**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**
**Національний університет «Острозька академія»**
**Навчально-науковий інститут інформаційних технологій та бізнесу**
**Кафедра інформаційних технологій та аналітики даних**

**КВАЛІФІКАЦІЙНА РОБОТА**
на здобуття освітнього ступеня бакалавра

на тему: «**Проєктування бази даних та архітектури серверної частини маркетплейсу вживаних автомобілів»**

**Виконав:** студент 4 курсу, групи КН-42
першого (бакалаврського) рівня вищої освіти
спеціальності 122 Комп'ютерні науки
освітньо-професійної програми «Комп'ютерні науки»
*Степанюк Дмитро Андрійович*

**Керівник:** *викладач* кафедри інформаційних технологій та аналітики даних
*Мацевич Денис Володимирович*

**Рецензент:** *ФОП в галузі інформаційних технологій, викладач кафедри менеджменту та маркетингу НаУОА*
*Шпак Андрій Григорович*

***РОБОТА ДОПУЩЕНА ДО ЗАХИСТУ***

Завідувач кафедри інформаційних технологій та аналітики даних _____
проф., д.е.н. Кривицька О.Р.
Протокол №10 від 28 травня 2025 р.

Острог, 2025

**АНОТАЦІЯ**

**кваліфікаційної роботи**

**на здобуття освітнього ступеня бакалавра**

***Тема: Проєктування БД та архітектури серверної частини маркетплейсу вживаних автомобілів***

***Автор: Степанюк Дмитро Андрійович***

***Науковий керівник: Мацевич Д. В., викладач кафедри ІТАД***

***Захищена «……»…… 2025 року.***

***Пояснювальна записка до кваліфікаційної роботи:*** *107 с., 10 рис., 1 табл., 0 додатків, 27 джерел.*

***Ключові слова:*** *маркетплейс, база даних, архітектура програмного забезпечення, Code-First, Entity Framework Core, Health Checks, OpenTelemetry, архітектурні тести, CI/CD, GitHub Actions.*

***Короткий зміст праці:*** *Ця робота присвячена проєктуванню бази даних та архітектури серверної частини маркетплейсу вживаних автомобілів. У рамках роботи реалізовано базу даних за підходом Code-First з використанням Entity Framework Core, включаючи відображення сутностей, їх конфігурацію за допомогою Fluent API та механізми попереднього заповнення даних (Data Seeding). Особливу увагу приділено забезпеченню якості та стабільності архітектури через впровадження автоматизованих архітектурних тестів. Для моніторингу стану застосунку та його зовнішніх залежностей інтегровано Health Checks, а для збору метрик продуктивності — систему телеметрії на базі OpenTelemetry. Додатково, для автоматизації процесів розробки та розгортання, налаштовано CI/CD конвеєр*

*за допомогою GitHub Actions, з політикою захисту основної гілки, що вимагає успішного проходження всіх тестів перед об'єднанням змін.*

***Short summary of the work:*** *This work is devoted to the design of the database and the architecture of the server side of the used car marketplace. As part of the work, a database was implemented using the Code-First approach using the Entity Framework Core, including the display of entities, their configuration using the Fluent API, and mechanisms for pre-filling data (Data Seeding). Special attention is paid to ensuring the quality and stability of the architecture through the implementation of automated architectural tests. Health Checks are integrated to monitor the state of the application and its external dependencies, and a telemetry system based on OpenTelemetry is used to collect performance metrics. Additionally, to automate the development and deployment processes, a CI/CD pipeline is configured using GitHub Actions, with a main branch protection policy that requires successful passing of all tests before merging changes.*

**ЗМІСТ**

# ВСТУП

**Актуальність** обраної теми дослідження визначається інтенсивним зростанням ринку вживаних автомобілів та невпинним збільшенням частки онлайн-торгівлі у загальному обсязі комерційних операцій. У цьому контексті, розробка спеціалізованого онлайн-майданчика (маркетплейсу) стає критично важливою для оптимізації процесу купівлі-продажу. Такий підхід дозволяє ефективно усунути ланцюг посередників, забезпечуючи прямий та прозорий зв'язок між власниками автомобілів, які бажають їх продати, та потенційними покупцями. Це не тільки спрощує угоду, але й потенційно знижує витрати для обох сторін.

**Основною метою** даного дослідження є розробка комплексного технічного рішення для створення повноцінного маркетплейсу вживаних автомобілів. Ця мета досягається шляхом проектування та реалізації надійної, високопродуктивної та масштабованої бази даних, використовуючи сучасні технології управління базами даних, що відповідають вимогам до обробки значних обсягів інформації. Паралельно з цим, критично важливим етапом є визначення та проектування оптимальної архітектури веб-застосунку, вибір відповідних технологічних стеків та фреймворків, які забезпечать його функціональність, ефективність, безпеку та зручність подальшого розвитку й масштабування. Не менш значущим завданням є планування та налаштування відповідної інфраструктури, що включатиме вибір та конфігурацію серверних ресурсів, розгортання застосунку, забезпечення резервного копіювання та безперебійної роботи системи для користувачів маркетплейсу.

**Проблематика**, яка стоїть перед даним дослідженням, полягає у необхідності створення стійкої та продуктивної системи для ефективного управління значним обсягом даних високої складності. Це зумовлено тим, що кожен об'єкт (автомобіль) володіє великою кількістю різнорідних характеристик (таких як модель, тип кузова, двигуна, приводу, колір, рік випуску, ціна тощо), що вимагає розробки комплексної та логічно організованої структури бази даних для забезпечення їхнього зберігання, обробки та оперативного доступу.

Крім того, система повинна бути спроможна витримувати значне навантаження, спричинене одночасним зверненням великої кількості користувачів, які активно виконують операції пошуку, перегляду та фільтрації даних за різними критеріями. Така інтенсивність взаємодії вимагає постійної оптимізації продуктивності системи та швидкості обробки запитів.

Забезпечення високої надійності та доступності бази даних є критично важливим аспектом для уникнення збоїв, простоїв та гарантування безперебійної роботи сервісу навіть за умов пікових навантажень. Для реалізації цих вимог, а також для забезпечення можливості подальшого масштабування системи відповідно до зростання кількості користувачів та даних, необхідне продумане проектування та налаштування всієї технічної інфраструктури, включаючи використання сучасних підходів, таких як контейнеризація та оркестрація.

Основними завданнями курсової роботи є:

1. Провести детальний аналіз функціональних та нефункціональних вимог до системи маркетплейсу вживаних автомобілів. Це включає вивчення потреб потенційних користувачів та ключових бізнес-процесів, необхідних для визначення оптимальної структури даних та взаємозв'язків між сутностями системи.
2. На основі зібраних вимог спроєктувати логічну та фізичну моделі бази даних. Це передбачає створення деталізованих схем таблиць, визначення типів даних, первинних та зовнішніх ключів, а також зв'язків між різними сутностями.
3. Реалізувати спроєктовану базу даних та механізми взаємодії з нею, використовуючи підхід Code First із застосуванням мови програмування C# та ORM (Object-Relational Mapper) Entity Framework Core.
4. Спроєктувати та розробити архітектуру веб-застосунку маркетплейсу, дотримуючись сучасних принципів розробки (наприклад, мікросервісної або монолітної архітектури з чітким розділенням рівнів), що забезпечить модульність, зручність розробки та подальшої підтримки.

5. Впровадити систему керування ідентичністю та доступом (Identity and Access Management), реалізувавши надійні механізми аутентифікації (перевірка особи користувача) та авторизації (надання прав доступу) користувачів із застосуванням спеціалізованої системи Keycloak.
6. Організувати процес початкового наповнення бази даних (Data Seeding) тестовими або демонстраційними даними. Це необхідно для зручності локальної розробки, тестування функціональності та демонстрації роботи застосунку.
7. Виконати контейнеризацію компонентів розробленого застосунку за допомогою інструментів Docker та Docker Compose. Такий підхід спростить процес розгортання, забезпечить узгоджене функціонування різних частин системи в ізольованих середовищах та створить міцну основу для майбутнього масштабування інфраструктури.
8. Інтегрувати механізми Health Checks (перевірок стану здоров'я), які дозволять автоматично моніторити працездатність та готовність окремих сервісів та застосунку в цілому.
9. Впровадити інструменти моніторингу та спостережуваності (observability), такі як OpenTelemetry (або інші аналогічні рішення), для збору метрик продуктивності, трасування запитів, агрегації логів та полегшення діагностики потенційних проблем у системі.
10. Розробити та реалізувати набір автоматизованих тестів, зокрема архітектурних тестів, для перевірки відповідності структури коду, залежностей та загальної організації застосунку визначеним архітектурним принципам та стандартам.

**Об’єктом дослідження** в межах даної роботи є комплексна інформаційна система, що проектується та розробляється для функціонування як онлайн-маркетплейс вживаних автомобілів. Ця система виступає платформою, що забезпечує взаємодію між продавцями та покупцями, сприяючи ефективному процесу купівлі-продажу транспортних засобів через Інтернет.

**Предметом дослідження** є ключові технічні аспекти та компоненти цієї інформаційної системи, які забезпечують її основну функціональність, стабільність та продуктивність. До них належать:

1. **База даних**: Її логічна та фізична структури, моделі організації даних, а також механізми управління, які відповідають за надійне зберігання, ефективну обробку та оперативний доступ до значного обсягу складних даних, що описують характеристики автомобілів та інформацію про користувачів і транзакції.
2. **Технічна інфраструктура**: Архітектура розгортання та сукупність програмно-апаратних засобів, що забезпечують функціонування веб-застосунку. Це охоплює принципи побудови масштабованої та високодоступної інфраструктури, налаштування середовища виконання, а також механізми, що гарантують безперебійну роботу системи під навантаженням та швидкий доступ користувачів до необхідної інформації.

# РОЗДІЛ 1

# ТЕОРЕТИЧНІ ОСНОВИ ПРОЄКТУВАННЯ БАЗИ ДАНИХ ТА АРХІТЕКТУРИ СЕРВЕРНОЇ ЧАСТИНИ МАРКЕТПЛЕЙСУ ВЖИВАНИХ АВТОМОБІЛІВ.

## 1.1. Дослідження ринку вживаних автомобілів

Аналіз динаміки українського ринку вживаних легкових автомобілів за період 2021-2023 років, проведений на основі даних “УкрАвтоПрому” [1], демонструє суттєві зміни, спричинені насамперед повномасштабною війною та економічними трансформаціями в країні.

Так, 2021 рік став знаковим для ринку, зафіксувавши історично високий показник із приростом у 46% порівняно з попереднім роком. Загальна кількість первинних реєстрацій вживаних авто тоді сягнула 517,4 тис. одиниць.

Незважаючи на початок повномасштабного вторгнення у 2022 році, ринок продемонстрував відносну стійкість, скоротившись лише на 25%. Цьому ефекту частково сприяло запровадження періоду "нульового розмитнення". За рік було вперше зареєстровано 388,5 тис. вживаних автомобілів, а їх середній вік становив 13 років. Серед марок-лідерів за кількістю реєстрацій були:

1. Volkswagen (73,4 тис.)
2. Renault (31,8 тис.)
3. Ford (28,0 тис.)
4. Opel (25,4 тис.)
5. Skoda (25,3 тис.).

2023 рік ознаменувався подальшим скороченням та трансформацією ринку. Загальна кількість вперше зареєстрованих вживаних автомобілів склала 214,4 тис. одиниць, що на 45% менше, ніж у 2022 році. При цьому спостерігалася тенденція до "омолодження" автопарку – середній вік зареєстрованих авто зменшився до 10 років. У структурі реєстрацій 2023 року серед моделей лідирували:

1. Volkswagen Golf (16 123 од.)
2. Renault Megane (14 392 од.)

3. Skoda Octavia (10 261 од.)
4. Volkswagen Passat (8 543 од.)
5. Електромобіль Nissan Leaf (5 190 од.).

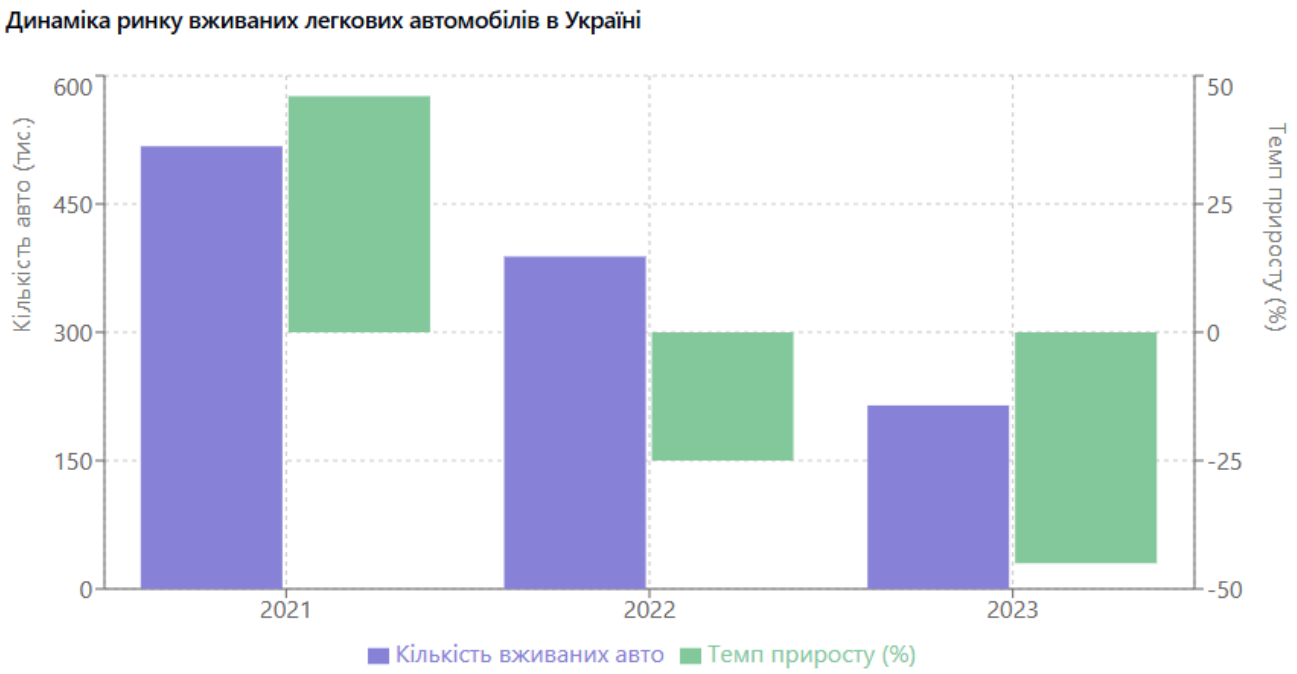


Рис. 1.1 – графічне представлення динаміки ринку вживаних автомобілів в Україні. Джерело: створено автором за даними “УкрАвтоДору” [1]

За результатами аналізу, проведеного "УкрАвтоДором" [1], після періоду значних трансформацій український ринок вживаних автомобілів демонструє чітку тенденцію до відновлення та подальшого зростання. Підтвердженням цього є дані Держстату, наведені в аналізі, згідно з якими за січень-серпень в Україну було імпортовано 215,6 тис. легкових автомобілів (як вживаних, так і нових), що на 15% перевищує обсяги імпорту за аналогічний період попереднього року. Це зростання свідчить про поступове відновлення споживчого попиту та активізацію ринку.

Існує вагоме припущення, що після завершення воєнних дій та з початком повномасштабного економічного відновлення країни, ринок вживаних автомобілів отримає ще потужніший імпульс для розвитку, що призведе до значного зростання його обсягів. Збільшення купівельної спроможності населення та потреба в оновленні автопарку після складного періоду можуть стати основними драйверами цього зростання.

Така позитивна динаміка та прогнозований значний потенціал для розвитку роблять дану галузь надзвичайно привабливою та економічно доцільною для інвестицій у сучасні технологічні рішення. Це, в свою чергу, переконливо обґрунтовує актуальність та стратегічну важливість реалізації проєкту зі створення сучасного онлайн-маркетплейсу для вживаних автомобілів, який зможе ефективно задовольнити потреби ринку, що відновлюється.

## 1.2. Фундаментальні поняття баз даних та специфіка їх застосування на платформах типу маркетплейс

В основі будь-якої сучасної інформаційної системи, зокрема онлайн-маркетплейсу, лежить база даних – організована колекція структурованих даних, що зазвичай зберігається в електронному вигляді та контролюється спеціалізованим програмним забезпеченням, відомим як система управління базами даних (СУБД). Сукупність самої бази даних, відповідної СУБД та пов'язаних із ними застосунків утворює повноцінну систему баз даних [2]. Ключовими перевагами використання баз даних є можливість ефективного зберігання значних обсягів різнорідної інформації, забезпечення швидкого доступу до неї та, що не менш важливо, реалізація механізмів розмежування доступу, які надають повноваження на управління даними лише авторизованим користувачам. Це критично важливо для забезпечення цілісності даних та їх захисту від несанкціонованої втрати чи модифікації.

Класифікація баз даних здійснюється за різними ознаками, проте найчастіше їх розділяють за моделлю організації та зберігання даних. Серед найбільш поширених моделей виділяють:

- **Реляційні бази даних (RDBMS)**. Це історично один із перших та найбільш широко використовуваних типів баз даних. В основі реляційної моделі лежить організація даних у вигляді двовимірних таблиць, що складаються зі стовпців (атрибутів) та рядків (записів). Взаємозв'язок між окремими сутностями та даними встановлюється через зв'язки між таблицями, які реалізуються за

допомогою первинних та зовнішніх ключів, що забезпечує цілісність даних. Для маніпулювання даними (вибірки, додавання, зміни, видалення) в RDBMS використовується стандартизована декларативна мова структурованих запитів – SQL (Structured Query Language) [3]. Типовими представниками реляційних СУБД, що знайшли широке застосування, є PostgreSQL (яка використовується в межах цього проєкту), MySQL, Oracle Database та Microsoft SQL Server.

- **NoSQL бази даних**. Термін NoSQL, що є скороченням від "not only SQL" ("не тільки SQL"), охоплює широкий клас нереляційних баз даних. Ключовою відмінністю від RDBMS є відсутність жорсткої табличної структури та використання більш гнучкої моделі схеми. NoSQL бази даних спроектовані для підтримки широкого спектру неструктурованих, напівструктурованих та поліструктурованих даних, таких як документи (документо-орієнтовані БД), пари ключ-значення, широкі стовпці або графіки [4]. Вони часто є оптимальним вибором для роботи з великими обсягами даних (так звані Big Data), оскільки забезпечують високу швидкість операцій, гнучкість та горизонтальну масштабованість, що дозволяє легко розподіляти дані та навантаження між багатьма серверами. На відміну від реляційних систем, для NoSQL баз даних відсутня єдина універсальна мова запитів, що може вважатися певним недоліком. Серед популярних NoSQL СУБД – MongoDB, Redis, Cassandra та інші.
- **Об'єктні бази даних (ООБД)**. Цей тип систем управління базами даних представляє інформацію у вигляді об'єктів, що відповідає концепції об'єктно-орієнтованого програмування (ООП) [5]. На відміну від реляційних баз даних, орієнтованих на табличну модель, ООБД працюють безпосередньо з об'єктами та їхніми взаємозв'язками. Їх часто розглядають як гібридний підхід. Головною перевагою ООБД є природна інтеграція з об'єктно-орієнтованими мовами програмування, що дозволяє розробникам працювати з даними у звичному для ООП стилі, зменшуючи необхідність у складному перетворенні даних між об'єктною моделлю застосунку та реляційною моделлю БД (так звана проблема "об'єктно-реляційного невідповідності"). Проте, через

відносну складність налаштування, управління та меншу поширеність інструментів, ООБД мають нижчий попит порівняно з реляційними та NoSQL рішеннями. Прикладами об'єктно-орієнтованих СУБД є db4o та ObjectDB.

У контексті інформаційної системи маркетплейсу вживаних автомобілів, база даних відіграє центральну, критично важливу роль. Вона є основою для ефективного управління всією інформацією про автомобілі, оголошення, користувачів, їхні профілі, історію транзакцій, відгуки тощо. Саме база даних забезпечує зберігання цих даних та є джерелом інформації для всіх функцій платформи.

Для успішного функціонування такого ресурсу, що має справу зі значним обсягом динамічних даних та високим рівнем взаємодії користувачів, база даних повинна відповідати ряду ключових характеристик:

- **Здатність обробляти великі обсяги даних**: Система має надійно зберігати та ефективно керувати інформацією про десятки тисяч унікальних оголошень про автомобілі, їхні детальні характеристики (модель, комплектація, історія експлуатації), численні фотографії та іншу пов'язану інформацію. Для ефективного управління такими обсягами критично важливим є застосування інструментів оптимізації зберігання, таких як індексація даних для значного прискорення пошукових операцій та розбиття даних на схеми (sharding) для забезпечення горизонтального масштабування при експоненційному зростанні обсягів інформації.
- **Висока продуктивність та швидкість доступу**: База даних повинна забезпечувати мінімальний час відгуку при обробці численних одночасних запитів від користувачів. Це включає швидке виконання складних пошукових запитів за різноманітними параметрами (марка, модель, ціна, рік випуску, тип двигуна тощо), ефективну фільтрацію результатів та миттєве завантаження детальної інформації про обраний автомобіль. Досягнення необхідної швидкості реалізується шляхом ретельної оптимізації індексів, ефективного використання агрегатних функцій та оптимізації самого виконання запитів.

- **Масштабованість**: База даних та супутня інфраструктура мають бути спроможними підтримувати зростання як обсягу даних, так і рівня навантаження, спричиненого збільшенням кількості активних користувачів та їхніх запитів. Масштабованість дозволяє системі розвиватися без необхідності суттєвої перебудови, забезпечуючи стабільну продуктивність навіть за умов значного росту популярності платформи.
- **Надійність та доступність**: Для онлайн-сервісу, який має працювати 24/7, критично важливим є забезпечення високої надійності та безперебійної доступності бази даних. Це вимагає впровадження механізмів резервного копіювання, реплікації даних та кластеризації, які мінімізують ризик втрати даних та забезпечують швидке відновлення роботи системи у випадку збоїв.
- **Безпека даних**: Необхідно забезпечити високий рівень захисту конфіденційної інформації користувачів (персональні дані, контактна інформація). Це досягається шляхом застосування сучасних методів шифрування для зберігання чутливих даних та впровадження суворих політик контролю доступу. Політики доступу повинні чітко розмежовувати права користувачів, надаючи доступ до даних лише уповноваженим особам та гарантуючи відповідність вимогам законодавства щодо захисту персональних даних.

Таким чином, вибір, проектування та реалізація відповідної моделі бази даних, а також забезпечення її високої продуктивності, масштабованості та безпеки є фундаментальними завданнями при створенні успішного маркетплейсу вживаних автомобілів.

## 1.3. Основи реляційної моделі баз даних

### 1.3.1. Основні поняття

Серед розмаїття підходів до організації та управління даними, реляційна модель є фундаментальною та домінує у сучасних системах баз даних, зокрема і в складних інформаційних системах, як-от онлайн-маркетплейси. Ця модель була вперше теоретично обґрунтована доктором Едгаром Коддом (Edgar Codd) у 1970

році та базується на чітких математичних принципах реляційної алгебри та теорії множин [6].

В основі реляційної моделі лежить концепція представлення даних у вигляді двовимірних таблиць, які формально називаються відношеннями (Relations). Кожна таблиця складається з:

- **Рядків (Кортежів, Записів)**: Кожен рядок представляє собою окремий екземпляр сутності або об'єкта реального світу чи абстрактного поняття, що моделюється в базі даних (наприклад, один конкретний автомобіль, зареєстрований користувач, здійснене замовлення).
- **Стовпців (Атрибутів, Полів)**: Кожен стовпець відповідає за певну характеристику, властивість або ознаку сутності, представленої рядками (наприклад, марка автомобіля, його ціна, рік випуску, ім'я користувача, дата замовлення).

Ключовими елементами, що визначають структуру та взаємозв'язки у реляційній моделі, є:

- **Таблиці (Relations)**: Як зазначено вище, це основні структури для зберігання даних, представлені у вигляді рядків та стовпців.
- **Ключі**: Атрибути або набори атрибутів, що слугують для ідентифікації рядків. **Первинний ключ (Primary Key)** – це один або кілька стовпців, значення яких однозначно ідентифікують кожен рядок у таблиці. Він гарантує унікальність кожного запису. **Зовнішні ключі (Foreign Keys)** використовуються для встановлення логічних зв'язків між таблицями. Зовнішній ключ в одній таблиці посилається на первинний ключ іншої таблиці, створюючи таким чином відношення між сутностями.
- **Зв'язки (Relationships)**: Логічні асоціації або відношення між сутностями, які на фізичному рівні реалізуються за допомогою механізму зовнішніх ключів. Вони описують, як екземпляри однієї таблиці (сутності) пов'язані з екземплярами іншої.

Для ефективного проектування реляційної бази даних, необхідно чітко розуміти фундаментальні концепції моделювання даних: **сутність, атрибут та зв'язок**. Ці поняття є наріжними каменями, що дозволяють формалізувати предметну область та побудувати логічну структуру бази даних. Вони визначають, які об'єкти реального світу або абстрактні поняття будуть представлені у вигляді таблиць (сутностей), які їхні характеристики будуть зберігатися у стовпцях (атрибутах) та як ці об'єкти пов'язані між собою (зв'язки). Розуміння цих концепцій є запорукою побудови зрозумілої, логічної та ефективної структури даних, яка зможе адекватно відобразити складну бізнес-логіку маркетплейсу.

- **Сутність (Entity)**: Об'єкт або поняття, що має самостійне значення та може бути чітко ідентифіковане в контексті предметної області. Кожна сутність у базі даних зазвичай відповідає окремій таблиці. Прикладами сутностей у маркетплейсі автомобілів можуть бути "Автомобіль", "Користувач", "Продавець", "Оголошення", "Замовлення", "Бренд (Виробник)".
- **Атрибут (Attribute)**: Властивість, характеристика або опис сутності. Атрибути стають стовпцями в таблиці, що представляє сутність, і визначають тип інформації, яку потрібно зберігати для кожного екземпляра сутності. Для сутності "Автомобіль" атрибутами можуть бути "Марка", "Модель", "Рік випуску", "Ціна", "Пробіг", "Тип кузова", "Колір" тощо.
- **Зв'язок (Relationship)**: Відношення або асоціація між двома чи більше сутностями. Зв'язки описують, як екземпляри однієї сутності пов'язані з екземплярами іншої (наприклад, "Користувач *створює* Оголошення", "Автомобіль *належить до* Бренда"). У реляційній моделі ці зв'язки реалізуються на фізичному рівні через використання зовнішніх ключів, що встановлюють посилання між відповідними таблицями.

Таблиця 1.1. Типи зв’язків в реляційній моделі баз даних [27]

| Тип зв’язку | Опис |
|---|---|
| Один-до-одного | Обидві таблиці можуть мати лише по одному запису з кожного боку зв'язку. Кожне значення первинного ключа пов'язане з жодним або тільки з одним записом у пов'язаній таблиці. Більшість зв'язків «один-до-одного» встановлюються бізнес-правилами і не випливають природно з даних. Без такого правила ви можете об'єднати обидві таблиці, не порушуючи жодних правил нормалізації. |
| Один-до-багатьох | Таблиця первинного ключа містить лише один запис, який пов'язаний з жодним, одним або багатьма записами у пов'язаній таблиці. |
| Багато-до-багатьох | Кожен запис в обох таблицях може бути пов'язаний з жодним або з будь-якою кількістю записів в іншій таблиці. Ці зв'язки потребують третьої таблиці, яка називається асоційованою або зв'язуючою таблицею, оскільки реляційні системи не можуть безпосередньо підтримувати ці зв'язки. |

### 1.3.2. Переваги

Основною та найбільш вагомою перевагою реляційної моделі є її висока гнучкість та потужні можливості для вибірки та маніпулювання даними, що забезпечується чіткою структурованістю інформації. Завдяки використанню

стандартизованої мови запитів SQL, користувачі та застосунки можуть отримувати саме ту інформацію, яка їм потрібна, використовуючи складні фільтри, сортування, групування та з'єднання даних з різних таблиць. Це дозволяє уникнути надлишковості даних у результатах запитів та суттєво підвищити швидкість їх отримання порівняно з менш структурованими моделями.

Реляційні бази даних також тісно пов'язані з концепцією транзакцій – послідовності операцій, які виконуються як єдине, неподільне ціле. Для гарантування надійності, цілісності та узгодженості даних під час виконання транзакцій, реляційні СУБД дотримуються набору властивостей, відомих як ACID:

- **Атомарність (Atomicity)**: Транзакція є атомарною, тобто виконується або повністю, або не виконується взагалі. У разі будь-якої помилки під час виконання транзакції всі зміни, внесені в базу даних в рамках цієї транзакції, автоматично відкочуються (rollback), повертаючи базу даних до стану, що передував її початку. Це унеможливлює перебування системи у проміжному, некоректному стані.
- **Консистентність (Consistency)**: Кожна успішно завершена транзакція (commit) переводить базу даних з одного коректного стану в інший коректний стан. Транзакції не порушують встановлені правила та обмеження цілісності даних (наприклад, обмеження первинних/зовнішніх ключів, унікальності, перевірочні обмеження).
- **Ізоляція (Isolation)**: Паралельно виконувані транзакції ізольовані одна від одної. Зміни, внесені однією транзакцією, стають видимими для інших транзакцій лише після її повного успішного завершення. Це гарантує, що транзакції не впливають на результати одна одної, створюючи ілюзію їх послідовного виконання [5].
- **Надійність (Durability)**: Після успішного завершення транзакції, зміни, внесені в базу даних, є постійними та зберігаються навіть у випадку системних збоїв, таких як відключення живлення або крах сервера. Це

досягається шляхом запису змін у стійке сховище даних (наприклад, лог транзакцій).

### 1.3.3. Нормалізація

Крім транзакцій, реляційні бази даних нерозривно пов'язані з процесом нормалізації. **Нормалізація** бази даних – це систематичний підхід до проектування схеми бази даних, спрямований на зменшення або повне усунення надмірності (дублювання) даних та покращення їх логічної структури та цілісності [7]. Надлишковість даних не тільки неефективно використовує дисковий простір, але й призводить до проблем з узгодженістю даних під час операцій вставки, оновлення та видалення (аномалії вставки, оновлення, видалення). Процес нормалізації передбачає приведення структури таблиць до певних "нормальних форм", яких існує кілька, але найбільш важливими та поширеними є перші три:

- **Перша нормальна форма (1NF)**: Таблиця знаходиться в 1NF, якщо кожен її атрибут є атомарним (неподільним) і в таблиці відсутні повторювані групи атрибутів. Це означає, що в кожній клітинці таблиці має бути лише одне одиничне значення, а не список значень.
- **Друга нормальна форма (2NF)**: Таблиця знаходиться в 2NF, якщо вона перебуває в 1NF, і кожен неключовий атрибут повністю функціонально залежить від усього складеного первинного ключа. Це усуває часткові залежності, коли атрибут залежить лише від частини складеного ключа. Наприклад, якщо первинний ключ складається з двох стовпців, а інший стовпець залежить тільки від одного з них, то таблиця не в 2NF.
- **Третя нормальна форма (3NF)**: Таблиця знаходиться в 3NF, якщо вона перебуває в 2NF, і жоден неключовий атрибут не має транзитивної функціональної залежності від первинного ключа. Транзитивна залежність існує, коли неключовий атрибут залежить від іншого неключового атрибута, який, у свою чергу, залежить від первинного ключа. По суті, це означає, що

атрибути мають залежати "від ключа, всього ключа і ні від чого іншого, крім ключа".

Нормалізація, як процес, має свої сильні та слабкі сторони. Перевагами є значне підвищення цілісності даних за рахунок уникнення їх дублювання, зменшення обсягів необхідного сховища, мінімізація ризику аномалій при оновленні даних та спрощення логічного управління структурою бази даних. Однак, основним недоліком високого ступеня нормалізації є те, що дані часто розподілені між багатьма таблицями, що вимагає виконання складних операцій з'єднання (JOIN) при отриманні агрегованої інформації. Це може негативно впливати на продуктивність запитів, особливо в системах зі значним навантаженням та великими обсягами даних. Також складність операцій вставки, оновлення та видалення може зростати через необхідність внесення змін у кілька пов'язаних таблиць.

### 1.3.4. Денормалізація

Ці потенційні проблеми з продуктивністю, спричинені високою нормалізацією, іноді призводять до необхідності застосування денормалізації. Денормалізація – це стратегія оптимізації бази даних, що полягає у свідомому та контрольованому додаванні певного рівня надмірності даних до однієї або декількох таблиць [8]. Основна мета денормалізації – покращити продуктивність запитів шляхом зменшення кількості або усунення необхідності виконання ресурсомістких операцій з'єднання (JOIN). Важливо наголосити, що денормалізація не є скасуванням нормалізації; це метод оптимізації, який застосовується після того, як схема вже була нормалізована до певного рівня. Під час денормалізації розробник приймає компромісне рішення: погодитися на певну надмірність та потенційно більші зусилля при оновленні даних задля отримання переваг у швидкості читання даних. Цей підхід використовується для тонкого налаштування продуктивності критично важливих за часом операцій.

Таким чином, нормалізація та денормалізація є важливими, хоча і протилежно спрямованими, концепціями при проектуванні реляційних баз даних. Нормалізація є фундаментальним кроком для забезпечення логічної цілісності, зменшення

дублювання та спрощення структури. Денормалізація ж є методом оптимізації, який може суттєво підвищити продуктивність запитів, але вимагає ретельного аналізу та управління, щоб уникнути проблем з узгодженістю даних та аномаліями при їх модифікації. Оптимальне проектування реляційної бази даних часто передбачає балансування між цими двома підходами, вибираючи ступінь нормалізації та застосовуючи денормалізацію там, де це виправдано вимогами до продуктивності.

## 1.4. Специфікація вимог до бази даних інформаційної системи маркетплейсу вживаних автомобілів

Процес проектування та реалізації бази даних для інформаційної системи онлайн-маркетплейсу вживаних автомобілів є критично важливим етапом, що вимагає глибокого та всебічного визначення комплексу вимог. Ці вимоги покликані забезпечити не тільки належну функціональність, високу продуктивність та надійну безпеку самої бази даних, але й тісно корелювати з бізнес-цілями та потребами платформи. База даних є основою, яка повинна ефективно підтримувати всі ключові операції маркетплейсу: від розміщення та відображення оголошень, здійснення пошуку та фільтрації автомобілів за різноманітними критеріями до безпечної обробки транзакцій та забезпечення безперебійної взаємодії між усіма категоріями користувачів – продавцями, покупцями та адміністраторами.

Для систематизації та забезпечення повноти, вимоги до бази даних були деталізовані та розділені на три основні категорії: функціональні вимоги, нефункціональні вимоги та вимоги до безпеки.

### 1.4.1. Функціональні вимоги

Ця категорія визначає специфічний набір операцій, можливостей та функцій, які база даних має надавати для коректної реалізації основної бізнес-логіки інформаційної системи. Функціональні вимоги включають:

- **Структурованість та модель даних**: База даних повинна мати чітко визначену та логічно організовану структуру таблиць та сутностей, яка адекватно відображає предметну область маркетплейсу. Це передбачає

проектування та створення таблиць для зберігання детальної інформації про автомобілі (включаючи марку, модель, рік випуску, колір, пробіг, стан), даних про зображення, ціни, а також вичерпного набору технічних характеристик (тип двигуна, трансмісії, кузова). Критично важливим є створення коректних зв'язків між цими сутностями (наприклад, зв'язок між автомобілем та власником, автомобілем та його виробником) для забезпечення цілісності даних та логічної узгодженості інформації в системі.

- **Підтримка розширеного пошуку та ефективної фільтрації**: База даних має бути спроможною обробляти та швидко виконувати складні запити (SQL-запити у випадку реляційної моделі) для здійснення пошуку автомобілів за широким спектром параметрів, таких як діапазон ціни, пробігу, конкретний рік випуску або діапазон років, тип кузова, тип пального, трансмісії тощо. Крім того, необхідно реалізувати підтримку гнучких, динамічних фільтрів, які дозволяють користувачам комбінувати кілька критеріїв одночасно для максимально точного пошуку бажаного автомобіля серед тисяч оголошень.
- **Гарантування транзакційності операцій**: Система управління базами даних повинна повністю підтримувати транзакційний підхід до обробки операцій модифікації даних. Це означає, що всі операції додавання нових оголошень, оновлення інформації про існуючі автомобілі або видалення даних мають виконуватись як єдине, неподільне ціле (атомарно) з дотриманням властивостей ACID – Атомарності, Консистентності, Ізоляції та Надійності. Це гарантує, що всі зміни будуть застосовані коректно, дані залишаться узгодженими, а системні збої не призведуть до часткового виконання операцій та порушення цілісності бази даних.

### 1.4.2. Нефункціональні вимоги

Ця категорія описує якісні характеристики системи, які не стосуються безпосередньо функціоналу, але суттєво впливають на її продуктивність, надійність, зручність використання та загальну ефективність роботи маркетплейсу. Нефункціональні вимоги включають:

- **Продуктивність (Швидкодія)**: База даних повинна демонструвати високу швидкість виконання всіх операцій, особливо запитів на вибірку та фільтрацію даних, навіть за умови високого паралельного навантаження від значної кількості одночасних користувачів. Досягнення необхідної швидкодії вимагає оптимального проектування схеми даних, ретельної організації індексації відповідних полів, а також постійної оптимізації виконання складних запитів для мінімізації часу їх обробки.
- **Масштабованість (Scalability)**: Система повинна бути спроможна ефективно обробляти експоненційне зростання як обсягу даних (збільшення кількості оголошень, зареєстрованих користувачів), так і рівня навантаження, спричиненого зростанням популярності платформи. Має бути передбачена можливість гнучкого масштабування, зокрема горизонтального – розподілу даних та обробного навантаження між кількома серверами або вузлами, наприклад, за допомогою технік реплікації (створення та підтримка копій бази даних) або шардингу (логічного розподілу великої бази даних на менші, більш керовані частини).
- **Надійність (Reliability)**: База даних має бути стійкою до різного роду збоїв, включаючи апаратні відмови, помилки програмного забезпечення або інші непередбачені обставини, що можуть призвести до некоректної роботи або потенційної втрати даних. Для запобігання втраті даних та забезпечення можливості їх відновлення у разі інцидентів, критично важливим є регулярне, автоматизоване створення та безпечне зберігання резервних копій (бекапів).
- **Доступність (Availability)**: Система бази даних повинна забезпечувати мінімально можливий час простою (downtime) та гарантувати користувачам практично безперебійний, цілодобовий доступ до функціоналу маркетплейсу. Це досягається шляхом використання кластерних рішень, механізмів відмовостійкості (failover), моніторингу стану системи та автоматичного перемикання на резервні ресурси у випадку збоїв.

### 1.4.3. Вимоги до безпеки

Враховуючи, що база даних маркетплейсу буде містити значний обсяг чутливої та конфіденційної інформації про користувачів (персональні дані, контактна інформація, можливо, фінансові дані або історія транзакцій), забезпечення високого рівня безпеки є одним із пріоритетних завдань. Ключові вимоги до безпеки включають:

- **Контроль доступу (Access Control)**: Необхідно реалізувати гнучку та надійну систему управління доступом на основі ролей та привілеїв користувачів. Ця система має чітко визначати, які операції з даними (читання, запис, зміна, видалення) дозволені для різних категорій користувачів (наприклад, адміністратори, модератори, зареєстровані продавці, зареєстровані покупці, анонімні користувачі). Це дозволяє обмежувати доступ до даних відповідно до їхньої ролі та повноважень у системі.
- **Захист від втрати, несанкціонованої модифікації та ін'єкцій**: Потрібно вжити комплексних заходів для захисту цілісності та конфіденційності даних. Це включає організацію регулярного та автоматизованого створення резервних копій для швидкого відновлення даних у разі їх втрати, пошкодження або видалення. Важливим інструментом є ведення детального журналу змін (audit log), який фіксує всі операції модифікації даних, ідентифікує користувача, що їх виконав, та час виконання, що дозволяє проводити аудит та відстежувати джерело змін. Крім того, необхідно забезпечити захист від типових веб-вразливостей, таких як SQL-ін'єкції, шляхом використання параметризованих запитів замість динамічного формування SQL-рядків та впровадження суворої валідації всіх вхідних даних, що надходять від користувачів.
- **Шифрування даних (Data Encryption)**: Вся чутлива інформація, що зберігається в базі даних (наприклад, паролі користувачів мають зберігатися у хешованому вигляді, а не у відкритому тексті; можливо, частина контактних

даних або іншої персональної інформації), повинна бути належним чином зашифрована.

## 1.5. Значення та інструментальні засоби візуалізації при проектуванні структури баз даних

На початковому етапі розгляду даного аспекту, необхідно визначити ключове значення візуалізації як важливого інструменту для документування та ефективної комунікації в процесі розробки програмного забезпечення. Візуалізація є невід'ємною складовою життєвого циклу розробки будь-якого програмного продукту, і особливо критичною вона стає при проектуванні та роботі з базами даних. Вона дозволяє представити складну технічну інформацію у наочному, легкому для сприйняття форматі, значно спрощує спільну роботу над проєктом у команді та забезпечує краще, уніфіковане розуміння структури системи серед усіх її учасників, незалежно від їхньої технічної спеціалізації чи ролі [9]. Розглянемо детальніше конкретні переваги візуалізації:

Візуалізація суттєво спрощує розуміння складних систем. Графічне представлення схеми бази даних, виконане у вигляді діаграми "сутність-зв'язок" (Entity-Relationship Diagram, ERD), дає можливість розробникам, архітекторам, аналітикам та іншим учасникам команди швидко охопити загальну структуру даних, побачити, як різні сутності (таблиці) пов'язані між собою, та зрозуміти логіку взаємодії компонентів системи. Такий наочний підхід ефективно зменшує когнітивне навантаження, оскільки людський мозок набагато легше та швидше сприймає та обробляє візуальну інформацію (схеми, діаграми, зв'язки) порівняно з аналізом великих обсягів текстових описів, специфікацій чи безпосередньо програмного коду, який створює структуру бази даних.

Візуалізація значно покращує комунікацію та взаємодію в команді. Візуальні матеріали, такі як ER-діаграми, стають своєрідною спільною "мовою", яка дозволяє розробникам, бізнес-аналітикам, дизайнерам інтерфейсів, менеджерам проєктів та навіть клієнтам мати єдине, узгоджене та чітке уявлення про структуру даних

системи, над якою вони спільно працюють. Наочно зображені структури таблиць, їхні атрибути, ключі та зв'язки допомагають уникнути неоднозначностей та неправильних інтерпретацій, які часто виникають при використанні виключно текстових специфікацій або вербального опису. Візуалізація також робить процес обговорень більш продуктивним та інклюзивним, дозволяючи учасникам з різним рівнем технічної експертизи ефективно долучатися до дискусій та процесу прийняття важливих рішень щодо архітектури та структури даних.

Візуалізація приносить значні дивіденди в довгостроковій перспективі. Детально розроблені та актуальні діаграми структури бази даних стають невід'ємною та цінною частиною технічної документації. Ця документація є критично важливою для подальшої підтримки, модифікації та сталого розвитку системи протягом усього її життєвого циклу. Крім того, наочне візуальне представлення структури суттєво спрощує процес навчання та введення в курс справи нових членів команди, полегшує передачу проєкту іншим командам (наприклад, команді підтримки чи подальшої розробки) та сприяє отриманню більш конструктивного зворотного зв'язку від зовнішніх експертів або зацікавлених сторін, оскільки вони можуть швидше зрозуміти логіку системи. Доступність візуальної моделі також підвищує прозорість проєкту.

Візуалізація є потужним інструментом для раннього виявлення потенційних проблем. Наочне представлення зв'язків та залежностей між таблицями дозволяє ще на етапі проектування помітити надлишкові зв'язки, порушення принципів нормалізації, потенційні "вузькі місця", що можуть негативно вплинути на продуктивність системи, або логічні неузгодженості у моделі даних – задовго до того, як буде написано значний обсяг коду чи база даних буде розгорнута в робочому середовищі.

Візуальні інструменти часто надають більш інтуїтивний та інтерактивний підхід до проектування. Працювати над схемою бази даних, вносячи поступові зміни безпосередньо на візуальній діаграмі, перетягуючи елементи, додаючи зв'язки, є значно зручніше, швидше та інтуїтивніше, ніж редагувати складний формальний текстовий опис структури.

З огляду на всі вищезазначені значні переваги, можна з упевненістю зробити висновок, що візуалізація структури бази даних є не просто корисною, а незамінною практикою для ефективного проектування, документування, комунікації та раннього виявлення проблем у сучасних інформаційних системах. Тому, в межах даної курсової роботи, для візуального моделювання та документування структури бази даних буде використовуватися спеціалізований інструмент.

Для попереднього проектування, розробки логічної моделі та візуалізації структури бази даних було обрано онлайн-платформу **dbdiagram.io**. Dbdiagram.io – це зручний, інтуїтивно зрозумілий та потужний онлайн-інструмент, спеціально розроблений для швидкого створення, редагування та наочної візуалізації діаграм "сутність-зв'язок" (ERD) для баз даних. Сервіс дозволяє представити навіть досить складну структуру бази даних у візуально зрозумілому та естетичному форматі, що значно полегшує її сприйняття, аналіз та обговорення. Серед його ключових переваг – простий та інтуїтивно зрозумілий інтерфейс, висока доступність як хмарної онлайн-платформи, що не потребує встановлення, та можливість інтеграції з іншими сервісами. Важливою перевагою є те, що dbdiagram.io є безкоштовним для індивідуального використання та більшості потреб студентських проєктів.

Особливістю та одночасно сильною стороною інструменту є використання спеціалізованої мови моделювання – **DBML (Database Markup Language)** [10]. DBML – це проста, легка для читання та написання DSL (доменно-специфічна) мова, яка була розроблена саме авторами dbdiagram.io спеціально для текстового опису структури баз даних. Вона дозволяє швидко та лаконічно визначити таблиці, їхні атрибути, типи даних, первинні та зовнішні ключі, індекси, а також зв'язки між таблицями, при цьому є можливість вбудовувати документацію безпосередньо у код моделі. Платформа автоматично генерує візуальну ER-діаграму на основі DBML-коду в реальному часі.

Крім ключових функцій моделювання та візуалізації, dbdiagram.io надає зручні можливості імпортування існуючих структур баз даних (наприклад, шляхом вставки SQL-скриптів CREATE TABLE) для їх подальшої візуалізації та модифікації. Також доступний функціонал експорту готової діаграми у різні формати (зображення, PDF),

а головне – можливість генерації SQL-скриптів на основі розробленої DBML-моделі. Це дозволяє швидко перейти від етапу логічного проектування до фізичної реалізації структури бази даних, виконавши згенерований SQL-скрипт у відповідній СУБД.

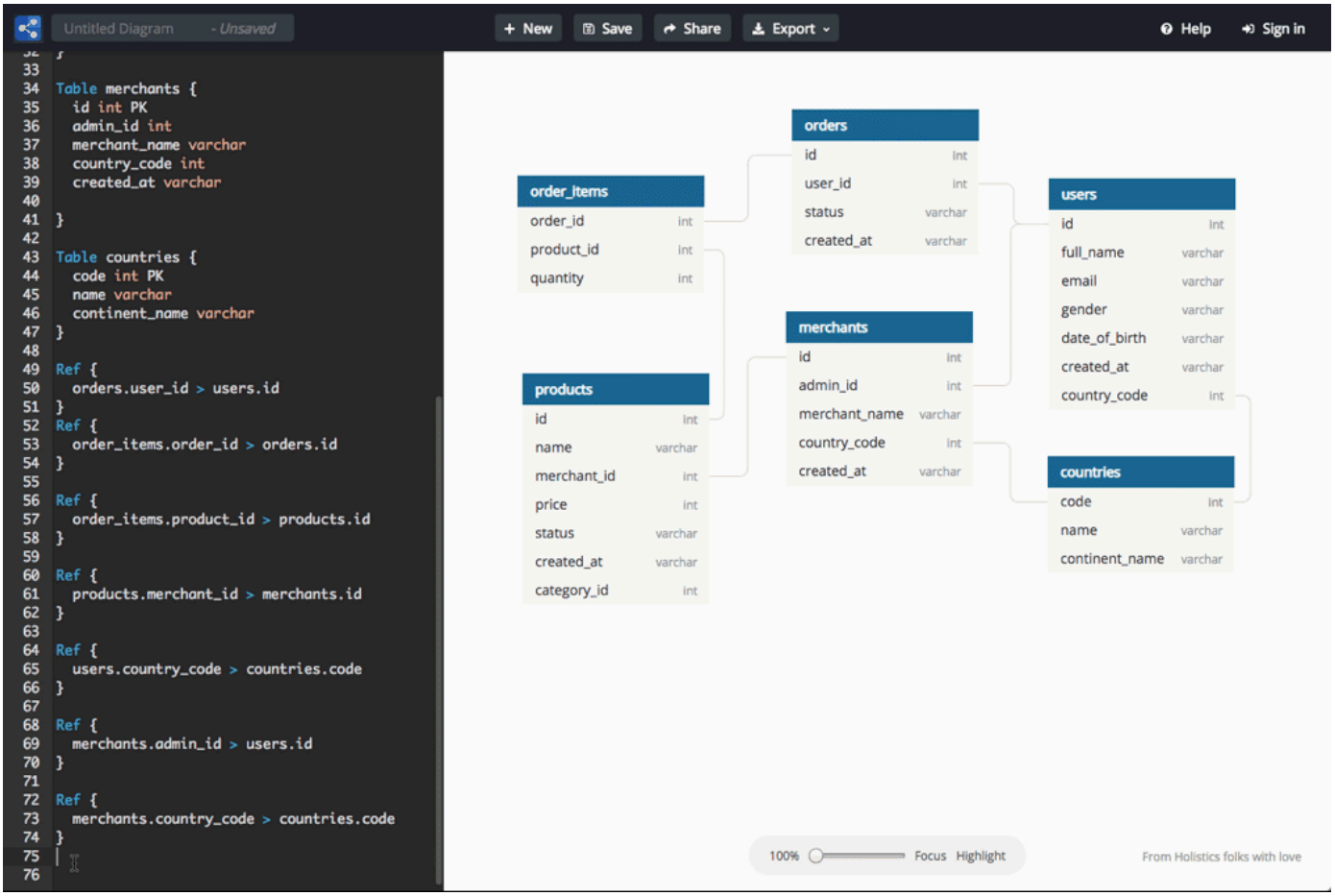


Рис. 1.2. Інтерфейс dbdiagrams.io

Джерело: https://docs.dbdiagram.io/

Варто зазначити, що dbdiagram.io – не єдиний онлайн інструмент, що надає такий функціонал. Серед безкоштовних конкурентів можна виділити draw.io (раніше Diagrams.net), Lucidchart, QuickDBD, Mermaid та MySQL Workbench, Visual Paradigm, PowerDesigner серед платних. Кожен інструмент має свої особливості, але dbdiagram.io найкраще підійшов для вирішення проблематики теми роботи.

### 1.6. PostgreSQL як інструментальна основа для роботи з базою даних проєкту

Для реалізації бази даних в межах даного проєкту було обрано PostgreSQL – високопродуктивну, функціонально насичену реляційну систему управління базами

даних (СУБД) з відкритим вихідним кодом. Ця система не тільки повністю підтримує, але й значно розширює можливості стандартної мови структурованих запитів SQL, надаючи розробникам потужний та гнучкий інструментарій для надійного зберігання та ефективного масштабування найскладніших робочих навантажень з даними. Історія PostgreSQL сягає корінням 1986 року, коли вона була започаткована як частина дослідницького проєкту POSTGRES у Каліфорнійському університеті в Берклі. З того часу система безперервно та активно розвивається спільнотою розробників вже понад 35 років, здобувши репутацію однієї з найнадійніших та найфункціональніших платформ для роботи з даними у світі. Важливою перевагою PostgreSQL є високий рівень відповідності міжнародним стандартам SQL: на момент написання вона підтримує 170 зі 177 пунктів стандарту SQL:2023, що є одним з найкращих показників серед усіх існуючих реляційних баз даних [11]. Ця відповідність гарантує переносимість коду та використання загальноприйнятих практик.

Розглянемо детальніше ключові можливості цієї СУБД, що роблять її оптимальним вибором для реалізації складних проєктів.

### 1.6.1. Агрегатні функції

Отримання, маніпулювання та управління даними в PostgreSQL здійснюється за допомогою стандартної та надзвичайно гнучкої мови **SQL**. Це дозволяє будувати запити практично будь-якого рівня складності, необхідні для реалізації всієї бізнес-логіки маркетплейсу – від простих вибірок конкретних оголошень до складних аналітичних запитів для статистики ринку. Для ефективної роботи зі об’ємними даними, передбачена підтримка вкладених запитів, широкого набору агрегатних функцій (SUM, AVG, COUNT тощо), умовних виразів (CASE), а також різноманітних операцій з'єднання таблиць (JOIN), групування (GROUP BY), фільтрації груп (HAVING), віконних функцій (WINDOW) та багатьох інших.

Для підвищення читабельності та зручності структурування складних запитів, особливо тих, що містять проміжні результати, PostgreSQL пропонує механізм Common Table Expressions (CTE), який дозволяє визначати тимчасові, іменовані

набори результатів в межах одного запиту за допомогою конструкції WITH. Це робить запити більш модульними та зрозумілими. PostgreSQL також демонструє відмінні можливості роботи з неструктурованими та напівструктурованими даними, зокрема має нативну підтримку роботи з форматом JSON (включаючи оптимізований бінарний формат JSONB), дозволяючи зберігати та ефективно виконувати запити до структурованих JSON-документів за допомогою спеціальних операторів і функцій. Для прискорення виконання об'ємних запитів на багатопроцесорних системах, система здатна виконувати їх паралельно, розподіляючи обчислювальне навантаження між кількома процесорними ядрами.

### 1.6.2. Індекси

Однією з найважливіших складових, що забезпечують високу продуктивність реляційної бази даних, є індекси. **Індекси** у PostgreSQL – це спеціалізовані структури даних, які значно прискорюють виконання операцій вибірки даних, зокрема пошуку, сортування та фільтрації за певними критеріями. Принцип їх роботи полягає у створенні упорядкованих копій значень одного або кількох стовпців (або посилань на відповідні рядки), що дозволяє СУБД швидко знайти потрібні записи без необхідності повного перегляду всієї таблиці (повного сканування таблиці). Це можна порівняти з предметним покажчиком або змістом у кінці книги: замість послідовного перечитування всього тексту, можна швидко знайти потрібне слово, термін чи розділ у покажчику і перейти безпосередньо на вказану сторінку. PostgreSQL підтримує різноманітні типи індексів, кожен з яких оптимізований для певних видів даних, типів запитів та сценаріїв використання. Оптимізатор запитів PostgreSQL є досить "інтелектуальним" і самостійно аналізує запит та доступні індекси, приймаючи рішення про доцільність використання того чи іншого індексу для досягнення максимальної швидкості виконання. Важливо пам'ятати, що використання індексів, хоч і прискорює операції читання даних, дещо уповільнює операції модифікації (вставка, оновлення, видалення), оскільки при цих операціях необхідно також оновлювати відповідні індексні структури.

Серед основних та найбільш поширених типів індексів, що підтримуються у PostgreSQL:

- **B-Tree (Балансоване дерево):** Найбільш універсальний, стандартний та поширений тип індексу за замовчуванням. Ефективний для точного пошуку (=) та операцій порівняння (<, >, <=, >=), а також для сортування даних за індексованим стовпцем.
- **GIN (Generalized Inverted Index)**: Оптимізований для роботи з даними, що містять багато елементів в одному полі, такими як масиви, JSONB-документи та для реалізації повнотекстового пошуку. Дозволяє швидко знайти записи, що містять певний набір елементів (наприклад, пошук оголошень, що містять всі задані ключові слова в описі).
- **GiST (Generalized Search Tree)**: Підтримує індексацію складних структур даних, зокрема багатовимірних даних та геопросторових об'єктів. Ефективний для запитів на пошук у діапазонах (наприклад, пошук об'єктів у певній географічній області або перетин геометричних фігур).
- **BRIN (Block Range Index)**: Розроблений для ефективної індексації дуже великих таблиць, де дані мають природний порядок за індексованим стовпцем (наприклад, часові ряди або дані, вставлені послідовно з наростаючим ID). Індекс зберігає узагальнену інформацію про діапазони значень у блоках даних, а не для кожного окремого рядка, що робить його дуже компактним та швидким для сканування великих лінійних діапазонів.
- **Hash (Хеш-індекс)**: Використовується для дуже швидкого точного порівняння значень (=). Менш універсальний порівняно з B-Tree, оскільки не підтримує операції порівняння за діапазоном чи сортування.

### 1.6.3. Тригери

Тригери у PostgreSQL – це потужний механізм автоматизації, який дозволяє автоматично виконувати певний блок коду (зазвичай у вигляді збереженої функції) у відповідь на визначені події, що відбуваються з даними у таблиці [12]. Типовими подіями, що можуть активувати тригер, є операції модифікації даних: вставка

(INSERT), оновлення (UPDATE) або видалення (DELETE) даних. Тригери широко використовуються для автоматизації обробки даних (наприклад, автоматичне оновлення полів з датою останньої зміни, ведення історії змін записів), забезпечення дотримання складних правил цілісності даних, які важко або неможливо реалізувати за допомогою стандартних обмежень (constraints), та реалізації частини бізнес-логіки безпосередньо на рівні бази даних.

Тригери мають різні часи виконання відносно події, що їх викликала:

- **BEFORE**: Тригери виконуються до виконання основної операції (INSERT, UPDATE, DELETE). Часто використовуються для перевірки коректності даних перед їх записом, модифікації даних або відміни операції.
- **AFTER**: Тригери виконуються після завершення основної операції. Типово застосовуються для запису логів змін (аудит даних), ініціювання подальших дій в інших таблицях або системах.
- **INSTEAD OF**: Спеціальний тип тригерів, що використовуються для перехоплення операцій INSERT, UPDATE або DELETE на поданнях (VIEW), які інакше не підлягають модифікації. Тригер замінює стандартну операцію на виконання свого коду.

Крім того, тригери можуть мати різні рівні виконання:

- **FOR EACH ROW**: Тригер викликається окремо для кожного рядка, який зачіпає операція (наприклад, для кожного вставленого, оновленого або видаленого рядка).
- **FOR EACH STATEMENT**: Тригер викликається лише один раз для всієї операції, незалежно від кількості рядків, на які вона вплинула.

### 1.6.4. Функції

**Функції** у PostgreSQL, також часто звані збереженими процедурами або користувацькими функціями, є програмними блоками, що інкапсулюють певний набір операцій та логіки, які можуть бути визначені та виконані безпосередньо на стороні сервера бази даних. Вони є невід'ємною частиною розширення

функціональності СУБД, дозволяючи автоматизувати рутинні задачі, реалізувати складну бізнес-логіку, інкапсулювати повторюваний код та розширити можливості стандартного SQL. PostgreSQL підтримує написання функцій різними мовами, включаючи чистий SQL (для простих операцій), мову PL/pgSQL (для складнішої логіки з керуючими структурами, циклами, умовами), а також функції, написані на інших мовах програмування (наприклад, Python, Perl, C), що можуть підключатися як розширення. Функції можуть приймати будь-яку кількість вхідних параметрів, але, згідно з концепцією функцій, повертають один результат – це може бути скалярне значення, ціла таблиця, набір рядків, або набір значень різних типів. Однією з ключових переваг функцій є їх виконання на стороні сервера бази даних. Це дозволяє ефективно обробляти та фільтрувати дані без необхідності передачі великих обсягів інформації до клієнтського застосунку через мережу, що суттєво зменшує мережевий трафік, знижує навантаження на клієнтську сторону та підвищує загальну продуктивність системи. Функції широко використовуються для виконання складних обчислень, автоматизації повторюваних завдань, інкапсуляції складної логіки, яку було б важко або неефективно виразити виключно стандартними SQL-запитами, та для забезпечення безпечного доступу до даних через їх виклик.

Узагальнюючи, PostgreSQL пропонує винятково широкий та збалансований набір інструментів, функціональних можливостей та високу гнучкість для роботи з базами даних, що робить її одним із найкращих, якщо не найкращим, вибором для реалізації складних, високопродуктивних, масштабованих та надійних проєктів, таких як інформаційна система маркетплейсу вживаних автомобілів. Серед ключових переваг цієї системи управління базами даних, що стали вирішальними факторами для її вибору в рамках даного проєкту, можна виділити:

- Висока продуктивність та ефективність, що забезпечується сучасним оптимізатором запитів, різноманітними типами індексів та підтримкою паралельних запитів.

- Відмінна масштабованість, включаючи підтримку горизонтального масштабування через реплікацію та шардинг, а також високу стійкість до зростання обробного навантаження.
- Повна підтримка складних SQL-запитів, аналітичних функцій та надійна реалізація транзакцій з гарантуванням властивостей ACID для забезпечення цілісності даних.
- Розширені засоби для забезпечення цілісності даних на рівні схеми, створення та управління складними зв'язками між таблицями.
- Вражаюча гнучкість у роботі з різними типами даних, включаючи нативну, ефективну підтримку напівструктурованих даних у форматі JSON/JSONB та геопросторових даних за допомогою розширення PostGIS.
- Наявність великої, активної та відкритої спільноти розробників, що забезпечує постійний розвиток системи, своєчасні оновлення, виправлення помилок та наявність вичерпної, актуальної документації.
- Відкритий вихідний код, що гарантує прозорість, гнучкість налаштування та відсутність ліцензійних відрахувань, роблячи його економічно вигідним рішенням.
- Наявність потужних інструментів для автоматизації (тригери, функції), що дозволяють перенести частину бізнес-логіки на рівень бази даних, підвищуючи її ефективність.

## 1.7. Підхід Code-First у реалізації бази даних та використання Entity Framework Core

У сучасній розробці додатків, що інтенсивно працюють з даними, **Code-First** є одним з найпопулярніших підходів до роботи з **ORM (Object-Relational Mapping)**. Суть Code-First полягає у визначенні структури бази даних безпосередньо за допомогою написання коду класів сутностей у застосунку, а не через попереднє візуальне моделювання схеми бази даних або генерацію коду на основі існуючої схеми. Прийнявши цей підхід, розробник переносить фокус на доменну модель застосунку. Спершу створюються класи, що відображають бізнес-сутності,

визначаються їхні властивості та взаємозв'язки. Вже після цього ORM використовується для автоматичної генерації або оновлення відповідної схеми бази даних на основі цієї моделі, визначеної в коді.

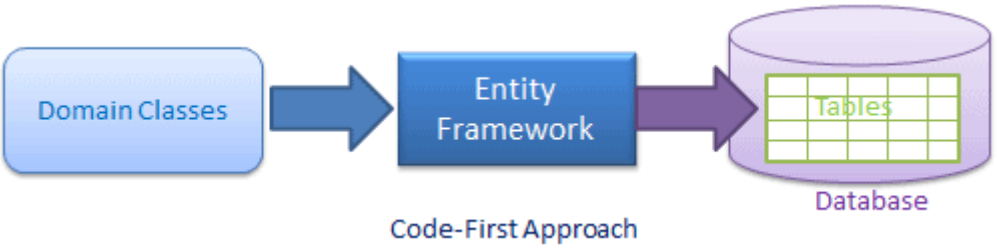


Рис. 1.3. Flow Code-First підходу.

Джерело: [13]

**Entity Framework Core (EF Core)** – це сучасний, високопродуктивний та легкий ORM-фреймворк з відкритим вихідним кодом, який активно розвивається та підтримується корпорацією Microsoft. EF Core є стандартом ORM для розробки додатків на платформі .NET. Він дозволяє розробникам працювати з даними, використовуючи високорівневі об'єкти специфічних для домену класів, що значно абстрагує процес від низькорівневих деталей взаємодії з базою даних, таких як безпосереднє написання SQL-запитів та ручне відображення даних на таблиці та стовпці реляційної бази даних, де ці дані зберігаються [14]. Використання Entity Framework Core суттєво спрощує процес розробки та подальшої підтримки додатків, орієнтованих на дані, дозволяючи реалізувати потрібну функціональність з меншим обсягом коду порівняно з традиційними методами доступу до даних, що базуються на SQL.

Центральною точкою взаємодії між застосунком та базою даних при використанні EF Core є об'єкт контексту бази даних (Database Context). Цей клас, який успадковується від базового класу DbContext, інкапсулює сесію роботи з базою даних та є основним інтерфейсом для виконання запитів, відстеження змін у сутностях та збереження цих змін у базі даних. Контекст містить властивості типу DbSet<TEntity>, які представляють колекції сутностей, що безпосередньо відображають таблиці бази даних. Крім того, саме контекст акумулює конфігурацію підключення до бази даних (Connection string), а також різноманітні налаштування

для окремих сутностей, їхніх властивостей та зв'язків. Налаштування відображення моделі даних (класів сутностей) на схему бази даних в EF Core може здійснюватися двома основними способами:

- Атрибути даних (Data Annotations) – конфігурація виконується шляхом додавання спеціальних .атрибутів безпосередньо до класів сутностей та їхніх властивостей. Ці атрибути надають EF Core метадані про те, як слід відобразити клас або властивість на елементи схеми бази даних.
- Fluent API: Конфігурація здійснюється програмно, шляхом використання "ланцюжка" (чейнингу) методів у перевантаженому методі OnModelCreating класу контексту бази даних. Цей підхід є більш гнучким та надає повніший контроль над процесом відображення, дозволяючи налаштовувати складні зв'язки, індекси, обмеження та інші аспекти схеми, які неможливо виразити за допомогою атрибутів.

Ключовим механізмом, який забезпечує ефективний життєвий цикл розробки та управління схемою бази даних при використанні Code-First підходу з EF Core, є **міграції (Migrations)**. Міграції – це вбудований механізм, що дозволяє автоматично управляти змінами структури бази даних. Вони забезпечують синхронізацію поточної схеми бази даних зі змінами, внесеними в модель даних (класи сутностей) у коді застосунку, без необхідності вручну писати SQL-скрипти для зміни схеми. В Entity Framework Core кожна міграція генерується як окремий файл, який являє собою клас з кодом, що описує покрокові інструкції для перетворення схеми бази даних з однієї версії до іншої. Типовий файл міграції містить два основні методи:

- Метод Up(), що вміщує код (як правило, виклики методів EF Core API, що генерують відповідні SQL-операції), який описує зміни, що мають бути застосовані до бази даних для оновлення її схеми (наприклад, створення нової таблиці, додавання стовпця, зміни типу даних).
- Метод Down(), що містить код, який описує, як скасувати зміни, внесені методом Up(), повертаючи схему бази даних до попереднього стану (наприклад, видалення таблиці, видалення стовпця).

EF Core автоматично генерує нові міграції, порівнюючи поточну модель даних у коді з тією, що була використана для створення попередньої міграції. Інформація про застосовані міграції відстежується системою у спеціальній внутрішній таблиці бази даних з назвою __EFMigrationsHistory. Це дозволяє EF Core знати, які міграції вже були застосовані до конкретної бази даних, та управляти версіями схеми: застосовувати нові міграції для оновлення до актуального стану або відкатувати зміни за потреби, викликаючи відповідні методи **Up()** та **Down()**.

Механізм міграцій надає значні переваги: він дозволяє легко трансформувати структуру бази даних відповідно до еволюції моделі даних у застосунку, надаючи при цьому можливість відкату змін у разі виникнення проблем. Крім того, цей підхід є надзвичайно зручним у командній розробці, оскільки файли міграцій є частиною кодової бази проєкту і можуть синхронізуватися між розробниками через систему контролю версій (наприклад, Git). Це гарантує, що всі учасники команди працюють з узгодженою версією схеми бази даних та можуть легко застосовувати або відміняти зміни, внесені іншими.

Лістинг 1.1 – приклад міграції, згенерованої EF Core

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "Color",
        table: "Vehicles",
        nullable: true);
}

protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.DropColumn(
        name: "Color",
```

```
        table: "Vehicles");
}
```

Джерело: створено автором

## 1.8. Застосування технологій контейнеризації: Docker та Docker Compose

У сучасному ландшафті розробки, доставки та управління програмним забезпеченням, **контейнеризація** виступає як одна з ключових технологій, що докорінно змінює підходи до пакування, розповсюдження та запуску додатків. Контейнеризація – це технологія віртуалізації на рівні операційної системи, яка дозволяє упакувати застосунок разом з усіма його залежностями (бібліотеками, конфігураційними файлами, виконуваними файлами та іншим необхідним оточенням) в стандартизовану, ізольовану та портативну одиницю – **контейнер**.

На відміну від традиційних віртуальних машин (VM), які емулюють цілі операційні системи з власним ядром для кожного екземпляра, контейнери є "легшими", оскільки вони спільно використовують ядро операційної системи хост-машини, ізолюючи лише простір користувача. Це робить контейнери значно більш продуктивними, вимагає менше ресурсів (CPU, пам'ять, дисковий простір) та забезпечує набагато вищу швидкість запуску порівняно з VM. Контейнеризація є сучасним та ефективним підходом до розгортання програмного забезпечення, що гарантує високий рівень ізоляції середовища виконання для кожного застосунку, чудову портативність між різними інфраструктурними середовищами та максимально ефективне використання ресурсів обчислювальної інфраструктури [15]. Ця технологія набуває особливого, критично важливого значення при розробці, тестуванні та експлуатації складних, розподілених інформаційних систем, таких як багатокомпонентні онлайн-маркетплейси, де різні частини системи (веб-сервер, база даних тощо) потребують власного ізольованого середовища. Серед ключових переваг, які надає контейнеризація:

- Контейнер, що містить всі необхідні для роботи застосунку залежності, може бути зібраний один раз і без змін розгорнутий на будь-якому сервері, локальній машині розробника, тестовому середовищі чи в будь-якій публічній або приватній хмарній платформі, що підтримує технології контейнеризації. Це забезпечує консистентність середовищ та усуває проблеми, пов'язані з несумісністю інфраструктури.

- Кожен контейнер працює у власному ізольованому середовищі, відокремленому як від інших контейнерів, що працюють на тому ж хості, так і від самої хост-системи. Це забезпечує високий рівень конфіденційності та безпеки процесів, файлової системи, мережевих портів та системних ресурсів для кожного застосунку, запобігаючи конфліктам залежностей між різними додатками.
- Оскільки контейнерам не потрібно завантажувати повну гостьову операційну систему (як це роблять віртуальні машини), вони запускаються практично миттєво (за секунди), що суттєво прискорює цикли розробки, тестування, процеси автоматичного масштабування та розгортання додатків у продуктивному середовищі.
- Контейнери використовують ресурси обчислювальної інфраструктури (процесорний час, оперативну пам'ять, дисковий простір) значно ефективніше, ніж віртуальні машини. Це досягається завдяки спільному використанню ядра операційної системи хоста та відсутності потреби у виділенні ресурсів для окремої операційної системи для кожного віртуалізованого екземпляра.

**Docker** є провідною платформою для роботи з контейнерами, яка революціонізувала процеси розробки, доставки (shipping) та запуску додатків. Docker надає можливість ефективно відокремити додатки від базової інфраструктури, на якій вони виконуються, стандартизуючи процес пакування та виконання. Це дозволяє командам розробки та експлуатації (DevOps) швидше, надійніше та з більшою впевненістю доставляти програмне забезпечення від моменту написання коду до його запуску в продуктивному середовищі. Використання переваг методологій, що пропонуються Docker для пакування, доставки, тестування та розгортання коду, дозволяє значно скоротити так звану "затримку" (lead time) між написанням коду розробником та його успішним функціонуванням у продакшені.

Docker надає стандартизований та інтуїтивно зрозумілий спосіб пакувати та запускати будь-який застосунок у вільно ізольованому середовищі, яке ми називаємо контейнером. Ізоляція, що забезпечується контейнерами Docker, у поєднанні з

вбудованими механізмами безпеки, дозволяє безпечно та ефективно розгортати та запускати багато контейнерів, які можуть містити різні застосунки з різними залежностями, одночасно на одному хост-сервері, не боячись конфліктів. Контейнери Docker є легкими, містять все необхідне для запуску конкретного застосунку (бібліотеки, конфігураційні файли тощо), тому вам не потрібно покладатися на те, яке специфічне програмне забезпечення чи бібліотеки встановлені на хост-системі. Однією з потужних переваг є можливість легко ділитися образами контейнерів з колегами чи іншими командами через реєстри образів (наприклад, Docker Hub), будучи абсолютно впевненими, що кожен, хто запустить контейнер з цього образу, отримає і працюватиме в абсолютно ідентичному, відтворюваному середовищі.

Архітектурно Docker використовує модель **клієнт-сервер**. Клієнт Docker (інструмент командного рядка docker, який використовують розробники та адміністратори) взаємодіє з Docker daemon (фоновим процесом, що працює на хост-системі), який і виконує основну, "важку" роботу зі створення, запуску, зупинки, моніторингу та розповсюдження контейнерів та образів. Клієнт і Docker daemon можуть працювати на одній машині (локально), або клієнт може бути налаштований на підключення до віддаленого Docker daemon через мережу. Взаємодія між клієнтом та daemon відбувається через REST API, що може передаватися через сокети UNIX або мережевий інтерфейс [16].

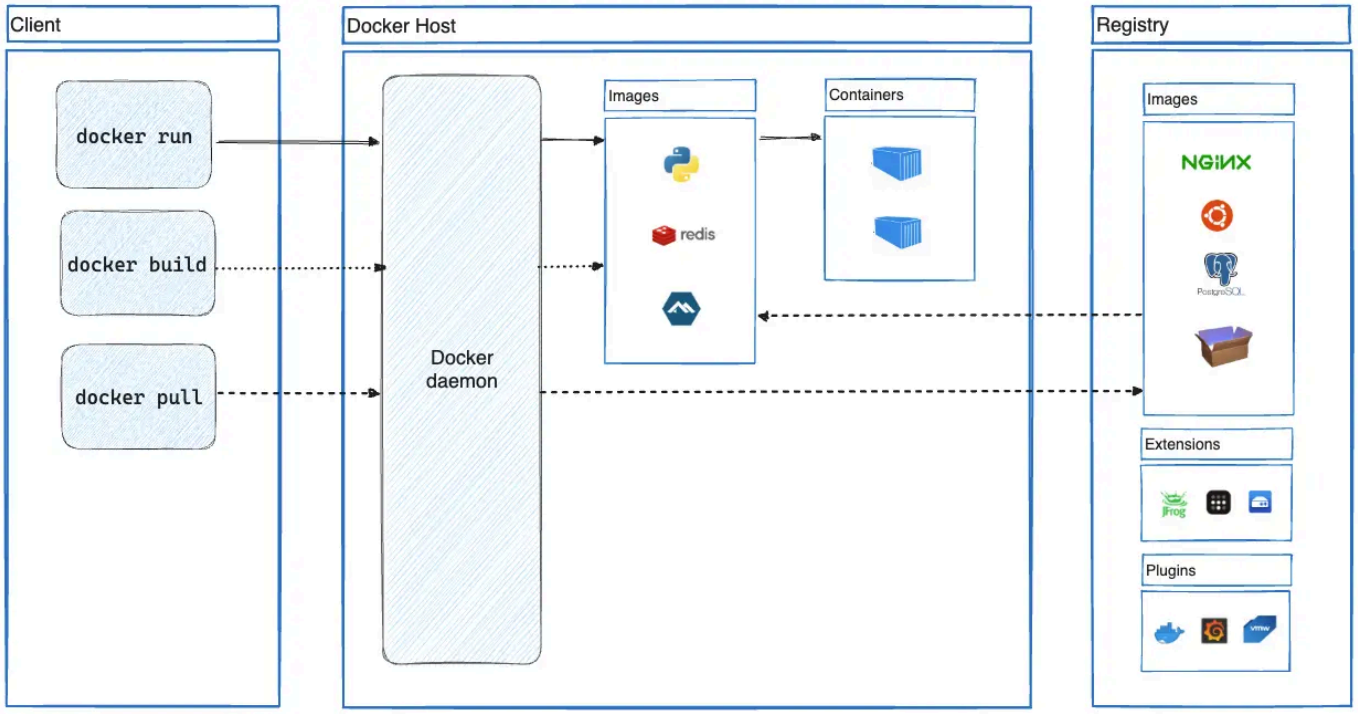


Рис. 1.4 – схема роботи архітектури Docker.

Джерело: [16]

Ще одним важливим механізмом Docker, який набуває особливого значення при роботі з багатокомпонентними додатками, є **Docker Compose**. Docker Compose – це спеціалізований інструмент, розроблений для спрощення процесу визначення, конфігурування та управління життєвим циклом багатоконтейнерних додатків. Він є ключовим інструментом для спрощення та підвищення ефективності процесів розробки, тестування та розгортання застосунків, які складаються з кількох взаємозалежних сервісів, що працюють у окремих контейнерах. Compose дозволяє централізовано описувати всю архітектуру багатокомпонентного додатку, включаючи всі сервіси (контейнери, з яких складається додаток), їхні образи, залежності між ними, мережі, які вони використовують для взаємодії, порти, що публікуються, та томи (Volumes) для зберігання персистентних даних. Вся ця конфігурація описується в одному зрозумілому файлі у форматі YAML [17]. Після визначення конфігурації у файлі docker-compose.yml, за допомогою однієї простої команди (docker compose up), можна автоматично створити, налаштувати та запустити всі сервіси, описані у конфігураційному файлі, у правильному порядку, з необхідними мережевими налаштуваннями та прив'язаними томами.

Docker Compose також надає зручні можливості для управління даними контейнерів, дозволяючи зберігати важливі дані (наприклад, дані бази даних, файли користувачів) у зовнішній файловій системі (за допомогою Volumes). Це робить ці дані незалежними від життєвого циклу самого контейнера: навіть якщо контейнер буде видалено або перестворено (наприклад, при оновленні образу), дані, збережені у томі, залишаться цілими та неушкодженими. Використання Docker та Docker Compose є стандартом сучасної розробки, що суттєво спрощує розгортання, масштабування та управління складними багатокомпонентними системами в будь-якому середовищі.

Лістинг 1.2 – структура docker-compose.yml

```
services:
  app:
    build: .
    ports:
      - "5000:5000"
    depends_on:
      - db
  db:
    image: postgres:latest
    environment:
      POSTGRES_USER: user
      POSTGRES_PASSWORD: password
      POSTGRES_DB: marketplace
    volumes:
      - db_data:/var/lib/postgresql/data
volumes:
  db_data:
```

Джерело: згенеровано автором

### 1.9. Основи архітектури веб-серверного застосунку

**Архітектура програмного забезпечення** являє собою високорівневий план організації системи, що визначає її основні структурні компоненти, їхні

взаємозв'язки, принципи взаємодії та обмеження. По суті, це каркас, що визначає, як має бути побудована система для досягнення поставлених цілей та забезпечення необхідних якісних характеристик. Цей план слугує фундаментальним орієнтиром для команди розробки, встановлюючи загальні керівні принципи та обмеження на етапі реалізації. Однак, слід розуміти, що ступінь суворого дотримання початково визначених архітектурних рішень може варіюватися залежно від специфіки проєкту, його динаміки та зовнішніх обставин. Під впливом різноманітних чинників – таких як зміна бізнес-вимог, накопичення технічного боргу, обмеження ресурсів, необхідність швидкого виходу на ринок чи адаптація до нових технологій – початково спроектована архітектура може з часом зазнавати модифікацій, "деформуватися" або еволюціонувати, проходячи певні "метаморфози".

Розробка продуманої архітектури є критично важливою для управління зростаючою складністю великих програмних систем, забезпечення необхідних якісних атрибутів (таких як масштабованість, надійність, продуктивність, безпека, зручність підтримки, тестування та подальшого розвитку) та ефективної координації роботи команди розробників над спільним проєктом.

У контексті веб-застосунків, зокрема тих, що функціонують як бекенд інформаційної системи маркетплейсу, архітектура має відповідати специфічним викликам та вимогам. Вона повинна бути спроможна ефективно обробляти численні одночасні вхідні запити від клієнтських додатків (веб-браузерів, мобільних застосунків тощо), взаємодіяти з різноманітними джерелами даних (як-от база даних, зовнішні API), виконувати складну бізнес-логіку та забезпечувати високу доступність і масштабованість для підтримки зростаючої кількості користувачів, обсягів даних та операцій. Розробка продуманої архітектури є фундаментальною для створення надійного, високопродуктивного та масштабованого веб-сервера маркетплейсу вживаних автомобілів. Вибір відповідного архітектурного стилю та застосування певних архітектурних шаблонів відіграє ключову роль у здатності системи ефективно відповідати цим вимогам.

Одним із сучасних та широко рекомендованих підходів до побудови архітектури програмних застосунків, спрямованих на створення гнучких,

легкотестованих та зручних у підтримці систем, є **"Чиста архітектура" (Clean Architecture)**. Цей архітектурний стиль, популяризований відомим фахівцем у галузі розробки програмного забезпечення Робертом Мартіном (Uncle Bob), ґрунтується на принципі суворого розділення відповідальностей та встановленні чіткого правила залежностей: залежності у вихідному коді завжди мають бути спрямовані всередину, до більш центральних шарів абстракції, а не назовні. Чиста архітектура передбачає організацію коду в концентричні шари, де кожен зовнішній шар залежить від внутрішнього, але внутрішні шари не знають про існування зовнішніх. Найбільш внутрішні шари містять критично важливу бізнес-логіку та сутності домену, тоді як зовнішні шари (наприклад, інтерфейс користувача, репозиторії, шлюзи до зовнішніх сервісів) є деталями реалізації. Така структура забезпечує незалежність доменної логіки від конкретних технологій та інфраструктурних рішень, що суттєво полегшує автоматизоване тестування та дозволяє легко заміняти зовнішні залежності без впливу на ядро системи.

Ще одним важливим архітектурним шаблоном, який часто використовується для оптимізації роботи з даними та підвищення масштабованості у складних системах, є **CQRS (Command Query Responsibility Segregation)** – принцип розділення відповідальності між операціями читання (Queries) та операціями запису/модифікації (Commands) [26]. Замість використання єдиної уніфікованої моделі даних або набору сервісів для виконання як операцій отримання даних, так і операцій зміни стану системи, CQRS пропонує розділити модель та/або реалізацію на дві окремі частини: одна відповідає виключно за виконання команд (операції, що змінюють стан системи, наприклад, створення оголошення, оновлення даних користувача), а інша – за виконання запитів (операції, що лише отримують дані, наприклад, пошук автомобілів, перегляд деталей оголошення). Цей підхід дозволяє незалежно оптимізувати кожну частину: модель для читання може бути спрощена, денормалізована та оптимізована спеціально для швидкого виконання запитів та масштабування (наприклад, використання окремої бази даних або кешу), тоді як модель для запису може бути більш строго нормалізована та оптимізована для гарантування цілісності даних та коректної обробки бізнес-правил при їх

модифікації. CQRS часто застосовується в поєднанні з іншими архітектурними стилями, включаючи чисту архітектуру, для підвищення масштабованості, продуктивності та гнучкості систем, особливо тих, де спостерігається значно вищий обсяг операцій читання порівняно з операціями запису.

### 1.10. Форсування архітектури

Після визначення та детального документування архітектури веб-застосунку, що включає вибір архітектурного стилю та застосування певних шаблонів, критично важливим етапом стає забезпечення її дотримання протягом усього життєвого циклу розробки проєкту. **Форсування архітектури** – це комплекс практик та набір інструментів, що дозволяють автоматизовано або напів автоматизовано контролювати відповідність реалізованого коду спроєктованій структурі системи, встановленим архітектурним принципам та правилам взаємодії між компонентами [18]. Це життєво необхідно для запобігання відходу від архітектурного стилю, накопиченню технічного боргу, що виникає внаслідок відхилень від початкового задуму, та збереження необхідних якісних атрибутів системи (таких як зручність підтримки, масштабованість, тестування) з часом.

Існує кілька рівнів та підходів до реалізації форсування архітектури. Базовий рівень може бути впроваджений за допомогою інструментів статичного аналізу коду та вбудованих функцій середовищ розробки (IDE). Сучасні IDE, такі як Visual Studio або Rider, надають можливості для конфігурації правил стилю коду та базових структурних обмежень через файли конфігурації, наприклад, **.editorconfig**. Ці файли дозволяють налаштувати та автоматично перевіряти дотримання стандартів кодування, правил іменування змінних, класів, методів, форматування коду, а також деяких простих правил організації кодової бази. Хоча ці інструменти переважно стосуються стилю коду та синтаксичних правил, вони є важливим першим кроком до підтримки загальної консистентності та порядку в кодовій базі. Більш просунуті інструменти статичного аналізу (наприклад, Roslyn Analyzers для .NET) можуть перевіряти складніші патерни використання коду та деякі базові залежності, але їхні

можливості щодо перевірки високорівневих архітектурних правил, що стосуються залежностей між шарами чи модулями, є обмеженими.

Для забезпечення дотримання саме вищих рівнів архітектурних принципів та структурних правил, таких як правила залежностей між шарами в "Чистій архітектурі", відсутність циклічних залежностей між модулями, або коректне розділення команд і запитів у CQRS, використовуються архітектурні тести.

**Архітектурні тести** – це спеціалізований тип автоматизованих тестів, що сфокусовані на перевірці відповідності реалізованої структури коду, його дизайну та взаємозв'язків між компонентами задуманій архітектурі системи. Їх основна та найбільш вагома мета – це гарантувати, що ключові архітектурні принципи, зокрема правила залежностей між різними модулями, шарами чи компонентами застосунку, а також принципи розподілу обов'язків, послідовно дотримуються всіма членами команди розробки протягом усього процесу роботи над проєктом [18].

Архітектурні тести дозволяють формалізувати та автоматизувати перевірку правил, які важко або практично неможливо проконтролювати за допомогою традиційних юніт- чи інтеграційних тестів. Наприклад, в контексті "Чистої архітектури", архітектурні тести можуть перевіряти, чи не з'являються небажані залежності – наприклад, чи не має шар доменної логіки (Domain) залежностей від шару інфраструктури (Infrastructure), чи не звертається шар представлення (Presentation) безпосередньо до шару доступу до даних (Data Access), обходячи проміжні шари бізнес-логіки чи застосування. Вони дозволяють визначити та автоматично перевірити правила щодо розміщення класів у певних просторах імен, дозволених залежностей між окремими проєктами (збірками) у рішенні Visual Studio, обмеження на використання певних зовнішніх бібліотек лише в дозволених шарах, або правила щодо іменування певних типів класів (наприклад, класи, що реалізують команди CQRS, мають закінчуватися на "Command" або "CommandHandler").

Реалізація архітектурних тестів зазвичай здійснюється за допомогою спеціалізованих бібліотек або фреймворків, які надають API для статичного аналізу структури коду та його залежностей (наприклад, бібліотека NetArchTest є

популярним вибором для проєктів на платформі .NET). Ці інструменти дозволяють написати декларативні тести, які чітко описують бажані архітектурні обмеження та правила у вигляді коду тестів.

Використання архітектурних тестів приносить значні переваги:

- **Раннє виявлення помилок**: Вони дозволяють автоматично виявляти порушення архітектурних правил практично відразу після внесення змін до коду, що є набагато ефективнішим, ніж виявлення цих проблем на пізніх етапах розробки або під час ручного рефакторингу. Інтеграція архітектурних тестів у процес безперервної інтеграції (CI) дозволяє перехоплювати порушення на етапі збірки.
- **Запобігання накопиченню технічного боргу**: Систематичне застосування архітектурних тестів допомагає команді підтримувати структуру коду в належному стані, запобігаючи "розповзанню" залежностей та накопиченню технічного боргу, пов'язаного зі структурними проблемами.
- **Жива документація**: Архітектурні тести слугують своєрідною "живою" та завжди актуальною документацією архітектурних правил, оскільки вони чітко визначають, що дозволено, а що заборонено на рівні структури коду та залежностей.
- **Підвищення впевненості**: Наявність пройдених архітектурних тестів дає команді впевненість у тому, що система розвивається відповідно до початкового архітектурного задуму, що є критично важливим для підтримки масштабованості, зручності підтримки та високої якості системи в довгостроковій перспективі.

## 1.11. CI/CD

CI/CD розшифровується як Continuous Integration (безперервна інтеграція) та Continuous Deployment (або Continuous Delivery) (безперервне розгортання або безперервна доставка). Це набір практик та інструментів, призначених для покращення процесу розробки програмного забезпечення шляхом автоматизації

збірки, тестування та розгортання, що дозволяє вам швидше та надійніше відправляти зміни коду.

- **Безперервна інтеграція (CI):** Автоматично збирає, тестує та інтегрує зміни коду в спільному репозиторії
- **Безперервна доставка (CD):** автоматично доставляє зміни коду в готові до виробництва середовища для затвердження
- **Безперервне розгортання (CD):** автоматично розгортає зміни коду безпосередньо для клієнтів

CI/CD складається з безперервної інтеграції та безперервної доставки або безперервного розгортання. Разом вони утворюють «конвеєр CI/CD» – серію автоматизованих робочих процесів, які допомагають командам DevOps скоротити кількість ручних завдань.

Приклад CI/CD пайплайну можна побачити на рис. 1.5.

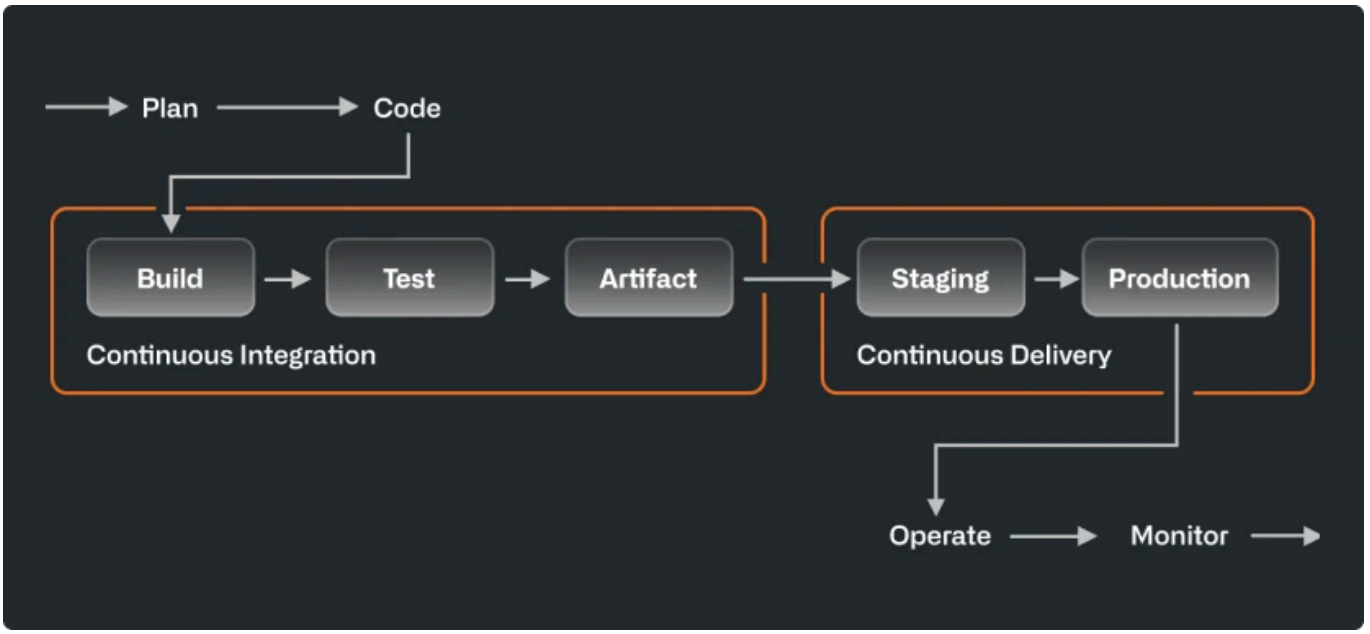


Рис. 1.5. Приклад CI/CD пайплайну

Джерело: [23]

Коли хтось каже CI/CD, «CD», яке вони мають на увазі, зазвичай означає безперервну доставку, а не безперервне розгортання. У чому різниця? У конвеєрі CI/CD, який використовує безперервну доставку, автоматизація припиняється, коли

розробники переходять до виробництва. Командам з операцій, безпеки або відповідності – все одно потрібно вручну підтверджувати дію перед фінальним випуском, що додає ще більше затримок. З іншого боку, безперервне розгортання автоматизує весь процес випуску. Зміни коду розгортаються для клієнтів, як тільки вони проходять всі необхідні тести.

Безперервне розгортання є найкращим прикладом автоматизації DevOps. Це не означає, що це єдиний спосіб виконання CI/CD або «правильний» спосіб. Оскільки безперервне розгортання спирається на суворі інструменти тестування і зрілу культуру тестування, більшість команд розробників програмного забезпечення починають з безперервного розгортання і з часом інтегрують більш автоматизоване тестування. CI/CD надає наступні переваги:

- **Швидкість розробки**: Постійний зворотній зв'язок дозволяє розробникам вносити невеликі зміни частіше, а не чекати на один реліз.
- **Стабільність та надійність**: Автоматизоване безперервне тестування гарантує, що кодові бази залишаються стабільними і готовими до випуску в будь-який час.
- **Зростання бізнесу**: Звільнившись від ручної роботи, організації можуть зосередити ресурси на інноваціях, задоволенні потреб клієнтів та погашенні технічної заборгованості.

Це все робить CI/CD незамінним інструментом у роботі з великими проєктами [23].

**Висновки до розділу 1**

У першому розділі кваліфікаційної роботи було закладено теоретичний фундамент, необхідний для всебічного проєктування бази даних та архітектури серверної частини маркетплейсу вживаних автомобілів. Проаналізовано ключові принципи та сучасні підходи, що забезпечують розробку надійної, масштабованої та легко підтримуваної програмної системи.

Зокрема, обґрунтовано фундаментальне значення ретельного проєктування реляційної бази даних як основи для ефективного зберігання, управління та

швидкого доступу до даних про транспортні засоби, користувачів та операції маркетплейсу. Висвітлено роль ER-моделювання як інструменту візуалізації та структурування даних, а також переваги використання сучасних ORM-рішень, таких як Entity Framework Core, що дозволяють реалізувати підхід Code-First та спрощують взаємодію з базою даних.

Детально розглянуто теоретичні аспекти архітектури серверної частини застосунку, підкреслено важливість застосування принципів чіткого розділення відповідальностей, шаруватості та модульності. Це забезпечує гнучкість, розширюваність та полегшує супровід кодової бази. Визначено, що вибір архітектурного стилю має вирішальне значення для створення стійкої та адаптивної системи.

Крім того, було висвітлено теоретичні основи та обґрунтовано необхідність інтеграції сучасних інструментів та практик, таких як архітектурні тести для забезпечення структурної цілісності коду, Health Checks для моніторингу стану сервісів та їхніх залежностей, а також систем телеметрії для збору метрик продуктивності. Ці компоненти є критично важливими для створення високоякісного, керованого та відмовостійкого застосунку.

Таким чином, проведений теоретичний аналіз у даному розділі сформував міцну методологічну базу, яка дозволить підійти до практичної реалізації маркетплейсу з урахуванням найкращих практик у проєктуванні баз даних та архітектури програмного забезпечення, забезпечуючи його функціональність, надійність та готовність до подальшого розвитку.

# РОЗДІЛ 2

# ПРИКЛАДНА ЧАСТИНА ПРОЄКТУВАННЯ БАЗИ ДАНИХ ТА АРХІТЕКТУРИ СЕРВЕРНОЇ ЧАСТИНИ МАРКЕТПЛЕЙСУ ВЖИВАНИХ АВТОМОБІЛІВ

## 2.1. Проєктування бази даних

Першим етапом проектування бази даних є визначення її логічної структури та вмісту. Це передбачає ідентифікацію ключових сутностей, що відображатимуть бізнес-логіку маркетплейсу, визначення їхніх атрибутів, а також встановлення зв'язків між цими сутностями. Розуміння того, які дані зберігатимуться, у якому вигляді та як вони взаємодіятимуть, є фундаментом для побудови ефективної бази даних.

Для покращення організації та полегшення навігації в базі даних, сутності логічно згруповано за схемами: **"identity"**, **"locations"** та **"vehicles"**.

**Схема "identity"**: Включає сутності, пов'язані з управлінням користувачами, їхніми ролями, правами доступу та персональними списками побажань.

- **"Permissions" (Дозволи)**
    - Містить перелік можливих прав доступу або дозволів, що можуть бути призначені користувачам через ролі.
    - Основні атрибути:
        - Code (PK): Унікальний код дозволу (рядок), що слугує первинним ключем.
- **"UserRoles" (Ролі користувачів)**
    - Зберігає інформацію про різні ролі користувачів у системі (наприклад, "Admin", "Seller", "Buyer").
    - Основні атрибути:
        - Name (PK): Унікальна назва ролі (рядок), що є первинним ключем.
- **"RolePermissions" (Зв'язок Ролей та Дозволів)**

    - Ця проміжна таблиця реалізує зв'язок "багато-до-багатьох" між ролями (UserRoles) та дозволами (Permissions), визначаючи, які саме дозволи надаються кожній ролі.
    - Основні атрибути:
        - UserRoleName (FK): Назва ролі (зовнішній ключ до UserRoles).
        - PermissionCode (FK): Код дозволу (зовнішній ключ до Permissions).
        - (Примітка: Комбінація UserRoleName та PermissionCode формує складений первинний ключ таблиці зв'язку).
- **"Users" (Користувачі)**
    - Основна таблиця, що зберігає облікові дані та базову інформацію про зареєстрованих користувачів маркетплейсу.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор користувача (UUID).
        - UserRoleName (FK): Назва ролі, до якої належить даний користувач (зовнішній ключ до UserRoles).
        - WishlistId (FK): Ідентифікатор списку побажань користувача
- **"UserWishlists" (Списки побажань користувачів)**
    - Зберігає інформацію про персональні списки побажань користувачів, куди вони можуть додавати цікаві автомобілі.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор списку побажань (UUID).
        - UserId (FK): Ідентифікатор користувача, якому належить даний список побажань (UUID, зовнішній ключ до Users). (Примітка: Цей атрибут має обмеження унікальності, що реалізує зв'язок один-до-одного між користувачем та його списком побажань).

**Схема "locations"**: Містить сутності, пов'язані з географічним розташуванням транспортних засобів.

- **"VehicleLocationRegions" (Регіони розташування)**
    - Описує регіони, у яких розташовані автомобілі.

    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва регіону.
- **"VehicleLocationTownTypes" (Типи населених пунктів)**
    - Визначає типи населених пунктів (місто, село).
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва типу населеного пункту.
- **"VehicleLocationAreas" (Області розташування)**
    - Описує області у межах регіонів.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва області.
        - LocationRegionId (FK): Зв’язок із таблицею VehicleLocationRegions.
- **"VehicleLocationTowns" (Населені пункти)**
    - Зберігає дані про міста та села.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва населеного пункту.
        - LocationAreaId (FK): Зв’язок із таблицею VehicleLocationAreas.
        - LocationTownTypeId (FK): Зв’язок із таблицею VehicleLocationTownTypes.
        - LocationRegionId (FK): Зв’язок із таблицею VehicleLocationRegions.

**Схема "vehicles"**: Охоплює сутності, що описують самі транспортні засоби, їх характеристики, моделі, зображення та ціни.

- **"VehicleBodyTypes" (Типи кузовів)**

    - Містить перелік типів кузовів автомобілів (наприклад, седан, хетчбек, універсал).
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва типу кузова.
- **"VehicleBrands" (Марки автомобілів)**
    - Зберігає список брендів автомобілів (наприклад, Toyota, BMW).
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва бренда.
- **"VehicleColors" (Кольори автомобілів)**
    - Зберігає кольори, доступні для автомобілів.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - HexValue: Hex значення кольору.
        - Name: Назва кольору.
- **"VehicleDrivetrainTypes" (Типи приводу)**
    - Визначає типи приводу (передній, задній, повний).
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва типу приводу.
- **"VehicleEngineTypes" (Типи двигунів)**
    - Містить інформацію про типи двигунів (бензиновий, дизельний, електричний).
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва типу двигуна.
- **"VehicleTransmissionTypes" (Типи трансмісій)**
    - Описує типи трансмісій (механічна, автоматична, варіатор).

    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва трансмісії.
- **"VehicleTypes" (Типи транспортних засобів)**
    - Перелік типів транспортних засобів (легковий, вантажний, мотоцикл).
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва типу.
- **"VehicleModels" (Моделі автомобілів)**
    - Зберігає моделі автомобілів, що прив'язані до брендів.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Name: Назва моделі.
        - MinimalProductionYear: Рік початку випуску моделі
        - MaximumProductionYear: Рік закінчення випуску моделі
        - VehicleBrandId (FK): Зв’язок із таблицею VehicleBrands.
        - VehicleTypeId (FK): Зв’язок із таблицею VehicleTypes.
- **"Vehicles" (Транспортні засоби)**
    - Основна таблиця, що зберігає дані про транспортні засоби.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - Description: Опис автомобіля.
        - Vincode: VIN-код автомобіля.
        - Displacement: Об’єм двигуна.
        - Mileage: Пробіг автомобіля.
        - ProductionYear: Рік випуску автомобіля.
        - CreatedAt: Дата та час створення оголошення.
        - LocationTownId (FK): Зв’язок з таблицею VehicleLocationTowns.
        - BrandId (FK): Зв’язок з таблицею VehicleBrands.

        - ModelId (FK): Зв’язок із таблицею VehicleModels.
        - BodyTypeId (FK): Зв’язок із таблицею VehicleBodyTypes.
        - ColorId (FK): Зв’язок із таблицею VehicleColors.
        - DrivetrainTypeId (FK): Зв’язок із таблицею VehicleDrivetrainTypes.
        - EngineTypeId (FK): Зв’язок із таблицею VehicleEngineTypes.
        - TransmissionTypeId (FK): Зв’язок із таблицею VehicleTransmissionTypes.
        - VehicleTypeId (FK): Зв’язок із таблицею VehicleTypes.
        - OwnerId (FK): Зв'язок із таблицею identity.Users, вказує користувача, який є власником або розмістив дане оголошення.
- **"VehicleImages" (Зображення транспортних засобів)**
    - Містить посилання на зображення автомобілів.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - VehicleId (FK): Зв’язок із таблицею Vehicles.
        - CloudImageId: Ідентифікатор зображення в хмарному сервісі.
        - ImageUrl: Посилання на зображення.
        - IsMainImage: Вказівник на те, чи є зображення головним в оголошенні.
- **"VehiclePrices" (Ціни транспортних засобів)**
    - Зберігає інформацію про ціни автомобілів.
    - Основні атрибути:
        - Id (PK): Унікальний ідентифікатор.
        - VehicleId (FK): Зв’язок із таблицею Vehicles.
        - Value: Ціна автомобіля.
        - IssueDate: Дата та час, коли ціна була вказана.

Зв'язки між сутностями визначають, як дані в різних таблицях логічно пов'язані між собою, встановлюючи відношення "один-до-одного" (1:1), "один-до-багатьох" (1:N) або "багато-до-багатьох" (M:N). На основі аналізу

DBML-коду та визначених первинних і зовнішніх ключів, у базі даних встановлено наступні типи зв'язків:

**Зв'язки в схемі "identity":**

- **Один-до-багатьох (1:N)**:
  - UserRoles → Users: Кожна роль може належати багатьом користувачам, але кожен користувач має лише одну роль. Зв'язок реалізовано через зовнішній ключ UserRoleName у таблиці Users, що посилається на Name у таблиці UserRoles.
  - UserRoles → RolePermissions: Кожна роль може мати багато призначених дозволів через проміжну таблицю RolePermissions. Зв'язок реалізовано через зовнішній ключ UserRoleName у таблиці RolePermissions, що посилається на Name у таблиці UserRoles.
  - Permissions → RolePermissions: Кожен дозвіл може бути призначений багатьом ролям через проміжну таблицю RolePermissions. Зв'язок реалізовано через зовнішній ключ PermissionCode у таблиці RolePermissions, що посилається на Code у таблиці Permissions.
- **Один-до-одного (1:1)**:
  - Users → UserWishlists: Кожен користувач має один унікальний список побажань, і кожен список побажань належить одному користувачу. Зв'язок реалізовано через зовнішній ключ UserId у таблиці

UserWishlists, що посилається на Id у таблиці Users і має обмеження унікальності.

**Зв'язки в схемі "locations":**

- **Один-до-багатьох (1:N)**:
  - VehicleLocationRegions → VehicleLocationAreas: Кожен регіон може включати багато областей, але кожна область належить лише до одного регіону. Зв'язок реалізовано через зовнішній ключ LocationRegionId у таблиці VehicleLocationAreas.
  - VehicleLocationAreas → VehicleLocationTowns: Кожна область може включати багато населених пунктів, але кожен населений пункт належить лише до однієї області. Зв'язок реалізовано через зовнішній ключ LocationAreaId у таблиці VehicleLocationTowns.
  - VehicleLocationTownTypes → VehicleLocationTowns: Кожен тип населеного пункту може використовуватись для багатьох населених пунктів, але кожен населений пункт має лише один тип. Зв'язок реалізовано через зовнішній ключ LocationTownTypeId у таблиці VehicleLocationTowns.
  - VehicleLocationRegions → VehicleLocationTowns: Кожен регіон може включати багато населених пунктів, але кожен населений пункт має прямий зв'язок зі своїм регіоном. Зв'язок реалізовано через

зовнішній ключ LocationRegionId у таблиці VehicleLocationTowns.

**Зв'язки в схемі "vehicles" (включаючи зв'язки з іншими схемами):**

- Один-до-багатьох (1:N):
  - VehicleBrands → VehicleModels: Кожна марка може мати багато моделей, але кожна модель належить лише до однієї марки. Зв'язок реалізовано через зовнішній ключ VehicleBrandId у таблиці VehicleModels.
  - VehicleTypes → VehicleModels: Кожен загальний тип транспортного засобу може мати багато моделей, але кожна модель належить лише до одного типу. Зв'язок реалізовано через зовнішній ключ VehicleTypeId у таблиці VehicleModels.
  - VehicleBodyTypes → Vehicles: Кожен тип кузова може використовуватись багатьма автомобілями, але кожен автомобіль має лише один тип кузова. Зв'язок реалізовано через зовнішній ключ BodyTypeId у таблиці Vehicles.
  - VehicleColors → Vehicles: Кожен колір може використовуватись багатьма автомобілями, але кожен автомобіль має лише один колір. Зв'язок реалізовано через зовнішній ключ ColorId у таблиці Vehicles.

- VehicleDrivetrainTypes → Vehicles: Кожен тип приводу може використовуватись багатьма автомобілями, але кожен автомобіль має лише один тип приводу. Зв'язок реалізовано через зовнішній ключ DrivetrainTypeId у таблиці Vehicles.
- VehicleEngineTypes → Vehicles: Кожен тип двигуна може використовуватись багатьма автомобілями, але кожен автомобіль має лише один тип двигуна. Зв'язок реалізовано через зовнішній ключ EngineTypeId у таблиці Vehicles.
- VehicleTransmissionTypes → Vehicles: Кожен тип трансмісії може використовуватись багатьма автомобілями, але кожен автомобіль має лише один тип трансмісії. Зв'язок реалізовано через зовнішній ключ TransmissionTypeId у таблиці Vehicles.
- VehicleTypes → Vehicles: Кожен загальний тип транспортного засобу може використовуватись багатьма автомобілями, але кожен автомобіль має лише один загальний тип. Зв'язок реалізовано через зовнішній ключ VehicleTypeId у таблиці Vehicles.
- Vehicles → VehicleImages: Кожен автомобіль може мати багато зображень, але кожне зображення належить лише до одного автомобіля. Зв'язок реалізовано через зовнішній ключ VehicleId у таблиці VehicleImages.

- Vehicles → VehiclePrices: Кожен автомобіль може мати багато записів історії цін, але кожен запис ціни пов'язаний лише з одним автомобілем. Зв'язок реалізовано через зовнішній ключ VehicleId у таблиці VehiclePrices.
- locations.VehicleLocationTowns → vehicles.Vehicles: Кожен населений пункт може мати багато автомобілів, розташованих у ньому, але кожен автомобіль розташований лише в одному населеному пункті. Зв'язок реалізовано через зовнішній ключ LocationTownId у таблиці Vehicles.
- identity.Users → vehicles.Vehicles: Кожен користувач може бути власником багатьох автомобілів (розмістити багато оголошень), але кожен автомобіль має лише одного власника в контексті оголошення. Зв'язок реалізовано через зовнішній ключ OwnerId у таблиці Vehicles.

- **Багато-до-багатьох (M:N)**:
  - VehicleBodyTypes ↔ VehicleModels: Одна модель автомобіля може мати кілька доступних типів кузовів, і один тип кузова може бути доступним для кількох моделей. Зв'язок реалізовано через проміжну таблицю vehicles.VehicleBodyTypesVehicleModels.
  - VehicleDrivetrainTypes ↔ VehicleModels: Одна модель автомобіля може підтримувати кілька типів приводу

(наприклад, передній та повний), і один тип приводу може використовуватись у кількох моделях. Зв'язок реалізовано через проміжну таблицю vehicles.VehicleDrivetrainTypesVehicleModels.

- VehicleEngineTypes ↔ VehicleModels: Одна модель автомобіля може підтримувати різні типи двигунів (наприклад, бензиновий та дизельний), і один тип двигуна може використовуватись у кількох моделях. Зв'язок реалізовано через проміжну таблицю vehicles.VehicleEngineTypesVehicleModels.
- VehicleTransmissionTypes ↔ VehicleModels: Одна модель автомобіля може мати кілька варіантів трансмісій (наприклад, механічна та автоматична), і одна трансмісія може бути доступною для кількох моделей. Зв'язок реалізовано через проміжну таблицю vehicles.VehicleTransmissionTypesVehicleModels.
- Vehicles ↔ UserWishlists: Кожен автомобіль може бути доданий до списку побажань багатьох користувачів, і кожен список побажань користувача може містити багато різних автомобілів. Зв'язок реалізовано через проміжну таблицю vehicles.VehiclesWishlists.

Враховуючи все зазначене раніше, включаючи визначені сутності, їх атрибути та встановлені зв'язки, а також застосовуючи schema-separated design (згрупувавши таблиці за схемами "identity", "locations" та "vehicles"), можна перейти безпосередньо

до візуалізації цих сутностей та їхніх зв'язків у ER (Entity-Relationship) діаграмі з використанням сервісу dbdiagram.io та мови DBML.

Лістинг 2.1 – DBML код таблиці VehicleBodyTypes.

```
Table "vehicles"."VehicleBodyTypes" {
  "Id" uuid [pk, not null]
  "Name" varchar(64) [not null]
  Indexes {
    Name [unique, name: "IX_VehicleBodyTypes_Name"]
  }
}
```

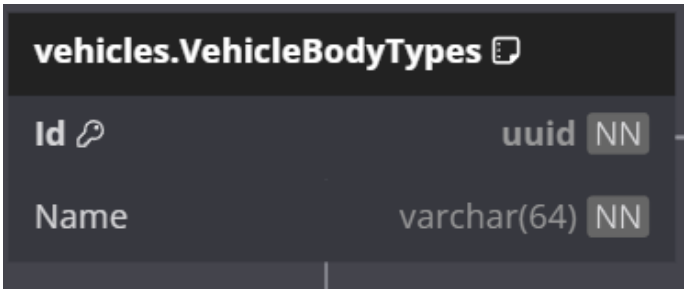


Рис. 2.1. Вигляд таблиці VehicleBodyTypes у ER-діаграмі

На рис. 2.2 можна побачити результат візуалізації сутностей з їх атрибутами, зв’язками, індексами тощо.

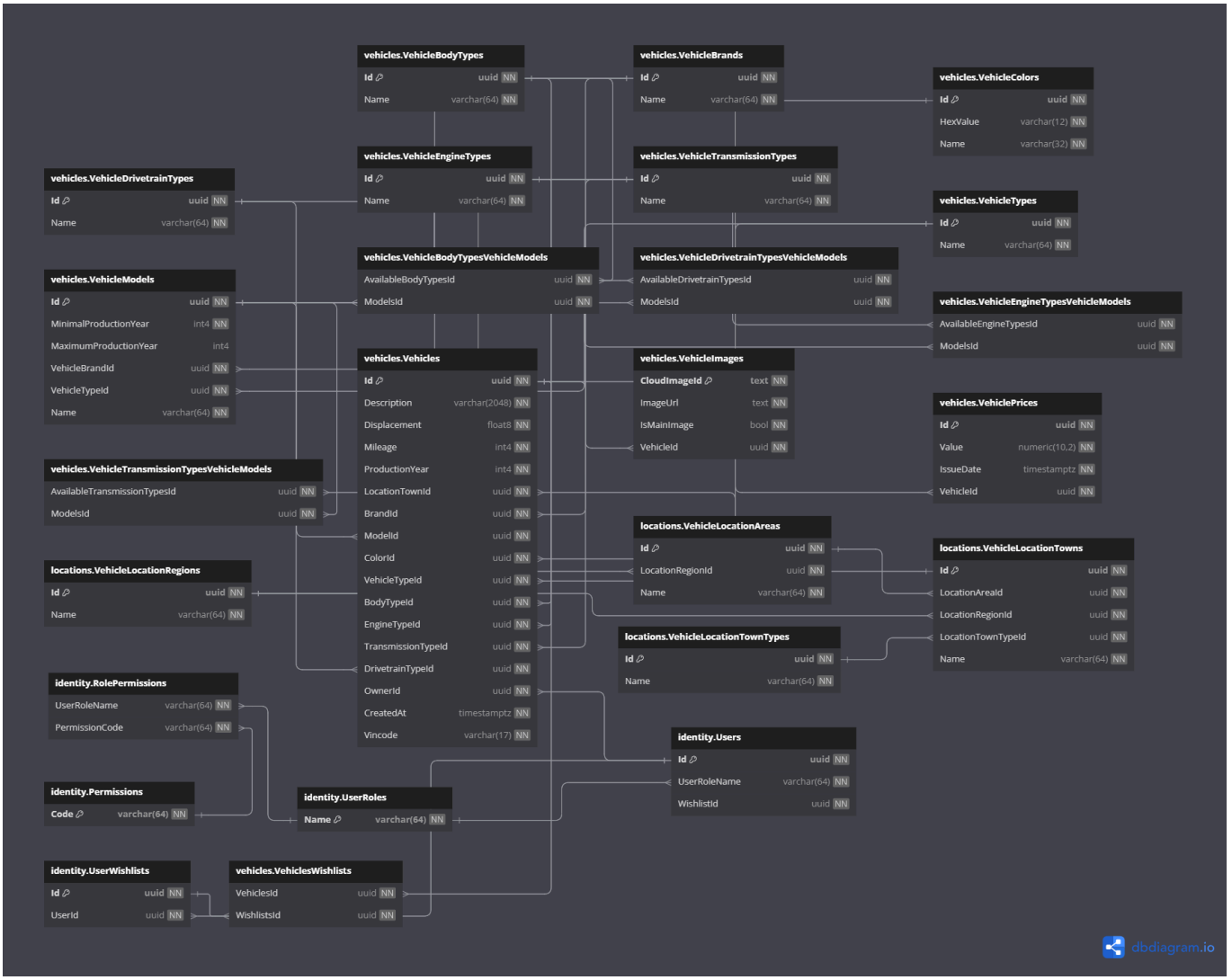


Рис. 2.2 – ER-діаграма сутностей бази даних

## 2.2. Підготовка та розгортання інфраструктури з використанням контейнеризації

Перед початком реалізації бази даних важливо підготувати інфраструктуру, яка стане основою для її розміщення та роботи інших компонентів системи. Розгортання інфраструктури є ключовим процесом, оскільки він забезпечує готовність середовища, в якому будуть розміщені та функціонуватимуть усі складові інформаційної системи маркетплейсу – як у локальному оточенні для розробки та тестування, так і в цільовому серверному чи хмарному середовищі для продуктивної експлуатації. Для реалізації цього етапу та забезпечення високої гнучкості, портативності й ефективності розгортання використовуються сучасні підходи до

контейнеризації, головним чином за допомогою інструментів Docker та Docker Compose.

Першим кроком у процесі контейнеризації веб-застосунку, який є основним компонентом, що взаємодіятиме з базою даних, є створення файлу **Dockerfile**. Dockerfile – це простий текстовий файл, що містить послідовність команд (інструкцій), які Docker виконує для автоматизованого створення образу Docker. Образ Docker, у свою чергу, являє собою незмінний шаблон (знімок файлової системи та конфігураційних налаштувань), на основі якого потім створюється контейнер. За допомогою Dockerfile розробник може детально описати весь процес збірки образу: визначити базовий образ, який буде основою, встановити необхідне системне програмне забезпечення та бібліотеки-залежності, скопіювати файли застосунку, налаштувати змінні середовища та вказати команди, які мають виконуватись при старті контейнера з цього образу. Такий декларативний підхід забезпечує відтворюваність процесу збірки образу та гарантує, що контейнер завжди буде мати необхідне та стандартизоване оточення для коректної роботи застосунку, незалежно від середовища, де він запускається.

Лістингу 2.2 – вміст Dockerfile, який використовується для збірки образу веб-застосунку даного проєкту

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
EXPOSE 5000
EXPOSE 5001

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY
["src/LanosCertifiedStore.Presentation/LanosCertifiedStore.Presen
```

```
tation.csproj", "src/LanosCertifiedStore.Presentation/"]
COPY
["src/LanosCertifiedStore.Application/LanosCertifiedStore.Applica
tion.csproj", "src/LanosCertifiedStore.Application/"]
COPY
["src/LanosCertifiedStore.Domain/LanosCertifiedStore.Domain.cspro
j", "src/LanosCertifiedStore.Domain/"]
COPY
["src/LanosCertifiedStore.Infrastructure/LanosCertifiedStore.Infr
astructure.csproj", "src/LanosCertifiedStore.Infrastructure/"]
COPY
["src/LanosCertifiedStore.Persistence/LanosCertifiedStore.Persist
ence.csproj", "src/LanosCertifiedStore.Persistence/"]
RUN dotnet restore
"src/LanosCertifiedStore.Presentation/LanosCertifiedStore.Present
ation.csproj"
COPY . .
WORKDIR "/src/src/LanosCertifiedStore.Presentation"
RUN dotnet build "LanosCertifiedStore.Presentation.csproj" -c
$BUILD_CONFIGURATION -o /app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "LanosCertifiedStore.Presentation.csproj" -c
$BUILD_CONFIGURATION -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
```

```
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "LanosCertifiedStore.Presentation.dll"]
```

Вміст цього файлу описує процес збірки Web-застосунку. Спочатку вказуються залежності, які необхідно встановити для того, щоб додаток міг функціонувати у контейнері:

- образ .NET 8 SDK (його максимально урізана версія)
- образ ASP.NET Core 8, який містить runtime ASP.NET.

Також на цьому етапі відразу встановлюється користувач “app” для запуску застосунку та робоча директорія “/app”, в якій зберігатимуться бінарні файли, створені у результаті збірки застосунку. Після чого відбуваються процеси збірки, публікації та створення фінального образу, який буде використовуватись в Docker.

Маючи робочий Dockerfile, можна перейти до формування файлу docker-compose.yml, який буде служитиме точкою входу до всієї системи. Цей файл опише структуру всіх контейнерів, що використовуватимуться в системі та буде за потреби вмикати або створювати їх.

Лістинг 2.3 – вміст файлу docker-compose.yml.

```
services:
 lanoscertifiedstore.presentation:
   image: ${DOCKER_REGISTRY-}lanoscertifiedstorepresentation
   build:
     context: .
     dockerfile: src/LanosCertifiedStore.Presentation/Dockerfile
   ports:
     - "5000:5000"
     - "5001:5001"
   environment:
     - ASPNETCORE_ENVIRONMENT=Development
```

```
      - ASPNETCORE_URLS=https://+:5001;http://+:5000
      -
ASPNETCORE_Kestrel__Certificates__Default__Password=password
      -
ASPNETCORE_Kestrel__Certificates__Default__Path=/https/aspnetapp.
pfx
      -
OTEL_EXPORTER_OTLP_ENDPOINT=http://lanoscertifiedstore.dashboard:
18889
    volumes:
      - ~/.aspnet/https:/https:ro
    depends_on:
      lanoscertifiedstore.database:
        condition: service_healthy
        restart: true

 lanoscertifiedstore.database:
    image: postgres:16.2-alpine3.19
    healthcheck:
      test: [ "CMD-SHELL", "pg_isready" ]
      interval: 10s
      retries: 5
      start_period: 30s
      timeout: 10s
    container_name: lsc_db
    environment:
      - POSTGRES_DB=LanosCertifiedStore
      - POSTGRES_USER=postgres
```

```
      - POSTGRES_PASSWORD=postgres
    volumes:
      - ./.containers/lsc_db:/var/lib/postgresql/data
    ports:
      - "5432:5432"

 lanoscertifiedstore.identity:
    image: quay.io/keycloak/keycloak:25.0.1
    container_name: lsc_identity
    command: start-dev --import-realm
    environment:
      - KC_HEALTH_ENABLED=true
      - KEYCLOAK_ADMIN=admin
      - KEYCLOAK_ADMIN_PASSWORD=admin
      -
KC_HTTPS_CERTIFICATE_FILE=/opt/keycloak/certs/localhost_cert.pem
      -
KC_HTTPS_CERTIFICATE_KEY_FILE=/opt/keycloak/certs/localhost_key.p
em
      -
API_REGISTRATION_MESSAGE_ENDPOINT=https://host.docker.internal:50
01/api/identity
    volumes:
      - ./.containers/identity:/opt/keycloak/data
      -
./.files/keycloak/realms/lsc-realm-export.json:/opt/keycloak/data
/import/realm.json
      -
```

```
./.files/keycloak/themes/lsc-theme.jar:/opt/keycloak/providers/ls
c-theme.jar
      -
./.files/keycloak/validators/unique-attribute-validator.jar:/opt/
keycloak/providers/unique-attribute-validator.jar
      -
./.files/keycloak/listeners/custom-event-listener.jar:/opt/keyclo
ak/providers/custom-event-listener.jar
      - ./.certs:/opt/keycloak/certs
    ports:
      - "18080:8080"
      - "8843:8443"
      - "9000:9000"
    healthcheck:
      test: [
        "CMD-SHELL",
        "exec 3<>/dev/tcp/127.0.0.1/9000;echo -e 'GET
/health/ready HTTP/1.1\r\nhost:
        http://localhost\r\nConnection: close\r\n\r\n' >&3;if [ $?
-eq 0 ];
        then echo 'Healthcheck Successful';exit 0;else echo
'Healthcheck Failed';exit 1;fi;"
      ]
      interval: 30s
      timeout: 10s
      retries: 3
    extra_hosts:
      - "host.docker.internal:host-gateway"
```

```
 lanoscertifiedstore.seq:
   image: datalust/seq:latest
   container_name: lsc_seq
   volumes:
     - ./.containers/seq:/data
   environment:
     - ACCEPT_EULA=Y
   ports:
     - "5341:5341"
     - "8081:80"

 lanoscertifiedstore.dashboard:
   image:
mcr.microsoft.com/dotnet/nightly/aspire-dashboard:latest
   container_name: lsc_dashboard
   ports:
     - 18888:18888
```

У даному файлі описано п'ять основних сервісів, що формують локальне середовище розробки та тестування маркетплейсу вживаних автомобілів. Це веб-застосунок, база даних PostgreSQL, сервер identity Keycloak, сервер централізованого збору логів Seq та панель моніторингу Aspire Dashboard. Велика перевага Docker Compose полягає в тому, що архітектура системи є гнучкою – нові сервіси можна легко інтегрувати, просто додавши їх опис до цього конфігураційного файлу.

Розглянемо конфігурацію деяких сервісів, а саме Presentation, Database та Seq. Решту сервісів буде розглянуто далі.

- **Сервіс Presentation (Веб-застосунок)**: Цей сервіс відповідає за виконання основного веб-застосунку маркетплейсу. Його образ не використовує

попередньо зібраний образ з реєстру, а будується локально на основі Dockerfile. Налаштовано важливі змінні середовища для конфігурації ASP.NET Core, включаючивизначення внутрішніх URL-адрес для Kestrel параметрів сертифіката Kestrel для HTTPS та кінцевої точки для експорту телеметрії OpenTelemetry що вказує на сервіс моніторингу Aspire Dashboard. Для доступу до сертифіката HTTPS у контейнер монтується локальний каталог "*~/.aspnet/https*" у контейнерний шлях "*/https*". Визначено критичну залежність від сервісу бази даних з умовою **service_healthy**, що гарантує, що веб-застосунок не буде запущено доти, доки контейнер бази даних не повідомить про свою повну готовність та коректне функціонування. Параметр restart: true вказує на спробу перезапуску контейнера, якщо залежність стане нездоровою.

- **Сервіс Database (База даних PostgreSQL)**: Цей сервіс використовує стандартний образ PostgreSQL на базі мінімалістичного дистрибутиву Alpine Linux як систему управління базою даних. Для моніторингу стану контейнера налаштовано health check, який регулярно перевіряє готовність бази даних приймати підключення. Ця перевірка є важливим елементом моніторингу та телеметрії стану ключового компонента системи. Встановлено змінні середовища для ініціалізації бази даних при першому запуску контейнера, включаючи назву бази даних, ім'я суперкористувача та його пароль. Зберігання даних бази даних забезпечується монтуванням локального каталогу "*./.containers/lsc_db*" на стандартний шлях зберігання даних PostgreSQL у контейнері, що гарантує збереження даних між перезапусками контейнера.
- **Сервер логів Seq**: Цей сервіс використовує образ datalust/seq:latest для розгортання сервера централізованого збору, зберігання та аналізу логів застосунків. Встановлено змінну середовища ACCEPT_EULA=Y для автоматичного прийняття ліцензійної угоди Seq при першому запуску. Сервіс мапує порти 5341 хост-машини на порт 5341 контейнера (стандартний порт для прийому логів від клієнтів) та порт 8081 хост-машини на порт 80 контейнера (для доступу до веб-інтерфейсу Seq).

Після того як контейнеризацію було налаштовано, систему можна запустити просто виконавши команду “**docker-compose up -d**”, яка відразу запустить всі сервіси вказані у файлі.

### 2.3. Реалізація бази даних за допомогою підходу Code-First

Логічним продовженням етапу проектування є реалізація бази даних. В рамках даного проєкту, це відбулось за допомогою підходу Code First. Першим кроком є трансформація спроєктованої логічної моделі, що була представлена, на структуру класів на мові програмування C#. Цей крок є фундаментальним у Code-First, оскільки саме ці класи, що відображають сутності доменної області застосунку (наприклад, "Автомобіль", "Користувач", "Оголошення"), стають декларативним описом бажаної схеми бази даних та використовуються Entity Framework Core для її генерації та управління.

На основі моделі, представленої у dbdiagram, кожна визначена таблиця бази даних відповідає окремому класу-сутності у коді застосунку. Властивості цих C#-класів відповідають стовпцям відповідних таблиць бази даних, зберігаючи при цьому відповідні типи даних.

Крім простого відображення таблиць та стовпців, класи-сутності також включають спеціальні властивості, відомі як навігаційні властивості. Саме вони є ключовим елементом, що дозволяє представляти зв'язки між різними сутностями на рівні об'єктної моделі застосунку. В Code-First підході, Entity Framework Core аналізує ці навігаційні властивості, їхні типи, а також застосовані конвенції іменування або явні налаштування для автоматичного визначення типу зв'язку між сутностями (один-до-одного, один-до-багатьох, багато-до-багатьох) та коректного відображення його на реляційну схему бази даних.

Наприклад, для реалізації зв'язку "один до багатьох" (1:N), такого як зв'язок між сутністю VehicleModel (одна модель) та сутністю Vehicle (багато автомобілів, що належать до цієї моделі), у класі VehicleModel буде додано навігаційну властивість типу колекції, в даному випадку – ICollection<Vehicle> Vehicles. Ця властивість дозволить отримати доступ до колекції всіх об'єктів Vehicle, пов'язаних з конкретним

об'єктом VehicleModel. У відповідному класі Vehicle буде присутня навігаційна властивість типу VehicleModel для доступу до об'єкта моделі, а також властивість для зовнішнього ключа, що встановлює зв'язок. EF Core розпізнає цей патерн як зв'язок "один до багатьох".

Зв'язки "багато до багатьох" (M:N), які у реляційній моделі традиційно реалізуються через окрему проміжну таблицю зв'язку (join table) з двома зовнішніми ключами, в Code-First підході з EF Core можуть бути налаштовані більш абстрактно та зручно на рівні класів сутностей. Для встановлення зв'язку M:N між двома сутностями (наприклад, між VehicleModel та VehicleBodyType, оскільки одна модель може мати багато типів кузовів, а один тип кузова може бути доступним для багатьох моделей) розробник зазвичай додає колекції навігаційних властивостей у обох класах сутностей, що беруть участь у зв'язку. Entity Framework Core, за умови дотримання конвенцій іменування або при явному налаштуванні зв'язку за допомогою Fluent API, автоматично розпізнає такий патерн "багато до багатьох" і самостійно генерує необхідну проміжну таблицю зв'язку в базі даних під час застосування міграцій, без необхідності явного створення окремого класу для цієї проміжної таблиці у коді застосунку (хоча це також є можливою опцією для складніших сценаріїв M:N, коли таблиця зв'язку містить додаткові атрибути). Це спрощує об'єктну модель у коді, дозволяючи розробнику зосередитись на бізнес-логіці, а не на технічних деталях реляційного відображення M:N зв'язків.

Лістинг 2.4. Таблиця VehicleModels у вигляді класу на мові програмування C#.

```
public sealed class VehicleModel : NamedAspect
{
    public Guid Id { get; init; }
    public string Name { get; set; } = null!;
    public int MinimalProductionYear { get; set; }
    public int? MaximumProductionYear { get; set; }
    public Guid VehicleBrandId { get; set; }
    public VehicleBrand VehicleBrand { get; set; } = null!;
```

```
    public Guid VehicleTypeId { get; set; }
    public VehicleType VehicleType { get; set; } = null!;
    public ICollection<VehicleEngineType> AvailableEngineTypes {
get; set; } = [];
    public ICollection<VehicleTransmissionType>
AvailableTransmissionTypes { get; set; } = [];
    public ICollection<VehicleDrivetrainType>
AvailableDrivetrainTypes { get; set; } = [];
    public ICollection<VehicleBodyType> AvailableBodyTypes { get;
set; } = [];
    public ICollection<Vehicle> Vehicles { get; set; } = [];
}
```

Лістинг 2.5. Таблиця Vehicles у вигляді класу на мові програмування C#.

```
public sealed class Vehicle : IIdentifiable<Guid>
{
   public Guid Id { get; set; } = Guid.NewGuid();
   public string Description { get; set; } = null!;
   public double Displacement { get; set; }
   public int Mileage { get; set; }
   public int ProductionYear { get; set; }
   public string Vincode { get; set; }
   public Guid LocationTownId { get; set; }
   public VehicleLocationTown LocationTown { get; set; } = null!;
   public Guid BrandId { get; set; }
   public VehicleBrand Brand { get; set; } = null!;
   public Guid ModelId { get; set; }
   public VehicleModel Model { get; set; } = null!;
```

```
    public Guid ColorId { get; set; }
    public VehicleColor Color { get; set; } = null!;
    public Guid VehicleTypeId { get; set; }
    public VehicleType VehicleType { get; set; } = null!;
    public Guid BodyTypeId { get; set; }
    public VehicleBodyType BodyType { get; set; } = null!;
    public Guid EngineTypeId { get; set; }
    public VehicleEngineType EngineType { get; set; } = null!;
    public Guid TransmissionTypeId { get; set; }
    public VehicleTransmissionType TransmissionType { get; set; }
= null!;
    public Guid DrivetrainTypeId { get; set; }
    public VehicleDrivetrainType DrivetrainType { get; set; } =
null!;
    public ICollection<VehicleImage> Images { get; set; } = [];
    public ICollection<VehiclePrice> Prices { get; set; } = [];
    public ICollection<UserWishlist> Wishlists { get; set; } = [];
    public Guid OwnerId { get; set; }
    public User Owner { get; set; } = null!;
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
}
```

На додаток до можливості конфігурації моделі даних за допомогою атрибутів даних (Data Annotations) безпосередньо у класах сутностей, Entity Framework Core надає розробникам більш потужний та гнучкий інструмент – **Fluent API**. Fluent API базується на використанні ланцюжка методів конфігурації у спеціальному API, що дозволяє детально налаштовувати відображення об'єктної моделі на реляційну схему бази даних. Використання Fluent API забезпечує значно більший контроль над

процесом конфігурації та дозволяє реалізовувати складніші сценарії відображення та обмежень, які неможливо виразити за допомогою атрибутів.

Для покращення організації коду конфігурації Fluent API та забезпечення чистоти самих класів сутностей, рекомендовано використовувати інтерфейс IEntityTypeConfiguration<TEntity>. Реалізація цього інтерфейсу для кожної сутності дозволяє винести всю специфічну логіку конфігурації для даного класу-сутності в окремий, спеціалізований клас. Такий підхід робить класи сутностей чистішими, вільними від деталей конфігурації бази даних, та суттєво підвищує зручність управління конфігурацією моделі даних, оскільки вона стає більш модульною та легко локалізується.

Як наочний приклад використання Fluent API у поєднанні з IEntityTypeConfiguration, розглянемо конфігурацію сутності VehicleColor. На лістингу 2.6, що демонструє відповідний клас конфігурації, можна побачити, як за допомогою методів Fluent API детально налаштовуються окремі властивості цієї сутності та її зв'язки з іншими сутностями. Зокрема, для атрибутів Name та HexValue встановлюється, що вони є обов'язковими для заповнення і визначається їх максимальна допустима довжина, конфігурується зв'язок між VehicleColor та Vehicle, вказується назва таблиці в базі даних, яка відповідатиме сутності, а також схема, до якої належатиме ця таблиця ("vehicles"), що відповідає schema-separated дизайну.

Лістинг 2.6. Налаштування таблиці VehicleColor за допомогою FluentAPI.

```
internal sealed class VehicleColorEntityConfiguration :
IEntityTypeConfiguration<VehicleColor>
{
    public void Configure(EntityTypeBuilder<VehicleColor>
builder)
    {
        builder.Property(p => p.Name)
            .IsRequired()
            .HasMaxLength(32);
```

```
        builder.Property(p => p.HexValue)
            .IsRequired()
            .HasMaxLength(12);

        builder.HasMany(m => m.Vehicles)
            .WithOne(v => v.Color)
            .OnDelete(DeleteBehavior.Cascade);
        builder.HasIndex(p => p.Name).IsUnique();
        builder.ToTable(DatabaseConstants.Tables.VehicleColors,
DatabaseConstants.Schemas.VehiclesSchema);
    }
}
```

Після налаштування всіх сутностей потрібно оновити контекст бази даних, вказавши шлях до збірки, з якої EF Core має зчитувати конфігурації таблиць. Далі йде процес створення та подальшого застосування міграцій, які розгортатимуть ці таблиці у базі даних, результат якого можна побачити на рис. 2.3.

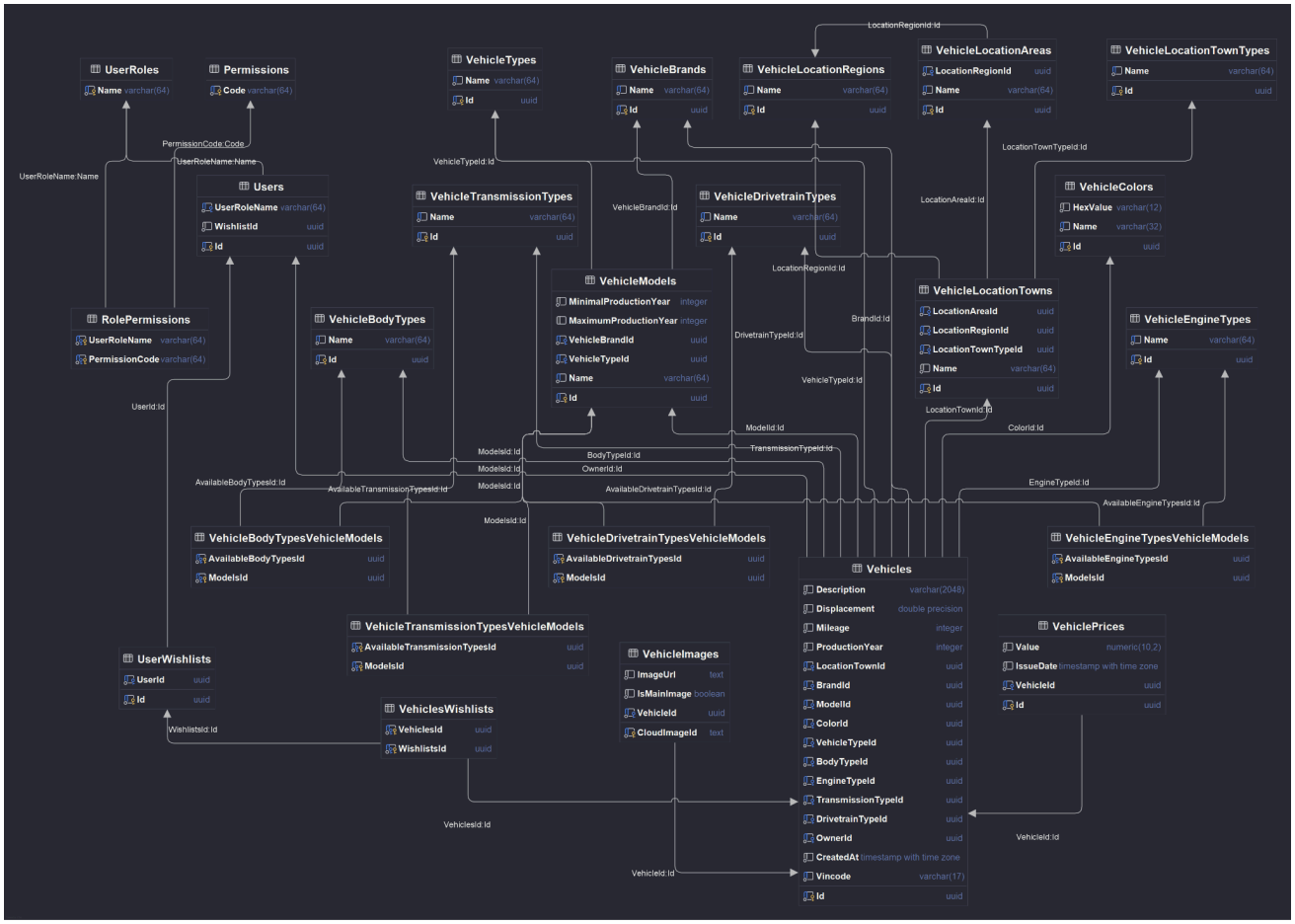


Рис. 2.3 – імплементована діаграма сутностей

Джерело: згенеровано автором

## 2.4. Налаштування сідингу бази даних

**Data Seeding** – це процес попереднього заповнення бази даних початковими даними, що необхідні для нормальної роботи застосунку, його тестування або демонстрації. Він гарантує наявність базових відомостей для належного функціонування системи одразу після розгортання, що значно спрощує процес її ініціалізації та налаштування середовища. До основних цілей Data Seeding входять:

- Забезпечення наявності обов'язкових довідникових даних, конфігураційних параметрів або стандартних записів, які потрібні для коректної роботи застосунку з першого запуску.
- Автоматичне заповнення бази даних стабільним і передбачуваним набором даних для тестів, що сприяє відтворюваності тестових сценаріїв і знижує ризик помилок, пов'язаних з відсутністю необхідних даних.

- Гарантування наявності базових даних у всіх середовищах розгортання (розробка, тестування, продуктивне), що спрощує процеси безперервної інтеграції та розгортання (CI/CD).

У проєкті механізм Data Seeding імплементовано шляхом створення окремих класів, кожен з яких відповідає за генерацію списків початкових даних для певної сутності або логічної групи. Ці класи представляють структуровану форму зберігання даних для сідингу. Загальна структура процесу показана на лістингу 2.7. Дані генеруються у відповідних класах, збираються у центральному методі Seed класу SeedData та додаються до бази даних за допомогою контексту Entity Framework Core. Класи для сідингу даних поділяються на два основні типи за їхнім призначенням:

**Класи, необхідні для функціонування застосунку**: Ці класи відповідають за наповнення бази даних основними референсними (довідниковими) даними, без яких коректна робота певних модулів або всієї системи є неповною. До них належать:

- SeedRegions.cs – наповнює базу списком регіонів.
- SeedTowns.cs – додає у базу даних міста, які належать до вказаних регіонів.
- SeedVehicleBodyTypes.cs – формує перелік типів кузовів (седан, хетчбек тощо).
- SeedVehicleDrivetrainTypes.cs – містить дані про типи приводів (передній, задній, повний).
- SeedVehicleEngineTypes.cs – додає інформацію про типи двигунів (бензин, дизель, електро тощо).
- SeedVehicleTransmissionTypes.cs – містить дані про види трансмісій (механічна, автоматична).
- SeedVehicleTypes.cs – загальна класифікація транспортних засобів (легкові, вантажні, мото тощо).
- SeedColors.cs – забезпечує кольори, доступні для відображення або вибору при створенні/редагуванні транспортних засобів.

**Класи, що використовуються для тестових екземплярів:** Ці класи зазвичай застосовуються для наповнення бази тестовими або демонстраційними даними, які

не є критичними для базового функціонування застосунку, але значно спрощують процеси розробки та тестування:

- SeedModels.cs – створює тестові моделі транспортних засобів (назви моделей, специфікації тощо).
- SeedVehicles.cs – генерує список готових «демо»-транспортних засобів, які можуть бути використані в тестових сценаріях або демонстраційних прикладах.

Лістинг 2.7 – Повна структура сідингових класів

```
│   SeedBrands.cs
│   SeedColors.cs
│   SeedData.cs
│   SeedImages.cs
│   SeedModels.cs
│   SeedVehicles.cs
│
├───LocationRelated
│       SeedAreas.cs
│       SeedRegions.cs
│       SeedTowns.cs
│
└───TypeRelated
        SeedVehicleBodyTypes.cs
        SeedVehicleDrivetrainTypes.cs
        SeedVehicleEngineTypes.cs
        SeedVehicleTransmissionTypes.cs
        SeedVehicleTypes.cs
```

У результаті маємо механізм автоматичного заповнення бази даних необхідними даними, який дозволяє знизити ризик помилок при розгортанні, спрощує тестування та сприяє стабільній роботі системи у різних середовищах.

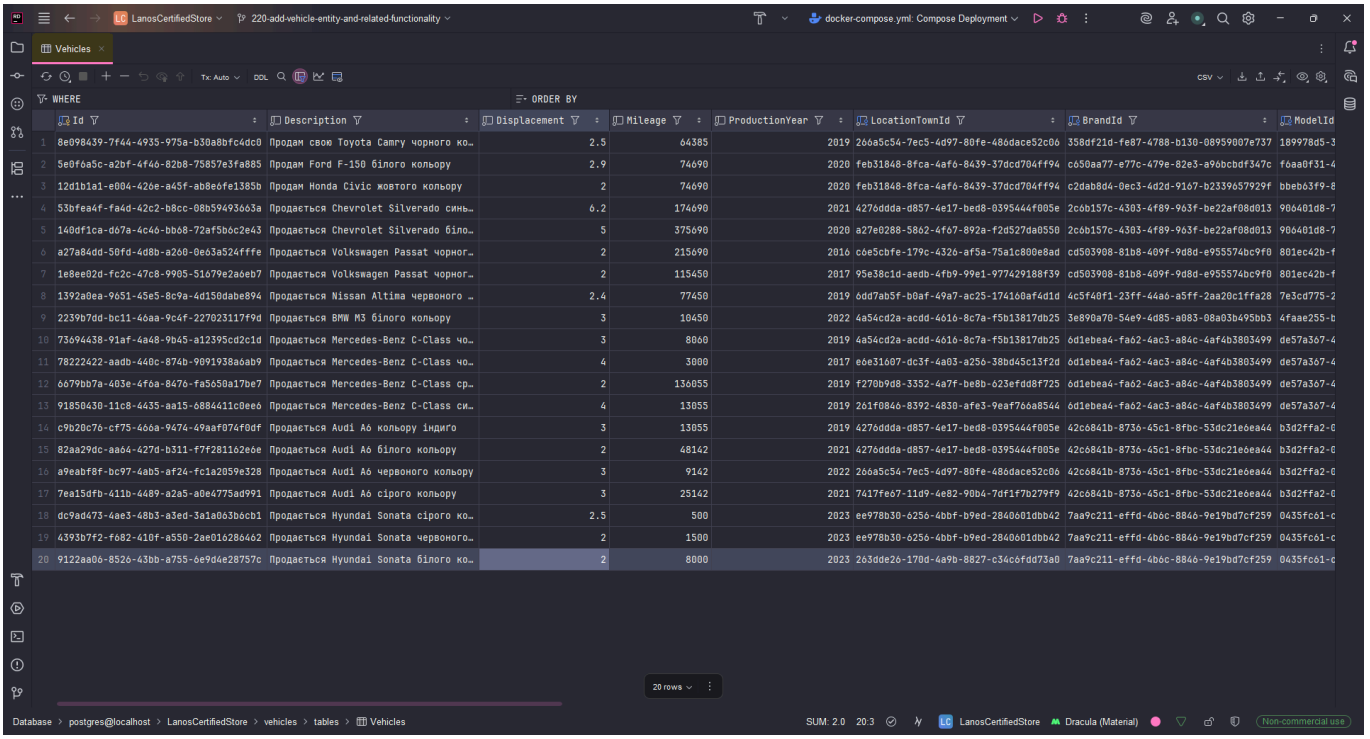


Рис. 2.4. Вигляд заповненої таблиці Vehicles у базі даних

## 2.5. Створення архітектурних тестів для форсування архітектури

Як вже було згадано раніше, архітектурні тести – це автоматизовані перевірки, які аналізують структуру та дизайн коду застосунку. Їхня основна мета полягає у гарантуванні, що реалізована структура застосунку відповідає задуманій архітектурі, ефективно запобігаючи порушенням принципів залежностей між компонентами та некоректному розподілу їхніх обов'язків.

Для форсування архітектури застосунку та автоматичного контролю за дотриманням цих правил було використано бібліотеку з відкритим кодом **NetArchTest.** Цей проект дозволяє створювати тести, які забезпечують дотримання конвенцій щодо дизайну класів, іменування та залежностей у кодових базах .NET. Їх можна використовувати з будь-яким фреймворком для модульного тестування і включати в конвеєр збірки. Він використовує Fluent API, який дозволяє створювати

зручні для читання правила, які можна використовувати в тестових твердженнях [19]. Тести було поділено на п’ять категорій та реалізовано наступним чином:

### 2.5.1. Тести на залежності між проєктами (dependency rules)

Ці тести призначені для контролю дозволених зв'язків між окремими проєктами (або логічними шарами) застосунку на рівні збірок. Вони дозволяють підтримувати чітку роздільність відповідальностей та сильну ізоляцію логіки між різними частинами системи. Наприклад, такий тест гарантує, що основна збірка домену (Domain), яка містить сутності та бізнес-логіку, не має прямих залежностей від інших проєктів рішення, таких як Infrastructure, Persistence чи Presentation.

Лістинг 2.8. Код тесту Domain_Should_NotHaveReferencesToOtherProjects

```
[Fact]
    public void Domain_Should_NotHaveReferencesToOtherProjects()
    {
        var result = Types
            .InAssembly(DomainAssembly)
            .ShouldNot()
            .HaveDependencyOnAny(
                "LanosCertifiedStore.Application",
                "LanosCertifiedStore.Infrastructure",
                "LanosCertifiedStore.Persistence",
                "LanosCertifiedStore.Presentation")
            .GetResult();
        result.IsSuccessful.Should().BeTrue();
    }
```

### 2.5.2. Тести на коректність модифікаторів доступу та інших властивостей класів

Ці перевірки забезпечують правильне використання модифікаторів доступу (наприклад, internal, sealed) для управління видимістю типів та їхніх членів, що є важливим для контролю зовнішнього API компонентів та уникнення небажаного

успадкування чи розширення класів за межами визначених меж. Наприклад, тест може перевіряти, що всі класи, які реалізують певний інтерфейс, такий як IRequestHandler<,> (з бібліотеки MediatR, що часто використовується в CQRS-подібних архітектурах), не є публічними (public) або є запечатаними (sealed), обмежуючи їх використання внутрішньою збіркою або запобігаючи успадкуванню.

Лістинг 3.2. Код тесту RequestHandlers_Should_BeInternalAndSealed

```
[Fact]
    public void RequestHandlers_Should_BeInternalAndSealed()
    {
        var types = Types.InAssembly(ApplicationAssembly)
            .That()
            .ImplementInterface(typeof(IRequestHandler<,>))
            .And().ArePublic()
            .And().AreNotSealed()
            .GetTypes();

        types.Should().BeEmpty();
    }.
```

### 2.5.3. Тести на відповідність іменувальним конвенціям

Ці тести автоматично перевіряють, чи імена класів, що реалізують певні інтерфейси або належать до визначених категорій, відповідають встановленим правилам іменування. Дотримання іменувальних конвенцій є важливим для підтримки зрозумілості, однорідності та передбачуваності кодової бази, що спрощує її читання та розуміння. Наприклад, тест може перевіряти, що всі класи, які реалізують інтерфейс IRequestHandler<,>, мають імена, які послідовно закінчуються суфіксом "RequestHandler".

Лістинг 2.9. Код тесту RequestHandlersNames_Should_EndWithRequestHandler

```
[Fact]
```

```
public void RequestHandlersNames_Should_EndWithRequestHandler()
{
    var handlerTypes =
GetTypesImplementingInterfaceNotEndingWith(typeof(IRequestHandler
<,>), "RequestHandler");
    handlerTypes.Should().BeEmpty();
}
```

2.5.4. Тести на структурні вимоги (структурні контракти)

Ці перевірки забезпечують виконання певних фундаментальних конструктивних умов або "контрактів", які мають дотримуватися певні класи або сутності. Ці умови можуть бути необхідні для коректної роботи застосунку або фреймворків, що використовуються (наприклад, ORM, серіалізації). Наприклад, тест може перевіряти, що всі класи, які представляють сутності домену (наприклад, успадковують певний базовий клас або реалізують інтерфейс IIdentifiable<>), повинні мати публічний конструктор без параметрів. Це типова вимога для багатьох ORM (як-от Entity Framework Core) для можливості створення екземплярів сутностей через рефлексію під час завантаження даних з бази даних.

Лістинг 2.10. Код тесту Entities_Should_HavePublicParameterlessConstructor

```
public void Entities_Should_HavePublicParameterlessConstructor()
    {
        var entityTypes = Types.InAssembly(DomainAssembly)
            .That()
            .Inherit(typeof(IIdentifiable<>))
            .GetTypes();

        var failingTypes = new List<Type>();

        // ReSharper disable once LoopCanBeConvertedToQuery
        foreach (var entityType in entityTypes)
        {
            var constructors =
entityType.GetConstructors(BindingFlags.Default);

            if (!constructors.Any(c => c.IsPublic &&
c.GetParameters().Length == 0))
            {
                failingTypes.Add(entityType);
            }
        }

        failingTypes.Should().BeEmpty();
    }
```

### 2.5.5. Тести на наявність необхідних атрибутів

Ці тести перевіряють, що класи певних типів (наприклад, контролери, сервіси) позначені потрібними атрибутами, які забезпечують їхню належну конфігурацію та

інтеграцію у фреймворки або бібліотеки. Наявність атрибутів часто є обов'язковою умовою для коректної роботи механізмів фреймворку. Наприклад, тест може перевіряти, що всі класи контролерів веб-API, які успадковують базовий клас BaseApiController, повинні бути позначені атрибутом [Route]. Це гарантує, що для кожного контролера визначено маршрут, що є необхідним для правильної маршрутизації вхідних HTTP-запитів до відповідних дій контролерів.

Лістинг 2.11. Код тесту Controllers_Should_HaveRouteAttribute

```
[Fact]
public void Controllers_Should_HaveRouteAttribute()
{
    var result = Types
        .InAssembly(PresentationAssembly)
        .That()
        .Inherit(typeof(BaseApiController))
        .Should()
        .HaveCustomAttribute(typeof(RouteAttribute))
        .GetResult();

    var reason = "All controllers should have [Route] attribute,
but found: " +
     result.FailingTypeNames?.Aggregate((src, res) => src + ", "
+ res);
    result.IsSuccessful.Should().BeTrue(because: reason);
}
```

В сукупності ці архітектурні тести формують комплексну систему автоматизованих перевірок, яка дозволяє систематично контролювати та гарантувати дотримання встановлених архітектурних стандартів, принципів проектування та структурних вимог проєкту. Інтеграція цих тестів у конвеєр безперервної інтеграції/безперервного розгортання (CI/CD pipeline) дозволяє автоматично

виявляти будь-які відхилення від архітектурних вимог на ранніх етапах розробки. Це значно підвищує надійність, стійкість, підтримуваність та якість архітектури застосунку в довгостроковій перспективі, запобігаючи накопиченню технічного боргу.

### 2.6. Впровадження перевірок стану застосунку (Health checks)

**Health checks** – це механізми моніторингу, що надають інформацію про поточний стан застосунку або його окремих компонентів. Вони дозволяють внутрішнім системам (наприклад, балансувальникам навантаження) або зовнішнім сервісам моніторингу отримувати стандартизовану інформацію про "здоров'я" API чи сервісу. Health checks дають можливість контролювати стан додатка шляхом виконання невеликих тестів, які повертають один із трьох результатів: здоровий (healthy), погіршений (degraded) або нездоровий (unhealthy). Це корисно не тільки для перевірки внутрішнього стану самого застосунку, але й для моніторингу доступності та коректності роботи зовнішніх залежностей, на які покладається програма [20].

Впровадження перевірок стану має низку ключових переваг для забезпечення надійності та стабільності системи:

- **Раннє виявлення проблем**: Автоматизовані перевірки дозволяють оперативно виявити збої або погіршення продуктивності компонентів, що допомагає запобігти більш серйозним проблемам у майбутньому.
- **Підтримка високої доступності**: Завдяки інформації від health checks, системи оркестрації або балансувальники навантаження можуть автоматично виключати недоступні (unhealthy) інстанції застосунку з пулу обробки запитів, спрямовуючи трафік лише до "здорових" серверів.
- **Моніторинг стану сервісів**: Регулярні перевірки дають змогу командам розробки та експлуатації оперативно реагувати на зміни стану системи, що є основою для планування обслуговування, масштабування або вжиття заходів для відновлення.

- **Автоматизація управління**: В оркестраційних середовищах (наприклад, Kubernetes) health checks використовуються як тригери для автоматичних операцій, таких як самовідновлення застосунків, перезапуск непрацюючих контейнерів або автоматичне масштабування.

Чому health checks важливі? Застосунок, що має інтегровані health checks, демонструє вищу стійкість до збоїв, оскільки проблемні компоненти швидко виявляються та ізолюються. Автоматизоване управління життєвим циклом інстанцій на основі їх "здоров'я" дозволяє оптимально використовувати апаратні та програмні ресурси інфраструктури. Завдяки постійному моніторингу та швидкій реакції на проблеми, користувачі отримують доступ до сервісів, що працюють стабільно, без несподіваних перебоїв у роботі. Інтегровані health checks надають цінну діагностичну інформацію про стан окремих компонентів, що суттєво скорочує час на локалізацію та вирішення проблем.

ASP.NET Core має вбудовану підтримку для імплементації health checks, представлену бібліотекою Microsoft.Extensions.Diagnostics.HealthChecks. Ця бібліотека надає зручний та розширюваний інтерфейс для легкої інтеграції health checks у застосунок, дозволяючи як створювати власні перевірки, так і впроваджувати вже існуючі, специфічні для певних сервісів або технологій. Хорошою практикою є використання готових health checks, які надаються фреймворками, бібліотеками або самими сервісами, оскільки вони розроблені, протестовані та оптимізовані для стандартних завдань моніторингу стану. Це забезпечує надійність, стабільність та легкість інтеграції. Створення власних health checks доцільне лише у виняткових випадках, коли готові рішення не відповідають унікальним вимогам застосунку – наприклад, при необхідності перевірки специфічної бізнес-логіки або стану нестандартних зовнішніх залежностей.

Щоб перейти до фактичної імплементації health checks у застосунку, необхідно спочатку визначити всі зовнішні сервіси та ключові компоненти, від стану яких критично залежить коректна робота системи. У контексті даного проєкту до них належать: база даних PostgreSQL, сервіс для обробки медіа-ресурсів Cloudinary та сервіс авторизації Keycloak.

Нижче наведено опис кроків, які були виконані під час інтеграції health checks у веб-застосунок:

1. Спочатку було впроваджено перевірку працездатності бази даних PostgreSQL. За допомогою розширення AddNpgSql(), що інтегрується з бібліотекою Microsoft.Extensions.Diagnostics.HealthChecks, забезпечується стандартна перевірка з'єднання з базою даних. Це дозволяє оперативно виявляти проблеми з підключенням до БД, які є критичними для роботи застосунку.
2. Оскільки для сервісу Cloudinary не використовується стандартний health check з бібліотек, був створений власний кастомний health check. Реалізація у класі CloudinaryHealthCheck, що імплементує інтерфейс IHealthCheck, здійснює перевірку доступності сервісу Cloudinary шляхом виклику його вбудованого Ping-методу. Логіка перевірки проста: якщо відповідь від Cloudinary має статус OK, перевірка вважається успішною, і метод CheckHealthAsync повертає статус Healthy. В іншому випадку – повертається статус Unhealthy. Це дозволяє контролювати доступність сервісу Cloudinary в режимі реального часу та реагувати на можливі проблеми з його боку.

Лістинг 2.12. Вигляд імплементації класу CloudinaryHealthCheck

```
internal sealed class CloudinaryHealthCheck(ICloudinary
cloudinarySource) : IHealthCheck
{
    public async Task<HealthCheckResult>
CheckHealthAsync(HealthCheckContext context,
        CancellationToken cancellationToken = new())
    {
        var result = await
cloudinarySource.PingAsync(cancellationToken);
        return result.StatusCode == HttpStatusCode.OK
            ? HealthCheckResult.Healthy()
```

```
            : HealthCheckResult.Unhealthy();
    }
}
```

3. Після створення та визначення необхідних health checks (як стандартних, так і кастомних), вони були зареєстровані у DI-контейнер. Додавання кастомної перевірки для Cloudinary здійснюється методом AddCheck<CloudinaryHealthCheck>(). Стандартна перевірка для PostgreSQL додається методом AddNpgSql(), передаючи рядок підключення до бази даних.

Лістинг 2.13. Реєстрація Health Checks в DI контейнер.

```
services.AddHealthChecks()
    .AddCheck<CloudinaryHealthCheck>("cloudinary",
HealthStatus.Unhealthy)
.AddNpgSql(configuration.GetConnectionString("PostgreSqlConnectio
n")!);
```

4. Для забезпечення можливості отримання деталізованої та структурованої відповіді щодо стану кожного окремого health check при запиті до відповідного ендпоінту, було інтегровано клієнтську бібліотеку AspNetCore.HealthChecks.UI.Client. Ця бібліотека надає зручний формат відповіді, який включає загальний статус застосунку та статус кожного окремого health check з деталями, що полегшує моніторинг та візуалізацію результатів перевірок зовнішніми інструментами.
5. На фінальному етапі, у конвеєрі обробки запитів застосунку, було зареєстровано спеціальне middleware для health checks. Використано метод MapHealthChecks(), який прив'язує обробник запитів health checks до визначеного шляху (у цьому випадку /api/health). Для генерації відповіді на запит health check застосовано кастомний ResponseWriter, що надається

бібліотекою AspNetCore.HealthChecks.UI.Client. Цей writer форматує результат перевірок у деталізований, зрозумілий формат (JSON), що включає статус кожного зареєстрованого health check, дозволяючи зовнішнім системам парсити та використовувати ці дані.

Лістинг 2.14. Реєстрація HealthChecks Middleware

```
app.MapHealthChecks("api/health", new HealthCheckOptions
{
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
```

### 2.7. Впровадження телеметрії та метрик до застосунку

**Телеметрія** в контексті розробки програмного забезпечення – це систематизований процес збору, передачі та подальшого аналізу даних, які генеруються застосунком під час його виконання в реальному часі. Ці дані можуть включати показники продуктивності, інформацію з логів подій, трасування шляху виконання запитів та інші метрики, що дозволяють отримати глибоке розуміння внутрішнього стану та зовнішньої поведінки системи [21].

Впровадження телеметрії вирішує низку важливих операційних та діагностичних завдань:

- Моніторинг продуктивності: Дозволяє постійно відстежувати ключові показники роботи застосунку, оперативно виявляти "вузькі місця" та оптимізувати використання обчислювальних ресурсів.
- Виявлення та діагностика збоїв: Збирання логів та трасування допомагає швидко локалізувати першопричини помилок та аномалій у роботі системи, прискорюючи час реакції на інциденти та їх усунення.
- Аналіз поведінки користувачів: Надання даних про взаємодію користувачів із застосунком, що дозволяє приймати обґрунтовані рішення щодо

вдосконалення функціоналу та покращення загального користувацького досвіду.

- Планування масштабування: Аналіз зібраних даних про навантаження та використання ресурсів допомагає заздалегідь прогнозувати потреби у додаткових ресурсах та своєчасно проводити масштабування системи.

Загальновизнаним стандартом у галузі телеметрії є **OpenTelemetry (OTEL)** – відкритий фреймворк, що надає єдиний набір API, бібліотек та інструментів для стандартизованого збору, обробки та експорту телеметричних даних (метрик, логів, трасування) з різноманітних застосунків [22]. OpenTelemetry забезпечує незалежність від конкретних постачальників рішень для моніторингу, підтримує широкий спектр мов програмування та технологій, що робить його універсальним рішенням для сучасних розподілених систем та мікросервісних архітектур. Фреймворк легко інтегрується з популярними інструментами для візуалізації та аналізу даних (наприклад, Prometheus, Grafana, Jaeger) та дозволяє гнучко налаштовувати збір даних відповідно до специфічних потреб застосунку, ефективно працюючи з великими обсягами інформації.

У даному застосунку вже налагоджено ефективну систему логування подій та трасування запитів за допомогою бібліотек Serilog та Seq, яка покриває потреби проєкту у зборі логів та візуалізації трасування. Оскільки ці механізми вже повністю функціонують, OpenTelemetry буде інтегровано виключно для збору метрик. Це дозволить сфокусувати зусилля на моніторингу числових показників продуктивності та стану компонентів системи, не дублюючи функціонал, який вже успішно реалізовано іншими засобами.

Реалізація інтеграції OpenTelemetry для збору метрик передбачала виконання наступних кроків:

1. **Встановлення необхідних пакетів NuGet**: До проєкту було додано бібліотеки, що забезпечують підтримку OpenTelemetry у .NET-застосунку. Це включало пакети OpenTelemetry.Exporter.OpenTelemetryProtocol (для експорту зібраних метрик через стандартний протокол OTLP), OpenTelemetry.Extensions.Hosting (для інтеграції OpenTelemetry з механізмом

хостингу .NET), OpenTelemetry.Instrumentation.AspNetCore (для автоматичного збору стандартних метрик із конвеєра обробки запитів ASP.NET Core) та OpenTelemetry.Instrumentation.Http (для автоматичного збору метрик, пов'язаних з вихідними HTTP-запитами, які виконує застосунок).

2. Конфігурація телеметрії: У коді застосунку було виконано налаштування OpenTelemetry, як правило, на етапі конфігурації сервісів. Ця конфігурація включала:
    - Встановлення ресурсу з ідентифікатором сервісу "LanosCertifiedStore" для кращої ідентифікації даних.
    - Додавання інструментування для ASP.NET Core та HTTP клієнта, що дозволяє автоматично збирати відповідні метрики.
    - Використання OTLP експорту для передачі зібраних метрик до систем моніторингу.

Лістинг 2.15. Налаштування OpenTelemetry в DI контейнері

```
       services.AddOpenTelemetry()
           .ConfigureResource(resource =>
resource.AddService("LanosCertifiedStore"))
           .WithMetrics(metrics =>
           {
               metrics.AddAspNetCoreInstrumentation()
                   .AddHttpClientInstrumentation();

               metrics.AddOtlpExporter();
           });
```

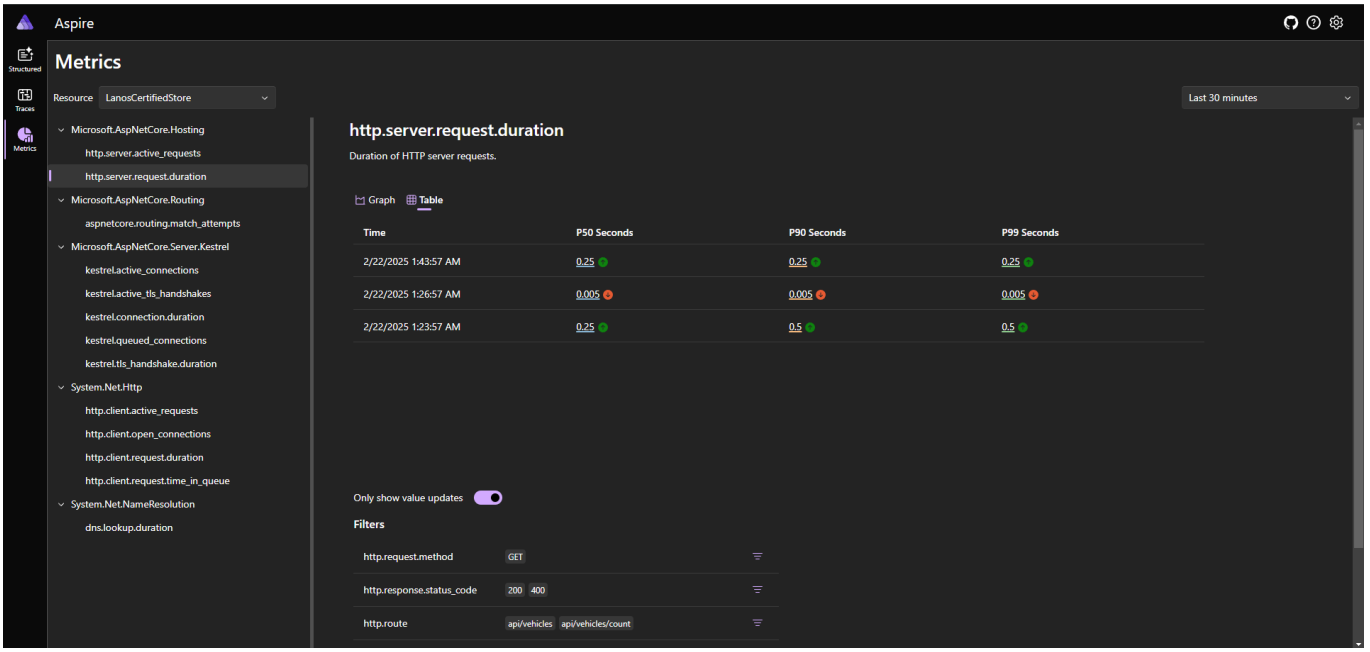


Рис. 2.5. Вигляд Aspire Dashboard з метриками

## 2.8. Впровадження CI/CD за допомогою GitHub Actions

GitHub Actions – це платформа для безперервної інтеграції та безперервної доставки (CI/CD), яка дозволяє автоматизувати конвеєр збірки, тестування та розгортання. Вона дозволяє створювати робочі процеси, які збирають і тестують кожен запит до вашого сховища, або розгортати об'єднані запити у виробництво.

GitHub Actions виходить за рамки DevOps і дозволяє запускати робочі процеси, коли у вашому сховищі відбуваються інші події. Наприклад, ви можете запустити робочий процес для автоматичного додавання відповідних міток щоразу, коли хтось створює новий випуск у вашому сховищі. GitHub надає віртуальні машини Linux, Windows і macOS для запуску робочих процесів [24].

Оскільки в даному проєкті для контролю версій використовується платформа GitHub, логічним, ефективним та зручним рішенням для реалізації CI/CD конвеєра є використання вбудованого сервісу **GitHub Actions**. GitHub Actions дозволяє визначати, автоматизувати та виконувати робочі процеси (workflows) безпосередньо в репозиторії GitHub, реагуючи на різноманітні події, такі як push-операції, створення pull request-ів, відкриття Issues тощо.

Для забезпечення стабільності та надійності основної гілки розробки (master), в проєкті налаштовано політику захисту гілки за допомогою інструментів GitHub. Згідно з цією політикою, прямі зміни до гілки master заборонені. Внесення будь-яких змін до основної гілки можливе виключно через механізм Pull Request (PR) – запитів на інтеграцію змін з інших гілок.

Ключовим елементом цієї стратегії захисту гілки master є вимога успішного проходження певних автоматизованих перевірок перед тим, як Pull Request може бути об'єднаний (merged) у гілку master. Ці перевірки включають, зокрема, виконання автоматизованих тестів. Якщо автоматизовані тести, що виконуються як частина CI конвеєра, завершуються з помилкою, система контролю версій автоматично блокує можливість об'єднання відповідного Pull Request, сигналізуючи команді розробки про наявність проблем у запропонованих змінах. Це гарантує, що в гілку master потрапляє тільки той код, який відповідає визначеним стандартам якості та пройшов необхідні автоматизовані перевірки.

Для автоматичного виконання необхідних тестів у проєкті реалізовано робочий процес GitHub Actions, конфігурація якого представлена нижче у форматі YAML:

Лістинг 2.16. Вміст файлу .github/workflows/main.yml

```yaml
name: Run Tests 🚀

on:
  workflow_dispatch:
  push:
    branches:
      - master

jobs:
  run-tests:
    runs-on: ubuntu-latest
```

```
    steps:
      - uses: actions/checkout@v3

      - name: Setup .NET
        uses: actions/setup-dotnet@v3
        with:
          dotnet-version: '8.0.107'

      - name: Restore
        run: dotnet restore ./LanosCertifiedStore.sln

      - name: Build
        run: dotnet build ./LanosCertifiedStore.sln --no-restore

      - name: Test
        run: dotnet test ./LanosCertifiedStore.sln --no-build
```

Цей файл конфігурації визначає робочий процес GitHub Actions під назвою “Run Tests 🚀”. Він налаштований на виконання за двома основними тригерами (on):

1. **Вручну через інтерфейс GitHub (workflow_dispatch)**: Дозволяє запустити цей робочий процес вручну безпосередньо з вкладки Actions у репозиторії GitHub.
2. **Автоматично при кожній push-операції до гілки master (push: branches: [master])**: Цей тригер запускає виконання тестів щоразу, коли зміни відправляються (pushed) до основної гілки master.

Робочий процес складається з одного завдання (jobs: run-tests), яке налаштовано на виконання на віртуальній машині під управлінням останньої версії операційної системи Ubuntu (runs-on: ubuntu-latest), що надається GitHub як раннер.

Завдання включає послідовність кроків, що виконуються послідовно:

- **actions/checkout@v3**: Використання стандартної дії GitHub Actions для отримання актуальної версії коду репозиторію на віртуальну машину раннера.
- **Setup .NET**: Цей крок використовує дію actions/setup-dotnet@v3 для встановлення необхідної версії .NET SDK (вказано 8.0.107), яка потрібна для подальшої збірки та виконання тестування проєкту.
- **Restore**: Виконує команду dotnet restore для відновлення всіх залежностей проєкту з NuGet-пакетів, використовуючи вказаний шлях до файлу рішення (./LanosCertifiedStore.sln).
- **Build**: Виконує команду dotnet build для компіляції проєкту. Використання прапорця --no-restore вказує, що відновлення залежностей вже виконано на попередньому кроці, і його не потрібно повторювати.
- **Test**: Запускає виконання всіх автоматизованих тестів, присутніх у рішенні, за допомогою команди dotnet test. Прапорець --no-build вказує, що перед запуском тестів не потрібно виконувати повторну збірку проєкту, оскільки вона вже була виконана на попередньому кроці.

Таким чином, хоча наведений конфігураційний файл безпосередньо описує виконання тестів при внесенні змін до гілки **master**, він є ключовим елементом для реалізації політики захисту гілки master. У поєднанні з налаштуваннями Pull Request-ів на GitHub, успішне завершення цього робочого процесу стає обов'язковою перевіркою статусу, без проходження якої неможливе об'єднання Pull Request у основну гілку. Це гарантує, що в гілці master завжди знаходиться стабільний код, який пройшов всі необхідні автоматизовані перевірки якості, що є фундаментальним принципом CI/CD.

### 2.9. Інтеграція системи управління ідентифікацією та доступом Keycloak

У будь-якому сучасному веб-застосунку, особливо такому як маркетплейс, де взаємодіють різні типи користувачів, безпека та управління доступом є критично важливими аспектами. Для реалізації надійних, масштабованих та стандартизованих механізмів аутентифікації та авторизації в даному проєкті було обрано **Keycloak** –

потужну, відкриту систему управління ідентифікацією та доступом (Identity and Access Management, IAM).

**Keycloak** – це програмний продукт з відкритим вихідним кодом, що дозволяє здійснювати єдиний вхід з управлінням ідентифікацією та доступом до сучасних додатків і сервісів. До квітня 2023 року цей проект спільноти WildFly перебував під керівництвом компанії Red Hat, яка використовувала його як попередній проект для своєї збірки Red Hat для Keycloak. У квітні 2023 року Keycloak був подарований CNCF і приєднався до фонду як інкубаційний проект.

Keycloak підтримує різні протоколи, такі як OpenID, OAuth версії 2.0 та SAML, і надає такі функції, як управління користувачами, двофакторна аутентифікація, управління дозволами та ролями, створення токен-сервісів тощо.

Використання Keycloak дозволяє централізувати управління користувачами, їхніми ролями та дозволами, ефективно знімаючи цю складну логіку з основного застосунку. Це значно підвищує рівень безпеки системи, спрощує процес розробки та дозволяє легко інтегрувати розширені функції, такі як єдиний вхід (Single Sign-On, SSO), федерація ідентифікацій та багатофакторна аутентифікація.

Реалізація інтеграції Keycloak у проєкті передбачала наступні ключові кроки:

1. **Створення та налаштування окремого Realm**: Для забезпечення повної ізоляції та гнучкості управління ідентифікаціями в Keycloak, для потреб маркетплейсу вживаних автомобілів було створено окремий Realm. Realm в Keycloak функціонує як ізольований простір для управління користувачами, клієнтами (додатками, що взаємодіють з Keycloak), ролями, групами та конфігураціями аутентифікації. В рамках цього realm'у було визначено та налаштовано клієнтів для взаємодії з API застосунку, створено необхідні ролі (наприклад, "admin", "user", "seller") та сконфігуровано стандартні потоки аутентифікації (наприклад, Authorization Code Flow для веб-додатків). Це забезпечує чітке розмежування простору ідентифікацій цього проєкту від інших потенційних сервісів, що можуть використовувати той самий екземпляр Keycloak

2. **Експорт Realm для перевикористання з Docker Compose**: Для зручності розгортання та забезпечення відтворюваності середовища розробки та тестування, налаштований realm було експортовано у зовнішній файл конфігурації (зазвичай у форматі JSON). Цей файл потім інтегрується в конфігурацію Docker Compose проєкту. Такий підхід дозволяє автоматично імпортувати попередньо налаштований realm при запуску Keycloak як Docker-контейнера в локальному середовищі або в CI/CD конвеєрі. Це значно спрощує початкове налаштування середовища для нових розробників, забезпечуючи, що Keycloak завжди буде запущений з коректними ідентифікаціями, користувачами та конфігураціями без необхідності ручних налаштувань. Завдяки цьому забезпечується консистентність середовищ.

3. **Інтеграція з ASP.NET Core застосунком**: Після налаштування Keycloak як окремого сервісу, було виконано інтеграцію серверної частини маркетплейсу, розробленої на базі ASP.NET Core, з Keycloak. Це включало конфігурацію застосунку як OpenID Connect клієнта, що дозволяє делегувати процеси аутентифікації користувачів безпосередньо Keycloak. Застосунок налаштований на автоматичну валідацію JWT-токенів (JSON Web Tokens), отриманих від Keycloak. Ця валідація включає перевірку підпису токена, його терміну дії та аудиторії. На основі інформації, що міститься в токенах (зокрема, ролей користувача), застосунок ефективно управляє доступом до API-ендпоінтів, застосовуючи політики авторизації. Такий підхід забезпечує надійну, масштабовану та гнучку систему безпеки, де бізнес-логіка аутентифікації та авторизації залишається централізованою у Keycloak, а застосунок лише перевіряє права доступу на основі отриманих та валідованих токенів.

Загалом, інтеграція Keycloak для управління ідентифікацією та доступом значно спрощує розробку функціоналу, пов'язаного з користувачами, підвищує загальний рівень безпеки застосунку та забезпечує гнучкість для майбутніх

розширень, таких як інтеграція з корпоративними системами ідентифікації або додавання нових провайдерів соціальної аутентифікації.

**Висновки до розділу 2**

У другому розділі було реалізовано комплекс прикладних заходів, спрямованих на практичне втілення теоретичних засад проєктування бази даних та архітектури серверної частини маркетплейсу вживаних автомобілів.

Перш за все, виконано повноцінне проєктування бази даних із побудовою структури сутностей, їх зв'язків та атрибутів відповідно до вимог предметної області. Реалізація бази даних була здійснена з використанням підходу Code-First на платформі Entity Framework Core, що забезпечує зручність підтримки, рефакторингу та автоматичного мігрування структури до фізичного сховища.

Інфраструктура проєкту розгорнута за допомогою Docker та Docker Compose, що дозволяє забезпечити ізольованість середовищ та спрощене масштабування компонентів. Крім того, впроваджено Data Seeding для ініціалізації системи демонстраційними даними.

Для підвищення надійності та дотримання архітектурних стандартів, у проєкті реалізовано набір архітектурних тестів, що контролюють відповідність іменувальних конвенцій, залежностей між проєктами, доступності атрибутів та структурних вимог. Додатково інтегровано Health Checks та OpenTelemetry для моніторингу стану застосунку, збору метрик продуктивності та виявлення потенційних проблем у роботі системи.

Окрему увагу приділено впровадженню системи аутентифікації та авторизації на основі Keycloak, яка забезпечує централізоване управління ідентичністю користувачів. Завдяки цій інтеграції реалізовано підтримку механізмів OAuth 2.0 та OpenID Connect, що дозволяє надійно контролювати доступ до захищених ресурсів маркетплейсу. Keycloak забезпечує гнучке керування ролями, дозволами та політиками безпеки, що значно підвищує загальний рівень захищеності системи, а

також спрощує масштабування та підтримку механізмів контролю доступу в майбутньому.

Завершальним етапом стала реалізація повноцінного CI/CD конвеєра на GitHub Actions, який автоматизує процеси тестування, збірки та розгортання, підвищуючи загальну надійність та ефективність життєвого циклу застосунку.

Таким чином, прикладна частина проєкту демонструє високий рівень інтеграції сучасних технологій та підходів у сфері розробки веб-застосунків, підтверджуючи доцільність обраної архітектурної моделі, ефективність організації даних та технічну готовність системи до подальшого масштабування.

# ВИСНОВКИ

У межах виконання кваліфікаційної роботи було здійснено повний цикл проєктування та реалізації технічної основи маркетплейсу вживаних автомобілів, що охоплює як фундаментальні теоретичні аспекти, так і практичну імплементацію із застосуванням сучасних інструментів розробки та супроводу ПЗ.

На першому етапі проведено всебічний аналіз предметної області: визначено специфіку ринку вживаних автомобілів в Україні, ключові особливості таких платформ та сформульовано чіткі функціональні, нефункціональні й безпекові вимоги до інформаційної системи. Це дозволило задати чіткі орієнтири для подальшого технічного проєктування.

Було обґрунтовано вибір реляційної моделі баз даних та обрано PostgreSQL як надійну, масштабовану та функціонально насичену СУБД. Проведено нормалізацію структури, спроєктовано ER-діаграми, визначено ключові сутності та зв’язки, а також враховано продуктивнісні аспекти через оптимізацію індексації, агрегацій та можливу денормалізацію при потребі.

На практичному рівні реалізація бази даних здійснена за підходом Code-First з використанням Entity Framework Core, що забезпечило тісну інтеграцію моделі домену з логікою додатку. Конфігурації сутностей були виконані через Fluent API, а система отримала стартовий набір демонстраційних даних за допомогою Data Seeding.

Особливу увагу приділено побудові надійної, масштабованої архітектури серверної частини. Обрано шарову архітектуру з дотриманням принципів Clean Architecture, що забезпечує модульність, тестованість та легкість супроводу. Розроблені архітектурні тести автоматично перевіряють структуру коду, іменувальні конвенції, залежності між проєктами та відповідність визначеним контрактам.

Для забезпечення стабільності й контролю над працездатністю системи впроваджено Health Checks, а для збору метрик і діагностики — OpenTelemetry, що створює основу для надійного моніторингу в продуктивному середовищі.

З метою безпеки, масштабованості автентифікації та авторизації, інтегровано систему Keycloak з підтримкою OAuth 2.0 та OpenID Connect, що надало проєкту централізовану модель керування доступом і ролями користувачів.

Завершальним етапом стала побудова CI/CD-конвеєра на базі GitHub Actions — він автоматизує процеси тестування, збірки та деплою, забезпечує цілісність основної гілки репозиторію та підвищує якість життєвого циклу застосунку.

Таким чином, у межах роботи було реалізовано повноцінну технічну основу сучасного онлайн-маркетплейсу, з урахуванням реальних вимог до продуктивності, масштабованості, безпеки та зручності підтримки. Запропоноване рішення не лише підтверджує доцільність обраної моделі, але й створює потенціал для подальшого розвитку системи в умовах реального комерційного середовища.

# СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ


1. УкрАвтоПром. Telegram-канал. URL: https://t.me/ukrautoprom (дата звернення: 18.11.2024 р.).
2. What is Database? | Oracle. URL: https://www.oracle.com/database/what-is-database (дата звернення: 18.11.2024 р.).
3. What is Relational Database? | IBM. URL: https://www.ibm.com/topics/relational-databases (дата звернення: 18.11.2024 р.).
4. What is NoSQL? Database Explained | Google Cloud. URL: https://cloud.google.com/discover/what-is-nosql (дата звернення: 18.11.2024 р.).
5. Database – Wikipedia. URL: https://en.wikipedia.org/wiki/Database (дата звернення: 18.11.2024 р.).
6. Relational database - Wikipedia. URL: https://en.wikipedia.org/wiki/Relational_database (дата звернення: 18.11.2024 р.).
7. Introduction of Database Normalization – GeeksForGeeks – URL: https://www.geeksforgeeks.org/introduction-of-database-normalization/ (дата звернення: 18.11.2024 р.).
8. Denormalization in Databases – GeeksForGeeks – URL: https://www.geeksforgeeks.org/denormalization-in-databases/ (дата звернення: 18.11.2024 р.).
9. Documentation and Visualisation of Workflows for Effective Communication, Collaboration and Publication – URL: https://ijdc.net/index.php/ijdc/article/view/12.1.72/468 (дата звернення: 21.11.2024 р.).
10. Introduction | dbdiagram Docs – URL: https://docs.dbdiagram.io/ (дата звернення 21.11.2024 р.).
11. PostgreSQL: About – URL: https://www.postgresql.org/about/ (дата звернення 21.11.2024 р.).

12. PostgreSQL: Documentation: 17: 37.1 Overview of Trigger Behavior – URL: https://www.postgresql.org/docs/17/trigger-definition.html (дата зверенення 22.11.2024 р.).
13. What is Code-First? – URL: https://www.entityframeworktutorial.net/code-first/what-is-code-first.aspx (дата звернення: 24.11.2024 р.).
14. What is Entity Framework? – URL: https://www.entityframeworktutorial.net/entityframework6/what-is-entityframework.aspx (дата звернення: 24.11.2024 р.).
15. What Is Containerization? What Is Orchestration? - Alibaba Cloud – URL: https://www.alibabacloud.com/en/knowledge/what-is-containerization?_p_lc=1 (дата звернення: 24.11.2024 р.).
16. What is Docker? | Docker Docs – URL: https://docs.docker.com/get-started/docker-overview/ (дата звернення 24.11.2024 р.).
17. Docker Compose | Docker Docs – URL: https://docs.docker.com/compose/ (дата звернення 25.11.2024 р.).
18. Enforcing Software Architecture With Architecture Tests. URL: https://www.milanjovanovic.tech/blog/enforcing-software-architecture-with-architecture-tests (дата звернення: 23.02.2025 р.).
19. GitHub - BenMorris/NetArchTest: A fluent API for .Net that can enforce architectural rules in unit tests. URL: https://github.com/BenMorris/NetArchTest (дата звернення: 05.05.2025 р.).
20. Health Checks. URL: https://alhardy.github.io/app-metrics-docs/getting-started/health-checks/index.html (дата звернення: 03.02.2025 р.).
21. What is telemetry and how does it work? URL: https://www.techtarget.com/whatis/definition/telemetry (дата звернення: 13.02.2025 р.).

22. What is OpenTelemerty? | OpenTelemetry. URL: https://opentelemetry.io/docs/what-is-opentelemetry/ (дата звернення: 18.02.2025)
23. What is CI/CD? | GitHub. URL: https://github.com/resources/articles/devops/ci-cd (дата звернення: 13.05.2025 р.).
24. Understanding GitHub Actions - GitHub Docs. URL: https://docs.github.com/en/actions/about-github-actions/understanding-github-actions (дата звернення: 14.05.2025 р.).
25. Keycloak - Wikipedia. URL: https://en.wikipedia.org/wiki/Keycloak (дата звернення: 15.05.2025 р.).
26. CQRS Pattern with MediatR. URL: https://www.milanjovanovic.tech/blog/cqrs-pattern-with-mediatr (дата звернення: 14.02.2025 р.).
27. Database relationships - IBM Documentation. URL: https://www.ibm.com/docs/en/eamfoc/7.6.0?topic=structure-database-relationships (дата звернення: 18.11.2024 р.).