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

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Реляційна база даних у вебдодатку "Магазин побутової техніки" на основі Spring»

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

_____ *Іван САМБОР*
(підпис) *Ім'я, ПРІЗВИЩЕ здобувача*

Виконав: здобувач вищої освіти гр. КІД-41
Іван САМБОР
Ім'я, ПРІЗВИЩЕ
Керівник: Ігор БУЧЕНКО
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'ютерної інженерії

Освітньо-професійна програма Комп'ютерна інженерія

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

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Самбор Іван Валерійович
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: Реляційна база даних у вебдодатку "Магазин побутової техніки" на основі Spring
керівник кваліфікаційної роботи Ігор БУЧЕНКО,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025р. № 56
2. Строк подання кваліфікаційної роботи «5» червня 2025р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література; Wallach, J., B. (2003). PostgreSQL: Introduction and Concepts. Addison-Wesley Professional PostgreSQL Documentation; Spring Framework Documentation
4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Аналіз типів баз даних та вибір реляційної субд для вебдодатку
2. Огляд архітектури spring та його модулів для роботи з базою даних
3. Проєктування бази даних
4. Налаштування та інтеграція бази даних з вебдодатком, робота з даними
5. Перелік ілюстративного матеріалу: *презентація*
6. Дата видачі завдання «24» лютого 2025р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Збір даних	25.02-24.03.2025	Виконано
2	Обробка матеріалу	23.03-25.03.2025	Виконано
3	Розробка теоретичної частини	26.03-10.04.2025	Виконано
4	Розробка практичної частини	11.04-01.05.2025	Виконано
5	Висновки за результатами аналізу	02.05-06.05.2025	Виконано
6	Оформлення пояснювальної записки	07.05-08.05.2025	Виконано
7	Розробка презентації	09.05-11.05.2025	Виконано
8	Передзахист	27.05-29.05.2025	Виконано
9	Здача в деканат	02.06-05.06.2025	Виконано

Здобувач вищої освіти _____ Іван САМБОР
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник кваліфікаційної роботи _____ Ігор БУЧЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 71 стор., 55 рис., 4 табл., 27 джерел.

Мета роботи – дослідження та впровадження реляційної бази даних для вебдодатку "Магазин побутової техніки" на основі Spring.

Об'єкт дослідження – реляційна база даних у вебдодатоку на основі Spring.

Предмет дослідження – реляційна база даних, що використовується для зберігання та обробки даних у вебдодатку.

Короткий зміст роботи: У роботі здійснено детальний аналіз існуючих типів баз даних з акцентом на реляційні СУБД. Обґрунтовано вибір PostgreSQL як оптимального рішення для реалізації бази даних для вебдодатку інтернет-магазину.

Описано процес логічного та фізичного проєктування структури бази даних із застосуванням принципів нормалізації, побудови первинних та зовнішніх ключів, зв'язків «один до багатьох» та «багато до багатьох».

Проведено реалізацію основних сутностей (користувачі, клієнти, товари, замовлення, категорії, виробники) з використанням ORM-технології Hibernate у складі Spring Data JPA. Визначено правила валідації даних, реалізовано транзакції, CRUD-операції та підключення до бази через Spring Boot.

Показано інтеграцію бази даних до вебдодатку з урахуванням вимог продуктивності, масштабованості та безпеки.

КЛЮЧОВІ СЛОВА: РЕЛЯЦІЙНА БАЗА ДАНИХ, POSTGRESQL, SPRING BOOT, SPRING DATA JPA, HIBERNATE, ORM, ВЕБДОДАТОК, ЗВ'ЯЗКИ, CRUD.

ЗМІСТ

Стор.
ВСТУП **8**
1 АНАЛІЗ ТИПІВ БАЗ ДАНИХ ТА ВИБІР РЕЛЯЦІЙНОЇ СУБД ДЛЯ ВЕБДОДАТКУ **9**
1.1 Типи баз даних 9
1.2 Робота з реляційними базами даних. Мова запитів SQL 17
1.3 Реляційні системи керування базами даних 24
2 ОГЛЯД АРХІТЕКТУРИ SPRING ТА ЙОГО МОДУЛІВ ДЛЯ РОБОТИ З БАЗОЮ ДАНИХ **30**
2.1 Контейнер інверсії керування Spring IoC 30
2.2 Spring Boot 33
2.3 Spring MVC 34
2.4 Spring Security 36
2.5 Spring Data 39
3 ПРОЄКТУВАННЯ БАЗИ ДАНИХ **48**
3.1 Аналіз вимог до структури бази даних 48
3.2 Проєктування моделі бази даних 50
3.3 Оптимізація бази даних: індекси, зовнішні ключі та обмеження 59
4 НАЛАШТУВАННЯ ТА ІНТЕГРАЦІЯ БАЗИ ДАНИХ З ВЕБДОДАТКОМ, РОБОТА З ДАНИМИ **61**
4.1 Встановлення та інтеграція PostgreSQL з вебдодатком за допомогою Docker 61
4.2 Створення структури бази даних 65
4.3 Використання Spring Data JPA для взаємодії з БД 74
ВИСНОВКИ **77**
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ **78**
Додаток А. Скрипт schema.sql для ручного створення таблиць 81
Додаток Б. Скрипт data.sql для заповнення таблиць початковими даними 82
Додаток В. Заповнені таблиці employees, appliances, manufacturers 83
Додаток Г. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ(презентація) 84

ВСТУП

У сучасному світі інформаційні технології відіграють ключову роль у розвитку бізнесу, а ефективне управління даними є основою успішного функціонування будь-якої компанії. В умовах стрімкого зростання онлайн-торгівлі особливо важливою стає оптимізація роботи баз даних, які забезпечують зберігання, обробку та швидкий доступ до інформації.

Реляційні бази даних є найбільш поширеним способом організації та структуризації даних у вебдодатках. Вони дозволяють гарантувати цілісність і узгодженість даних, що критично важливо для електронної комерції, зокрема для магазинів побутової техніки. Вебдодатки, побудовані на базі Spring Framework, надають можливість ефективно працювати з базами даних, використовуючи потужні інструменти, такі як Spring Data JPA та Hibernate, що спрощують взаємодію між програмною логікою та сховищем даних.

З розвитком технологій виникає потреба в оптимізації роботи бази даних, що включає правильне проєктування структури, використання індексів, забезпечення транзакційної цілісності та кешування запитів для підвищення продуктивності. Крім того, питання безпеки даних та резервного копіювання відіграють ключову роль у стабільності роботи вебдодатків.

Мета даної роботи полягає в проєктуванні та впровадженні реляційної бази даних для вебдодатку "Магазин побутової техніки" на основі Spring, з урахуванням сучасних підходів до моделювання, управління та інтеграції даних.

1 АНАЛІЗ ТИПІВ БАЗ ДАНИХ ТА ВИБІР РЕЛЯЦІЙНОЇ СУБД ДЛЯ ВЕБДОДАТКУ

1.1 Типи баз даних

Бази даних дозволяють нам ефективно працювати з великими обсягами даних. Вони роблять оновлення даних простим і надійним, а також пропонують функції безпеки для контролю доступу до інформації та допомагають нам уникати надмірності. База даних – це структура, яка зберігає інформацію в організованому, послідовному, надійному та доступному для пошуку вигляді. Вона надає можливість додати структурованості даним і визначити правила, яким ці дані повинні відповідати. Весь сенс наявності даних полягає в тому, щоб мати можливість використовувати їх для чогось, а база даних надає нам інструменти як для захисту даних, так і для зручного їх використання.

Існує два типи баз даних: реляційні та нереляційні (SQL та NoSQL).

Нереляційна бази даних (NoSQL). Використовує різні моделі даних для зберігання, керування та доступу до даних. Найпоширенішими нереляційними моделями даних є: документно-орієнтована, ключ-значення, широкостовпчикова та графова.

Документно-орієнтована база даних. Документно-орієнтована база даних зберігає дані в документах, схожих на об'єкти JSON (JavaScript Object Notation). Кожен документ містить пари полів і значень. Значення, як правило, можуть бути найрізноманітніших типів, включаючи рядки, числа, булеві числа, масиви або навіть інші об'єкти. Прикладами документно-орієнтованих баз даних є MongoDB та Couchbase[1].



Рисунок 1.1 – Структура документа в документно-орієнтованій базі даних (JSON-формат)

Бази даних ключ-значення. Сховище ключ-значення – це простіший тип бази даних, де кожен елемент містить ключі та значення. Кожен ключ унікальний і пов'язаний з одним значенням. Вони використовуються для кешування та управління сеансами і забезпечують високу продуктивність при читанні та записі, оскільки мають тенденцію зберігати дані в пам'яті. Прикладами є Amazon DynamoDB і Redis[1][10].

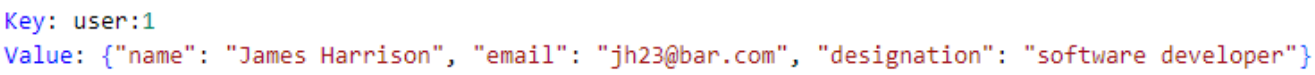


Рисунок 1.2 – Вигляд даних, що зберігаються в базі даних ключ-значення

Широкостовпчикові бази даних. Широкостовпчикові бази даних зберігають дані у вигляді стовпців, а не рядків. Дані зберігаються в таблицях. Однак, на відміну від традиційних SQL баз даних, широкостовпчикові бази даних є гнучкими, де кожен запис може мати різну кількість стовпців. У реляційній базі даних ми б мали заповнювати всі колонки або використовувати NULL. Але в широкостовпчиковій БД кожен рядок може містити унікальний набір стовпців, зберігаючи тільки необхідні дані. Прикладами сховищ з широкими стовпчиками є Apache Cassandra та HBase[1][10].

Таблиця 1.1 – Вигляд даних у сховищ із широкими стовпчиками

Id	Name	Email	Age	Address	Phone
1	Denys	denys23@gmail.com	21	Київ, вул.Максименка	380501234567
2	Oleksandr	Oleks56@gmail.com	32	-	-
3	Julia	julia@gmail.com	-	Київ, вул.Васильківська	380631112233

Графові бази даних. База даних графів зберігає дані у вигляді вузлів (nodes) і зв’язків (edges). Вузли зазвичай зберігають інформацію про людей, місця та речі, тоді як зв’язки показують, як вузли пов’язані між собою. Вони мають напрям і можуть містити додаткові властивості. Властивості (Properties) – атрибути вузлів і зв’язків, які зберігають додаткові дані. Вони добре працюють для сильно пов'язаних даних, де взаємозв'язки або закономірності можуть бути не дуже очевидними на початку. Прикладами графових баз даних є Neo4J та Amazon Neptune[1].

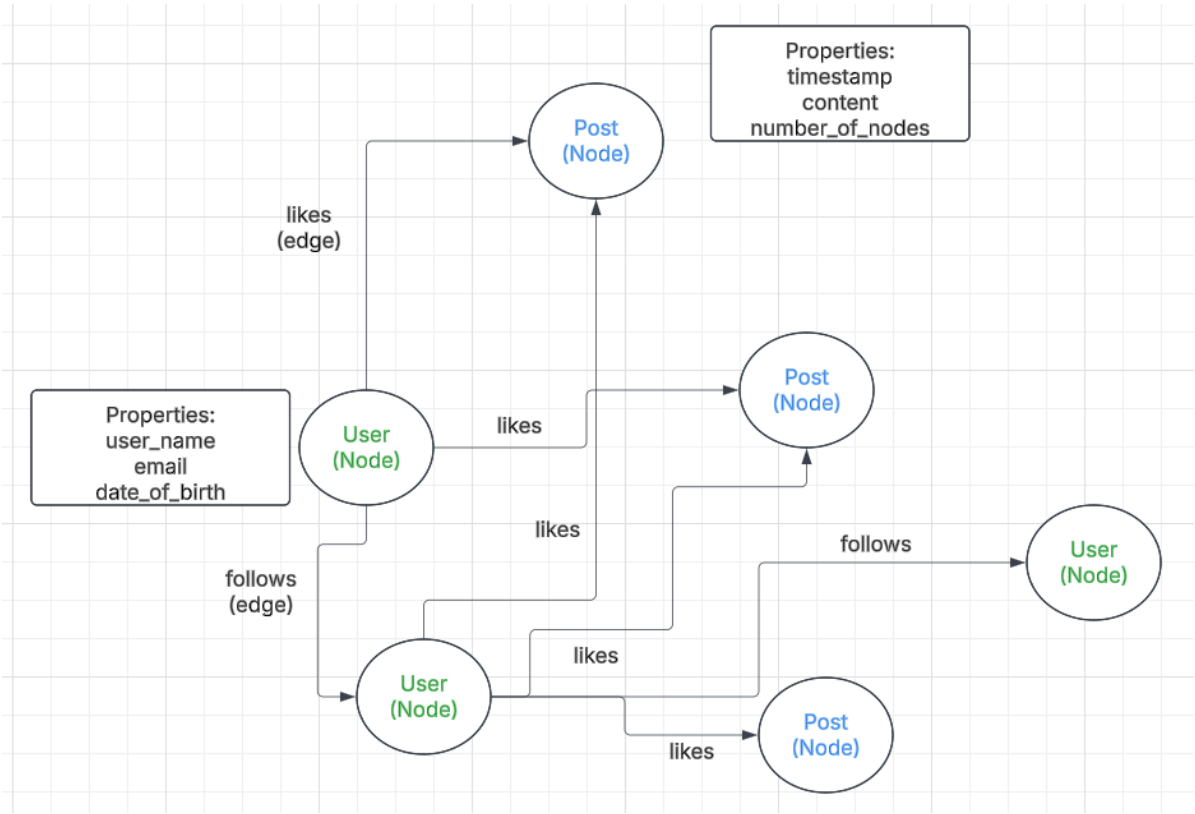


Рисунок 1.3 – Приклад зберігання даних у графовій базі даних

Реляційна база даних. Реляційні бази даних є безумовно найпопулярнішим типом баз даних. За результатами опитування серед розробників на порталі Stackoverflow, у 2024 році реляційні бази даних займають лідируючі позиції. Зокрема, такі РБД, як PostgreSQL, MySQL та SQLite, користуються найбільшим попитом(рис. 1.4).

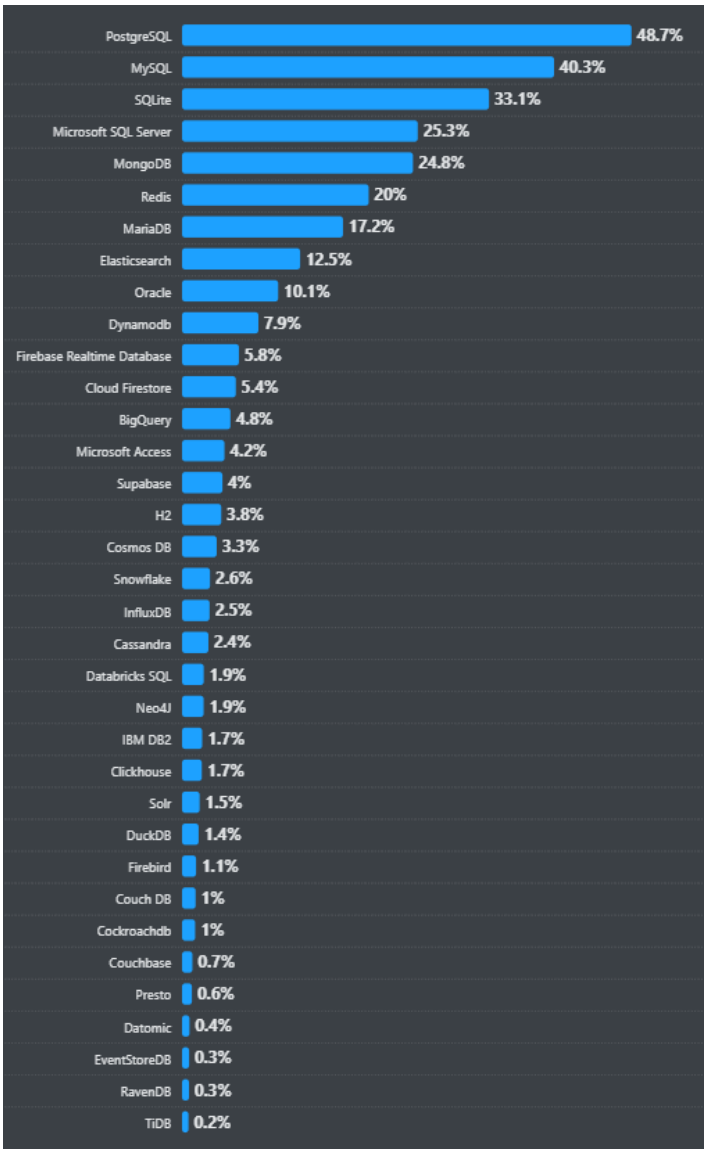


Рисунок 1.4 – Результат опитування розробників на Stackoverflow за 2024 рік[2]

Реляційна бааза даних – це тип бази даних, що організовує дані в таблиці, що складаються з рядків (записів, екземпляри певної сутності) і стовпців (атрибутів).

Рядки – кожен рядок є окремим записом, що містить інформацію про певну

сутність. Кожен запис є унікальним і визначається за допомогою ключа.

Стовпці – це характеристики об’єктів, представлених у таблиці. Для кожного стовпця встановлюється тип даних, що визначає, яку інформацію він містить (наприклад, число, текстовий рядок, дата або логічне значення). Усі значення в одному стовпці мають однаковий тип, хоча самі дані можуть відрізнятися.



Рисунок 1.5 – Вигляд таблиці в реляційній базі даних

Таблиці в реляційних базах даних з’єднані з іншими таблицями за допомогою зв’язків первинного або зовнішнього ключа.

Первинний ключ (Primary Key) – це унікальний ідентифікатор для кожного запису в таблиці, який забезпечує унікальність записів.

Зовнішній ключ (Foreign Key) – це стовпець або набір стовпців в одній таблиці, який посилається на первинний ключ в іншій таблиці, встановлюючи зв’язок між ними.

Зв’язок (Relationship) – визначає, які стовпці в одній таблиці пов’язані з відповідними стовпцями в іншій таблиці. Зв’язки між таблицями можуть бути один до одного, один до багатьох або багато до багатьох[11].

Один до одного – будь-якому значенню атрибуту A відповідає тільки одне значення атрибуту B, і навпаки.

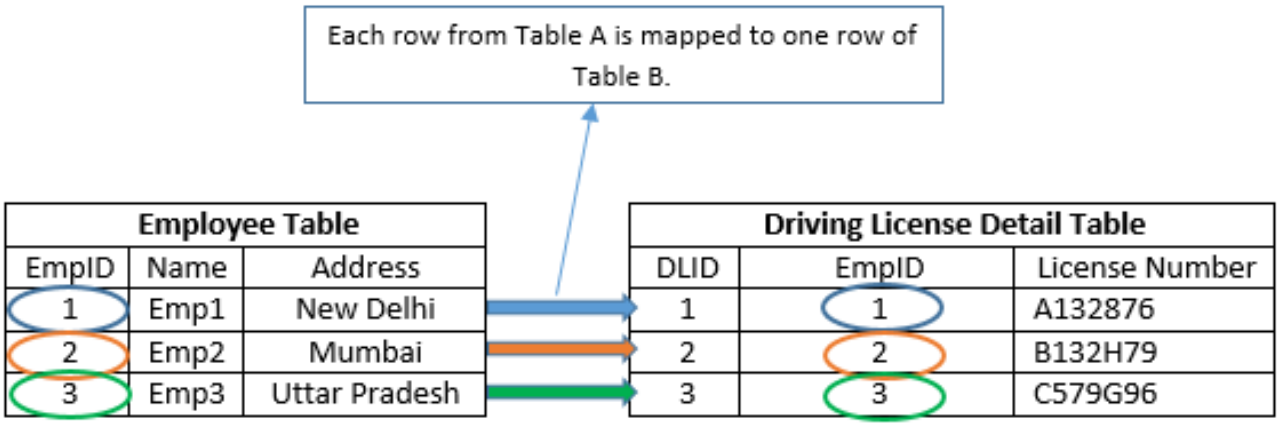


Рисунок 1.6 – Співідношення один до одного[3]

Один до багатьох – будь-якому значенню атрибуту A відповідає 0, 1 або кілька значень атрибуту B[3].

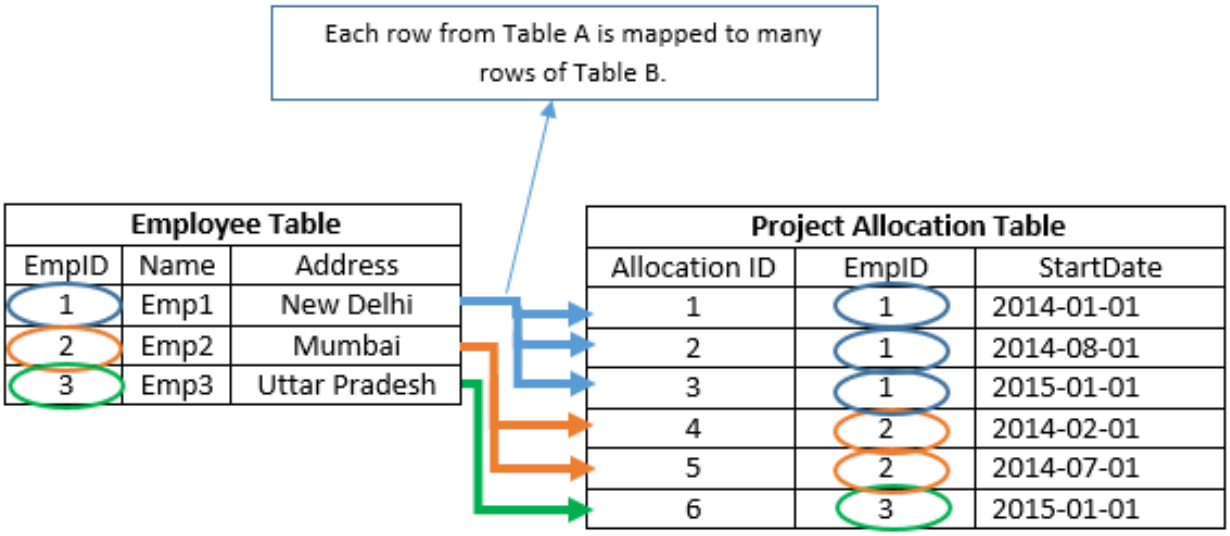


Рисунок 1.7 – Співідношення один до багатьох[3]

Багато до багатьох – будь-якому значенню атрибуту A відповідає 0, 1 або кілька значень атрибуту B, і будь-якому значенню атрибуту B відповідає 0, 1 або кілька значень атрибуту A[3].

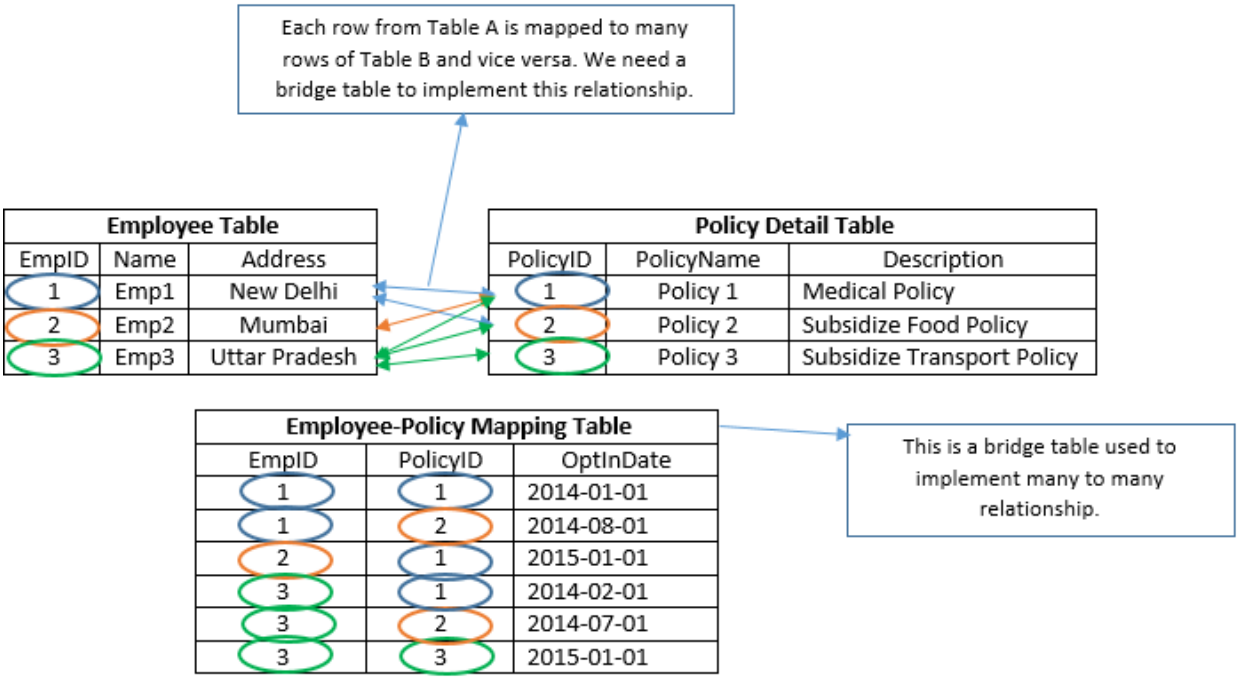


Рисунок 1.8 – Співвідношення багато до багатьох[3]

Зв’язки дозволяють підійти до проєктування баз даних з точки зору нормалізації.

Нормалізація – це процес упорядкування структури бази даних для мінімізації надмірності та уникнення суперечностей даних. Вона не спрямована на покращення продуктивності чи зменшення обсягу даних, а допомагає зробити схему більш логічною та ефективною для роботи. Щоб досягти нормалізації, використовуються система правил, відома як нормальні форми[4].

Нормальні форми – це правила, які допомагають правильно структурувати базу даних та уникнути проблем з надмірністю і суперечливістю даних.

Перша нормальна форма (1NF) – кожне поле має містити тільки одне значення, а всі рядки мають бути унікальні.

Друга нормальна форма (2NF) – усі дані залежать тільки від усього первинного ключа.

Третя нормальна форма (3NF) – Всі стовпці повинні залежати тільки від первинного ключа і не мати залежностей між собою.

Крім нормалізації, у реляційних база даних є зворотний процес денормалізація.

Денормалізація бази даних – це процес усвідомленого приведення бази даних до виду, в якому вона не буде відповідати правилам нормалізації. Зазвичай це необхідно для підвищення продуктивності і швидкості вилучення даних, за рахунок збільшення надмірності даних.

Оптимізація запитів у базах даних – це процес покращення продуктивності SQL-запитів, щоб зменшити час виконання і використання ресурсів. Одним із ключових інструментів для цього є індекси.

Індексація – це один з основних механізмів оптимізації продуктивності запитів у реляційних базах даних. Індекс є спеціальною структурою даних, яка дозволяє прискорити виконання запитів до таблиць. Індекс діє як вказівник на рядки таблиці, що дозволяє СУБД швидше знаходити потрібні дані без повного перегляду всієї таблиці[3].

Транзакція – це вплив на базу даних, що переводить її з одного цілісного стану в інший і виражається у зміні даних, що зберігаються в базі даних.

Основні принципи транзакції (ACID):

— *атомарність (Atomicity).* Гарантує, що всі операції в межах однієї транзакції будуть виконані повністю або не буде виконано жодної;

— *консистентність даних (Consistency).* Транзакція яка досягла свого нормального завершення і тим самим фіксує свої результати, зберігає узгодженість бази даних;

— *ізольованість (Isolation).* Під час виконання, паралельні транзакції не повинні впливати на результат;

— *довговічність (Durability).* Після успішного завершення транзакції її результати повинні залишитися збереженими, навіть у випадку збою системи.

Переваги реляційної бази даних:

— підримка зв’язків між об’єктами;

— структурована мова запитів;

— підримка транзакцій;

— нормалізація та денормалізація даних;

— багата історія розвитку та широке поширення (один із основних інструментів для розробки застосунків і сервісів).

Недоліки реляційної бази даних:

— жорстка структура;

— проблеми з продуктивністю (Сповільнення системи при складних JOIN запитах при великих обсягах даних);

— високі вимоги до ресурсів (При великих обсягах даних вимагає потужного обладнання та оптимізації запитів).

1.2 Робота з реляційними базами даних. Мова запитів SQL

Робота з реляційними базами даних включає виконання запитів і маніпулювання даними за допомогою струтурованої мови запитів SQL (Structured Query Language).

Запит (Query) – це інструкція, яка використовується для управління даними, визначення стркутури бази даних, управління доступом та транзакцій.

SQL – це мова, яка дозволяє змінювати та видаляти данні, визначати та змінювати структуру бази даних, створювати вибірки, виконувати агрегації та угрупування, керувати доступом користувачів до певних операцій тощо.

SQL поділяється на кілька категорій:

— *DML (Data Manipulation Language)* – мова маніпулювання даними. DML команди використовуються для роботи з даними, що зберігаються в таблицях. Ці запити дозволяють додавати, оновлювати, видаляти дані.

INSERT – додає нові дані в таблицю. Приклад використання INSERT: “INSERT INTO Users (UserID, Name, Email, Age) VALUES (3, 'Alice Johnson', 'alice@example.com', 28);”.

UPDATE – оновлює існуючі дані в таблиці. Приклад використання

UPDATE: “UPDATE Users SET Email = 'alice.johnson@example.com' WHERE UserID = 3;”.

DELETE – видаляє дані з таблиці. Приклад використання DELETE: “DELETE FROM Users WHERE UserID = 3;”;

— *DQL (Data Query Language)* – використовується виключно для отримання даних із бази. Основний оператор SELECT.

SELECT – витягує дані з таблиць. Приклад використання SELECT для вибірки всіх колонок з таблиці Users: “SELECT * FROM Users;”.

Приклад використання SELECT для вибірки імені та віку з таблиці Users: “SELECT Name, Age FROM Users;”.

Оператори DQL для фільтрації та умов: *WHERE, IN, BETWEEN, LIKE, IS NULL, ORDER BY;*

WHERE – визначає умови фільтрації даних. Приклад використання WHERE для філтрування за віком: “SELECT * FROM Users WHERE Age > 25;”.

IN – дозволяє вибирати значення, що належать до певного списку. Приклад використання IN: “SELECT * FROM Users WHERE Age IN (25, 30, 35);”.

BETWEEN – використовується для вибору значень в певному діапазоні. Приклад використання BETWEEN: “SELECT * FROM Users WHERE Age BETWEEN 20 AND 30;” Цей запит вибирає користувачів, чий вік знаходиться між 20 та 30 включно.

LIKE – використовується для пошуку шаблонів у текстових значеннях. Приклад використання LIKE: “SELECT * FROM Users WHERE Name LIKE 'J%';”. Цей запит вибирає всіх користувачів, чиї імена починаються з літери 'J'. Символ ‘%’ означає будь-яку кількість символів. Також можна використовувати символ ‘_’ що означає один будь-який символ.

IS NULL – використовується для перевірки наявності значення NULL. Приклад використання IS NULL: “SELECT * FROM Users WHERE Email IS NULL;”.

ORDER BY – використовується для сортування. ‘Asc’ сортує від найменшого до найбільшого. ‘Desc’ сортує від найбільшого донайменшого.

Приклад сортування за допомогою ORDER BY: “SELECT * FROM Users ORDER BY Age asc;”. “SELECT * FROM Users ORDER BY Age desc;”.

Функції DQL: *MIN(), MAX(), COUNT(), SUM(), AVG();*

MIN() – Повертає мінімальне значення в стовпці. Приклад використання MIN(): “SELECT MIN(Age) FROM Users;”. Цей запит поверне найменший вік серед всіх користувачів.

MAX() – Повертає максимальне значення в стовпці. Приклад використання MAX(): “SELECT MAX(Age) FROM Users;”. Цей запит поверне найбільший вік серед всіх користувачів.

COUNT() – Повертає кількість рядків, що відповідають умові. Приклад використання COUNT(): “SELECT COUNT(*) FROM Users;”. Цей запит поверне кількість всіх користувачів у таблиці.

SUM() – Повертає суму значень у стовпці (зазвичай числовому). Приклад використання SUM(): “SELECT SUM(Age) FROM Users;”. Цей запит поверне суму всіх віків користувачів.

AVG() – Повертає середнє значення у стовпці. Приклад використання AVG(): “SELECT AVG(Age) FROM Users;”. Цей запит поверне середній вік усіх користувачів.

Види джойнів в DQL: INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL (OUTER) JOIN;

Джоїни (з’єднання) – дозволяють комбінувати дані з двох або більше таблиць за певними умовами.

INNER JOIN – повертає записи, які мають відповідні значення в обох таблицях.

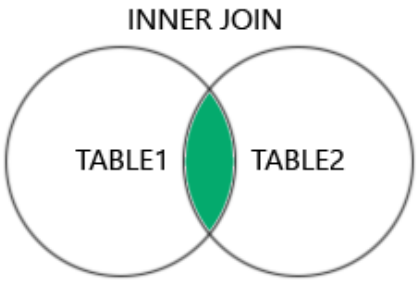


Рисунок 1.9 – Схематичне зображення операції INNER JOIN між двома таблицями[5]

Приклад використання INNER JOIN: “SELECT Users.Name, Orders.OrderID FROM Users INNER JOIN Orders ON Users.UserID = Orders.UserID;”. Цей запит вибирає імена користувачів і їхні замовлення тільки в тому випадку, якщо існує замовлення для цього користувача.

LEFT JOIN – повертає всі записи з лівої таблиці та відповідні записи з правої таблиці.

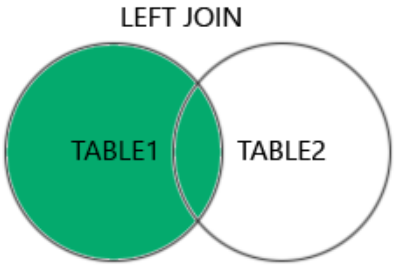


Рисунок 1.10 – Схематичне зображення операції LEFT JOIN між двома таблицями[5]

Приклад використання LEFT JOIN: “SELECT Users.Name, Orders.OrderID FROM Users LEFT JOIN Orders ON Users.UserID = Orders.UserID;”. Цей запит вибирає всіх користувачів і їхні замовлення. Якщо користувач не зробив жодного замовлення, то в результаті буде NULL для OrderID.

RIGHT JOIN – повертає всі записи з правої таблиці та відповідні записи з лівої таблиці.

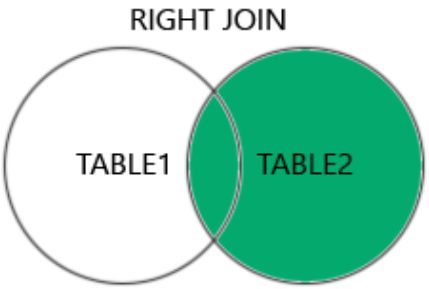


Рисунок 1.11 – Схематичне зображення операції RIGHT JOIN між двома таблицями[5]

Приклад використання RIGHT JOIN: “SELECT Users.Name, Orders.OrderID FROM Users RIGHT JOIN Orders ON Users.UserID = Orders.UserID;” Цей запит вибирає всі замовлення та відповідних користувачів. Якщо для замовлення немає користувача, то в результаті буде NULL для Name.

FULL (OUTER) JOIN – повертає всі записи, якщо є відповідність у лівій або правій таблиці.

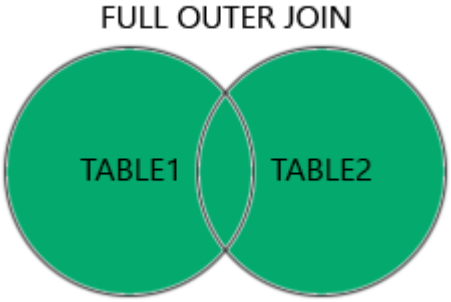


Рисунок 1.12 – Схематичне зображення операції FULL OUTER JOIN між двома таблицями[5]

Приклад використання FULL JOIN: “SELECT Users.Name, Orders.OrderID FROM Users FULL OUTER JOIN Orders ON Users.UserID = Orders.UserID;:. Цей запит вибирає всіх користувачів та замовлення, навіть якщо для одного з них немає відповідних записів.

— *DDL (Data Definition Language)* – це команди для створення, зміни та

видалення структури бази даних.

Основні команди DDL: CREATE, ALTER, DROP, TRUNCATE;

CREATE – створює таблиці. Приклад використання CREATE для створення таблиці Users: “CREATE TABLE Users (UserID INT PRIMARY KEY, Name VARCHAR(50), Email VARCHAR(100), Age INT);”.

ALTER – змінює структуру таблиці. Приклад використання ALTER для додавання колонки: “ALTER TABLE Users ADD Address VARCHAR(255);:.

Приклад використання ALTER для зміни типу колонки: “ALTER TABLE Users MODIFY COLUMN Age SMALLINT;”.

DROP – видаляє таблицю з бази даних. Приклад використання DROP для видалення таблиці: “DROP TABLE Users;”.

TRUNCATE – видаляє всі дані з таблиці (але не структуру). Приклад використання TRUNCATE: “TRUNCATE TABLE Users;”;

— *DCL (Data Control Language)* – використовується для управління доступом і правами користувачів у базі.

Основні команди DCL: GRANT, REVOKE;

GRANT – надає права користувачам. Приклад використання GRANT для надання прав на SELECT на таблицю Users для користувача John: “GRANT SELECT ON Users TO 'john';”.

REVOKE – відкликає права. Приклад використання REVOKE для відклике прав на SELECT на таблицю Users для користувача John: “REVOKE SELECT ON Users FROM 'john';”;

— *TCL (Transaction Control Language)* – відповідає за управління транзакціями.

Основні команди TCL: BEGIN, COMMIT, SAVEPOINT, ROLLBACK ;

BEGIN – команда, яка починає транзакцію. Після її виконання всі зміни будуть збережені тільки тоді, коли ми виконаємо команду COMMIT.

COMMIT – зберігає зміни зроблені під час транзакції.

SAVEPOINT – встановлює точку збереження, до якої можна відкотити транзакцію.

ROLLBACK – скасовує всі зміни зроблені в межах транзакції або відкочую

зміни до певної точки збереження.

Приклад управління транзакціями з SAVEPOINT і ROLLBACK(рис. 1.13):

1. почати транзакцію (BEGIN);
2. виконати INSERT нового користувача Bob Brown;
3. створити точку збереження (SAVEPOINT sp1);
4. виконати UPDATE для користувача, якого не існує (bob123@example.com);
5. якщо оновлення не вдалося, виконати ROLLBACK TO SAVEPOINT sp1;
6. виконати INSERT іншого користувача Jane Brown;
7. зафіксувати зміни (COMMIT).

```
BEGIN;

INSERT INTO Users (UserID, Name, Email, Age)
VALUES (4, 'Bob Brown', 'bob@example.com', 35);

SAVEPOINT sp1;

UPDATE Users SET Age = 20
WHERE email = 'bob123@example.com';

ROLLBACK TO SAVEPOINT sp1;

INSERT INTO Users (UserID, Name, Email, Age)
VALUES (4, 'Jane Brown', 'jane@example.com', 26);

COMMIT;
```

Рисунок 1.13 – Приклад використання TCL команд для управління транзакціями

1.3 Реляційні системи керування базами даних

З базою даних можна працювати напряму, якщо знати в якому вигляді зберігаються дані на фізичному рівні, напркилад у файлай або у певних структурах даних тощо. Для цього потрібно провести багато рутинної роботи, тому як праило напряму з базами даних не працюють, а використовують спеціальні застосунки. Для роботи з реляційними базами даних використовують РСКБД (Реляційні Системи Керування Базами Даних) – це програмне забезпечення, яке використовується для обробки SQL запитів, створення, управління та доступу до бази даних. Тобто РСКБД діє як посередник між базою даних, програмами та інтерфейсами користувача для ефективного керування та організації даних. Найпопулярнішими РСКБД є MySql, PostgreSQL, SQLite[3].

SQLite – це вбудована файлова реляційна СУБД з відкритим кодом, відома своєю портативністю, надійністю та високою продуктивністю навіть у середовищах із низьким обсягом пам’яті. Оскільки SQLite є файловою СУБД, то база даних знаходиться в одному дисковому файлі, і всі операції читання та запису відбуваються безпосередньо на цьому дисковому файлі. Це спрощує процес налаштування SQLite, оскільки усуває потребу підключати базу даних до сервера[12].

Переваги:

— *невеликий розмір*. Бібліотека SQLite є однією з найкомпактніших бібліотек, де розмір бібліотеки може бути менше 600 КБ. Завдяки своєму дуже малому розміру, він дуже добре підходить для IoT та вбудованих пристроїв[6];

— *зручна для користувача*. SQLite готова до використання з коробки, бо непрацює як серверний процес, це означає, що його ніколи не потрібно зупиняти, запускати чи перезапускати. Це допомагає спростити шлях від встановлення SQLite до його інтеграції з програмою[6];

— *портативність*. На відміну від інших систем керування базами даних, які зазвичай зберігають дані як великий пакет окремих файлів, уся база даних SQLite

зберігається в одному файлі. Цей файл можна розташувати будь-де в ієрархії каталогів і надавати спільний доступ за допомогою знімного носія або протоколу передачі файлів[6].

Недоліки:

— *обмежений паралелізм*. Хоча декілька процесів можуть одночасно отримувати доступ до бази даних SQLite та надсилати запити, лише один процес може вносити зміни до бази даних у будь-який момент часу. Це може спричинити проблеми з продуктивністю великих наборів даних через обмеження файлової системи. У цих випадках було б доцільно вибрати клієнт/серверні бази даних, такі як MySQL і PostgreSQL, особливо при роботі з великими наборами даних[8];

— *немає керування користувачами*. Оскільки SQLite читає та записує безпосередньо у звичайний дисковий файл, єдиними застосовними дозволами доступу є типові дозволи доступу основної операційної системи. Це робить SQLite поганим вибором для програм, які потребують кількох користувачів зі спеціальними правами доступу[9];

— *безпека*. Іншим недоліком SQLite є відсутність налаштуваннь безпеки, окрім шифрування самого файлу бази даних[6].

MySQL – є однією з найпопулярніших широкомасштабних систем RDBMS з відкритим кодом. На відміну від SQLite, він використовує архітектуру сервер/клієнт, яка складається з багатопотокового сервера SQL. Ця багатопотокова природа MySQL забезпечує більшу продуктивність, оскільки потоки ядра можуть легко використовувати кілька ЦП. База даних написана мовами C і C++ і підтримує різні платформи , такі як операційні системи Windows Server і дистрибутиви Linux. Також дотримується принципів ACID для узгодженості транзакцій і надає різні конектори та API, для C, C++, Java, PHP тощо[6; 7].

Переваги:

— *популярність і простота використання*. Одна з найпопулярніших у світі систем баз даних, існує велика кількість документації в друкованому вигляді та в Інтернеті про те, як встановити та керувати базою даних MySQL. Це включає

низку інструментів сторонніх розробників, таких як phpMyAdmin, які мають на меті спростити процес початку роботи з базою даних[6];

— *безпека*. MySQL встановлюється разом зі скриптом, який допомагає вам покращити безпеку бази даних шляхом встановлення пароля, визначення пароля для користувача root, видалення анонімних облікових записів і видалення тестових баз даних, які за замовчуванням доступні для всіх користувачів. Крім того, на відміну від SQLite, MySQL підтримує керування користувачами та дозволяє надавати привілеї доступу для кожного користувача[6];

— *швидкість*. Вирішивши не використовувати певні функції SQL, розробники MySQL змогли підвищити швидкодію. У той час як порівняльні тести показують, що інші RDBMS, такі як PostgreSQL, можуть зрівнятися з MySQL за швидкістю, але MySQL все ще має репутацію надзвичайно швидкого рішення для баз даних[6].

Недоліки:

— *обмеження*. Оскільки MySQL розроблено для швидкості та простоти використання, а не для повної сумісності з SQL, він має певні функціональні обмеження. Наприклад, у MySQL відсутня підтримка FULL JOIN[6];

— *проблеми з продуктивністю при певних операціях*. Групові INSERT можуть уповільнювати роботу, оскільки MySQL тимчасово зберігає старі версії даних у спеціальних областях пам'яті (сегментах відкату). Це створює додаткове навантаження на систему. Відсутність повнотекстового пошуку означає що можливості пошуку по великому тексту у MySQL обмежені в порівнянні з такими СУБД як PostgreSQL[7].

PostgreSQL – це об’єктна СУБД з відкритим вихідним кодом. Подібно до MySQL, PostgreSQL використовує модель бази даних клієнт/сервер, серверний процес, який обробляє комунікації клієнта, керує файлами та операціями бази даних, відомий як процес postgres. PostgreSQL обробляє одночасні сеанси клієнта шляхом створення нового процесу для кожного з’єднання. Цей процес є відокремленим від головного процесу postgres і створюється та знищується протягом підключення клієнта. PostgreSQL використовує технологію, відому як

Multiversion Concurrency Control або MVC, для підтримки узгодженості даних під час одночасного доступу до даних, це забезпечує атомарність, узгодженість, ізоляцію та довговічність транзакцій, також відомі як принципи ACID[6][7].

Переваги:

— *відповідність SQL*. Більше, ніж SQLite або MySQL, PostgreSQL прагне чітко дотримуватися стандартів SQL. Відповідно до офіційної документації, PostgreSQL підтримує 160 із 179 функцій, необхідних для повної відповідності ядру SQL[6];

— *паралельна обробка*. Завдяки своїм чудовим можливостям паралельної обробки, PostgreSQL виходить на перше місце (порівняно з MySQL) при виконанні довгих SELECT. Яскравим прикладом є розширення для PostgreSQL, Timescale DB, яке дозволяє вставляти 1 мільйон записів за секунду[7];

— *підтримка з відкритим кодом і спільнотою*. Повністю відкритий проєкт, вихідний код PostgreSQL розроблено великою та відданою спільнотою. Подібним чином спільнота Postgres підтримує численні онлайн-ресурси, які описують, як працювати з СУБД, включаючи офіційну документацію та різноманітні онлайн форуми[6].

Недоліки:

— *ефективність використання пам'яті*. Для кожного нового підключення клієнта PostgreSQL створює новий процес. Кожному новому процесу виділяється близько 10 МБ пам’яті, яка може швидко збільшуватися для баз даних із великою кількістю з’єднань[6];

— *складність налаштування та адміністрування*. PostgreSQL надає широкі можливості, але це також означає потребу в глибших технічних знаннях. Для ефективного використання часто потрібне тонке налаштування параметрів продуктивності, безпеки, що може ускладнити впровадження для початківців.

— *високе споживання ресурсів при складних запитах*. Складні SQL-запити з багатьма об’єднаннями (JOIN), підзапитами або агрегаціями можуть потребувати значних обсягів оперативної пам’яті. Без належної оптимізації це може призвести до навантаження на систему та зниження швидкодії[7].

Порівняння та вибір:

SQLite, призначений для невеликих баз даних, які працюють локально на одному пристрої. SQLite добре підходить для вбудованих пристроїв (мікроконтролери, IoT-пристрої, автомобільна техніка тощо), мобільних додатків або невеликих проєктів. Вебдодаток “магазин побутової техніки” буде багатокористувацьким, з потенційоно великим навантаженням, тому SQLite буде поганим вибором.

MySQL порівняно з PostgreSQL, має певні обмеження, наприклад відсутність підтримки FULL JOIN, немає підтримки повнотекстового пошуку та інших SQL функцій, що може створити певні проблеми у майбутньому, якщо наш додаток буде потребувати більш продвинутої логіки обробки даних. MySQL може працювати швидше при обробці простих запитів, але при обробці великих запитів, продуктивність може бути меншою ніж у PostgreSQL, що може бути важливо для магазину, який зберігає багато товарів.

За результатами опитування серед розробників на порталі Stackoverflow, у 2024 році найпопулярнішою реляційною базою даних є PostgreSQL(рис. 1.14).

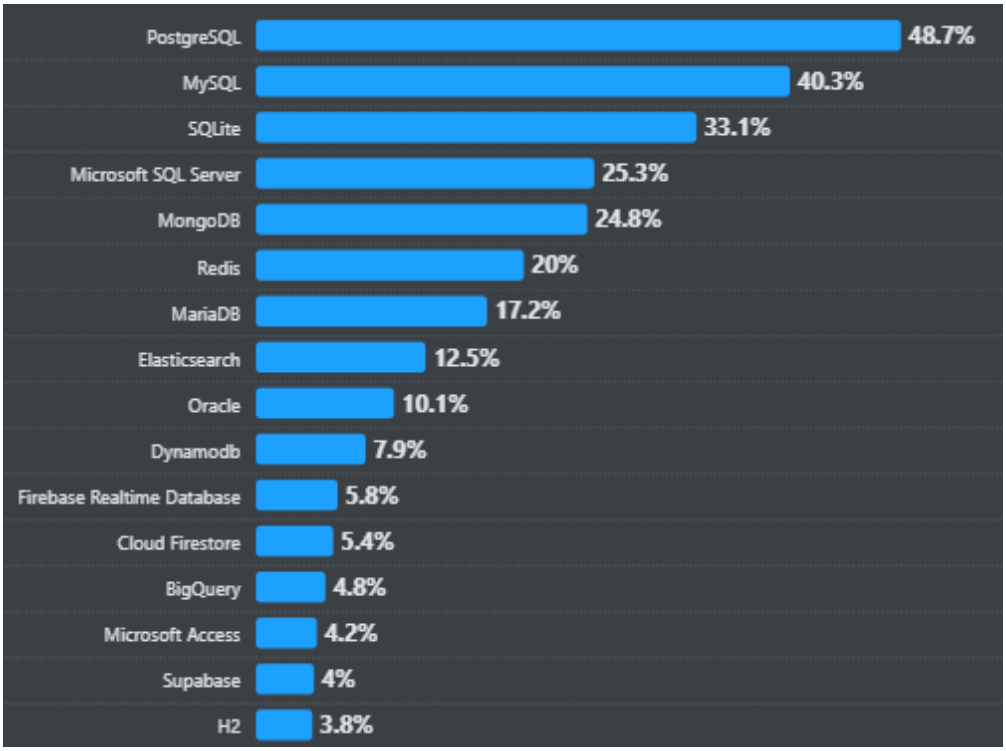


Рисунок 1.14 – Статистика баз даних з порталу Stackoverflow[2]

PostgreSQL підтримує більшість стандартів SQL, такі як підзапити, повний текстовий пошук, агрегатні функції тощо. Це надає змогу працювати з великими і складними запитами без обмежень. Підримка паралельної обробки запитів та кешування, може бути корисною для вебдодатку, який потенційно буде зберігати багато товарів. PostgreSQL гарантує високу консистентність даних за рахунок атомарних транзакцій, що дозволяє безпечно працювати з даними.

Отже, PostgreSQL є найбільш оптимальним вибором вебдодатку “магазин побутової техніки”. Здатність обробляти складні запити, підтримка високої продуктивності при великих навантаженнях та забезпечення узгодженості даних роблять PostgreSQL найкращим та перспективним рішення для масштабованого та багатокористувацького інтернет-магазину.

2 ОГЛЯД АРХІТЕКТУРИ SPRING ТА ЙОГО МОДУЛІВ ДЛЯ РОБОТИ З БАЗОЮ ДАНИХ

2.1 Контейнер інверсії керування Spring IoC

Spring – це потужний, відкритий фреймворк для розробки вебдодатків на Java. Він бере на себе рутинні завдання, щоб розробник міг зосередитися на бізнес-логіці та створенні ефективних рішень.

Spring IoC – це модуль фреймворку, який відповідає за автоматичне створення та управління об'єктами в програмі. Замість того, щоб розробник самостійно створював об'єкти, контейнер бере на себе обов’язки за виконання цієї рутиної роботи[15].

Інверсія керування (Inversion of Control, IoC) – це принцип у розробці програмного забезпечення, який передає керування об’єктами до контейнера Spring IoC. Компоненти програми повинні отримувати свої залежності ззовні, а не створювати їх самостійно. Це забезпечує гнучкість у заміні залежностей і спрощує управління їхнім життєвим циклом[15].

Контейнер Spring IoC. У структурі Spring, інтерфейс ApplicationContext представляє контейнер IoC. Контейнер Spring відповідає за створення, конфігурацію об’єктів, відомих як bean-компоненти, а також за керування їхніми життєвими циклами і автоматичне передавання цих об’єктів через ін’єкцію залежностей[15].

Bean-компоненти – це об’єкти, які були анотовані спеціальними анотаціями (@Component, @Service, @Repository, @Controller) і таким чином були покладені у контейнер Spring.

Ін’єкція залежностей – це шаблон, який ми можемо використовувати для реалізації IoC. Він використовується для зменшення зв'язності між компонентами програми, що спрощує їх тестування, розвиток і підтримку[15].

У контексті ін’єкції залежностей (Dependency Injection, DI) залежності – це об'єкти, від яких залежить певний клас. Іншими словами, це інші компоненти або сервіси, які потрібні класу для виконання його завдань.

В традиційному програмуванні залежність об’єкта створюється за допомогою створення екземпляру Item: “item = new ItemImpl1();” у самому класі Store (рис.2.1).

```
public class Store {
    private Item item;

    public Store() {
        item = new ItemImpl1();
    }
}
```

Рисунок 2.1 – Приклад створення об’єкта в традиційному програмуванні

Використовуючи DI, ми можемо переписати приклад, не вказуючи реалізацію Item. Реалізація об’єкта Item буде надана ззовні. Це може бути реалізовано через конструктор (constructor injection), сеттери (setter injection) або ін’єкція в поле (field injection).

Constructor injection (конструкторна ін’єкція) – залежність передається через конструктор класу (рис.2.2). Найкраща практика, бо забезпечує незмінність (final поля) та гарантує наявність залежності.

```
@Component
public class Store {
    private final Item item;

    public Store(Item item) {
        this.item = item;
    }
}
```

Рисунок 2.2 – Приклад впровадження залежностей на основі конструктора

Setter injection (сеттерна ін’єкція) – залежність передається через

спеціальний метод який називається setter-метод (рис.2.3). Недоліком такого підходу є нестабільність, бо залежність може бути не ініціалізована.

```
@Component
public class Store {
    private final Item item;

    @Autowired
    public void setItem(Item item) {
        this.item = item;
    }
}
```

Рисунок 2.3 – Приклад впровадження залежностей через setter-метод

Field injection (ін'єкція в поле) – Залежність передається напряму в поле класу через анотацію @Autowired (рис.2.4). Такий підхід є поганою практиктикою, бо ускладнює тестування (немає можливості передати залежність через конструктор).

```
@Component
public class Store {

    @Autowired
    private final Item item;
}
```

Рисунок 2.4 – Приклад впровадження залежностей напряму через поле

Рекомендується конструкторна ін'єкція, як найбезпечніша і найкраща для тестування. Сеттерну ін'єкцію краще використовувати якщо залежність не є обов'язковою. Ін'єкція в поле не рекомендується для продуктивного коду.

2.2 Spring Boot

Spring Boot – цей компонент який пряцює як надбудова над Spring, яка спрощує розробку шляхом автоматичного налаштування залежностей, автоконфігурації та вбудованого сервера[16].

Spring Boot має великий функціонал, але його найбільш значущими особливостями є:

— *простота управління залежностями.* Spring Boot спрощує роботу з залежностями, автоматично підбираючи потрібні бібліотеки для різних типів застосунків. Він надає спеціальні "стартери" (наприклад, spring-boot-starter-web для вебзастосунків або spring-boot-starter-data-jpa для роботи з базами даних), які вже містять усі необхідні залежності, щоб розробнику не доводилося додавати їх вручну[16][18].

Стартери – це готові набори залежностей, які можна просто додати в застосунок, звільняючи розробника від необхідності самостійно шукати потрібні бібліотеки та їхні конфігурації. Завдяки цьому процес налаштування значно спрощується. Наприклад, для того щоб почати використовувати Spring Data JPA для доступу до бази даних, до проєкту додається залежність spring-boot-starter-data-jpa і все буде готово (не потрібно шукати сумісні драйвери баз даних і бібліотеки Hibernate). Інакше кажучи, Spring Boot вже зібрав усі необхідні бібліотеки в одному місці, тому розробникам не потрібно щоразу вручну шукати та налаштовувати залежності, достатньо просто підключити потрібний стартер і використовувати його;

— *автоматична конфігурація.* Другою можливістю Spring Boot є автоматична конфігурація. Коли ви додаєте певний стартер-пакет, Spring Boot самостійно налаштовує ваш застосунок на основі підключених залежностей. Наприклад, якщо підключити spring-boot-starter-jdbc, він сам створить необхідні об’єкти для роботи з базою даних і буде зчитувати інформацію для підключення до бази даних з application.properties[17][18].

— *вбудований web-сервер.* Кожен Spring Boot вебзастосунок включає вбудований вебсервер, що дозволяє запускати програму як звичайний Java-додаток із методом main(). Spring Boot може автоматично вбудувати сервери Tomcat або Jetty безпосередньо в застосунок, тому вам не потрібно налаштовувати сервер і розгортати WAR-файли на окремому сервері. Додатки, створені за допомогою Spring Boot, легко упаковуються в JAR або WAR-файли, що спрощує їх розгортання як у традиційних серверних середовищах, так і на хмарних платформах[17].

2.3 Spring MVC

Spring MVC – цей модуль надає інструменти для розробки вебдодатків за шаблоном Model-View-Contoller(MVC), який в свою чергу працює навколо переднього контролера, Dispatcher Servlet(Сервлет диспечера). Dispatcher Servlet обробляє та відправляє всі вхідні HTTP-запити до відповідного контролера[19] [20]. Терміни модель, представлення та контролер визначаються наступним чином:

Модель (Model) – модель містить дані програми. Набір даних може бути одним об'єктом або групою об'єктів. Один об'єкт, наприклад, якщо ви працюєте з магазином, то модель Client може містити дані про конкретного клієнта: ім'я, email, адреса. Група об'єктів, то модель Product може представляти один товар, а список таких товарів буде набором моделей у вигляді колекції (наприклад, List<Product>)[21].

Представлення (View) – це представлення інформації у певному форматі. Для створення сторінки іноді використовується JSP(Java Server Pages). Також, Spring підтримує додаткові технології представлення, такі як Thymeleaf, Apache Velocity[21].

Контролер (Controller) – Контролер обробляє запити користувача і передає їх для відображення у представлення[21].

Dispatcher Servlet. Це передній контролер, який керує всім процесом обробки HTTP-запитів і відповідей[22].

Як працює Spring MVC (рис.2.5):

— *перехоплення запиту*. Коли користувач надсилає запит (наприклад, відкриває сторінку або натискає кнопку), цей запит спочатку потрапляє до DispatcherServlet. Це головний контролер, який керує всіма запитами у Spring MVC[22];

— *пошук контролера*. DispatcherServlet звертається до інтерфесу Handler Mapping, який визначає, який контролер повинен обробити цей запит[22];

— *обробка запиту*. Визначений контролер отримує запит, виконує необхідні операції (наприклад, отримує дані з бази) і повертає об'єкт ModelAndView. Model містить дані, які потрібно відобразити. View визначає, який шаблон (HTML, JSP, Thymeleaf) слід використовувати для відображення даних[22];

— *відображення відповіді*. DispatcherServlet перевіряє view resolver (налаштування, які відповідають за вибір шаблону) і передає дані у відповідний файл відображення, щоб сформувати HTML-сторінку або інший формат відповіді[22].

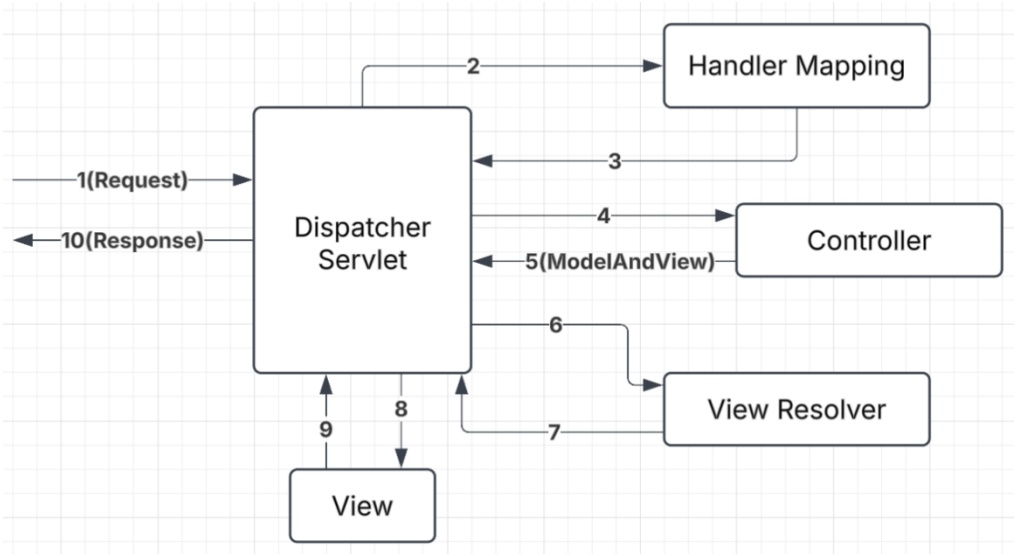


Рисунок 2.5 – Блок-схема роботи MVC Spring

2.4 Spring Security

Spring Security – це потужний фреймворк, який надає комплексні функції безпеки для Java-додатків, включаючи автентифікацію, авторизацію та захист від вразливостей. Як ключовий компонент Spring Framework, він легко інтегрується з проєктами на основі Spring, пропонуючи надійні рішення безпеки. Завдяки вбудованій підтримці різних методів автентифікації та вдосконаленому захисту від загроз, Spring Security гарантує, що ваш додаток залишатиметься безпечним[23].

Автентифікація (Authentication) – процес перевірки особи користувача. Spring Security підтримує різні механізми автентифікації, включаючи збереження облікових записів у пам’яті, використання бази даних через JDBC, OAuth2[23].

Авторизація (Authorization) – контроль доступу до ресурсів додатку. Фреймворк дозволяє визначати рівні доступу до ендпоінтів на основі ролей користувача (Role-Based Access Control – RBAC) або на основі атрибутів (Attribute-Based Access Control – ABAC). Spring Security дозволяє налаштовувати безпеку додатку за допомогою використання анотацій @PreAuthorize, @PostAuthorize, а також конфігурування доступу через SecurityFilterChain[23].

Ендпоінт (endpoint) – це конкретний URL-адрес (маршрут) у вебдодатку, до якого можна звернутися через HTTP-запити (GET, POST, PUT, DELETE). Це точки доступу до функціональності застосунку, які можуть бути захищені механізмами авторизації.

Анотація @PreAuthorize використовується коли потрібно здійснити перевірку перед виконанням метода.

```
@DeleteMapping("/orders/{id}/delete")
@PreAuthorize("hasRole('ADMIN')")
public void deleteProduct(Long productId) {
    // Логіка видалення продукту
}
```

Рисунок 2.6 – Приклад використання анотації @PreAuthorize

Тільки користувач з ролю адмін зможе здійснити видалення продукту (рис.2.6).

Анотація @PostAuthorize використовується коли потрібно здійснити перевірку після виконанням метода.

```
@GetMapping("/orders/{id}")
@PostAuthorize("returnObject.owner == authentication.name")
public Order getOrderById(Long orderId) {
    // Логіка отримання замовлення
    return new Order(); // Симуляція повернення об'єкта
}
```

Рисунок 2.7 – Приклад використання анотації @PostAuthorize

Користувач може переглядати лише ті замовлення, де він є власником (рис.2.7).

SecurityFilterChain. Дозволяє визначити правила доступу до ендпоінтів.

```
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
            .csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                    .requestMatchers("/admin/**").hasRole("ADMIN") // Доступ тільки для ADMIN
                    .requestMatchers("/user/**").hasAnyRole( ...roles: "USER", "ADMIN") // USER і ADMIN можуть заходити
                    .anyRequest().authenticated() // Усі інші запити потребують автентифікації
            )
            .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS));
    return http.build();
}
```

Рисунок 2.8 – Приклад конфігурування доступу через SecurityFilterChain

Ендпоінт “/admin/**” доступний лише для користувача який має роль ADMIN. Ендпоінт “/user/**” доступний для користувачів які мають роль USER та ADMIN. Всі інші запити вимагають автентифікації (рис.2.8).

Role-Based Access Control (RBAC) – Контроль доступу на основі ролей. У RBAC доступ до ресурсів визначається роллю користувача (наприклад, USER, ADMIN, MODERATOR). Якщо користувач має певну роль, він отримує доступ до відповідних ендпоінтів або функцій. Приклад RBAC у Spring Security (рис2.8). Приклад RBAC у методах з @PreAuthorize (рис2.6).

Attribute-Based Access Control (ABAC) – Контроль доступу на основі атрибутів. У моделі ABAC контроль доступу визначається не лише роллю, а й додатковими атрибутами користувача або ресурсу.

```
@PostMapping("/{id}/update")
@PreAuthorize("hasRole('EMPLOYEE') or #id == authentication.principal.id")
public String update(@PathVariable("id") Long id, @Validated @ModelAttribute("clientUpdate") ClientUpdate clientUpdate,
                     BindingResult result, Principal principal, RedirectAttributes redirectAttributes) {
```

Рисунок 2.9 – Приклад ABAC у Spring Security

Користувач може отримати змінити дані тільки про себе. Робітник (EMPLOYEE) може змінити дані про іншого користувача (рис.2.9).

Можливість створення власних фільтрів, механізмів автентифікації та авторизаційних правил забезпечує гнучкість та дозволяє інтегрувати специфічні бізнес-логіки без шкоди для рівня захисту. Це робить Spring Security універсальним рішенням для побудови безпечних вебдодатків різної складності.

2.5 Spring Data

Spring Data – це компонент який значно спрощує роботу з базами даних, надаючи абстракцію доступу до даних та підтримку для різних технологій доступу, таких як JPA, Hibernate та JDBC. Він автоматизує процеси створення репозиторіїв та операцій з даними.

Spring Data – це модуль, який забезпечує зручну абстракцію для роботи з базамами даних, позбавляючи розробника написанню рутинного коду та спрощуючи процес взаєможії з даними.

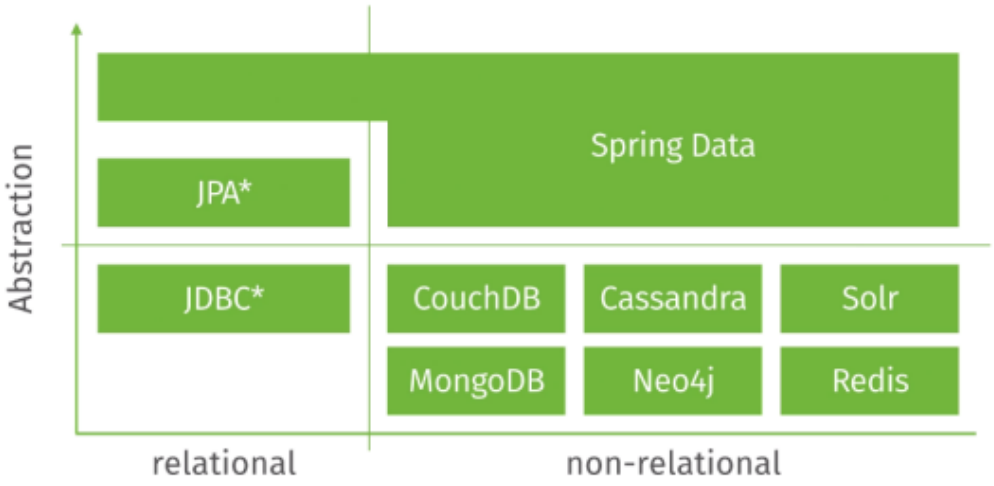


Рисунок 2.10 – Підмодулі Spring Data

Підмодулі Spring Data для роботи з реляційними базами даних. Spring Data містить в собі різні підмодулі для доступу до конкретної бази даних. В рамках застосунку “магазин побутової техніки” використовується реляційна база даних, тому розглядаються два основні підмодулі для роботи з нею: Spring JDBC та Spring Data JPA.

Spring JDBC – це модуль Spring, який спрощує взаємодію з базою даних через JDBC (Java Database Connectivity)[14].

Основні можливості Spring JDBC:

— *JdbcTemplate*. Основний інструмент для взаємодії з базою даних, за допомогою якого ми виконуємо SQL запити;

— *RowMapper*. Надає можливість перетворювати результати запиту в об’єкти Java;

— *перетворення exceptions у DataAccessException.* Spring обгортає винятки низького рівня у DataAccessException або один із його підкласів[13].

Плюси:

— *прямий контроль над SQL-запитами*. Spring JDBC дозволяє явно писати SQL запити, надаючи повний контроль над взаємодією з базою даних;

— *простота для малих проєктів.* для простих застосунків Spring JDBC може бути легшим у використанні без складності ORM[13].

Мінуси:

— *більше шаблонного коду*. Навіть з JdbcTemplate усе ще потрібно писати та керувати SQL-запитами та вручну відображати результати;

— *не підримує ORM*. потрібно в ручну перетворювати результати запитів у Java об’єкти;

— *менше автоматизації*. Навідмінну від JPA, JDBC не надає автоматичних механізмів для свторення, оновлення, видалення таблиць або записів[13].

Spring Data JPA – Spring Data JPA є потужною абстракцією над JPA. Він значно спрощує роботу з базою даних, автоматично реалізуючи основні CRUD-операції. За замовчуванням Spring Data JPA використовує Hibernate як ORM провайдера.

JPA (Java Persistence API) – це інтерфейс для роботи з реляційними базами даних у Java. Він визначає стандартний набір правил та анотацій для збереження, оновлення, читання та видалення даних у базі. Тобто JPA не є самим інструментом для роботи з базою даних, а задає лише специфікацію, яку реалізують ORM-фреймворки, такі як Hibernate.

ORM (Object Relational Mapping) – це технологія, яка забезпечує автоматичне перетворення даних між світом програмування та реляційними базами даних.

Основні характеристики:

— *JpaRepository*. Дозволяє автоматично використовувати велику кількість базових методів, таких як save(), findById(), findAll(), deleteById() та інші, без необхідності ручного написання SQL[13];

```
public interface ClientRepository extends JpaRepository<Client, Long> {
    1 usage  new *
    User findByEmail(String email);
```

Рисунок 2.11 – Приклад JpaRepository

— *запити за допомогою JPQL (Java Persistence Query Language) або Native Query*.

JPQL (Java Persistence Query Language) – це об'єктно-орієнтована мова запитів, яка працює з сутностями (Entity) замість таблиць бази даних.

Приклад операції читання за допомогою JpaRepository використовуючи JPQL (рис. 2.12):

1. @Query – вказує, що метод використовує кастомний JPQL-запит;
2. Client c – це назва Entity-класу, не таблиці;
3. c.email – це поле класу Client;
4. :email – це іменований параметр, який підставляється через @Param("email").

```
public interface ClientRepository extends JpaRepository<Client, Long> {
    1 usage  new *
    @Query("SELECT c FROM Client c WHERE c.email = :email")
    User findByEmail(@Param("email") String email);
```

Рисунок 2.12 – Операція читання використовуючи JPQL

Native Query – це звичайний SQL-запит, який виконується напряму до бази даних. Приклад операції читання за допомогою JpaRepository використовуючи Native Query (рис. 2.13):

1. value = "..." – звичайний SQL-запит;
2. nativeQuery = true – вказує, що це не JPQL, а саме native SQL;
3. client – це назва таблиці в базі даних;
4. email – це назва стовпця.

```
public interface ClientRepository extends JpaRepository<Client, Long> {
    1 usage  new *
    @Query(value = "SELECT * FROM clients WHERE email = :email", nativeQuery = true)
    User findByEmail(@Param("email") String email);
```

Рисунок 2.13 – Операція читання використовуючи Native Query

Hibernate генерує SQL автоматично. Запити за допомогою JPQL або Native Query доцільно використовувати тоді, коли стандартних методів репозиторію недостатньо для реалізації складної логіки вибірки даних, або коли потрібна оптимізація продуктивності шляхом написання точних SQL-запитів;

— *автоматичне створення та збереження таблиць та об'єктів*. За допомогою анотацій @Entity та @Id можна визначити об'єкти Java як сутності, які будуть зберігатися в базі даних. Spring Data JPA автоматично відслідковує зміни цих об'єктів і синхронізує їх із базою даних. Анотація @Entity вказує, що клас є сутністю і повинен зберігатися в базі даних, а @Id позначає поле як первинний ключ цієї сутності;

```java
@Entity
@Table(name = "persons")
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    no usages
    @Column(name = "name")
    private String name;
}
```

Рисунок 2.14 – Приклад використання анотацій @Entity та @Id

— *підтримка пагінації та сортування за допомогою інтерфейса Pageble*. Метод findAll(Pageable pageable) автоматично виконує запит до бази з урахуванням переданих параметрів сторінки та сортування. Наприклад, у фрагменті коду (рис. 2.15), створюється об'єкт PageRequest для отримання першої сторінки (індекс 0), яка містить 10 користувачів, відсортованих за іменем у зростаючому порядку;

```java
Page<User> findAll(Pageable pageable);

PageRequest pageRequest = PageRequest.of(0, 10, Sort.by("name").ascending());
Page<User> usersPage = userRepository.findAll(pageRequest);
```

Рисунок 2.15 – Приклад використання інтерфейса Pageble

— *підтримка транзакцій*. Spring Data JPA інтегрується з Spring Transaction Management, що дозволяє зручно працювати з транзакціями для забезпечення цілісності даних.

```java
@Transactional(readOnly = true)
public List<User> getAllUsers() {
    return userRepository.findAll();
}
```

Рисунок 2.16 – Приклад використання транзакціїї в Spring Data JPA

Краще використовувати @Transactional(readOnly = true) для запитів тільки на читання, щоб Hibernate не тримав зміни в пам'яті.

— *анотації для створення зв'язків між сутностями*. Spring Data JPA дозволяє легко моделювати зв'язки між таблицями бази даних за допомогою JPA-анотацій.

OneToOne. Анотація @OneToOne означає, що один запис в одній таблиці відповідає одному запису в іншій таблиці. @JoinColumn(name = "user_id") – вказує назву зовнішнього ключа в поточній таблиці, який посилається на первинний ключ в таблиці. Припустимо, кожен клієнт (Client) має окремий профіль користувача (User). Тобто один клієнт дорівнює один користувач. Можна зробити двосторонній зв'язок, вказавши, з якого боку він керується. Параметр mappedBy = "user" – означає, що ця сторона є не основною, а керування зв'язком здійснюється з боку Client(рис. 2.17).

```
// У класі Client
@OneToOne
@JoinColumn(name = "user_id")
private User user;

// У класі User
@OneToOne(mappedBy = "user")
private Client client;
```

Рисунок 2.17 – Приклад використання анотації @OneToOne

ManyToOne. Анотація @ManyToOne вказує на зв'язок «багато до одного» – багато записів з поточної таблиці можуть посилатися на один запис іншої(рис. 2.18). Анотація @JoinColumn(name = "employee_id") – визначає назву зовнішнього ключа в цій таблиці. @NotNull вказує на те, що поле не може бути порожнім.

```
@ManyToOne
@JoinColumn(name = "employee_id")
@NotNull
private Employee employee;
```

Рисунок 2.18 – Приклад використання анотації @ManyToOne

ManyToMany. Анотація @ManyToMany – вказує на зв’язок «багато до багатьох», клієнт може мати багато улюблених товарів, і кожен товар може бути улюбленим у кількох клієнтів. Анотація @JoinTable – створює окрему таблицю client_favorite_appliances яка використовується для реалізації зв’язку. joinColumns = @JoinColumn(name = "client_id") – колонка, що містить зовнішній ключ до таблиці Client. inverseJoinColumns = @JoinColumn(name = "appliance_id") – колонка, що містить зовнішній ключ до таблиці Appliance.

```
@ManyToMany
@JoinTable(
    name = "client_favorite_appliances",
    joinColumns = @JoinColumn(name = "client_id"),
    inverseJoinColumns = @JoinColumn(name = "appliance_id")
)
private List<Appliance> favoriteAppliances;
```

Рисунок 2.19 – Приклад використання анотації @ManyToMany

Для двостороннього зв’язку @ManyToMany потрібно додати відповідне поле і в іншій сутності (наприклад, List<Client> clients; в Appliance) і позначити його як mappedBy = "favoriteAppliances";

— *валідація даних за допомогою анотацій із пакету jakarta.validation.constraints*. Ці анотації використовуються для автоматичної перевірки правильності введених даних перед їхнім збереженням у базі даних або обробкою у бізнес-логіці.

@NotNull – гарантує, що значення поля не може бути null.

@Size(min=, max=) – використовується для перевірки довжини рядків або кількості елементів у колекціях. Наприклад, для валідації імені користувача можна вказати обмеження мінімальної та максимальної кількості символів.

@Min(value=) та @Max(value=) – обмежують мінімальне та максимальне допустиме значення числових полів.

@Positive та @PositiveOrZero – перевіряють, чи є числове значення

додатним або нульовим. Використовується, наприклад, для цін товарів або кількості на складі.

@Email – перевіряє, що рядок має правильний формат електронної адреси (наявність символу "@" і коректного домену).

@Pattern(regexp=) – дозволяє перевірити, чи відповідає рядок певному регулярному виразу. Це корисно, наприклад, для перевірки форматів номерів телефонів або поштових індексів.

Використання цих анотацій допомагає централізовано контролювати якість даних на рівні моделі та значно зменшує кількість ручної перевірки у коді сервісів і контролерів.

Переваги використання Spring Data JPA:

— *автоматизація базових операцій*. Більшість стандартних операцій (збереження, пошук, видалення, оновлення) можна виконати без написання SQL;

— *гнучкість у створенні запитів*. Можна комбінувати автоматично згенеровані запити з написанням власних;

— *менше шаблонного коду*. Інтерфейси типу JpaRepository або CrudRepository забезпечують готову реалізацію, і немає потреби кожного разу писати одні й ті самі методи;

— *інтеграція з екосистемою Spring*. Легка інтеграція з іншими модулями Spring'a: Spring Security, Spring Boot, Spring Transaction Management;

— *підтримка пагінації та сортування*. Можна легко отримувати дані частинами (Pageable) та задавати порядок без ручного дописування SQL;

— *транзакційність за замовчуванням*. Автоматичне управління транзакціями через анотацію @Transactional, менше шансів порушити цілісність даних;

— *підтримка зв'язків між сутностями*. Легко налаштовувати відношення типу @OneToMany, @ManyToOne, @ManyToMany через анотації без потреби писати SQL для з'єднання таблиць.

Недоліки використання Spring Data JPA:

— *низька продуктивність при великих або складних запитах*. При

використанні автогенерації Hibernate може будувати неефективні SQL-запити. Для складних випадків часто потрібно переходити на нативний SQL або окремий підхід;

— *необхідність глибокого розуміння JPA/Hibernate*. Для правильного налаштування кешування, lazy loading, роботи з транзакціями треба знати нюанси JPA і Hibernate;

— *велика магія "під капотом"*. Автоматичні механізми іноді можуть вести себе неочікувано, особливо коли проєкт стає великим і має складні зв'язки між об'єктами.

Таблиця 2.1 – Порівняння Spring JDBC та Spring Data JPA

Характеристика	Spring JDBC	Spring Data JPA
Написання SQL-запитів	Так	Не обов’язково (можна використовувати JPQL)
ORM підтримка	Ні	Так (Hibernate)
Автоматичне створення таблиць	Ні	Так (@Entity)
CRUD операції	Потрібно писати вручну	Автоматично через JpaRepository
Контроль над SQL-запитами	Високий	Середній (JPA генерує SQL)
Продуктивність	Висока (прямий доступ)	Залежить від налаштувань ORM
Кількість шаблонного коду	Висока	Мінімальна

Spring JDBC підходить для проєктів, де потрібен повний контроль над SQL-запитами. Spring Data JPA ідеальний вибір для більшості застосунків, оскільки спрощує роботу з базою даних та автоматизує рутинні операції.

3 ПРОЄКТУВАННЯ БАЗИ ДАНИХ

3.1 Аналіз вимог до структури бази даних

Проєктування бази даних для інтернет-магазину побутової техніки починається з визначення основних сутностей, що відображають предметну область і забезпечують ефективне управління товарами, клієнтами, замовленнями та іншими бізнес-процесами.

Основні сутності бази даних. Для функціонування магазину побутової техніки необхідно визначити ключові сутності, які відображають предметну область[24]:

— *User (Користувач).* Містить інформацію про всіх користувачів (спільні поля для клієнтів та працівників);

— *Client (Клієнт)*. Розширює сутність User, містить додаткове поле card для збереження даних про клієнтську карту;

— Employee (Працівник). Також базується на User, містить поле department для збереження даних про відділ співробітника;

— *Appliance (Товар)*. Містить дані про побутову техніку, включаючи категорію (Category), виробника (Manufacturer), тип живлення (PowerType), модель, характеристики, опис, потужність та ціну;

— *Manufacturer (Виробник).* Містить інформацію про виробників техніки;

— *Orders (Замовлення)*. Містить інформацію про клієнта (Client) який створив замовлення, працівника (Employee) який обробляє замовлення, список товарів (orderRowSet) та статус підтвердження (approved/unapproved);

— *OrderRow (Рядок замовлення). З*в'язує конкретний товар (Appliance) із замовленням, вказує його кількість (number) та суму (amount);

— *Category (Категорія). Е*num, визначає категорію техніки (BIG, SMALL);

— *PowerType (Тип живлення)*. Enum, визначає тип живлення (AC220, AC110, ACCUMULATOR).

Функціональні вимоги до бази даних. База даних повинна забезпечувати:

— збереження інформації про товари та їх характеристики (назва, модель, виробник, потужність, ціна тощо);

— оформлення замовлень (збереження даних про клієнтів, товари у замовленні, статуси замовлення);

— облік фінансових операцій (ціни товарів, загальна сума замовлення);

— категоризацію товарів для зручності пошуку та фільтрації;

— забезпечення зв'язків між сутностями для уникнення дублювання інформації.

Нефункціональні вимоги. Крім основного функціоналу, база даних повинна відповідати таким вимогам:

— *продуктивність*. Швидка обробка запитів (особливо під час пошуку товарів та обробки замовлень);

— *цілісність даних*. Використання обмежень цілісності (унікальність, зовнішні ключі, обов’язкові поля);

— *масштабованість*. Можливість розширення бази без значного перегляду структури;

— *резервне копіювання*. Забезпечення можливості відновлення у разі збою.

Вибір СУБД. Як вже було визначено в розділі 1, пункт 1.3, для реалізації бази даних обрана PostgreSQL, оскільки вона підтримує складні зв’язки між сутностями, дозволяє обробляти складні запити, має високу продуктивність при великих навантаженнях, а також вбудовані механізми транзакцій та індексування роблять PostgreSQL найкращим та перспективним рішення для масштабованого та багатокористувацького інтернет-магазину.

3.2 Проєктування моделі бази даних

Процес проєктування бази даних складається з кількох послідовних етапів, ключовими з яких є створення логічної моделі(опис структури, властивостей і зв’язків сутностей), та побудова фізичної моделі, яка буде реалізована у вибраній СУБД (PostgreSQL).

Логічна модель бази даних. Визначає основні сутності, їхні атрибути та зв’язки між ними. На основі аналізу предметної області, у розділі 3, пункт 3.1, було виділено такі основні сутності:

— *User*. Зберігає інформацію про користувачів системи. id (Long) – унікальний ідентифікатор користувача. name (String) – ім'я користувача. email (String) – електронна пошта. password (String) – пароль;

— *Employee*. Є підкласом User, що розширює його додатковим атрибутом: department (String) – відділ, у якому працює співробітник;

— Client. також підклас User, що має додатковий атрибут: card (String) – номер картки клієнта;

— *Orders*. Представляє замовлення, яке може містити декілька позицій. id (Long) – унікальний ідентифікатор замовлення. employee (Employee) – працівник, який обробив замовлення. client (Client) – клієнт, що зробив замовлення. orderRowSet (Set) – набір товарів у замовленні. approved (Boolean) – статус підтвердження замовлення;

— *Manufacturer*. Містить інформацію про виробників товарів. id (Long) – унікальний ідентифікатор виробника. name (String) – назва виробника;

— *Appliance*. Описує товари, які можна замовити. id (Long) – унікальний ідентифікатор товару. name (String) – назва товару. category (Category) – категорія товару (перерахування BIG або SMALL). model (String) – модель товару. manufacturer (Manufacturer) – виробник товару. powerType (PowerType) – тип живлення (AC220, AC110, ACCUMULATOR). characteristic (String) –

характеристики товару. description (String) – опис товару. power (Integer) – потужність товару. price (BigDecimal) – ціна товару;

— *OrderRow. П*редставляє один рядок у замовленні. id (Long) – унікальний ідентифікатор рядка. appliance (Appliance) – товар у замовленні. number (Long) – кількість одиниць товару. amount (BigDecimal) – загальна сума по цьому товару в замовленні.

Зв'язки між сутностями. В логічній моделі бази даних зв'язки між сутностями визначають структуру взаємодії між об'єктами предметної області. У розробленій моделі використовуються як зв'язки наслідування, так і асоціації між сутностями, що дозволяє точно передати бізнес-логіку системи.

Сутності Employee та Client є підкласами сутності User, тобто реалізують зв'язок успадкування. Це означає, що і клієнти, і співробітники мають загальні атрибути, такі як id, name, email, password, але також володіють унікальними властивостями: Employee має атрибут department, а Client – атрибут card.

Між сутністю Orders та сутністю Employee встановлено зв'язок типу багато до одного (many-to-one). Це означає, що кожне замовлення обробляється лише одним співробітником, але один співробітник може обробити багато замовлень. Аналогічний тип зв'язку багато до одного існує між Orders і Client: одне замовлення належить лише одному клієнту, проте клієнт може створити багато замовлень.

Кожне замовлення (Orders) може включати декілька позицій товарів, що реалізується через зв'язок із сутністю OrderRow. Цей зв'язок має тип один до багатьох (one-to-many) і буде організований через проміжну таблицю order_order_row, а не пряме зовнішнє посилання order_id в OrderRow. Підхід з проміжною таблицею не є класичним рішенням, але в цьому випадку має свої переваги, а саме:

— *нормалізація та чистота структури даних*. OrderRow стає незалежною сутністю, яка не прив'язана до якогось конкретного замовлення;

— *менше тісного зв'язування*. OrderRow не знає про Order, що зменшує зв'язність (coupling) і дозволяє легше змінювати структуру в майбутньому;

— *відсутність перенасиченості в OrderRow*. Якщо OrderRow це виключно модель рядка з товарами, то краще зберігати її максимально універсальною, без прив'язки до конкретного замовлення, щоб уникнути дублювання інформації при повторному використанні.

Кожен OrderRow відноситься до одного товару (Appliance), тому зв'язок між OrderRow і Appliance є багато до одного (many-to-one).

Сутність Appliance пов'язана з Manufacturer через зв'язок типу багато до одного (many-to-one): кожен товар має одного виробника, але виробник може виробляти багато товарів. Крім того, Appliance має атрибути category та powerType, які реалізовані як перерахування (enum). Зв'язок із ними є один до одного (one-to-one), оскільки кожен товар належить до однієї категорії (наприклад, BIG або SMALL) і має лише один тип живлення (наприклад, AC220, AC110 або ACCUMULATOR).

Така структура дозволяє забезпечити гнучке та логічно цілісне представлення даних у майбутній фізичній реалізації бази даних.

Фізична модель. Відображає логічну структуру у вигляді таблиць реляційної бази даних. Для реалізації використовується PostgreSQL, яка підтримує сучасні можливості моделювання даних, зокрема наслідування таблиць, зовнішні ключі, типи ENUM тощо.

— *Users*. Ця таблиця не створюється, бо її дані будуть успадковані. Це лише базовий клас, і вона не створюється у базі даних;

— *Employees.* Представляє користувачів системи з роллю працівник.

id – унікальний ідентифікатор працівника (первинний ключ, PRIMARY KEY), автоматично генерується (BIGSERIAL).

name – ім'я працівника. Обов'язкове поле (NOT NULL).

email – електронна адреса працівника. Обов'язкове та унікальне поле (NOT NULL, UNIQUE) для ідентифікації працівника.

password – пароль для доступу до системи. Обов'язкове поле (NOT NULL).

department – відділ, у якому працює співробітник. Обов'язкове поле (NOT NULL);

```
CREATE TABLE employees (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    department VARCHAR(255) NOT NULL
);
```

Рисунок 3.1 – Скрипт для створення таблиці employees

— *Clients. З*берігає інформацію про клієнтів, які можуть створювати замовлення.

id – унікальний ідентифікатор клієнта (первинний ключ, PRIMARY KEY), автоматично генерується (BIGSERIAL).

name – ім’я клієнта. Обов’язкове поле (NOT NULL).

email – електронна адреса клієнта. Обов’язкове та унікальне поле (NOT NULL, UNIQUE) для ідентифікації користувача.

password – пароль для входу. Обов’язкове поле (NOT NULL).

card – номер платіжної картки. Обов’язкове та унікальне поле (NOT NULL, UNIQUE) для ідентифікації клієнта та безпеки платежів;

```
CREATE TABLE clients (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    card VARCHAR(50) NOT NULL UNIQUE
);
```

Рисунок 3.2 – Скрипт для створення таблиці clients

— *створення перерахуваннь (ENUM)*.

Category – цей тип використовується для класифікації побутової техніки за розміром. 'BIG' велика побутова техніка. 'SMALL' дрібна побутова техніка.

```
CREATE TYPE category AS ENUM ('BIG', 'SMALL');
```

Рисунок 3.3 – Скрипт для створення типу category

Power type – Цей тип визначає тип живлення побутового пристрою. 'AC220' змінний струм 220В (стандарт для Європи). 'AC110' змінний струм 110В (стандарт для США, Японії). 'ACCUMULATOR' живлення від акумулятора (наприклад, бездротові прилади).

```
CREATE TYPE power_type AS ENUM ('AC220', 'AC110', 'ACCUMULATOR');
```

Рисунок 3.4 – Скрипт для створення типу power_type

Використання ENUM забезпечує контроль правильності введення даних, не дозволяючи записати інші значення, окрім визначених. Це спрощує перевірку логіки на прикладному рівні та підвищує надійність бази даних;

— *Orders*. Зберігає інформацію про замовлення, створені клієнтами.

id – унікальний ідентифікатор замовлення (первинний ключ, PRIMARY KEY), автоматично генерується (BIGSERIAL).

employee_id – зовнішній ключ на таблицю employee, який вказує на працівника, що обслуговує замовлення. Обов’язкове поле (NOT NULL).

client_id – зовнішній ключ на таблицю client, який вказує на клієнта, що створив замовлення. Обов’язкове поле (NOT NULL).

approved – логічне поле, яке позначає, чи підтверджено замовлення. За замовчуванням встановлено FALSE;

```
CREATE TABLE orders (
    id BIGSERIAL PRIMARY KEY,
    employee_id BIGINT NOT NULL REFERENCES employees(id),
    client_id BIGINT NOT NULL REFERENCES clients(id),
    approved BOOLEAN DEFAULT FALSE
);
```

Рисунок 3.5 – Скрипт для створення таблиці Orders

— *Manufacturers.* Описує інформацію про виробників товарів.

id – унікальний ідентифікатор виробника (первинний ключ, PRIMARY KEY), автоматично генерується (BIGSERIAL).

name – назва виробника. Обов’язкове поле (NOT NULL);

```
CREATE TABLE manufacturers (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
);
```

Рисунок 3.6 – Скрипт для створення таблиці Manufacturers

— *Appliance.* Описує одиницю побутової техніки.

id – унікальний ідентифікатор приладу (первинний ключ, PRIMARY KEY), автоматично генерується (BIGSERIAL).

name – назва приладу. Обов’язкове поле (NOT NULL).

category – категорія приладу. Тип даних перелічення (ENUM) category, обов’язкове поле (NOT NULL).

model – модель приладу. Обов’язкове поле (NOT NULL).

manufacturer_id – зовнішній ключ на таблицю manufacturers, що вказує на виробника. Обов’язкове поле (NOT NULL).

power_type – тип живлення приладу (наприклад, електричний, батарейковий тощо). Тип даних ENUM power_type. Обов’язкове поле (NOT NULL).

characteristic – технічні характеристики приладу. Обов’язкове текстове поле (NOT NULL).

description – додатковий опис приладу. Необов’язкове поле.

power – споживана потужність приладу (у Вт). Обов’язкове поле (NOT NULL) з обмеженням: значення має бути більше 0 (CHECK (power > 0)).

price – ціна приладу. Обов’язкове поле (NOT NULL) з обмеженням: значення має бути більше або дорівнювати 0 (CHECK (price >= 0));

```
CREATE TABLE appliances (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    category category NOT NULL,
    model VARCHAR(100) NOT NULL,
    manufacturer_id BIGINT NOT NULL REFERENCES manufacturers(id),
    power_type power_type NOT NULL,
    characteristic TEXT NOT NULL,
    description TEXT,
    power INTEGER NOT NULL CHECK (power > 0),
    price NUMERIC(10, 2) NOT NULL CHECK (price >= 0)
);
```

Рисунок 3.7 – Скрипт для створення таблиці Appliances

— *Order_rows*. Містить інформацію про окремі позиції в замовленні (товари).

id – унікальний ідентифікатор рядка замовлення (первинний ключ, PRIMARY KEY), автоматично генерується (BIGSERIAL).

appliance_id – зовнішній ключ на таблицю appliance, що вказує на товар. Обов’язкове поле (NOT NULL).

number – кількість одиниць товару в даному рядку замовлення. Обов’язкове поле (NOT NULL) з обмеженням, значення має бути більше 0 (CHECK (number > 0)).

amount – загальна вартість товару у цьому рядку замовлення. Обов’язкове поле (NOT NULL) з обмеженням, значення має бути більше або дорівнювати 0 (CHECK (amount >= 0));

```
CREATE TABLE order_rows (
    id BIGSERIAL PRIMARY KEY,
    appliance_id BIGINT NOT NULL REFERENCES appliances(id),
    number BIGINT NOT NULL CHECK (number > 0),
    amount NUMERIC(10, 2) NOT NULL CHECK (amount >= 0)
);
```

Рисунок 3.8 – Скрипт для створення таблиці Order_rows

— *Order_order_row*. Проміжна таблиця для реалізації зв’язку між таблицями orders та order_rows (зв'язок OneToMany).

order_id – зовнішній ключ на таблицю orders, який вказує на замовлення. order_row_id – зовнішній ключ на таблицю order_rows, який вказує на конкретну позицію товару в замовленні.

PRIMARY KEY (order_id, order_row_id) – складовий первинний ключ, який забезпечує унікальність кожної пари "замовлення–рядок замовлення";

```
CREATE TABLE order_order_row (
    order_id BIGINT NOT NULL REFERENCES orders(id),
    order_row_id BIGINT NOT NULL REFERENCES order_rows(id),
    PRIMARY KEY (order_id, order_row_id)
);
```

Рисунок 3.9 – Скрипт для створення проміжної таблиці order_order_rows

ER-діаграма бази даних (Entity-Relationship Diagram), є графічним представленням структури даних, їхніх атрибутів та зв’язків між сутностями. Це дозволяє візуально відобразити логічні залежності, що існують у предметній області.

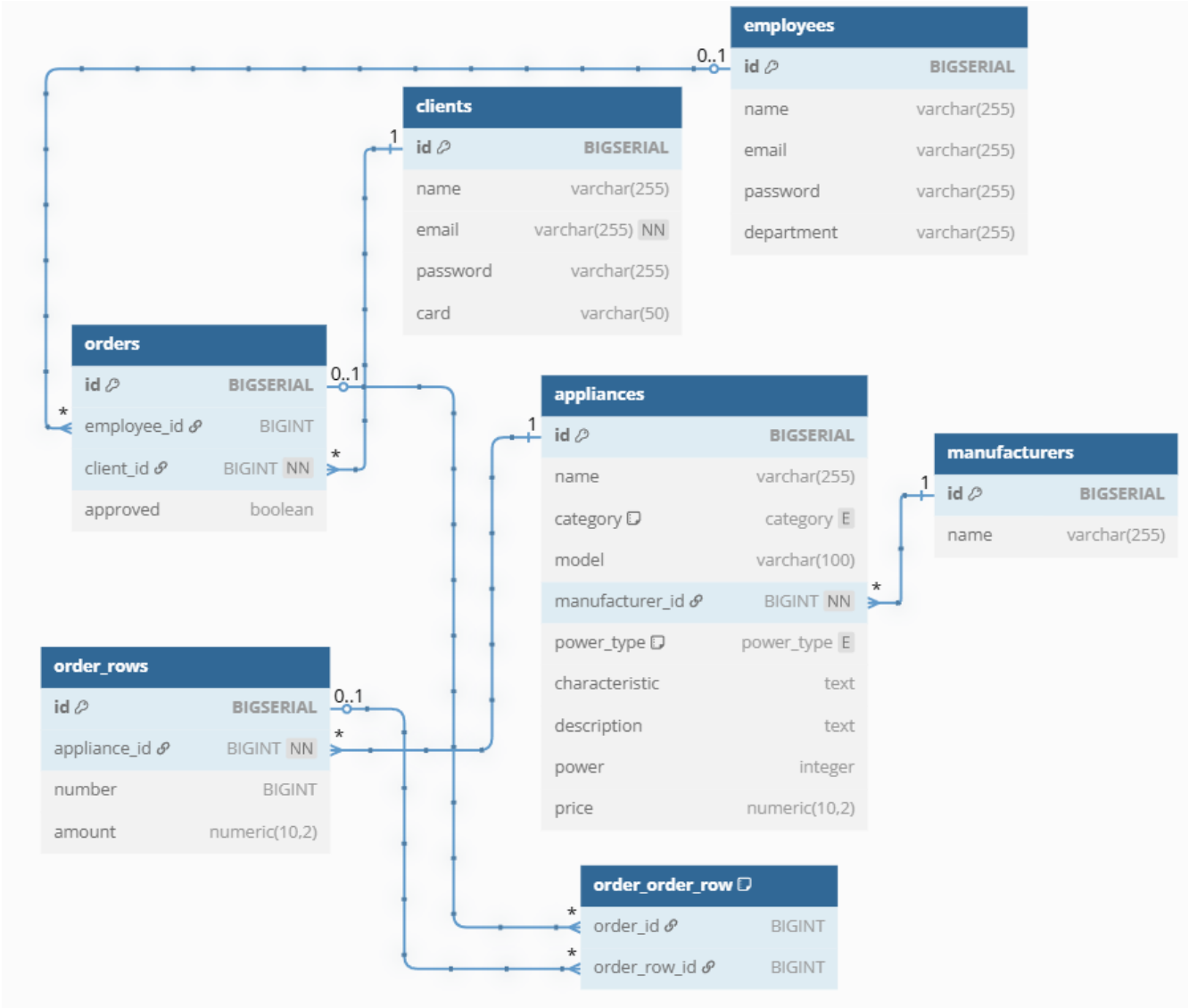


Рисунок 3.10 – ER діаграма для вебдодатку Магазин побутової техніки[26]

3.3 Оптимізація бази даних: індекси, зовнішні ключі та обмеження

Процес оптимізації структури бази даних є важливою частиною проєктування, оскільки він покращує ефективність системи, забезпечує цілісність даних і дозволяє уникнути дублювання. У реалізованій моделі бази даних з використанням PostgreSQL були впроваджені такі механізми оптимізації:

— *індекси*. За замовчуванням, усі первинні ключі (PRIMARY KEY) автоматично супроводжуються унікальним індексом, який дозволяє швидко здійснювати пошук, сортування та з'єднання таблиць за цими полями. Зокрема, індекси створюються для полів id у таблицях clients, employees, orders, order_rows, appliances, manufacturers. Вони забезпечують ефективний доступ до записів за ідентифікатором. Для полів email у таблицях clients та employees. Поля мають обмеження UNIQUE, що автоматично супроводжується індексом для швидкої перевірки унікальності електронної пошти. А також поле card у таблиці clients має унікальність, отже індексується для швидкого пошуку клієнтів за номером картки.

Для оптимізації продуктивності системи, особливо в сценаріях частого пошуку або фільтрації за моделлю побутової техніки, доцільно створити кастомний індекс по полю category у таблиці appliances. Такий індекс дозволяє значно зменшити час відповіді при виконанні запитів, що включають умову WHERE category = '...', оскільки забезпечує прямий доступ до відповідних рядків без необхідності повного сканування таблиці.

Це особливо важливо при великому обсязі даних, наприклад, у випадках, коли система обробляє десятки або сотні тисяч записів про товари.

```
CREATE INDEX idx_appliances_category ON appliances(category);
```

Рисунок 3.11 – Скрипт для створення індексу по полю category

— *зовнішні ключі*. Щоб забезпечити посилальну цілісність даних у структурі бази були визначені зовнішні ключі (FOREIGN KEY), які встановлюють зв’язки між таблицями. Вони виконують функцію контролю узгодженості при додаванні, оновленні та видаленні записів.

Зв’язок manufacturer_id у таблиці appliances посилається на id у таблиці manufacturers.

У таблиці orders, поле client_id посилається на таблицю clients.

Поле employee_id пов’язане з employees.

У таблиці order_rows, appliance_id зв'язується з appliances.

Проміжна таблиця order_order_row містить два зовнішні ключі – на orders та order_rows;

— *обмеження*. Використання обмежень (CONSTRAINTS) дозволяє не лише підвищити цілісність даних, а й частково реалізувати бізнес-логіку на рівні бази.

NOT NULL – використовується для обов’язкових атрибутів (наприклад, name, email, password у користувачів, model у товарів тощо), що гарантує наявність значення.

UNIQUE – застосовано до полів email та card у таблиці clients, email у employees для забезпечення унікальності записів у системі.

CHECK – обмеження використовуються для числових полів, щоб уникнути некоректних даних.

4 НАЛАШТУВАННЯ ТА ІНТЕГРАЦІЯ БАЗИ ДАНИХ З ВЕБДОДАТКОМ, РОБОТА З ДАНИМИ

4.1 Встановлення та інтеграція PostgreSQL з вебдодатком за допомогою Docker

Щоб забезпечити зручне, ізольоване та контрольоване середовище для розгортання бази даних у вебдодатку, було вирішено використовувати технологію контейнеризації Docker. Одна з головних переваг цього підходу полягає в тому, що базу даних та сам вебдодаток можна швидко розгорнути без інсталяції PostgreSQL та Java на основну операційну систему, що спрощує конфігурацію та покращує портативність застосунку[25].

Docker – це програмне забезпечення, яке дає можливість на певній ділянці пам'яті ізольовано встановити необхідну операційну систему, базу даних, налаштувати змінні середовища, встановити різні залежності тощо. При цьому дану програму абсолютно не буде хвилювати, що відбувається навколо.

Основні поняття Docker:

Docker file – це буквально текстовий файл де прописано інструкції для побудови Docker image.

Docker image – це такий пакет, в якому зберігаються всі необхідні файли, дані, конфігурації для запуску програмного забезпечення, на підставі якого в подальшому буде запущений контейнер.

Container – контейнер, який запускається на підставі image (образа). Створений контейнер можна запускати і зупиняти. Паралельно можна запустити кілька контейнерів, незалежних і залежних один від одного.

Docker-compose – це інструмент, який дозволяє запустити кілька контейнерів, пов'язаних між собою. Він може мати свої налаштування і може використовувати описані раніше налаштування з Dockerfile.

У даному випадку використовується файл docker-compose.yml, який описує два основні сервіси:

— postgres – контейнер з базою даних PostgreSQL;

— appliance-store-app – вебдодаток, створений на базі Spring Boot, який підключається до PostgreSQL.

```yaml
services:
  postgres:
    container_name: postgres-sql-appliance-store
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: appliance_store
    ports:
      - "5432:5432"

  appliance-store-app:
    container_name: appliance-store-app
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-sql-appliance-store:5432/appliance_store
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=postgres
    depends_on:
     - postgres
```

Рисунок 4.1 – Вміст файлу docker-compose.yml

services – це секція, яка описує усі сервіси, які потрібно підняти в рамках поточного середовища.

postgres – назва сервісу бази даних.

container_name – ім’я контейнера.

image – вказує образ, що буде використаний (у данному випадку офіційний образ PostgreSQL). Цей image автоматично завантажується з Docker Hub.

environment – змінні середовища, які налаштовують початкову конфігурацію PostgreSQL.

POSTGRES_USER – логін адміністратора.

POSTGRES_PASSWORD – пароль адміністратора.

POSTGRES_DB – назва створюваної бази даних.

ports – порти хоста у контейнері, що дозволяє підключатися до бази ззовні.

appliance-store-app – сервіс вебдодатку на Spring Boot.

container_name – ім’я контейнера.

build: . – вказує, що образ додатку збирається з Dockerfile у поточному каталозі(рис.4.2).

ports – порт 8080, на якому працює додаток.

environment – змінні середовища, які конфігурують доступ до бази даних.

SPRING_DATASOURCE_URL – URL підключення до БД.

SPRING_DATASOURCE_USERNAME – ім’я користувача для підключення.

SPRING_DATASOURCE_PASSWORD – пароль користувача.

depends_on – задає залежність, що дозволяє запустити додаток лише після підняття контейнера з базою.

```
FROM openjdk:17-alpine
COPY target/ApplianceStore-0.0.1-SNAPSHOT.jar appliance-store-app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "appliance-store-app.jar"]
```

Рисунок 4.2 – Dockerfile вебдодатку

Цей Dockerfile використовується для створення image вебдодатку у docker-compose.yml файлі.

FROM openjdk:17-alpine – базовий образ з Java 17 на Alpine Linux.

COPY – копіює зібраний JAR-файл у контейнер.

EXPOSE 8080 – відкриває порт 8080 для доступу до вебдодатку.

ENTRYPOINT – вказує команду для запуску програми в контейнері.

Апаратні вимоги та характеристики середовища запуску. Для успішного використання Docker необхідно дотримуватись певних мінімальних апаратних вимог, які забезпечують стабільну роботу контейнерів, особливо якщо одночасно запускається кілька сервісів (наприклад, база даних та вебдодаток).

Таблиця 4.1 – Мінімальні системні вимоги для Docker[27]

Компонент	Характеристики
Процесор	x86_64 з підтримкою віртуалізації (Intel VT-x або AMD-V)
Оперативна пам’ять	не менше 4 ГБ (рекомендовано від 8 ГБ для одночасної роботи кількох контейнерів)
Вільне місце на диску	мінімум 5–10 ГБ для зберігання образів і контейнерів
Операційна система	Linux, Windows 10/11 Pro або macOS (з підтримкою Hyper-V / WSL2 / Virtualization)

Docker був розгорнутий на локальному комп'ютері(табл. 4.2).

Таблиця 4.2 – Характеристики ПК на якому був розгорнутий Docker

Компонент	Характеристики
Операційна система	Windows 11 Pro, 64-bit
Процесор	Intel Core i7-12700K, 12 ядер, 20 потоків, частота 3.6GHz
Оперативна пам’ять	16 ГБ DDR4
Накопичувач	SSD 512 ГБ (NVMe)
Docker Engine версія	27.5.1
Docker Compose версія	2.32.4

Ці характеристики дозволили безперешкодно запустити одночасно два контейнери (PostgreSQL та Spring Boot застосунок), забезпечуючи стабільну роботу, швидкий час запуску та ефективне використання ресурсів.

Переваги використання Docker:

— ізольованість середовища – база даних працює в окремому контейнері, що не потребує встановлення PostgreSQL на систему;

— повторюваність конфігурації – середовище може бути легко відтворене на іншій машині або сервері;

— швидкість запуску – швидке розгортання додатку з БД у кілька команд;

— можливість масштабування – у майбутньому легко додати нові сервіси, наприклад Redis для кешування.

4.2 Створення структури бази даних

Одним із ключових етапів у розробці вебдодатку є формування чіткої, логічної та стійкої структури бази даних, яка б не лише забезпечувала зберігання всієї необхідної інформації, але й дозволяла ефективно взаємодіяти з нею під час виконання бізнес-операцій. У розділі 3 було розглянуто логічну та фізичну моделі бази даних, які стали основою для створення реальної структури сховища даних.

Під час реалізації структури бази даних у розділі 3 було розглянуто ручне створення таблиць з використанням мови SQL, а також у розділі 2, пункт 2.5 – автоматичне генерування таблиць із застосуванням технології ORM (Object-Relational Mapping), зокрема за допомогою Spring Data JPA, яка забезпечує зручну інтеграцію між об'єктною моделлю застосунку та структурою бази даних. Обидва ці методи мають свої переваги, а в межах практичного застосування часто поєднуються залежно від конкретних вимог проєкту.

Ручне створення структури бази даних є класичним підходом, який дозволяє розробнику мати повний контроль над архітектурою сховища даних. У цьому підході таблиці, зв’язки між ними, типи полів, індекси, обмеження та інші елементи створюються за допомогою SQL-скриптів, які потім виконуються на сервері бази даних. Це дозволяє досягти високої точності в побудові схеми, уникати автоматично згенерованих помилок та забезпечувати повну відповідність специфікаціям проєкту.

Ручне створення починається з розробки SQL-інструкцій для побудови таблиць. Спираючись на розділ 3, пункт 3.2, де було описано фізчину модель бази данних, було створено SQL файл schema.sql в ресурсах проєкту (src/main/resources) в який було внесено інструкції для створення бази даних (дод. А).

Для автоматичного заповнення бази даних початковими даними, було створено файл data.sql (дод. Б). Це робиться для того, щоб одразу після запуску додатку в базі вже були облікові записи (наприклад admin) та попередньо створені

товари, категорії тощо.

Для того, щоб Spring Boot знав, що потрібно використовувати власноруч створений SQL-скрипт, у конфігураційному файлі application.properties задаються наступні параметри:

— *spring.sql.init.encoding=UTF-8* – цей параметр визначає кодування, яке використовується для читання SQL-скриптів;

— *spring.sql.init.mode=always* – цей параметр вказує Spring Boot, що SQL ініціалізація (тобто виконання скриптів schema.sql або data.sql) повинна відбуватись завжди, незалежно від того, запущено застосунок на вбудованій або зовнішній базі даних. Це гарантує, що структура бази даних буде створена щоразу при запуску;

— *spring.sql.init.schema-locations=classpath:schema.sql ma spring.sql.init.schema locations=classpath:data.sql* – ці параметри явно вказують, де знаходиться SQL-скрипти зі схемою бази даних та з даними які потрібно ініціалізувати при запуску БД. У даному випадку Spring Boot шукає файли schema.sql та data.sql у ресурсах (src/main/resources) і виконує їх при старті застосунку. Це дозволяє створити всі таблиці та зв’язки в базі згідно з написаною вручну структурою та заповнити базу даних початковими даними;

— *spring.jpa.hibernate.ddl-auto=none* – цей параметр вимикає автоматичне створення або оновлення структури бази даних, яке виконується через Hibernate (тобто через ORM). Завдяки цьому Hibernate не намагатиметься втрутитись у структуру, яку ми самостійно створили через SQL-скрипт.

```
spring.sql.init.encoding=UTF-8
spring.sql.init.mode=always
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.data-locations=classpath:data.sql
spring.jpa.hibernate.ddl-auto=none
```

Рисунок 4.3 – Приклад конфігурації для ручного виконання schema.sql та data.sql

Генерування структури за допомогою ORM. Окрім ручного створення структури бази даних за допомогою SQL-скриптів, у вебдодатку також можливо автоматизовано генерувати схему бази даних за допомогою об’єктно-реляційного відображення (ORM), яке реалізується через фреймворк Spring Data JPA. Детальний опис принципів роботи цього модуля вже було подано в пункті 2.5 другого розділу, де розглядалися його можливості для організації доступу до даних у Java-додатках. Цей підхід дозволяє розробникам описувати структуру бази даних у вигляді класів Java, не відволікаючись на написання SQL-коду. Такі класи-сутності зберігають логіку зберігання даних і відображають її у вигляді таблиць автоматично.

Принцип створення структури бази даних за допомогою Spring Data JPA є однаковим для всіх сутностей, які використовуються у вебдодатку. У якості прикладу розглянемо роботу з сутністю Order, яка відображає замовлення клієнтів. Клас моделі Order, що представляє замовлення, буде описано нижче для демонстрації цього підходу:

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "employee_id")
    @NotNull
    private Employee employee;

    @ManyToOne
    @JoinColumn(name = "client_id")
    @NotNull
    private Client client;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinTable(
            name = "order_order_row",
            joinColumns = @JoinColumn(name = "order_id"),
            inverseJoinColumns = @JoinColumn(name = "order_row_id")
    )
    private Set<OrderRow> orderRowSet = new HashSet<>();

    private Boolean approved = false;
```

Рисунок 4.4 – Клас моделі Order

@Data – анотація з бібліотеки Lombok, яка автоматично генерує геттери, сеттери, методи equals(), hashCode() та toString(). Це значно скорочує обсяг шаблонного коду, що підвищує читабельність і зручність підтримки класу;

@AllArgsConstructor і @NoArgsConstructor – також анотації Lombok. Перша створює конструктор, який приймає всі поля класу як аргументи, друга – безаргументний конструктор;

@Entity – ця анотація вказує, що клас є JPA-сутністю і буде зіставлений із таблицею в базі даних;

@Table(name = "orders") – дозволяє явно вказати ім'я таблиці в базі даних (orders), до якої буде прив'язаний цей клас;

Для поля id проставлені такі анотації:

@Id – позначає це поле як первинний ключ таблиці;

@GeneratedValue(strategy = GenerationType.IDENTITY) – вказує, що значення первинного ключа буде генеруватися автоматично базою даних при вставці нового запису. Використання стратегії IDENTITY особливо поширене у поєднанні з PostgreSQL, яка має механізм автоінкременту.

Для поля employee проставлені такі анотації:

@ManyToOne – вказує на зв'язок "багато до одного", тобто багато замовлень можуть бути пов'язані з одним співробітником;

@JoinColumn(name = "employee_id") – задає ім'я стовпця у таблиці orders, який є зовнішнім ключем і посилається на таблицю employee;

@NotNull – означає, що замовлення не може існувати без прив'язки до співробітника.

Для поля client все аналогічно полю employee, це посилання на клієнта, який зробив замовлення. Тут також встановлено зв'язок "багато до одного" і обмеження обов'язкової присутності значення.

Для поля orderRowSet проставлені такі анотації:

@OneToMany – визначає зв'язок "один до багатьох", тобто одне замовлення може містити багато позицій товарів (OrderRow);

cascade = CascadeType.ALL – всі операції (збереження,

оновлення,видалення) автоматично розповсюджуються на пов'язані об'єкти;

@JoinTable – задає проміжну таблицю order_order_row, яка реалізує зв'язок між замовленнями і позиціями в замовленні через два стовпці: order_id та order_row_id.

Поле approved, яке визначає, чи було замовлення затверджено. За замовчуванням кожне нове замовлення вважається не затвердженим (false).

Щоб увімкнути автоматичне створення або оновлення схеми бази даних при запуску застосунку, достатньо налаштувати один єдиний параметр spring.jpa.hibernate.ddl-auto у файлі application.properties. Цей параметр має кілька можливих значень:

— *none.* Hibernate нічого не робить зі схемою (використовується при ручному управлінні структурою);

— *update*. Hibernate оновлює схему бази даних відповідно до моделей сутностей без втрати наявних даних;

— *create*. При кожному запуску застосунку структура бази даних створюється заново, попередні дані видаляються;

— *create-drop*. Схема створюється при запуску і видаляється після завершення роботи застосунку.

Зазвичай використовується значення update, оскільки воно дозволяє вносити зміни у модель без потреби вручну змінювати структуру бази. У підсумку, використання Spring Data JPA спрощує процес створення бази даних, автоматизуючи рутинні завдання, зменшуючи кількість помилок і пришвидшуючи розробку вебзастосунку.

У результаті створення структури бази даних було реалізовано комбінований підхід, що поєднує автоматичне створення таблиць за допомогою ORM та ручне наповнення початковими даними. Основу структури бази даних було сформовано за допомогою механізму об’єктно-реляційного відображення (ORM), реалізованого через анотації у класах моделі. Це дало змогу автоматично згенерувати всі необхідні таблиці, встановити зв’язки між ними та забезпечити узгодженість даних. Для ініціалізації початкових даних використовувався файл

data.sql (дод. Б), в якому вручну задано SQL-запити для вставки записів у відповідні таблиці. У цьому файлі були вказані основні дані, необхідні для коректної роботи вебзастосунку, зокрема початковий перелік товарів та виробників, адміністраторські облікові записи. Такий підхід забезпечує готовність системи до використання відразу після розгортання. Застосування комбінованого підходу є доцільним, оскільки він поєднує переваги обох методів. Автоматизація створення структури БД завдяки ORM спрощує розробку та зменшує кількість помилок. Ручне наповнення через data.sql дає повний контроль над початковими даними та забезпечує гнучкість у процесі тестування. Конфігурація для комбінованого підходу матиме такий вигляд (рис. 4.5)

```
spring.sql.init.encoding=UTF-8
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
spring.sql.init.data-locations=classpath:data.sql.
spring.jpa.hibernate.ddl-auto=update
```

Рисунок 4.5 – Конфігурація для комбінованого підходу створення структури БД

В конфігурації було додано параметр spring.jpa.defer-datasource-initialization=true (рис. 4.5). Використання цього параметра гарантує, що, спочатку Spring Boot ініціалізує схему бази даних за допомогою JPA, а потім, після ініціалізації схеми, буде виконано завантаження SQL-даних з файлу, вказаного в параметрі spring.sql.init.data-locations. Таким чином, цей порядок допомагає уникнути ситуацій, коли SQL-дані з data.sql намагаються вставити значення в таблиці, які ще не були створені. Як результат було створено 7 таблиць (рис. 4.6).

```
appliance_store=# \dt
              List of relations
 Schema |      Name       | Type  |  Owner
--------+-----------------+-------+----------
 public | appliances      | table | postgres
 public | clients         | table | postgres
 public | employees       | table | postgres
 public | manufacturers   | table | postgres
 public | order_order_row | table | postgres
 public | order_row       | table | postgres
 public | orders          | table | postgres
(7 rows)
```

Рисунок 4.6 – Ствоернні таблиці в БД

А також таблиці: employees, appliances, manufacturers було заповнено початковими даними (дод. В).

Окремо, для оптимізації пошуку по полю category у таблиці appliances, був створений індекс idx_appliances_category (рис. 3.11). Його створення було виконане вручну безпосередньо через консоль керування базою даних PostgreSQL. Цей індекс дозволяє оптимізувати пошук незалежно від регістру символів. Для перевірки створення індекса було виконано запит “SELECT indexname, indexdef FROM pg_indexes WHERE tablename = 'appliances';” (рис. 4.7).

```
appliance_store=# SELECT indexname, indexdef FROM pg_indexes WHERE tablename = 'appliances';
        indexname        |                                        indexdef
-------------------------+-----------------------------------------------------------------------------------
 appliances_pkey         | CREATE UNIQUE INDEX appliances_pkey ON public.appliances USING btree (id)
 idx_appliances_category | CREATE INDEX idx_appliances_category ON public.appliances USING btree (category)
(2 rows)
```

Рисунок 4.7 – Результат запиту перевірки створення індекса

Індекс idx_appliances_category був створений і знаходиться в спеціальній таблиці індексів pg_indexes (рис. 4.7).

Перевірка роботи індекса через команду “EXPLAIN SELECT * FROM appliances WHERE category = 'BIG';”.

```
appliance_store=# EXPLAIN SELECT * FROM appliances WHERE category = 'BIG';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx_appliances_category on appliances  (cost=0.13..12.20 rows=4 width=200)
   Index Cond: ((category)::text = 'BIG'::text)
(2 rows)
```

Рисунок 4.8 – Результат запиту перевірки роботи індекса

PostgreSQL використовує індекс для пошуку за категорією, на це вкзує “Index Scan” (рис. 4.8).

cost=0.13..12.20 – це оцінка вартості виконання запиту. 0.13 – це вартість початкової частини запиту. 12.20 – це вартість виконання операцій після того, як знайдені відповідні рядки. Чим менша вартість, тим швидше запит.

rows=4 – це оцінка кількості рядків, які буде повернуто в результаті запиту.

width=200 – це оцінка середнього розміру рядка, який буде повернуто в результаті запиту (у байтах).

Index Cond: ((category)::text = 'BIG'::text) – це умова індексу, що використовується для пошуку: PostgreSQL шукає рядки, де значення в колонці category дорівнює 'BIG'.

Створення резервної копії бази даних. Для забезпечення збереження інформації в разі збоїв або втрати даних, було здійснено резервне копіювання бази даних PostgreSQL, що працює у Docker-контейнері. Резервну копію виконано за допомогою стандартної утиліти pg_dump, яка запускається всередині контейнера, а результат зберігається у файл на хост-машині. Команда для створення копії має такий вигляд: “docker exec -i ffa2276467cd pg_dump -U postgres -d appliance_store -F c > appliance_store_backup.dump”.

docker exec -t ffa2276467cd – виконує команду всередині контейнера з ідентифікатором ffa2276467cd;

pg_dump -U postgres -d appliance_store – створює дамп бази даних appliance_store від імені користувача postgres;

-F c – вказує формат дампу (custom), що зручно для подальшого відновлення через pg_restore;

> appliance_store_backup.dump – перенаправляє вихідний дамп у файл на локальній машині, а не в контейнері.

У результаті файл appliance_store_backup.dump буде створено в тій директорії, де виконувалась команда (рис. 4.9).

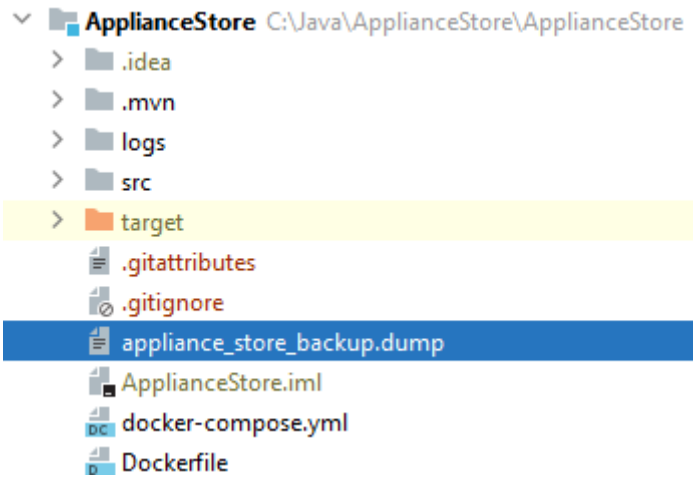


Рисунок 4.9 – Файл резервної копії БД appliance_store_backup.dump

Для відновлення бази даних з резервної копії, створеної за допомогою pg_dump, використовується утиліта pg_restore. Коли база даних працює всередині Docker-контейнера, процес відновлення передбачає кілька послідовних дій:

— створення нової бази даних у PostgreSQL, яка прийматиме відновлені дані. Це можна зробити за допомогою команди “docker exec -it <docker_container_id>cd createdb -U postgres appliance_store_restored”;

— передача файлу резервної копії з хост-машини до Docker-контейнера. Це робиться за допомогою команди “docker cp appliance_store_backup.dump <docker_container_id>cd:/appliance_store_backup.dump”. Ця команда скопіює файл appliance_store_backup.dump всередину контейнера;

— виконати відновлення бази даних за допомогою команди “docker exec -it <docker_container_id> pg_restore -U postgres -d appliance_store_restored /appliance_store_backup.dump”. У цій команді pg_restore читає дамп-файл і відновлює всі структури та дані в базу appliance_store_restored. Після успішного завершення цього процесу база даних буде повністю відновлена в новій копії.

Цей підхід дозволяє не лише захистити інформацію від втрати, а й забезпечити можливість швидкого відновлення роботи вебзастосунку без залучення складних інструментів або ручного відновлення структури та даних.

Таким чином, на основі реалізованих підходів сформована повноцінна, функціональна та масштабована структура бази даних, яка повністю відповідає вимогам до вебдодатку “Інтернет-магазин побутової техніки”.

4.3 Використання Spring Data JPA для взаємодії з БД

У даному підрозділі розглянуто практичне використання технології Spring Data JPA для реалізації доступу до бази даних у вебдодатку "Магазин побутової техніки". Spring Data JPA забезпечує високий рівень абстракції, що дозволяє значно зменшити обсяг шаблонного коду і спростити реалізацію CRUD-операцій та складних запитів. Основні можливості Spring Data JPA було описано в другому розділі.

У якості прикладу взаємодії з базою було розглянуто роботу з сутністю Order, яка відображає замовлення клієнтів. Для взаємодії з таблицею бази даних використовується інтерфейс OrdersRepository, який розширює стандартний інтерфейс JpaRepository<Orders, Long> (рис. 4.10). Це дозволяє автоматично використовувати велику кількість базових методів, таких як save(), findById(), findAll(), deleteById() та інші, без необхідності писати SQL-запити вручну. Наприклад, метод findAll(Pageable pageable) реалізує пагіновану вибірку всіх замовлень, а findAllByClientId(Long clientId, Pageable pageable) – вибірку замовлень лише для конкретного клієнта. Запит findAllByClientId(Long clientId, Pageable pageable) написано за допомогою JPQL (рис. 4.10).

Варто зазначити, що всі репозиторії мають схожий вигляд, оскільки всі вони наслідують JpaRepository<..., Long> і, відповідно, мають однаковий набір базових методів. Це дозволяє дотримуватись єдиного стилю та спрощує підтримку коду.

```java
public interface OrdersRepository extends JpaRepository<Orders, Long> {
    new *
    Page<Orders> findAll(Pageable pageable);

    2 usages  new *
    @Query("SELECT o FROM Orders o WHERE o.client.id = :clientId")
    Page<Orders> findAllByClientId(@Param("clientId") Long clientId, Pageable pageable);
}
```

Рисунок 4.10 – Приклад репозиторія OrderRepository

У той час як у репозиторіях реалізуються лише методи безпосередньої взаємодії з базою даних, сервісні класи (Service) виконують роль проміжної логіки. У них можуть міститися додаткові перевірки, обробка винятків, трансформація даних, логування тощо.

```java
@Service
@AllArgsConstructor
@Transactional
public class OrderServiceImpl implements OrderService {

    private final OrdersRepository ordersRepository;
    private final OrderMapper orderMapper;
    private final ApplianceService applianceService;
    private final ApplianceMapper applianceMapper;

    new *
    @Loggable
    @Override
    public OrderDto add(Order order) {
        if (order != null) {
            return orderMapper.mapToDto(ordersRepository.save(order));
        }
        throw new NullEntityReferenceException("Order cannot be 'null'");
    }

    new *
    @Loggable
    @Override
    @Transactional(readOnly = true)
    public OrderDto readById(Long id) {
        return orderMapper.mapToDto(ordersRepository.findById(id).orElseThrow(
                () -> new EntityNotFoundException("Order with id: " + id + " not found")));
    }

    new *
    @Loggable
    @Override
    public OrderDto update(Order order) {
        if (order != null) {
            readById(order.getId());
            return orderMapper.mapToDto(ordersRepository.save(order));
        }
        throw new NullEntityReferenceException("Order cannot be 'null'");
    }
```

Рисунок 4.11 – Приклад сервіс класу Order

Однією з важливих особливостей сервісного рівня є використання транзакційності, яка забезпечується за допомогою анотації @Transactional. Транзакції в Spring дозволяють гарантувати цілісність даних – тобто, або всі дії в межах транзакції успішно виконуються, або жодна з них не вступає в силу у разі помилки. Наприклад, метод update() оновлює дані про товар, тому повинен виконуватись як єдина атомарна операція. Якщо в процесі щось піде не так, наприклад, не буде знайдено товару, тоді всі зміни скасовуються автоматично.

У класі OrderServiceImpl транзакційність забезпечується на рівні всього класу анотацією @Transactional, це означає, що всі публічні методи будуть виконуватись у контексті транзакції. Для операцій читання (наприклад, readById() або getAll()), транзакція позначається лише для читання (@Transactional(readOnly = true)), що дозволяє оптимізувати продуктивність.

Завдяки цьому підходу з використанням Spring Data JPA, система обробки даних у вебдодатку є масштабованою, зручною у підтримці та має високий рівень надійності при роботі з базою даних.

ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досліджено роль реляційних баз даних у побудові вебдодатків, а також здійснено розробку повноцінної бази даних для системи електронної комерції – інтернет-магазину побутової техніки. Проаналізовано різні типи систем керування базами даних, їх переваги та недоліки, а також обґрунтовано вибір PostgreSQL як оптимальної СУБД для проєкту.

У процесі роботи детально вивчено архітектуру Spring Framework з особливою увагою до модуля Spring Data JPA, що дозволив спростити доступ до даних, реалізувати CRUD-операції, зв’язки між сутностями, пагінацію, транзакційність та автоматичну генерацію SQL-запитів. Такий підхід забезпечив ефективну інтеграцію логіки застосунку з базою даних без надмірного ручного кодування.

Практична частина роботи передбачала створення структури реляційної бази даних, включно з ключовими сутностями: користувачі, товари, замовлення, виробники, категорії тощо. Реалізовано логічні зв’язки між таблицями (OneToOne, ManyToOne, ManyToMany), застосовано індексацію для оптимізації запитів, а також впроваджено базовий механізм резервного копіювання для підвищення надійності системи.

Впровадження даної бази даних показало, що реляційна модель, при правильному проєктуванні, здатна забезпечити високу цілісність, узгодженість і масштабованість системи. Робота також підтвердила ефективність використання Spring Data JPA як потужного інструмента для інтеграції реляційної СУБД із вебдодатками, створеними на Java.

У перспективі розвиток застосунку може передбачати впровадження складніших запитів, використання кешування та оптимізацію схем даних. Подальше вдосконалення системи управління даними дозволить забезпечити ще вищу продуктивність, безпеку та зручність користування інтернет-магазином.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. What Is NoSQL? NoSQL Databases Explained. *MongoDB*. URL: https://www.mongodb.com/resources/basics/databases/nosql-explained (дата звернення: 05.03.2025).
2. Technology 2024 Stack Overflow Developer Survey. *Stack Overflow*. URL: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database (дата звернення: 07.03.2025).
3. ProgIngContrSystems. *ProgIngContrSystems*. URL: https://pupenasan.github.io/ProgIngContrSystems/Лекц/11_db.html (дата звернення: 10.03.2025).
4. Реляційні бази даних: структура та застосування у практиці. *FoxmindEd*. URL: https://foxminded.ua/reliatsiini-bazy-danykh/ (дата звернення: 02.04.2025).
5. W3Schools.com. *W3Schools Online Web Tutorials*. URL: https://www.w3schools.com/sql/ (дата звернення: 11.03.2025).
6. ostezer, Drake M. SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems. *DigitalOcean | Cloud Infrastructure for Developers*. URL: https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems (дата звернення: 14.03.2025).
7. Sqlite vs. MySQL vs. PostgreSQL: A Comparison of Relational Databases | Logz.io. *Logz.io*. URL: https://logz.io/blog/relational-database-comparison/ (дата звернення: 15.03.2025).
8. Relational Database Management System: DB Options Compared - Raima. *RaimaDB - Embedded database for mission-critical applications*. URL: https://raima.com/relational-database-management-system-db-options-compared/ (дата звернення: 20.03.2025).

9. Team A. E. Database Management Systems (DBMS) Comparison: MySQL, Postgr. *AltexSoft*. URL: https://www.altexsoft.com/blog/comparing-database-management-systems-mysql-postgresql-mssql-server-mongodb-elasticsearch-and-others/ (дата звернення: 23.03.2025).

10. What is NoSQL? Databases Explained | Google Cloud. *Google Cloud*. URL: https://cloud.google.com/discover/what-is-nosql (дата звернення: 25.03.2025).

11. Зв'язок у базі даних: основні типи та як встановлюється. *FoxmindEd*. URL: https://foxminded.ua/zviazok-u-bazi-danykh/ (дата звернення: 26.03.2025).

12. Система управління базами даних: сучасні тенденції у розробці. *FoxmindEd*. URL: https://foxminded.ua/systema-upravlinnia-bazamy-danykh/ (дата звернення: 27.03.2025).

13. Dudka M. Spring JDBC проти Spring Data JPA. *Dou*. URL: https://dou.ua/forums/topic/46082/ (дата звернення: 01.04.2025).

14. Paraschiv E. Spring JDBC. *Baeldung*. URL: https://www.baeldung.com/spring-jdbc-jdbctemplate (дата звернення: 02.04.2025).

15. Crusoveanu L. Intro to Inversion of Control and Dependency Injection with Spring. *Baeldung*. URL: https://www.baeldung.com/inversion-control-and-dependency-injection-in-spring (дата звернення: 04.04.2025).

16. Spring і Spring Boot: Основні відмінності та особливості. *FoxmindEd*. URL: https://foxminded.ua/vidminnosti-mizh-spring-i-spring-boot/ (дата звернення: 06.04.2025).

17. Spring boot: структура проекту та розробка веб-додатків. *FoxmindEd*. URL: https://foxminded.ua/spring-boot/ (дата звернення: 06.04.2025).

18. Terek H. Introducing Spring Boot. *dzone.com*. URL: https://dzone.com/articles/introducing-spring-boot (дата звернення: 07.04.2025).

19. Web MVC framework. *Spring | Home*. URL: https://docs.spring.io/spring-framework/docs/3.2.x/spring-framework-reference/html/mvc.html#:~:text=A%20key%20design%20principle%20in,to%20supply%20your%20own%20behavior. (дата звернення: 08.04.2025)

20. Paraschiv E. Spring MVC Tutorial. *Baeldung*. URL: https://www.baeldung.com/spring-mvc-tutorial (дата звернення: 09.04.2025).

21. GeeksforGeeks. Learn Spring MVC Tutorial - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/spring-mvc/ (дата звернення: 09.04.2025).

22. GeeksforGeeks. Spring - MVC Framework - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/spring-mvc-framework/ (дата звернення: 09.04.2025).

23. GeeksforGeeks. Spring Security Tutorial - GeeksforGeeks. *GeeksforGeeks*. URL: https://www.geeksforgeeks.org/spring-security-tutorial/ (дата звернення: 10.04.2025).

24. Самбор І. В., Бученко І. А. Визначення вимог до бази даних для вебдодатку «Магазин побутової техніки» на основі SPRING. Застосування програмного забезпечення в ІКТ : VI Всеукр. науково-техн. конф., м. Київ, 24 квіт. 2025 р. Київ, 2025. С. 454–457. URL: https://duikt.edu.ua/uploads/p_2779_56719466.pdf

25. Самбор І. В., Бученко І. А. Розробка вебдодатку «магазин побутової техніки» на основі SPRING. Застосування програмного забезпечення в ІКТ : VI Всеукр. науково-техн. конф., м. Київ, 24 квіт. 2025 р. Київ, 2025. С. 22–23. URL: https://duikt.edu.ua/uploads/p_2779_56719466.pdf

26. dbdiagram.io - Database Relationship Diagrams Design Tool. *dbdiagram.io - Database Relationship Diagrams Design Tool*. URL: https://dbdiagram.io/d/67fba4704f7afba1846716f7 (дата звернення: 20.04.2025).

27. Install Docker Desktop. *Docker Documentation*. URL:https://docs.docker.com/desktop/setup/install/windows-install/(дата звернення: 25.04.2025)

Скрипт schema.sql для ручного створення таблиць

```sql
CREATE TABLE appliances (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    category category NOT NULL,
    model VARCHAR(100) NOT NULL,
    manufacturer_id BIGINT NOT NULL REFERENCES manufacturers(id),
    power_type power_type NOT NULL,
    characteristic TEXT NOT NULL,
    description TEXT,
    power INTEGER NOT NULL CHECK (power > 0),
    price NUMERIC(10, 2) NOT NULL CHECK (price >= 0)
);

CREATE TYPE category AS ENUM ('BIG', 'SMALL');

CREATE TYPE power_type AS ENUM ('AC220', 'AC110', 'ACCUMULATOR');

CREATE TABLE clients (
    id BIGSERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    card VARCHAR(50) NOT NULL UNIQUE
);

CREATE TABLE employees (
    id BIGSERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    email TEXT NOT NULL UNIQUE,
    password TEXT NOT NULL,
    department VARCHAR(255) NOT NULL
);

CREATE TABLE manufacturers (
    id BIGSERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE order (
    id BIGSERIAL PRIMARY KEY,
    employee_id BIGINT NOT NULL REFERENCES employees(id),
    client_id BIGINT NOT NULL REFERENCES clients(id),
    approved BOOLEAN DEFAULT FALSE
);

CREATE TABLE order_rows (
    id BIGSERIAL PRIMARY KEY,
    appliance_id BIGINT NOT NULL REFERENCES appliances(id),
    number BIGINT NOT NULL CHECK (number > 0),
    amount NUMERIC(10, 2) NOT NULL CHECK (amount >= 0)
);

CREATE TABLE order_order_row (
    order_id BIGINT NOT NULL REFERENCES order(id),
    order_row_id BIGINT NOT NULL REFERENCES order_rows(id),
    PRIMARY KEY (order_id, order_row_id)
);
```

Скрипт data.sql для заповнення таблиць початковими даними

```sql
-- Manufacturers
INSERT INTO manufacturers (name) VALUES ('Samsung');
INSERT INTO manufacturers (name) VALUES ('Dell');
INSERT INTO manufacturers (name) VALUES ('HP');
INSERT INTO manufacturers (name) VALUES ('Apple');
INSERT INTO manufacturers (name) VALUES ('Lenovo');
INSERT INTO manufacturers (name) VALUES ('Acer');
INSERT INTO manufacturers (name) VALUES ('AMD');

-- Employees
INSERT INTO employees (name, email, password, department)
VALUES ('Vanya', 'sambor@gmail.com', '$2a$12$d997OeyHKsgJMnf3PLXiJuPRBvSjzR5qcE6P8vvalaGYD53Sx2oJm', 'security');

INSERT INTO employees (name, email, password, department)
VALUES ('Denys', 'dan23@gmail.com', '$2a$12$psaL2af.ZjJTVYpR7BygWODJ1uSPDpXBAjsYMXzQkDnwBDi71qbZS', 'security');

-- Appliances
INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Claw', 'BIG', 'CLW-900X', 1, 'ACCUMULATOR', 'Літій-іонний акумулятор, 12 годин роботи', 'Потужний пилосос для великого будинку', 600, 129.99);

INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Bane', 'SMALL', 'BN-S220', 3, 'AC110', 'Міні-компресор для надувних виробів', 'Компактний та потужний прилад для дому та подорожей', 2200, 79.50);

INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Ecu', 'SMALL', 'EC-800', 2, 'ACCUMULATOR', 'Акумулятор 4000 мА·г, зарядка USB-C', 'Ручний міксер із трьома насадками', 800, 45.00);

INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Kang Dae', 'BIG', 'KD-3600', 4, 'AC220', 'Інверторний двигун, режим енергозбереження', 'Пральна машина з великим барабаном', 3600, 499.99);

INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Gust', 'BIG', 'GST-PRO', 5, 'ACCUMULATOR', 'Автономність до 5 год, 3 швидкості обертання', 'Газонокосарка з регулюванням висоти скошування', 650, 259.00);

INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Ancile', 'SMALL', 'ANC-230', 6, 'AC220', 'Потужність 1000 Вт, 2 температурні режими', 'Фен для волосся з функцією іонізації', 230, 39.99);

INSERT INTO appliances (name, category, model, manufacturer_id, power_type, characteristic, description, power, price)
VALUES ('Halo', 'BIG', 'HL-300', 7, 'ACCUMULATOR', 'Інтелектуальне очищення повітря, сенсори CO2', 'Побутовий очищувач повітря для великих кімнат', 300, 179.49);
```

Заповнені таблиці employees, appliances, manufacturers

```
appliance_store=# select * from employees;
 id |      email      | name |                          password                          | department
----+-----------------+------+------------------------------------------------------------+------------
  1 | sambor@gmail.com | Vanya | $2a$12$psaL2af.ZjTVYpR7BygWOOJ1uSPDpXBAjsYMXzQkOnwBDi71qbZS | security
  2 | dan23@gmail.com  | Denys | $2a$12$psaL2af.ZjTVYpR7BygWOOJ1uSPDpXBAjsYMXzQkOnwBDi71qbZS | security
(2 rows)

appliance_store=# select * from appliances;
 id | category |          characteristic          |                        description                        | model  |   name   | power | power_type  | price  | manufacturer_id
----+----------+----------------------------------+-----------------------------------------------------------+--------+----------+-------+-------------+--------+-----------------
  1 | BIG      | Літій-іонний акумулятор, 12 годин роботи | Потужний пилосос для великого будинку                     | CLW-900X | Claw     |   600 | ACCUMULATOR | 129.99 |               1
  2 | SMALL    | Міні-компресор для надувних виробів | Компактний та потужний прилад для дому та подорожей      | BN-S220 | Bane     |  2200 | AC110       |  79.50 |               3
  3 | SMALL    | Акумулятор 4000 мА·г, зарядка USB-C | Ручний міксер із трьома насадками                         | EC-800 | Ecu      |   800 | ACCUMULATOR |  45.00 |               2
  4 | BIG      | Інверторний двигун, режим енергозбереження | Пральна машина з великим барабаном                        | KD-3600 | Kang Dae |  3600 | AC220       | 499.99 |               4
  5 | BIG      | Автономність до 5 год, 3 швидкості обертання | Газонокосарка з регулюванням висоти скошування            | GST-PRO | Gust     |   650 | ACCUMULATOR | 259.00 |               5
  6 | SMALL    | Потужність 1000 Вт, 2 температурні режими | Фен для волосся з функцією іонізації                      | ANC-230 | Ancile   |   230 | AC220       |  39.99 |               6
  7 | BIG      | Інтелектуальне очищення повітря, сенсори CO2 | Побутовий очищувач повітря для великих кімнат             | HL-300 | Halo     |   300 | ACCUMULATOR | 179.49 |               7

appliance_store=# select * from manufacturers;
 id |  name
----+---------
  1 | Samsung
  2 | Dell
  3 | HP
  4 | Apple
  5 | Lenovo
  6 | Acer
  7 | AMD
(7 rows)
```

Додаток Г

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

(Презентація)



Державний університет інформаційно-комунікаційних технологій
Інститут інформаційних технологій
Кафедра комп’ютерної інженерії

Бакалаврська робота

на тему: Реляційна база даних у вебдодатку "Магазин побутової техніки" на основі Spring

Виконав студент групи КІД-41
Самбор Іван Валерійович

Керівник дипломної роботи:
Бученко Ігор Анатолійовчи,
старший викладач кафедри КІ

Київ – 2025

- Об`єкт дослідження – реляційна база даних у вебдодатоку на основі Spring.
- Предмет дослідження – реляційна база даних, що використовується для зберігання та обробки даних у вебдодатку.
- Мета роботи – дослідити та впровадити реляційну базу даних у вебдодаток інтернет-магазин побутової техніки.

Актуальність обраної теми

У сучасному світі інформаційні технології відіграють ключову роль у розвитку бізнесу, а ефективне управління даними є основою успішного функціонування будь-якої компанії. В умовах стрімкого зростання онлайн торгівлі бази даних відіграють важливу роль у забезпеченні зберігання, обробки та швидкого доступу до даних.

Реляційні бази даних є найбільш поширеним способом організації та структуризації даних у вебдодатках.

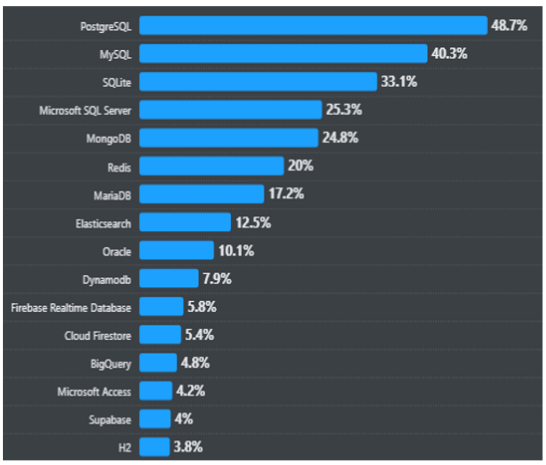
Реляційна база даних



ID	NAME	EMAIL	AGE	ADDRESS	PHONE
1	Денис	denys23@gmail.com	21	Київ, вул. Малинська 5	380501234567
2	Олександр	oleks56@gmail.com	32	Київ, вул. Садова 3	380675234589
3	Юлія	julia@gmail.com	27	Київ, вул. Васильківська 1	380631112233

Вигляд таблиці в реляційній базі даних

Порівняння та вибір РСКБД

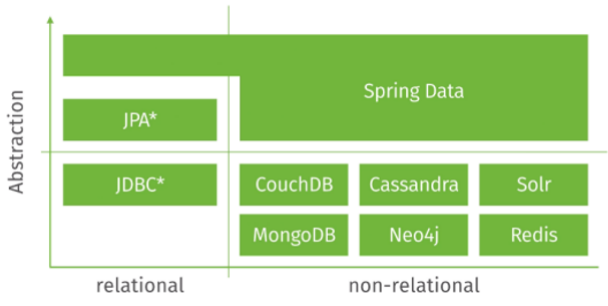


Результат опитування розробників на Stackoverflow за 2024 рік

На початковому етапі я порівняв популярні системи керування базами даних: SQLite, MySQL та PostgreSQL.

- SQLite — проста, але не підходить для багатокористувацьких додатків.
- MySQL — має високу продуктивність, але певні функціональні обмеження (наприклад, не підтримує FULL JOIN).
- PostgreSQL — обрана мною система

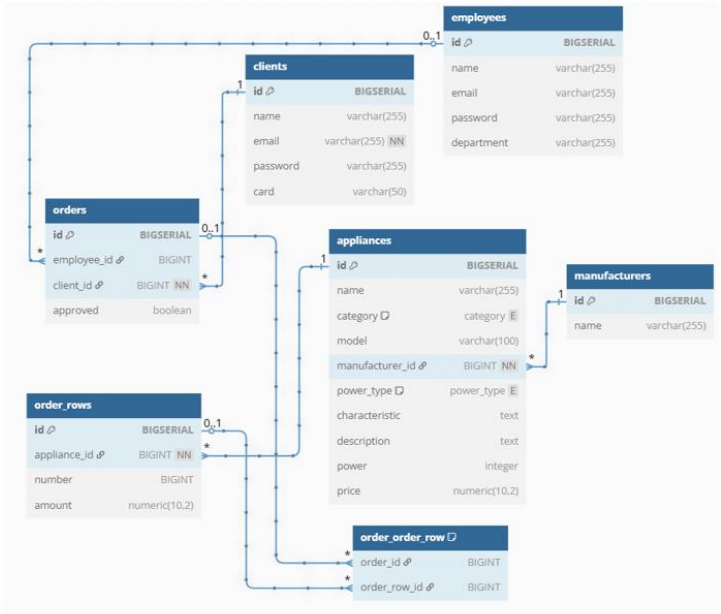
Робота з базою даних за допомогою Spring



Підмодулі Spring Data

- Spring Data JPA – Spring Data JPA є потужною абстракцією над JPA.
- JPA (Java Persistence API) – це інтерфейс для роботи з реляційними базами даних у Java.
- ORM (Object Relational Mapping) – це технологія, яка забезпечує автоматичне перетворення даних між світом програмування та реляційними базами даних.

ER-Diagram для вебдодатку **Магазин побутової техніки**



Встановлення та інтеграція PostgreSQL з вебдодатком

```yaml
services:
  postgres:
    container_name: postgres-sql-appliance-store
    image: postgres
    environment:
      POSTGRES_USER: postgres
      POSTGRES_PASSWORD: postgres
      POSTGRES_DB: appliance_store
    ports:
      - "5432:5432"

  appliance-store-app:
    container_name: appliance-store-app
    build: .
    ports:
      - "8080:8080"
    environment:
      - SPRING_DATASOURCE_URL=jdbc:postgresql://postgres-sql-appliance-store:5432/appliance_store
      - SPRING_DATASOURCE_USERNAME=postgres
      - SPRING_DATASOURCE_PASSWORD=postgres
    depends_on:
     - postgres
```

docker-compose

```
FROM openjdk:17-alpine
COPY target/ApplianceStore-0.0.1-SNAPSHOT.jar appliance-store-app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "appliance-store-app.jar"]
```

Dockerfile вебдодатку

Апаратні вимоги та характеристики середовища запуску

Таблиця 4.1 – Мінімальні системні вимоги для Docker[27]

Компонент	Характеристики
Процесор	x86_64 з підтримкою віртуалізації (Intel VT-x або AMD-V)
Оперативна пам'ять	не менше 4 ГБ (рекомендовано від 8 ГБ для одночасної роботи кількох контейнерів)
Вільне місце на диску	мінімум 5–10 ГБ для зберігання образів і контейнерів
Операційна система	Linux, Windows 10/11 Pro або macOS (з підтримкою Hyper-V / WSL2 / Virtualization)

Таблиця 4.2 – Характеристики ПК на якому був розгорнутий Docker

Компонент	Характеристики
Операційна система	Windows 11 Pro, 64-bit
Процесор	Intel Core i7-12700K, 12 ядер, 20 потоків, частота 3.6GHz
Оперативна пам'ять	16 ГБ DDR4
Накопичувач	SSD 512 ГБ (NVMe)
Docker Engine версія	27.5.1
Docker Compose версія	2.32.4

Створення структури БД

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne
    @JoinColumn(name = "employee_id")
    @NotNull
    private Employee employee;

    @ManyToOne
    @JoinColumn(name = "client_id")
    @NotNull
    private Client client;

    @OneToMany(cascade = CascadeType.ALL)
    @JoinTable(
            name = "order_order_row",
            joinColumns = @JoinColumn(name = "order_id"),
            inverseJoinColumns = @JoinColumn(name = "order_row_id")
    )
    private Set<OrderRow> orderRowSet = new HashSet<>();

    private Boolean approved = false;
```

Приклад автоматичного створення за допомогою ORM

```
spring.sql.init.encoding=UTF-8
spring.sql.init.mode=always
spring.sql.init.schema-locations=classpath:schema.sql
spring.sql.init.data-locations=classpath:data.sql
spring.jpa.hibernate.ddl-auto=none
```

Приклад конфігурації для ручного виконання schema.sql та data.sql

```
spring.sql.init.encoding=UTF-8
spring.sql.init.mode=always
spring.jpa.defer-datasource-initialization=true
spring.sql.init.data-locations=classpath:data.sql.
spring.jpa.hibernate.ddl-auto=update
```

Приклад конфігурації для комбінованого підходу

Оптимізація бази даних

Для оптимізації продуктивності системи, особливо в сценаріях частого пошуку або фільтрації за моделлю побутової техніки, було вирішено створити кастомний індекс по полю category у таблиці appliances.

```
CREATE INDEX idx_appliances_category ON appliances(category);
```

Скрипт для створення індексу по полю category

```
appliance_store=# SELECT indexname, indexdef FROM pg_indexes WHERE tablename = 'appliances';
        indexname        |                                      indexdef
-------------------------+-----------------------------------------------------------------------------------
 appliances_pkey         | CREATE UNIQUE INDEX appliances_pkey ON public.appliances USING btree (id)
 idx_appliances_category | CREATE INDEX idx_appliances_category ON public.appliances USING btree (category)
(2 rows)
```

Результат запиту перевірки створення індекса

```
appliance_store=# EXPLAIN SELECT * FROM appliances WHERE category = 'BIG';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------
 Index Scan using idx_appliances_category on appliances  (cost=0.13..12.20 rows=4 width=200)
   Index Cond: ((category)::text = 'BIG'::text)
(2 rows)
```

Результат запиту перевірки роботи індекса

Використання Spring Data JPA для взаємодії з БД

```java
@Service
@AllArgsConstructor
@Transactional
public class OrderServiceImpl implements OrderService {

    private final OrdersRepository ordersRepository;
    private final OrderMapper orderMapper;
    private final ApplianceService applianceService;
    private final ApplianceMapper applianceMapper;

    new *
    @Loggable
    @Override
    public OrderDto add(Order order) {
        if (order != null) {
            return orderMapper.mapToDto(ordersRepository.save(order));
        }
        throw new NullEntityReferenceException("Order cannot be 'null'");
    }

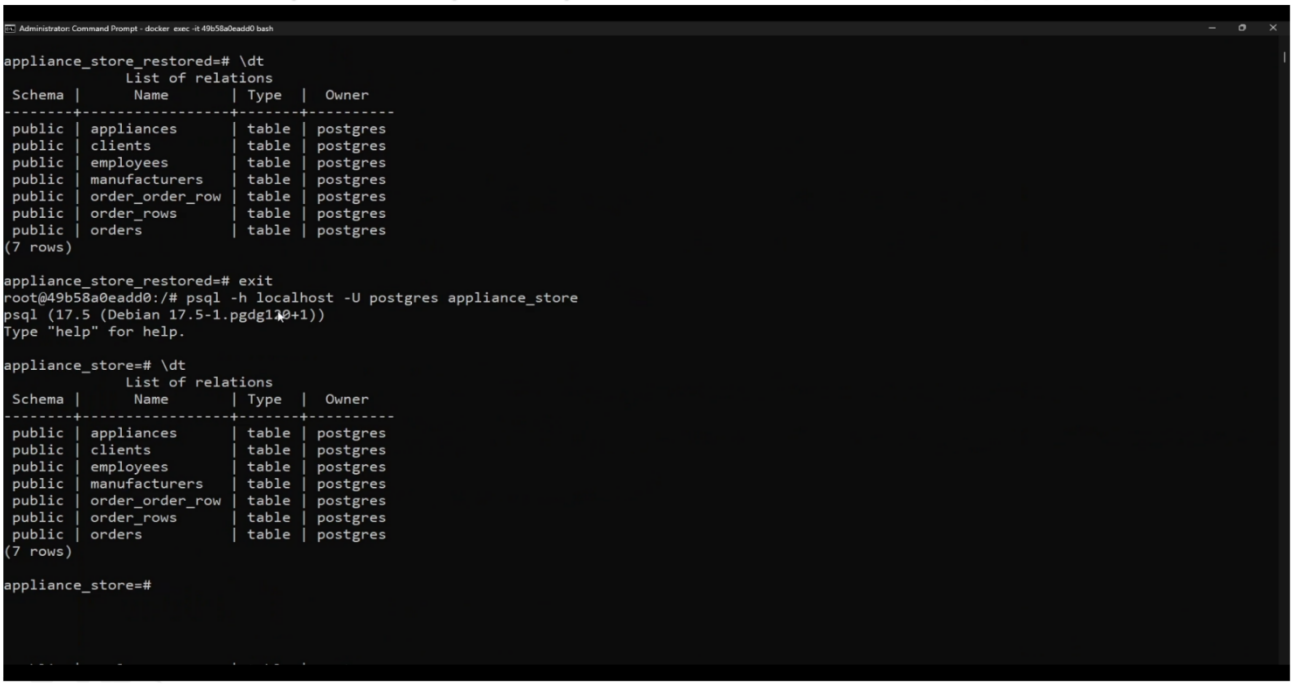
    new *
    @Loggable
    @Override
    @Transactional(readOnly = true)
    public OrderDto readById(Long id) {
        return orderMapper.mapToDto(ordersRepository.findById(id).orElseThrow(
                () -> new EntityNotFoundException("Order with id: " + id + " not found")));
    }

    new *
    @Loggable
    @Override
    public OrderDto update(Order order) {
        if (order != null) {
            readById(order.getId());
            return orderMapper.mapToDto(ordersRepository.save(order));
        }
        throw new NullEntityReferenceException("Order cannot be 'null'");
    }
```

```java
public interface OrdersRepository extends JpaRepository<Orders, Long> {
    new *
    Page<Orders> findAll(Pageable pageable);

    2 usages  new *
    @Query("SELECT o FROM Orders o WHERE o.client.id = :clientId")
    Page<Orders> findAllByClientId(@Param("clientId") Long clientId, Pageable pageable);
}
```

Створення резервної копії бази даних



ВИСНОВКИ

У ході виконання кваліфікаційної роботи було досліджено роль реляційних баз даних у побудові вебдодатків, а також здійснено розробку повноцінної бази даних для системи електронної комерції – інтернет-магазину побутової техніки. Проаналізовано різні типи систем керування базами даних, їх переваги та недоліки, а також обґрунтовано вибір PostgreSQL як оптимальної СУБД для проєкту.

У процесі роботи детально вивчено архітектуру Spring Framework з особливою увагою до модуля Spring Data JPA, що дозволив спростити доступ до даних, реалізувати CRUD-операції, зв’язки між сутностями, пагінацію, транзакційність та автоматичну генерацію SQL-запитів. Такий підхід забезпечив ефективну інтеграцію логіки застосунку з базою даних без надмірного ручного кодування.

Практична частина роботи передбачала створення структури реляційної бази даних, включно з ключовими сутностями: користувачі, товари, замовлення, виробники, категорії тощо. Реалізовано логічні зв’язки між таблицями (OneToOne, ManyToOne, ManyToMany), застосовано індексацію для оптимізації запитів, а також впроваджено базовий механізм резервного копіювання для підвищення надійності системи.

ПРОДОВЖЕННЯ ВИСНОВКІВ

Впровадження даної бази даних показало, що реляційна модель, при правильному проєктуванні, здатна забезпечити високу цілісність, узгодженість і масштабованість системи. Робота також підтвердила ефективність використання Spring Data JPA як потужного інструмента для інтеграції реляційної СУБД із вебдодатками, створеними на Java.

У перспективі розвиток застосунку може передбачати впровадження складніших запитів, використання кешування та оптимізацію схем даних. Подальше вдосконалення системи управління даними дозволить забезпечити ще вищу продуктивність, безпеку та зручність користування інтернет-магазином.

РЕЗУЛЬТАТИ ВПРОВАДЖЕННЯ

1. Самбор І. В., Бученко І. А. Визначення вимог до бази даних для вебдодатку «Магазин побутової техніки» на основі SPRING. Застосування програмного забезпечення в ІКТ : VI Всеукр. науково-техн. конф., м. Київ, 24 квіт. 2025 р. Київ, 2025. С. 454–457. URL: https://duikt.edu.ua/uploads/p_2779_56719466.pdf
2. Самбор І. В., Бученко І. А. Розробка вебдодатку «магазин побутової техніки» на основі SPRING. Застосування програмного забезпечення в ІКТ : VI Всеукр. науково-техн. конф., м. Київ, 24 квіт. 2025 р. Київ, 2025. С. 22–23. URL: https://duikt.edu.ua/uploads/p_2779_56719466.pdf