після завершення з’єднання закривається. Це запобігає накопиченню тестових даних і дозволяє перевірити поведінку системи в передбачуваних умовах. Фрагмент ініціалізації показано нижче (див. рис. 3.13).

Одним з ключових елементів системи є модуль аутентифікації. Його тестування починається з перевірки можливості успішної реєстрації

користувача, створення відповідного запису в базі даних та отримання у відповідь токену JWT. На рисунку 3.14 показано тестовий код, який перевіряє можливість реєстрації користувача.

```
//Створюємо чисту БД перед усіма тестами
beforeAll(async () => {
  await sequelize.sync({ force: true });
});

// Закриваємо підключення після завершення всіх тестів
afterAll(async () => {
  await sequelize.close();
});
```

Рисунок 3.13 – Ініціалізація тестового середовища

```
it('створює користувача та повертає JWT-токен', async () => {
  // Тестові вхідні дані
  const payload = {
    first_name: 'John',
    second_name: 'Doe',
    email: 'john.doe@example.com',
    password: 'strongPassword1',
    phone_number: '+380981234567',
    address: 'Kyiv, Khreshchatyk 1',
  };

  //Надсилаємо HTTP-запит на маршрут реєстрації
  const res = await request(app)
    .post('/api/auth/register')
    .send(payload)
    .expect(201); //очікуємо статус 201 Created

  //Перевіряємо базову структуру відповіді (user + token)
  expect(res.body).toEqual(
    expect.objectContaining({
      user: expect.any(Object),
      token: expect.any(String),
    })
  );

  //Переконуємось, що повернені дані коректні, а пароль не присутній
  expect(res.body.user).toMatchObject({
    id: expect.any(Number),
    email: payload.email,
    first_name: payload.first_name,
    second_name: payload.second_name,
  });
  expect(res.body.user).not.toHaveProperty('password');

  //Переконуємось, що запис дійсно створено у БД
  const created = await User.findOne({ where: { email: payload.email } });
  expect(created).not.toBeNull();
  expect(created.first_name).toBe(payload.first_name);
});
```

Рисунок 3.14 – Перевірка можливості реєстрації користувача

Даний тест підтверджує, що маршрут “/api/auth/register” коректно обробляє POST-запити, створює новий обліковий запис у базі даних та повертає токен для автентифікації.

Не менш важливою частиною системи є механізм виведення асортименту піц. Для цього створено набір тестів, які перевіряють:

– повернення повного списку піц із категоріями, розмірами та типами тіста;
– фільтрацію за категоріями;
– сортування за ціною у зростаючому та спадному порядку.

На рисунку 3.15 зображено тест, який перевіряє загальне отримання списку піц. Подібним чином протестовано фільтрацію за категоріями. Наприклад, якщо користувач хоче переглянути лише вегетаріанські піци, система має повернути відповідну підмножину асортименту (див. рис. 3.16).

```
it('повертає всі піци з повними залежностями', async () => {
  const res = await request(app).get('/api/pizzas').expect(200);

  //Переконуємось, що прийшло саме 3 піци
  expect(res.body.length).toBe(3);

  //У кожної піци має бути category, sizes, types
  res.body.forEach((p) => {
    expect(p).toEqual(
      expect.objectContaining({
        id: expect.any(Number),
        title: expect.any(String),
        category: expect.objectContaining({ id: expect.any(Number) }),
        sizes: expect.any(Array),
        types: expect.any(Array),
      })
    );
  });
});
```

Рисунок 3.15 – Тест повернення повного списку піц

```
it('фільтрує за categoryId', async () => {
  //Знаходимо id категорії «Веган» у БД
  const vegan = await Category.findOne({ where: { name: 'Веган' } });

  const res = await request(app)
    .get('/api/pizzas')
    .query({ categoryId: vegan.id }) // /api/pizzas?categoryId=...
    .expect(200);

  //Повертається тільки 1 піца Veggie
  expect(res.body.length).toBe(1);
  expect(res.body[0].title).toBe('Veggie');
});
```

Рисунок 3.16 – Тест фільтрації піц за категорією

Оскільки користувачі можуть сортувати список піц за ціною, необхідно впевнитися, що бекенд підтримує відповідні параметри сортування. На рисунках 3.17 та 3.18 наведено приклади перевірки коректності роботи сортування за зростанням та спаданням.

```
it('сортує за ціною за зростанням (sortBy=1)', async () => {
  const res = await request(app)
    .get('/api/pizzas')
    .query({ sortBy: 1 }) // /api/pizzas?sortBy=1
    .expect(200);

  //отримуємо масив цін та переконуємось, що вони відсортовані ASC
  const prices = res.body.map((p) => p.price);
  expect(prices).toEqual([...prices].sort((a, b) => a - b));
});
```

Рисунок 3.17 – Тест сортування за зростанням

```
it('сортує за ціною за спаданням (sortBy=2)', async () => {
  const res = await request(app)
    .get('/api/pizzas')
    .query({ sortBy: 2 }) // → /api/pizzas?sortBy=2
    .expect(200);

  const prices = res.body.map((p) => p.price);
  expect(prices).toEqual([...prices].sort((a, b) => b - a));
});
```

Рисунок 3.18 – Тест сортування за спаданням

На рисунку 3.19 показано приклад виводу в консоль, який з’являється в разі успішного проходження всіх інтеграційних тестів. Такий результат підтверджує, що основні компоненти системи функціонують належним чином і взаємодіють один з одним злагоджено.

```
Test Suites: 2 passed, 2 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        1.549 s
```

Рисунок 3.19 – Вивід у консоль при виконанні всіх тестів

Завдяки використанню Jest та Supertest було реалізовано надійний набір інтеграційних тестів, що охоплюють ключові функціональні можливості

вебдодатку. Реєстрація користувачів, отримання асортименту піц, фільтрація та сортування – всі ці сценарії були протестовані в ізольованому середовищі, що забезпечує високу надійність результатів.

3.4 Висновки до розділу 3

В процесі реалізації вебдодатку піцерії були реалізовані ключові компоненти, що забезпечують повноцінну роботу системи. Клієнтська частина, створена на основі бібліотеки React, реалізована у вигляді незалежних компонентів, які взаємодіють між собою та забезпечують зручний і динамічний інтерфейс для користувачів.

Серверна частина побудована з використанням Node.js та фреймворку Express, що дозволило реалізувати RESTful API з чітким поділом логіки на маршрути, контролери та моделі. Реалізовано механізми аутентифікації та авторизації на основі JWT. Контролери обробляють запити до основних сутностей системи з урахуванням фільтрації, сортування та взаємозв’язків даних.

Для перевірки працездатності додатку були розроблені інтеграційні тести з використанням Jest та Supertest. Тести охоплювали такі критичні сценарії, як реєстрація користувача, отримання списку піц, фільтрація за категоріями та сортування за ціною.

Таким чином, основний функціонал вебдодатку піцерії було успішно реалізовано та протестовано.

ВИСНОВКИ

Результатом проєкту є розроблений вебзастосунок, який включає в себе сучасний користувацький інтерфейс, серверну частину та базу даних. Проєкт охоплює всі основні етапи життєвого циклу програмного забезпечення – від аналізу вимог і проєктування до реалізації та тестування.

Клієнтська частина системи реалізована з використанням бібліотеки React, яка забезпечує гнучкий компонентний підхід, дозволяє легко оновлювати інтерфейс у відповідь на зміни даних та підтримує повторне використання логіки. Завдяки використанню React Context API забезпечується ефективне управління глобальним станом додатку, зокрема для реалізації кошика замовлень та системи авторизації.

Серверна частина була створена з використанням Node.js та Express.js, що дозволило організувати RESTful API для взаємодії клієнта з базою даних. Реалізована система аутентифікації на основі JWT, яка забезпечує безпечний доступ до персональних даних користувача. Обробка запитів організована за принципами розподілу обов’язків, що спрощує підтримку та масштабування проєкту. Для реалізації бази даних була використана реляційна система PostgreSQL. За допомогою ORM-бібліотеки Sequelize була спроєктована повноцінна структура бази даних, що охоплює всі ключові сутності – користувачів, замовлення, піци, кошики, категорії та параметри піци (розмір, тісто).

Таким чином, мета розробки функціонального вебзастосунку для піцерії була повністю досягнута. Створений програмний продукт є результатом комплексної технічної реалізації, що поєднує в собі сучасні засоби розробки фронтенду та бекенду, засновані на принципах надійності, масштабованості та зручності. Отримані результати можуть бути використані як основа для подальшого розширення функціоналу, інтеграції з платіжними системами, системами доставки та впровадження в реальну інфраструктуру бізнесу.

ПЕРЕЛІК ПОСИЛАНЬ

1. Архітектура застосунку: компоненти та популярні патерни. *FoxmindEd*. URL: https://foxminded.ua/arkhitektura-zastosunku/ (дата звернення: 14.03.2025).

2. Початок роботи. *Jest*. URL: https://jestjs.io/uk/docs/getting-started (дата звернення: 21.03.2025).

3. Built-in react hooks. *React*. URL: https://react.dev/reference/react/hooks (дата звернення: 22.04.2025).

4. Component. *React*. URL: https://react.dev/reference/react/Component (дата звернення: 11.02.2025).

5. JWT decoder. URL: https://fusionauth.io/dev-tools/jwt-decoder (дата звернення: 15.03.2025).

6. Khatri M. F. API Testing using SuperTest. *Medium*. URL: https://medium.com/@iamfaisalkhatri/api-testing-using-supertest-ea37522fa329 (дата звернення: 21.03.2025).

7. Passing props to a component. *React*. URL: https://react.dev/learn/passing-props-to-a-component (дата звернення: 12.02.2025).

8. React virtual DOM. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/reactjs-virtual-dom/ (дата звернення: 11.02.2025).

9. Virtual DOM and internals. *React*. URL: https://uk.legacy.reactjs.org/docs/faq-internals.html (дата звернення: 11.02.2025).

10. What is next.js? *Sanity.io*. URL: https://www.sanity.io/glossary/next-js (дата звернення: 14.02.2025).