МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

другого (магістерського) рівня вищої освіти

на тему: **“Розробка архітектури веб-сервера із застосуванням Node.js та NestJS Framework”**

Виконав: __ст. гр. Іт-61________________

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

_____Третяк Михайло Віталійович___________
(Прізвище та ініціали)

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

Рецензенти: ________________________________
(Прізвище та ініціали)

ДУБЛЯНИ-2024

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ПРИРОДОКОРИСТУВАННЯ

ФАКУЛЬТЕТ МЕХАНІКИ, ЕНЕРГЕТИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
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

Третяку Михайлу Віталійовичу

1. Тема роботи: «Розробка архітектури веб-сервера із застосуванням Node.js та NestJS Framework»

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 15.01.2024 р.
3. Початкові дані до роботи: 1. Структура веб-сервера; 2. Технології Node.js та NestJS Framework; 3. Технологія доступу та обміну файлами Backblaze B2 Storage; 4. Стратегія аутентифікації – PassportJS.

4. Зміст розрахунково-пояснювальної записки:

1. *Аналіз головних понять веб-сайтів та архітерури веб-серверів*
2. *Технології розробки архітектури веб-сервера*
3. *Методика побудови архітектури веб-сервера*
4. *Програмна реалізація веб-сервера*
5. *Охорона праці та безпека в надзвичайних ситуаціях*

Висновки та пропозиції.
Бібліографічний список.
Додатки.

5. Перелік графічного матеріалу: 1 та 2 – Тема, мета, завдання роботи; 3 – Огляд використаних техногій; 4 – Архітектура веб-сервера; 5 та 6 – Безпека та аутентифікація; 7 та 8 – Інтеграція з базою даних; 9 – Схема бази даних; 10 – Документація та API; 11 – Діаграми програмної реалізації веб-сервера; 12 – Фрагменти практичного використання веб-сервера; 13-15 – Результати практичного використання веб-сервера; 16 – Порівняльна оцінка використання кеш сервісу для веб-серверу.

6. Консультанти з розділів:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1, 2, 3, 4	*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	Примітка
1.	*Написання першого розділу та означення головних завдань роботи*	*28.04-20.06.23*	
2.	*Виконання другого розділу та опис інформаційних технологій для виконання завдань роботи*	*21.06-14.08.23*	
3.	*Виконання третього розділу, методика вирішення завдань та елементи наукових досліджень*	*15.08-31.10.23*	
4.	*Виконання четвертого розділу, опис розробки*	*15.08-31.10.23*	
5.	*Написання розділу: «Охорона праці та безпека в надзвичайних ситуаціях»*	*01.11-10.11.23*	
6.	*Головні результати отримані в роботі, оцінення розроблених пропозицій*	*20.11-30.11.23*	
7.	*Завершення оформлення розрахунково-пояснювальної записки та презентаційних матеріалів*	*30.11-01.12.23*	
	Завершення роботи в цілому	*01.12.23-10.01.24*	

Студент _________________ Третяк М.В.
(підпис)

Керівник роботи _________________ Луб П.М.
(підпис)

УДК: 004.94: 631.1

Кваліфікаційна робота: 78 с. текст. част., 33 рис., 1 табл., 17 слайдів, 34 джерела.

Розробка архітектури веб-сервера із застосуванням Node.js та NestJS Framework. Третяк М. В. Кафедра ІТ. – Дубляни, Львівський НУП, 2024.

Наведено ключові поняття веб-сайтів та їх класифікацію. Проаналізовано архітектуру веб-додатків. Виконано аналіз монолітної та мікросервісної архітектури веб-серверів.

Розкрито особливості клієнт-серверної архітектури. Запропоновано вибір технологій розробки архітектури веб-сервера. Наведено опис архітектури веб-сервера. Розкрито спосіб забезпечення безпеки та аутентифікації.

Описано мови програмування та інструменти розробки. Розкрито спосіб інтеграції із базою даних комплексних за стосунків. Наведено схему взаємодії із базою даних та документацію, API.

Наведено результати розробки бази даних. Виконано проектування та програмування системи. Подано інструкцію з використання.

Здіснено порівняльну оцінку використання кеш сервісу для веб-серверу.

Означено вимоги охорона праці та безпека в надзвичайних ситуаціях.

Ключові слова: сервіс, архітектура, розробка, веб-сервер, Node.js, NestJS, Framework, ефективність.

ЗМІСТ

ВСТУП……………………………………………………………………… 6
РОЗДІЛ 1
АНАЛІЗ ГОЛОВНИХ ПОНЯТЬ ВЕБ-САЙТІВ ТА АРХІТЕРУРИ ВЕБ-СЕРВЕРІВ……………………………………………………..……….. 8
1.1. Ключові поняття веб-сайтів та їх класифікація ………………….. 8
1.2. Аналіз архітектури веб-додатків………………………………… 11
1.3. Аналіз монолітної та мікросервісної архітектури веб-серверів… 15
РОЗДІЛ 2
ТЕХНОЛОГІЇ РОЗРОБКИ АРХІТЕКТУРИ ВЕБ-СЕРВЕРА……………….. 22
2.1. Особливості клієнт-серверної архітектури…………………….. 22
2.2. Вибір технологій розробки архітектури веб-сервера…………….. 26
2.3. Архітектура веб-сервера……………………………..………... 30
2.4. Безпека та аутентифікація……………..……..………………….. 31
РОЗДІЛ 3
МЕТОДИКА ПОБУДОВИ АРХІТЕКТУРИ ВЕБ-СЕРВЕРА……………..… 34
3.1. Мови програмування та інструменти розробки………………... 34
3.2. Інтеграція із базою даних комплексних застосунків…………... 37
3.3. Схема взаємодії із базою даних……...………………………….. 39
3.4. Документація та API…………………………………………... 41
РОЗДІЛ 4.
ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-СЕРВЕРА……………………………….. 43
4.1. Результати розробки бази даних…………………………………… 43
4.2. Проектування та програмування системи ………………………… 45
4.3. Інструкція з використання ……………………………………… 54
4.4. Порівняльна оцінка використання кеш сервісу для веб-серверу... 60
РОЗДІЛ 5.
ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ…... 61
5.1. Розробка логіко-імітаційної моделі виникнення травм і аварій…. 61
5.2. Планування заходів із покращення умов праці…………………… 63
5.3. Безпека в надзвичайних ситуаціях……………………………..... 64
ВИСНОВКИ І РЕКОМЕНДАЦІЇ……………………………………..... 65
БІБЛІОГРАФІЧНИЙ СПИСОК……………………………………………. 67
ДОДАТКИ……………………………………………………………….. 70

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 []. 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 високоефективного, масштабованого веб-сервера з використанням сучасних технологій, таких як Node.js та NestJS Framework.

Завдання роботи: 1) дослідити можливості та переваги Node.js та NestJS у

контексті розробки веб-серверів; 2) розробити архітектуру веб-сервера, з урахуванням модульності, масштабованості та безпеки; 3) реалізація основних функцій веб-сервера, включаючи аутентифікацію, управління даними через Prisma, обробку зображень та інтеграцію з внутрішніми/зовнішніми API; 4) підготовка детальної документації API.

Об’єкт роботи – веб-сервер для інтернет-магазину, розроблений з використанням Node.js та NestJS Framework.

Предмет роботи – роцес розробки архітектури веб-сервера, включаючи вибір стеку технологій, проектування структури сервера, реалізацію функціональності.

Новизна результатів:

- використання комбінації Node.js та NestJS Framework забезпечує високу продуктивність, гнучкість та зручність у розробці;
- розробка модульної структури сервера, яка дозволяє легко масштабувати та розширювати функціональність;
- впровадження ефективних методів для управління даними через Prisma ORM, що покращує продуктивність та безпеку взаємодії з базами даних;
- реалізація ефективної інтеграції з внутрішніми та зовнішніми сервісами.

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 міцну основу для високопродуктивного веб-додатку, здатного ефективно обробляти великі обсяги даних і забезпечувати надійний захист інформації. Така структура і підхід до розробки полегшують майбутнє розширення та адаптацію системи під нові вимоги, забезпечують зручність управління та підтримки системи.

РОЗДІЛ 1
АНАЛІЗ ГОЛОВНИХ ПОНЯТЬ ВЕБ-САЙТІВ ТА АРХІТЕРУРИ ВЕБ-СЕРВЕРІВ

1.1. Ключові поняття веб-сайтів та їх класифікація

Веб-сторінка – це інформаційний ресурс доступний в мережі World Wide Web (Всесвітня павутина), який можна переглянути у веб-браузері зазвичай, ця інформація записана в форматі HTML або XHTML, і може містити гіпертекст з навігаційними гіперпосиланнями на інші веб-сторінки.

Гіпертекст – це текст для перегляду на комп'ютері, який містить зв'язки з іншими документами («гіперзв'язки» чи «гіперпосилання»); читач має змогу перейти до пов'язаних документів безпосередньо з вихідного (первинного) тексту, активізувавши посилання. Загальновідомим та яскраво вираженим прикладом сучасного гіпертексту слугують веб-сторінки, підготовлені за допомогою HTML (мова розмітки гіпертексту) й розміщені у web-мережі. Відповідно, в комп'ютерній термінології під гіпертекстом розуміється текст, сформований за допомогою мови розмітки, що потенційно містить у собі гіперпосилання. Гіперпосилання дозволяють переходити від вихідного (одного) тексту до безлічі інших текстів, розміщених у web-мережі.

Елементи веб-сторінок:

Контент – змістове наповнення веб-сторінки, доступне користувачеві:

1) за формою подачі: текст, зображення, аудіо, відео;

2) за доступністю: безкоштовний, платний;

3) за унікальністю: унікальний, неунікальний.

Елементи навігації – засоби для переходу на інші веб-сторінки.

Елементи дизайну – елементи структурування контенту та його форматування, оформлення сторінки.

Додаткові елементи веб-сторінок:

Банер – це невеличке рекламне зображення, що розміщується на одній

сторінці, а посилається на іншу.

Форма – частина веб-сторінки, до якої користувач може записувати свої відомості та надсилати їх на сервер, де розміщується веб-сторінка.

Веб-сайт – сукупність веб-сторінок, доступних у Інтернеті, які об'єднані як за змістом, так і навігаційно.

Сайтом також називають вузол мережі Інтернет, комп'ютер, за яким закріплена унікальна IP-адреса, і взагалі будь-який об'єкт в Інтернеті, за яким закріплена адреса, що ідентифікує його в мережі (FTP-site, WWW-site тощо).

Структура сайту

- Внутрішня – дозволяє всі необхідні дані розмістити зручно і доступно в розділах та підрозділах.

- Зовнішня – визначає спосіб розміщення відомостей на сторінці при виборі кожного пункту меню, визначає дизайн сайту.

Внутрішня структура сайту

- Лінійна (послідовна) – доцільно використовувати у разі послідовного подання даних та відомостей; перегляд послідовний: від початкової до останньої сторінки сайту.

- Ієрархічна – одна сторінка головна, яка не має попередніх, решта сторінок мають лише одну попередню сторінку.

Довільна – кожна сторінка може містити посилання на довільну кількість сторінок сайту.

- Гібридна – поєднання ієрархічної і послідовної структур.

- Мережна – базується на побудові системи такої навігації сайту, коли між горизонтальними та вертикальними елементами існує взаємний зв'язок та можливості швидкого переходу з однієї сторінки на іншу без додаткової необхідності відвідування проміжних сторінок.

Головна сторінка сайту має задовольняти правило “3 по 3”, тобто 3 абзаци по 3 речення про тематику, зміст, авторів та призначення сайту. Інколи замість головної сторінки з'являється заставка – велике фото зі звуковим супроводом або відеороликом. Такі сторінки створюють, що привернути увагу користувачів.

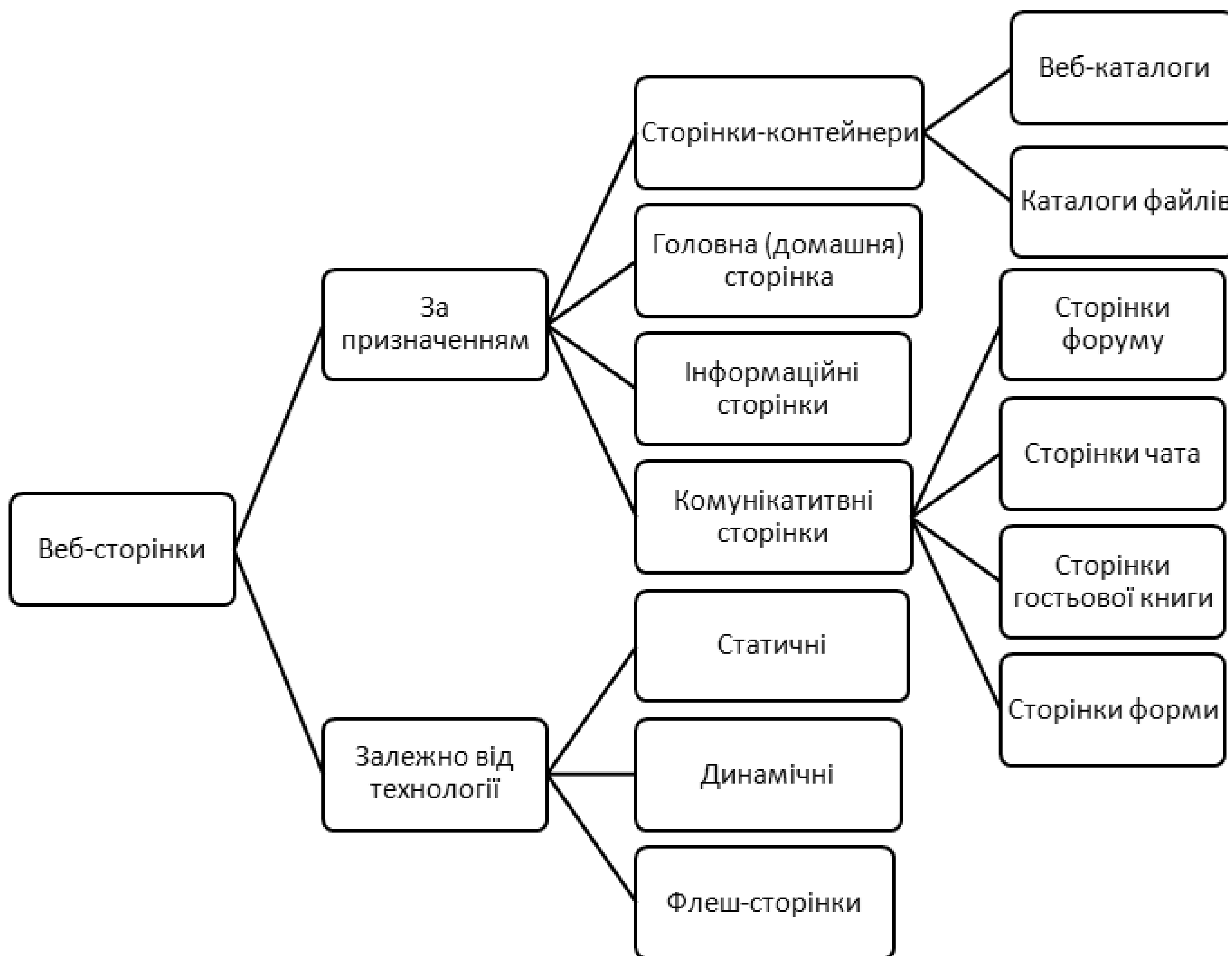


Рисунок 1.1 – Класифікація веб-сторінок

Основні вимоги до веб-сайтів:

- Наявність головної сторінки зі стислим каталогом усіх розділів (рубрик) сайту. З кожної сторінки сайту мають бути посилання на головну сторінку.

- Наявність швидкої і зручної навігації та пошуку.

- Наявність на головній сторінці сайту стислої інформації для відвідувачів сайту про його структуру і змістове наповнення.

- Оновлення сторінок сайту.

- За наявності великої кількості текстової інформації на сайті доречно розміщувати окрім web-сторінок з текстами також архіви у форматі ***Zip*** чи ***Rar***.

Сторінки сайту мають бути зручними для користувачів сайту, а також для роботи з ними.

- Інформація на сайті може бути представлена українською, російською, англійською мовами.

- Сайт повинен містити засоби зворотнього зв'язку.

Оцінювання веб-сайтів:

- Ступінь інформативності;
- Частота оновлення даних на сайті;
- Наявність інтерактивних засобів;
- Доступні мови для відтворення веб-сайту;
- Візуальна привабливість сайту;
- Зручність користування;
- Зрозумілість для користувача.

Таким чином, багато сайтів на якийсь спосіб займаються заробітком, через розміщення реклами або надання платних послуг користувачам. Однак, вони були створені спочатку не для прибутку, а для задоволення потреб користувачів, тому їх можна вважати некомерційними. Але в таких випадках границя між комерційними і некомерційними сайтами стає дещо нечіткою, і багато з цих сайтів можна назвати "умовно некомерційними".

1.2. Аналіз архітектури веб-додатків

Для початку слід подати уточнення: Веб-сайти це узагальнена назва. Вони можуть бути:

- **статичні**: ті що не містять серверної частини. І являють собою набір файлів на хостингу. А для зміни потрібно просто редагування в HTML, JS, CSS.
- **динамічними або веб-додатки**: являють собою програму, що будує відповідь на запит користувача та може включати використання баз даних та інших технологій.

Архітектура веб-додатків описує взаємозв’язок між компонентами веб-додатку, їхню структуру та спосіб взаємодії. Вона визначає, як розподіляються функції та обов’язки між різними частинами додатку, щоб забезпечити його ефективну роботу та легкість розширення.

Одним з найпоширеніших підходів до архітектури веб-додатків є клієнт-серверна модель. У цій моделі, веб-додаток складається з двох основних компонентів: клієнта та сервера. Клієнт (зазвичай веб-браузер) відповідає за відображення інтерфейсу користувача та збирання введених даних, тоді як сервер обробляє запити клієнта, виконує бізнес-логіку та надає необхідні дані.

Веб-додатки складаються з різних компонентів, які співпрацюють між собою для забезпечення функціональності додатку. Основні компоненти включають:

- *Клієнтська частина (Front-end)*: Це інтерфейс, який відображається користувачеві у веб-браузері. Вона включає HTML, CSS та JavaScript, які використовуються для створення інтерактивних елементів та забезпечення користувацького досвіду.
- *Серверна частина (Back-end)*: Це сервер, який обробляє запити клієнта, зберігає та обробляє дані, виконує бізнес-логіку та повертає відповіді клієнту. Вона може включати різні технології, такі як веб-сервери, бази даних, мови програмування тощо.

У архітектурі веб-додатків використовуються різні шаблони, які допомагають організувати та структурувати компоненти додатку. Деякі з найпоширеніших архітектурних шаблонів включають монолітну архітектуру, клієнт-серверну архітектуру, розподілену архітектуру та мікросервісну архітектуру.

Розглянемо саме серверну частину, від самого простого налаштування з розвитком до більш масштабованих та відмовостійких рішень. Розглянемо набір з 2 компонентів:

- *додаток* (*Application*);
- *база даних* (*DB*).

На один сервер встановлюємо і додаток, і базу даних.

Це робочий варіант. Для початку цього достатньо і багато веб додатків працюють саме в такій конфігурації. До плюсів можем віднести простоту. А до мінусів обмеженість в маштабованості та надійності.

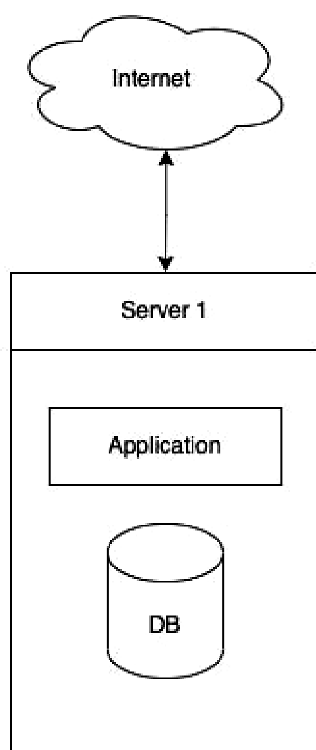


Рисунок 1.2 – App + DB

Маштабованість та надійність (відмовостійкість) це основні критерії поліпшення яких будує шлях розвитку архітектури додатку.

Кешування — це процес збереження копій даних, або результатів обчислень з метою їх подальшого використання. У контексті веб-додатків, кешування використовується для збереження сторінок, ресурсів, запитів до бази даних або інших частин контенту, що часто використовуються.

Зазвичай в веб-додатках маємо справу з додавання та видачею інформації. І співвідношення цих операцій зазвичай таке, що додаємо інформацію не так часто як потрібно її віддавати.

Сервер може зберігати копії статичних, або динамічних ресурсів у своєму кеші. Це дозволяє серверу швидше генерувати відповіді на запити, оскільки він може повертати кешовані результати замість виконання складних операцій обробки даних.

Кешування доволі розповсюджена практика. Можна зберігати результат рендерингу сторінки і на кожен новий запит віддавати уже сторінку з кешу не роблячи запити в базу. Таке рішення дозволяє зменшити час відповіді сервера та затрати на мережевий трафік, оскільки кешовані ресурси можуть бути швидше доступні для клієнтів. Налаштування кешування потребує додаткової логіки в вашому додатку. Перевірки наявності кешу та інвалідації кеш запису у випадку зміни того контенту що вже був закешований.

Система може працювати без кешу, але його використання збільшує відмовостійкість додатку.

Зберігання медіа контенту. Ще один з суттєвих моментів для веб додатків це зберігання медіа контенту. У випадку того ж блогу ви захочете додавати до ваших статей додаткову візуалізацію. Як то картинки чи відео. І цей матеріал потрібно десь зберігати.

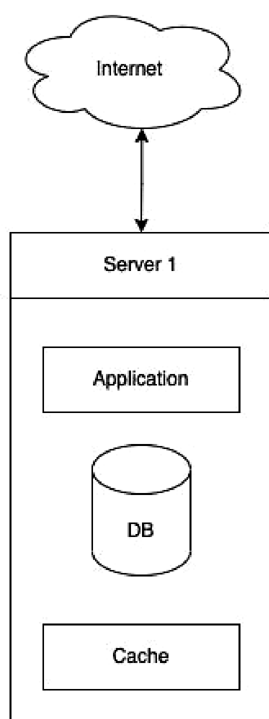


Рисунок 1.3 – App + DB + Cache

Для початку це може бути диск вашого серверу. Зазвичай дисковий простір доволі дорого коштує, а також, є вірогідність що вибраного вами диску з часом буде не достатньо за розмірами, і доведеться мігрувати систему на новий сервер. Це потребуватиме додаткових робіт. Тому якщо медіа матеріалів планується зберігати багато, або ви плануєте дати вашим клієнтам можливість завантажувати якісь матеріали, краще розглядати варіант зберігання такого контенту в зовнішніх сервісах. Більшість хмарних провайдерів надають сервіси об’єктних сховищ. Для прикладу це може бути хмарне об’єктне сховище Amazon S3 або Cloud Storage від Google Cloud. Також ви можете розглянути можливість розгорнути свій власний сервіс на базі MinIO.

Масштабування. Наступними кроками в поліпшені архітектури можемо здійсните відокремлення бази даних та кешу в окремий сервер або сервери.

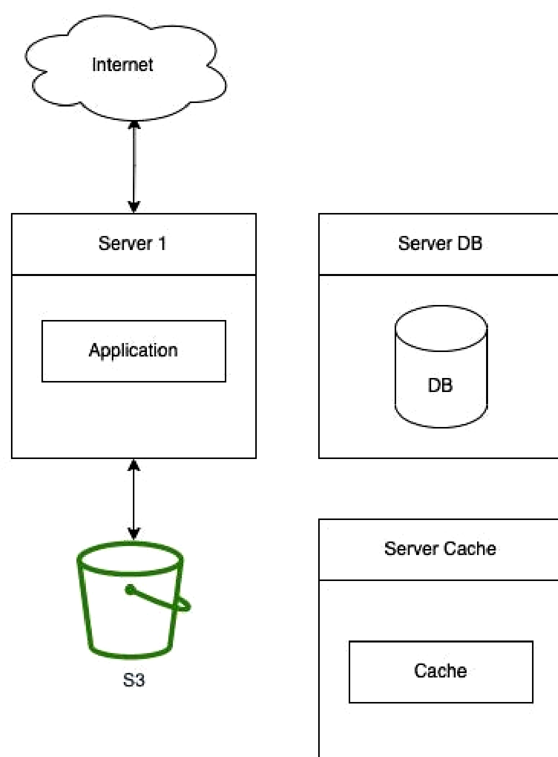


Рисунок 1.4 – Складові масштабування

Розділення компонентів дає можливість масштабувати їх окремо. Відносно масштабування є два підходи.

Вертикальне масштабування – це збільшення або зниження ресурсів доступних для використання програмним забезпеченням. По суті у випадку зростання потреб у ресурсах, це збільшення самого серверу за ресурсами.

Сервер може мати обмеження по використанню CPU, пам’яті чи дискового простору. Тому вертикальне масштабування має межі. Досягнувши межі доведеться купувати новий сервер,

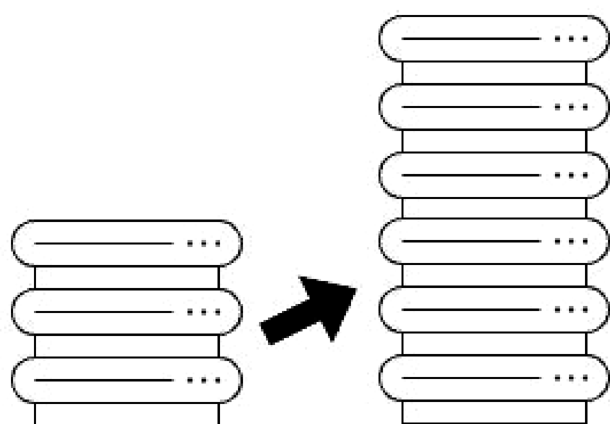


Рисунок 1.5 – Вертикальне масштабування

з більшою кількістю ресурсів і переносити ваше рішення на нього. Тому, рекомендують зазделегідь планувати який підхід до масштабування обирати.

Горизонтальне масштабування — це можливість додавати до системи нові вузли чи сервер, збільшуючи таким чином загальну кількість ресурсів.

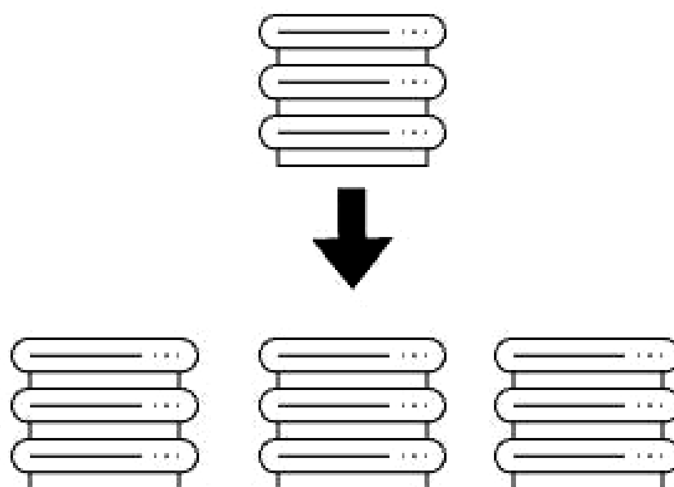


Рисунок 1.6. – Горизонтальне масштабування

Обрана вище схема дає нам гнучкість в можливості збільшувати як окремо кількість серверів додатку так і окремо серверів баз даних в залежності від наших потреб.

1.3. Аналіз монолітної та мікросервісної архітектури веб-серверів

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.

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.

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. 1.7) — це традиційна уніфікована модель розробки програмного забезпечення. Монолітний у цьому контексті означає складений з одного шматка [5]. 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.

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.

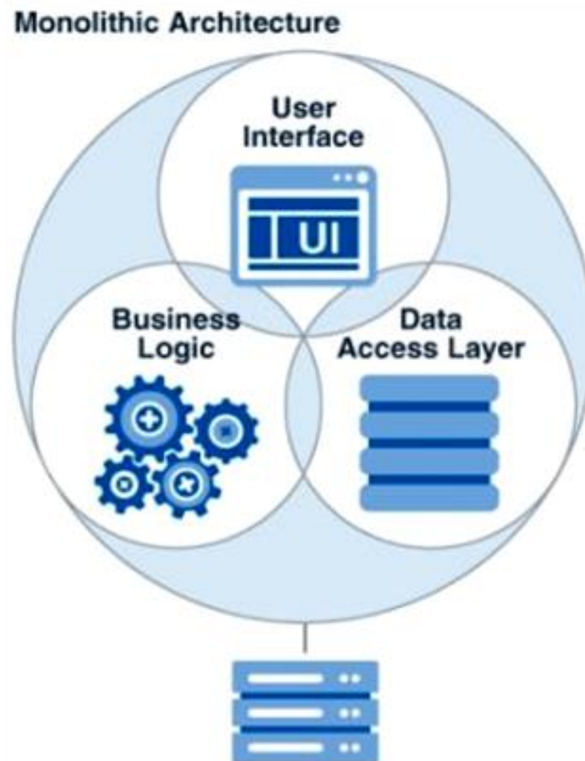


Рис. 1.7 – Монолітна архітектура веб-серверів

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. Дана область функціональності стосується лише поточного одного програмного застосунку, тому і легше працювати з цією системою.

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. 1.8). 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 [6]. Ключові концепції, які потрібно використовувати при розробці мікросервісної архітектури — це декомпозиція та розгрупування. Функціональність додатків має бути повністю незалежною одна від одної.

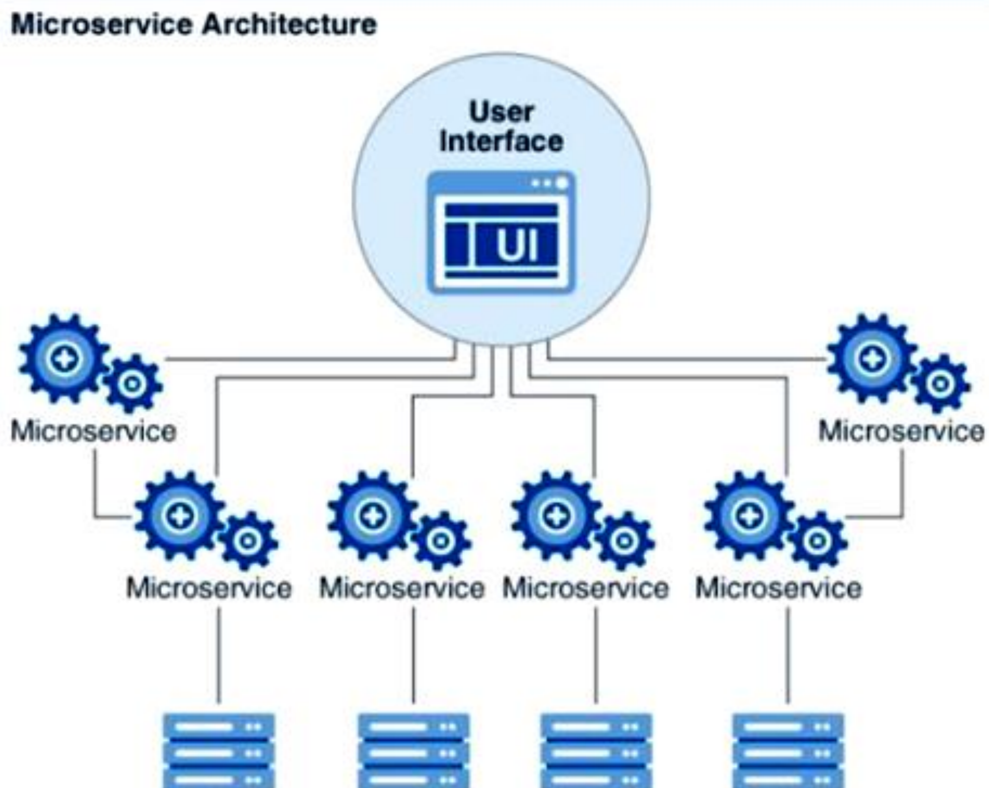


Рис. 1.8 – Мікросервісна архітектура

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, якщо потрібно мати вигоду від даної архітектури, та, щоб вона забезпечувала слабкий зв’язок.

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.

Для використання всередині окремих організацій, вони не дуже підходять, так як, можуть виявитися складними у використанні. Мають меншу безпеку порівняно з монолітними застосунками. Це по'вязано з тим, що зв’язок між сервісами відбувається через мережу.

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

РОЗДІЛ 2
ТЕХНОЛОГІЇ РОЗРОБКИ АРХІТЕКТУРИ ВЕБ-СЕРВЕРА

2.1. Особливості 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.

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.

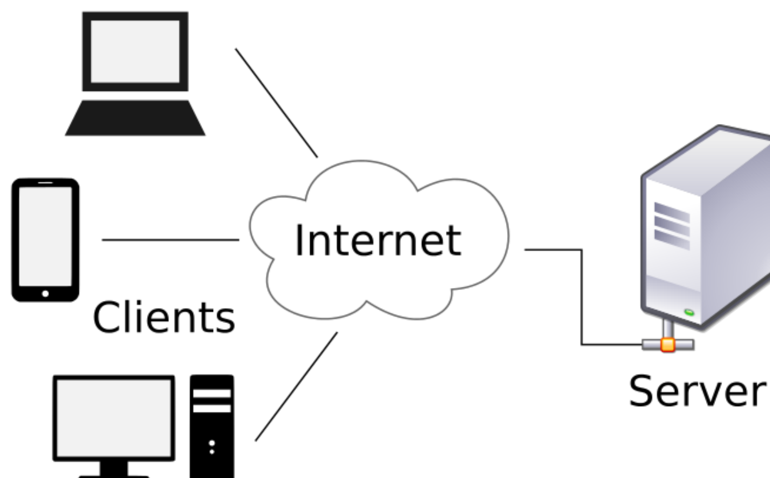


Рисунок. 2.1 – Клієнт-серверна архітектура

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. 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:

- 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.

2.2. Вибір технологій розробки архітектури веб-сервера

Для виконання завдань кваліфікаційної роботи використано набір різноманітних технологій. Зокрема, **Node.js** і **NestJS** для створення міцної основи додатку. Це забезпечує асинхронне програмування та модульну архітектуру, а також сприяє підвищенню продуктивності та організації коду.

Для ефективного зберігання великих обсягів даних, зокрема зображень, використано **Backblaze B2 Storage**, який надає високу швидкість доступу із низькими витратами.

В роботі імплементовано декілька стратегій аутентифікації через **PassportJS**, включаючи **JWT** для безпеки сесій, **Google** для соціальної аутентифікації, а також стратегії для роботи з **Access** та **Refresh Tokens**. Це забезпечуює комплексний підхід до безпеки та зручності користувачів.

Вибір **PostgreSQL** як системи управління базами даних забезпечує міцність та надійність в управлінні даними, підтримуючи складні запити та транзакції, що є критично важливим для стабільної роботи великих додатків.

Опишемо особливості та доцільність використання цих інструментів.

Node.js – програмна платформа, заснована на двигуні V8 (компілюючому JavaScript в машинний код), що перетворює JavaScript з вузькоспеціалізованої мови на мову загального призначення. Node.js додає можливість JavaScript взаємодіяти з пристроями вводу-виводу через свій API, написаний на C++, підключати інші зовнішні бібліотеки, написані різними мовами, забезпечуючи виклики до них з JavaScript-коду. Node.js застосовується переважно на сервері, виконуючи роль веб-сервера, але є можливість розробляти на Node.js і десктопні віконні програми (за допомогою NW.js , AppJS або Electron для Linux, Windows та

macOS) і навіть програмувати мікроконтролери (наприклад, tessel, low.js та Espruino). В основі Node.js лежить подієво-орієнтоване та асинхронне (або реактивне) програмування з неблокуючим введенням/виводом.

Зразок створення та запуску HTTP-сервера на Node.js, що видає Hello, world! має вигляд:

```
// Підключення модуля http
const http = require ( "http" );
// Створюємо веб-сервер з обробником запитів
const server = http . createServer (( incomingMessage , response ) => { console . log ( "Початок обробки запиту" ); // Передаємо код відповіді та заголовки http response . writeHead ( 200 , { "Content-Type" : "text/plain; charset=UTF -8" }); response . end ( "Hello, world!" ); });
// Запускаємо веб-сервер
server . listen ( 80 , "127.0.0.1" , () => { const info = server . address (); console . log ( `Сервер запущений ${ info } ` ); });
```

NestJS – програмна платформа для розробки швидких і масштабованих web-додатків, клієнтська і серверна частина яких написані на JavaScript. Node.js використовує двигун V8 від Google та перетворює ДжаваСкрипт код в бінарний, який може працювати не тільки в браузері, але також в додатках на ПК, або мобільних ОС.

На практиці, Node.js застосовують як web-сервер через його подієво-орієнтовану модель та неблокуючі введення / виведення архітектури.

Все це фундаментальні можливості, для використання яких у вашому проєкті, потрібні надбудови вищого рівня, такі як фреймворк Nest.JS. Завдання фреймворку – зробити розробку простіше, швидше і більш структурованою. Для Node.js є понад двоє десятків популярних фреймворків, заснованих, як і Nest.JS, на «стандартному» Express.JS. Головна перевага Nest.JS в порівнянні з базовою платформою – чітка архітектура, що складається з трьох компонентів:

контролерів, модулів і провайдерів. Завдяки їй додаток з самого початку розробки просто розділити на мікросервіси та працювати над кожним окремо, поєднуючи в єдину систему.

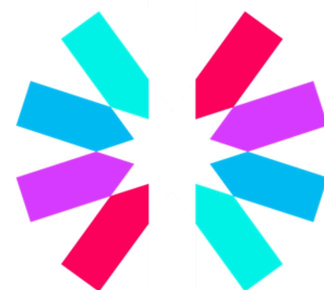
Backblaze B2 Storage – це послуга IaaS, яка пропонує необмежене хмарне сховище за доступними цінами.

PassportJS – це middleware для авторизації під node.js. Passport підтримує авторизацію за допомогою величезної кількості сервісів. Бібліотека Passport JS це сторонні ПЗ, що слугує для реалізації системи авторизації користувача на вашому веб-сайті. Passport JS це бібліотека що покликана працювати виключно з Node JS проектами. Для створення авторизації на сайті ви можете використовувати одну з багатьох доступних стратегій авторизації користувача. Переглянути всі стратегії можна на їхньому офіційному сайті.

Passport JS насамперед забезпечує захищене підключення користувача до вашого сайту. Тим самим авторизація відбувається коректно і ніхто, крім дійсно зареєстрованих користувачів, не зможе увійти до вас на сайт. Також Passport JS дозволяє створити захищені розділи сайту, тобто розділи, доступ до яких буде забезпечений лише після авторизації користувача на сайті:

```
app.get('/profile', passport.authenticationMiddleware(), renderProfile)
```

JWT – це стандарт токена доступу на основі JSON, стандартизованого в RFC 7519. Як правило, використовується для передачі даних для аутентифікації в клієнт-серверних програмах. Токени створюються сервером, підписуються секретним ключем і передаються клієнту, який надалі використовує цей токен для підтвердження своєї особи.



Токен JWT складається з трьох частин: заголовка (header), корисного навантаження (payload) та підпису або даних шифрування. Перші два елементи – це JSON об'єкти певної структури. Третій елемент обчислюється на основі

перших і залежить від обраного алгоритму (у разі використання непідписаного JWT може бути опущений).

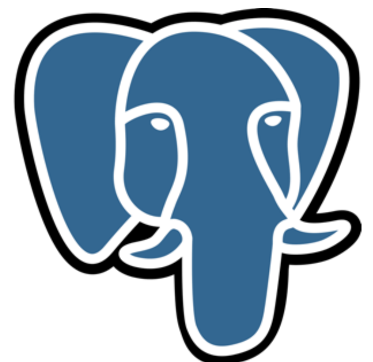
Access token – програмний об'єкт операційних систем класу Microsoft Windows, містить інформацію з безпеки сеансу та ідентифікує користувача, групу користувачів та користувальницькі привілеї. Маркер доступу – це об'єкт, що інкапсулює дескриптор безпеки процесу. Доданий до процесу дескриптор безпеки ідентифікує власника об'єкта. Поки маркер використовується для подання інформації з безпеки, він технічно вільний і може містити будь-які дані.

Refresh Tokens. Є багато різних токенів – звичайні, криптографічні, "access key", "session token", різні схеми отримання, використання та revoke. При цьому одна з ключових ідей полягає в тому, що якщо хтось поганий отримає чужий токен, то найнеприємніше, що трапиться – це доступ викрадача до сервісу, від якого токен викрадено. Пароль, той самий, який один на всі послуги, викрадач не отримає. А користувач, якщо зрозуміє, що сервісу отримав доступ хтось інший, може токен відкликати. Після чого отримати собі новий, маючи логін та пароль.

Схема використання у токенів наступна:

- Користувач логіниться в програмі, передаючи логін та пароль на сервер. Вони не зберігаються на пристрої, а сервер повертає два токена і час їхнього життя.
- Програма зберігає токени та використовує **access token** для наступних запитів
- Коли час життя **access token** добігає кінця (додаток може сам перевіряти час життя, або дочекатися поки під час чергового використання сервер відповість «ой, все»), програма використовує **refresh token** , щоб оновити обидва токена і продовжити використовувати новий **access token**

PostgreSQL – об'єктно-реляційна система керування базами даних (СКБД). Є альтернативою як комерційним СКБД (Oracle Database, Microsoft SQL Server, IBM DB2 та інші), так і СКБД з відкритим кодом (MySQL, Firebird, SQLite).



Порівняно з іншими проєктами з відкритим кодом, такими як Apache, FreeBSD або MySQL, PostgreSQL не контролюється якоюсь однією компанією, її розробка можлива завдяки співпраці багатьох людей та компаній, які хочуть використовувати цю СКБД та впроваджувати у неї найновіші досягнення.

Функції дозволяють виконувати деякий код безпосередньо сервером бази даних. Ці функції можуть бути написані на SQL, який має деякі примітивні програмні оператори, такі як галуження та цикли. Але гнучкішою буде функція написана на одній із мов програмування, з якими PostgreSQL може працювати.

2.3. Архітектура веб-сервера

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.

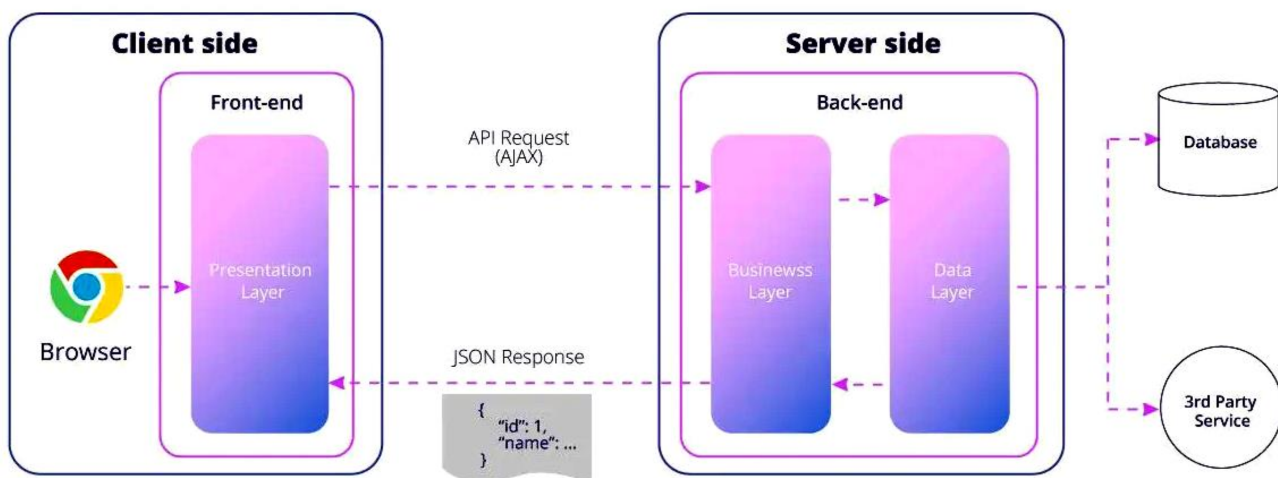


Рисунок. 2.2 – Зв'язок між клієнтською і серверною частинами веб-сервісу

Переваги цієї архітектури такі:

- **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.

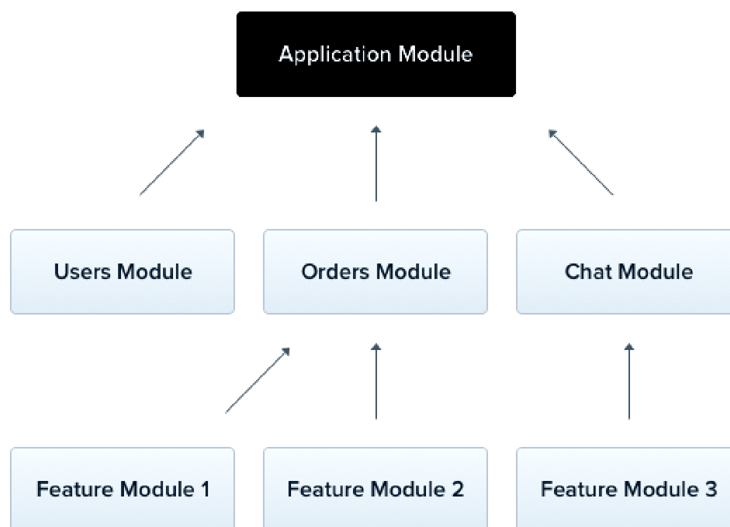


Рисунок. 2.3 – Структура модулів веб-сервісу

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. 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. Безпека та аутентифікація

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:

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 та 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.

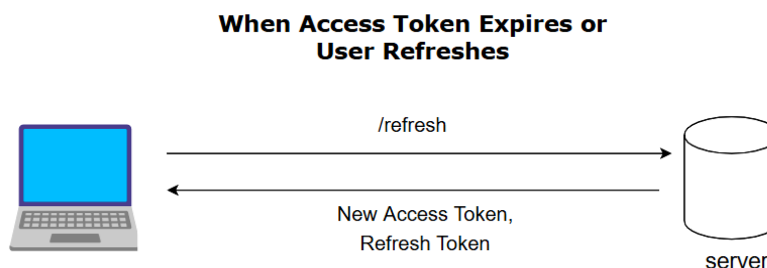


Рисунок. 2.4 – Взаємодія та оновлення точена для клієнта веб-серверу

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:

```
google.strategy.ts    accessToken.strategy.ts    local.strategy.ts ×
src > modules > auth > strategies > local.strategy.ts > ...
You, 6 days ago | 1 author (You)
1  import { Injectable, UnauthorizedException } from '@nestjs/common';     You, 6 days ago
2  import { PassportStrategy } from '@nestjs/passport';
3  import { Strategy } from 'passport-local';
4  import { LOCAL_AUTH } from '../auth.constants';
5  import { AuthService } from '../auth.service';
6
You, 6 days ago | 1 author (You)
7  @Injectable()
8  export class LocalStrategy extends PassportStrategy(Strategy, LOCAL_AUTH) {
9    constructor(private authService: AuthService) {
10     super({
11       usernameField: 'email',
12     });
13   }
14
15   async validate(email: string, password: string): Promise<any> {
16     const user = await this.authService.validateUserWithPassword(email, password);
17
18     if (!user) {
19       throw new UnauthorizedException();
20     }
21
22     return user;
23   }
24 }
25
```

```
async validateUserWithPassword(email: string, password: string) {
  const user = await this.userRepository.findOneAndThrowIfNotExist({ where: { email } });
  const isPasswordMatching = await argon2.verify(user.passwordHash, password);
  if (!isPasswordMatching) throw new BadRequestException('Wrong credentials provided');

  return this.userRepository.exclude(user, ['passwordHash']);
}
```

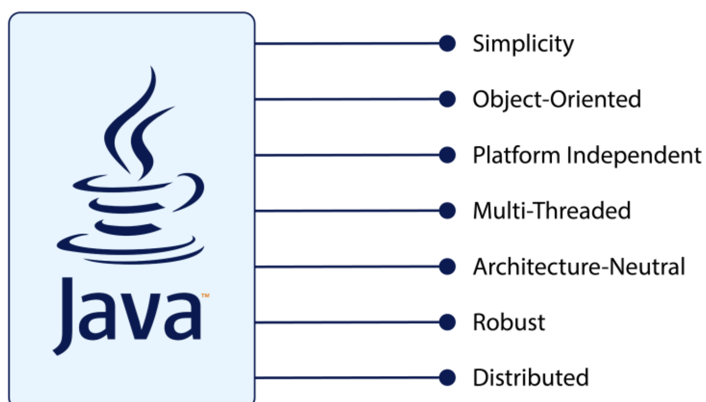
Залежно від міри довірчих стосунків, структури, особливостей мережі і віддаленості об'єкта перевірка може бути односторонньою або взаємною. Також розрізняють однофакторну і строгу (двофакторну) автентифікації. В однофакторних системах, найпоширенішими в цей час є парольні системи автентифікації.

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 ПОБУДОВИ АРХІТЕКТУРИ ВЕБ-СЕРВЕРА

3.1. Мови програмування та інструменти розробки

За останні декілька років професія програміста стала однією з найбільш затребуваних та популярних в Україні. Кількість IT компаній, яким потрібні нові розробники зростає з кожним днем. Це призводить до великого попиту на IT спеціалістів та росту заробітної плати в даній сфері []. Освітній центр “YOLO” пропонує перелік 5-ти найпопулярніших мов програмування, на які потрібно звернути увагу.



Надзвичайно поширена мова, адже 90% компаній, що входять в список Fortune (500 найбільших корпорацій світу), так чи інакше використовують в своїх розробках Java. До речі її використовують при розробці операційної системи Android, яка на даний час є найбільш мобільною платформою в світі. Мова була розроблена ще в 1995 році, компанією Oracle і досі залишається на вершині.ЇЇ використовують для розробки десктопних додатків, операційних систем, “back end” систем та багато іншого. Основна її перевага це кросплатформеність.

Хоч і назва цієї мови схожа з попередньою мовою Javascript, це зовсім інша мова програмування. Згідно з сайтом Stackoverflow, Javascript – це найпопулярніша мова програмування серед розробників. Ця мова використовується як одна з основних технологій для створення інтерактивних сайтів разом з HTML та CSS. Адже більшість браузерів використовують саме ці три основні технології. Також, використовуючи її, можна створювати мобільні додатки, ігри та десктопні

програми. Тому, якщо ви плануєте розвиватись саме в цих сферах, варто спробувати починати саме з цієї мови програмування [24].

HTML та CSS – це не зовсім мови програмування, і досвідчені програмісти вважають, якщо людина знає лише HTML та CSS, вона не може називати себе програмістом. Проте, ці технології використовуються практично на кожному сайті. Адже саме через них прописується візуальний стиль сайтів, кнопки, іконки та ефекти кожної сторінки. Тому ці мови корисно знати не лише програмістам, але і веб дизайнерам. Тому якщо ви хочете почати свій розвиток в веб розробці чи дизайні, тоді вам в першу чергу варто звернути увагу саме на ці технології.

Популярність цієї мови програмування пов’язана з тим, що її використовують в найпоширенішій платформі для створення сайтів Wordpress, а 80% найбільш відвідуваних сайтів в світі використовують мову PHP тим чи іншим способом. Вона вважається однією з базових та найпростіших мов програмування, яку має знати кожен, хто називає себе програмістом. Не важливо, чи створюєте ви сайти, складні інтернет магазини чи серверні рішення, вам точно стане в нагоді PHP [11].

Мова, яку розробили ще у 1983 році і на якій створені Microsoft Windows і Google Chrome. Завдяки широкому набору інструментів мова легко адаптується для застосування в різноманітних сферах життя, будь то банківська сфера, розробка ігор, торгівля чи інше. Тому саме на цій мові можна створювати складні комерційні системи з багатьма елементами, так і розробляти прості застосунки та програми. І якщо ви шукаєте мову з широким інструментарієм та

функцією, тоді C++ саме для вас.

Об'єктно-орієнтована мова програмування з безпечною системою типізації для платформи .NET. Розроблена Андерсом Гейлсбергом, Скотом Вілтамутом та Пітером Гольде під егідою Microsoft Research (належить Microsoft).

Синтаксис C# близький до C++ і Java. Мова має строгу статичну типізацію, підтримує поліморфізм, перевантаження операторів, вказівники на функції-члени класів, атрибути, події, властивості, винятки, коментарі у форматі XML. Перейнявши багато від своїх попередників — мов C++, Object Pascal, і Smalltalk — C#, спираючись на практику їхнього використання, виключає деякі моделі, що зарекомендували себе як проблематичні при розробці програмних систем, наприклад, мова C#, на відміну від C++, не передбачає множинне успадкування класів.

Для побудови програми у ***Microsoft Visual Studio*** широко використовують засоби, які надає система. Є дві частини побудови: перша – проектування інтерфейсу з використанням стандартних елементів (компонент) та маніпулювання їхніми розмірами й розташуванням; друга – написання фрагментів програмного коду для виконання завдання.

Багато варіантів постачання також включають – Microsoft SQL Server або MSDE Visual Source Safe – файл-серверна система управління версіями. У минулому, до складу Visual Studio також входили продукти: 1) Visual InterDev; 2) Visual J++; 3) Visual J#; 4) Visual FoxPro.

Для початку, треба підготувати папку (каталог), де будуть зберігатися всі файли майбутньої програми. Це ліпше робити до запуску Visual Studio. Нову папку створити за звичайними правилами операційної системи Windows. Наприклад, запустити інструмент Мій комп’ютер (MyComputer), розташований на робочому столі системи, відшукати потрібний диск та вже наявну папку, відкрити її, після чого через меню вибрати команди Файл→Створити→Папка.

3.2. Інтеграція із базою даних комплексних застосунків

Як уже зазначалося, 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.

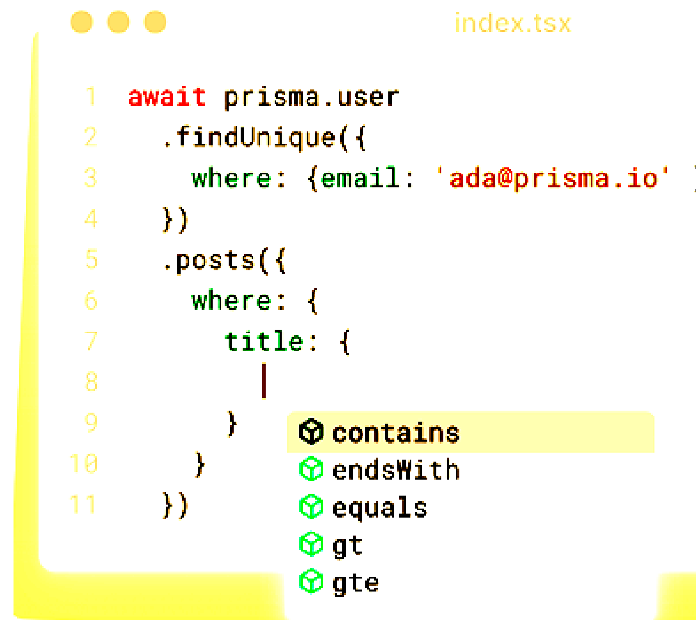
schema.prisma

```
1  datasource db {
2    provider = "postgresql"
3    url      = env("DATABASE_URL")
4  }
5
6  model Post {
7    id        Int @id @default(autoincrement(
8    title     String
9    content   String?
10   published Boolean @default(false)
11   author    User?   @relation(fields: [auth
12   authorId  Int?
13 }
14
15 model User {
16   id    Int @id @default(autoincrement())
17   email String @unique
```

Процес інтеграції з базою даних включає наступні кроки:

- **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.



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.

3.3. Схема взаємодії із 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.

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. 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).

Рисунок. 3.1. – Технологія LAMP

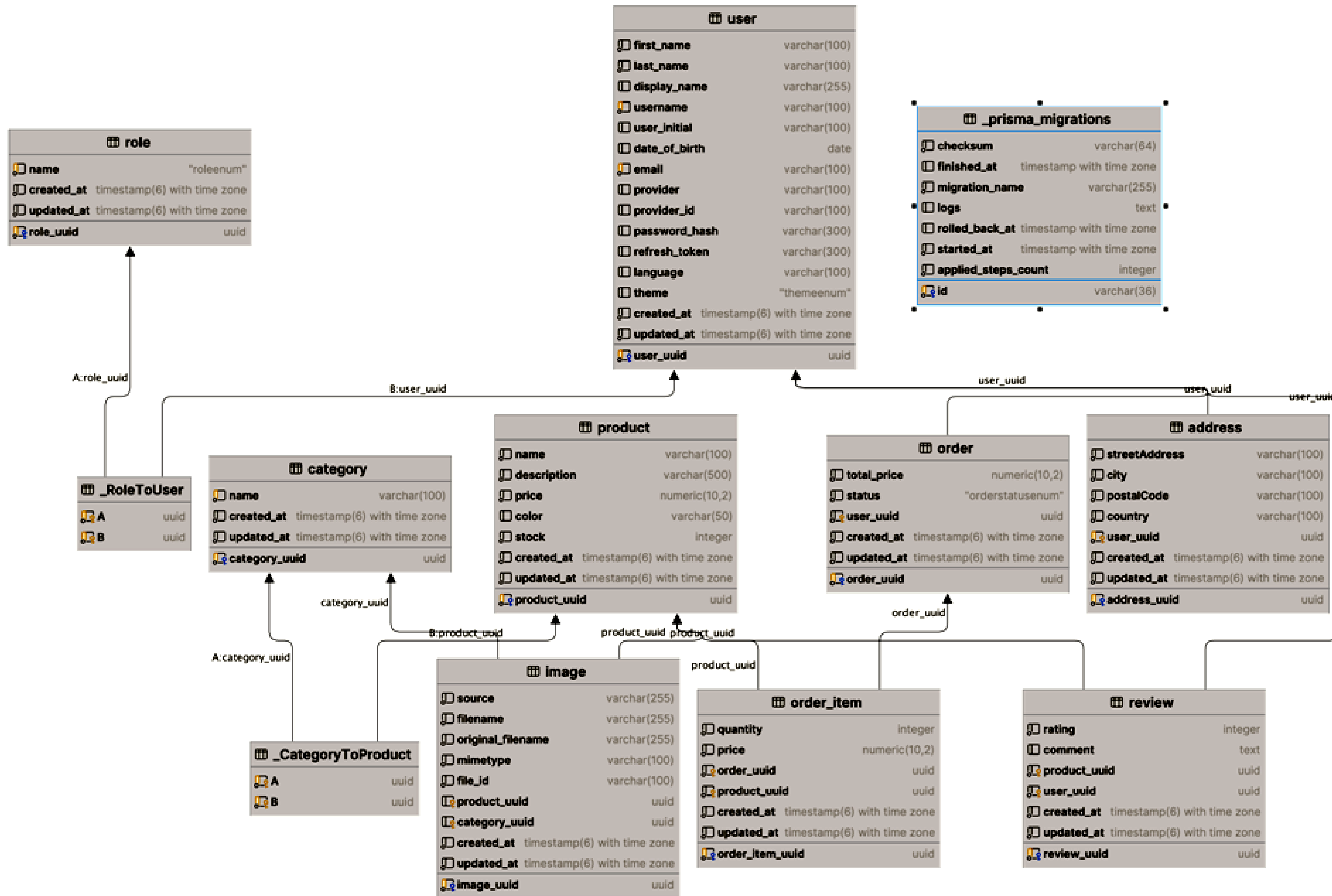


Рисунок 3.2 – Схема взаємодії із базою даних

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

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".

3.4. 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.

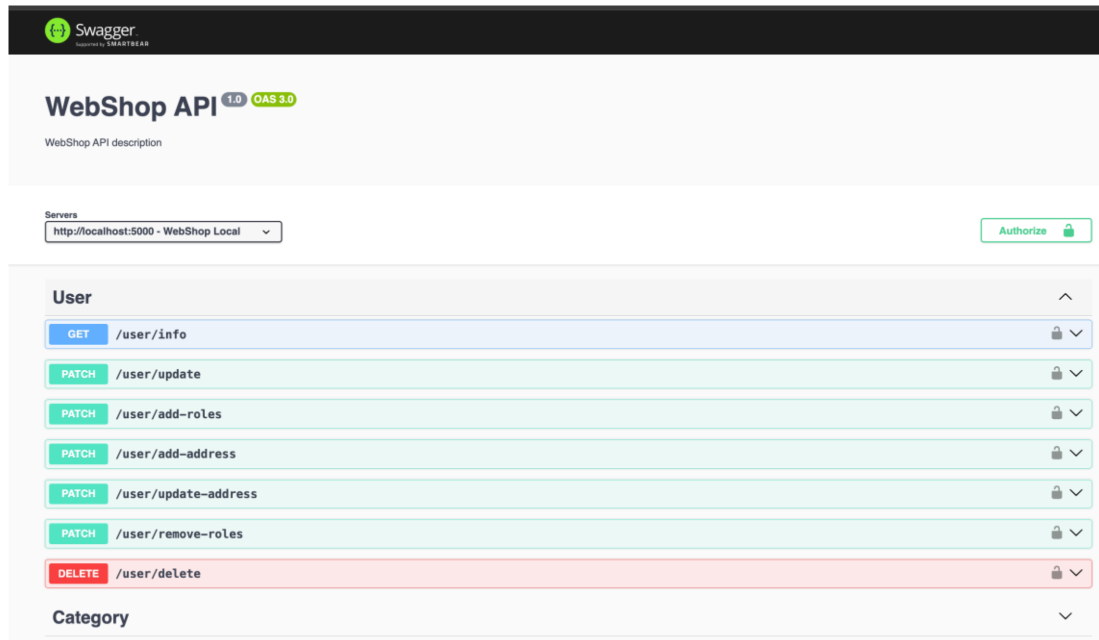


Рисунок 3.3 – Swagger із описом Web Shop SPI

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.
ПРОГРАМНА РЕАЛІЗАЦІЯ ВЕБ-СЕРВЕРА

4.1. Результати розробки бази даних

Розробка бази даних починається з визначення полів для всіх сутностей системи. Створено 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.

А також декілька системних таблиць які потрібні для визначення звязків та міграцій для Prisma ORM такі як: **_prisma_migrations**, **_RoleToUser**, **_CategoryToProduct.**

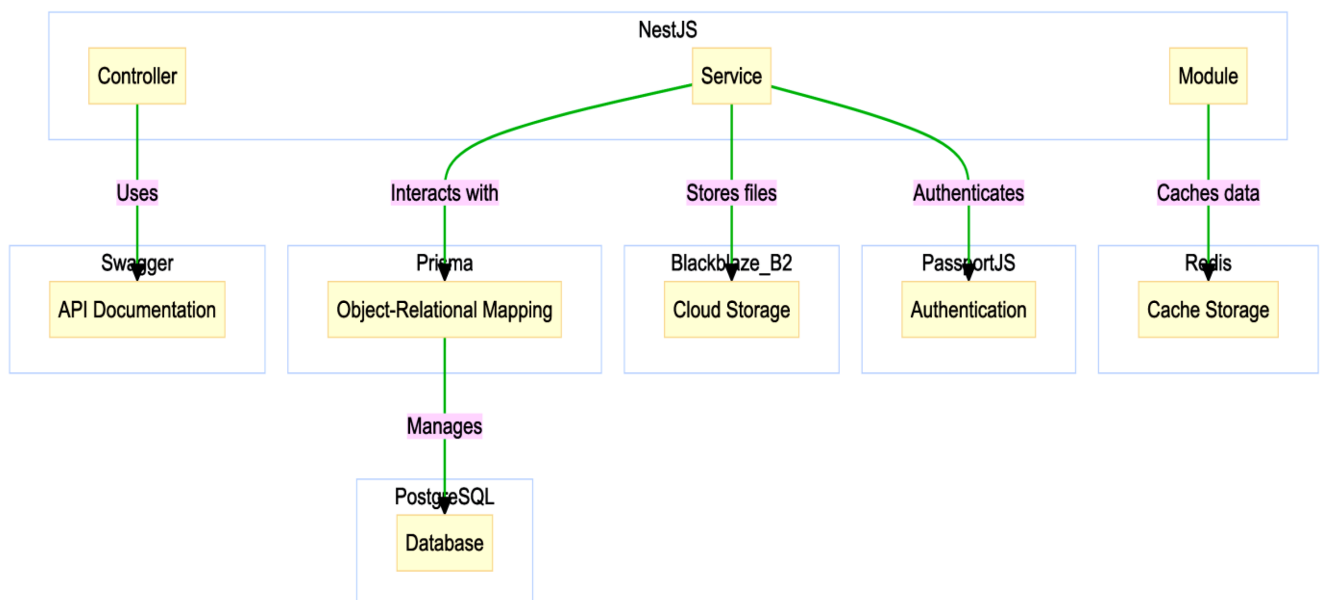


Рисунок 4.1 – Діаграми програмної реалізації веб-сервера

Як помічник роботи з базою даних, використовується Prisma ORM, який має багато функцій для роботи з базою даних які 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)
1   /*
2   https://www.prisma.io/docs/concepts/components/prisma-client/crud
3   https://www.prisma.io/docs/concepts/components/prisma-client/relation-queries
4   */
5
6   import { NotFoundException } from '@nestjs/common';
7   import { Prisma } from '@prisma/client';
8   import { NOT_EXIST_MSG, NOT_EXIST_TYPE } from '../constants/server.constants';
9   import { ErrorsHandleService } from '../services/error.service';
10  import { PrismaService } from '../services/prisma.service';
11  import { ErrorType } from '../types';
12
    You, 2 weeks ago | 1 author (You)
13  export abstract class BaseRepository<MODEL, FU, FF, FM, C, CM, U, UM, UPS, D, DM, CC> {
14    private readonly model: string;
15    private readonly ORM: PrismaService;
16    private readonly errorsHandleService: ErrorsHandleService;
17
18    protected constructor(model: string, database: PrismaService, errorsHandleService: ErrorsHandleService) {
19      this.model = model;
20      this.ORM = database;
21      this.errorsHandleService = errorsHandleService;
22    }
23
24    async queryRaw(data: string, optionalErrorMessage?: string): Promise<MODEL[]> {…
32    }
33
34    async findOneAndThrowIfNotExist(data: FF, optionalErrorMessage?: string): Promise<NonNullable<MODEL>> {…
56    }
57
58    async findOneWithoutChecking(data: FF, optionalErrorMessage?: string): Promise<MODEL> {
59      try {
60        // @ts-ignore
61        return await this.ORM[this.model].findFirst(data);
62      } catch (e) {
63        const err = e as ErrorType;
64        throw this.errorsHandleService.throwErrorIfServerError(
65          err,
66          'findOneWithoutChecking',
67          this.model,
68          optionalErrorMessage
69        );
70      }
71    }      You, 2 weeks ago • first commit
72
```

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 прецедентів 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.

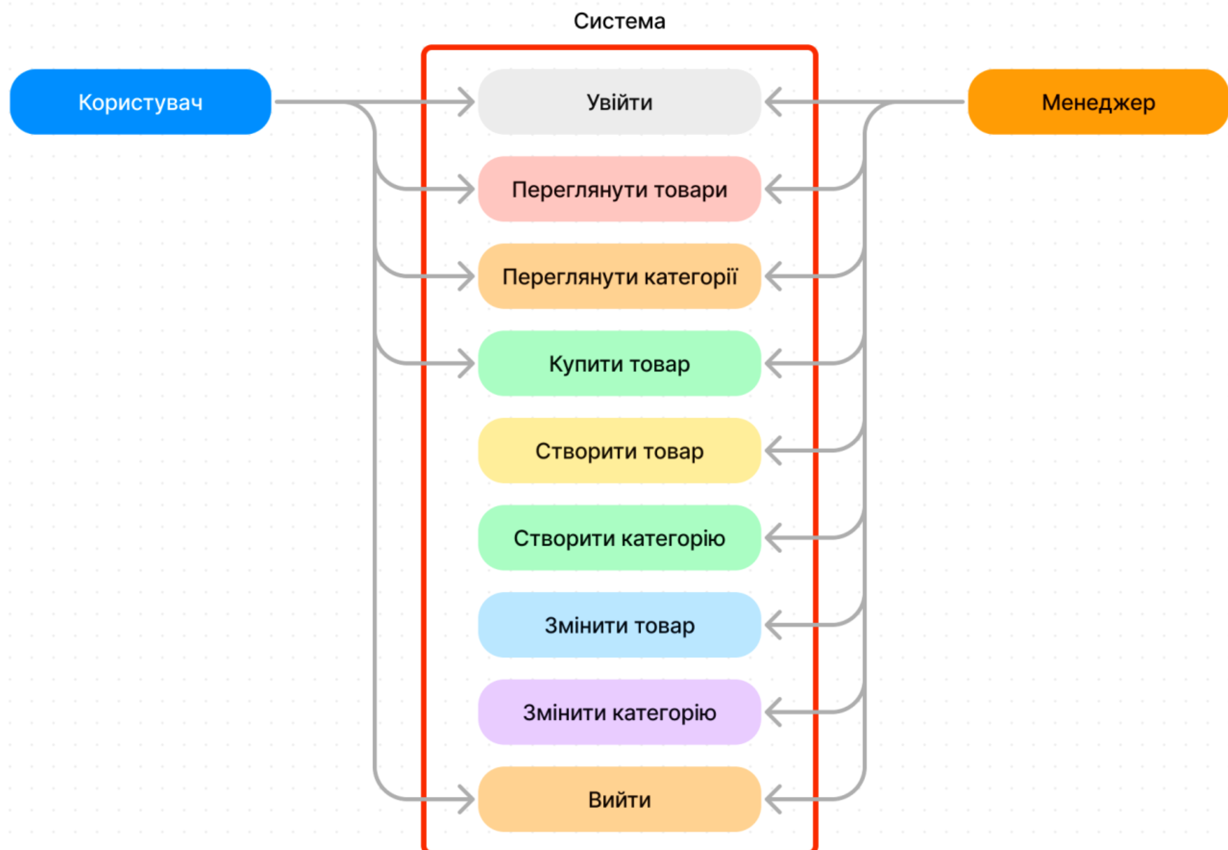


Рисунок. 4.2 – Діаграма прецедентів.

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.

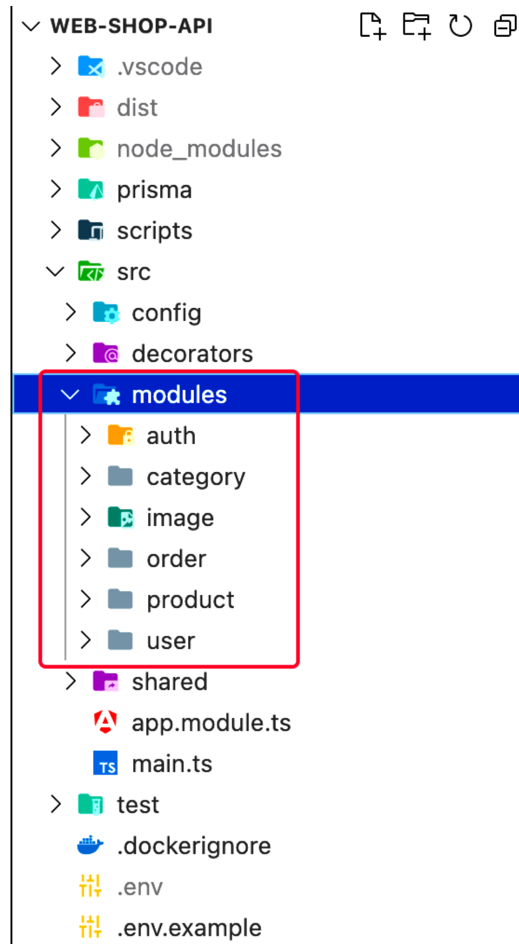


Рисунок. 4.3 – Каталог проекту

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.

.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.

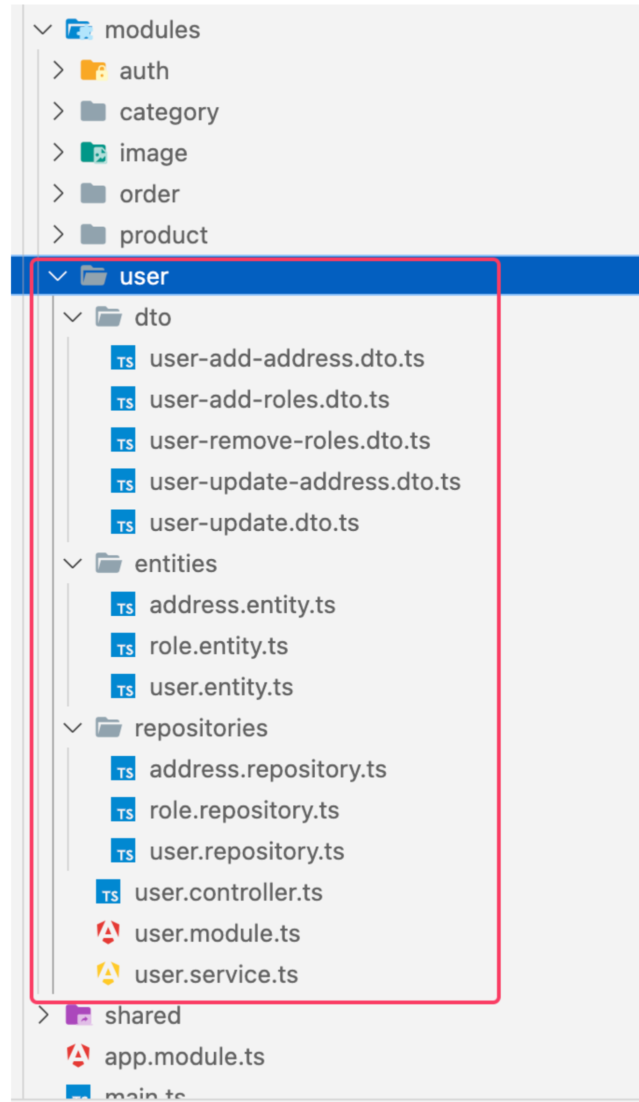


Рисунок 3.5 – Структура модуля **user**

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

Спочатку 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. Інструкція з використання

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. 3.6 – Форма реєстрації користувача

Вводимо у форму поштову адресу і бажаний пароль для користувача та інші дані повязані із користувачем після цього натискаємо на кнопку Execute. Далі Swagger формує та виконує запит на сервер.

Curl

```
curl -X 'POST' \
  'http://localhost:5000/auth/register' \
  -H 'accept: */*' \
  -H 'Content-Type: application/json' \
  -d '{
  "firstName": "Misha",
  "lastName": "Tretyak",
  "language": "uk",
  "dateOfBirth": "2000-11-19",
  "email": "misha.tret.com@gmail.com",
  "password": "super_password"
}'
```

Request URL

```
http://localhost:5000/auth/register
```

Рисунок. 3.7 – 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.

201

Response body


```
{
  "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJkMDFjYmUyYS1iNzFkLTQ0ZTItYWI5Ni00MGUwMTY1NzYzZWQiLCJlbWFpbCI6Im1pc2hhLnRyZXQuY29tQGdtYWlsLmNvbSIsImlhdCI6MTcwNTg2NzA0MCwiZXhwIjoxNzA1ODY3OTQwfQ.VokogFtLO8PnEwu85fU-oMB2VtwMiBQMfesWB7xrDv4",
  "refreshToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiJkMDFjYmUyYS1iNzFkLTQ0ZTItYWI5Ni00MGUwMTY1NzYzZWQiLCJlbWFpbCI6Im1pc2hhLnRyZXQuY29tQGdtYWlsLmNvbSIsImlhdCI6MTcwNTg2NzA0MCwiZXhwIjoxNzA2NDcxODQwfQ.iLAsmzd0s9Javj_LMUkp8rbSRD-eU_feEtz8SOZ6iI4"
}
```


Download

Рисунок. 3.8 – 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.

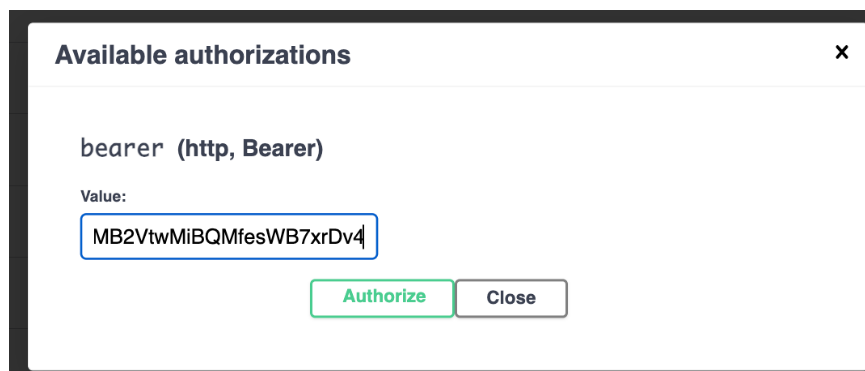


Рисунок. 3.9 – Модальне вікно авторизації

Тепер можна використовувати точки які потребують авторизації (вони позначаються замком). Створимо новий продукт.

POST /product/create

Parameters | Cancel | Reset

No parameters

Request body required — multipart/form-data

files * required (array): Choose file SCR-20240121-tblb.png — Add string item
name * required (string): Test Product
description * required (string): My super description
price * required (number): 128.5
stock * required (number): 35
color (string): green — Send empty value
categoryUuids * required (array): 2a701dca-75e3-483a-b146-b4417e67ac81 — Add string item

Execute | Clear

Рисунок. 3.10 – Форма створення задачі

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

201

Response body


```
{
  "productUuid": "72288802-63c4-4f82-93a1-c611bb69bae2",
  "name": "Test Product",
  "description": "My super description",
  "price": "128.5",
  "color": "green",
  "stock": 35,
  "createdAt": "2024-01-21T20:02:45.586Z",
  "updatedAt": "2024-01-21T20:02:45.586Z",
  "images": [
    {
      "imageUuid": "462f8a4b-6b29-4bb8-a68d-f3361fb2830b",
      "source": "https://f005.backblazeb2.com/file/web-shop-api/87992abf-609d-470b-8cd1-ed9498abf384-1705867364569.png",
      "filename": "87992abf-609d-470b-8cd1-ed9498abf384-1705867364569.png",
      "originalFilename": "SCR-20240121-tblb.png",
      "mimetype": ".png",
      "fileId": "4_z2c9ba2ef17b2d2f48fc30415_f1115b3bb3cd5e42b_d20240121_m200245_c005_v0501003_t0018_u01705867365710",
      "productUuid": "72288802-63c4-4f82-93a1-c611bb69bae2",
      "categoryUuid": null,
      "createdAt": "2024-01-21T20:02:45.586Z",
      "updatedAt": "2024-01-21T20:02:45.586Z"
    }
  ],
  "categories": [
```


Рисунок. 3.11 – Відповідь на створення продукту

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

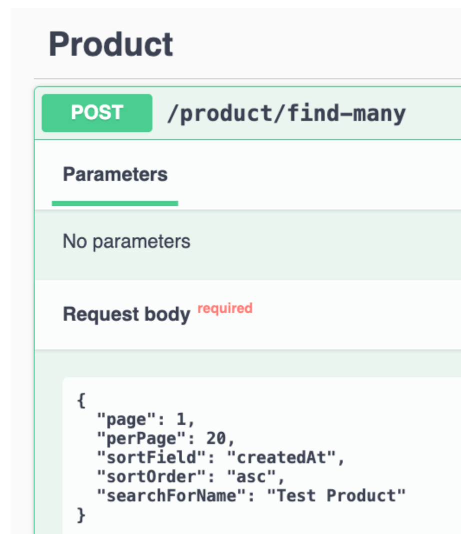


Рисунок. 3.12 – Форма пошуку продуктів

Натискаємо Execute та перевіряємо відповідь від сервера.

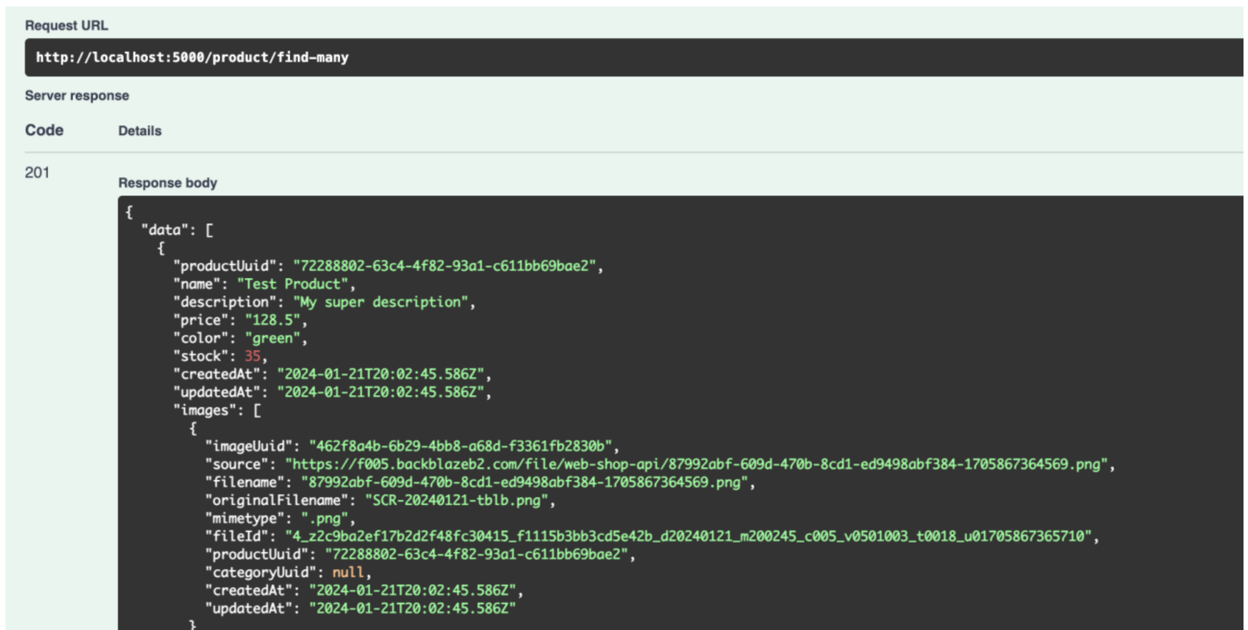


Рисунок. 3.13 – Відповідь на пошук продуктів

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

PATCH /product/update

Parameters

No parameters

Request body required

files array — Add string item — Send empty value

name string — name — Send empty value

description string — description — Send empty value

price number — 305.55 — Send empty value

stock number — stock — Send empty value

Рисунок. 3.14 – Форма редагування продукту.

Натискаємо Execute та перевіряємо відповідь від сервера.

Request URL

http://localhost:5000/product/update

Server response

Code Details

200

Response body


```
{
  "productUuid": "085878d8-1ce5-4082-bbf9-042daefb001f",
  "name": "Test Product",
  "description": "My super description",
  "price": "305.55",
  "color": "dark",
  "stock": 35,
  "createdAt": "2024-01-21T20:10:24.088Z",
  "updatedAt": "2024-01-21T20:11:40.563Z",
  "images": [
    {
      "imageUuid": "b3cc4859-494f-4a35-9615-2704428d508d",
      "source": "https://f005.backblazeb2.com/file/web-shop-api/e804dbd0-d101-4a7b-abdb-7dc6518cb8f0-1705867821954.png",
      "filename": "e804dbd0-d101-4a7b-abdb-7dc6518cb8f0-1705867821954.png",
      "originalFilename": "SCR-20240121-tblb.png",
      "mimetype": ".png",
      "fileId": "4_z2c9ba2ef17b2d2f48fc30415_f1131034f5e146be8_d20240121_m201023_c005_v0501004_t0036_u01705867823210",
      "productUuid": "085878d8-1ce5-4082-bbf9-042daefb001f",
      "categoryUuid": null,
      "createdAt": "2024-01-21T20:10:24.088Z",
      "updatedAt": "2024-01-21T20:10:24.088Z"
    }
  ],
  "categories": [
```


Рисунок. 3.15 – Відповідь на редагування продукту

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

DELETE /product/delete

Parameters Cancel

Name Description

productUuid * required string (query) The UUID of the product to delete

72288802-63c4-4f82-93a1-c611bb69bae2

Execute

Рисунок. 3.16 – Форма видалення продукту

Натискаємо кнопку Execute та перевіряємо відповідь.

Request URL

http://localhost:5000/product/delete?productUuid=72288802-63c4-4f82-93a1-c611bb69bae2

Server response

Code Details

200 Response body


```
{
  "msg": "Product was deleted successfully",
  "deletedEntity": {
    "productUuid": "72288802-63c4-4f82-93a1-c611bb69bae2",
    "name": "Test Product",
    "description": "My super description",
    "price": "128.5",
    "color": "green",
    "stock": 35,
    "createdAt": "2024-01-21T20:02:45.586Z",
    "updatedAt": "2024-01-21T20:02:45.586Z"
  }
}
```


Рисунок. 3.17 – Відповідь на видалення продукту

Тепер можна вийти з системи натиснувши кнопку Authorize яку натискали для того щоб увійти до системи.

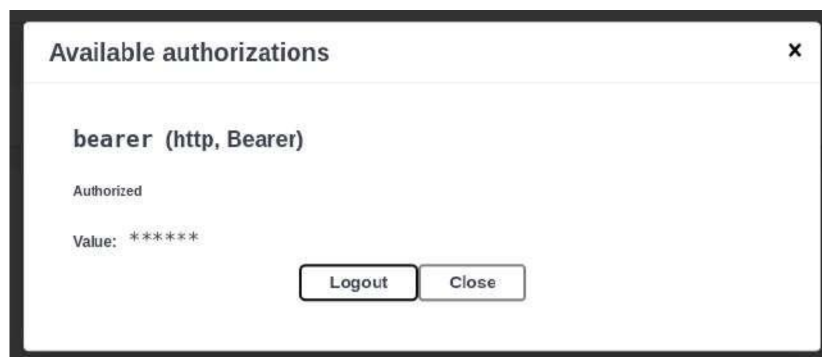


Рисунок. 3.18 – Модальне вікно в авторизованому стані

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

4.4. Порівняльна оцінка використання кеш сервісу для веб-серверу

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.

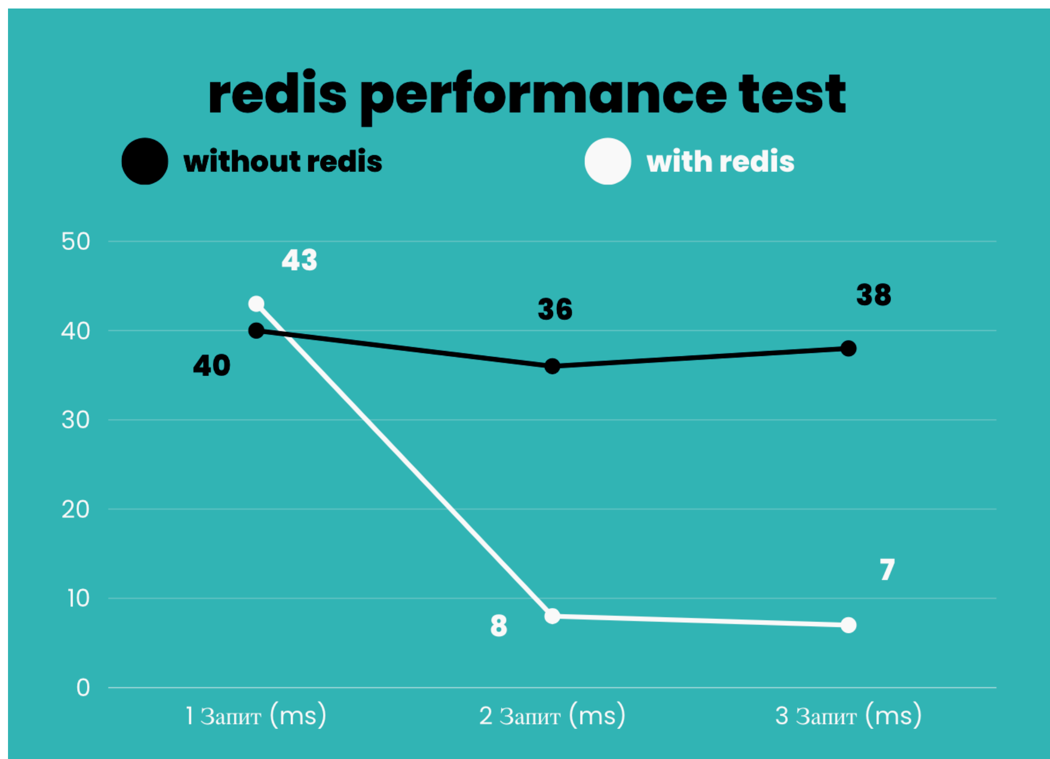


Рисунок. 3.19 – Оцінка використання Redis з NestJS

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

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.

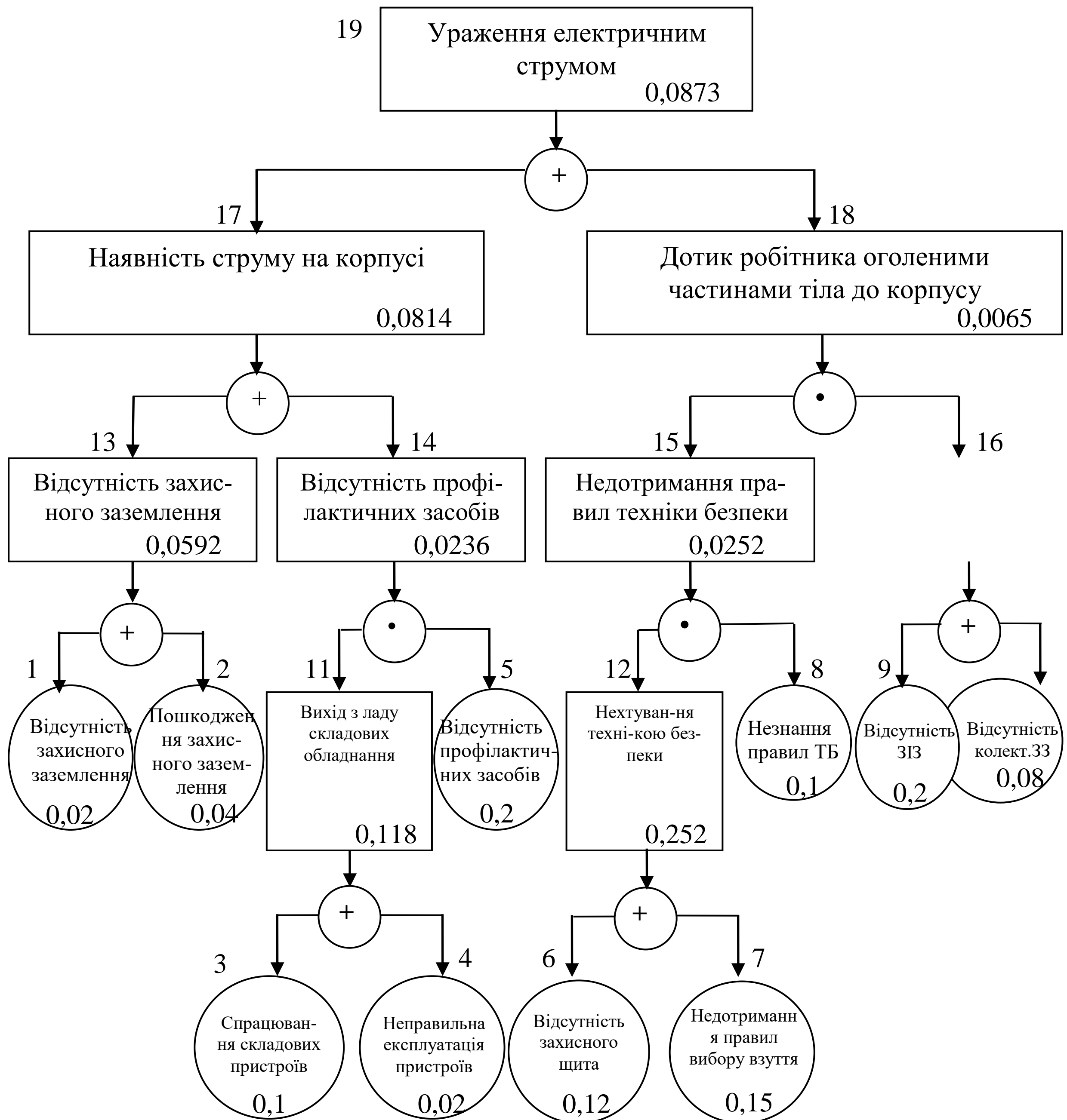


Рис. 5.1. Матриця логічних взаємозв’язків між окремими подіями травмонебезпечної ситуації [5]

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.

ВИСНОВКИ І РЕКОМЕНДАЦІЇ

1. 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.

2. Веб-сервер, розроблений на основі Node.js, надає можливість асинхронної обробки запитів, що сприяє високій продуктивності та знижує час відгуку. З використанням NestJS Framework отримали структурований підхід до розробки, де кожен аспект додатку поділяється на окремі модулі, що полегшує розширення та підтримку розробки.

3. Інтеграція з PostgreSQL як системою управління реляційними базами даних забезпечує надійне зберігання та управління великими обсягами даних, підтримуючи складні запити та транзакції.

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

5. Важливою частиною системи є застосування Passport.js для аутентифікації користувачів. Я використовую 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.

6. 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. 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.

7. Backblaze B2 Storage - це хмарне сховище даних, яке надає рішення для зберігання великих обсягів даних з високою продуктивністю та низькими витратами. У цій розробці, Backblaze B2 використовується для ефективного зберігання та управління медіафайлами, зокрема зображеннями, які є частиною веб-сервера. Крім зберігання зображень та медіафайлів, Backblaze B2 також ефективний для зберігання резервних копій, архівних даних, та інших великих наборів даних, що потребують довготривалого зберігання.

8. Загалом, 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: 27.12.2023).
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: 20.12.2023).
4. Комплексна система IT-рішень для управління агробізнесом. URL: https://agrichain.com.ua/ (дата звернення: 20.12.2023).
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. Макфарланд, Д. CSS: Практичний посібник. O'Reilly Media, 2017. 840с.
8. Модель «Клієнт-Сервер» URL: http://window.edu.ua/ (дата звернення: 27.12.2023).
9. Монолітна архітектура. URL: https://www.techtarget.com/whatis/definition/monolithic-architecture (дата звернення: 10.12.2023).
10. Новіков Ф.В., Богатов О.І., Навчальний посібник «Безпека життєдіяльності», Харків 2018, 250 с.
11. Пасічник В.В. Глобальні інформаційні системи та технології: моделі ефективного аналізу, опрацювання та захисту даних. Монографія / В.В. Пасічник, П. І. Жежнич, Р. Б. Кравець, А. М. Пелещишин, Д. О. Тарасов – Львів: Видавництво Львівської політехніки, 2006. – 348 с. ISBN: 966-553-578-1
12. Переваги та недоліки використання хмарних технологій

підприємствами України. URL: http://www.bsfa.edu.ua/files/konf2013/62.pdf (дата звернення: 20.12.2023).

13. 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.

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

15. Що таке мережевий порт? URL: https://2ip.ua/ua/blog/network-port Дата доступпу: травень 2020 (дата звернення: 20.12.2023).

16. API Gateway. URL: https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do (дата звернення: 04.12.2023).

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

18. Desktop vs Browser not just another Thick vs Thin Client debate. URL: https://www.msafocus.com/news/desktop-vs-browser. (дата звернення: 20.12.2023).

19. Docker Compose. URL: https://docs.docker.com/compose/ (дата звернення: 08.12.2023).

20. Docker. URL: https://docs.docker.com/get-started/overview/ (дата звернення: 03.12.2023).

21. Gillis A. REST API (RESTful API) / Alexander Gillis // TechTarget. 2020. URL: https://searchapparchitecture.techtarget.com/definition/RESTful-API (дата звернення: 27.12.2023).

22. Introduction to the server side // MDN. URL: https://developer.mozilla.org/en-US/docs/Learn/Serverside/First_steps/Introduction (дата звернення: 27.12.2023).

23. Kubernates Concepts. URL: https://kubernetes.io/docs/concepts/overview/components/ (дата звернення: 02.12.2023).

24. 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: 20.12.2023).

25. Martin Fowler — Microservices. URL: http://martinfowler.com/articles/microservices.html (дата звернення: 26.12.2023).

26. 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: 27.12.2023)

27. Pattern: Microservice Architecture. URL: https://microservices.io/patterns/microservices.html (дата звернення: 27.12.2023).

28. 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: 27.12.2023).

29. Tilkov S. "How small should your microservice be?". innoq.com. Retrieved 4 January 2017.

30. What is a Thin Client. URL: https://www.clearcube.com/posts/what-is-a-thin (дата звернення: 27.12.2023).

31. What is an API? // MuleSoft. URL: https://www.mulesoft.com/resources/api/what-is-an-api (дата звернення: 27.12.2023).

32. What is Docker Compose: Example, Benefits and Basic Commands. 2021. URL: https://www.simplilearn.com/tutorials/docker-tutorial/docker-compose (дата звернення: 27.12.2023).

33. What Is MongoDB? URL: https://www.mongodb.com/what-is-mongodb (дата звернення: 27.12.2023)

34. What Is Software Architecture? // Cast Software. URL: https://www.castsoftware.com/glossary/what-issoftware-architecture-tools-design-definition-explanation-best (дата звернення: 27.12.2023).

ДОДАТКИ

Додаток А.

Фрагмент коду Main.ts – файл з якого починається робота додатку

```
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;
  }
}
```