ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Веб-сайт з інтеграцією PostgreSQL для адміністрування серверних ресурсів»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 - Комп'ютерна інженерія
(код, найменування спеціальності)
освітньо-професійної програми Комп'ютерна інженерія
(назва)

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ *Володимир МАТВІЄНКО*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. КІД-41
Володимир МАТВІЄНКО
Ім'я, ПРІЗВИЩЕ

Керівник: Олександр ЗУБ
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Рецензент: _____
науковий ступінь, вчене звання Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут Інформаційних технологій

Кафедра Комп'ютерної інженерії

Ступінь вищої освіти бакалавр

Спеціальність 123 - Комп'ютерна інженерія

Освітньо-професійна програма Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____ Наталія ЛАЩЕВСЬКА
_____
«_____» _____ 2025р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Матвієнко Володимир Дмитрович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Веб-сайт з інтеграцією PostgreSQL для адміністрування серверних ресурсів
керівник кваліфікаційної роботи Олександр ЗУБ,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025р. № 56
2. Строк подання кваліфікаційної роботи«05» червня 2025р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література з веб-технологій та розробки програмного забезпечення; архітектура клієнт-серверних додатків; СУБД, PostgreSQL; методології розробки веб-додатків Agile, Scrum, Waterfall; технології побудови клієнтської частини: Angular, TypeScript; технології взаємодії клієнта з сервером через REST API; стандарти безпечного доступу до баз даних; інтерфейси для визначення типів даних у TypeScript; принципи організації CRUD операцій в веб-додатках; особливості адміністрування та роботи з адміністративною панеллю; особливості побудови та обробки SQL-запитів у PostgreSQL; сучасні підходи до забезпечення продуктивності та надійності серверної частини веб-додатків; інструменти для управління базами даних: pgAdmin; технології маршрутизації та компоненти в Angular Router, standalone components; засоби аутентифікації та авторизації в веб-додатках; формати передачі даних між клієнтом та сервером JSON.
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Архітектурні та технологічні засади створення сучасного веб-додатку
2. Аналіз вимог і вибір технологій для розробки веб-сайту з серверною частиною
3. Проектування та реалізація веб-додатку
5. Перелік ілюстративного матеріалу: *презентація*
6. Дата видачі завдання «24» лютого 2025

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір джерел інформації	24.02-28.02.2025	Виконано
2	Постановка задачі	01.03-09.03.2025	Виконано
3	Аналіз	10.03-14.03.2025	Виконано
4	Розробка проекту	15.03-20.04.2025	Виконано
5	Написання роботи	21.04-06.05.2025	Виконано
6	Висновок	07.05-08.05.2025	Виконано
7	Підготовка демонстраційного матеріалу	09.05-19.05.2025	Виконано
8	Попередній захист роботи	27.05-29.05.2025	Виконано
9	Пред'явлення роботи в деканат	03.06-05.06.2025	Виконано

Здобувач вищої освіти _____ Володимир МАТВІЄНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Олександр ЗУБ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 49 стор., 7 рис., 5 табл., 23 джерел.

Мета роботи – розробити веб-сайт з інтеграцією PostgreSQL для адміністрування серверних ресурсів

Об'єкт дослідження – розробка веб-сайту з серверною частиною PostgreSQL та клієнтською частиною Angular

Предмет дослідження – веб-сайт, серверна частина PostgreSQL та клієнтська частина Angular

Короткий зміст роботи: У кваліфікаційній роботі розглянуто створення веб-додатку з використанням сучасного підходу до клієнт-серверної архітектури. У процесі дослідження було здійснено аналіз предметної області, визначено основні вимоги до функціоналу системи, а також обґрунтовано вибір технологій. Для розробки клієнтської частини використано фреймворк Angular. Серверна частина реалізована з використанням PostgreSQL як об'єктно-реляційної системи керування базами даних та Node.js з REST API для обробки запитів.

Було розглянуто безпечні доступи до бази даних, організації структури таблиць та опису їх взаємодії. Реалізовано адміністративну панель, яка дозволяє здійснювати CRUD-операції над контентом сайту, що суттєво спрощує керування інформацією. У роботі наведено ER-діаграму бази даних, описано компоненти системи, інтерфейси та сервіси.

У результаті було створено повноцінний веб-додаток з можливістю масштабування, зручним інтерфейсом для користувачів та адміністраторів, який повністю відповідає поставленій меті та вирішує окреслені задачі.

КЛЮЧОВІ СЛОВА: API, SPA, PWA, ANGULAR, REACT, JS, NODE.JS, POSTGRESQL, REST-API, КЛІЄНТ-СЕРВЕР, СУБД, SQL, AGILE, NODE.JS.

ЗМІСТ

ЗМІСТ 7
ВСТУП 9
1 АРХІТЕКТУРНІ ТА ТЕХНОЛОГІЧНІ ЗАСАДИ СТВОРЕННЯ СУЧАСНОГО ВЕБ-ДОДАТКУ 10
1.1 Основи побудови веб-додатків 10
1.1.1 Переваги, відмінності та види веб-додатків 11
1.1.2 Архітектура клієнт-сервер 12
1.1.3 Роль серверної частини 15
1.2 Огляд сучасних СУБД 16
1.2.1 Реляційні та нереляційні СУБД 18
1.2.2 Об’єктно-реляційна система PostgreSQL 19
1.3 Методології розробки веб-додатків 20
1.3.1 REST API 20
1.3.2 Принципи безпечного доступу до бази даних 22
1.4 Технології клієнтської частини 23
1.4.1 Можливості Angular 23
1.4.2 Взаємодія з сервером 25
2 АНАЛІЗ ВИМОГ І ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ВЕБ-САЙТУ З СЕРВЕРНОЮ ЧАСТИНОЮ 26
2.1 Проблема, яку вирішує веб-додаток 26
2.2 Користувачі та їх потреби 28
2.3 Аналіз вимог до серверної частини 29
2.3.1 Функціональні вимоги 30
2.3.2 Нефункціональні вимоги 32
2.4 Огляд схожих систем та їх серверних реалізацій 33
2.5 Вибір технологій і обґрунтування 35
3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ 37
3.1 Клієнтська частина 37
3.1.1 Загальна схема клієнтської частини 37
3.1.2 CRUD-операції 39
3.1.3 Інтерфейси 41
3.2 Серверна частина 43
3.2.1 Опис таблиць 43

3.2.2 Під’єднання до бази даних 45
3.2.3 Rest API контролер 46
3.3 Панель адміністратора 49
3.3.1 Загальна схема панелі адміністратора 50
3.3.2 CRUD операції 51
3.3.3 Angular Material 53
ВИСНОВКИ 56
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 58
ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 60
ЛІСТИНГИ ОСНОВНИХ КОМПОНЕНТ 69

ВСТУП

У наш час веб-додатки стали невід’ємною частиною сучасного цифрового середовища. Вони використовуються у найрізноманітніших сферах діяльності — від електронної комерції та соціальних мереж до освітніх платформ і систем управління підприємствами. Завдяки широкому розповсюдженню Інтернету веб-додатки забезпечують користувачам швидкий та зручний доступ до інформації та послуг з будь-якої точки світу. Сучасні веб-додатки значно відрізняються між собою за своєю структурою, функціональністю та технічною реалізацією. Простіші реалізації включають лише кілька сторінок з обмеженим функціоналом, тоді як більш складні системи складаються з багаторівневих архітектур, що включають клієнтську частину, серверну логіку та базу даних. Для забезпечення надійної роботи таких додатків використовуються передові веб-технології та фреймворки, які дозволяють досягти високої продуктивності, безпеки та масштабованості. Одним з важливих компонентів сучасного веб-додатку є база даних, яка відповідає за зберігання, обробку та доступ до структурованої інформації. Використання ефективної системи керування базами даних дозволяє реалізувати стабільну та гнучку роботу додатку, особливо в умовах обробки великого обсягу даних. Ще одним важливим елементом є серверна частина, яка обробляє запити від користувачів, взаємодіє з базою даних та забезпечує логіку виконання дій. Також важливою складовою є адміністративна панель, яка дозволяє керувати контентом додатку без необхідності змінювати програмний код. Це особливо корисно у випадках, коли інформація на сайті часто оновлюється або має керуватися кількома відповідальними особами. Наявність гнучкої та зручної системи адміністрування дозволяє забезпечити ефективне управління ресурсами веб-додатку, зменшити залежність від розробників і прискорити роботу з контентом. Таким чином, створення повноцінного веб-додатку потребує комплексного підходу до проектування, розробки та інтеграції різних технологічних компонентів, що дає змогу досягти високої якості, надійності та зручності в користуванні.

1 АРХІТЕКТУРНІ ТА ТЕХНОЛОГІЧНІ ЗАСАДИ СТВОРЕННЯ СУЧАСНОГО ВЕБ-ДОДАТКУ

У сучасному цифровому середовищі веб-додатки стали невід’ємною частиною бізнес-процесів, комунікацій і повсякденного життя користувачів. Їх популярність пов’язана з широкими можливостями і доступністю з будь-якого пристрою. Для ефективної розробки таких додатків необхідно глибоко розуміти як архітектурні принципи побудови програмних рішень, так і технології, що лежать в основі їх функціонування.

1.1 Основи побудови веб-додатків

Веб-додаток - це програмне забезпечення, або програма, яка відкривається за допомогою будь-якого браузера. У нього можуть бути присутні різні функції, в залежності від виду веб-додатка наприклад, заповнення форм, редагування документів, перегляд персоналізованого вмісту, покупки в Інтернеті та керування грошима. На відміну від звичайного програмного забезпечення, яке потрібно встановлювати на комп'ютер, веб-додаток працює інакше: користувачу не потрібно нічого встановлювати, тому що все зберігається на сервері, і доступ отримується до нього через Інтернет.

Ключова відмінність між веб-додатком і звичайним веб-сайтом полягає в тому, як вони функціонують. Звичайний веб-сайт здебільшого відображає інформацію, а веб-додаток дозволяє активно взаємодіяти з його функціями. Користувач може вводити дані, отримувати результати в режимі реального часу і налаштовувати вміст відповідно до своїх потреб.

1.1.1 Переваги, відмінності та види веб-додатків

Веб-додатки є невід'ємними інструментами сучасного цифрового середовища. Їх популярність пояснюється перевагами, які роблять цей формат програмного забезпечення зручним, економічно вигідним, а головне масштабованим як для розробників, так і для кінцевих користувачів.

Однією з ключових переваг веб-додатків є їх доступність. Користувач може отримати доступ до функціоналу додатка з будь-якого пристрою, що має веб-браузер і підключення до Інтернету, без потреби встановлювати додаткове програмне забезпечення. Це значно полегшує використання додатків на різних платформах, зокрема Windows, macOS, Linux, Android чи iOS.

Ще однією важливою перевагою є централізоване оновлення. Оскільки веб-додаток розміщується на сервері, усі зміни або виправлення одразу стають доступними всім користувачам без необхідності ручного оновлення клієнтської частини. Це забезпечує високий рівень контролю безпеки й захисту даних, бо більша частина інформації зберігається переважно на сервері, а не на пристрої користувача.

Крім того, веб-додатки краще масштабуються відповідно до потреб бізнесу. Їх легко адаптувати під зростаючу кількість користувачів або інтегрувати з іншими сервісами через API. Використання хмарних технологій дозволяє ще ефективніше керувати ресурсами та забезпечити високу доступність сервісу.

За типом реалізації веб-додатки поділяються на низку основних видів:

- *Статичні веб-додатки*. Прості за структурою, містять завчасно сформовані HTML-сторінки, без складної логіки або взаємодії з сервером. Прикладом може бути портфоліо або лендінг;
- *Динамічні веб-додатки.* Формують контент у реальному часі, залежно від запитів користувача. У таких додатках використовується серверна логіка, бази даних і шаблонізатори;
- *Односторінкові додатки (SPA, Single Page Application).* Працюють без перезавантаження сторінки завдяки активному використанню JavaScript

Завантаження нових даних відбувається через API-запити;

– *Прогресивні веб-додатки (PWA).* Поєднують переваги класичних веб-сайтів і мобільних додатків, дозволяють працювати офлайн, надсилати push-сповіщення і мати власну іконку на екрані пристрою;

– *Портали й корпоративні системи*. Масштабні рішення з інтеграцією модулів: CRM, ERP, бухгалтерія, документообіг тощо.

1.1.2 Архітектура клієнт-сервер

Архітектура клієнт-сервер передбачає розділення системи на два основні компоненти — клієнтську частину, що взаємодіє з користувачем, і серверну частину, яка обробляє запити, виконує логіку та працює з базами даних. Такий підхід дозволяє створювати масштабовані, гнучкі та зручні в обслуговуванні системи.

Клієнт — це інтерфейс зі сторони front-end, через який користувач взаємодіє з додатком. Зазвичай це браузер або мобільний застосунок, що відображає веб-сторінки, приймає інформацію та відправляє запити на сервер.

Сервер — це логічна основа додатку зі сторони back-end, яка приймає запити, виконує обробку даних, звертається до баз даних і повертає клієнту потрібні йому дані. Основні завдання сервера забезпечувати безпеку, контроль доступу, зберігати інформацію та обробляти логіку бізнес-процесів.

Залежно від складності системи та вимог до масштабованості, застосовуються різні рівні архітектури:

Однорівнева архітектура(1-tier). Це найпростіша модель, де інтерфейс, логіка та база даних працюють в одному середовищі і в більшості випадків на одному пристрої. Такий підхід може бути застосований у локальних або автономних програмах, де не потрібно підключатися до віддалених ресурсів.

Прикладом може бути проста локальна база даних для зберігання контактів у форматі Excel, в яку дані завантажуються розробником особисто. У веб-розробці така архітектура майже не використовується, через неможливість забезпечити доступ з різних пристроїв і роботи у мережі.

Дворівнева архітектура(2-tier). У цьому випадку клієнт взаємодіє безпосередньо із базою даних, як видно на рисунку (рис. 1.1). Частина логіки розміщується на клієнті, решта — на сервері. Наприклад, клієнт надсилає запити(request) у форматі SQL-запитів, а сервер виконує отримані команди і повертає результат(response).

Така архітектура підходить для локальних мереж або невеликих корпоративних систем, де кількість користувачів обмежена, а вимоги до безпеки та масштабованості невисокі.

Недоліками такої архітектури є:

– Слабка безпека, бо клієнт може бачити структуру бази даних;
– Погана масштабованість;
– Необхідність встановлення клієнтської частини на кожен пристрій.

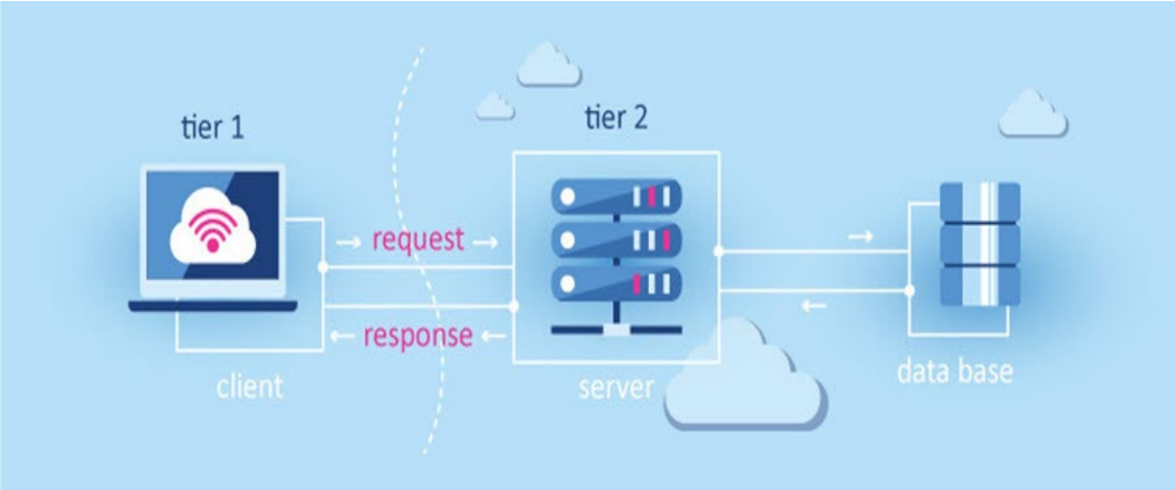


Рис. 1.1 – Дворівнева архітектура сервера

Трирівнева архітектура(3-tier). Це найбільш поширена модель для сучасних веб-додатків(рис. 1.2).

У цій архітектурі система ділиться на три логічні рівні:

– *Клієнтський рівень (client).* Цей рівень відповідає за відображення інформації та взаємодії з користувачем. Це веб-інтерфейс, написаний за допомогою технологій

HTML, CSS, JavaScript, фреймворках типу Angular або бібліотеках типу React;

– *Рівень бізнес-логіки (application server).* Цей рівень відповідає за обробку запитів, реалізацію логіки додатку. Реалізується на сервері який приймає дані від клієнта, обробляє їх і передає до бази даних. Реалізується за допомогою Node.js, Java, Python тощо;

– *Рівень даних (data base server).* Відповідає за роботу з базою даних. На цьому рівні зберігаються всі структури, таблиці, записи, журнали змін.

Перевагами такої архітектури є:

– Безпека, централізоване керування логікою, масштабованість, гнучкість;

– Клієнт не має прямого доступу до бази, що дозволяє контролювати та фільтрувати всі запити.

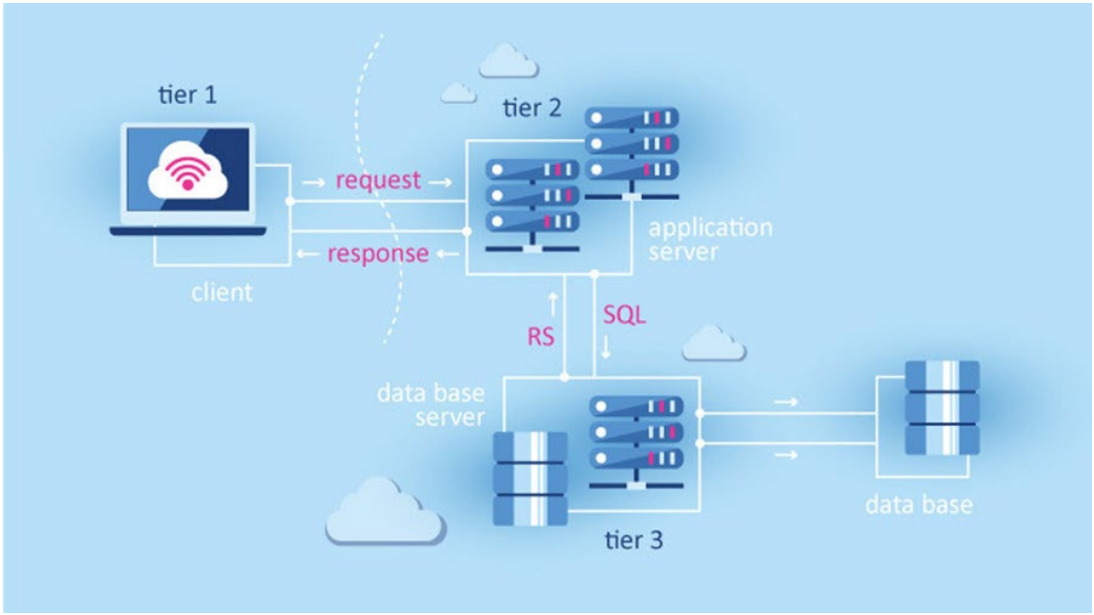


Рис. 1.2 – Трирівнева архітектура сервера

Багаторівнева архітектура(n-tier). Це розширення трирівневої архітектури, коли між основними компонентами додаються додаткові рівні(рис. 1.3), такі як:

– *API-рівень.* Окремий сервер для обробки API-запитів наприклад, REST або GraphQL;

– *Рівень мікросервісів*. Кожна бізнес-функція реалізується як окремий

мікросервіс, який працює незалежно;

– *Кеш-рівень.* Використовується Redis або Memcached для збереження часто використовуваних даних;

– *Балансувальник навантаження.* Розподіляє трафік між кількома серверами;

– *Сервіси безпеки або аналітики*. Для захисту та збору даних про використання системи.

Така архітектура є оптимальним рішенням для складних, великих, розподілених систем. Вона забезпечує найвищий рівень гнучкості, відмовостійкості та масштабованості.

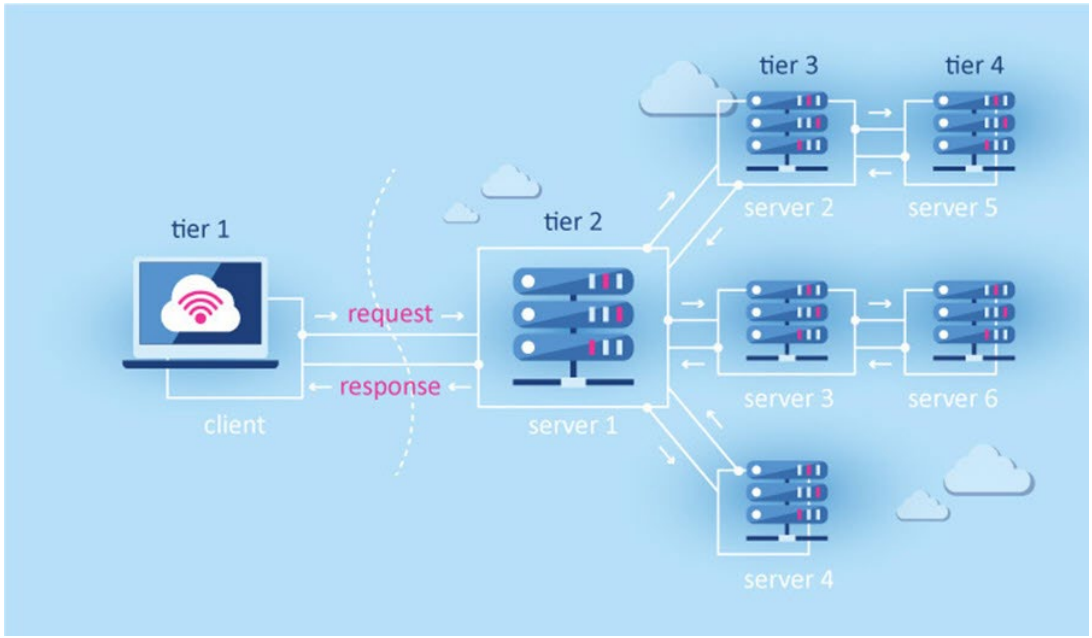


Рис. 1.3 – Багаторівнева структура сервера

1.1.3 Роль серверної частини

Серверна частина веб-додатку відповідає за логіку, обробку запитів, взаємодію з базами даних, безпеку та стабільність функціонування всієї системи. На відміну від клієнтської частини, яка зосереджена на інтерфейсі користувача, серверна виконує всі внутрішні та невидимі для користувача процеси.

Основна роль серверної частини — приймати запити від клієнта з браузера або мобільного застосунку, обробляти ці запити відповідно до бізнес-логіки

додатку та взаємодіяти з базою даних і повертати клієнтові необхідну інформацію. Такий механізм дозволяє централізовано керувати логікою, зменшуючи навантаження на клієнтську сторону, і гарантує контроль над потоком даних.

Серверна частина реалізується у більшості випадків у вигляді окремого застосунку, написаного на мовах програмування, таких як JavaScript Node.js, Python, Java, PHP або C#. Цей застосунок працює на сервері, де виконується обробка даних, маршрутизація запитів, автентифікація користувачів, обробка форм, генерування звітів, інтеграція з іншими сервісами через API тощо.

Сервер є посередником між клієнтом і базою даних, що дозволяє забезпечити безпеку, перевірку правильності запитів, фільтрацію небезпечних запитів, а також запис та прослуховування дій користувачів.

Крім того, серверна частина відповідає за реалізацію механізмів безпеки: реєстрацію, авторизацію, контроль доступу, шифрування даних, обмеження доступу до ресурсів. У випадку багатокористувацького середовища сервер керує сесіями, куками, токенами автентифікації, а також визначає, які дії дозволені кожному користувачеві.

У разі зростання кількості користувачів або навантаження на систему саме серверна логіка адаптується до нових умов. Це може бути реалізовано шляхом балансування навантаження, горизонтального або вертикального масштабування, використання мікросервісів або кешування даних.

1.2 Огляд сучасних СУБД

Серед найбільш популярних реляційних СУБД(Системи управління базами даних) сьогодні виділяється PostgreSQL — відкрита об'єктно-реляційна система, що підтримує складні запити, транзакції та розширення. Вона успішно використовується як альтернатива комерційним СУБД на кшталт Oracle чи

Microsoft SQL Server, демонструючи високу продуктивність навіть у масштабних проектах. Іншою відомою реляційною СУБД є MySQL, яка залишається популярною завдяки простоті, широкій підтримці та швидкодії при виконанні базових запитів. MariaDB як MySQL також набула поширення, особливо серед прихильників відкритого програмного забезпечення.

Таблиця 1.1 – Порівняння сучасних СУБД

Назва СУБД	Тип СУБД	Ключові особливості
PostgreSQL	Об'єктно-реляційна	Підтримка складних SQL-запитів, транзакцій, функцій, розширень, роботи з JSON
MySQL	Реляційна	Простий SQL, ефективна обробка запитів, активна підтримка
MariaDB	Реляційна	Форк MySQL, відкритий код, сумісність з MySQL
Oracle Database	Реляційна (комерційна)	Висока масштабованість, складні транзакції, оптимізатори, реплікації
Microsoft SQL Server	Реляційна (комерційна)	Інтеграція з екосистемою Windows, підтримка T-SQL
MongoDB	Документо-орієнтована (NoSQL)	Зберігання у BSON-форматі, висока гнучкість структури даних

На сьогоднішній день зростає популярність NoSQL-рішень, що з’явилися у відповідь на потребу в обробці великих обсягів неструктурованої інформації,

наприклад логів, подій, мультимедійного контенту чи даних з IoT-пристроїв. Одним з лідерів у цій категорії є MongoDB, яка працює з документами у форматі BSON, розширення JSON, що дозволяє розробникам зберігати гнучкі структури даних без попередньої схематики. MongoDB часто використовується у веб-додатках, де важлива динамічність моделей даних і швидкий доступ.

Також використовуються гібридні підходи, коли в одному проекті поєднуються різні типи СУБД залежно від специфіки завдань. Наприклад, реляційна СУБД може обслуговувати фінансові транзакції, а NoSQL — забезпечувати зберігання тимчасових сесій, історії активності чи повідомлень у реальному часі.

1.2.1 Реляційні та нереляційні СУБД

Найбільш поширеними є два типи СУБД — реляційні та нереляційні. Їх вибір залежить від виду даних, потреб проекту та вимог до масштабованості, швидкодії та гнучкості.

Реляційні СУБД. Побудовані на основі математичної теорії множин і зберігають дані у вигляді таблиць (реляцій). Кожна таблиця містить записи (рядки) та поля (стовпці), а зв’язки між таблицями будуються за допомогою ключів. Найвідомішими представниками реляційних СУБД є PostgreSQL, MySQL, Oracle, Microsoft SQL Server.

Перевагою реляційних СУБД є структурованість і чіткість схеми. Перед збереженням даних створюється структура таблиць, де чітко визначені типи полів, обмеження, ключі та зв’язки. Це забезпечує цілісність, несуперечливість і узгодженість даних. Для запитів до таких баз даних використовується мова SQL (Structured Query Language).

Нереляційні СУБД(NoSQL). З’явилися як альтернатива реляційним у відповідь на потребу зберігати великі обсяги неструктурованих або слабо структурованих даних для сучасних веб-сервісів. NoSQL-бази не мають фіксованої схеми та підтримують гнучкі формати збереження, зокрема документи наприклад, JSON у MongoDB, графи Neo4j, ключ-значення Redis, колонки Cassandra.

Головною перевагою NoSQL є горизонтальна масштабованість — можливість легко розподіляти дані між багатьма вузлами, що забезпечує високу продуктивність при роботі з великими даними або високонавантаженими системами. Вони дозволяють зберігати неоднорідні за структурою дані без попереднього створення жорсткої схеми.

Але NoSQL має і недоліки: обмежена підтримка транзакцій, нижчий рівень узгодженості даних, складніші механізми зв'язків між сутностями. Такі СУБД не завжди підходять для критичних систем, де важлива цілісність і контроль над даними.

1.2.2 Об'єктно-реляційна система PostgreSQL

PostgreSQL — об'єктно-реляційна система управління базами даних, яка поєднує в собі надійність реляційної моделі з можливістю використання об'єктно-орієнтованих ідей. Вона активно розвивається з 1986 року і на сьогоднішній день є однією з провідних СУБД у світі, особливо серед веб-розробників, аналітиків даних, фінансових організацій та підприємств, що працюють із критично важливою інформацією.

Особливість PostgreSQL є у підтримці розширеної моделі даних, яка дозволяє не лише працювати з класичними таблицями, але й використовувати об'єкти, типи, наслідування, індексацію за різними алгоритмами, а також створювати власні функції й типи даних. Саме така гнучкість робить PostgreSQL об'єктно-реляційною системою, яка зберігає переваги реляційного підходу і водночас надаючи інструменти об'єктного програмування.

Серед важливих функціональних можливостей цієї СУБД можна виділити:

- Підтримку транзакцій з повною ACID-гарантією;
- Систему ролей і розширену модель доступу;
- Повнотекстовий пошук;
- Реплікацію, створення резервних копій та кластеризацію;
- Масштабованість через модулі та розширення.

PostgreSQL добре масштабується як вертикально, на одній машині, так і горизонтально, за допомогою кластерних рішень. Завдяки високій продуктивності, можливості обробляти великі обсяги даних і широкій підтримці мов програмування, вона є універсальним інструментом для створення серверної частини веб-сервісів.

1.3 Методології розробки веб-додатків

У розробки веб-додатків найчастіше застосовуються гнучкі Agile методології, які дозволяють швидко адаптуватися до змін вимог і забезпечити постійну взаємодію з користувачем. Найпоширенішим підходом є Scrum — ітераційна модель, в якій проект поділяється на спринти з чітко визначеним набором задач.

Також використовується методологія Kanban, що зосереджена на візуалізації робочого процесу, безперервній доставці та обмеженні незавершених завдань.

Для складніших, формалізованих проектів актуальною залишається модель водоспаду Waterfall, де весь процес проходить через послідовні етапи: аналіз, проектування, реалізація, тестування, впровадження. Цей підхід менш гнучкий, але забезпечує чітку документацію та контроль за кожною фазою.

Також є підхід DevOps, який інтегрує розробку і підтримку. Основною метою DevOps є автоматизація процесів розгортання, безперервної інтеграції та безперервної доставки, що дозволяє швидко та безпечно впроваджувати зміни у веб-додатках.

1.3.1 REST API

REST API (Representational State Transfer Application Programming Interface) — це архітектурний стиль взаємодії між компонентами програмного забезпечення. REST дозволяє клієнтській частині наприклад, браузеру або мобільному

застосунку ефективно обмінюватися даними із серверною частиною веб-додатка через HTTP-запити(рис. 1.4).

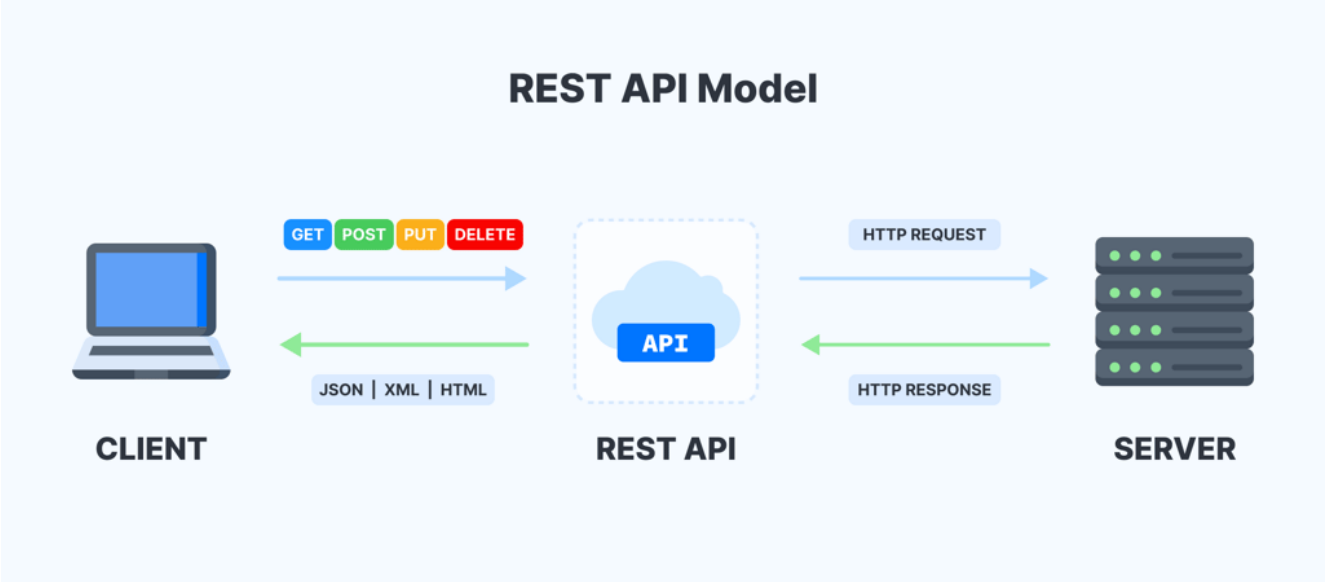


Рис. 1.4 – Модель REST API

REST-архітектура базується на кількох ключових принципах, серед яких головним є використання стандартних методів HTTP:

– *GET*. Для отримання даних;
– *POST*. Для створення нових об'єктів;
– *PUT / PATCH*. Для оновлення існуючих даних;
– *DELETE*. Для видалення ресурсів.

Кожен ресурс у REST-API ідентифікується унікальним URL-адресом. Наприклад, запит «GET /users/123» може повертати інформацію про користувача з ідентифікатором 123.

REST API є незалежним від технологій, що дозволяє використовувати його з будь-якою клієнтською платформою — від JavaScript-додатків у браузері до мобільних застосунків на iOS чи Android. Обмін даними зазвичай відбувається у форматі JSON.

REST API є ключовим у розробці SPA, він дозволяє реалізувати розподілену архітектуру, де клієнт і сервер працюють незалежно один від одного.

У веб-додатках REST API слугує мостом між клієнтською частиною та сервером, на якому відбувається обробка логіки, зберігання та робота з даними. Саме через REST API клієнт може надсилати дані(HTTP request) на сервер і отримувати потрібну інформацію(HTTP response).

1.3.2 Принципи безпечного доступу до бази даних

Безпека бази даних є основою будь-якого веб-додатку яким користуються клієнти, оскільки саме вона забезпечує збереження, конфіденційність та цілісність даних, які користувачі надають для можливості та швидкості обробки їх запитів. У процесі розробки серверної частини потрібно приділити особливу увагу на організації безпечного доступу до бази даних, аби уникнути витоків, атак і зловживань.

Головним принципом безпеки є автентифікація та авторизація. Кожен запит до бази даних має виконуватись тільки після перевірки особистості користувача (автентифікації) та визначення його прав доступу(авторизації). Ролі та права повинні бути чітко закріпленні за кожним хто працює з даними, щоб надавати доступ лише до дозволених саме йому операцій.

Ще одним принципом безпеки є використання параметризованих запитів. Це дозволяє уникнути однієї з найпоширеніших атак, а саме: SQL-ін’єкцій, коли зловмисник намагається відправити шкідливий SQL-код у вхідні дані. Замість створення динамічного запиту з рядка, варто використовувати підготовлені конструкції, що автоматично екранують небезпечні символи.

Наступним принципом буде шифрування даних. Це стосується як передавання даних через мережу, так і збереження їх у базі даних. Для передавання даних наполегливо рекомендується використовувати протокол HTTPS щоб створити безпечний екрануючий шлях для даних, а для зберігання — шифрування конфіденційної інформації наприклад, паролів за допомогою bcrypt або інших криптостійких алгоритмів.

Також не треба забувати про регулярні перевірки журналів доступу, а саме логів програми. Це дозволить виявити підозрілу активність до того як ймовірна

загроза почне впливати на роботу додатку наприклад, несанкціоновані спроби доступу, масові запити або помилки авторизації. Своєчасне виявлення таких подій дозволить оперативно відреагувати і, що найголовніше захистити персональні дані клієнтів.

1.4 Технології клієнтської частини

Для створення клієнтської частини веб-додатків використовуються комбінації технологій, фреймворків і бібліотек. Основою клієнтської частини є HTML для структури, CSS для стилізації та JavaScript для логіки та взаємодії з користувачем.

Серед JavaScript-фреймворків однією з найпопулярніших технологій є Angular, що надає засоби для створення масштабованих SPA сторінок з двостороннім зв’язком даних, системою компонентів, службами, маршрутизацією та вбудованими механізмами безпеки.

Іншими поширеними технологіями є React — бібліотека для побудови інтерфейсів від компанії Meta, що забезпечує гнучкий підхід до компонування елементів, та Vue.js, яка вирізняється простотою та легкістю інтеграції у проекти різного масштабу.

Для стилізації активно використовуються препроцесори на зразок Sass або LESS, а також фреймворки Bootstrap і Tailwind CSS, що дозволяють швидко створювати адаптивні, кросбраузерні дизайни.

Керування станом програми у складних проектах реалізується через бібліотеки, такі як NgRx для Angular, або Redux для React, що дозволяють централізовано зберігати та оновлювати дані.

1.4.1 Можливості Angular

Angular — це сучасний фреймворк для розробки динамічних веб-додатків на

мові TypeScript, який підтримується компанією Google.

Angular має двостороннє зв’язування даних. Це означає, що зміни в інтерфейсі автоматично оновлюють дані в логіці програми, і навпаки. Такий механізм спрощує синхронізацію між моделлю та шаблоном, зменшуючи обсяг коду, необхідного для реалізації інтерфейсу.

Фреймворк має систему компонентів, які є будівельними блоками додатку. Кожен компонент складається з HTML-шаблону, стилів і логіки на мові програмування TypeScript. Компонентний підхід дозволяє легко підтримувати, розширювати й тестувати проект. Компоненти можуть бути повторно використані в різних частинах додатку, що знижує дублювання коду.

Angular також має вбудований механізм маршрутизації (router), який дає змогу легко створювати SPA-додатки з кількома «уявними сторінками», не перезавантажуючи браузер. Система маршрутизації дозволяє переходити між наповненням, передавати параметри в URL, захищати маршрути від несанкціонованого доступу тощо.

Сильною стороною Angular є інжекція залежностей. Це патерн, що дозволяє уникнути жорсткої прив’язки модулів один до одного, що сприяє більш гнучкому, модульному коду та полегшує тестування. Angular автоматично створює та впроваджує потрібні об’єкти туди, де вони необхідні.

Angular підтримує розширене тестування через інтеграцію з такими інструментами як Jasmine, Karma та Protractor. Це дає можливість писати юніт-тести, інтеграційні тести та e2e-тести. Для оптимізації продуктивності Angular надає Ahead-of-Time компіляцію, яка компілює HTML і TypeScript до JavaScript ще до завантаження у браузер, що скорочує час запуску програми. Для легшого керування фреймворком можна використовувати Angular CLI — зручна командна утиліта, яка автоматизує створення компонентів, збірку, запуск серверів розробки, тести та багато іншого.

Angular також чудово для створення масштабованих проектів завдяки підтримці модульності, можливості lazy loading, та суворому типізуванню з використанням TypeScript, що дає команді розробників чітку структуру проекта і

зменшує вірогідність помилки.

Завдяки підтримці великої спільноти, розвинутій документації та численним бібліотекам, Angular активно використовується в корпоративному середовищі для створення великих, стабільних і надійних додатків. Його використання дозволяє реалізовувати як прості сайти, так і складні системи керування даними, адміністративні панелі, дашборди та мобільні гібридні додатки.

1.4.2 Взаємодія з сервером

Angular забезпечує широкі можливості для організації обміну даними між клієнтом та сервером, зокрема через REST API. Для роботи з віддаленими сервісами Angular використовує вбудований сервіс HttpClient. Він дозволяє здійснювати HTTP-запити, а саме: GET, POST, PUT, DELETE, обробляти відповіді сервера, реалізовувати обробку помилок, інтерцептори, кешування, а також використовувати токени автентифікації. Процес взаємодії відбувається наступним чином: користувач у клієнтській частині виконує певну дію, що активує функцію в Angular-компоненті, ця функція викликає метод відповідного сервісу, в якому реалізовано логіку надсилання HTTP-запиту на сервер і у відповідь сервер надсилає дані у вигляді JSON формату, які Angular обробляє й відображає користувачеві.

Фреймворк підтримує асинхронну взаємодію з використанням RxJS — бібліотеки реактивного програмування. Об’єкти типу Observable, які повертає HttpClient, дозволяють обробляти потоки даних, реагувати на зміни. Це особливо корисно при створенні складних інтерфейсів, де необхідно взаємодіяти з кількома API-ресурсами одночасно або в певному порядку для кращої роботи додатка. Крім базового обміну даними, Angular дозволяє реалізовувати автентифікацію та авторизацію користувачів за допомогою токенів наприклад, JWT — JSON Web Tokens, які додаються до заголовків HTTP-запитів. Це дозволяє будувати захищені маршрути, контролювати доступ до певних частин додатку й реалізовувати механізми захисту на клієнтському рівні.

2 АНАЛІЗ ВИМОГ І ВИБІР ТЕХНОЛОГІЙ ДЛЯ РОЗРОБКИ ВЕБ-САЙТУ З СЕРВЕРНОЮ ЧАСТИНОЮ

Перед початком розробки сучасного веб-додатку критично важливо чітко визначити його функціональне призначення, цільову аудиторію, очікувані обсяги навантаження, а також вимоги до безпеки, продуктивності та масштабованості. Ретельне формулювання функціональних і нефункціональних вимог дає змогу створити ефективну систему, що відповідає потребам користувачів і забезпечує стабільну роботу в різних умовах.

Важливо обрати оптимальні мови програмування, фреймворки, засоби взаємодії з базами даних та архітектурні рішення, які дозволяють забезпечити гнучкість, розширюваність і простоту в подальшій підтримці. Використання REST API, підтримка авторизації, безпечна обробка даних, робота з форматами JSON, обробка помилок і реалізація взаємодії між клієнтом і сервером — усе це має бути враховано ще на етапі аналізу.

Ретельний вибір технологій і програмних компонентів дозволяє скоротити час розробки, зменшити витрати на підтримку та підвищити якість кінцевого продукту, а також забезпечити простоту інтеграції з іншими системами та сервісами.

2.1 Проблема, яку вирішує веб-додаток

Проблеми, що виникають у багатьох організацій, компаній і проектів, є необхідність забезпечення швидкого, зручного й постійного доступу до сервісів без прив’язки до конкретного пристрою або операційної системи. Традиційне програмне забезпечення часто потребує складного встановлення, ручного оновлення, а також займає пам’ять на пристрої. Це створює технічні бар’єри як для

користувачів, так і для адміністраторів системи.

Веб-додаток повністю усуває ці труднощі. Його не потрібно встановлювати: користувач отримує доступ через звичайний веб-браузер. Завдяки цьому зменшуються витрати на підтримку та оновлення, оскільки зміни реалізуються безпосередньо на сервері й автоматично стають доступними всім користувачам. Це критично важливо для масштабованих проектів, де постійне оновлення контенту та функціоналу — звична практика.

Окремо варто виділити проблему централізованого управління даними. У багатьох організаціях інформація розкидана по різних джерелах або файлах, що ускладнює облік, аналіз та забезпечення цілісності. Веб-додаток дозволяє створити єдину базу даних, що забезпечує систематизацію й доступ до актуальної інформації в режимі реального часу.

Ще одна поширена проблема — обмежений або незручний доступ до адміністративного керування сайтом. Без відповідної інфраструктури кожна зміна вмісту, оновлення новин, додавання товарів, редагування текстів тощо, вимагає втручання розробника. Це не тільки уповільнює роботу, а й створює залежність від технічних спеціалістів. Для власників малого або середнього бізнесу, освітніх установ чи приватних проектів така модель є затратною і малоефективною.

Саме цю проблему вирішує веб-додаток. Однією з його ключових функцій є наявність зручної адмін-панелі, яка дозволяє уповноваженим користувачам самостійно керувати контентом: додавати нові матеріали, змінювати інформацію, видаляти застарілі записи без потреби в знанні програмування.

Зокрема веб-додаток вирішує такі конкретні проблеми:

- *Проблема ручного редагування коду при зміні контенту.* Завдяки графічному інтерфейсу адміністратор може вносити зміни безпосередньо через браузер;
- *Проблема несистематизованого зберігання даних.* Інформація зберігається в централізованій базі даних PostgreSQL, що дає змогу здійснювати пошук, фільтрацію, редагування тощо;
- *Проблема відсутності швидкого доступу до оновлень.* Зміни одразу ж

доступні всім користувачам після їх внесення адміністратором;

– *Проблема масштабування.* Завдяки клієнт-серверній архітектурі додаток може обслуговувати велику кількість користувачів одночасно.

2.2 Користувачі та їх потреби

При створенні будь-якого сучасного додатку першочергове значення має розуміння цільової аудиторії. Глибокий аналіз потреб користувачів дозволяє сформувати ефективний, зручний і функціональний інструмент, який відповідатиме очікуванням реальних людей. Без врахування специфіки поведінки, інтересів і цілей користувачів. Навіть технічно досконалий продукт може залишитися непоміченим або викликати незадоволення.

У випадку розробки веб-додатку, що містить динамічний контент, інтеграцію з базою даних і адміністративну панель керування, можна виділити дві основні категорії користувачів: кінцеві відвідувачі сайту та адміністратори або модератори, які відповідають за наповнення й обслуговування ресурсу.

Першу категорію становлять звичайні користувачі – ті, хто заходить на сайт для перегляду, пошуку чи взаємодії з контентом. Для них найважливішим є інтерфейс – він має бути зрозумілим, приємним на вигляд і адаптивним, тобто зручним для використання з мобільних пристроїв, планшетів та настільних комп’ютерів.

Користувачі також хочуть мати можливість швидко знаходити потрібну інформацію або товари, мати зручний фільтр або пошук, бачити актуальні дані й отримувати зворотний зв’язок, якщо така функція передбачена. Особливо цінується збереження часу та енергії, тому будь-які перешкоди у використанні знижують рівень задоволеності й зменшують шанс повторного візиту.

Важливо також враховувати інклюзивність: частина користувачів може мати

обмеження зору або інші особливості, тож додаток має бути максимально доступним – з контрастним дизайном, читабельними шрифтами та логічною структурою контенту.

Інша категорія користувачів – це адміністратори, контент-менеджери або інші відповідальні особи, які займаються наповненням та обслуговуванням сайту. Їх потреби мають технічний характер. Вони очікують, що веб-додаток дозволить швидко і без помилок додавати нові сторінки, записи, зображення, редагувати або видаляти інформацію, організовувати матеріали за категоріями. Важливим є зручний інтерфейс адмін-панелі, який дозволяє керувати контентом без знання мов програмування.

Адміністративна частина додатку має бути гнучкою й безпечною. Важливо, щоб можна було налаштовувати різні ролі доступу наприклад, одні користувачі мають повний доступ до структури сайту, інші – лише до розділу новин чи галереї. Це дозволяє ефективно розподілити обов’язки в команді та зменшити ризики ненавмисного пошкодження структури додатку.

Окрему категорію потреб можна виділити серед розробників або технічних адміністраторів, які відповідають за підтримку коду, оновлення функціоналу, безпеку й масштабованість додатку. Для них важливо, щоб код був структурованим, легко підтримуваним, щоб можна було швидко інтегрувати зовнішні сервіси або розширити систему новими модулями. Вони очікують наявності REST API, чіткої документації, розділення логіки клієнтської і серверної частин, підтримки сучасних стандартів і фреймворків.

2.3 Аналіз вимог до серверної частини

Сервер повинен забезпечувати коректну взаємодію з реляційною базою даних PostgreSQL. Це передбачає підтримку складних SQL-запитів, зв’язків між

таблицями, транзакцій, обмежень цілісності, а також можливість масштабування запитів у випадку зростання обсягів даних. Сервер має коректно обробляти всі CRUD-операції (створення, читання, оновлення, видалення), реалізуючи їх через REST API, доступний для клієнтської частини.

Окрему категорію вимог становить авторизація та автентифікація користувачів. Сервер повинен розмежовувати доступ на рівні ролей (адміністратор, редактор, звичайний користувач), забезпечуючи безпечне зберігання облікових даних наприклад, через хешування паролів, та захист маршрутів API від несанкціонованого доступу.

Для роботи адміністративної панелі необхідна підтримка таких дій, як створення та редагування контенту, керування користувачами, модерація, налаштування категорій і тегів. Сервер повинен дозволяти виконання цих функцій у реальному часі з гарантією цілісності даних і зручним форматуванням відповіді у JSON.

Важливою вимогою є підтримка системи логування дій на сервері для виявлення помилок, несанкціонованих спроб доступу або проблем із продуктивністю. Це дозволить ефективно відслідковувати активність і швидко реагувати на збої.

Окрім цього, сервер має бути розгорнутий у безпечному середовищі з підтримкою SSL-з’єднання, механізмів CORS, обмеженням частоти запитів (rate limiting), та мати можливість масштабування за рахунок модульної структури.

2.3.1 Функціональні вимоги

Функціональні вимоги визначають, які саме дії має виконувати веб-додаток з точки зору користувача та адміністратора. Вони формуються на основі аналізу цільової аудиторії, особливостей предметної області та загальних завдань, які має вирішувати система. У випадку розробки сучасного веб-додатку з базою даних PostgreSQL функціональні вимоги є ключовими для формування архітектури та подальшої реалізації проекту.

Кожен відвідувач має повинен мати можливість зареєструватися, увійти в

обліковий запис, відновити пароль або змінити особисті дані. Система має розрізняти права доступу для звичайних користувачів та адміністратора.

Користувачі повинні мати змогу переглядати контент, доданий до сайту, користуватися пошуком, фільтрами або навігацією, яка дозволяє зручно працювати з великою кількістю інформації. Контент повинен бути структурований та зручно представлений в інтерфейсі — у вигляді списків, карток або таблиць залежно від призначення.

Адміністратор, у свою чергу, має доступ до адміністративної панелі, через яку можна додавати, редагувати та видаляти контент. Це включає управління текстовими матеріалами, зображеннями, відео, товарами чи іншими елементами, залежно від характеру додатку. Усі зміни з боку адміністратора автоматично відображаються на клієнтській частині сайту без необхідності втручання в код або структуру бази даних.

Важливою функціональністю є взаємодія з базою даних: зберігання інформації, її запит, сортування, фільтрація, виведення результатів на сторінку тощо. Веб-додаток повинен ефективно працювати з великими обсягами даних без зниження продуктивності.

Окрему групу вимог складають функції, пов’язані з роботою REST API. Клієнтська частина має можливість надсилати запити до серверу, отримувати у відповідь необхідні дані та оновлювати інтерфейс без перезавантаження сторінки. Це дозволяє реалізувати зручні інтерфейси з динамічними елементами, що покращують користувацький досвід.

Також до функціональних вимог належать можливості пошуку та сортування інформації, зворотного зв’язку з користувачами, сповіщення про події, адаптивність до різних пристроїв: мобільні телефони, планшети, десктопи, а також підтримка багатомовності за потреби.

Усі функції мають працювати узгоджено, у реальному часі, з урахуванням ролей користувачів та з дотриманням принципів безпеки й надійності.

2.3.2 Нефункціональні вимоги

Нефункціональні вимоги — це сукупність характеристик, які не стосуються безпосередньо функцій веб-додатку, але визначають якість його роботи, надійність, продуктивність, зручність використання та інші параметри, що впливають на загальне враження від системи. Вони є критично важливими для забезпечення стабільної та безпечної експлуатації додатку, особливо якщо передбачається робота з великою кількістю користувачів та даних.

Однією з ключових нефункціональних вимог є продуктивність. Веб-додаток повинен працювати швидко та без затримок навіть за умови великого навантаження. Це передбачає оптимізацію запитів до бази даних PostgreSQL, використання кешування, ефективне управління ресурсами сервера та мінімізацію часу відгуку API. Завдяки цьому користувачі не відчуватимуть дискомфорту під час роботи з системою.

Не менш важливою є масштабованість. Архітектура додатку має дозволяти розширення — як у частині функціональності, так і в обсягах оброблюваних даних. Це означає, що система повинна підтримувати підключення нових модулів або сервісів без потреби повної перебудови інфраструктури. Зокрема, це стосується можливості розгортання кількох серверів або балансування навантаження в майбутньому.

Наступною вимогою є доступність. Додаток має бути доступний для користувачів 24/7, з мінімальними перервами на технічне обслуговування. Для цього слід передбачити резервне копіювання даних, захист від збоїв серверного обладнання, а також використання сучасних хостинг-платформ, які гарантують високу доступність і швидкість обробки запитів.

Важливим аспектом є безпека. Дані користувачів повинні бути захищені від несанкціонованого доступу, зловмисних дій та витоків. Це досягається через застосування шифрування, захист REST API, обмеження доступу на основі ролей, використання HTTPS, захист від SQL-ін’єкцій, CSRF та XSS атак. Також необхідно реалізувати систему логування та моніторингу для виявлення підозрілої активності.

Окремо варто зазначити юзабіліті — зручність користування. Інтерфейс додатку має бути інтуїтивно зрозумілим, адаптивним до різних типів пристроїв та підтримувати основні принципи доступності для людей з обмеженими можливостями. Це дозволяє розширити коло користувачів та підвищити їхню задоволеність.

Також важливою є сумісність. Веб-додаток повинен коректно працювати у всіх сучасних браузерах: Google Chrome, Mozilla Firefox, Microsoft Edge, Safari тощо, та на різних операційних системах. Для цього використовується кросбраузерна верстка та перевірені фреймворки.

Ще одним критичним параметром є надійність — система має витримувати неочікувані збої, відновлюватися після помилок і не втрачати дані. Це забезпечується за рахунок ретельного тестування, обробки виключень, використання транзакцій у роботі з базою даних, а також регулярного збереження резервних копій.

2.4 Огляд схожих систем та їх серверних реалізацій

Для розробки веб-додатку важливо орієнтуватися на досвід існуючих рішень, які вже успішно функціонують у подібній сфері. Аналіз схожих систем дозволяє краще зрозуміти ефективні підходи до реалізації серверної частини, обрати оптимальні технології та уникнути типових помилок.

Наприклад, системи управління контентом CMS, як WordPress, Joomla або Drupal, мають складну, але добре документовану серверну архітектуру. У більшості випадків вони використовують мову програмування PHP у поєднанні з реляційними базами даних, найчастіше MySQL або MariaDB. Серверна логіка у цих системах реалізована через модульний підхід, який дозволяє легко підключати плагіни та розширення. Однак попри гнучкість, такі рішення мають обмеження в

масштабованості й менш зручні для створення високонавантажених систем або SPA-додатків.

Таблиця 2.1 – Порівняння схожих систем

Система	Тип	Переваги	Недоліки
WordPress	CMS	Швидкий старт, велика спільнота, багато плагінів	Слабка масштабованість, складність для SPA
Strapi	Headless CMS	Гнучкість, сучасна адмінка, підтримка кастомних моделей	Вимагає налаштування, не підходить для дуже великих проектів
Ghost	CMS / Блог	Мінімалізм, швидкодія, зручний інтерфейс	Обмежена функціональність
Salesforce	Корпоративна CRM	Потужна бізнес-логіка, масштабування, безпека	Висока вартість, складність розширення
Custom REST API	Власна розробка	Повний контроль, масштабованість, адаптація під бізнес-логіку	Потребує повного циклу розробки та підтримки

Інший приклад — сучасні веб-платформи на основі Node.js, наприклад, Strapi або Ghost. У цих системах серверна частина реалізована на JavaScript, що дозволяє використовувати єдину мову як для клієнта, так і для сервера. Вони орієнтовані на роботу з REST або GraphQL API, що робить їх гнучкими в контексті інтеграцій з іншими сервісами.

Окремо варто розглянути корпоративні рішення, наприклад, Salesforce або SAP. У таких системах серверна частина є централізованою, часто побудована на Java або C#, та включає потужну бізнес-логіку, складні моделі даних і розподілену інфраструктуру. Вони інтегруються з кількома зовнішніми сервісами, підтримують високий рівень безпеки та мають гнучкі інструменти адміністрування.

У більшості сучасних систем спостерігається тенденція до використання REST API як основного способу взаємодії між клієнтською та серверною частинами. Такий підхід забезпечує відокремлення логіки представлення від логіки обробки даних, що дозволяє масштабувати додаток, змінювати клієнтську частину без зміни серверної та створювати мобільні або десктопні клієнти.

2.5 Вибір технологій і обґрунтування

Під час розробки веб-додатку важливим етапом є визначення технологій, які будуть використовуватись на всіх рівнях системи: клієнтському, серверному та на рівні бази даних. Вибір повинен враховувати вимоги до функціональності, масштабованості, безпеки, зручності розробки та підтримки.

Для реалізації клієнтської частини обрано Angular — популярний фреймворк від Google для створення SPA-додатків. Angular дозволяє будувати гнучкі, динамічні інтерфейси, підтримує модульність, двосторонню прив’язку даних, має власний механізм маршрутизації, сервісів та компонентів. Завдяки TypeScript у розробці забезпечується сувора типізація, що позитивно впливає на якість коду та полегшує його тестування й масштабування. Angular має активну спільноту, велику кількість бібліотек.

Серверна частина реалізується за допомогою REST API, що дозволяє клієнтському інтерфейсу взаємодіяти з сервером через стандартизовані HTTP-запити. REST-архітектура забезпечує незалежність клієнтської та серверної частин,

що дозволяє масштабувати його компоненти окремо та полегшує інтеграцію з іншими сервісами або мобільними клієнтами в майбутньому.

У якості системи управління базами даних обрано PostgreSQL. Це об’єктно-реляційна СУБД з відкритим кодом, яка підтримує ACID-властивості, має гнучкий механізм транзакцій, високий рівень безпеки, підтримку розширень, JSON-типів даних та широкі аналітичні можливості. PostgreSQL ідеально підходить для систем, де важлива цілісність даних та гнучкість при зміні структури таблиць. Крім того, PostgreSQL легко інтегрується з різними серверними фреймворками та має розвинуту екосистему інструментів.

Для адміністрування бази даних використовується pgAdmin — офіційний інструмент керування PostgreSQL. Він має зручний графічний інтерфейс, дозволяє створювати, редагувати й моніторити об’єкти бази даних, а також виконувати SQL-запити та переглядати статистику продуктивності.

Вибір зазначених технологій обґрунтовано їхньою стабільністю, активним розвитком, великою спільнотою, документацією та відповідністю вимогам даного проекту.

3. ПРОЕКТУВАННЯ ТА РЕАЛІЗАЦІЯ ВЕБ-ДОДАТКУ

3.1 Клієнтська частина

Клієнтська частина розробленого веб-додатку реалізована з використанням фреймворку Angular. Розробка здійснювалася відповідно до принципів компонентно-орієнтованої архітектури, що дозволило ефективно управляти структурою інтерфейсу та розділяти відповідальність між окремими частинами коду.

Для організації навігації між сторінками та підкомпонентами застосунку було використано Angular Router, який надає можливість реалізувати маршрутизацію на стороні клієнта. Завдяки цьому користувачі зможуть переміщуватись між різними розділами інтерфейсу без повного перезавантаження сторінки.

Компонентна структура була побудована таким чином, щоб кожен окремий компонент відповідав за свою частину функціоналу, а саме: відображення головної сторінки, перехід до детальної інформації про товар, меню навігації тощо Компоненти було створено у форматі standalone, що означає їхню незалежність від зовнішніх модулів. Такий підхід дозволяє повторне використання компонентів, дозволяє зменшити обсяг імпортів, покращуючи продуктивність на етапі збірки.

Для взаємодії між клієнтською частиною і сервером були реалізовані Angular-сервіси. Вони відповідають за надсилання HTTP-запитів до REST API серверної частини, отримання даних та обробку відповідей.

3.1.1 Загальна схема клієнтської частини

На рисунку 3.1 видно загальну схему клієнтської частини з залежностями компонент від сервісів(admin-panel.service) та змінних(UrlLinks) з посиланнями. Також можна побачити другорядні компоненти(site-nav, site-content, site-footer), які

завантажуються до основної компоненти(site-content) і також маршрутизація з основної компоненти.

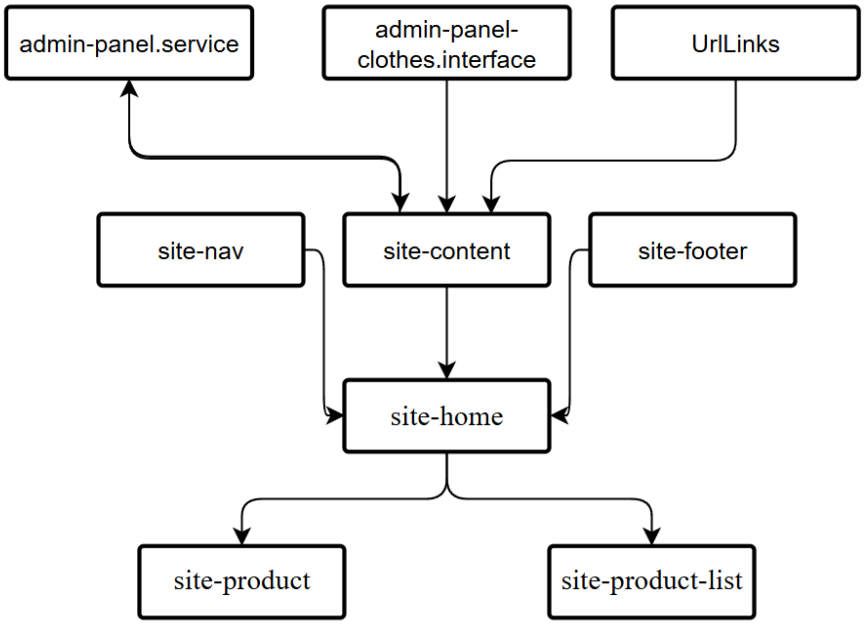


Рис. 3.1 – Загальна схема клієнтської частини веб-додатку

Сервіс admin-panel.service відповідає за виконання основних CRUD-операцій при взаємодії з серверною частиною веб-додатку. У сервісі написана логіка надсилання HTTP-запитів.

Файл UrlLinks призначений для зберігання посилань до REST API, що використовуються у веб-додатку. У цьому файлі визначено змінні, які містять відповідні URL-адреси для здійснення HTTP-запитів до серверної частини. Також цей файл експортує необхідні змінні до основної компоменти зі вмістом(site-content).

Компонента з основним вмістом клієнтського інтерфейсу — site-content — отримує дані від сервісу за допомогою GET-запиту, який починає працювати під час ініціалізації компоненти. У відповідь сервіс повертає масив даних, що містить інформацію про доступні товари одягу. Після успішного отримання цієї інформації, компонента site-home відображає отримані дані у відповідному шаблоні інтерфейсу.

Компонента site-home – є контейнером для інших підкомпонент. Вона відповідає за організацію основного макета сторінки та забезпечує логічне розділення інтерфейсу на окремі функціональні частини.

У межах компоненти site-home реалізовано відображення таких підкомпонент, як site-nav, site-content та site-footer. Для інтеграції цих елементів інтерфейсу використовується директива router-outlet, що забезпечує динамічне завантаження відповідного вмісту залежно від маршруту, обраного користувачем.

Компонента site-product-list призначена для відображення переліку товарів одягу, отриманих із серверної частини веб-додатку. Вона взаємодіє з відповідним сервісом, який надсилає GET-запит до REST API. Після успішного отримання масиву товарів, компонента здійснює їх візуалізацію у зручному для користувача форматі.

Компонента site-product реалізує отримання з серверної частини даних про конкретний товар, який було обрано користувачем. Після взаємодії з REST API за унікальним ідентифікатором товару, компонента здійснює запит до відповідної таблиці серверу. Отримані дані містять детальну інформацію про товар та відображає її у компоненті.

Інтерфейс admin-panel.clothes.interface використовується для визначення структури об’єкта, що містить дані про одяг. Цей інтерфейс експортується до компоненти site-content з метою чіткої типізації отриманих даних від серверної частини.

3.1.2 CRUD-операції

Як було раніше описано, що сервіс AdminPanelService виконує функції взаємодії з REST API серверної частини. Цей сервіс відповідає за виконання основних CRUD-операцій (створення, зчитування, оновлення, видалення) для керування даними адміністративної панелі та отримання даних клієнтської частиною.

Він імпортується як сервіс з декоратором @Injectable та реєструється у кореневому модулі додатка за допомогою { providedIn: 'root' }, що забезпечує його

доступність упродовж усього життєвого циклу застосунку.

```
@Injectable({
  providedIn: 'root'
});
```

У тілі класу сервісу використовується Angular-сервіс HttpClient, що надсилає HTTP-запити до зовнішнього сервера:

– *Метод fetchAllData().* Реалізує HTTP GET-запит для отримання всіх наявних даних із заданої адреси;

– *Метод createData().* HTTP POST-запит для створення нового запису;

– *Метод updateData().* HTTP PUT-запит для оновлення наявного об'єкта за його ідентифікатором;

– *Метод deleteData().* HTTP DELETE-запит для видалення ресурсу з бази даних за відповідним ідентифікатором.

```
fetchAllData(url: String):Observable<any> {
 return this._httpClient.get<any>(${url});
}
createData(data:any, url:String) {
 return this._httpClient.post<any>(${url}, data);
}
updateData(data:any, url: String) {
 return this._httpClient.put<any>(${url}/${data.id}, data);
}
deleteData(id: Number, url: String) {
 return this._httpClient.delete<any>(${url}/${id});
}
```

Кожен із методів приймає як параметр відповідний URL-адрес, що дозволяє повторно використовувати сервіс для різних REST-запитів. Це забезпечує

гнучкість та масштабованість при роботі з різними типами ресурсів.

Використання AdminPanelService дозволяє централізувати логіку взаємодії з серверому. Такий підхід є рекомендованим у межах архітектурної моделі Angular і сприяє побудові підтримуваного та розширюваного програмного забезпечення.

3.1.3 Інтерфейси

У проекті було створені окремі інтерфейси AdminsInterface, ClothesInterface та UsersInterface, які визначають структуру відповідних об'єктів: адміністраторів, товарів та користувачів. Ці інтерфейси реалізовані у відповідних файлах — admin-panel-admins.interface.ts, admin-panel-clothes.interface.ts та admin-panel-users.interface.ts. Вони слугують для формалізації типів даних у TypeScript та забезпечують статичну типізацію на етапі компіляції, що дозволяє уникати помилок під час обробки даних.

Інтерфейс AdminsInterface описує об’єкти адміністраторів, які містять такі поля, як id, login, password та token. Це дає змогу забезпечити типізовану роботу з обліковими записами адміністраторів у процесі автентифікації та авторизації доступу до адміністративної панелі.

```
export interface AdminsInterface {
 id:Number,
 login:String,
 password:String,
 token: String
}
```

Інтерфейс ClothesInterface описує дані, що стосуються товарів одягу. Він включає ідентифікатор, назву, ціну, рейтинг, опис, знижку, розмір, колір, посилання на зображення, а також кількість одиниць товару на складі. Завдяки цьому інтерфейсу компоненти, які відповідають за відображення або редагування товарів, мають чітке уявлення про формат і тип даних, які надходять із сервера або які передаються до нього.

```
export interface ClothesInterface {
 id:Number,
 name:String,
 prise:Number,
 rating:String,
 description:String,
 discount:Number,
 size:String,
 color:String,
 img:String,
 quantity:Number
}
```

Інтерфейс UsersInterface відповідає за структуру об'єктів зареєстрованих користувачів. Він включає поля id, login, password, name та поточний баланс рахунку balance, що дозволяє коректно обробляти дані користувачів на клієнтському рівні, зокрема під час автентифікації, перегляду особистої інформації або здійснення покупок.

```
export interface UsersInterface {
 id:Number,
 login:String,
 password:String,
 name:String,
 balance:Number
}
```

3.2 Серверна частина

У реалізації серверної частини веб-додатку використовується система управління базами даних PostgreSQL, яка виступає в ролі основного сервера зберігання даних. Взаємодія з базою даних здійснюється через набір таблиць, що відображають відповідні набори даних, а саме: товари, користувачів, адміністраторів тощо.

Для забезпечення обміну даними між клієнтською частиною та сервером використовується архітектурний підхід REST API. Основна логіка обробки запитів реалізована у файлі контролера, який виконує такі функції: приймає HTTP-запити, ініціює відповідні SQL-запити та формує відповіді для клієнта у форматі JSON. Контролер також виконує маршрутизацію запитів та викликає необхідні методи залежно від типу операції: створення, зчитування, оновлення або видалення даних.

Підключення до PostgreSQL здійснюється через спеціальний файл, де описано параметри з’єднання з базою даних, а саме: хост, порт, ім’я бази даних, користувач і пароль. Також ініціалізовано екземпляр клієнта для виконання SQL-запитів. Усі запити до бази даних виконуються з використанням мови SQL.

3.2.1 Опис таблиць

У базі даних реалізовано три незалежні таблиці: admins, users та clothes. Вони не мають зовнішніх ключів, тобто не пов’язані між собою безпосередньо. Кожна таблиця виконує окрему функцію в рамках загальної системи та містить поля з чітко визначеними типами даних.

Тип integer використовується для зберігання цілих чисел у межах від -2 147 483 648 до 2 147 483 647. Цей тип даних займає 4 байти пам’яті та є оптимальним варіантом для полів, у яких зберігаються лічильники, ідентифікатори, кількість товарів тощо.

Тип character varying або varchar використовується для зберігання рядків символів довільної довжини.

Таблиця 3.1 – Таблиця даних admins

Поле	Тип даних	Опис
id	integer	Ідентифікатор
login	character varying	Логін адміністратора
password	character varying	Пароль для авторизації
token	character varying	Токен для авторизації

Таблиця 3.2 – Таблиця даних users

Поле	Тип даних	Опис
id	integer	Ідентифікатор
login	character varying	Логін користувача
password	character varying	Пароль для авторизації
name	character varying	Ім’я користувача
balance	character varying	Баланс користувача
token	character varying	Токен для авторизації

Таблиця 3.3 – Таблиця даних clothes

Поле	Тип даних	Опис
id	integer	Ідентифікатор товару
name	character varying	Назва товару
price	integer	Ціна товару
rating	character varying	Рейтинг товару
description	character varying	Опис товару
discount	integer	Знижка товару
size	character varying	Розмір одягу
color	character varying	Колір товару
img	character varying	Посилання на зображення товару
quantity	integer	Кількість товару

3.2.2 Під’єднання до бази даних

У проекті використовується файл db.js, який виконує ключову роль у забезпеченні взаємодії між серверною частиною веб-додатку та базою даних PostgreSQL. У цьому файлі реалізовано логіку підключення до бази даних шляхом використання модуля pg(PostgreSQL client for Node.js) та створення пулу з’єднань (connection pool).

Імпортується клас Pool з модуля pg, який відповідає за створення пулу з’єднань з базою даних.

```
const pgSqlPool = require("pg").Pool;
```

Створюється екземпляр пулу, який містить конфігураційні параметри підключення до локальної бази даних PostgreSQL: ім’я користувача, пароль, назву бази, хост, порт, а також максимальну кількість одночасних з’єднань (max: 10).

```
const pool = new pgSqlPool({
   user: "postgres",
   password: "HogvardssPostgreSQL",
   database: "ShopCO",
   host: "localhost",
   port: "5432",
   max: 10
});
```

Встановлюється тестове з’єднання з базою даних з метою перевірки правильності підключення. У разі успішного з’єднання воно одразу закривається через метод connection.release(), а в консолі виводиться повідомлення про успішне підключення. У разі помилки — ініціюється виключення.

```
pool.connect((err, connection) => {
    if (err) throw err;
    console.log("Connected to PosgreSQL DB successfull!");
```

```
    connection.release();
})
```

Експортується об'єкт pool для подальшого використання в інших частинах серверного коду, а саме до файлів які відповідають за обробку запитів до бази даних.

```
module.exports = pool;
```

3.2.3 Rest API контролер

У проекті використовується файл controller.js, який виконує функцію контролера в архітектурі типу «клієнт-сервер» і забезпечує реалізацію REST API для взаємодії клієнтської частини веб-додатку з базою даних PostgreSQL. Цей файл написаний на основі фреймворку Node.js з використанням бібліотеки Express і дозволяє здійснювати базові CRUD-операції над обліковими записами користувачів.

Підключаються необхідні бібліотеки: Express для створення маршрутизатора, модуль db для роботи з базою даних, body-parser для обробки тіла HTTP-запитів, а також jsonwebtoken для потенційної реалізації механізмів аутентифікації.

```
const express = require("express");
const router = express.Router();
const db = require("../DB/db");
const bodyParser = require("body-parser");
const jwt = require("jsonwebtoken");
```

Налаштування middleware для автоматичного парсингу JSON і даних форми у вхідних запитах, що дозволяє працювати з полями тіла запиту як з об'єктами.

```
router.use(bodyParser.json());
router.use(bodyParser.urlencoded({ extended: true }));
```

GET-запит. Реалізує HTTP-запит типу GET, який повертає всі записи з таблиці admins у схемі acc. Цей запит дозволяє клієнтській частині отримувати повний перелік користувачів.

```
router.get("/api/v1/admins", (req, res) => {
    const sql = "Select * from acc.admins";
    db.query(sql, (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(result.rows);
    })
});
```

POST-запит. Метод POST дозволяє додати новий запис до бази даних, зокрема створити нового адміністратора. Для цього використовуються параметри з тіла запиту — логін і пароль.

```
router.post("/api/v1/admins", (req, res) => {
    const {login, password} = req.body;
    const sql = "insert into acc.admins(login, password)
    VALUES ($1,$2) RETURNING *";
    db.query(sql, [login, password], (err, result) => {
        if (err) throw res.json(err);
        return res.status(201).json(result.rows);
    })
});
```

PUT-запит. Метод PUT дозволяє оновити інформацію про користувача за вказаним ідентифікатором id. Запит змінює відповідні поля логін і пароль у таблиці account.

```
router.put("/api/v1/admins/:id", (req, res) => {
    const accId = Number(req.params.id);
```

```
    const {login, password} = req.body;
    const sql = "update acc.admins SET login=$1,password=$2
    Where id=$3";
    db.query(sql, [login, password, accId], (err, result) =>
    {
        if (err) throw res.json(err);
        return    res.status(200).json(Item    is    updated
        successfully for id: ${accId});
    })
});
```

DELETE-запит. Метод DELETE реалізує видалення користувача з бази даних за унікальним ідентифікатором. Даний запит дозволяє адміністратору або системі очищати базу від непотрібних записів.

```
router.delete("/api/v1/admins/:id", (req, res) => {
    const id = Number(req.params.id);
    const sql = "DELETE from acc.admins where id=$1";
    db.query(sql, [id], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is deleted
        successfully for id: ${id});
    })
});
```

Файл controller.js є необхідним елементом серверної логіки веб-додатку, оскільки:

- Забезпечує централізовану обробку HTTP-запитів від клієнтської частини;
- Реалізує інтерфейс REST API для виконання CRUD-операцій;
- Дозволяє ефективно керувати ресурсами бази даних;

– Забезпечує зручне масштабування коду та можливість розширення функціоналу (наприклад, додавання аутентифікації через JWT).

3.3 Панель адміністратора

Панель адміністратора є окремим функціональним модулем веб-додатку та слугує засобом керування даними системи. Основне призначення панелі адміністратора — забезпечити зручне адміністрування облікових записів адміністраторів, користувачів та товарів у системі проекта.

Опис функціоналу панелі адміністратора:

– *Авторизація адміністратора.* Панель адміністратора реалізована з обов’язковою процедурою авторизації. Адміністратор повинен пройти перевірку логіну й пароля, після чого отримає доступ до функціональних можливостей;

– *Керування адміністраторами.* У рамках інтерфейсу реалізовано можливість додавання нових адміністраторів до системи. Передбачено функції редагування облікових даних адміністраторів, а також їх видалення з бази даних при необхідності;

– *Керування користувачами*. Система дозволяє адміністраторам переглядати список зареєстрованих користувачів. Реалізовано функції створення, оновлення та видалення облікових записів користувачів, що дозволяє контролювати активність у системі;

– *Керування товарами.* Адміністратор має змогу додавати нові товари до бази даних, зазначаючи назву, ціну, розмір, знижку, колір, опис, рейтинг, кількість тощо. Також доступне редагування існуючих товарів і можливість їх повного видалення з каталогу.

3.3.1 Загальна схема панелі адміністратора

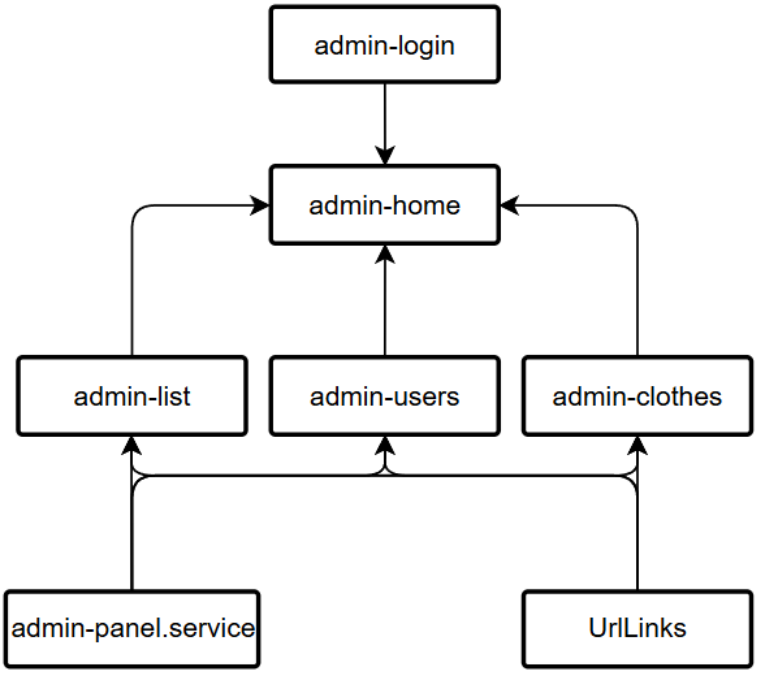


Рис. 3.2 – Загальна схема панелі адміністратора

Компонента admin-login відповідає за процес авторизації адміністратора для доступу до панелі керування. Авторизація відбувається шляхом заповнення спеціальної форми, що містить поля для введення логіна та пароля. Після введення облікових даних і підтвердження дій, інформація передається на сервер для верифікації.

На серверній частині здійснюється перевірка відповідності введених даних існуючим записам у базі даних PostgreSQL. У разі успішної автентифікації адміністратор отримує доступ до функціоналу адміністративної панелі. У протилежному випадку система повідомляє про помилку авторизації.

Компонента admin-home виступає основною домашньою сторінкою адміністративної панелі веб-додатку. Вона реалізує структуру інтерфейсу керування, до якої динамічно підключаються підкомпоненти admin-list, admin-users та admin-clothes. Завдяки використанню маршрутизації Angular, зазначені підкомпоненти завантажуються у визначену область компоненти admin-home

залежно від дій адміністратора та обраного пункту навігації.

Компонента admin-list виконує функцію управління адміністраторами в межах адміністративної панелі веб-додатку. Вона відповідає за відображення списку адміністраторів, дані про яких отримуються з серверної частини через REST API. Отримана інформація виводиться у структурованому вигляді та читабельному вигляді. Також дана компонента надає функціонал редагування даних адміністраторів.

Компонента admin-users виконує функцію управління користувачами у межах адміністративної панелі веб-додатку. Вона забезпечує отримання та відображення списку зареєстрованих користувачів, дані про яких надходять із серверної частини через REST API. Також у цій компоненті є можливість редагувати та видаляти користувачів.

Компонента admin-clothes виконує функцію управління асортиментом товарів одягу в адміністративній панелі веб-додатку. Вона забезпечує отримання даних із серверної частини через REST API та відображає повний перелік товарів, що зберігаються у базі даних PostgreSQL. Також у цій компоненті є можливість редагувати та видаляти відповідні товари.

3.3.2 CRUD операції

У компонентах панелі адміністратора реалізовано функціонал для виконання CRUD-операцій над відповідними даними. Такий підхід забезпечує повний цикл роботи з інформацією на стороні клієнта, дозволяючи отримувати актуальні дані з сервера, відображати їх користувачу, а також оперативно вносити зміни або видаляти непотрібні записи.

GET-запит. Метод ngAfterViewInit() є частиною життєвого циклу компонента Angular і викликається одразу після ініціалізації. У цьому методі за допомогою сервісу adminPanelService викликається метод fetchAllData, який формує запит до визначеного ресурсу за допомогою змінної this.baseAdminListURL і надсилає його на сервер.

Результат запиту обробляється в підписці subscribe(), де отримані дані

зберігаються в локальну змінну this.admins, яка використовується для відображення в інтерфейсі клієнтської частини.

```
ngAfterViewInit(){
      this.adminPanelService.fetchAllData(this.baseAdminLis
      tURL).subscribe((data) => {
         this.admins = data;
   })
  }
```

POST-запит. Використовується сервіс adminPanelService, в якому визначений метод createData, що приймає два аргументи: об'єкт admin, який містить інформацію про нового адміністратора, а саме: логін, пароль, токен, та змінну this.baseAdminListURL, яка містить URL-адресу ресурсу API, до якого надсилається запит.

```
this.adminPanelService.createData(admin,
     this.baseAdminListURL).subscribe({
       next: (data) => {
          console.log(New Admin created Successfully!);
          window.location.reload();
      },
         error: (err) => {
          console.log(err);
          window.location.reload();
      }
    })
```

Запит виконується за допомогою функції subscribe(), яка обробляє відповідь сервера у два блоки:

– *Next.* У разі успішного створення запису виводиться повідомлення про успіх

у консоль, після чого виконується перезавантаження сторінки, щоб відобразити оновлений список адміністраторів;

– *Error.* У разі виникнення помилки також виконується виведення інформації про неї в консоль та оновлення сторінки для збереження цілісності інтерфейсу.

PUT-запит. Для PUT-запиту використовується такий самий метод як і у POST-запиті, головна відмінність, що ми звертаємося до метода updateData, який знаходить користувача у базі даних та змінює йому значення колонок.

DELETE-запит. Функція deleteAdminFormField приймає як аргумент ідентифікатор адміністратора, якого необхідно видалити. Перед виконанням запиту користувачеві виводиться стандартне діалогове вікно підтвердження дії. У разі підтвердження дії, викликається метод deleteData() з сервісу adminPanelService, якому передаються відповідний id і базова URL-адреса ресурсу. Після успішного виконання запиту отриманий локальний масив адміністраторів оновлюється методом, що видаляє об'єкт з відповідним ідентифікатором з відображуваного списку, а сторінка перезавантажується, щоб візуально оновити дані інтерфейсу.

```
deleteAdminFormField(id: Number) {
    const  isConfirmed = window.confirm("Are you sure you
     want to DELETE?");
    if (isConfirmed) {
      this.adminPanelService.deleteData(id,
       this.baseAdminListURL).subscribe((data) => {
           this.admins = this.admins.filter(item => item.id
           !== id);
           window.location.reload();
      });
    }
```

3.3.3 Angular Material

У панелі адміністратора для відображення табличної інформації про

адміністраторів, користувачів або товари використовується бібліотека Angular Material, яка забезпечує зручні та сучасні UI-компоненти відповідно до принципів Material Design. Реалізовано використання компоненти MatTable, це дозволяє формувати динамічну таблицю з підтримкою сортування, пагінації та фільтрації.

Дані для таблиці підвантажуються з відповідного TypeScript-файлу компоненти, де вони отримуються через HTTP-запити з серверної частини та передаються в таблицю як джерело даних.

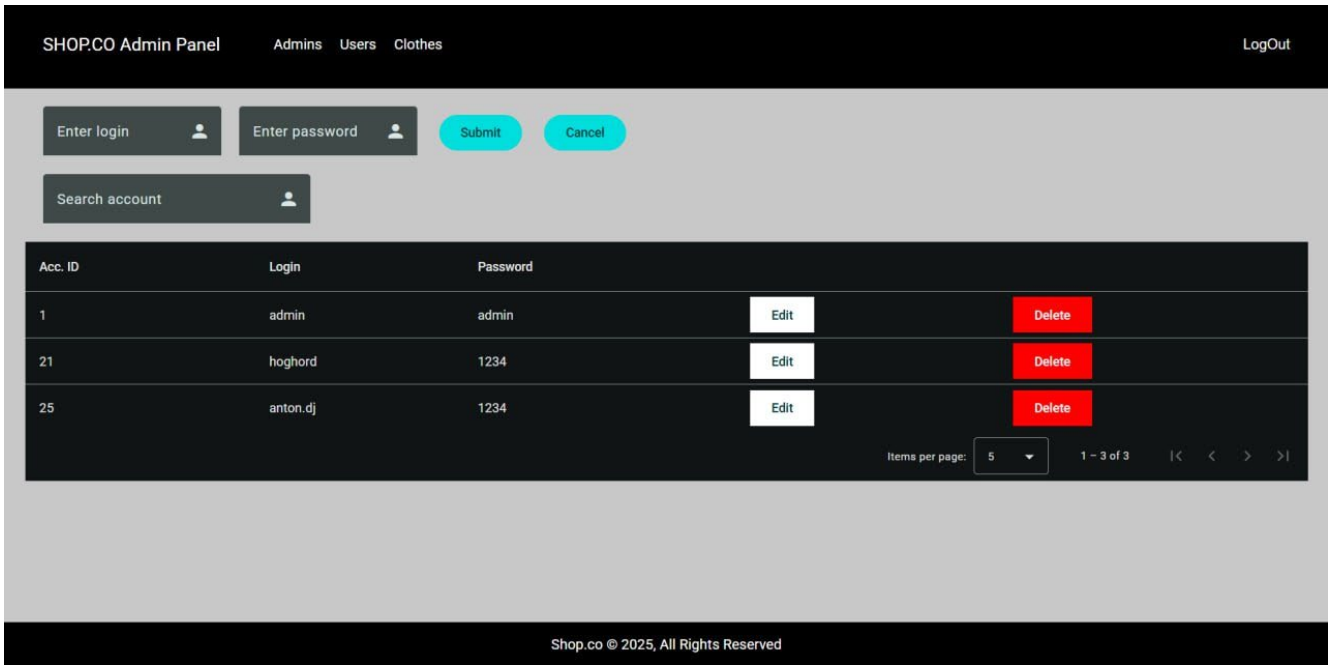


Рис. 3.3 – Побудована таблиця за допомогою Angular Material

На рисунку 3.3 видно відображену таблицю у якій реалізовано пагінацію, а саме праворуч унизу. Для цього було використано компонент MatPaginator з бібліотеки Angular Material, який інтегрується з MatTable через прив’язку @ViewChild у TypeScript-коді. Пагінація надає користувачеві змогу вибирати кількість записів, що відображаються на одній сторінці, а також перемикатися між сторінками даних.

Також на рисунку 3.3 можна побачити фільтрацію по запиту, а саме ліворуч угорі. Для реалізації фільтрації використано вбудовані можливості MatTableDataSource. У компоненті передбачено поле вводу, пов’язане з методом фільтрації, що реагує на зміну значення та автоматично оновлює вміст таблиці.

Введене значення обробляється і після чого застосовується до джерела даних таблиці.

ВИСНОВКИ

У межах кваліфікаційної роботи було виконано повноцінне дослідження, проектування та реалізацію сучасного веб-додатку з використанням архітектури клієнт–сервер. Основною метою проекту було створення функціонального веб-рішення з адміністративною панеллю для управління даними користувачів, адміністраторів і товарів. У процесі розробки було детально проаналізовано технологічні та архітектурні особливості сучасних веб-додатків, розглянуто основні методології створення безпечної взаємодії між клієнтською частиною та сервером, а також обґрунтовано доцільність використання певних інструментів та технологій.

У першому розділі було представлено теоретичні засади побудови веб-додатків, серед яких розглянуто переваги й відмінності веб-архітектури, класифікацію веб-додатків, принципи клієнт-серверної архітектури, роль серверної частини, а також особливості використання реляційних і об’єктно-реляційних систем управління базами даних. Особливу увагу було приділено PostgreSQL — потужній, надійній і відкритій СУБД, що стала основною платформою для зберігання даних у розробленому проекті.

У другому розділі було проаналізовано реальні потреби користувачів, окреслено проблему, яку вирішує веб-додаток, та сформовано вимоги до функціональності, продуктивності, безпеки та зручності використання системи. Було розглянуто аналогічні системи на ринку, що дозволило зробити обґрунтований вибір стеку технологій, який охоплює: Angular як фронтенд-фреймворк, Node.js з Express для побудови серверної логіки, PostgreSQL як СУБД, а також REST API як метод взаємодії між компонентами.

У третьому розділі безпосередньо реалізовано веб-додаток. Розроблено клієнтську частину з використанням Angular, яка дозволяє ефективно взаємодіяти з сервером, здійснювати авторизацію, надсилати та отримувати дані, а також відображати інформацію у зручному вигляді. Було реалізовано CRUD-операції:

створення, читання, оновлення, видалення, що дозволяє адміністраторам керувати сутностями системи.

Панель адміністратора створено як окремий інтерфейс, що містить декілька взаємопов’язаних компонентів. Авторизація адміністраторів виконується через токени за допомогою JWT, що забезпечує безпечний доступ до системи. Всі дані отримуються з бази даних PostgreSQL через REST API, реалізоване на сервері засобами Express.

В рамках реалізації клієнтської частини особливу увагу приділено UI/UX. Для цього було інтегровано Angular Material, який дозволив створити адаптивну та сучасну таблицю для відображення даних з підтримкою сортування, фільтрації, пагінації та модального редагування. Такий підхід значно покращує зручність роботи адміністратора з великим обсягом інформації.

Кодова база сервера дозволяє обробляти HTTP-запити відповідно до методів GET, POST, PUT, DELETE, забезпечуючи повноцінний набір REST-операцій. Комунікація з базою даних організована через пул підключень PostgreSQL, що гарантує стабільність, масштабованість і відмовостійкість роботи сервера.

У процесі реалізації кваліфікаційної роботи були досягнуті всі поставлені цілі. Розроблено повноцінний веб-додаток із адміністративною панеллю, налагоджено інтеграцію з базою даних, забезпечено надійне керування користувачами, адміністраторами та товарами. Використання сучасного технологічного стеку забезпечує гнучкість і можливість подальшого масштабування розробленого рішення.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Angular. Home • Angular. URL: https://angular.dev/overview (дата звернення: 24.02.2025).
2. PostgreSQL: Documentation. PostgreSQL: The world's most advanced open source database. URL: https://www.postgresql.org/docs/ (дата звернення: 24.02.2025).
3. About this documentation | Node.js v24.0.2 Documentation. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org/api/documentation.html (дата звернення: 24.02.2025).
4. Express - Node.js web application framework. Express - Node.js web application framework. URL: https://expressjs.com/ (дата звернення: 24.02.2025).
5. RxJS. RxJS. URL: https://rxjs-dev.firebaseapp.com/ (дата звернення: 24.02.2025).
6. Angular. Angular Material. Angular Material. URL: https://material.angular.dev/ (дата звернення: 25.02.2025).
7. Documentation. pgAdmin - PostgreSQL Tools. URL: https://www.pgadmin.org/docs/ (дата звернення: 25.02.2025).
8. Learn REST API Design - REST API Tutorial. REST API Tutorial. URL: https://www.restapitutorial.com/ (дата звернення: 25.02.2025).
9. What is Agile Software Development?. Agile Alliance |. URL: https://www.agilealliance.org/agile101/ (дата звернення: 25.02.2025).
10. Home | Scrum Guides. Home | Scrum Guides. URL: https://scrumguides.org/ (дата звернення: 25.02.2025).
11. JSON Web Token Introduction - jwt.io. JSON Web Tokens - jwt.io. URL: https://jwt.io/introduction (дата звернення: 26.02.2025).
12. PostgreSQL Tutorial. Tutorials on Technical and Non Technical Subjects. URL: https://www.tutorialspoint.com/postgresql/index.htm (дата звернення: 26.02.2025).
13. W3Schools.com. W3Schools Online Web Tutorials. URL:

https://www.w3schools.com (дата звернення: 26.02.2025).

14. GeeksforGeeks. Express.js Tutorial - GeeksforGeeks. GeeksforGeeks. URL: https://www.geeksforgeeks.org/express-js/ (дата звернення: 26.02.2025).

15. What is REST?: REST API Tutorial. REST API Tutorial. URL: https://restfulapi.net/ (дата звернення: 27.02.2025).

16. OWASP Top Ten | OWASP Foundation. OWASP Foundation, the Open Source Foundation for Application Security | OWASP Foundation. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 27.02.2025).

17. Optimize CSS Delivery | PageSpeed Insights | Google for Developers. Google for Developers. URL: https://developers.google.com/speed/docs/insights/OptimizeCSSDelivery (дата звернення: 27.02.2025).

18. Types of SQL Commands: DDL, DML, DQL, DCL, and TCL SitePoint. SitePoint Learn HTML, CSS, JavaScript, PHP, Ruby & Responsive Design. URL: https://www.sitepoint.com/sql-commands/ (дата звернення: 27.02.2025).

19. Git. Git. URL: https://git-scm.com/book/en/v2 (дата звернення: 27.02.2025).

20. Continuous integration vs. delivery vs. deployment | Atlassian. Atlassian. URL: https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment (дата звернення: 28.02.2025).

21. What is Cloud Computing? - Cloud Computing Services, Benefits, and Types - AWS. Amazon Web Services, Inc. URL: https://aws.amazon.com/what-is-cloud-computing/ (дата звернення: 28.02.2025).

22. Progressive Web Apps | web.dev. web.dev. URL: https://web.dev/explore/progressive-web-apps (дата звернення: 28.02.2025).

23. WCAG 2 Overview. Web Accessibility Initiative (WAI). URL: https://www.w3.org/WAI/standards-guidelines/wcag/ (дата звернення: 28.02.2025

Додаток А

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація)

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ



Навчально-науковий інститут
Інформаційних технологій
Кафедра Комп'ютерної інженерії

КВАЛІФІКАЦІЙНА РОБОТА
НА ЗДОБУТТЯ ОСВІТНЬОГО СТУПЕНЯ БАКАЛАВРА
на тему: «Веб-сайт з інтеграцією PostgreSQL для адміністрування серверних ресурсів»

Виконав студент групи КІД-41
Матвієнко Володимир Дмитрович
Керівник кваліфікаційної роботи:
Олександр Зуб Вікторович, асистент
кафедри Комп'ютерної інженерії

Київ – 2025

Мета роботи – розробити веб-сайт з інтеграцією PostgreSQL для адміністрування серверних ресурсів

Об`єкт дослідження – розробка веб-сайту з серверною частиною PostgreSQL та клієнтською частиною Angular

Предмет дослідження – веб-сайт, серверна частина PostgreSQL та клієнтська частина Angular

Актуальність обраної теми

У сучасному цифровому середовищі веб-додатки відіграють ключову роль у наданні інтерактивних сервісів користувачам у реальному часі. Переважна більшість інформаційних систем, що використовуються у бізнесі, освіті, державному управлінні та повсякденному житті, реалізовані саме у форматі веб-додатків. Їх популярність пояснюється зручністю доступу через браузер, відсутністю потреби в установці, підтримкою різних пристроїв та можливістю швидкого масштабування.

Окреме значення має ефективна організація серверної частини веб-додатку, адже саме вона відповідає за зберігання, обробку та захист даних, взаємодію з базою даних, авторизацію і реалізацію бізнес-логіки. Використання надійних систем управління базами даних, таких як PostgreSQL, дозволяє створювати стійкі, масштабовані та безпечні рішення, що відповідають сучасним вимогам.

Функціональні та нефункціональні вимоги веб-додатка

Функціональні вимоги

- Реєстрація та авторизація;
- Адміністративна панель керування;
- Відображення інформації у вигляді таблиці;
- Збереження та обробка даних у базі даних PostgreSQL;
- Інтеграція з REST API;
- Реактивність та зручність інтерфейсу.

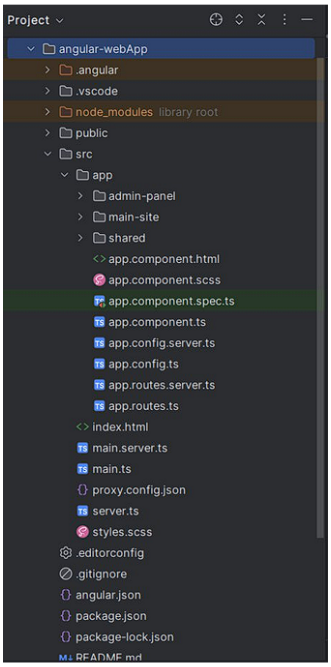
Нефункціональні вимоги

- Продуктивність;
- Масштабованість;
- Надійність;
- Безпека;
- Узгодженість даних;
- Зручність користування;
- Сумісність;
- Підтримка та супровід.

Вибір мови програмування, фреймворків та бібліотек

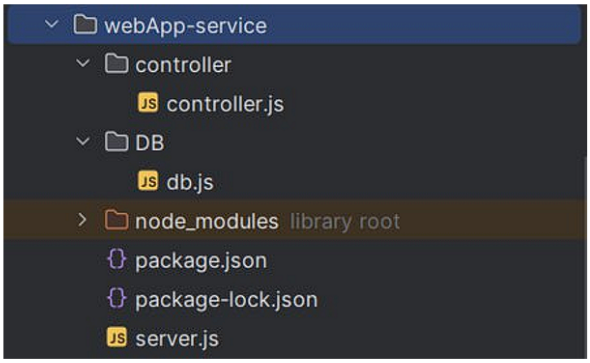
Мови програмування: JavaScript, TypeScript
Мови верстки(розмітка, стилі): HTML, SCSS
Додаткові технології: JWT, body-parser
Засіб керування проектом: Node.JS
Фреймворк: Angular, Express
СУБД: PostgreSQL

Структура клієнтської частини



public – зберігає картинки
admin-panel – компоненти панелі адміністратора
app.component – головна проектна компонента
app.routes – маршрутизація
main-site – компоненти клієнтської частини
shared – типізація, API-посилання, обмеження шляхів, обробка та перевірка даних.

Структура серверної логіки

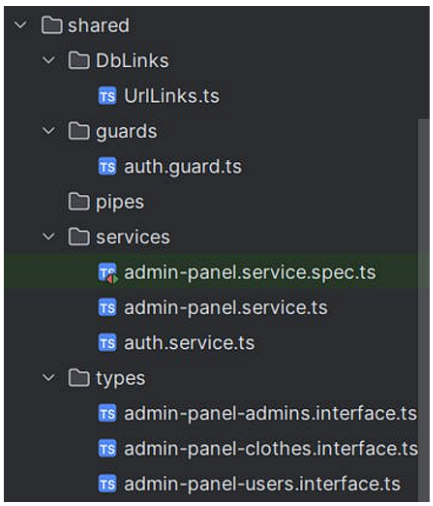


server.js – запуск серверної логіки

controller.js – реалізація REST API

db.js – підключення до бази даних

Допоміжні технології серверної та клієнтської логіки



UrlLinks.ts – API посилання

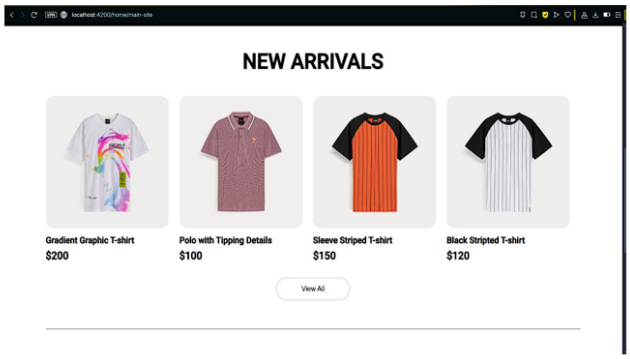
auth.guard.ts – обмеження шляхів

admin-panel.service.ts – CRUD операції

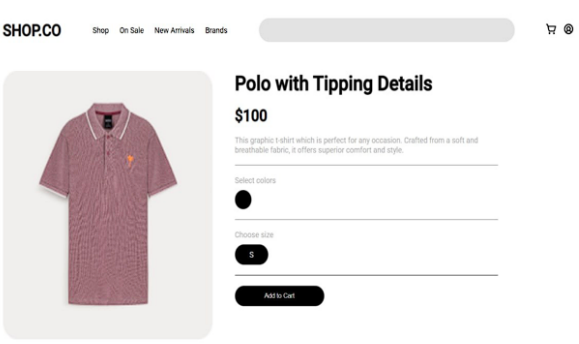
auth.service.ts – перевірка токена авторизації

admin-panel.interface.ts – типізація серверних даних

Приклад UI/UX дизайну клієнтської частини

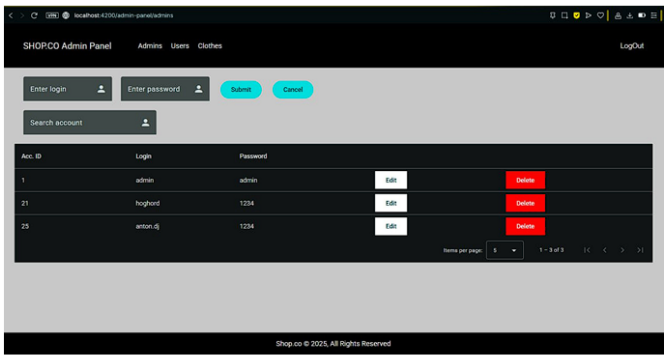


Список товарів



Сторінка товару

UI/UX дизайн панелі адміністратора

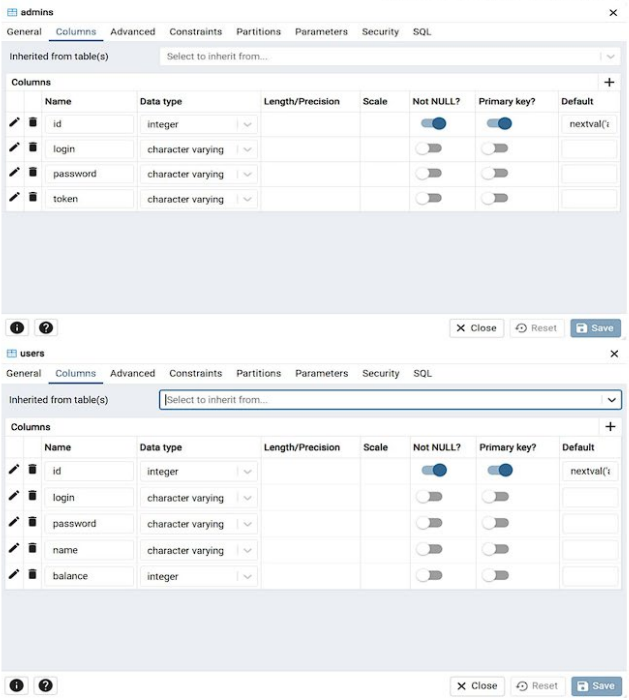


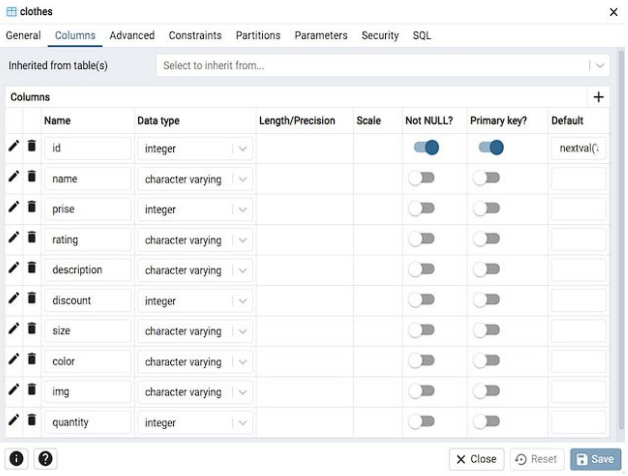
Дизайн панелі



Авторизація адміністратора

Таблиці бази даних PosgreSQL





REST API запити

```
//Put method
router.put("/api/v1/admins/:id", (req, res) => {  Volodymyr
    const accId = Number(req.params.id);
    const {login, password} = req.body;
    const sql = "update acc.admins SET login=$1,password=$2 Where id=$3";

    db.query(sql, [login, password, accId], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(`Item is updated successfully for id: ${accId}`);
    })
});
```

PUT метод

```
//Post method
router.post("/api/v1/admins", (req, res) => {  Volodymyr
    const {login, password} = req.body;
    const sql = "insert into acc.admins(login, password) VALUES ($1,$2) RETURNING *";

    db.query(sql, [login, password], (err, result) => {
        if (err) throw res.json(err);

        const user = result.rows[0];

        const token = jwt.sign(
            payload: { id: user.id, login: user.login }, // payload
            secretOrPrivateKey: "u92Dj#4r@9dkLS!92mdkqPz6vG!ks1Nd"
        );

        return res.status(201).json(result.rows);
    })
});
```

POST метод

```
//Delete method
router.delete("/api/v1/admins/:id", (req, res) => {  Volodymyr
    const id = Number(req.params.id);
    const sql = "DELETE from acc.admins where id=$1";

    db.query(sql, [id], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(`Item is deleted successfully for id: ${id}`);
    })
});
```

DELETE метод

```
//Get method
router.get("/api/v1/admins", (req, res) => {  Volodymyr
    const sql = "Select * from acc.admins";

    db.query(sql, (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(result.rows);
    })
});
```

GET метод

CRUD операції

```
fetchAllData(url: String):Observable<any> {
  return this._httpClient.get<any>(${url});
}
```

Отримання усіх даних з таблиці БД

```
fetchParticularData(id: Number, url: String):Observable<any> {
  return this._httpClient.get<any>(${url}/${id});
}
```

Отримання конкретного масиву даних з БД

```
createData(data:any, url:String) {
  return this._httpClient.post<any>(${url}, data);
}
```

Створення нових даних у БД

```
updateData(data:any, url: String) {
  return this._httpClient.put<any>(${url}/${data.id}, data);
}
```

Оновлення даних у БД

```
deleteData(id: Number, url: String) {
  return this._httpClient.delete<any>(${url}/${id});
}
```

Видалення даних з БД

ВИСНОВКИ

У кваліфікаційній роботи було реалізовано повноцінний веб-додаток, що поєднує технології клієнтської та серверної частини. Основною метою розробки було створення зручного, функціонального та масштабованого інструменту з можливістю керування даними через інтерфейс адміністратора.

З клієнтської сторони застосовано фреймворк Angular, який забезпечує високу інтерактивність, підтримку компонентного підходу, а також ефективну роботу з API через сервіси. Завдяки використанню Angular Material реалізовано адаптивний інтерфейс із зручною таблицею, що підтримує пагінацію, фільтрацію та CRUD-операції. Впровадження guard-механізмів та інтерфейсів дозволило гарантувати контроль доступу до окремих частин додатку та забезпечити типову безпеку й стабільність коду.

Серверна частина реалізована з використанням Node.js і бібліотеки Express, що забезпечило гнучке управління маршрутизацією та REST API. Взаємодія із базою даних організована через PostgreSQL, яка гарантує збереження, цілісність і консистентність даних. Уся логіка серверної частини була структурована в окремому контролері, що підвищує масштабованість.

ПРОДОВЖЕННЯ ВИСНОВКІВ

Було визначено функціональні та нефункціональні вимоги до системи, проаналізовано наявні технології та сформовано оптимальну архітектуру проекту. Особливу увагу було приділено безпеці, структурі коду, ефективності взаємодії між клієнтом і сервером, а також зручності користування.

Результатом виконаної розробки є стабільний веб-додаток, що може бути легко масштабований, доповнений новими функціями, адаптований під мобільні пристрої або інтегрований з іншими сервісами. Отриманий програмний продукт демонструє практичне втілення сучасних методологій розробки веб-систем і є придатним до подальшого розвитку та використання в реальних умовах.

РЕЗУЛЬТАТИ ВПРОВАДЖЕННЯ

1. Матвієко В.Д., Джух А.В. Майбутнє за віртуальною та доповненою реальністю «Світ інформацій та телекомунікацій» : Міжнародна науково-технічна конференція студентства та молоді, м. Київ, 21 жовтня 2021 р. С. 174–176. URL: https://duikt.edu.ua/uploads/p_2626_33497568.pdf.
2. Зуб О.В., Матвієнко В.Д. Використання штучного інтелекту у відеоіграх «Сучасні інтелектуальні інформаційні технології в науці та освіті» : V Всеукраїнській науково-практ. конф., м. Київ, 15 травня 2025 р.
Подано до друку.

Додаток Б

ЛІСТИНГИ ОСНОВНИХ КОМПОНЕНТ

Б.1. Компонента admin-clothes

```
import {AfterViewInit, Component, ViewChild} from
'@angular/core';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatInputModule} from '@angular/material/input';
import {MatButtonModule} from '@angular/material/button';
import {MatIconModule} from '@angular/material/icon';
import {MatTableDataSource, MatTableModule} from
'@angular/material/table';
import {AdminPanelService} from '../../shared/services/admin-
panel.service';

import {MatSort, MatSortModule} from '@angular/material/sort';
import {CommonModule} from '@angular/common';
import {FormsModule} from '@angular/forms';
import {MatPaginator, MatPaginatorModule} from
'@angular/material/paginator';
import {adminPanelClothesURL} from
'../../shared/DbLinks/UrlLinks';
import {ClothesInterface} from '../../shared/types/admin-panel-
clothes.interface';

@Component({
  selector: 'app-admin-clothes',
  imports: [MatFormFieldModule, MatInputModule, MatButtonModule,
MatIconModule, MatTableModule,
    MatSortModule, MatPaginatorModule, CommonModule,
FormsModule],
  templateUrl: './admin-clothes.component.html',
  standalone: true,
  styleUrl: './admin-clothes.component.scss'
})
export class AdminClothesComponent implements AfterViewInit{
  displayedColumns: string[] = ["id", "name", "prise", "rating",
"description", "discount", "size", "color", "img", "quantity",
"edit", "delete"];
  dataSource = new MatTableDataSource<ClothesInterface>();

  baseAdminClothesURL = adminPanelClothesURL;

  constructor(private adminPanelService: AdminPanelService) {}

  @ViewChild(MatSort) sort: any;
```

```
  @ViewChild(MatPaginator) paginator: any;

  cloth:ClothesInterface = {
    id: 0,
    name: "",
    prise: 0,
    rating: "",
    description: "",
    discount: 0,
    size: "",
    color: "",
    img: "",
    quantity: 0
  }

  clothes:ClothesInterface[] = [];
  filteredClothes:ClothesInterface[] = [];

  ngAfterViewInit() {
    this.getClothes();

    setTimeout(() => {
      this.dataSource.paginator = this.paginator;
      this.dataSource.sort = this.sort;
    }, 100);
  }

  getClothes() {

this.adminPanelService.fetchAllData(this.baseAdminClothesURL).su
bscribe((data) => {
      this.clothes = data;
      this.dataSource = new
MatTableDataSource<ClothesInterface>(data);
    })
  }

  searchClothes(input: any) {
    this.filteredClothes = this.clothes.filter(item =>
      item.name.toLowerCase().includes(input.toLowerCase())
      item.rating.toLowerCase().includes(input.toLowerCase())

item.description.toLowerCase().includes(input.toLowerCase())
```

```
       item.size.toLowerCase().includes(input.toLowerCase())
      item.id.toString().includes(input)
    );

    this.dataSource = new
MatTableDataSource<ClothesInterface>(this.filteredClothes);
    this.dataSource.paginator = this.paginator;
    this.dataSource.sort = this.sort;
  }

  PostPutClothes(cloth:ClothesInterface) {
    if (cloth.id !== 0) {
      this.adminPanelService.updateData(cloth,
this.baseAdminClothesURL).subscribe({
        next: (data) => {
          console.log(Admin updated Successfully!);
          this.getClothes();
        },
        error: (err) => {
          console.log(err);
          this.getClothes();
        }
      })
    } else {
      this.adminPanelService.createData(cloth,
this.baseAdminClothesURL).subscribe({
        next: (data) => {
          console.log(New Admin created Successfully!);
          this.getClothes();
        },
        error: (err) => {
          console.log(err);
          this.getClothes();
        }
      })
    }
  }

  putClothesFormField(cloth: ClothesInterface) {
    this.cloth.id = cloth.id;
    this.cloth.name = cloth.name;
    this.cloth.prise = cloth.prise;
    this.cloth.rating = cloth.rating;
```

```
    this.cloth.description = cloth.description;
    this.cloth.discount = cloth.discount;
    this.cloth.size = cloth.size;
    this.cloth.color = cloth.color;
    this.cloth.img = cloth.img;
    this.cloth.quantity = cloth.quantity;
  }

  deleteClothesFormField(id: Number) {
    const  isConfirmed = window.confirm("Are you sure you want
to DELETE?");
  if (isConfirmed) {
      this.adminPanelService.deleteData(id,
this.baseAdminClothesURL).subscribe((data) => {
        this.clothes = this.clothes.filter(item => item.id !==
id);

        this.getClothes();
      });
    }
  }
}
```

Б.2. Компонента admin-home

```
import {Component, inject} from '@angular/core';
import {Router, RouterLink, RouterLinkActive, RouterOutlet} from
'@angular/router';

@Component({
  selector: 'app-admin-home',
  imports: [RouterOutlet, RouterLink, RouterLinkActive],
  templateUrl: './admin-home.component.html',
  standalone: true,
  styleUrl: './admin-home.component.scss'
})
export class AdminHomeComponent {

  router = inject(Router);

  logOut(): void{
    console.log("logOut");
    localStorage.setItem("token", "");
```

```
    this.router.navigateByUrl("/admin-panel/login");
  }
}
```

Б.3. Компонента admin-list

```
import {AfterViewInit, Component, ViewChild} from
'@angular/core';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatInputModule} from '@angular/material/input';
import {MatButtonModule} from '@angular/material/button';
import {MatIconModule} from '@angular/material/icon';
import {MatTableDataSource, MatTableModule} from
'@angular/material/table';
import {AdminPanelService} from '../../shared/services/admin-
panel.service';

import {MatSort, MatSortModule} from '@angular/material/sort';
import {CommonModule} from '@angular/common';
import {FormsModule} from '@angular/forms';
import {MatPaginator, MatPaginatorModule} from
'@angular/material/paginator';
import {adminPanelListURL} from '../../shared/DbLinks/UrlLinks';
import {AdminsInterface} from '../../shared/types/admin-panel-
admins.interface';

@Component({
  selector: 'app-admin-list',
  imports: [MatFormFieldModule, MatInputModule, MatButtonModule,
MatIconModule, MatTableModule,
    MatSortModule, MatPaginatorModule, CommonModule,
FormsModule],
  templateUrl: './admin-list.component.html',
  standalone: true,
  styleUrl: './admin-list.component.scss'
})
export class AdminListComponent implements AfterViewInit{
  displayedColumns: string[] = ["id", "login", "password",
"edit", "delete"];
  dataSource = new MatTableDataSource<AdminsInterface>();

  baseAdminListURL = adminPanelListURL;
```

```
  constructor(private adminPanelService: AdminPanelService) {}

  @ViewChild(MatSort) sort: any;
  @ViewChild(MatPaginator) paginator: any;

  admin:AdminsInterface = {
    id: 0,
    login: "",
    password: "",
    token: ""
  }

  admins:AdminsInterface[] = [];
  filteredAdmins:AdminsInterface[] = [];

  ngAfterViewInit() {
    this.getAdmins();

    setTimeout(() => {
      this.dataSource.paginator = this.paginator;
      this.dataSource.sort = this.sort;
    }, 100);
  }

  getAdmins() {

this.adminPanelService.fetchAllData(this.baseAdminListURL).subsc
ribe((data) => {
      this.admins = data;
      this.dataSource = new
MatTableDataSource<AdminsInterface>(data);
    })
  }

  searchAdmin(input: any) {
    this.filteredAdmins = this.admins.filter(item =>
      item.login.toLowerCase().includes(input.toLowerCase()) ||
      item.id.toString().includes(input)
    );

    this.dataSource = new
MatTableDataSource<AdminsInterface>(this.filteredAdmins);
```

```
    this.dataSource.paginator = this.paginator;
    this.dataSource.sort = this.sort;
  }

  PostPutAdmin(admin:AdminsInterface) {
    if (admin.id !== 0) {
      this.adminPanelService.updateData(admin,
this.baseAdminListURL).subscribe({
        next: (data) => {
          console.log(Admin updated Successfully!);
          this.getAdmins();
        },
        error: (err) => {
          console.log(err);
          this.getAdmins();
        }
      })
    } else {
      this.adminPanelService.createData(admin,
this.baseAdminListURL).subscribe({
        next: (data) => {
          console.log(New Admin created Successfully!);
          this.getAdmins();
        },
        error: (err) => {
          console.log(err);
          this.getAdmins();
        }
      })
    }
  }

  putAdminFormField(admin: AdminsInterface) {
    this.admin.id = admin.id;
    this.admin.login = admin.login;
    this.admin.password = admin.password;
  }

  deleteAdminFormField(id: Number) {
    const  isConfirmed = window.confirm("Are you sure you want
to DELETE?");

    if (isConfirmed) {
```

```
      this.adminPanelService.deleteData(id,
this.baseAdminListURL).subscribe((data) => {
        this.admins = this.admins.filter(item => item.id !==
id);

        this.getAdmins();
      });
    }
  }
}
```

Б.4. Компонента admin-login

```
import {Component, inject} from '@angular/core';
import {FormBuilder, FormsModule, ReactiveFormsModule,
Validators} from '@angular/forms';
import {AuthService} from '../../shared/services/auth.service';
import {Router} from '@angular/router';
import {AdminsInterface} from '../../shared/types/admin-panel-
admins.interface';
import {AdminPanelService} from '../../shared/services/admin-
panel.service';
import {adminPanelLoginURL} from
'../../shared/DbLinks/UrlLinks';

@Component({
  selector: 'app-admin-login',
  imports: [
    FormsModule,
    ReactiveFormsModule
  ],
  templateUrl: './admin-login.component.html',
  standalone: true,
  styleUrl: './admin-login.component.scss'
})
export class AdminLoginComponent {
  fb = inject(FormBuilder);
  authService = inject(AuthService);
  router = inject(Router);

  baseAdminLoginURL = adminPanelLoginURL;

  constructor(private adminPanelService: AdminPanelService) {}
```

```
  form = this.fb.nonNullable.group({
    login: ['', Validators.required],
    password: ['', Validators.required]
  });

  requestData:AdminsInterface = {
    id: 0,
    login: "",
    password: "",
    token: ""
  }

  onSubmit(): void {

    this.requestData.login = this.form.getRawValue().login;
    this.requestData.password =
this.form.getRawValue().password;

    this.adminPanelService.createData(this.requestData,
this.baseAdminLoginURL).subscribe({
      next: (response) => {
        console.log("response: ", response);
        localStorage.setItem("token", response.admin.token);
        this.authService.currentAdminSig.set(response.admin);
        this.router.navigateByUrl("/admin-panel");
        alert("You Successfully Loged In!")
      },
      error: (err) => {
        console.log(err);
        alert(err.error.message);
      }
    })
  }
}
```

Б.5. Компонента admin-users

```
import {AfterViewInit, Component, ViewChild} from
'@angular/core';
import {MatFormFieldModule} from '@angular/material/form-field';
import {MatInputModule} from '@angular/material/input';
import {MatButtonModule} from '@angular/material/button';
```

```
import {MatIconModule} from '@angular/material/icon';
import {MatTableDataSource, MatTableModule} from
'@angular/material/table';
import {MatSort, MatSortModule} from '@angular/material/sort';
import {MatPaginator, MatPaginatorModule} from
'@angular/material/paginator';
import {CommonModule} from '@angular/common';
import {FormsModule} from '@angular/forms';

import {adminPanelUsersURL} from
'../../shared/DbLinks/UrlLinks';
import {AdminPanelService} from '../../shared/services/admin-
panel.service';
import {UsersInterface} from '../../shared/types/admin-panel-
users.interface';

@Component({
  selector: 'app-admin-users',
  imports: [MatFormFieldModule, MatInputModule, MatButtonModule,
MatIconModule, MatTableModule,
    MatSortModule, MatPaginatorModule, CommonModule,
FormsModule],
  templateUrl: './admin-users.component.html',
  standalone: true,
  styleUrl: './admin-users.component.scss'
})
export class AdminUsersComponent implements AfterViewInit{
  displayedColumns: string[] = ["id", "login", "password",
"name", "balance", "edit", "delete"];
  dataSource = new MatTableDataSource<UsersInterface>();

  baseAdminListURL = adminPanelUsersURL;

  constructor(private adminPanelService: AdminPanelService) {}

  @ViewChild(MatSort) sort: any;
  @ViewChild(MatPaginator) paginator: any;

  user:UsersInterface = {
    id: 0,
    login: "",
    password: "",
    name: "",
```

```
    balance: 0
  }

  users:UsersInterface[] = [];
  filteredAdmins:UsersInterface[] = [];

  ngAfterViewInit() {
    this.getUsers();

    setTimeout(() => {
      this.dataSource.paginator = this.paginator;
      this.dataSource.sort = this.sort;
    }, 100);
  }

  getUsers() {

this.adminPanelService.fetchAllData(this.baseAdminListURL).subsc
ribe((data) => {
      this.users = data;
      this.dataSource = new
MatTableDataSource<UsersInterface>(data);
    })
  }

  searchUser(input: any) {
    this.filteredAdmins = this.users.filter(item =>
      item.login.toLowerCase().includes(input.toLowerCase())
      item.name.toLowerCase().includes(input.toLowerCase())
      item.balance.toString().includes(input) ||
      item.id.toString().includes(input)
    );

    this.dataSource = new
MatTableDataSource<UsersInterface>(this.filteredAdmins);
    this.dataSource.paginator = this.paginator;
    this.dataSource.sort = this.sort;
  }

  PostPutAdmin(user:UsersInterface) {
    if (user.id !== 0) {
      this.adminPanelService.updateData(user,
this.baseAdminListURL).subscribe({
```

```
        next: (data) => {
          console.log(Admin updated Successfully!);
          this.getUsers();
        },
        error: (err) => {
          console.log(err);
          this.getUsers();
        }
      })
    } else {
      this.adminPanelService.createData(user,
this.baseAdminListURL).subscribe({
        next: (data) => {
          console.log(New Admin created Successfully!);
          this.getUsers();
        },
        error: (err) => {
          console.log(err);
          this.getUsers();
        }
      })
    }
  }

  putAdminFormField(user: UsersInterface) {
    this.user.id = user.id;
    this.user.login = user.login;
    this.user.password = user.password;
    this.user.name = user.name;
    this.user.balance = user.balance;
  }

  deleteAdminFormField(id: Number) {
    const  isConfirmed = window.confirm("Are you sure you want
to DELETE?");

    if (isConfirmed) {
      this.adminPanelService.deleteData(id,
this.baseAdminListURL).subscribe((data) => {
        this.users = this.users.filter(item => item.id !== id);

        this.getUsers();
      });
```

```
    }
  }
}
```

Б.6. Компонента site-content

```
import {AfterViewInit, Component} from '@angular/core';
import {AdminPanelService} from '../../shared/services/admin-
panel.service';
import {adminPanelClothesURL} from
'../../shared/DbLinks/UrlLinks';
import {ClothesInterface} from '../../shared/types/admin-panel-
clothes.interface';
import {CommonModule} from '@angular/common';
import {RouterLink, RouterLinkActive} from '@angular/router';

@Component({
  selector: 'app-site-content',
  imports: [CommonModule, RouterLink, RouterLinkActive],
  templateUrl: './site-content.component.html',
  standalone: true,
  styleUrl: './site-content.component.scss'
})
export class SiteContentComponent implements AfterViewInit{

  baseAdminListURL = adminPanelClothesURL;

  constructor(private adminPanelService: AdminPanelService) {}

  clothes:ClothesInterface[] = [];
  filteredNewClothes:ClothesInterface[] = [];
  filteredTopClothes:ClothesInterface[] = [];

  visibleNewClothes:ClothesInterface[] = [];
  visibleTopClothes:ClothesInterface[] = [];

  ngAfterViewInit() {

this.adminPanelService.fetchAllData(this.baseAdminListURL).subsc
ribe((data) => {
      this.clothes = data;
    })
```

```
    this.filteredNewClothes = this.clothes.filter(item =>
item.rating == "new");
    this.filteredTopClothes = this.clothes.filter(item =>
item.rating == "top");

    this.visibleNewClothes = this.filteredNewClothes.slice(0,
4);
    this.visibleTopClothes = this.filteredTopClothes.slice(0,
4);
  }
}
```

Б.7. Компонента site-detail

```
import {AfterViewInit, Component} from '@angular/core';
import {mainSiteParticularItem} from
'../../shared/DbLinks/UrlLinks';
import {AdminPanelService} from '../../shared/services/admin-
panel.service';
import {ClothesInterface} from '../../shared/types/admin-panel-
clothes.interface';
import {ActivatedRoute} from '@angular/router';
import {CommonModule, NgForOf} from '@angular/common';

@Component({
  selector: 'app-site-detail',
  imports: [
    CommonModule
  ],
  templateUrl: './site-detail.component.html',
  standalone: true,
  styleUrl: './site-detail.component.scss'
})
export class SiteDetailComponent implements AfterViewInit{
  baseAdminListURL = mainSiteParticularItem;

  constructor(private adminPanelService: AdminPanelService,
private route: ActivatedRoute) {}

  clothItem: ClothesInterface[] = [];
```

```
  ngAfterViewInit() {

this.adminPanelService.fetchParticularData(Number(this.route.sna
pshot.paramMap.get("id")),
this.baseAdminListURL).subscribe((data) => {
      this.clothItem = data;
      console.log(this.clothItem);
    })
  }
}
```

Б.8. Контролер

```
// Controller-implement
const express = require("express");
const router = express.Router();
const db = require("../DB/db");
const bodyParser = require("body-parser");
const jwt = require("jsonwebtoken");

router.use(bodyParser.json());
router.use(bodyParser.urlencoded({
    extended: true
}))

//Get method
router.get("/api/v1/admins", (req, res) => {
    const sql = "Select * from acc.admins";

    db.query(sql, (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(result.rows);
    })
});

router.get("/api/v1/users", (req, res) => {
    const sql = "Select * from acc.users";

    db.query(sql, (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(result.rows);
    })
```

```
});

router.get("/api/v1/clothes", (req, res) => {
    const sql = "Select * from acc.clothes";

    db.query(sql, (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(result.rows);
    })
});

router.get("/api/v1/detail/:id", (req, res) => {
    const {id} = req.params;

    if (!id) {
        return res.status(400).json({ message: "ID are required"
});
    }

    const sql = "SELECT * FROM acc.clothes WHERE id = $1";

    db.query(sql, [id], (err, result) => {
        if (err) return res.status(500).json({ message:
"Database error", error: err });

        if (result.rows.length === 0) {
            return res.status(401).json({ message: "Invalid ID"
});
        }

        return res.status(200).json(result.rows);
    });
})

//Post method
router.post("/api/v1/admins", (req, res) => {
    const {login, password} = req.body;
    const sql = "insert into acc.admins(login, password) VALUES
($1,$2) RETURNING *";

    db.query(sql, [login, password], (err, result) => {
        if (err) throw res.json(err);
```

```
        const user = result.rows[0];

        const token = jwt.sign(
            { id: user.id, login: user.login }, // payload
            "u92Dj#4r@9dkLS!92mdkqPz6vG!ks1Nd"
        );

        return res.status(201).json(result.rows);
    })
});

router.post("/api/v1/login", (req, res) => {
    const { login, password } = req.body;

    if (!login || !password) {
        return res.status(400).json({ message: "Login and
password are required" });
    }

    const sql = "SELECT * FROM acc.admins WHERE login = $1 AND
password = $2";

    db.query(sql, [login, password], (err, result) => {
        if (err) return res.status(500).json({ message:
"Database error", error: err });

        // Якщо користувача не знайдено
        if (result.rows.length === 0) {
            return res.status(401).json({ message: "Invalid
login or password" });
        }

        // Отримуємо користувача
        const user = result.rows[0];

        // Генеруємо токен
        const token = jwt.sign(
            { id: user.id, login: user.login },
            "u92Dj#4r@9dkLS!92mdkqPz6vG!ks1Nd",
            { expiresIn: "1h" } // токен діє 1 годину
        );
```

```
        // Повертаємо дані
        return res.status(200).json({
            message: "Authentication successful",
            admin: {
                id: user.id,
                login: user.login,
                token: token
            }
        });
    });
});

router.post("/api/v1/users", (req, res) => {
    const {login, password, name, balance} = req.body;
    const sql = "insert into acc.users(login, password, name,
balance) VALUES ($1,$2, $3, $4) RETURNING *";

    db.query(sql, [login, password, name, balance], (err,
result) => {
        if (err) throw res.json(err);
        return res.status(201).json(result.rows);
    })
});

router.post("/api/v1/clothes", (req, res) => {
    const {name, prise, rating, description, discount, size,
color, img, quantity} = req.body;
    const sql = "insert into acc.clothes(name, prise, rating,
description, discount, size, color, quantity, img) VALUES
($1,$2,$3,$4,$5,$6,$7,$8,$9) RETURNING *";

db.query(sql, [name, prise, rating, description, discount, size,
color, quantity, img], (err, result) => {
        if (err) throw res.json(err);
        return res.status(201).json(result.rows);
    })
});

//Put method
router.put("/api/v1/admins/:id", (req, res) => {
    const accId = Number(req.params.id);
    const {login, password} = req.body;
```

```
    const sql = "update acc.admins SET login=$1,password=$2
Where id=$3";

    db.query(sql, [login, password, accId], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is updated successfully
for id: ${accId});
    })
});

router.put("/api/v1/users/:id", (req, res) => {
    const accId = Number(req.params.id);
    const {login, password, name, balance} = req.body;
    const sql = "update acc.users SET
login=$1,password=$2,name=$3,balance=$4 Where id=$5";

    db.query(sql, [login, password, name, balance, accId], (err,
result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is updated successfully
for id: ${accId});
    })
});

router.put("/api/v1/clothes/:id", (req, res) => {
    const accId = Number(req.params.id);
    const {name, prise, rating, description, discount, size,
color, img, quantity} = req.body;
    const sql = "update acc.clothes SET
name=$1,prise=$2,rating=$3,description=$4,discount=$5,size=$6,co
lor=$7,img=$8,quantity=$9 Where id=$10";

    db.query(sql, [name, prise, rating, description, discount,
size, color, img, quantity, accId], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is updated successfully
for id: ${accId});
    })
});

//Delete method
router.delete("/api/v1/admins/:id", (req, res) => {
    const id = Number(req.params.id);
```

```
    const sql = "DELETE from acc.admins where id=$1";

    db.query(sql, [id], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is deleted successfully
for id: ${id});
    })
});

router.delete("/api/v1/users/:id", (req, res) => {
    const id = Number(req.params.id);
    const sql = "DELETE from acc.users where id=$1";

    db.query(sql, [id], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is deleted successfully
for id: ${id});
    })
});

router.delete("/api/v1/clothes/:id", (req, res) => {
    const id = Number(req.params.id);
    const sql = "DELETE from acc.clothes where id=$1";

    db.query(sql, [id], (err, result) => {
        if (err) throw res.json(err);
        return res.status(200).json(Item is deleted successfully
for id: ${id});
    })
});

module.exports = router;
```

Б.9. Підключення до бази даних

```
// Postgresql connection and pool
const pgSqlPool = require("pg").Pool;
const pool = new pgSqlPool({
   user: "postgres",
   password: "HogvardssPostgreSQL",
   database: "ShopCO",
   host: "localhost",
   port: "5432",
```

```
   max: 10
});

pool.connect((err, connection) => {
    if (err) throw err;
    console.log(Connected to PosgreSQL DB successfull!);
    connection.release();
})

module.exports = pool;
```

Б.10. Запуск серверної частини

```
const express = require("express");
const app = express();
const cors = require("cors");
const port = process.env.port || 3005;
const bodyParser = require("body-parser");

app.use(cors());
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({
    extended: true
}))

const controller = require("./controller/controller");
app.use(controller);

app.listen(port, (err) => {
    if (err) throw err;
    console.log(Server is running on port: ${port});
})
```

Б.11. API посилання

```
export const adminPanelListURL: String = "/api/v1/admins";
export const adminPanelUsersURL: String = "/api/v1/users";
export const adminPanelClothesURL: String = "/api/v1/clothes";
export const adminPanelLoginURL: String = "/api/v1/login";
export const mainSiteParticularItem: String = "/api/v1/detail";
```

Б.12. Guard

```
// src/app/shared/guards/auth.guard.ts
```

```
import { Injectable } from '@angular/core';
import {
  CanActivate,
  CanActivateChild,
  ActivatedRouteSnapshot,
  RouterStateSnapshot,
  Router
} from '@angular/router';
import { AuthService } from '../services/auth.service';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate, CanActivateChild
{

  constructor(private auth: AuthService, private router: Router)
{}

  canActivate(route: ActivatedRouteSnapshot, state:
RouterStateSnapshot): boolean {
    return this.checkAuth();
  }

  canActivateChild(childRoute: ActivatedRouteSnapshot, state:
RouterStateSnapshot): boolean {
    return this.checkAuth();
  }

  private checkAuth(): boolean {
    if (this.auth.isLogged()) {
      return true;
    }
    this.router.navigate(['/admin-panel/login']);
    return false;
  }
}
```

Б.13. Сервіс admin-panel

```
import { Injectable } from '@angular/core';
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs';
```

```
@Injectable({
  providedIn: 'root'
})
export class AdminPanelService {

  constructor(private _httpClient: HttpClient) { }

  fetchAllData(url: String):Observable<any> {
    return this._httpClient.get<any>(${url});
  }

  fetchParticularData(id: Number, url: String):Observable<any> {
    return this._httpClient.get<any>(${url}/${id});
  }

  createData(data:any, url:String) {
    return this._httpClient.post<any>(${url}, data);
  }

  updateData(data:any, url: String) {
    return this._httpClient.put<any>(${url}/${data.id}, data);
  }

  deleteData(id: Number, url: String) {
    return this._httpClient.delete<any>(${url}/${id});
  }
}
```

Б.14. Сервіс auth

```
import {Injectable, signal} from '@angular/core';
import {AdminsInterface} from '../types/admin-panel-
admins.interface';

@Injectable({
  providedIn: "root"
})

export class AuthService {
  currentAdminSig = signal<AdminsInterface | undefined |
null>(undefined);

  isLogged(): boolean {
    if (typeof window === 'undefined') return false;
```

```
    return !!localStorage.getItem("token");
  }
}
```

Б.15. Типи даних

```
export interface AdminsInterface {
  id:Number,
  login:String,
  password:String,
  token: String
}
export interface ClothesInterface {
  id:Number,
  name:String,
  prise:Number,
  rating:String,
  description:String,
  discount:Number,
  size:String,
  color:String,
  img:String,
  quantity:Number
}
export interface UsersInterface {
  id:Number,
  login:String,
  password:String,
  name:String,
  balance:Number
}
```