МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЗАОЧНОЇ ТА ПІСЛЯДИПЛОМНОЇ ОСВІТИ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему: **“ Використання резидентної бази даних Redis на NestJS для підвищення продуктивності веб-сервера ”**

Виконав: _____ст. гр. Іт-71з_____

Спеціальності 126 – «Інформаційні системи та технології»_____
(шифр і назва)

____Пасічник Іван Миколайович_____
(Прізвище та ініціали)

Керівник: __к.т.н., доц. Луб П.М._____
(Прізвище та ініціали)

Рецензент: _к.т.н., доц. Кригуль Р.Є._____
(Прізвище та ініціали)

ДУБЛЯНИ-2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ЗАОЧНОЇ
ТА ПІСЛЯДИПЛОМНОЇ ОСВІТИ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Освітній ступінь «Магістр»
126 – «Інформаційні системи та технології»

"ЗАТВЕРДЖУЮ"
Завідувач кафедри_____
д.т.н., проф. А.М. Тригуба
"____"_____2023 р.

ЗАВДАННЯ

на кваліфікаційну роботу студенту

Пасічник Іван Миколайович

1. Тема роботи: «Використання резидентної бази даних Redis на NestJS для підвищення продуктивності веб-сервера»

Керівник роботи Луб Павло Миронович, к.т.н., доцент
Затверджені наказом по університету 28.04.2023 року № 133/к-с.
2. Строк подання студентом роботи 10.01.2025 р.
3. Початкові дані до роботи: 1. Ситстема Redis; 2. Вимоги до структури веб-сервера; 3. Технологія NestJS Framework; 4. Стратегія аутентифікації – PassportJS.

4. Зміст розрахунково-пояснювальної записки:

1. *Аналіз вимог та архітектури веб-серверів.*
2. *Особливості технологій розгортання архітектури веб-серверів.*
3. *Методика інтеграції із базою даних.*
4. *Проектування та програмування елементів веб-сервера.*
5. *Охорона праці та безпека в надзвичайних ситуаціях.*
6. *Оцінка використання кеш сервісу для веб-сервера.*

Висновки та пропозиції.
Бібліографічний список.
Додатки.

5. Перелік графічного матеріалу: 1 та 2 – Тема, мета, завдання роботи; 3 – Аналіз системи Redis; 4 – Архітектура веб-сервера; 5 – Безпека та аутентифікація; 6 та 7 – Інтеграція з базою даних; 8 – Схема бази даних; 9 – Документація та API; 10 – Діаграми програмної реалізації веб-сервера; 11 – Фрагменти програмної реалізації веб-сервера; 12 та 13 – Результати практичного використання веб-сервера; 14 – Порівняльна оцінка використання кеш сервісу для веб-сервера; 15 – Головні висновки робти.

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1, 2, 3, 4, 6	*Луб П.М., доцент кафедри інформаційних технологій*		
5	*Городецький І.М., доцент кафедри фізики, інженерної механіки та безпеки виробництва*		

7. Дата видачі завдання – 28 квітня 2023 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Приміт-ка
1.	*Написання першого розділу та означення головних завдань роботи*	*28.04-20.06.24*	
2.	*Виконання другого розділу та опис інформаційних технологій для виконання завдань роботи*	*21.06-14.08.24*	
3.	*Виконання третього розділу, методика вирішення завдань та елементи наукових досліджень*	*15.08-31.10.24*	
4.	*Виконання четвертого розділу, опис розробки*	*15.08-31.10.24*	
5.	*Написання розділу: «Охорона праці та безпека в надзвичайних ситуаціях»*	*01.11-10.11.24*	
6.	*Головні результати отримані в роботі, оцінення розроблених пропозицій*	*20.11-30.11.24*	
7.	*Завершення оформлення розрахунково-пояснювальної записки та презентаційних матеріалів*	*30.11-01.12.24*	
	Завершення роботи в цілому	*01.12.24-10.01.25*	

Студент _____ Пасічник І.М.
(підпис)

Керівник роботи _____ Луб П.М.
(підпис)

УДК: 004.451.4:004.432:004.77

Кваліфікаційна робота: 75 с. текст. част., 33 рис., 2 табл., 13 слайдів, 30 джерел.

Використання резидентної бази даних Redis на NestJS для підвищення продуктивності веб-сервера. Пасічник І.М. Кафедра ІТ. – Дубляни, Львівський НУП, 2025.

У роботі здійснено аналіз вимог та архітектури веб-серверів із фокусом на сучасні підходи до проектування та розгортання. Розглянуто відкриту систему управління даними Redis, патерни проектування веб-застосунків, а також переваги й недоліки монолітної та мікросервісної архітектур.

Детально описано етапи переходу до мікросервісів, особливості клієнт-серверної взаємодії, модульну архітектуру з використанням NestJS та стратегії захисту даних користувачів.

Особливу увагу приділено методикам інтеграції веб-серверів із базами даних, зокрема комплексних застосунків, типових схем взаємодії та документації API.

Наведено результати проектування та програмування веб-сервера, включаючи розробку бази даних, інструкції з використання сервісу та оцінку впливу кеш-сервісів на продуктивність.

Робота спрямована на покращення продуктивності веб-серверів, оптимізацію архітектури, підвищення надійності та ефективності обробки даних, що робить її корисною для розробників сучасних веб-застосунків.

Означено вимоги охорона праці та безпека в надзвичайних ситуаціях.

Ключові слова: веб-сервер, архітектура, стек технологій, бази даних, Redis, NestJS, Framework, функціональність.

ЗМІСТ

ВСТУП

На даний час доступ користувачів до мережі Інтернет є майже в будь-якому куточку світу із будь-якого пристрою тощо. Завдяки такому широкому та швидкому розповсюдженню інформаційних технологій користувачі мають доступ до безкоштовного перегляду відео у високій якості, до поточної фінансової інформації, світових подій, бізнес-процесів, баз знань, банківських та науково-пізнавальних даних тощо [4]. Водночас, користувачі не обмежуються ресурсами розміщеними в тій країні де він фізично знаходиться, а мають доступ майже до будь-якого ресурсу та архівних даних.

Розробляючи застосунки, програмісти та системні архітектори зіштовхуються з проблемою вибору архітектури серверів та веб-додатків. Адже правильно вибрана архітектура дає змогу успішно спроектувати та запрограмувати діючий і стабільний продукт. Саме тому розробники у першу чергу обговорюють і розробляють загальну концепцію, а потім і тенденції, переваги та недоліки кожної із існуючих архітектур тощо. Однак, у кожного замовника є своя кон’юнктура бізнесу і комерційна потреба «перевершити конкурента» і саме через це використання загального шаблону не завжди є помічним.

Під час розробки майбутньої програмованої системи, програміст повинен розуміти, що додаток побудований на тій, чи іншій архітектурі, повинен мати такі характеристики, як відмовостійкість, розширюваність, надійність, масштабованість, супроводжуваність, безпеку, зручність використанні, тощо.

Відповідно до цього, актуальність кваліфікаційної роботи із вирішення завдання щодо розробки архітектури веб-сервера із застосуванням Node.js та NestJS Framework є досить актуальною.

Мета роботи – розробити архітектуру веб-сервера із використанням резидентної бази даних Redis на NestJS Framework, що забезпечуватиме високефективність та маштабованість його роботи.

Завдання роботи: 1) проаналізувати архітектури веб-серверів і технологій

управління даними; 2) розробити архітектуру веб-сервера; 3) виконати інтеграцію веб-сервера із базами даних; 4) виконати реалізацію та оцінка продуктивності.

Об’єкт роботи – веб-сервер інтернет-магазину, розроблений із використанням Redis та NestJS Framework.

Предмет роботи – показники роботи веб-сервера, його архітектура, стек технологій, проектування елементів підвищення функціональності.

Новизна результатів:

- ✓ застосовано відкриту резидентну базу даних Redis для оптимізації обробки запитів і кешування, що зменшує навантаження на основну базу даних;
- ✓ реалізовано модульну архітектуру веб-сервера із застосуванням сучасного фреймворка NestJS;
- ✓ запропоновано етапи переходу від монолітної до мікросервісної архітектури, враховуючи специфіку веб-застосунків, інтеграцію баз даних і вимоги до масштабованості системи;
- ✓ реалізовано стратегії забезпечення безпеки даних користувачів у клієнт-серверній архітектурі.

Практичне значення результатів. Використання технологій Node.js, NestJS, Prisma, PostgreSQL, а також інтеграція з Backblaze B2 Storage та впровадження багатошарової системи аутентифікації через Passport.js, створює високопродуктивну основу для веб-додатків, здатного ефективно обробляти великі обсяги даних і забезпечувати надійний захист інформації.

РОЗДІЛ 1
АНАЛІЗ ВИМОГ ТА АРХІТЕКТУРИ ВЕБ-СЕРВЕРІВ

1.1. Аналіз відкритої системи управління даними Redis

Redis (скорочено від REmote DIctionary Server) є однією з найпопулярніших відкритих систем управління даними типу "ключ-значення" [26]. Ця система використовується для різноманітних завдань і має безліч застосувань завдяки своїй швидкості та надійності.

Redis є відкритою системою управління даними, розробленою у 2009 році Сальватором Санфіліппо. Її ше називають – документно-орієнтованою базою даних. Основна ідея полягає у тому, щоб надати швидкий інтерфейс для роботи з даними. Кожен ключ у пов'язаний з певним значенням, і ці дані зберігаються у пам'яті комп'ютера, що дозволяє отримувати доступ до них надзвичайно швидко.

Redis позиціонується як сервер кешування, але його можливості виходять далеко за межі цієї ролі. Його також можна експлуатувати як базу даних, систему повідомлень, робочу пам'ять для інших додатків та навіть як координатор для розподілених систем.

Redis – розподілене сховище пар ключ-значення, які зберігаються в оперативній пам'яті, з можливістю забезпечувати довговічність зберігання на бажання користувача. Це програмне забезпечення з відкритим початковим кодом написане на ANSI C.

Redis надає схожі на Memcached функції для зберігання даних в форматі ключ/значення, розширені підтримкою структурованих даних, таких як списки, хеші і множини. На відміну від Memcached, Redis забезпечує постійне зберігання даних на диску і гарантує збереження БД у разі аварійного завершення роботи. Клієнтські бібліотеки доступні для більшості популярних мов, включаючи Perl,

Python, PHP, Java, Ruby і Tcl.

Систему застосовують у різних сферах завдяки унікальним можливостям.

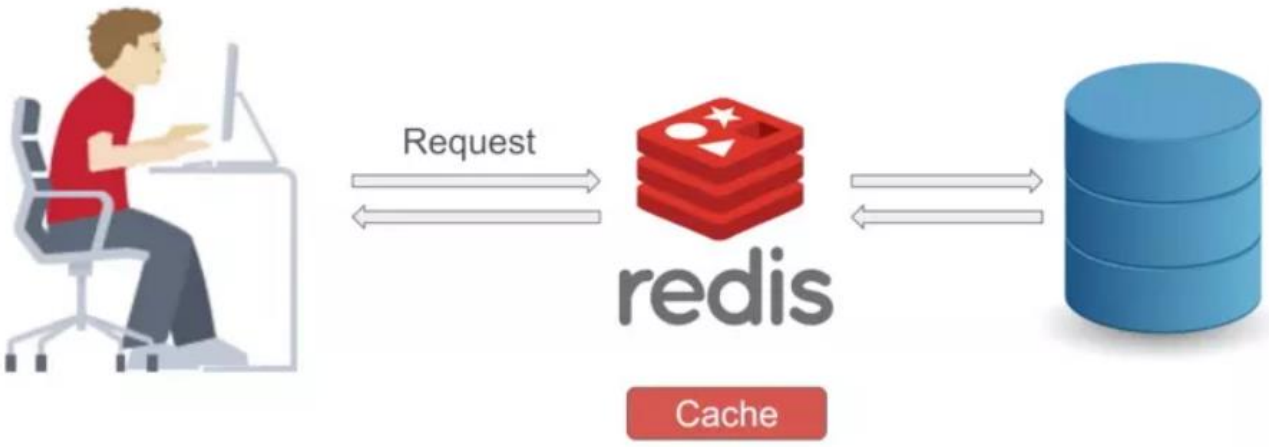

Рисунок 1.1 – Схема використання системи Redis [26]

Популярні сценарії використання системи наступні [26]:

Кешування: Redis підходить для реалізації інвентарного кешування в пам'яті з високою доступністю. Він дозволяє зменшити затримки доступу до даних, збільшити пропускну здатність та полегшити навантаження на реляційну або NoSQL базу даних і додаток. Система може обслуговувати запити до часто запитуваних об'єктів з відповідями менше однієї мілісекунди та може легко масштабуватися для роботи з вищими навантаженнями, не збільшуючи витрат на бекенд.

Чат, месенджери: Redis підтримує Pub/Sub з можливістю використання шаблонного зіставлення і різноманітні структури даних: списки, впорядковані множини та хеші. Це дозволяє системі підтримувати високопродуктивні чат-кімнати, потоки коментарів у реальному часі, стріми соціальних медіа та серверну взаємодію. Структура даних Redis List дозволяє легко реалізувати чергу.

Лідерборди в іграх: Redis користується популярністю серед розробників ігор, які хочуть створити лідерборди в реальному часі. Для цього використовують структуру даних Sorted Set, яка надає унікальності елементам і водночас підтримує впорядкування рейтингового списку. Створювати список в реальному часі легко, просто оновлюючи бал користувача при кожній зміні.

Збереження сесій: Систему застосовують для збереження й управління даними сесій з метою масштабування інтернет-застосунків. Вона надає мілісекундну затримку, масштаб і надійність, необхідні для управління даними сесій. Серед них: профілі користувачів, облікові дані, стан сесії та персоналізація користувача.

Стрімінг медіа: Redis пропонує швидке сховище даних в пам'яті для потужних сценаріїв. Систему можна використовувати для зберігання метаданих профілів користувачів і історії перегляду, інформації/токенів аутентифікації для мільйонів користувачів і файлів маніфестів для можливості потокової передачі відео на мільйони мобільних та десктопних пристроїв одночасно.

Геопросторові дані: Redis пропонує спеціалізовані вбудовані структури даних і оператори для управління геопросторовими даними в реальному часі на велику відстань і швидкість. Команди GEOADD, GEODIST, GEORADIUS і GEORADIUSBYMEMBER для зберігання, обробки та аналізу геопросторових даних в реальному часі роблять процес легким і швидким. Можна використовувати систему для додавання локалізованих функцій.

Машинне навчання: Сучасні додатки, орієнтовані на дані, потребують машинного навчання для швидкої обробки масивного обсягу, різноманітності й швидкості даних та автоматизації прийняття рішень. Для виявлення шахрайства в іграх і фінансових послугах, процесу вибору в сервісах дейтингу чи поїздок, можливість обробки живих даних і прийняття рішень за декілька десятків мілісекунд має велике значення.

Аналітика в реальному часі: Redis може використовуватися разом з потоковими рішеннями Apache Kafka та Amazon Kinesis, як сховище даних в пам'яті для введення, обробки й аналізу даних в реальному часі з затримкою менше однієї мілісекунди.

Основні функції Redis, які роблять його дуже корисним:

- зберігає дані у пам'яті, що робить доступ до них надзвичайно швидким;
- використовує просту структуру ключ-значення, що спрощує роботу з даними;

- підтримує ряд типів даних, серед яких рядки, хеші, списки, множини та багато інших;
- дозволяє групувати операції в транзакції для атомарності;
- підтримує механізм для створення системи обміну повідомленнями;
- може бути розгорнутий у режимі кластера для забезпечення високої доступності й масштабованості;
- має механізми резервного копіювання і відновлення даних для забезпечення надійності.

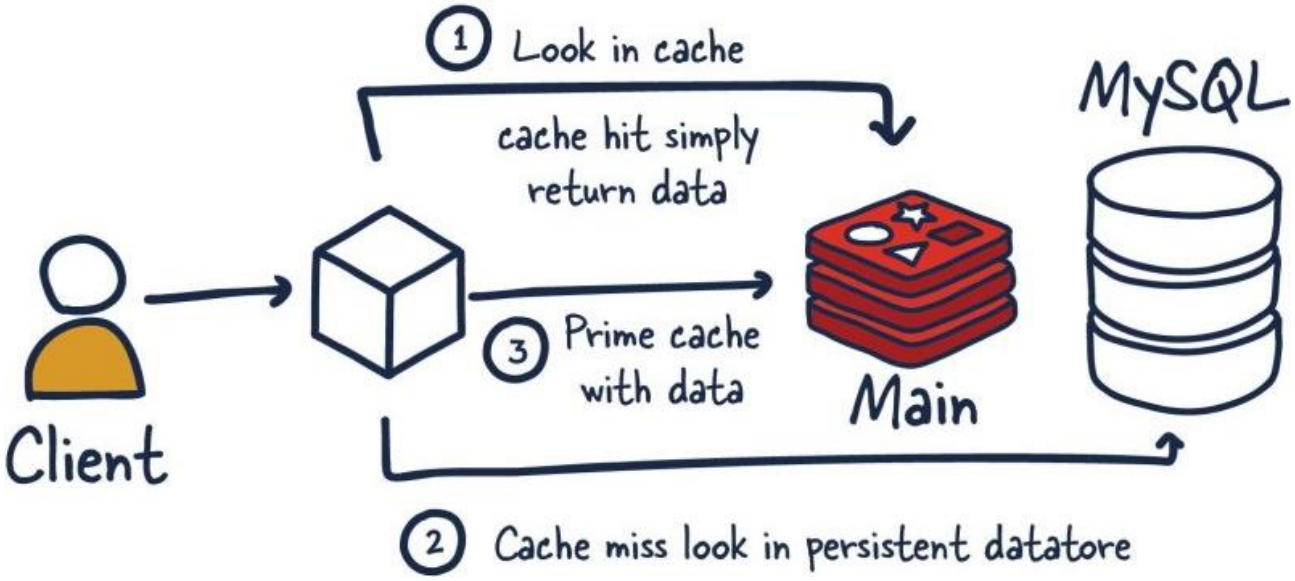

Рисунок 1.2 – Схема класичного використання системи Redis [26]

Redis — це потужний інструмент для роботи з даними, який відрізняється високою швидкістю та надійністю.

1.2. Патерни проектування веб-застосунків

Патерни скеровані на розділення елементів системи та повторне використання певного шаблону, надаючи рішення часто повторюваним проблемам. Коли певна проблема вирішена багатьма розробниками певним чином, і це трактується як ефективне рішення – це стає патерном проектування

веб-застосунку. Патерни використовують задану патерн-схему [14]. Вони мають такі компоненти:

- контекст (ситуація, що створює певну прикладну проблему);
- власне, сама проблема;
- рішення (подана зі структурними елементами).

Нам необхідно правильно підібрати архітектуру веб-застосунку за відповідними причинами [24]. Команда розробників програмного забезпечення приймає перші рішення саме під час вибору моделі архітектури. Для цього етапу доступні тільки бізнес – вимоги. Перший набір рішень являється вирішальним в любому проекті, оскільки неоптимальні рішення можуть привезти до збільшення або втрати бюджету з боку клієнта. Зацікавленим сторонам необхідно знати про хід робот над проектом. Вона діє як інстурмент візуальної коммунікації в команді. Вони будуть розуміти яку систему будується.

Патерни проектування полегшують повторне використання інструментів та підходів в інших та схоєжих проектах. Архітектура програмного забезпечення не обмежується архітектурним стилем, а часто являється комбінацією декількох, створюючих повну систему. На вибір патерну впливає багато факторів. Ці фактори включають в себе можливості команди, яка створює та реалізує веб-застосунок; досвід команди; інфраструктура та організаційні обмеження.

На даний момент виділяють наступні патерни проектування [19]:

- клієнт/Сервер;
- model – view – controller;
- n - tier архітектура;
- шарова архітектура *(* Layered Architecture);
- сервісно-орієнтована архітектура;
- об’єктно-орієнтована архітектура;
- p2p (Peer – to – peer).

Першим розглянемо **MVC**. Завдяки цьому шаблону систему можна поділити на три частини: модель, представлення (вигляд) і контроллер. Таким чином, модифікація кожного компонента може здійснюватися незалежно.

Концепція шаблону модель–вигляд–контролер (MVC) була описана Трюгве Реенскаугом в 1978 році, що працював в науково-дослідному центрі "Xerox PARC" над ПЗ "Smalltalk". Пізніше, Стів Бурбек реалізував шаблон в Smalltalk - 80. Остаточна версія концепції MVC була опублікована лише в 1988 році в журналі Technology Object. Згодом шаблон проектування став еволюціонувати – була представлена ієрархічна версія HMVC; MVA, MVVM.

Основна мета застосування цієї концепції полягає у відділенні бізнес-логіки (моделі) від її візуалізації (представлення, виду). За рахунок такого розділення підвищується можливість повторного використання коду. Найбільш корисне застосування цієї концепції в тих випадках, коли користувач повинен бачити ті ж самі дані одночасно в різних контекстах і/або з різних точок зору.

Приклад даного патерну представлений на рисунку 1.3.

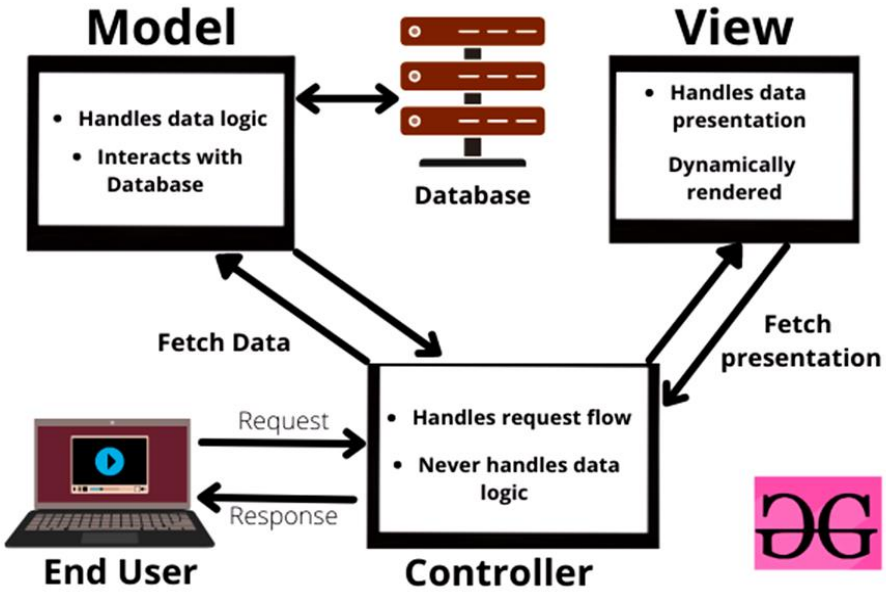

Рисунок 1.3 – Модель патерну MVC

У рамках архітектурного шаблону MVC програма поділяється на три окремі частини з розподілом функцій між компонентами.

Модель (***Model***) відповідає за зберігання даних та їх структуру.

Вигляд (***View***) відповідальний за інтерфейс програми.

Контролер (***Controller***) керує компонентами, отримує сигнали у вигляді реакції на дії користувача та передає дані у модель.

Модель являється центральним компонентом та відображає поведінку застосунку, незалежну від інтерфейсу користувача. Модель має безпосереднє відношення до прямого керування даними, логікою і правилами застосунку. Події, які зареєстровані, транслюються в різні запити, що спрямовуються компонентам моделі або об'єктам. Саме вони являються відповідальними за відображення необхідних даних. Відокремлення моделі від вигляду даних дозволяє незалежно використовувати різні компоненти для відображення інформації. Таким чином, вносячи зміни через контролер користувач до моделі даних, то інформація, подана одним або декількома візуальними компонентами, буде автоматично редагуватись відповідно до змін, що відбулися.

Існує декілька видів даного патерну. Найбільш поширені [1, 3]:

- Model-View-Controller;

- Model-View-Presenter;

- Model-View-View Model.

Цей патерн являється популярним. Багато веб-застосунків та веб-фраєморків, такі як Spring и Rails використовують данний патерн.

Розглянемо його переваги:

- використання даної модели пришвидчує розробку;

- команда розробників можуть представити користувачам декілька представлень;

- модель не форматує дані користувача, тому можуть використовувати цей патерн з любим інтерфейсом.

Також, розглянемо декілька недоліків даного підходу, наприклад:

- при використанні даного патерну, в коді з’являються нові шари програми, що складнює дану програму;

- як правило, даний патерн потребує використання декількох технологій, що збільшує час на його створення.

Мікросервіси. Програмна архітектура має не багато спільного з функціональними вимогами. Можна реалізувати набір сценаріїв (функціональних вимог до програми) з використанням будь-якої архітектури.

Звісно, архітектура також важлива, бо вона визначає так названі вимоги до якості обслуговування, відомі також як нефункціональні вимоги або атрибути якості. Сьогодні все більше спеціалістів збігаються на тому, що при будуванні великої, складної програми слід замислитися над використанням мікросервісів.

У мікросервісної архітектури є безліч визначень. Перші розуміють назву занадто буквально та затверджують, що сервіс повинен бути дуже малим, наприклад складатися зі 100 рядків коду. Інші вважають, що на розробку сервісу повинно відходити не більше двох тижнів.

Модель представлена в рисунку 1.4. Масштабування по осі X часто використовують в монолітних програмах. Принцип праці цього підходу такий: завантажуєтсья декілька прототипіва програми. Балансувальник розподіляє запити між однаковими екземплярами. Це відмінний спосіб покращити потужність та доступність програми.

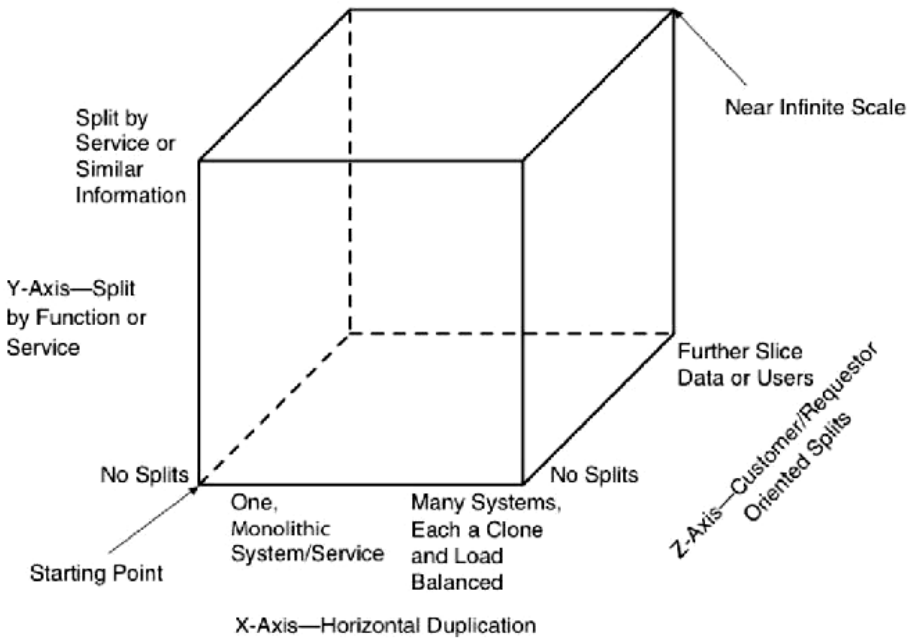

Рисунок 1.4 – Тривимірна модель масштабування

Масштабування по осі Z також передбачає запуск певних екземплярів монолітної програми, але в цьому випадку, на відміну від масштабування по осі X, кожний екземпляр відповідає за певну підмножину даних.

Маршрутизатор, що виставлений спереду, задіє атрибут запиту, щоб направити його до відповідного екземпляра. Для цього, наприклад, можна використовувати поле userId. В даному сценарії екземпляр програми відповідає за перелік користувачів. Маршрутизатор може перевірювати поле userId, що вказане в заголовку запиту авторизації, щоб обрати одну з ідентичних копій програми.

Масштабування за осями X та Z збільшує потужність та доступність програми [14]. Але не один з цих підходів не вирішує проблем з ускладненням коду та процесу розробки. Щоб впоратися з ними, потрібно використати масштабування по осі Y, або функціональну декомпозицію (розподіл). Те, як це працює, представлено на рисунку 1.5: монолітна програма розподіляється на окремі сервіси.

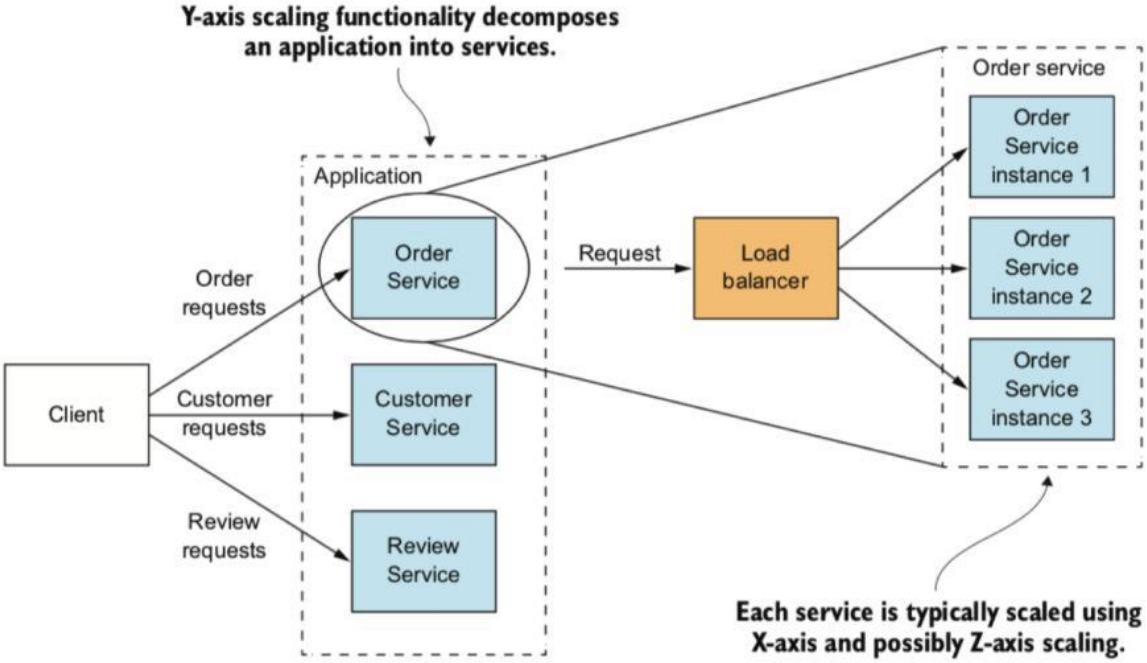

Рисунок 1.5 – Масштабування по осі Y

Це масштабування розподіляє програму на окремі сервіси. Вони відповідають за певну функцію та масштабуються по осі X і можливо, по осі Z.

Сервіс – це програма, що реалізує вузькоспеціалізовані функції. Наприклад це може бути управління клієнтами, управління запитами тощо.

Сервіси масштабуються по осі X, деякі з них можуть використовувати також ось Z. Наприклад сервіс Order має декілька копій, навантаження на які

балансуються. Загальне визначення мікросервісної архітектури (мікросервісів) - це стиль проектування, що розподіляє програму на певні сервіси за різними функціями.

В монолітному проекті модулі представляють собою поєднання концепцій мов програмування, таких як пакети Java, та ресурсів, що приймають участь в зборці, такі як JAR-файли. Одиницею модульності в мікросервісній архітектурі являється сервіс. Сервіси мають API, які служать непроникним бар'єром. На відміну від пакетів в Java API неможливо обійти, щоб звернутися до внутрішнього класу. У перспективі це набагато спрощує підтримку модульності програми.

Мікросервіси піддаються критиці, що в данному підході немає нічого нового. Вказувалось, що це різновидність сервіс-орієнтованої архітектури (service-oriented architecture, SOA). Але,проводячи аналіз на вищому рівні існує деяка схожість. SOA та мікросервісна архітектура – це стилі проектування, які структурують систему в перелік сервісів. Але при більш детальному розгляді можна знайти відмінності (таблиця 1.1).

Таблиця 1.1 – Порівняння SOA та мікросервісів [8, 16, 20, 30]

№ п/п	Параметр	SOA	Мікросервіси
1	Міжсервісна взаємодія	SOAP – як приклад каналів, такі як сервісна шина підприємства, з використанням великовагових протоколів	Примітивні канали, яка являє собою прямою взаємодією між сервісами за допомогою протоколів, наприклад REST або Grpc
2	Дані	Глобальна модель даних та загальні БД	Розподілені моделі даних та БД для кожного сервісу
3	Типовий сервіс	Монолітний застосунок	Сервіс невеликий за розміром

SOA і мікросервісна архітектура загалом використовують різні технології. У програмах на основі SOA використовуються великовагові стандарти веб-сервісів збіжно до SOAP. Їм часто потрібна шина підприємства (*Enterprise Service Bus, ESB*) – smart канал, який інтегрує сервіси завдяки бізнес-логіки та коду для обробки повідомлень. Програми, що спроектовані у виді мікросервісів, зазвичай

задіють легковагові технології з відкритим вихідним кодом. Сервіси взаємодіють через примітивні канали подібні до REST чи gRPC.

Мікросервісна архітектура має наступні переваги:

- вона робить можливими безперервну доставку та розгортання великих, складних програм;

- сервіси виходять невеликими та простими в обслуговуванні;

- сервіси розгортаються незалежно один від одного;

- сервіси масштабуються незалежно один від одного;

- мікросервісна архітектура забезпечує автономність команд розробників;

- вона дозволяє експериментувати та впроваджувати нові технології;

- в неї краще ізольовані несправності.

Очевидно, що ідеальних технологій не існує та в мікросервісній архітектурі теж є ряд суттєвих недоліків та проблем.

Такі є основні недоліки та проблеми мікросервісної архітектури:

- важко підібрати відповідний набір сервісів;

- складність розподілу систем ускладнює розробку, тестування та розгортання;

- розгортання функцій, що охоплюють декілька сервісів, потребує ретельної координації;

- рішення про те, коли слід переходити на мікросервісну архітектуру, є нетривіальним.

1.3. Переваги та недоліки веб-серверів за монолітної й мікросервісної архітектури

Архітектура повинна задовольняти поставлені вимоги, виконувати чітко свої бізнес-задачі, використовувати певні архітектурні стилі, різні шаблони проектування, які полегшують розробку та роблять систему більш гнучкою та масштабованою. Адже, якщо до кінця не буде правильно продумано задовільного

та обґрунтованого рішення, то в наслідок цього, розробка може «затягнутися» та призвести до великих змін в вже побудованій системі [8].

Основними функціями програмної архітектури є мінімізувати складність системи, зрозуміти, як краще зробити розбиття на частини великих компонентів, як правильно побудувати взаємодію між ними, як буде відбуватися обмін інформацією між цими компонентами, чи можна буде додатково розширити їх без великої зміни в інших, чи можемо це зробити за допомогою абстрагування та розділення задач, тощо.

Потрібно визначити, який архітектурний шаблон треба використовувати, щоб побудувати програмне забезпечення, яке буде стійким до збоїв, готовим до розширення, мати високий рівень безпеки, масштабованим, надійним та конфігурованим. Вибрати базу даних, яка буде в змозі обробляти достатню кількість запитів від клієнта. Сайт повинен мати стильний дизайн, бути зрозумілим для користувача і мав змогу зацікавлювати більше нових клієнтів.

Одними із популярних шаблонів проектування в програмуванні є монолітна та мікросервісна архітектури. Окрім аналізу і формулювання вимог, дані архітектури надають міцний фундамент для розробки програмного забезпечення.

Монолітна архітектура (рис. 1.6) — це традиційна уніфікована модель розробки програмного забезпечення. Монолітний у цьому контексті означає складений з одного шматка [8]. Він являється деяким додаток, розгортання якого, здійснюється наприклад в вигляді JAR-файлу, або Node з єдиною точкою входу. При побудові додатка на основі моноліта, всі процеси будуть управлятися всередині одного модуля.

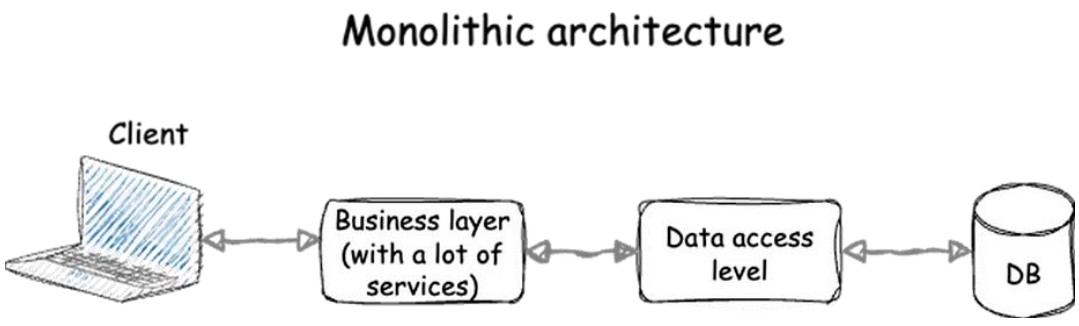

Рисунок 1.6 – Монолітна архітектура веб-серверів

Монолітна архітектура будується на основі клієнт-сервер, в якій функції презентації, обробки програм та управління даними фізично розділені. Рівень презентації відображає різну інформацію, на прикладі застосунку доставки, це може бути різні партнери, послуги, продукти, які доступні на цьому сайті. Рівень обробки програм займається бізнес-логікою, яка в свою чергу призначена для контролю функціональності програми, щоб виконувати детальні обробки. Рівень управління даними призначений для розміщення серверів баз даних, де в свою чергу зберігається інформація. Монолітний підхід є стандартною моделлю побудови та створення програмного забезпечення.

Переваги:

– Легко реалізувати. Даний підхід є стандартним способом створення застосунків, тому, будь-яка команда може мати необхідні знання та можливості для розробки застосунків на основі монолітної архітектури.

– Простий в розгортанні. В кінці розробки монолітної системи не потрібно працювати з багатьма розгортаннями — лише один файл або каталог.

– Високий рівень безпеки.

– Розробникам не потрібно вивчати різні програми, адже можуть зосередитися лише на одній програмі.

– Проблеми затримки мережі та безпеки відносно менші.

– Простіше тестування та налагодження. На відміну від мікросервісної архітектури, монолітну — набагато легше налагоджувати та тестувати. Оскільки вона є єдиною та неподільною і може запускати наскрізне тестування набагато швидше.

– Конфігурованість.

– Менша кількість наскрізних проблем — це проблеми, які впливають на все програмне забезпечення, наприклад введення журналу, кешування, обробка, моніторинг продуктивності, тощо.

Недоліки:

– Все ядро працює в одному і тому ж самому адресному просторі, тобто збій в одному з компонентів програми, порушить працездатність всієї системи.

– Створення нового додаткового функціоналу може вплинути на деяку зміну в інших компонентах.

– Зі збільшенням розмірів застосунків, збільшується час запуску та розгортання.

– Масштабування програми може бути складним. Кожна копія екземпляра програми матиме доступ до всіх даних, що робить збільшення споживання пам’яті, а кешування менш ефективним. Зі збільшенням обсягу даних, архітектура можливо не зможе масштабуватися.

– Ознайомлення з великою кількістю коду, може займати великий час як серед розробників, так і серед «новачків».

– Систему, можливо буде важко розуміти та робити зміни, що в наслідок цього, розвиток програмного забезпечення сповільнюється, якість коду знижується, модульність з часом руйнується.

Мікросервісна архітектура — це архітектурний шаблон, в якому застосунок будується за рахунок невеликих сервісів, кожен з яких працює у своєму власному процесі та спілкується з іншими, використовуючи прості та швидкі протоколи передачі даних HTTP (рис. 1.7). Ці послуги будуються з урахуванням бізнес-можливостей і можуть бути незалежно розгорнуті за допомогою повністю автоматизованого механізму розгортання.

Існує мінімум централізованого управління цими службами, які можуть бути написані різними мовами програмування та використовувати різні технології зберігання даних [15]. Ключові концепції, які потрібно використовувати при розробці мікросервісної архітектури — це декомпозиція та розгрупування. Функціональність додатків має бути повністю незалежною одна від одної.

Мікросервісна архітектура має суттєвий вплив на зв’язок між застосунком і базою даних. Замість використання спільної бази даних з іншими мікросервісами, кожен з них має свою власну. Часто це може призвести до дублювання даних. Але будування бази даних на кожний мікросервіс є важливим аспектом.

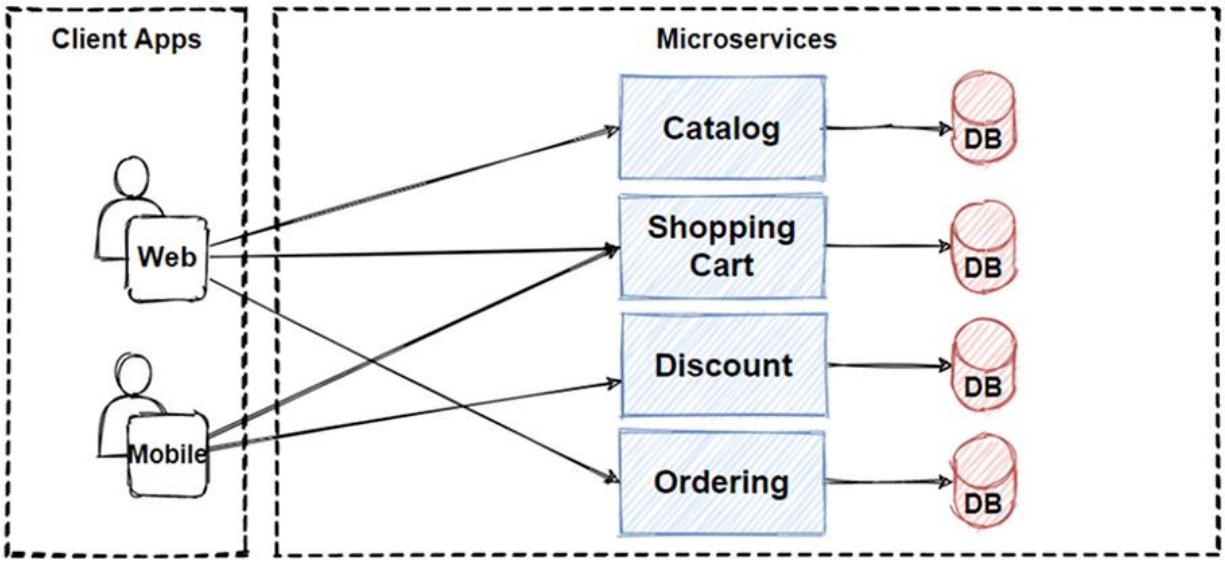

Рисунок 1.7 – Мікросервісна архітектура веб-серверів

Переваги:

– Кожен компонент розгортається незалежно від інших.

– Мікросервіси використовують прості протоколи зв’язку, такі як HTTP і JSON для обміну даними між сервісами.

– Логіка програми розбивається на невиликі компоненти з чітко визначеними межами відповідальності.

– Масштабованість. Система здатна обробляти великий обсяг роботи та бути легко розширеною. Якщо певний мікросервіс стикається з великим навантаженням через те, що клієнти використовують його в надлишку, тоді потрібно масштабувати лише цей мікросервіс. Це означає те, що мікросервісна система підтримує горизонтальне масштабування.

– Висока стійкість до відмов: збій в одному сервісі не вплине на інші. Таким чином, проблеми в інших сервісах, не завадять всій робочій системі.

– Простота — через невеликий розмір сервісу, коду в ньому менше, і тому не потрібно розбиратися у великій кількості деталей реалізацій, що в результаті дає розробникам менше часу для того, щоб розібратися, як система працює.

– В кожному сервісі можна вибрати або замінити стек технологій.

– Краще організовані. Через їх незалежний і розподілений характер, вони

дозволяють організаціям мати менші групи розробників із чітко визначеними сферами відповідальності.

Недоліки:

– Складність створення розподіленої системи. Дана складність зростає зі збільшенням кількості мікросервісів.

– Обмін повідомленнями між сервісами має складний характер. Тому що, кожен функціональний елемент ізольований. За рахунок цього потрібна особлива ретельність при побудові комунікації між ними. Зі збільшенням кількості сервісів, складність побудови комунікації зростає.

– Зростання кількості сервісів, несе за собою характер зростання кількості баз даних. Тому, що мікросервісна архітектура повинна дотримуватися патерну «Одна база даних на сервіс». Якщо для кожного створеного сервісу використовувати одну й ту ж саму базу, це призведе до використання анти-шаблону, а отже, призведе до поганої побудови архітектури.

– Складність розгортання. Самостійне розгортання мікросервісів є складним.

– Складність тестування — спочатку потрібно розібратися з роботою кожного сервісу, а згодом, проводити тестування взаємодії його з іншими мікросервісами.

Кожна з цих архітектур є потужною складовою в розробці програмного забезпечення. Вони мають ряд, як позитивних, так і негативних харакетристик, які можуть вплинути на майбутню розробку. Після ознайомлення та вивчення, можна порівняти дані архітектури за відповідними характеристиками.

РОЗДІЛ 2
ОСОБЛИВОСТІ ТЕХНОЛОГІЙ РОЗГОРТАННЯ АРХІТЕКТУРИ ВЕБ-СЕРВЕРІВ

2.1. Головні етапи переходу на мікросервіси

Перетворення монолітної програми в мікросервіс являється різновидом модернізації програмного забезпечення (ПЗ). Модернізація ПЗ включає в себе процес перекладу застарілого коду на нові архітектуру і стек технологію. Розробники десятиліттями займаються модернізацією своїх застосунків, а отриманий досвід, який вони накопичують за цей час, можна застосувати для міграції на мікросервісну архітектуру. Самий важливий урок полягає в тому, що не варто переписувати проект з нуля. Є декілька проблем при переході на нову архітектуру [16]:

Перша – це зв'язність бізнес-логіки. При переході на мікросервіси це стає проблемою: увесь код досить жорстко пов'язаний, і складно розділити сервіси.

Друга – це необіхдніть в додатковій кількості розробиків, які будуть інтегрувати нові технології та зв’язки між сервісами програми.

Замість того щоб починати з чистого аркуша, можна поступово трансформувати свій монолітний проект (рисунок 2.1), побудувавши нову програму. Вона складається з мікросервісів, які будуть працювати у зв'язці з монолітним кодом. З часом монолітна програма буде реалізовувати все менше функцій до тих пір, поки повністю не зникне або ж не перетвориться в ще один мікросервіс. Це менш ризиковано, ніж спроба переписування з нуля.

Важлива перевага поступового переходу на мікросервісну архітектуру полягає в тому, що ви відразу ж бачите плоди своєї роботи. Це сильно відрізняється від переписування з нуля, про користь якого можна судити тільки по його закінченні. При поступовому рефакторінгу моноліту кожен новий сервіс можна писати за допомогою нового стека технологій і сучасного

високошвидкісного процесу розробки і розгортання. Завдяки цьому ваша команда буде доставляти свій код все швидше.

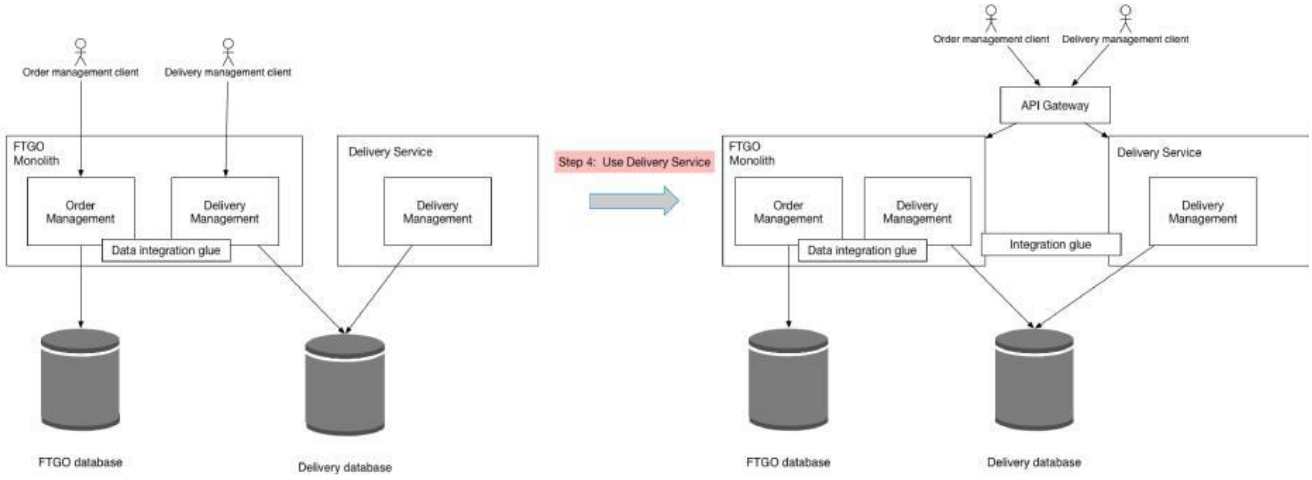

Рисунок 2.1 – Трансформація монолітної програми в мікросервісну

Єдине, без чого не можна обійтися – це процес розгортання з автоматичним тестуванням. Наприклад, якщо у вас всього кілька сервісів, вам не потрібні розвинені інструменти для розгортання та забезпечення спостережливості. З самого початку для пошуку послуг можна використовувати конфігурацію, вбудовану в вихідний код.

Існує три основні стратегії для «удушення» моноліту і поступової заміни його мікросервісами [20]:

а) реалізація нових можливостей у вигляді сервісів;

б) поділ рівня уявлення і внутрішніх компонентів;

в) розбиття моноліту шляхом оформлення функціональності у вигляді сервісів.

Перша стратегія запобігає подальший розвиток моноліту. Вона дозволяє швидко продемонструвати вигоду від використання мікросервісов, допомагаючи заручитися підтримкою керівництва. Реалізація нових можливостей у вигляді сервісів прискорює їх розробку. Це хороший спосіб швидко продемонструвати переваги мікросервісной архітектури. До того ж це уповільнює темпи розвитку моноліту. Дві інші стратегії розбивають моноліт на частини. Другий підхід може стати корисним у процесі рефакторінга, а без третього ви точно не обійдетеся,

оскільки саме так функціональність переноситься з моноліту в «задушливу» програму.

Альтернативна стратегія більше піклується про планування: модулям програми призначається рейтинг відповідно до того, яку користь планується отримати від їх вилучення. Існує дві причини, чому витяг сервісу може бути корисним:

а) прискорення розробки. Її прискорення шляхом можна збільшити через вилучення її в сервіс, якщо згідно з планом якась частина програми буде активно розвиватися протягом наступного року;

б) рішення проблем з продуктивністю, масштабованість та надійністю.

Якщо певна частина вашого застосування ненадійна або має проблеми з продуктивністю або масштабністю, буде корисно перетворити її в сервіс.

2.2. Особливості реалізації клієнт-серверної архітектури

Клієнт-серверна архітектура набула своєї популярності завдяки динамічному розвитку мережі Інтернет та зосередження значної частини інформації в базах даних на серверах. Клієнт-серверну архітектуру можна означити, як концепцію інформаційної мережі в якій основна частина її ресурсів зосереджена в серверах, обслуговуючих своїх клієнтів [3, 7, 28].

Така архітектура визначає такі типи компонентів:

- набір серверів, які надають інформацію або інші послуги програмам, які звертаються до них;
- набір клієнтів, які використовують сервіси, що надаються серверами;
- мережа, яка забезпечує взаємодію між клієнтами та серверами.

Правила взаємодії між клієнтом і сервером називаються протоколом обміну (протоколом взаємодії). Модель клієнт-серверної взаємодії визначається перш за все розподілом обов’язків між клієнтом та сервером.

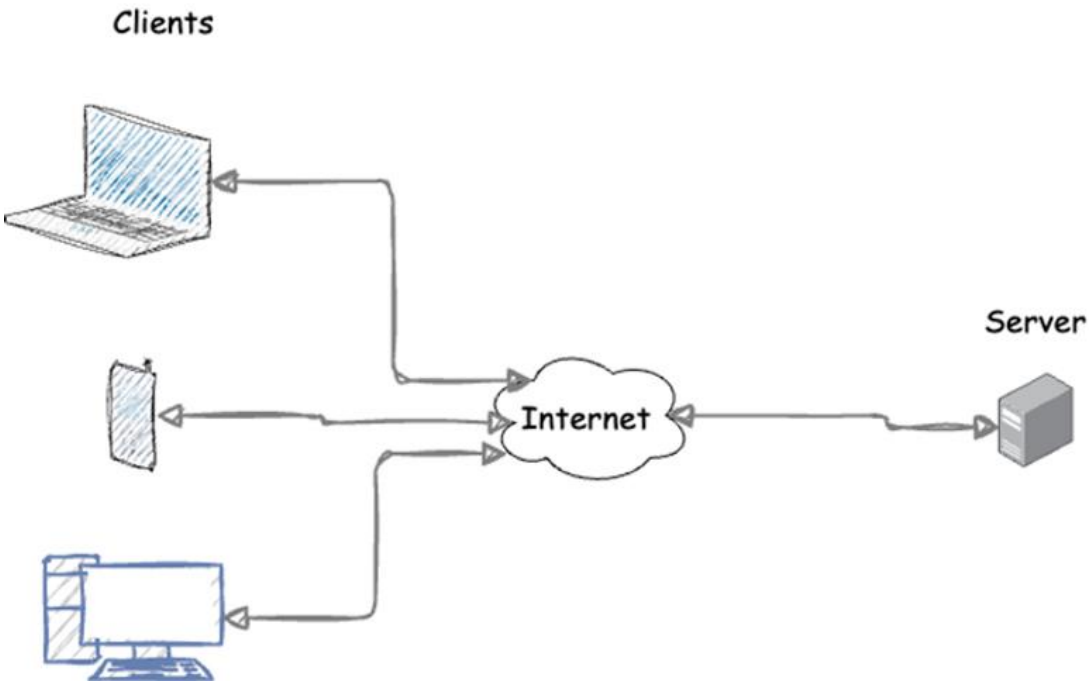

Рисунок 2.2 – Клієнт-серверна архітектура

Логічно можна відокремити три рівні операцій:

- рівень представлення даних, який по суті являє собою інтерфейс користувача і відповідає за представлення даних користувачеві і введення від нього керуючих команд;
- прикладний рівень, який реалізує основну логіку застосунку і на якому здійснюється необхідна обробка інформації;
- рівень управління даними, який забезпечує зберігання даних та доступ до них.

Дволанкова клієнт-серверна архітектура передбачає взаємодію двох програмних модулів — клієнтського та серверного. В залежності від того, як між ними розподіляються наведені вище функції, розрізняють:

- модель **тонкого клієнта**, в рамках якої вся логіка застосунку та управління даними зосереджена на сервері. Клієнтська програма забезпечує тільки функції рівня представлення;
- модель **товстого клієнта**, в якій сервер тільки керує даними, а обробка інформації та інтерфейс користувача зосереджені на стороні клієнта. Товстими клієнтами часто також називають пристрої з обмеженою потужністю:

кишенькові комп’ютери, мобільні телефони та ін.

Трьохланкова клієнт-серверна архітектура, яка почала розвиватися з середини 90-х років, передбачає відділення прикладного рівня від управління даними. Відокремлюється окремий програмний рівень, на якому зосереджується прикладна логіка застосунку. Програми проміжного рівня можуть функціонувати під управлінням спеціальних серверів застосунків, але запуск таких програм може здійснюватися і під управлінням звичайного веб-сервера. Нарешті, управління даними здійснюється сервером даних.

Дво-ланкова архітектура простіша, так як всі запити обслуговуються одним сервером, але саме через це вона менш надійна і висуває підвищені вимоги до продуктивності сервера.

Три-ланкового архітектура складніша, але завдяки тому, що функції розподілені між серверами другого і третього рівня, ця архітектура проявляє [3, 7, 28]:

- високий ступінь гнучкості і масштабованості.
- високу безпеку (тому що захист можна визначити для кожного сервісу або рівня).
- високу продуктивність (тому що завдання розподілені між серверами).

Прикладом клієнт-серверної взаємодії є сервіс WWW. Існує величезна кількість веб-серверів, на яких розміщується та чи інша інформація. У найпростішому випадку ця інформація являє собою набір веб-сторінок, які можуть зберігатися на сервері у вигляді файлів, розмічених за допомогою мови розмітки HTML. Але ситуація, як правило, є складнішою; значна частина веб-ресурсів на сучасному етапі є динамічними, тобто вони не існують в заздалегідь підготовленому вигляді, а створюються безпосередньо в процесі обробки запиту від користувача.

Ролі серверів — це функція сервера (наприклад поштовий, контролер домена тощо). Один сервер може відігравати як одну так і декілька ролей одночасно.

Взалежності від ролі, сервісу який надається, розрізняють такі сервери:

1) Веб-сервер (*Web Server*) – сервер, що приймає HTTP-запити від клієнтів, зазвичай веб-браузерів, видає їм HTTP-відповіді, зазвичай разом з HTML-сторінкою, зображенням, файлом, медіа-потоком або іншими даними. Веб-сервер — основа Всесвітньої павутини. Веб-сервером називають як програмне забезпечення, що виконує функції веб-сервера, так і комп'ютер, на якому це програмне забезпечення працює. Клієнти дістаються веб-сервера за URL-адресою потрібної їм веб-сторінки або іншого ресурсу.

2) Сервер застосунків (*Application Server*) – сервер, що виконує деякі прикладні програми. Термін також відноситься і до програмного забезпечення, що встановлено на такому сервері і забезпечує виконання прикладного ПЗ.

3) Сервери баз данних – використовуються для обробки запитів користувача на мові SQL. При цьому СУБД знаходиться на сервері, до якого підключаються клієнтські додатки.

4) Файловий сервер (*File Server*) – зберігає інформацію у вигляді файлів і представляє користувачам доступ до неї. Як правило файл-сервер забезпечує і певний рівень захисту від несакціонірованного доступу.

5) Сервер друку (*Print Server*) – використовується для надання та управління доступом до принтерів. Роль сервера друку дозволяє управляти принтерами через веб-оглядач, друкувати через URL принтера, використовуючи протокол IPP, а також підключати принтери, використовуючи Point and Print.

6) Поштовий сервер (*Mail Server*) – дозволяє обслуговувати базові поштові скриньки ваших користувачів і дозволяє приймати і відправляти пошту з сервера. Вхідна пошта може зберігатися на сервері, а потім забиратися користувачем по протоколу POP3. Для ролі поштового сервера ви повинні мати: Активне з'єднання з інтернет, Зареєстроване доменне ім'я, запис у провайдера, поштового домену тощо.

7) Термінальний сервер (*Terminal Server*) – надає клієнтам обчислювальні ресурси (процесорний час, пам'ять, дисковий простір) для вирішення завдань. Технічно термінальний сервер — надпотужний комп'ютер (або кластер), підключений до мережі з термінальними клієнтами — у котрих є, як правило,

малопотужні або застарілі робочі станції або спецрішення для доступу до термінального сервера. Термінальний сервер служить для віддаленого обслуговування користувача з наданням робочого столу.

8) Remote Access/VPN Server – сервери віддаленого доступу і VPN надають точку входу в вашу мережу для віддалених користувачів. Використовуючи роль Remote Access / VPN Server, ви можете реалізувати протоколи маршрутизації для середовищ LAN і WAN. Ця роль підтримує модемні з'єднання і VPN через інтернет.

9) DNS Server – дозволяє перетворювати доменні імена (FQDN) в адреси IP.

10) DHCP Server – дозволяє клієнтам отримувати свій IP за потребою. Сервер DHCP також надає додаткову інформацію для конфігурації мережі — адреса серверів DNS, WINS і т.п.

11) Streaming Media Server – використовуються для управління і доставки мультимедійного контенту — потокового відео та аудіо — через інтранет, або інтернет.

12) Ігровий сервер (G*ame server*) – сервер, що забезпечує зв'язок між різними клієнтами, надаючи їм можливість комунікації один з одним в рамках програмної оболонки конкретної гри.

2.3. Модульна архітектура веб-сервера із використанням NestJS

Як уже зазначалося, існують різні підходи до архітектури веб-серверів, зокрема монолітна, модульна, та мікросервісна. Обрано модульну архітектуру із використанням *NestJS*, де кожен аспект додатку розділений на модулі, такі як аутентифікація, управління продуктами, замовленнями та інші.

Переваги цієї архітектури такі [1, 14, 24]:

- Низька зв'язаність: кожен модуль зосереджений на конкретній функціональності, зменшуючи залежності між різними частинами системи.
- Легкість підтримки та розширення: модулі можуть бути легко оновлені

та розширені незалежно один від одного.

- Чітка організація коду: модульна структура забезпечує чистоту та організованість коду, що сприяє кращому розумінню та ефективній роботі з додатком.
- Спрощення тестування: кожен модуль може бути протестований окремо, що забезпечує високу якість тестування.

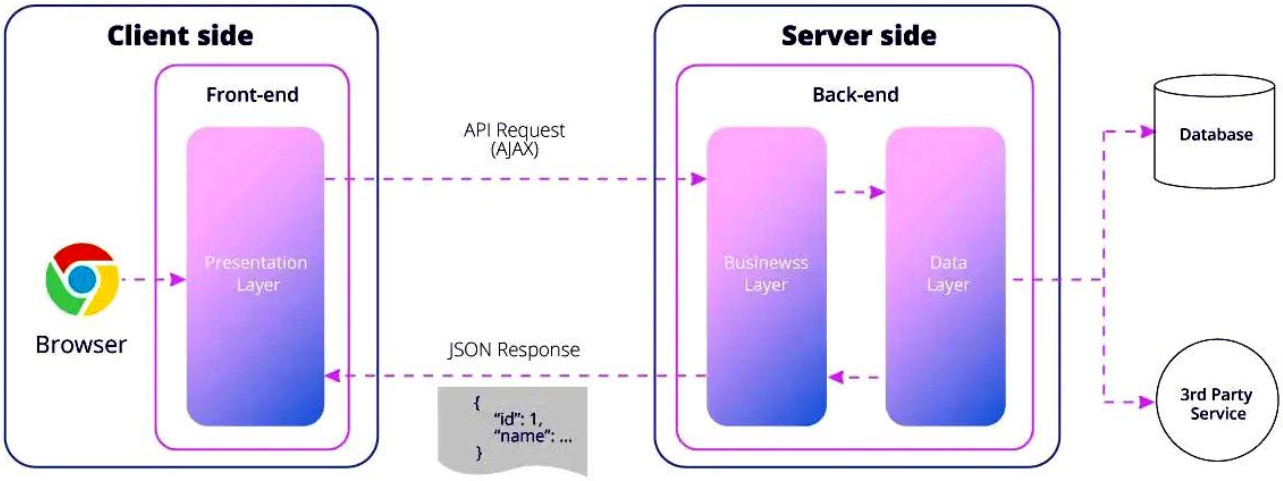

Рисунок 2.3 – Зв'язок між клієнтською і серверною частинами веб-сервісу

Отже, сервер – це фізичне центральне сховище, де зберігаються інформація та комп’ютерні програми. Веб-сервер – це програмне забезпечення, призначене для запуску необхідного обладнання, яке може задовольняти запити клієнтів і доставляти їм вебсторінки.

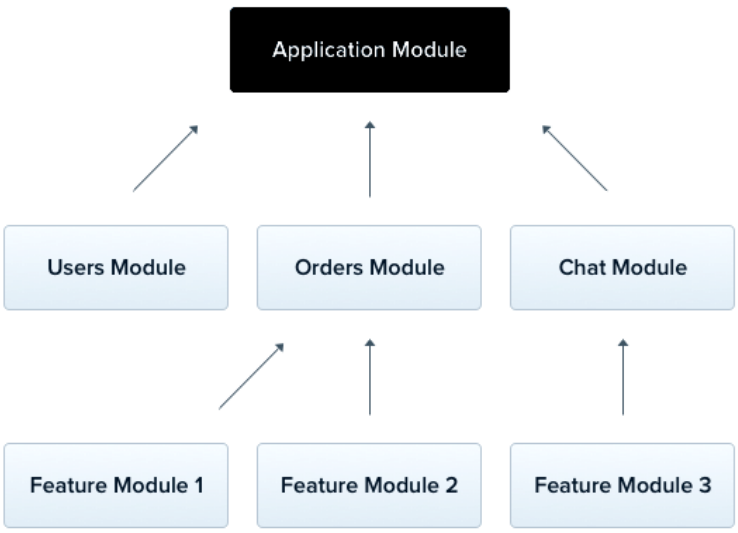

Рисунок. 2.4 – Структура модулів веб-сервісу [3, 28]

Клієнт – будь-який пристрій, який може отримати доступ до інтернету, наприклад, смартфон, розумний годинник, ноутбук або ПК, і запитувати дані з вебсервера, як правило, за допомогою браузера, наприклад, Chrome або Firefox.

2.4. Стратегія захисту даних користувачів

Значну увагу слід приділити безпеці та аутентифікації, оскільки потрібно використовувати різні стратегії Passport.js для забезпечення надійного захисту даних користувачів. Основні стратегії, які застосовано у вирішенні завдань кваліфікаційної роботи щодо розробки архітектури веб-сервера [2]:

JWT (JSON Web Tokens): Використання JWT для створення та верифікації токенів доступу. Ця стратегія забезпечує безпечну передачу інформації про користувача між клієнтом та сервером, дозволяючи ефективно управляти сесіями та доступом до ресурсів.

Google OAuth: Інтеграція з Google OAuth для зручної та швидкої аутентифікації через облікові записи Google. Це не тільки полегшує процес реєстрації та входу для користувачів, але й забезпечує додатковий рівень надійності завдяки надійним механізмам безпеки Google.

AccessToken ma RefreshToken: Використання пари access token та refresh token для підвищення безпеки сесій. Access Token має короткий термін дії і використовується для доступу до API, тоді як Refresh Token з довшим терміном дії дозволяє безпечно оновлювати Access Token.

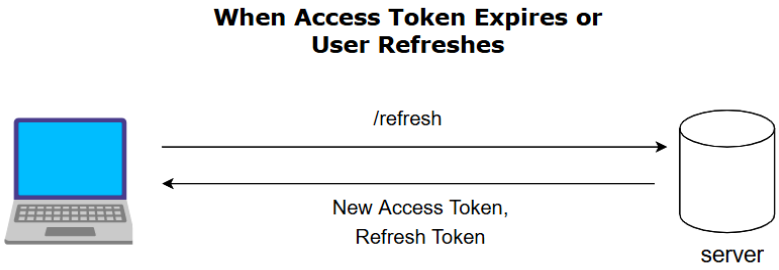

Рисунок. 2.4 – Взаємодія та оновлення точена для клієнта веб-серверу

Ці стратегії разом створюють багатошарову систему аутентифікації, що не

тільки підвищує безпеку додатку, але й забезпечує гнучкість та зручність для користувачів. Вони дозволяють нам ефективно ідентифікувати користувачів, управляти доступом та забезпечувати захист від несанкціонованого доступу та інших потенційних загроз у безпеці.

Реалізація Local Strategy через PassportJS:

Залежно від міри довірчих стосунків, структури, особливостей мережі і віддаленості об'єкта перевірка може бути односторонньою або взаємною. Також розрізняють однофакторну і строгу (двофакторну) автентифікації. В однофакторних системах, найпоширенішими в цей час є парольні системи автентифікації.

У користувача є ідентифікатор і пароль, тобто секретна інформація, відома тільки користувачеві (і можливо – системі), яка використовується для проходження автентифікації. Залежно від реалізації системи, пароль може бути одноразовим або багаторазовим.

РОЗДІЛ 3
МЕТОДИКА ІНТЕГРАЦІЇ ІЗ БАЗОЮ ДАНИХ

3.1. Інтеграція із базою даних комплексних застосунків

У роботі нами використовується PostgreSQL – потужна, відкрита система управління реляційними базами даних, яка славиться своєю надійністю, гнучкістю та підтримкою складних запитів. PostgreSQL є ідеальним вибором для комплексних застосунків завдяки своїй продуктивності та масштабованості.

Для інтеграції з PostgreSQL я використовую Prisma ORM. Prisma полегшує роботу з базою даних, надаючи інтуїтивно зрозумілий API для виконання операцій з даними, таких як створення, читання, оновлення та видалення. Вона дозволяє моделювати структуру даних у вигляді класів та об'єктів JavaScript, що робить код більш читабельним та зрозумілим.

Цей підхід до інтеграції з базою даних не тільки підвищує продуктивність розробки, але й гарантує безпеку та стабільність обробки даних, забезпечуючи високу якість та надійність нашого веб-додатку.

Процес інтеграції з базою даних включає наступні кроки [17, 21, 27]:

- **Визначення схеми даних**: За допомогою Prisma визначаємо схему нашої бази даних у Prisma schema файлі, де описуємо моделі, їх поля та відносини між ними.
- **Міграція даних**: Використовуючи Prisma Migrate, застосовуємо зміни до схеми бази даних, ефективно управляючи міграціями.

- **Взаємодія з даними**: За допомогою Prisma Client виконуємо запити до бази даних прямо з нашого коду на Node.js, що забезпечує гнучкість та ефективність у роботі з даними.

schema.prisma

```
datasource db {
  provider = "postgresql"
  url      = env("DATABASE_URL")
}

model Post {
  id        Int @id @default(autoincrement(
  title     String
  content   String?
  published Boolean @default(false)
  author    User?   @relation(fields: [auth
  authorId  Int?
}

model User {
  id    Int @id @default(autoincrement())
  email String @unique
  name  String?
```

Інформація зазвичай вноситься в базу даних за допомогою **форми**. Кожна форма зазвичай відповідає одному запису або рядку в базі даних. Кожне поле у формі зазвичай відповідає одному стовпчику або комірці в цьому записі. Форма допомагає забезпечити введення правильної інформації. Випадаючі меню, зокрема, допомагають гарантувати, що користувачі не вводять тарабарство.

Ще зовсім недавно тільки професійні програмісти могли зберігати інформацію про сайт в базу даних. Зараз є інструменти, які роблять цю функціональність легкодоступною. Найпростішим прикладом **інтеграції бази даних** є збереження інформації в одну табличну базу даних. Найпростіша таблична база даних – це електронна таблиця.

Інформація витягується з баз даних у вигляді звітів. Найпростіший звіт – це список всіх записів в базі даних. Однак більшість звітів узагальнюють інформацію таким чином, що корисно для бізнес-менеджерів, щоб вони могли приймати рішення на основі даних. Наприклад, звіт може відображати загальний обсяг продажів за продуктами, що дозволяє менеджерам коригувати запаси відповідно до попиту [17, 21, 27].

3.2. Загальна схема взаємодії із типовою базою даних

Типова СУБД ніколи не видима користувачеві. Вона не має графічного інтерфейсу користувача (Graphical User Interface, GUI), а лише видає або записує дані у відповідь на запити відповідною командному мовою. Найбільш поширеною мовою команд є SQL, або Structured Query Language. СУБД повинні бути використані у поєднанні із додатком, що забезпечує інтерфейс користувача. Зазвичай це набір форм з текстовими полями для введення даних і кнопками, або меню для виконання різних дій [17, 21, 27]:.

Системи управління базами даних загального призначення засновані на деякій формальній, загальній моделі організації даних. Найпоширенішим типом бази даних є – реляційна база даних. Всі популярні СУБД є реляційними базами даних, у тому числі Oracle, MySQL, Postgres, FileMaker Pro і Microsoft Access.

Інтерфейс може бути реалізований як додаток для **MS Windows,** встановлений на ПК. Але надзвичайно популярною альтернативою є поєднання СУБД зі створенням вебсторінок на льоту, що забезпечує інтернет-доступ до бази даних.

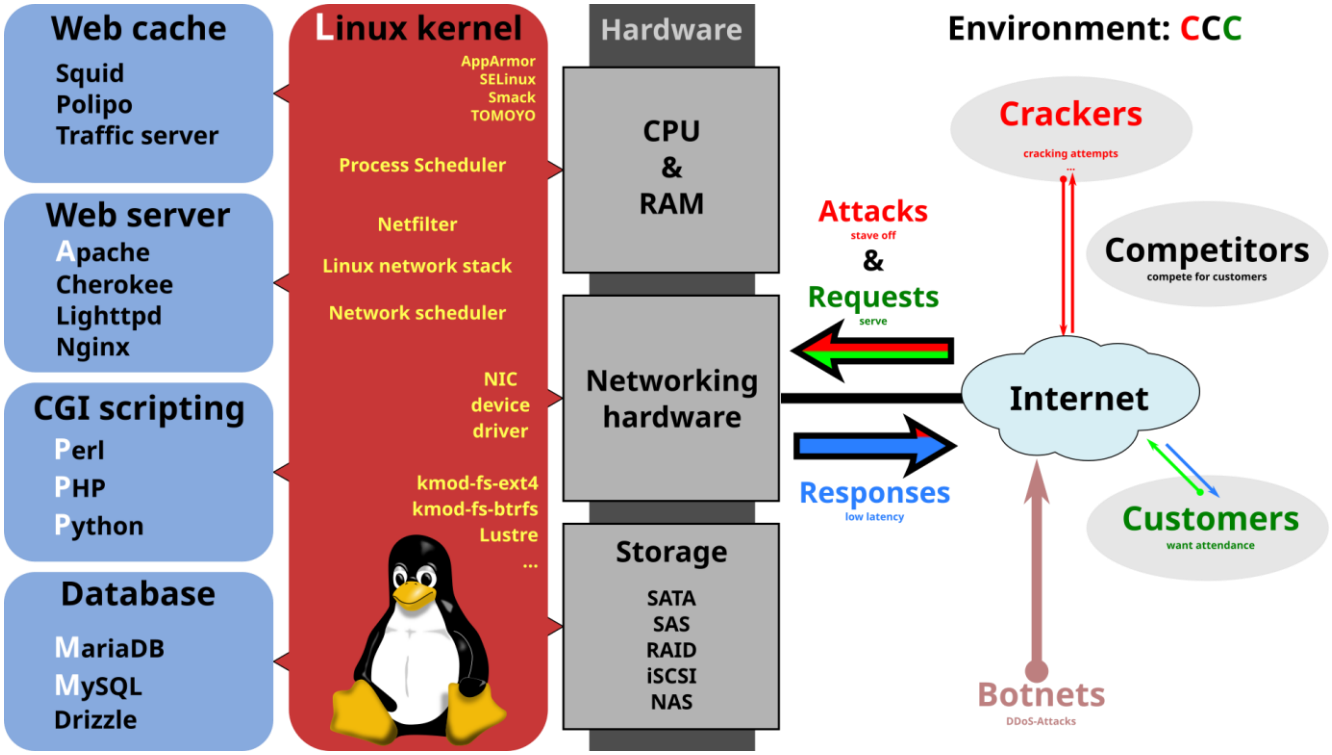

Рисунок. 3.1. – Технологія LAMP

Кінцеві користувачі використовують веб-браузер для перегляду сторінок, що генеруються віддаленим веб-сервером.

Веб-сервер і СУБД можуть розміщуватися на одному сервері або на різних комп'ютерах – це не має ніякого значення, так як веб-сервер має доступ до даних тільки через СУБД, а користувач має доступ до даних тільки через веб-сервер.
На рис. 3.1 показана типова схема технології LAMP (операційна система Linux + веб-сервер Apache + СУБД MySQL + мова програмування PHP).

Схема взаємодії із базою даних у розробленій архітектурі веб-сервера із застосуванням Node.js та NestJS Framework наведена на рис. 3.2. [2].

Таким чином, *найпростіший тип моделі даних* має одну таблицю, або файл. Кожен рядок відповідає деякому об'єкту, а кожен стовпець представляє властивість (атрибут), наприклад найменування, область використання, або переважний порядок слів.

Реляційна база даних складається з декількох таблиць (відносин) такого роду, пов'язаних один з одним певним чином.

Ієрархічна база даних організована не у вигляді таблиці, а у вигляді деревовидної структури, аналогічно структурі вкладених папок на диску комп'ютера. Кожен блок даних належить більш великому блоку і містить більш дрібні одиниці. Подібна організація прийнята в книгах, розділених на глави, розділи, потім підрозділи і т.д.

У об'єктно-орієнтованої БД дані моделюються як об'єкти різних типів. Об'єкти успадковують властивості залежно від їх типу, наприклад у БД про класи слів об'єкти класу "перехідний дієслово" успадковують властивості класу "дієслово".

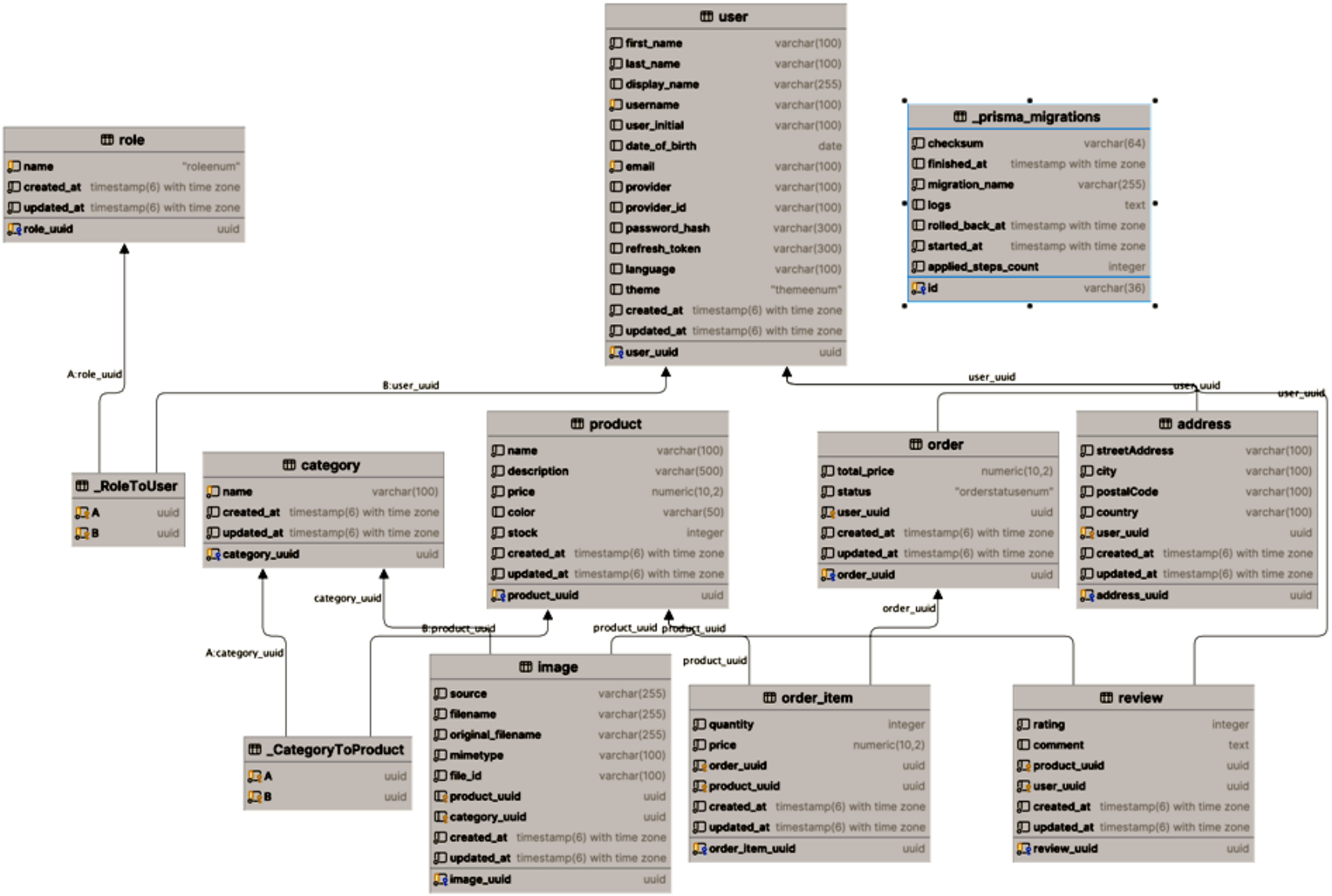

Рисунок 3.2 – Схема взаємодії із базою даних

3.3. Документація та API

Для забезпечення зручності розробників та користувачів API, використовується **Swagger**, який дозволяє автоматично генерувати документацію. Це включає опис усіх ендпоінтів, доступних параметрів запитів, форматів відповідей та можливих статус-кодів. Swagger також надає інтерактивний інтерфейс для тестування API.

Swagger – це технологія, яка дозволяє документувати REST-сервіси. Swagger підтримує безліч мов програмування і фреймворків. Також Swagger надає UI для перегляду документації, зрозумілому для користувача і комп'ютера.

Але Swagger – це не просто специфікація. Основна його перевага полягає в додаткових інструментах. Для нього існує величезна кількість безкоштовних утилітів, які можуть зробити життя QA трохи простішим. Ви можете встановити все це на свої власні сервера і подивитися, як це працює.

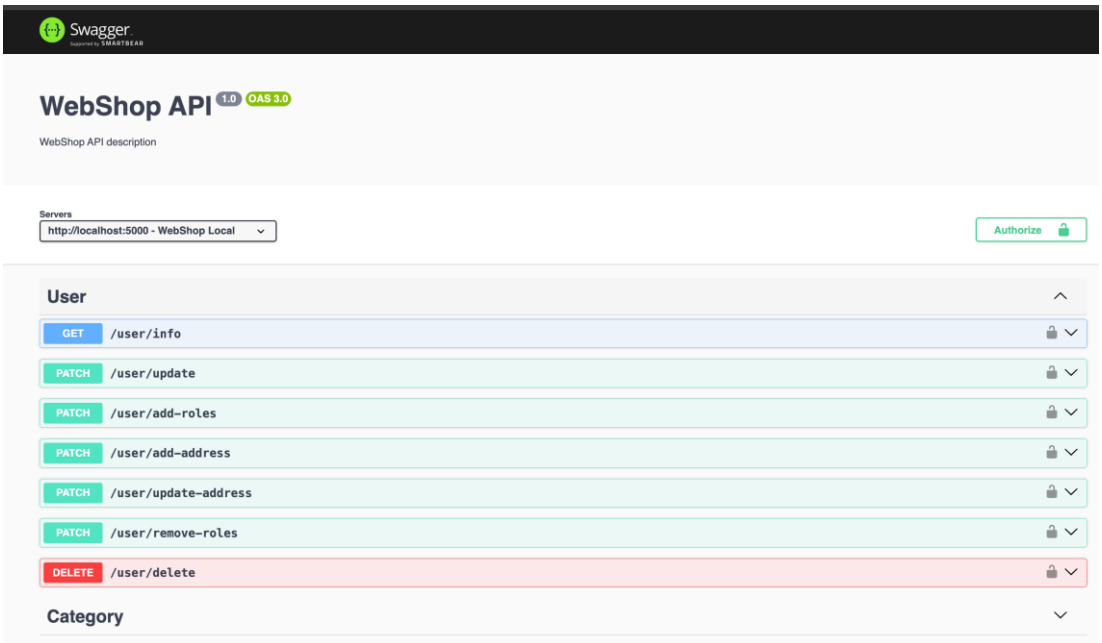

Рисунок 3.3 – Swagger із описом Web Shop SPI

Принцип роботи наступний: у кожного сервісу в певній папці лежить файл зі Swagger описом і зберігається це все прямо в git-репозиторії. Описи можуть бути як згенеровані за допомогою Swagger generator, так і записані туди вручну. Їх

легко розпарсити, і в процесі побудови системи можна автоматично перевіряти відповідність REST endpoints і документації тощо.

Завдання Swagger полягає у збиранні файлів з усіх мікросервісів, вносити деякі мінімальні модифікації (дедублікація, видалення непотрібних атрибутів) і на виході генерує єдиний Swagger файл, який містить повну актуальну інформацію для всієї розробки.

Централізоване зберігання і редагування документації – це тільки перший крок. Наступний – зробити її доступною для всіх розробників, тестерів та інших зацікавлених осіб. І за допомогою невеликої JavaScript бібліотеки Swagger UI генерує HTML елементи для всіх REST endpoints, які надалі можна впорядковувати за допомогою HTML розмітки.

РОЗДІЛ 4.
ПРОЕКТУВАННЯ ТА ПРОГРАМУВАННЯ ЕЛЕМЕНТІВ ВЕБ-СЕРВЕРА

4.1. Результати розробки бази даних та програмна реалізація веб-сервера

Реалізація бази даних починається з визначення полів для всіх сутностей системи. Сформовано 12 таблиць: таблиця **user**, яка зберігає інформацію про користувача, таблиця **product**, яка зберігає інформацію про продукти, таблиця **category**, яка зберігає інформацію про категорії продуктів, таблиця **role**, яка зберігає інформацію про ролі користувачів, таблиця **address**, яка зберігає інформацію про адреси користувачів, таблиця **order**, яка зберігає інформацію про замовлення користувачів, таблиця **image**, яка зберігає інформацію про зображення для продуктів та категорій, таблиця **review**, яка зберігає інформацію про відгуки на продукти, таблиця **order_item**, яка зберігає інформацію про продукти в замовленні та їхню кількість [17, 21, 27].

Окрім того, сформовано декілька системних таблиць які потрібні для визначення звязків та міграцій для Prisma ORM такі як: _prisma_migrations, _RoleToUser, _CategoryToProduct.

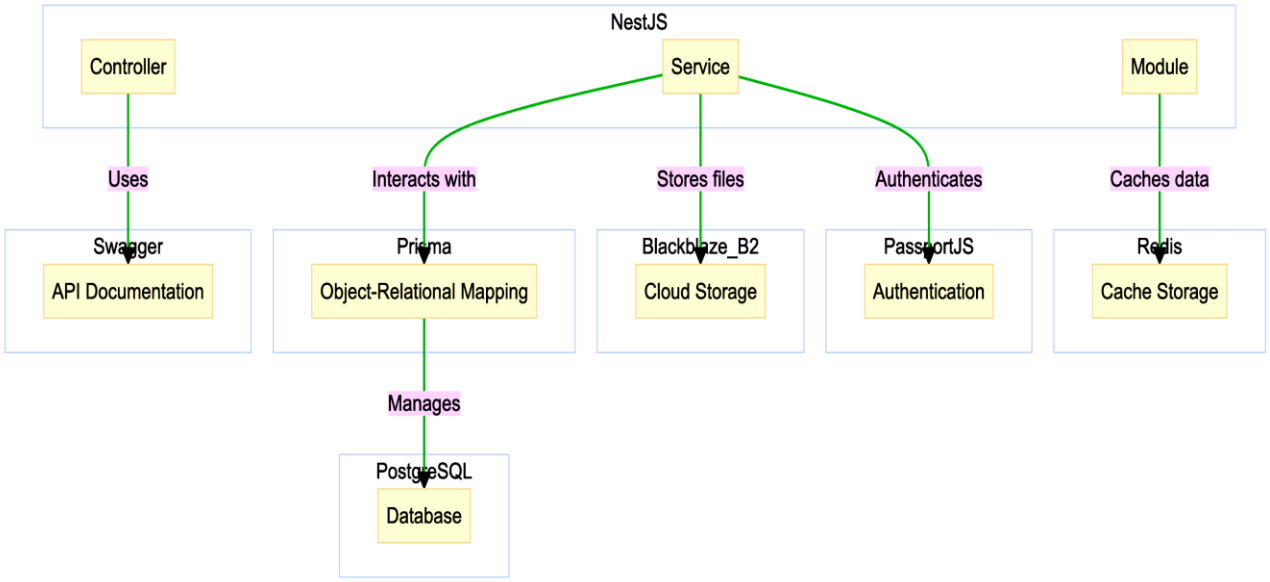

Рисунок 4.1 – Діаграми програмної реалізації веб-сервера

У вигляді помічника для роботи із базою даних, використано засіб Prisma ORM. Він має багато функцій для роботи із базою даних, котрі перекладаються на мову **SQL**, що суттєво спрощує розробку веб-сервера.

Приклад роботи з базою даних за допомогою «**Prisma**»:

```
import { Injectable } from '@nestjs/common';
import { Prisma } from '@prisma/client';
import { BaseRepository } from '../../../shared/repositories/base.repository';
import { ErrorsHandleService } from '../../../shared/services/error.service';
import { PrismaService } from '../../../shared/services/prisma.service';
import { UserWithRelations } from '../entities/user.entity';

@Injectable()
export class UserRepository extends BaseRepository<
  UserWithRelations,
  Prisma.UserFindUniqueArgs,
  Prisma.UserFindFirstArgs,
  Prisma.UserFindManyArgs,
  Prisma.UserCreateArgs,
  Prisma.UserCreateManyArgs,
  Prisma.UserUpdateArgs,
  Prisma.UserUpdateManyArgs,
  Prisma.UserUpsertArgs,
  Prisma.UserDeleteArgs,
  Prisma.UserDeleteManyArgs,
  Prisma.UserCountArgs
> {
  constructor(prismaService: PrismaService, errorsHandleService: ErrorsHandleService) {
    super('User', prismaService, errorsHandleService);
  }
  async getUserByAnyCaseEmail(email: string): Promise<UserWithRelations> {
    return this.findOneWithoutChecking({
      where: {
        email: {
          equals: email,
          mode: 'insensitive',
        },
      },
      include: {
        roles: true,
        address: true,
        orders: true,
        reviews: true,
      },
    });
  }}
```

schema.prisma | base.repository.ts | user.repository.ts

src > shared > repositories > base.repository.ts > BaseRepository > findOneWithoutChecking

```
You, 2 weeks ago | 1 author (You)
/*
https://www.prisma.io/docs/concepts/components/prisma-client/crud
https://www.prisma.io/docs/concepts/components/prisma-client/relation-queries
*/

import { NotFoundException } from '@nestjs/common';
import { Prisma } from '@prisma/client';
import { NOT_EXIST_MSG, NOT_EXIST_TYPE } from '../constants/server.constants';
import { ErrorsHandleService } from '../services/error.service';
import { PrismaService } from '../services/prisma.service';
import { ErrorType } from '../types';

You, 2 weeks ago | 1 author (You)
export abstract class BaseRepository<MODEL, FU, FF, FM, C, CM, U, UM, UPS, D, DM, CC> {
  private readonly model: string;
  private readonly ORM: PrismaService;
  private readonly errorsHandleService: ErrorsHandleService;

  protected constructor(model: string, database: PrismaService, errorsHandleService: ErrorsHandleService) {
    this.model = model;
    this.ORM = database;
    this.errorsHandleService = errorsHandleService;
  }

  async queryRaw(data: string, optionalErrorMessage?: string): Promise<MODEL[]> {…
  }

  async findOneAndThrowIfNotExist(data: FF, optionalErrorMessage?: string): Promise<NonNullable<MODEL>> {…
  }

  async findOneWithoutChecking(data: FF, optionalErrorMessage?: string): Promise<MODEL> {
    try {
      // @ts-ignore
      return await this.ORM[this.model].findFirst(data);
    } catch (e) {
      const err = e as ErrorType;
      throw this.errorsHandleService.throwErrorIfServerError(
        err,
        'findOneWithoutChecking',
        this.model,
        optionalErrorMessage
      );
    }
  }      You, 2 weeks ago • first commit

```

Функція getUserByAnyCaseEmail приймає email як обов’язковий аргумент та повертає користувача з таким мейлом в будь-якому регістрі якщо той присутній у базі даних.

Таким чином сервер працює з базою даних без взаємодії з неї напряму, що робить такий зв’язок більш абстрактним та дозволяє з легкістю змінити СУБД без потреби адаптувати код під нову систему.

4.2. Проектування та програмування системи

На рисунку 4.2 зображено діаграму прецендентів веб-сервера. Діаграма

містить два актора: Користувач та менеджер.

Користувач має обмежений функціонал, що складається з можливості увійти до системи, переглянути продукти, категорії, зробити покупку та вийти з системи.

Менеджер чи має той самий функціонал що й користувач окрім того що може створити новий товар чи категорію та редагувати вже існуючі товари чи категорії.

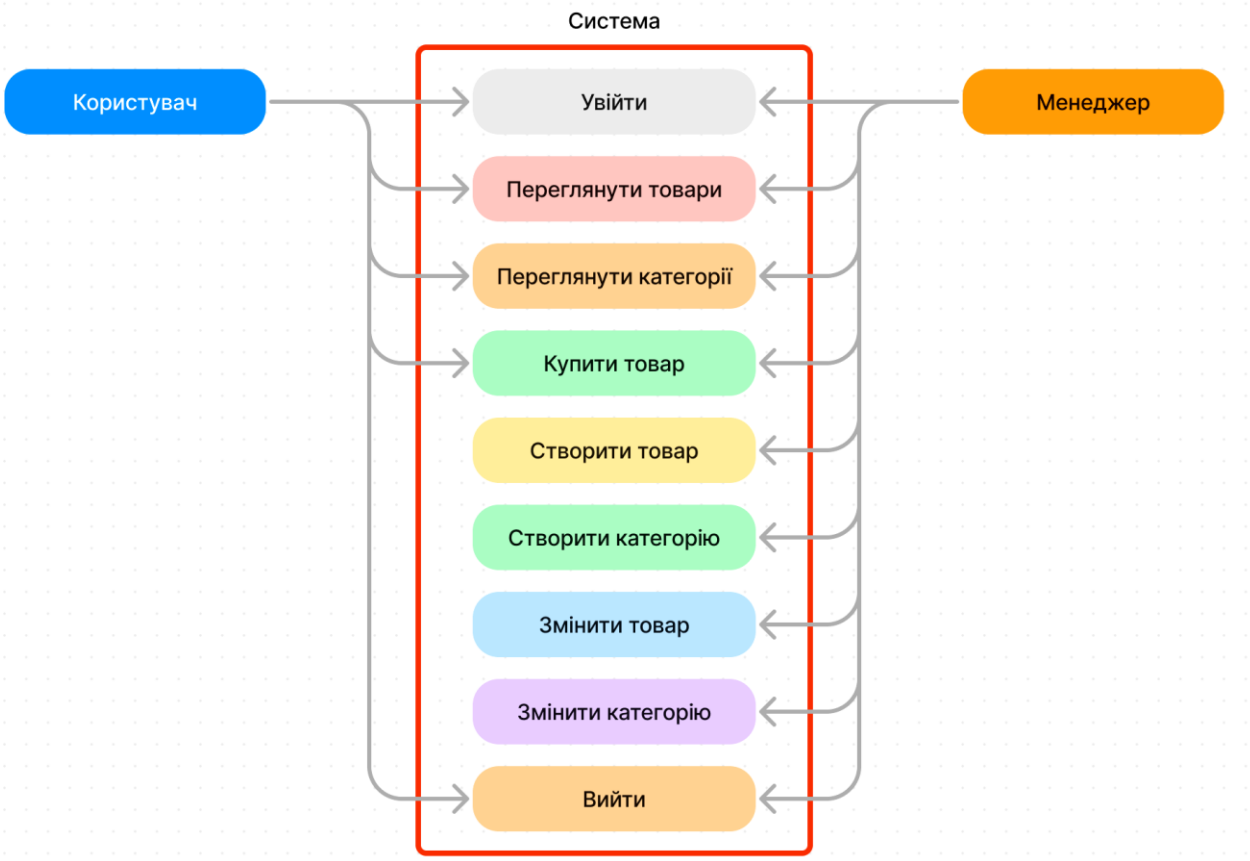

Рисунок 4.2 – Діаграма прецендентів "користувачі-система-менеджер"

Сервер складається з шести модулів:

- **Auth**: відповідає за авторизацію та перевірку ролі користувача;
- **Category**: відповідає за роботу з категоріями (їх створення, редагування та видалення);
- **Image**: відповідає за роботу з зображеннями (їх створення, завантаження у сховище та видалення);
- **Order**: відповідає за роботу з замовленнями (їх створення, додавання

нових товарів, та редагування);

- **Product**: відповідає за роботу з продуктами (їх створення, редагування та видалення);
- **User**: відповідає за роботу з користувачами (їх створення, редагування та видалення);

Також проект має директорію prisma, яка використовується СУБД для збереження даних про схему бази даних та міграції.

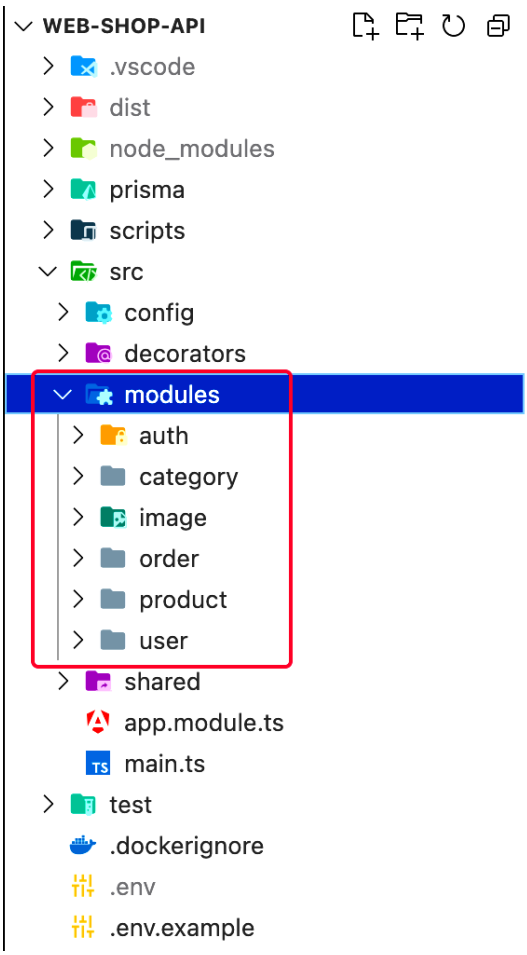

Рисунок 4.3 – Каталог структурних елементів проекту

Файл .env відповідає за збереження глобальних змінних проекту та має вигляд текстового файлу.

```
# ENV
NODE_ENV=development | production
SERVER_PORT=5000

# PostgreSQL DATABASE
POSTGRES_PORT=5432
POSTGRES_USER=user
```

```
POSTGRES_PASSWORD=password
POSTGRES_DB=web_shop
DATABASE_URL="postgresql://user:password@host:port/web_shop?schema=public"
ENABLE_DB_LOGS=1 # 0 - false | 1 - true

# Redis Cache
REDIS_HOST=localhost
REDIS_PORT=6379
REDIS_PASSWORD=password
REDIS_TTL=1800 # cache ttl in seconds

# Swagger API Docs
SWAGGER_URL_PATH=docs

# Pino Logging
PINO_LOG_LEVEL=debug # trace | debug | info | warn | error | fatal | silent
PINO_HIDE_OBJECTS=0 # 0 - false | 1 - true
PINO_MESSAGE_KEY=msg
PINO_IGNORE=hostname

# JWT SECRET
JWT_ACCESS_SECRET=MY_TEST_SECRET_SUPER_STRONG
JWT_REFRESH_SECRET=MY_TEST_SECRET_SUPER_STRONG

# B2 Storage
B2_BUCKET_ID=
B2_APPLICATION_KEY_ID=
B2_APPLICATION_KEY=
B2_DOMAIN=

# GOOGLE AUTH
GOOGLE_CLIENT_ID=
GOOGLE_CLIENT_SECRET=
GOOGLE_CALLBACK_URL=
```

Структура написання .env файлів – це КЛЮЧ=ЗНАЧЕННЯ та групування їх з нового рядка. Фактично, .env файли реалізовують використання резидентної бази заних із Redis.

.gitignore це файл який спрощує роботу з системою контролю версій GIT та має перелік назв файлів які потрібно проігнорувати та не додавати до репозиторію GIT.

Приклад .gitignore в системі:

```
dist // скомпільований код проекту
node_modules // модулі npm
logs // логі проекту
.env // глобальні зміни
```

Розглянемо структуру модуля на прикладі модуля user. Модуль має директорію dto (data transfer object), яка зберігає файли опису обєктів, директорію entities, яка зберігає файли опису сутностей, директорію repositories, яка зберігає файли репозиторіїв які розширюють основний репозиторій, файл модуля (user.module.ts) де ми підключаємо всі необхідні залежності, файл сервісу (user.service.ts) де знаходиться бізнес логіка, файл контролера (user.controller.ts) де ми описуємо всі точки доступу через HTTP запити.

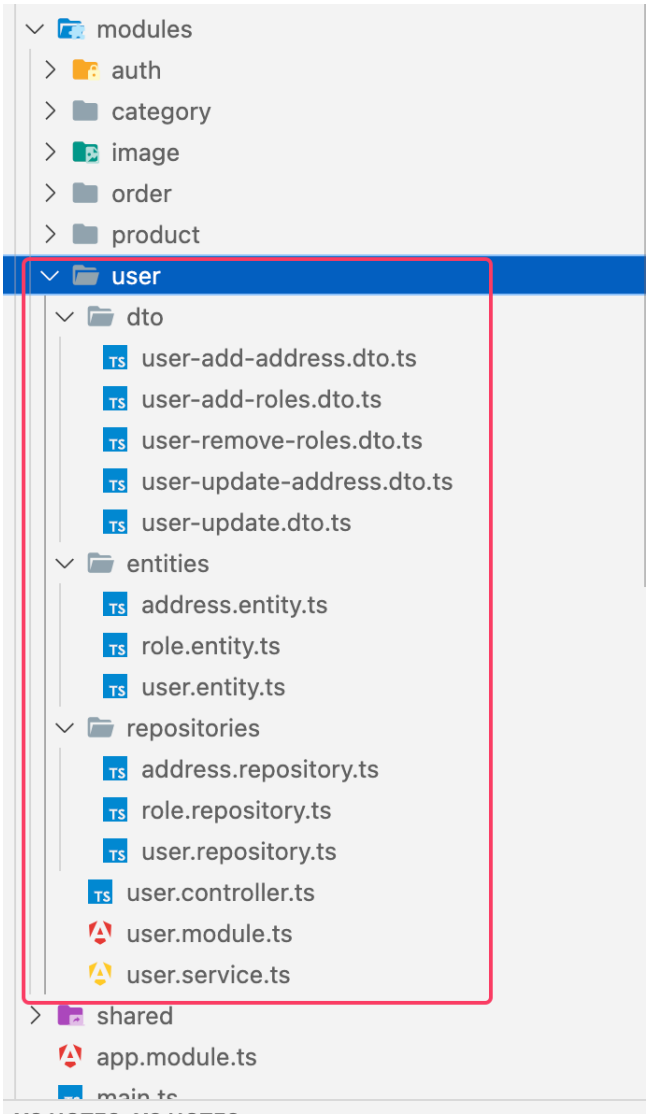

Рисунок 4.5 – Загальний вигляд структури модуля user

Main.ts це файл з якого починається робота додатку, наведений в Дод. А.

У першу чергу імпортуються класи фреймворка далі конфігурується та запускається функція bootstrap. В ній конфігурується додаток та виконується

валідація, логування, конфігурування swagger для опису API а також вказується порт на якому буде запускатись додаток.

Далі розглянемо головний файл-модуль app.module.ts. В нього імпортуються всі інші модулі програми.

```
import { CacheModule } from '@nestjs/cache-manager';
import { Module } from '@nestjs/common';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { APP_INTERCEPTOR } from '@nestjs/core';
import { LoggerModule } from 'nestjs-pino';
import { envConfiguration } from './config/env.config';
import { pinoConfigFactory } from './config/pino.config';
import { AuthModule } from './modules/auth/auth.module';
import { CategoryModule } from './modules/category/category.module';
import { OrderModule } from './modules/order/order.module';
import { ProductModule } from './modules/product/product.module';
import { UserModule } from './modules/user/user.module';
import { CustomHttpCacheInterceptor } from './shared/interceptors/httpCache.interceptor';
import { CacheConfigService } from './shared/services/cache.service';
import { PrismaService } from './shared/services/prisma.service';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      validationSchema: envConfiguration,
    }),
    LoggerModule.forRootAsync({
      imports: [ConfigModule],
      inject: [ConfigService],
      useFactory: pinoConfigFactory,
    }),
    CacheModule.registerAsync({
      useClass: CacheConfigService,
    }),
    UserModule,
    CategoryModule,
    ProductModule,
    OrderModule,
    AuthModule,
  ],
  providers: [
    PrismaService,
    {
      provide: APP_INTERCEPTOR,
      useClass: CustomHttpCacheInterceptor,
    },
  ],
})
export class AppModule {}
```

Створюється пустий клас AppModule який налаштовується директивою @Module. Розглянемо параметри директиви:

- Imports: це перелік імпортованих модулів які використовуються в цьому модулі. В цьому прикладі використовується всі модулі які в нас є, а також ті модулі які ми будемо використовувати для бізнес логіки. ConfigModule використовується для доступу до env змінних. LoggerModule використовується для логування. CacheModule використовується для підключення кешування;
- Providers: це перелік сервісів та інших допоміжних класів модуля. PrismaService це сервіс нашої ORM, також використовується Інтерцептор для кешування запитів;

Наступний файл user.module.ts який має схожу структуру. Він використовує controllers масив для підключення контролера а також має інший масив providers:

- **UserService** – сервіс користувача із бізнес логікою;
- **PrismaService** – сервіс orm для роботи із базою даних;
- **UserRepository** – репозиторій користувача;
- **AddressRepository** – репозиторій адрес;
- **RoleRepository** – репозиторій ролей;
- **ErrorsHandleService** – сервіс для обробки помилок;

```
import { Module } from '@nestjs/common';
import { ErrorsHandleService } from 'src/shared/services/error.service';
import { PrismaService } from 'src/shared/services/prisma.service';
import { AddressRepository } from './repositories/address.repository';
import { RoleRepository } from './repositories/role.repository';
import { UserRepository } from './repositories/user.repository';
import { UserController } from './user.controller';
import { UserService } from './user.service';

@Module({
  controllers: [UserController],
  providers: [UserService, PrismaService, UserRepository, AddressRepository,
RoleRepository, ErrorsHandleService],
})
export class UserModule {}
```

Далі розглянемо файл user.controller.ts , який наведено в Дод. Б.

Створюється клас UserController який позначається директивою @Controller. Клас імпортує сервіс UserService.

Розглянемо функції контролера:

- info: ця функція повертає інформацію про користувача;
- update: ця функція дозволяє оновити данні про користувача;
- addRoles: ця функція дозволяє додати ролі до користувача;
- addAddress: ця функція дозволяє додати адресу користувача;
- updateAddress: ця функція дозволяє оновити адресу користувача;
- removeRoles: ця функція дозволяє видалити ролі користувача;
- delete: ця функція дозволяє видалити користувача.

Файл user.service.ts наведено в Додатку В.

Створюється клас UserService з директивою @Injectable яка помічає цей клас як провайдер з подальшою змогою до імпортування. В конструкторі класа імпортується репозиторії бази даних за допомогою яких ми будемо доступатись до таблиць в базі даних. Таким чином вся бізнес логіка програми скривається від контролера що задає гарний архітектурний тон проекту.

Далі розглянемо структуру dto файлу на прикладі файлу user-add-address.dto.ts який використовується для опису навантаження для запиту на додавання адреси користувача:

```
import { ApiProperty } from '@nestjs/swagger';
import { IsString } from 'class-validator';

export class UserAddAddressDto {
  @ApiProperty({
    required: true,
    type: String,
  })
  @IsString()
  streetAddress: string;

  @ApiProperty({
    required: true,
    type: String,
  })
  @IsString()
  city: string;
```

```
  @ApiProperty({
    required: true,
    type: String,
  })
  @IsString()
  postalCode: string;

  @ApiProperty({
    required: true,
    type: String,
  })
  @IsString()
  country: string;
}
```

Створюється та експортується клас UserAddAddressDto в якому описано всі поля та додано декоратори на їх валідацію, а також декоратори для swagger.

Розглянемо структуру репозиторія на прикладі файлу user.repository.ts

```
import { Injectable } from '@nestjs/common';
import { Prisma } from '@prisma/client';
import { BaseRepository } from '../../../shared/repositories/base.repository';
import { ErrorsHandleService } from '../../../shared/services/error.service';
import { PrismaService } from '../../../shared/services/prisma.service';
import { UserWithRelations } from '../entities/user.entity';

@Injectable()
export class UserRepository extends BaseRepository<
  UserWithRelations,
  Prisma.UserFindUniqueArgs,
  Prisma.UserFindFirstArgs,
  Prisma.UserFindManyArgs,
  Prisma.UserCreateArgs,
  Prisma.UserCreateManyArgs,
  Prisma.UserUpdateArgs,
  Prisma.UserUpdateManyArgs,
  Prisma.UserUpsertArgs,
  Prisma.UserDeleteArgs,
  Prisma.UserDeleteManyArgs,
  Prisma.UserCountArgs
> {
  constructor(prismaService: PrismaService, errorsHandleService: ErrorsHandleService) {
    super('User', prismaService, errorsHandleService);
  }

  async getUserByAnyCaseEmail(email: string): Promise<UserWithRelations> {
    return this.findOneWithoutChecking({
      where: {
        email: {
          equals: email,
```

```
          mode: 'insensitive',
        },
      },
      include: {
        roles: true,
        address: true,
        orders: true,
        reviews: true,
      },
    });
  }
}
```

Ми створюємо клас UserRepository який розширює клас BaseRepository де описані всі загальні методи для всі репозиторіїв і тут ми можемо додати специфічні методи для окремих випадків такі як getUserByAnyCaseEmail.

4.3. Розробка інструкції з використання сервісу

Інструкція для користувача. При вході на сайт з'являться всі присутні точки виклику API. Точки поділяються на модулі, які ми описали.

Почнемо з реєстрації користувача натиснувши на точку POST /auth/register.

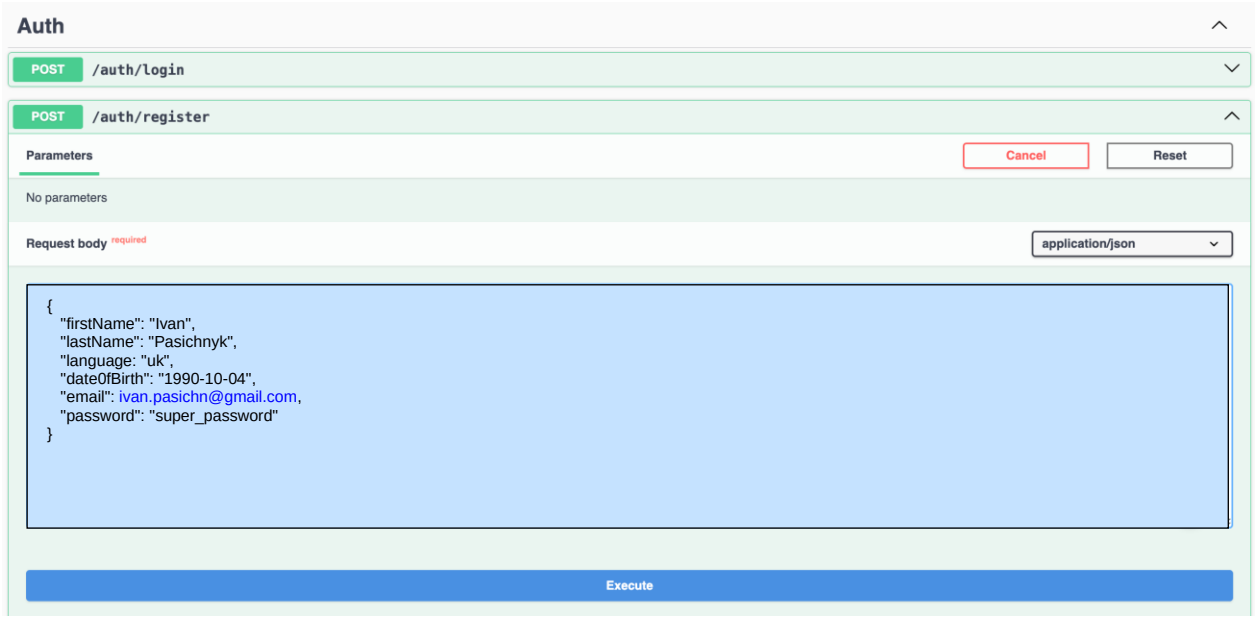

Рисунок 4.6 – Загальний вигляд вікна для реєстрації користувача

Записуємо у форму е-майл адресу і відповідний пароль для користувача та інші дані повязані із користувачем після цього натискаємо на кнопку Execute. Далі

Swagger формує та виконує запит на сервер.

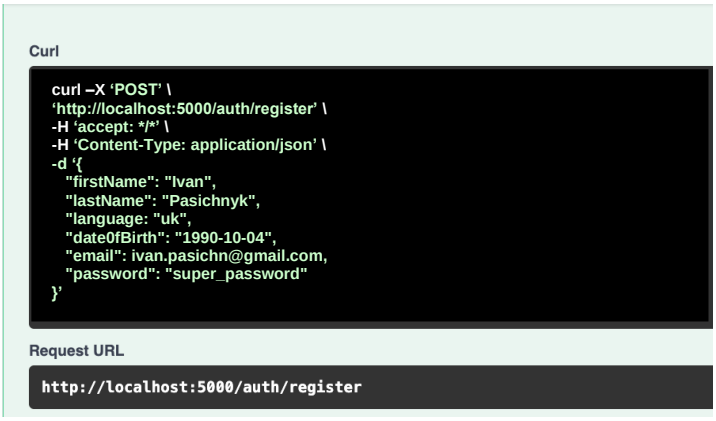

Рисунок 4.7 – Сформований запит

Після запиту отримуємо відповідь в якій маємо два поля з токенами: 1) accessToken використовувати для входу до системи; 2) refreshToken для оновлення основного accessToken токену.

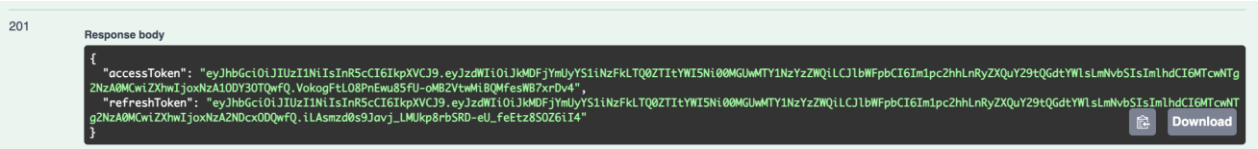

Рисунок 4.8 – Відповідь від сервера

Для того щоб увійти до системи використовуючи Swagger, потрібно використати accessToken який повертає сервер. Для цього натискаємо кнопку Authorize нагорі сайту.

Після натискання на кнопку відкривається модальне вікно яке містить поле для вводу токена. Вводимо скопійований токен у поле на натискаємо кнопку Authorize.

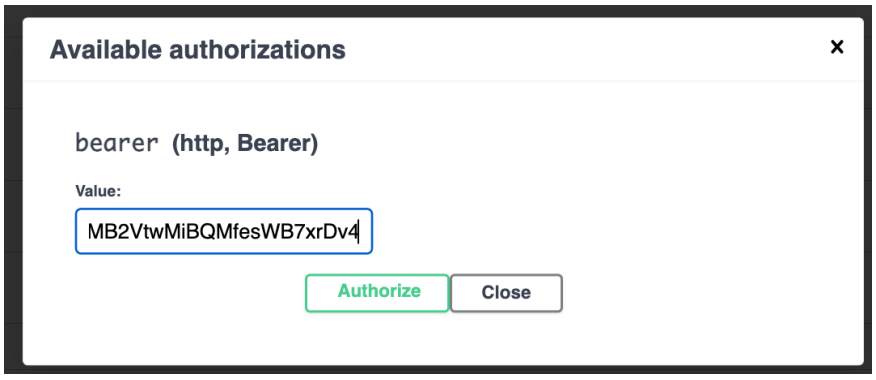

Рисунок 4.9 – Вікно авторизації через токен

Тепер можна використовувати точки, які потребують авторизації (вони позначаються замком). Сформуємо новий Тестовий продукт.

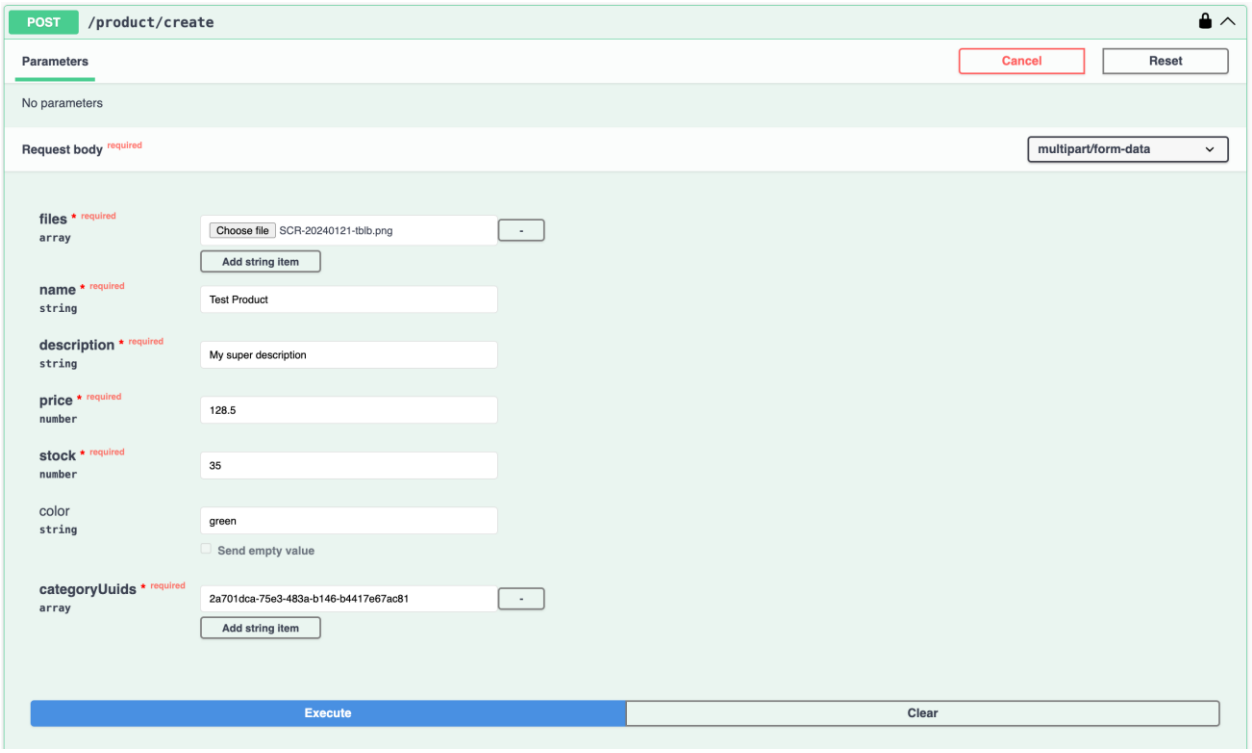

Рисунок 4.10 – Форма створення Тестового продукту

Заповнюємо потрібну нам інформацію. Натискаємо Execute та дивимось на відповідь від сервера.

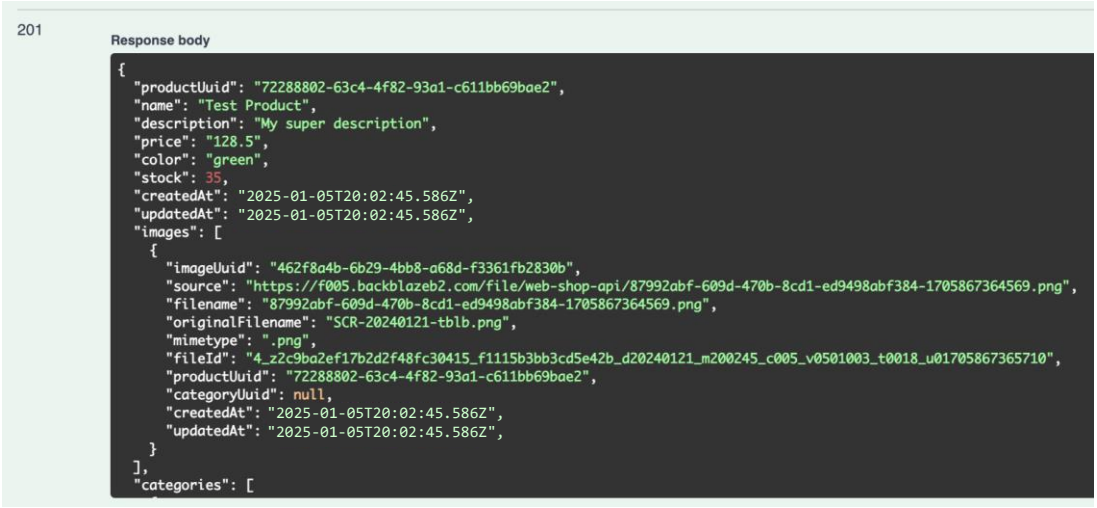

Рисунок 4.11 – Відповідь на створення Тестового продукту

Після цього цей "Тестовий продукт" можна знайти скориставшись запитом на пошук продуктів. Для цього відкриємо форму пошуку продуктів та додамо потрібні нам фільтри за назвою продукту.

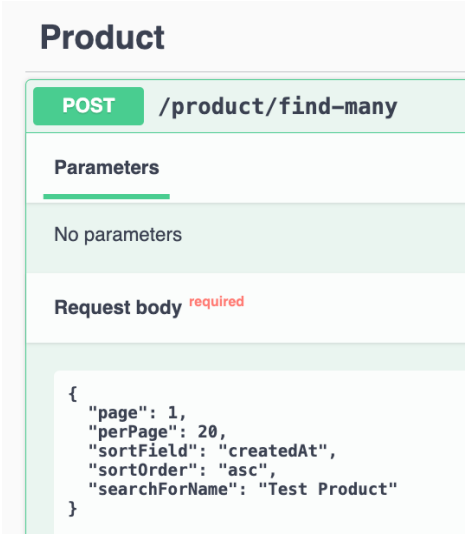

Рисунок 4.12 – Форма пошуку внесених записів про продукти

Натискаємо Execute та перевіряємо відповідь від сервера.

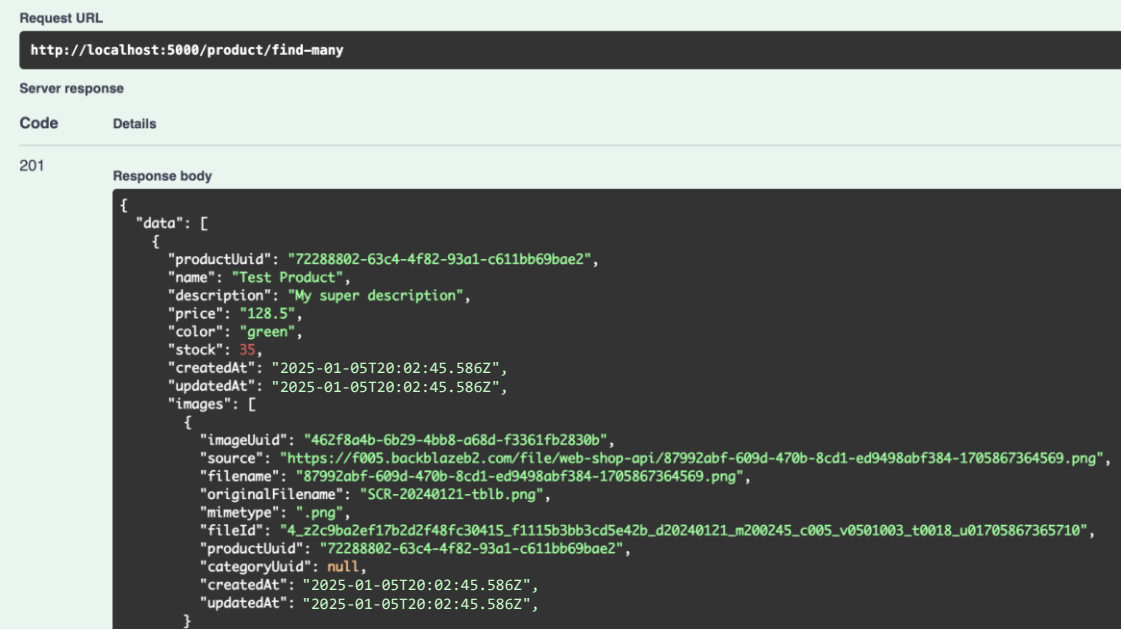

Рисунок 4.13 – Відповідь на пошук за вказаними критеріями

Тепер можна редагувати продукт, для цього відкриваємо форму редагування продукту та вводимо її унікальний ключ product_uuid та данні які хочемо змінити.

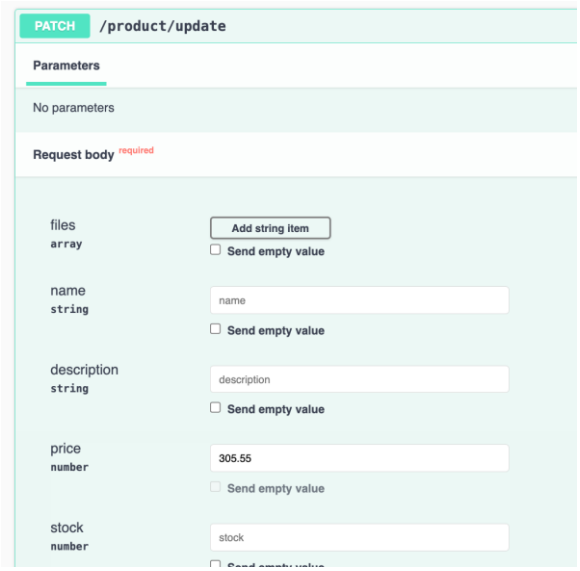

Рисунок 4.14 – Форма редагування показників продукту

Натискаємо Execute та перевіряємо відповідь від сервера.

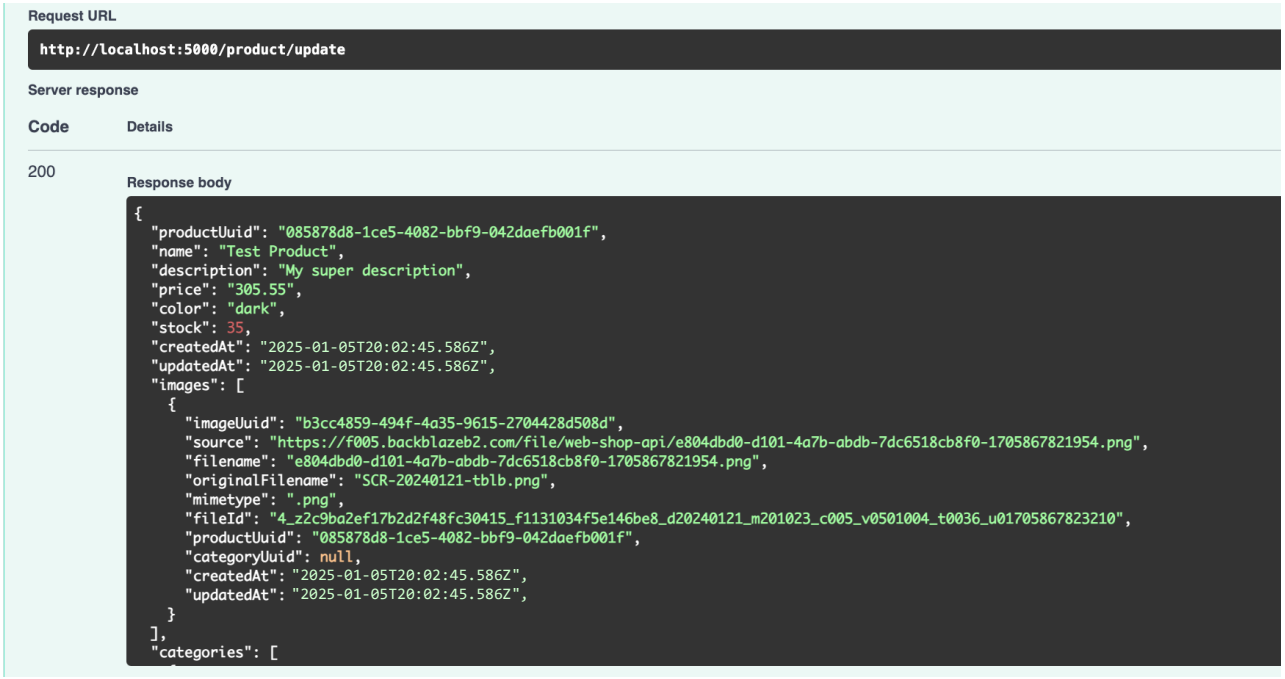

Рисунок 4.15 – Відповідь веб-сервера на редагування продукту

Тепер Тестовий продукт можна видалити, для цього відкриваємо форму видалення продукту та вводимо її унікальний ключ product_uuid.

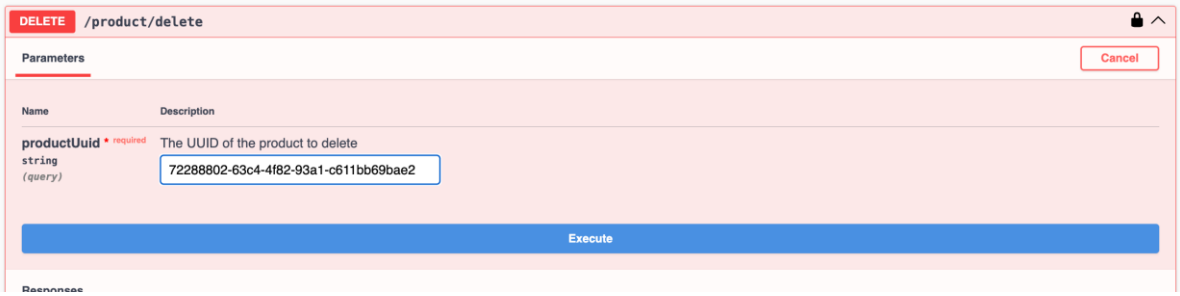

Рисунок 4.16 – Форма видалення продукту

Натискаємо кнопку Execute та перевіряємо відповідь.

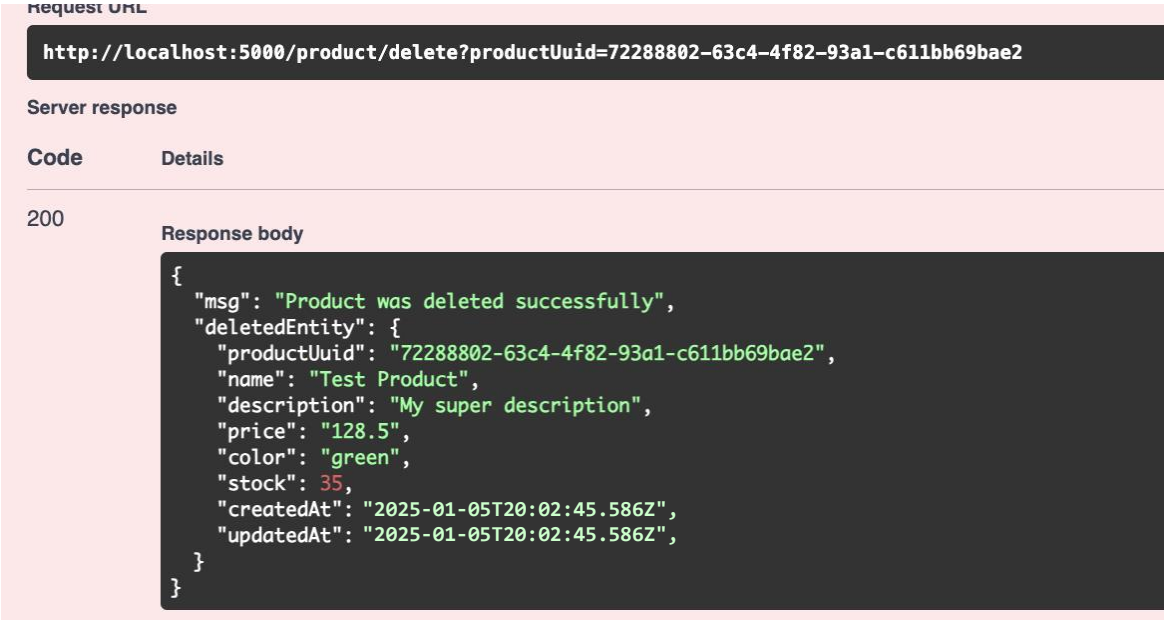

Рисунок 4.17 – Відповідь на видалення продукту

Після цього можна вийти зі системи натиснувши кнопку Authorize.

Рисунок. 3.18 – Модальне вікно авторизованого стану

Натискаємо кнопку Logout для того, щоб стерти токен та закінчити сесію користувача.

РОЗДІЛ 5.
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

5.1. Розробка логіко-імітаційної моделі виникнення травм і аварій

Методикою оцінки рівня небезпеки робочих місць, машин, виробничих процесів та окремих виробництв передбачено пошук об'єктивного критерію рівня небезпеки для конкретного об'єкта [5]. Таким показником вибрана ймовірність виникнення аварії, травми залежно від явища, що досліджується.

Для побудови логіко-імітаційної моделі процесу, формування і виникнення аварії та травми в процесі створення мікрокліматичних умов у приміщенні оцінюють відповідні небезпечні події. Кожній із них присвоїмо ймовірність виникнення:

Шифр	Назва події	Ймовірність
P_1	Відсутність захисного заземлення	0,02
P_2	Пошкодження захисного заземлення	0,04
P_3	Спрацювання складових захисту	0,1
P_4	Неправильна експлуатація захисту	0,02
P_5	Відсутність профілактичних заходів	0,2
P_6	Відсутність захисного щита	0,12
P_7	Недотримання правил вибору взуття	0,15
P_8	Незнання правил техніки безпеки	0,1
P_9	Відсутність засобів індивідуального захисту	0,2
P_{10}	Легковажність	0,08

На основі наведених подій будуємо матрицю логічних взаємозв'язків між окремими пунктами, графічна інтерпретація якої зображено на рис. 5.1.

Розрахуємо ймовірності виникнення подій, що формують логіко-імітаційну модель процесів створення мікрокліматичних умов. Розглянемо травмонебезпечну ситуацію, що виникає за умови роботи працівників із електронебезпекою.

Підставивши дані ймовірностей базових подій у формулу, отримаємо ймовірність події 13: P_{13} = 0,2+ 0,4 – 0,2·0,4 = 0,0592.

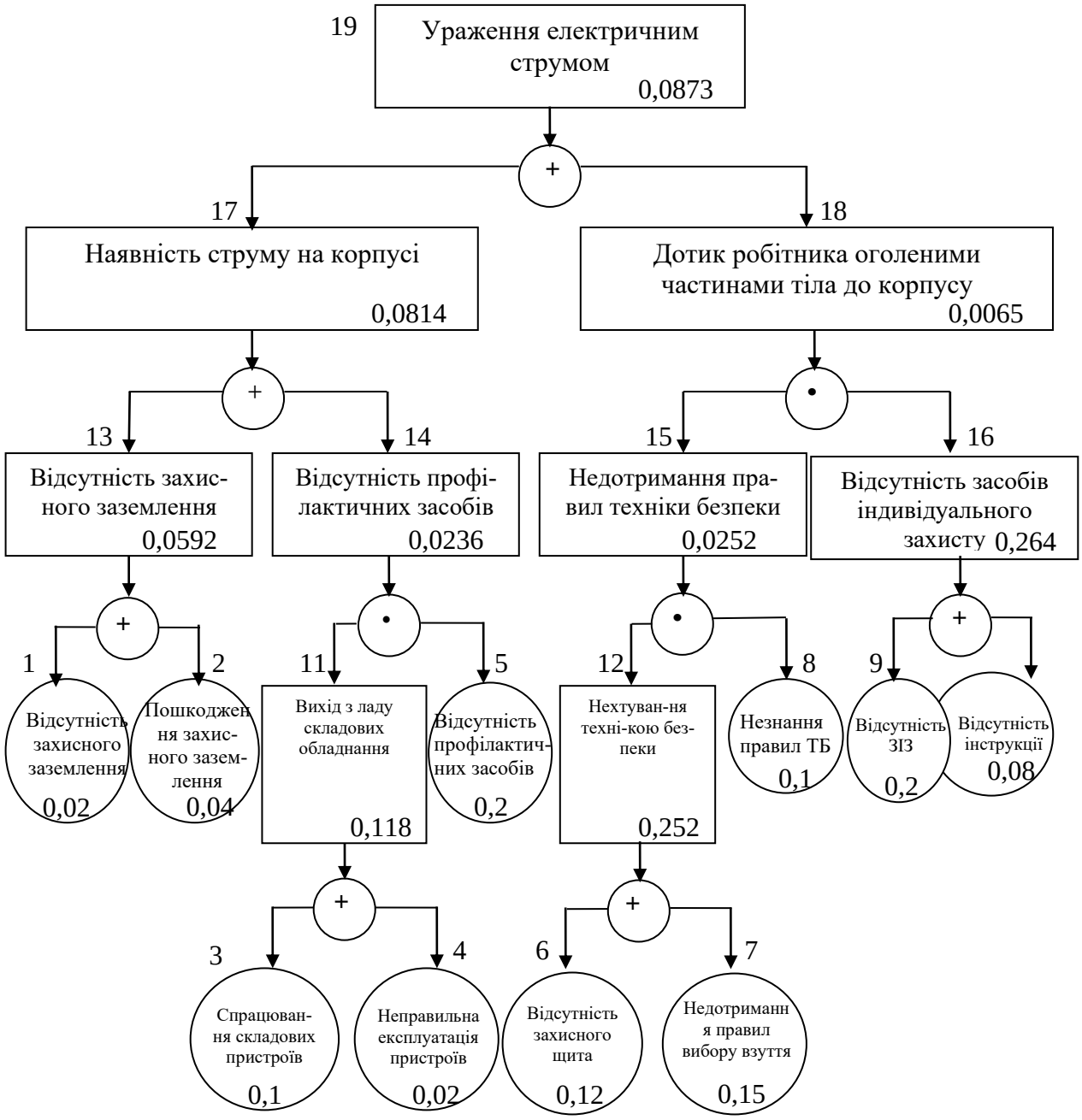

Рис. 5.1. Матриця логічних взаємозв’язків між окремими подіями травмонебезпечної ситуації [5]

Аналогічно визначаємо ймовірність інших подій:

$$P_{11} = P_4 + P_5 - P_4P_5 = 0{,}3 + 0{,}4 - 0{,}3 \cdot 0{,}4 = 0{,}118.$$

$$P_{12} = P_6 + P_7 - P_6P_7 = 0{,}3 + 0{,}5 - 0{,}3 \cdot 0{,}5 = 0{,}252.$$

$$P_{16} = P_9 + P_{10} - P_9P_{10} = 0{,}2 + 0{,}15 - 0{,}2 \cdot 0{,}15 = 0{,}264.$$

$$P_{14} = P_{11} \cdot P_5 = 0{,}118 \cdot 0{,}2 = 0{,}0236.$$

$$P_{15} = P_{12} \cdot P_8 = 0{,}252 \cdot 0{,}1 = 0{,}0252.$$

$$P_{17} = P_{13}+P_{14} - P_{13}\cdot P_{14} = 0{,}592+0{,}0236-0{,}0592\cdot 0{,}0236 = 0{,}0814.$$

$$P_{18} = P_{15}\cdot P_{16} = 0{,}264\cdot 0{,}0252 = 0{,}0065.$$

$$P_{19} = P_{17}+P_{18} - P_{17}\cdot P_{18} = 0{,}0065+0{,}0814 - 0{,}0065\cdot 0{,}0814 = 0{,}0873.$$

Таким чином, ймовірність перекидання машини та наслідкового виникнення травми працівника є досить мала і становить – $P_{19} = 0,0873$.

5.2. Планування заходів із покращення умов праці

До заходів щодо покращення умов праці належать всі види діяльності, спрямовані на попередження, нейтралізацію або зменшення негативної дії шкідливих і небезпечних виробничих факторів на працівників.

Рівень умов праці оцінюють порівнянням за фактичними і нормативними значеннями узагальнених (групових) показників.

Заходи щодо поліпшення умов праці здійснюють з метою створення безпечних умов праці шляхом:

- доведення до нормативного рівня показників виробничого середовища за елементами умов праці;

- захисту працівників від дії небезпечних і шкідливих виробничих факторів.

До показників ефективності заходів щодо поліпшення умов праці належать:

а) зміни стану умов праці:

- зміна кількості засобів виробництва, приведених у відповідність до вимог стандартів безпеки праці;

- покращання санітарно-гігієнічних показників;

- покращання психофізичних показників, зменшення фізичних і нервово-психічних навантажень, в т.ч. монотонних умов праці;

б) соціальні результати заходів:

- збільшення кількості робочих місць, що відповідають нормативним вимогам;

- зниження рівня виробничого травматизму;
- престиж та задоволення працею.

Отже, на покращення охорони праці потрібно виділити кошти на відновлення вентиляційних систем у ремонтних майстернях, естетично оформити приміщення офісу, відновити кабінет з охорони праці, поновити протипожежний інвентар.

5.3. Безпека в надзвичайних ситуаціях

Актуальність проблеми природно-техногенної безпеки для населення і території, зумовлена зростанням втрат людей, що спричиняється небезпечними природними явищами, промисловими аваріями та катастрофами. Ризик надзвичайних ситуацій природного та техногенного характеру невпинно зростає, тому питання захисту цивільного населення від надзвичайних ситуацій на сьогодні є дуже важливе [5]. У системі цивільної оборони окремого господарства необхідно забезпечити захист населення таким чином:

Укриття в захисних спорудах, якому підлягає усе населення відповідно до приналежності, досягається створенням фонду захисних споруд.

Евакуаційні заходи, які проводяться в містах та інших населених пунктах, які мають об’єкти підвищеної небезпеки, а також у воєнний час, основним способом захисту населення є евакуація і розміщення його у позаміській зоні.

Медичний захист проводиться для зменшення ступеня ураження людей, своєчасного надання допомоги постраждалим та їх лікування, забезпечення епідеміологічного благополуччя в районах надзвичайних ситуацій.

Радіаційний і хімічний захист включає заходи щодо виявлення і оцінки радіаційної та хімічної обстановки, організацію і здійснення дозиметричного та хімічного контролю, розроблення типових режимів радіаційного захисту, забезпечення засобами індивідуального захисту, організацію і проведення спеціальної обробки.

РОЗДІЛ 6.
ОЦІНКА ВИКОРИСТАННЯ КЕШ СЕРВІСУ ДЛЯ ВЕБ-СЕРВЕРА

Використання Redis з NestJS може знизити час відповіді сервера в рази, оскільки дані зберігаються у швидкодіючій оперативній пам'яті, а не на диску. Це особливо помітно при частому доступі до одних і тих самих даних.

Наприклад, веб-сервер без кешу може мати час відповіді 200 мс або більше, особливо при великому навантаженні, тоді як використання Redis може скоротити цей час до 20-30 мс, що значно підвищує продуктивність та забезпечує кращий досвід користувача.

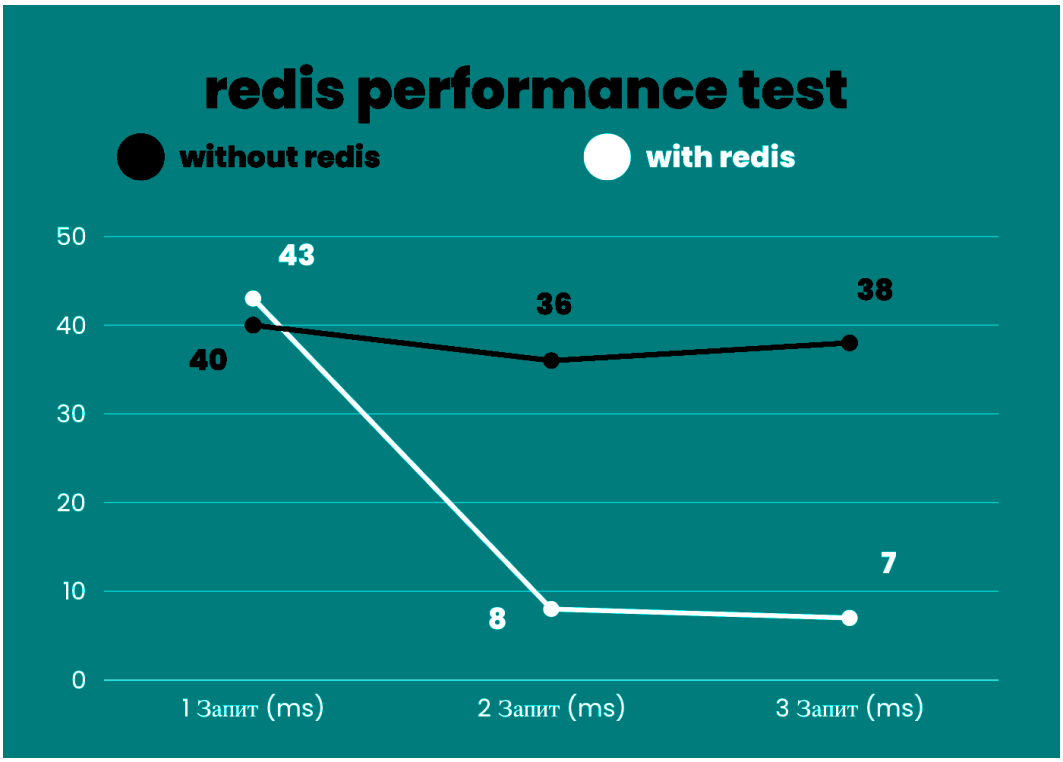

Рисунок 6.1. – Оцінка використання Redis з NestJS

Ця ефективність робить комбінацію Redis і NestJS відмінним вибором для ресурсоємних веб-додатків з високим рівнем взаємодії користувачів.

ВИСНОВКИ І РЕКОМЕНДАЦІЇ

1. Redis позиціонується як сервер кешування, але його можливості виходять далеко за межі цієї ролі. Його також можна експлуатувати як базу даних, систему повідомлень, робочу пам'ять для інших додатків та навіть як координатор для розподілених систем. Redis – розподілене сховище пар ключ-значення, які зберігаються в оперативній пам'яті, з можливістю забезпечувати довговічність зберігання на бажання користувача. Це програмне забезпечення з відкритим початковим кодом написане на ANSI C.

2. Веб-сервер – це сервер, що приймає HTTP-запити від клієнтів, зазвичай веббраузерів, видає їм HTTP-відповіді, зазвичай разом з HTML-сторінкою, зображенням, файлом, медіа-потоком або іншими даними. Веб-сервер – одна із основ Всесвітньої павутини. Веб-сервером називають як програмне забезпечення, що виконує функції вебсервера, так і комп'ютер, на якому це програмне забезпечення працює.

3. Використання Node.js для розробки веб-сервера надає можливість асинхронної обробки запитів, що сприяє високій продуктивності та знижує час відгуку. Використання NestJS Framework дає змогу отримати структурований підхід до розробки, де кожен аспект додатку поділяється на окремі модулі, що полегшує розширення та підтримку розробки.

4. Поєднання із PostgreSQL як системою управління реляційними базами даних забезпечує надійне зберігання та управління великими обсягами даних, підтримуючи складні запити та транзакції.

5. Використовувана у розробці Prisma ORM спрощує взаємодію з базою даних – перетворює складні SQL-запити в прості та інтуїтивно зрозумілі команди JavaScript, що підвищує ефективність розробки та знижує ймовірність помилок.

6. Застосування Passport.js для аутентифікації користувачів як декілька стратегій Passport, включаючи JWT для забезпечення безпеки сесій, Google OAuth для інтеграції з соціальними мережами та систему access та refresh tokens для

більш гнучкого управління доступом. Це дозволяє нам забезпечити різноманітні та безпечні способи аутентифікації для користувачів.

7. Оптимізація продуктивності веб-сервера в нашій архітектурі веб-сервера досягається через стратегічне використання технік кешування. Кешування даних відіграє ключову роль у зниженні часу відгуку сервера та зменшенні навантаження на базу даних, шляхом тимчасового зберігання важливих, часто запитуваних даних в швидкісній пам'яті.

8. Це не тільки покращує час відклику для користувачів, але й ефективно зменшує затримки при високій конкуренції запитів, забезпечуючи рівномірний розподіл навантаження на системні ресурси. В системі для реалізації кешування використовуємо Redis, високопродуктивну базу даних в пам'яті, яка служить як система для кешування та брокера повідомлень. Redis є ідеальним вибором для цих завдань.

9. Отже, архітектура веб-сервера відображає сучасні підходи до розробки веб-серверів, демонструючи високу гнучкість, надійність та масштабованість. Використання передових технологій, таких як Node.js, NestJS, Prisma, PostgreSQL, а також інтеграція з Backblaze B2 Storage та впровадження багатошарової системи аутентифікації через Passport.js, створює міцну основу для розробки високопродуктивного веб-додатку, здатного ефективно обробляти великі обсяги даних і забезпечувати надійний захист інформації. Така структура і підхід до розробки не тільки полегшують майбутнє розширення та адаптацію системи під нові вимоги, але й забезпечують зручність управління та підтримки системи, що є важливим аспектом для створення стійких та адаптивних веб-рішень.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Від мікросервісного моноліту до оркестратору. URL: https://blog.byndyu.ru/2020/04/blog-post_14.html?m=1 (дата звернення: 27.12.2024).

2. Джонсон Е. Розробка веб-сайтів з використанням Node.js, Express та MongoDB. O'Reilly Media, 2014. 352 с.

3. Клієнт-сервер. URL: https://developer.mozilla.org/ (дата звернення: 20.12.2024).

4. Комплексна система IT-рішень для управління агробізнесом. URL: https://agrichain.com.ua/ (дата звернення: 20.12.2024).

5. Лехман С.Д. та ін. Запобігання аварійності і травматизму у сільському господарстві / С.Д. Лехман, В.І. Рубльов, Б.І. Рябцев. К.: Урожай, 1993. 272 с.

6. Майер, Е. А. CSS: Визначний посібник: Візуальна презентація для вебу. O'Reilly Media, 2017. 1080 с.

7. Модель «Клієнт-Сервер» URL: http://window.edu.ua/ (дата звернення: 27.12.2024).

8. Монолітна архітектура. URL: https://www.techtarget.com/whatis/definition/monolithic-architecture (дата звернення: 10.12.2024).

9. Новіков Ф.В., Богатов О.І., Навчальний посібник «Безпека життєдіяльності», Харків 2018, 250 с.

10. Переваги та недоліки використання хмарних технологій підприємствами України. URL: http://www.bsfa.edu.ua/files/konf2013/62.pdf (дата звернення: 20.12.2024).

11. Спірін О. М. Зміст навчального матеріалу спецкурсу "Хмарні інформаційно-аналітичні технології у науково-дослідному процесі". Інформаційні технології і засоби навчання. 2016. Т. 52, вип. 2. С. 108-120.

12. Фрімен Е., Робсон Е. Програмування JavaScript, 2022, 672 с.

13. API Gateway. URL: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do (дата звернення: 04.12.2024).

14. Architectural patterns. URL : https://www.ou.nl/documents/40554/791670/IM0203_03.pdf/30dae517-691e-b3c7-22ed-a55ad27726d6 (дата звернення: 04.12.2024).

15. Big Data information technology and data space architecture / N. Shakhovska, O. Veres // Sensors &Transducers. 2015. Vol. 195, Is.12. P. 69-77.

16. Desktop vs Browser not just another Thick vs Thin Client debate. URL: https://www.msafocus.com/news/desktop-vs-browser. (дата звернення: 20.12.2024).

17. Docker Compose. URL: https://docs.docker.com/compose/ (дата звернення: 08.12.2024).

18. Gillis A. REST API (RESTful API) / Alexander Gillis // TechTarget. 2020. URL: https://searchapparchitecture.techtarget.com/definition/RESTful-API (дата звернення: 27.12.2024).

19. How to Design a Web Application. URL: https://www.educative.io/blog/how-to-design-a-webapplication-software-architecture-101 (дата звернення: 27.12.2024).

20. Introduction to the server side // MDN. URL: https://developer.mozilla.org/en-US/docs/Learn/Serverside/First_steps/Introduction (дата звернення: 27.12.2024).

21. Kubernates Concepts. URL: https://kubernetes.io/docs/concepts/overview/components/ (дата звернення: 02.12.2024).

22. Learn Microservices with Spring Boot. ISBN-13 (pbk): 978-1-4842-6130-9 ISBN-13 (electronic): 978-1-4842-6131-6 https://doi.org/10.1007/978-1-4842-6131-6 (дата звернення: 20.12.2024).

23. Mauro T. Adopting Microservices at Netflix: Lessons for Architectural Design / Tony Mauro // Nginx. 2015. URL: https://www.nginx.com/blog/microservices-at-netflix-architectural-bestpractices/ (дата звернення: 27.12.2024)

24. Most Popular Software Architectural Patterns. URL: https://nix-united.com/blog/10-common-softwarearchitectural-patterns-part-1/. (дата звернення: 27.12.2024)

25. Pattern: Microservice Architecture. URL: https://microservices.io/patterns/microservices.html (дата звернення: 27.12.2024).

26. Redis: The Complete Developer's Guide. Master Redis v7.0 with hands-on exercises. Includes Modules, Scripting, Concurrency, and Streams! URL: https://www.udemy.com/topic/redis/ (дата звернення: 27.12.2024).

27. Svrtan D. Ready for changes with Hexagonal Architecture / D. Svrtan, S. Makagon // Netflix Technology Blog. 2020. URL: https://netflixtechblog.com/ready-for-changes-withhexagonal-architecture-b315ec967749 (дата звернення: 27.12.2024).

28. What is an API? // MuleSoft. URL: https://www.mulesoft.com/resources/api/what-is-an-api (дата звернення: 27.12.2024).

29. What Is MongoDB? URL: https://www.mongodb.com/what-is-mongodb (дата звернення: 27.12.2024)

30. What Is Software Architecture? // Cast Software. URL: https://www.castsoftware.com/glossary/what-issoftware-architecture-tools-design-definition-explanation-best (дата звернення: 27.12.2024).

ДОДАТКИ

Додаток А.

Фрагмент коду Main.ts – файл запуску застосунку

```
import { ValidationPipe } from '@nestjs/common';
import { ConfigService } from '@nestjs/config';
import { NestFactory } from '@nestjs/core';
import { NestExpressApplication } from '@nestjs/platform-express';
import { DocumentBuilder, SwaggerCustomOptions, SwaggerModule } from '@nestjs/swagger';
import * as fs from 'fs';
import { PinoLogger } from 'nestjs-pino';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestExpressApplication>(AppModule);
  const configService: ConfigService = app.get(ConfigService);
  const SERVER_PORT = configService.get<number>('SERVER_PORT') || 5000;
  const isProduction = configService.get<string>('NODE_ENV') === 'production';
  const logger = new PinoLogger({});
  logger.setContext('bootstrap');
  const corsOptions = {
    origin: '*',
    methods: 'GET,HEAD,PUT,PATCH,POST,DELETE',
    preflightContinue: false,
    optionsSuccessStatus: 204,
    credentials: true,
  };
  const swaggerUrlPath = configService.get<string>('SWAGGER_URL_PATH');

  if (!isProduction) {
    const swaggerConfig = new DocumentBuilder()
      .setTitle('WebShop API')
      .setDescription('WebShop API description')
      .setVersion('1.0')
      .addServer('http://localhost:' + SERVER_PORT, 'WebShop Local')
      .addBearerAuth()
      .build();
    const swaggerCustomOptions: SwaggerCustomOptions = {
      swaggerOptions: {
        docExpansion: 'none',
      },
    };
    const swaggerDocument = SwaggerModule.createDocument(app, swaggerConfig);

    SwaggerModule.setup(swaggerUrlPath, app, swaggerDocument, swaggerCustomOptions);
    fs.writeFileSync('swagger.json', JSON.stringify(swaggerDocument), {
      encoding: 'utf8',
    });
  }

  app.enableCors(corsOptions);
  app.useGlobalPipes(
    new ValidationPipe({
```

```
      transform: true,
      whitelist: true,
      forbidNonWhitelisted: true,
      transformOptions: {
        enableImplicitConversion: true,
      },
      validationError: {
        value: true,
        target: true,
      },
    })
  );

  await app.listen(SERVER_PORT, () => {
    if (!isProduction) {
      logger.info(`Application is connected to the database: ${configService.get<string>('DATABASE_URL')}`);
      logger.info(`WebShop API Documentation on: http://localhost:${SERVER_PORT}/${swaggerUrlPath}`);
    }

    logger.info(`Application listen on: http://localhost:${SERVER_PORT}`);
  });
}

bootstrap();
```

Додаток Б.

Фрагмент файлу *user.controller.ts*

```
import { Body, Controller, Delete, Get, Patch, Req } from '@nestjs/common';
import { ApiBearerAuth, ApiTags } from '@nestjs/swagger';
import { RoleEnum } from '@prisma/client';
import { AuthWithRoles } from 'src/decorators/auth.decorator';
import { UserAddAddressDto } from './dto/user-add-address.dto';
import { UserAddRolesDto } from './dto/user-add-roles.dto';
import { UserRemoveRolesDto } from './dto/user-remove-roles.dto';
import { UserUpdateAddressDto } from './dto/user-update-address.dto';
import { UserUpdateDto } from './dto/user-update.dto';
import { UserService } from './user.service';

@ApiTags('User')
@Controller('user')
export class UserController {
  constructor(private readonly userService: UserService) {}

  @Get('info')
  @ApiBearerAuth()
  @AuthWithRoles()
  info(@Req() req) {
    return req.user;
  }

  @Patch('update')
  @ApiBearerAuth()
  @AuthWithRoles()
  update(@Body() body: UserUpdateDto, @Req() req) {
    return this.userService.update(body, req.user.userUuid);
  }

  @Patch('add-roles')
  @ApiBearerAuth()
  @AuthWithRoles([RoleEnum.ADMIN])
  addRoles(@Body() body: UserAddRolesDto) {
    return this.userService.addRoles(body);
  }

  @Patch('add-address')
  @ApiBearerAuth()
  @AuthWithRoles()
  addAddress(@Body() body: UserAddAddressDto, @Req() req) {
    return this.userService.addAddress(body, req.user.userUuid);
  }

  @Patch('update-address')
  @ApiBearerAuth()
  @AuthWithRoles()
  updateAddress(@Body() body: UserUpdateAddressDto, @Req() req) {
    return this.userService.updateAddress(body, req.user.userUuid);
  }
```

```
  @Patch('remove-roles')
  @ApiBearerAuth()
  @AuthWithRoles([RoleEnum.ADMIN])
  removeRoles(@Body() body: UserRemoveRolesDto) {
    return this.userService.removeRoles(body);
  }

  @Delete('delete')
  @ApiBearerAuth()
  @AuthWithRoles()
  delete(@Req() req) {
    return this.userService.delete(req.user.userUuid);
  }
}
```

Додаток В.

Фрагмент файлу *user.service.ts*

```
import { BadRequestException, Injectable } from '@nestjs/common';
import { Prisma } from '@prisma/client';
import { PinoLogger } from 'nestjs-pino';
import { UtilsService } from 'src/shared/utils/utils.service';
import { UserAddAddressDto } from './dto/user-add-address.dto';
import { UserAddRolesDto } from './dto/user-add-roles.dto';
import { UserRemoveRolesDto } from './dto/user-remove-roles.dto';
import { UserUpdateAddressDto } from './dto/user-update-address.dto';
import { UserUpdateDto } from './dto/user-update.dto';
import { AddressRepository } from './repositories/address.repository';
import { RoleRepository } from './repositories/role.repository';
import { UserRepository } from './repositories/user.repository';

@Injectable()
export class UserService {
  constructor(
    private readonly userRepository: UserRepository,
    private readonly roleRepository: RoleRepository,
    private readonly addressRepository: AddressRepository,
    private readonly logger: PinoLogger
  ) {
    this.logger.setContext(UserService.name);
  }

  async update(userUpdateDto: UserUpdateDto, userUuid: string) {
    const user = await this.userRepository.findOneAndThrowIfNotExist({
      where: { userUuid },
    });

    const userUpdateData: Prisma.UserUpdateInput = {
      firstName: userUpdateDto?.firstName || user.firstName,
      lastName: userUpdateDto?.lastName || user.lastName,
      dateOfBirth: userUpdateDto?.dateOfBirth || user.dateOfBirth,
      language: userUpdateDto?.language || user.language,
      theme: userUpdateDto?.theme || user.theme,
    };

    if (userUpdateDto?.username) {
      const isUsernameExists = await this.userRepository.findOneWithoutChecking({
        where: { username: userUpdateDto.username },
      });

      if (isUsernameExists) throw new BadRequestException('User with this username already exists!');

      userUpdateData.username = userUpdateDto?.username;
    }

    if (userUpdateDto?.password) {
      userUpdateData.passwordHash = await UtilsService.hashData(userUpdateDto?.password);
    }

    if (userUpdateDto.firstName || userUpdateDto.lastName) {
      userUpdateData.userInitial = `${userUpdateDto?.firstName?.charAt(0) || user.firstName?.charAt(0)}${
        userUpdateDto.lastName?.charAt(0) || user.lastName?.charAt(0)
      }`;
      userUpdateData.displayName = `${userUpdateDto?.firstName || user.firstName} ${
        userUpdateDto.lastName || user.lastName
      }`;
```

```
    }

    const updatedUser = await this.userRepository.update({
      where: { userUuid },
      data: userUpdateData,
      include: { roles: true, address: true },
    });

    return this.userRepository.exclude(updatedUser, ['passwordHash', 'refreshToken']);
  }

  async addRoles(userAddRolesDto: UserAddRolesDto) {
    await this.userRepository.findOneAndThrowIfNotExist({ where: { userUuid: userAddRolesDto.userUuid }
});
    await Promise.all(
      userAddRolesDto.roleUuids.map((roleUuid) =>
        this.roleRepository.findOneAndThrowIfNotExist({ where: { roleUuid } })
      )
    );

    const updatedUser = await this.userRepository.update({
      where: { userUuid: userAddRolesDto.userUuid },
      data: {
        roles: {
          connect: userAddRolesDto.roleUuids.map((roleUuid) => {
            return { roleUuid };
          }),
        },
      },
      include: { roles: true, address: true },
    });

    return this.userRepository.exclude(updatedUser, ['passwordHash', 'refreshToken']);
  }

  async addAddress(userAddAddressDto: UserAddAddressDto, userUuid: string) {
    await this.userRepository.findOneAndThrowIfNotExist({ where: { userUuid } });
    await this.addressRepository.create({
      data: {
        ...userAddAddressDto,
        user: {
          connect: { userUuid },
        },
      },
    });

    const userWithAddress = await this.userRepository.findOneByUniqueField({
      where: { userUuid },
      include: { address: true, roles: true },
    });

    return userWithAddress;
  }

  async updateAddress(userUpdateAddressDto: UserUpdateAddressDto, userUuid: string) {
    await this.userRepository.findOneAndThrowIfNotExist({ where: { userUuid } });
    await this.addressRepository.update({
      where: { userUuid },
      data: userUpdateAddressDto,
    });

    const userWithAddress = await this.userRepository.findOneByUniqueField({
      where: { userUuid },
```

```
      include: { address: true, roles: true },
    });

    return userWithAddress;
  }

  async removeRoles(userRemoveRolesDto: UserRemoveRolesDto) {
    await this.userRepository.findOneAndThrowIfNotExist({ where: { userUuid:
userRemoveRolesDto.userUuid } });
    await Promise.all(
      userRemoveRolesDto.roleUuids.map((roleUuid) =>
        this.roleRepository.findOneAndThrowIfNotExist({ where: { roleUuid } })
      )
    );

    const updatedUser = await this.userRepository.update({
      where: { userUuid: userRemoveRolesDto.userUuid },
      data: {
        roles: {
          disconnect: userRemoveRolesDto.roleUuids.map((roleUuid) => {
            return { roleUuid };
          }),
        },
      },
      include: { roles: true, address: true },
    });

    return this.userRepository.exclude(updatedUser, ['passwordHash', 'refreshToken']);
  }

  async delete(userUuid: string) {
    await this.userRepository.findOneAndThrowIfNotExist({
      where: { userUuid },
      include: {
        roles: true,
        address: true,
      },
    });

    const result = await this.userRepository.delete({ where: { userUuid } });

    return result;
  }
}
```