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: «**Розробка 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»
Мельничук Давид Олександрович

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: Розробка серверної частини маркетплейсу вживаних автомобілів

Автор: Мельничук Давид Олександрович

Науковий керівник: Мацевич Д. В., викладач-стажист кафедри ІТАД

Захищена «___» _____ 2025 року.

Пояснювальна записка до кваліфікаційної роботи: *88 сторінок, 13 рисунків, 5 таблиць, 4 додатки, 24 джерела*

Ключові слова: *маркетплейс, REST API, бекенд, безпека, архітектура*

Короткий зміст праці:

Завданням проєкту була розробка серверної частини маркетплейсу вживаних автомобілів "Kolesko". Основною вимогою до додатку було створення REST API, що забезпечує функціонал пошуку та фільтрації автомобілів та гарантує безпеку даних. Цільова аудиторія додатку - покупці та продавці транспортних засобів на українському ринку вживаних автомобілів. Для реалізації було обрано .NET 8 та PostgreSQL для бекенду та бази даних відповідно, а також Keycloak для системи аутентифікації користувачів. Під час розробки було використано найкращі практики для кожної з представлених технологій, архітектурний підхід Clean Architecture та патерн Command Query Responsibility Segregation (CQRS).

_____

ANNOTATION

of qualification paper

for bachelor's degree

Theme: **Used vehicles marketplace application's backend development**

Author: ***Davyd Melnychuk***

Scientific supervisor: Matsevych D., trainee lecturer at the ITDA Department

Defended «___» _____ of 2025.

Explanatory note to the qualification work: *88 pages, 13 images, 5 tables, 4 attachments, 24 sources*

Keywords: *marketplace, REST API, backend, security, architecture*

Summary of the work:

The task of the project was to develop the server side of the Kolesko used car marketplace. The main requirement for the application was the creation of a REST API which provides the functionality of searching and filtering vehicles and guarantees data security. The target audience of the application is buyers and sellers of cars in the Ukrainian used vehicles market. For implementation, .NET 8 and PostgreSQL were chosen for the backend and database respectively, along with Keycloak for the user authentication system. During the development, the best practices for each of the presented technologies, the Clean Architecture architectural approach, and the Command Query Responsibility Segregation (CQRS) pattern were used.

_____

ЗМІСТ

ВСТУП

Актуальність теми. В умовах стрімкої діджиталізації та трансформації традиційних моделей торгівлі особливої значущості набуває розробка інноваційних технологічних рішень у сфері електронної комерції, зокрема для ринку вживаних автомобілів. Створення ефективних та функціональних онлайн-платформ для автомобільної торгівлі є важливим завданням, що сприяє розвитку ринку та задоволенню зростаючих потреб користувачів. В умовах інтенсивної конкуренції та збільшення кількості запитів від потенційних клієнтів щодо впровадження розширеного функціоналу, необхідно розробляти рішення, які забезпечать:

- зручність використання через інтуїтивний інтерфейс та оптимізовану архітектуру, що дозволяє користувачам швидко освоїти платформу та ефективно виконувати необхідні операції;
- гнучкість інструментів для пошуку найпривабливіших пропозицій як для покупців, так і для продавців, забезпечуючи максимальну відповідність результатів запитам користувачів;
- надійний захист чутливої інформації від несанкціонованого доступу, системних збоїв чи зловмисних дій третіх осіб, що є критично важливим для збереження довіри користувачів;

Сучасні тенденції розвитку електронної комерції в автомобільному секторі України демонструють зростаючий попит на цифрові рішення, які оптимізують процес пошуку та придбання транспортних засобів. Потенційні покупці прагнуть отримати максимально повну інформацію про автомобілі, що їх цікавлять, а продавці шукають ефективні канали для реалізації своїх пропозицій. За таких умов, розробка спеціалізованого маркетплейсу з розширеним функціоналом та високим рівнем безпеки є надзвичайно актуальною.

Проблематика. Аналіз існуючих рішень на українському ринку автомобільних маркетплейсів виявляє ряд суттєвих недоліків, пов'язаних із зручністю використання, доступним функціоналом, надійністю та безпекою тощо. Значна кількість платформ

для реалізації транспортних засобів демонструють недостатню ефективність механізмів інтелектуального пошуку та обмежені можливості щодо фільтрації результатів. Зокрема, спостерігається недостатня точність у формуванні релевантних пропозицій, що негативно впливає на користувацький досвід та знижує загальну ефективність процесу пошуку. Також відсутність комплексного підходу до тестування наявного функціоналу досить часто призводить до виникнення критичних помилок у роботі існуючих платформ. Це, в свою чергу, зумовлює зниження довіри користувачів та їхньої готовності використовувати подібні сервіси.

Онлайн-маркетплейс "Kolesko", а саме серверна частина, що представляє собою API розроблений за REST підходом, спрямований на вирішення цих проблем шляхом:

- забезпечення високоефективних алгоритмів пошуку та фільтрації автомобілів, що враховують різноманітні користувацькі параметри;
- впровадження надійної системи аутентифікації та авторизації користувачів на базі Keycloak, що забезпечує високий рівень захисту персональних даних та попереджає несанкціонований доступ;
- гарантування масштабованості та гнучкості системи для легкого розширення інфраструктури та інтеграції з іншими сервісами, що є важливим аспектом для розвитку платформи відповідно до зростаючих потреб ринку та запитів користувачів.

Розробка такого комплексного рішення потребує глибокого аналізу бізнес-процесів, пов'язаних з продажем та купівлею автомобілів, а також впровадження сучасних технологічних підходів та методологій, що забезпечать стабільність та ефективність функціонування платформи.

Мета дослідження. Метою кваліфікаційної роботи є розробка повнофункціонального серверного рішення для маркетплейсу вживаних автомобілів "Kolesko", що включає проектування та реалізацію REST API з впровадженням комплексної системи пошуку оголошень, механізму релевантних пропозицій та

багаторівневої методології тестування. Це дозволить створити надійну та функціональну платформу для торгівлі вживаними автомобілями, яка відповідатиме сучасним вимогам ринку та забезпечить високу ефективність взаємодії користувачів із системою.

Досягнення поставленої мети передбачає не лише технічну реалізацію системи, але й врахування особливостей бізнес-процесів та користувацьких потреб, що є критично важливим для створення конкурентоспроможного продукту на ринку автомобільних маркетплейсів. Орієнтація на користувацький досвід та забезпечення високої якості функціонування системи є ключовими аспектами розробки, що визначають загальну успішність проекту.

Основні завдання кваліфікаційної роботи включають:

1. Проведення аналізу бізнес-аспектів предметної області для їх подальшої агрегації під час проєктування API, включаючи вивчення специфіки ринку вживаних автомобілів, аналіз поведінки потенційних користувачів та ідентифікацію ключових бізнес-процесів;
2. Розробка архітектури API, що включає:
    - опис основних ендпоінтів та їх документування з детальним визначенням типів запитів, потенційних відповідей та обробки помилок;
    - забезпечення масштабованості та розширюваності системи для майбутнього нарощування функціоналу та адаптації до зростаючих потреб користувачів;
    - імплементацію системи безпеки та аутентифікації на базі Keycloak, що забезпечить надійний захист даних та управління доступом користувачів до різних функцій платформи;
    - забезпечення коректної обробки та логування помилок для ефективного моніторингу системи та швидкого виявлення та виправлення потенційних проблем;

3. Розробка та імплементація механізму пошуку оголошень з алгоритмами забезпечення релевантності результатів, що враховує різноманітні користувацькі параметри для формування найбільш відповідних пропозицій;
4. Реалізація основних функціональних можливостей бекенду відповідно до визначених бізнес-правил, включаючи функції створення та управління оголошеннями, механізми пошуку та фільтрації, керування списком бажаних пропозицій користувача, розміщених на платформі тощо;
5. Імплементація власного централізованого сервісу аутентифікації для можливості гнучкої роботи з обліковими записами користовачів, що забезпечує високий рівень захисту персональних даних та допомагає запобігти несанкціонованому доступу до них;
6. Створення та впровадження комплексної системи тестування, що включає:
    - модульне тестування (unit-testing) для верифікації коректності роботи окремих компонентів системи та своєчасного виявлення потенційних проблем;
    - інтеграційне тестування (integration-testing) для валідації взаємодії між функціональними модулями платформи та забезпечення їхньої узгодженої роботи;

Об'єктом дослідження є система маркетплейсу вживаних автомобілів, що представляє собою онлайн-майданчик для реалізації пошуку та продажу транспортних засобів, включаючи всі її компоненти, відповідальні за забезпечення основних функцій та взаємодію з користувачами.

Предметом дослідження виступають підходи та технологічні рішення щодо розробки та проєктування бекенд частини застосунку, а саме: механізмів пошуку оголошень, керування списком бажаних пропозицій користувача, алгоритмів забезпечення релевантності результатів, методів забезпечення безпеки даних та організації ефективної взаємодії між компонентами системи. Особлива увага

приділяється дослідженню методологій впровадження надійної системи аутентифікації та авторизації, а також підходів до оптимізації алгоритмів пошуку та фільтрації даних для забезпечення максимальної відповідності результатів запитам користувачів.

РОЗДІЛ 1
ТЕОРЕТИЧНІ ОСНОВИ РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ ДЛЯ МАРКЕТПЛЕЙСУ ВЖИВАНИХ АВТОМОБІЛІВ

1.1. Дослідження трендів вторинного автомобільного ринку України

Відповідно до аналітичного дослідження, опублікованого в жовтні 2024 року, український ринок вживаних автомобілів демонструє декілька визначальних тенденцій, які варті детального розгляду. Дослідження висвітлює ключові напрямки розвитку вторинного автомобільного ринку, серед яких особливо помітними є[1]:

- **Диверсифікація платформ продажу транспортних засобів.** Спостерігається виразний перехід від класичних онлайн-майданчиків до альтернативних каналів реалізації автомобілів, зокрема: о групи та чати в соціальних мережах і месенджерах, присвячені купівлі-продажу вживаних авто; о спеціалізовані веб-ресурси від автодилерів; о традиційні офлайн автомобільні ринки.
- **Зменшення активності на традиційних маркетплейсах.** Статистичні дані підтверджують поступове скорочення кількості розміщених оголошень: якщо у вересні 2024 року було зафіксовано 214,5 тисяч оголошень, то в жовтні цей показник знизився до 210,1 тисяч, що відображає зменшення на 2,1%. Така динаміка може свідчити про відсутність достатньо гнучких рішень, які б відповідали швидкозмінним потребам користувачів. Повну картину змін у кількості оголошень та їхньої сумарної вартості за період жовтень 2023 — жовтень 2024 року представлено на графіку нижче.

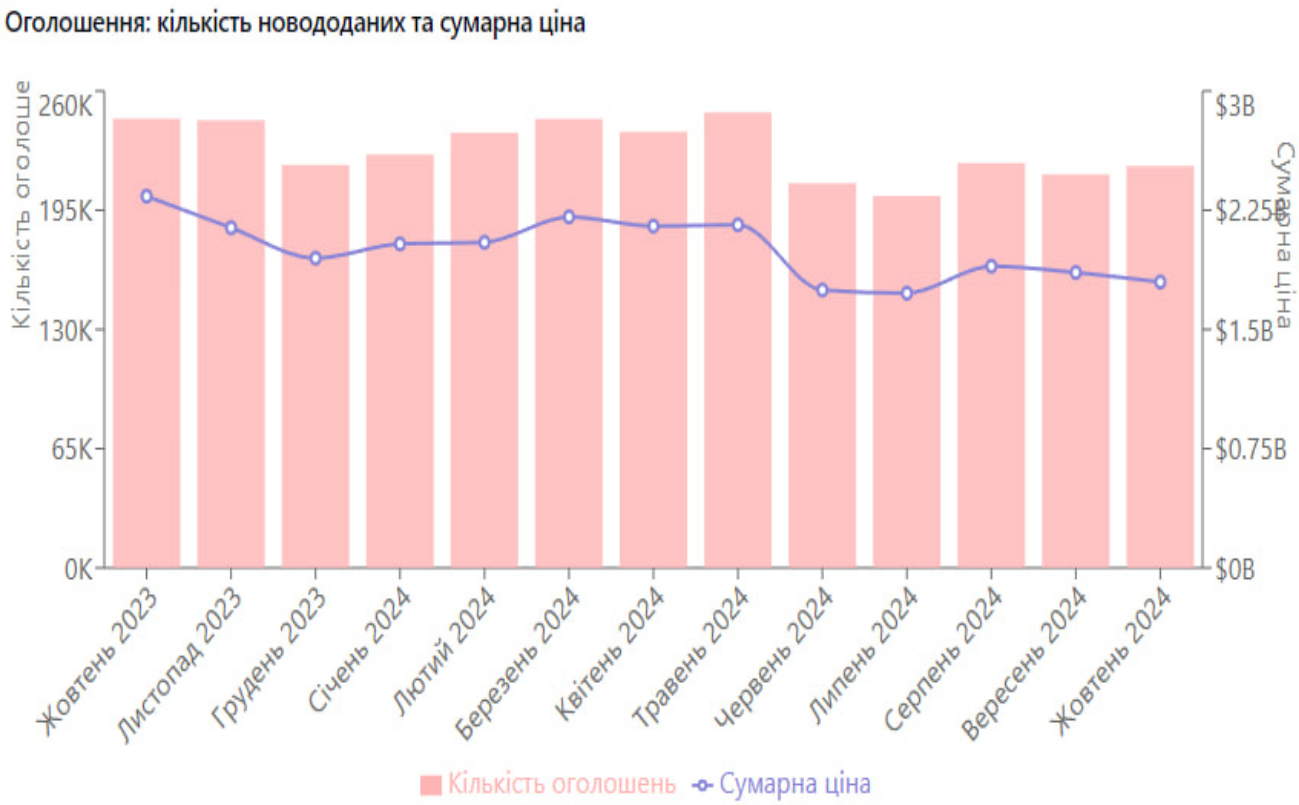

Рис. 1.1. Графік змін кількості оголошень та їхньої сумарної ціни в період з жовтня 2023 року по жовтень 2024 року. Джерело: [1]

- **Суттєве зниження купівельної спроможності населення.** Починаючи з 2022 року, в Україні, як і в усьому світі, спостерігаються кризові явища, викликані пандемією COVID-19, що значно вплинули на фінансові можливості споживачів. Ситуація ускладнилася через повномасштабне російське вторгнення, яке спричинило зменшення доходів значної частини населення, посилення інфляційних процесів та інші негативні економічні явища. Це змушує громадян заощаджувати та контролювати витрати. На автомобільному ринку ця тенденція проявляється у зростанні попиту на вживані авто замість нових. При цьому, навіть на вторинному ринку покупці демонструють обмежені фінансові можливості. Згідно з проаналізованими даними, середня вартість пропозиції вживаного автомобіля протягом досліджуваного періоду знизилася з $5700-$6000 (на початку та в

середині періоду) до $5500 в останні місяці. Ця тенденція представлена на графіку нижче.

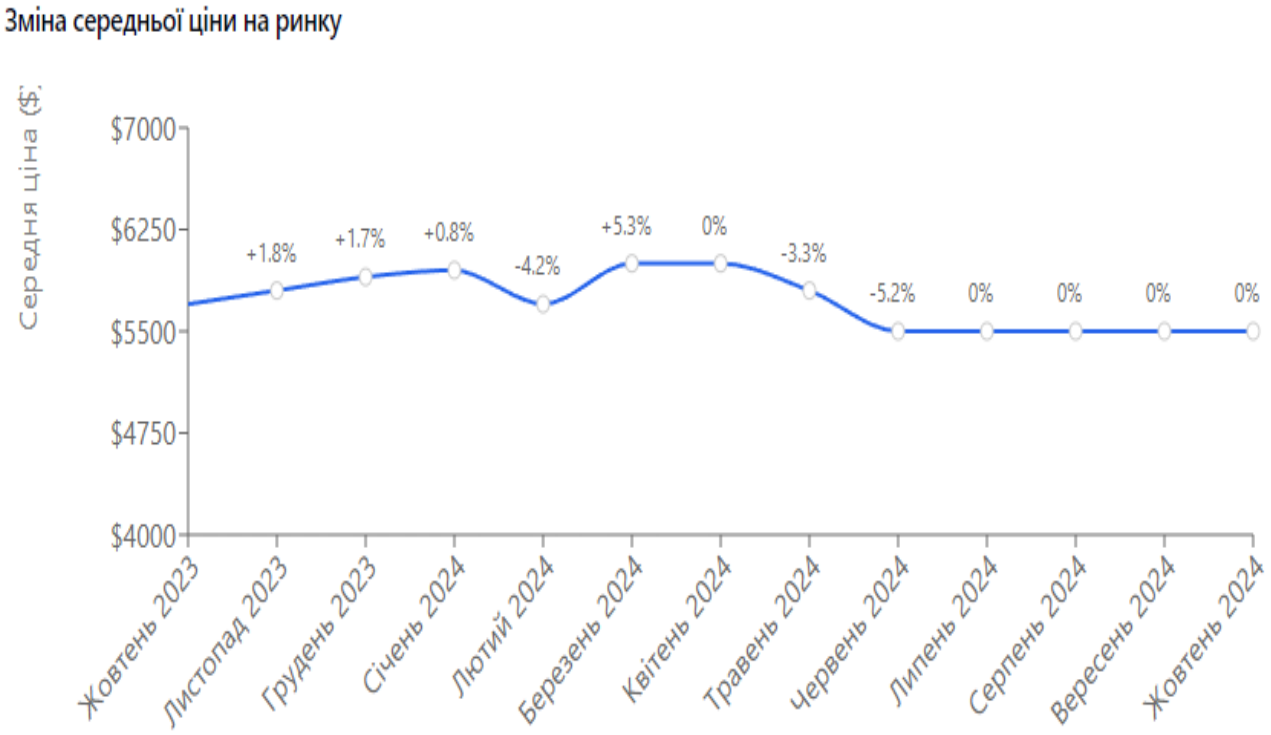

Рис. 1.2. Графік зміни середньої ціни, вказаної в оголошенні, в період з жовтня 2023 року по жовтень 2024 року. Джерело: [1]

З огляду на поточну економічну ситуацію, можна прогнозувати подальше поступове зниження середньої вартості пропозицій, оскільки значного та швидкого покращення макроекономічних показників в Україні наразі не передбачається через триваючий військовий конфлікт та економічну нестабільність. Однак, для розробників онлайн-платформ з продажу вживаних автомобілів така ситуація відкриває можливості для залучення більшої кількості користувачів, які активно шукають доступні варіанти в межах свого обмеженого бюджету.

- Підвищення інтересу до екологічних транспортних засобів. В Україні чітко простежується зростання популярності гібридних та електричних автомобілів. Це зумовлено як глобальними екологічними тенденціями щодо зменшення викидів CO2, так і практичними перевагами для власників — нижчими експлуатаційними витратами. Додатковим стимулом слугують державні програми підтримки придбання екологічного транспорту. Ця зміна споживчих пріоритетів створює нові можливості для онлайн-платформ, які можуть розширити спектр послуг, орієнтуючись на потенційних покупців екологічно чистих автомобілів.

Підсумовуючи аналіз українського ринку вживаних автомобілів, слід зазначити, що економічні виклики та обмежені фінансові можливості споживачів посилюють інтерес до вторинного ринку транспортних засобів. Одночасно, значне зростання сегменту екологічного транспорту відображає глобальні екологічні тенденції. Поєднання цих факторів формує потребу в сучасних онлайн-майданчиках для торгівлі автомобілями, які повинні забезпечувати зручний пошук за різноманітними критеріями, включаючи екологічні показники. Створення спеціалізованого маркетплейсу з такими можливостями є актуальним рішенням, що відповідає сучасним вимогам українського автомобільного ринку.

1.2. Архітектурні рішення для проектування та створення серверної частини онлайн-маркетплейсу вживаних автомобілів

Розробка надійної та ефективної архітектури серверної частини є фундаментальним етапом у створенні сучасних розподілених застосунків, зокрема онлайн-платформ електронної комерції, до яких належить і маркетплейс вживаних автомобілів "Kolesko". Серверна частина системи виконує ключові функції, включаючи обробку запитів від клієнтських застосунків, реалізацію бізнес-логіки предметної області, управління даними та забезпечення безпеки транзакцій, надаючи доступ до свого функціоналу через програмний інтерфейс.

Програмний інтерфейс застосунку (API) як інтеграційний рівень

Програмний інтерфейс застосунку (Application Programming Interface – API) є формалізованим набором правил та специфікацій, що визначають спосіб взаємодії між різними програмними компонентами. У контексті веб-застосунків, API серверної частини слугує інтерфейсом для комунікації з клієнтськими додатками або іншими сервісами. Цей інтерфейс дозволяє здійснювати обмін даними за моделлю "запит-відповідь", забезпечуючи необхідний рівень абстракції та сприяючи незалежному розвитку компонентів системи. Правильний вибір архітектурного стилю API є критично важливим, оскільки він впливає на керованість, масштабованість та потенціал інтеграції серверної частини з різноманітними зовнішніми системами. Ефективне використання сторонніх API, що відповідають потребам проєкту, може значно прискорити процес розробки та підвищити загальну ефективність рішення порівняно з імплементацією функціоналу з нуля.

Аналіз архітектурних стилів для реалізації API серверної частини

При проектуванні архітектури серверної частини "Kolesko" виникла необхідність у визначенні найбільш релевантного архітектурного стилю для реалізації її API. Серед найбільш поширених архітектурних стилів для проектування мережевих API можна виділити:

- Representational State Transfer (REST)
- GraphQL
- Simple Object Access Protocol (SOAP)
- Remote Procedure Call (RPC) та його варіанти.

В рамках даного дослідження було проведено аналіз трьох основних альтернатив: REST, GraphQL та SOAP, з метою оцінки їхньої відповідності специфічним вимогам маркетплейсу вживаних автомобілів.

1. **REST (Representational State Transfer):** Представлений Роєм Філдінгом у 2000 році, REST є архітектурним стилем, який базується на використанні стандартів мережевих протоколів, зокрема HTTP, для створення масштабованих веб-сервісів[2]. Ключові принципи REST включають:
    - **Ресурсно-орієнтований підхід:** Системні компоненти розглядаються як ресурси, кожен з яких ідентифікується унікальним ідентифікатором (URI).
    - **Відсутність стану (Stateless):** Кожен запит від клієнта містить всю необхідну для його обробки інформацію, сервер не зберігає даних про стан клієнтської сесії між запитами. Цей принцип є фундаментальним для забезпечення масштабованості.
    - **Уніфікований інтерфейс:** Використання обмеженого набору стандартизованих операцій (HTTP-методів: GET, POST, PUT, PATCH, DELETE) для маніпуляцій з ресурсами.
    - **Маніпулювання представленнями ресурсів:** Взаємодія відбувається шляхом обміну представленнями ресурсів (наприклад, у форматах JSON або XML).
    - **Розділення ролей клієнта та сервера:** Чіткий поділ відповідальності, що дозволяє незалежний розвиток обох сторін системи.
2. **GraphQL:** Це мова запитів для API та середовище виконання, розроблене для ефективного отримання даних від сервера[3]. GraphQL надає клієнтам можливість точно визначати структуру та обсяг необхідних даних. Його характерні риси[4]:
    - **Гнучкість запитів:** Клієнти можуть запитувати лише конкретні поля та пов'язані ресурси, що мінімізує надлишковість даних.
    - **Типізована схема:** Використовує сильну систему типів для визначення структури API та валідації запитів перед виконанням.

- **Єдина точка входу:** Вся комунікація здійснюється через єдину кінцеву точку API.
- **Еволюційність схеми:** Дозволяє поступово змінювати та розширювати API без створення нових версій кінцевих точок.

3. **SOAP (Simple Object Access Protocol):** Це протокол обміну структурованою інформацією, що базується на форматі XML і широко використовується для реалізації веб-сервісів у корпоративному середовищі[5]. Ключові характеристики SOAP[5]:
   - **Незалежність від транспортного протоколу:** Може функціонувати поверх різних протоколів, таких як HTTP, SMTP, TCP.
   - **Незалежність від мови та платформи:** Розроблений для забезпечення інтероперабельності між системами, реалізованими на різних технологіях.
   - **Обмін повідомленнями на основі XML:** Повідомлення форматуються у стандартизовані XML-структури (конверти), що забезпечує чіткість формату даних.
   - **Вбудовані механізми обробки помилок:** Надає стандартизовані засоби для повідомлення про помилки через елемент fault.

Порівняльний аналіз архітектурних стилів API за ключовими критеріями

Для систематизації аналізу та прийняття обґрунтованого рішення щодо стилю API для серверної частини Kolesko, було розроблено серію порівняльних таблиць, що систематизують ключові аспекти розглянутих архітектурних стилів.

Таблиця 1.2.1.

Порівняння архітектурних стилів API за механізмами взаємодії з даними та гнучкістю

Назва аспекту	**REST**	**GraphQL**	**SOAP**
Механізм отримання даних	Базується на ідентифікації ресурсів та використанні стандартизованих HTTP-методів (GET)	Реалізується через єдину кінцеву точку з використанням специфічної мови запитів для вибору необхідних полів	Здійснюється шляхом надсилання структурованих XML-повідомлень через різні протоколи
Механізми маніпулювання даними	Використання HTTP-методів (POST, PUT, PATCH, DELETE) для CRUD-операцій над ресурсами	Реалізується через концепцію "мутацій" (mutations)	Використання XML-повідомлень для виклику операцій та обміну даними
Гнучкість запитів	Запити прив'язані до чітко визначених ресурсів та їх представлень; може призводити до надлишковості даних	Висока гнучкість у формуванні запитів, що дозволяє отримувати лише необхідні дані	Жорстка структура повідомлень згідно з визначеними WSDL-схемами

Джерело: [створено автором]

Продовжуючи порівняльний аналіз, розглянемо аспекти, пов'язані з процесом розробки та підтримки API.

Таблиця 1.2.2.

Порівняння архітектурних стилів API за аспектами розробки та підтримки

Назва аспекту	**REST**	**GraphQL**	**SOAP**

Стратегія версіонування	Часто реалізується через версіонування URI або використання заголовків	Управління версіями здійснюється переважно через еволюцію схеми API	Версіонування, як правило, застосовується на рівні всього сервісу
Інструментарій та екосистема	Зріла та широко представлена екосистема інструментів, бібліотек та документації	Динамічно зростаюча екосистема, орієнтована на специфіку GraphQL	Стабільна та розвинена екосистема, особливо в корпоративному сегменті

Джерело: [створено автором]

Наступний набір критеріїв стосується операційних характеристик API, включаючи його продуктивність та механізми обробки помилок і безпеки.

Таблиця 1.2.3.

Порівняння архітектурних стилів API за операційними характеристиками та безпекою

Назва аспекту	**REST**	**GraphQL**	**SOAP**
Продуктивність	Ефективний для стандартних ресурсних операцій; потенційна проблема N+1 запитів або over-fetching	Висока ефективність для складних запитів та зв'язаних даних; дозволяє уникнути over- та under-fetching	Ефективний для структурованого обміну даними; накладні витрати на обробку XML

Обробка помилок	Використання стандартних кодів стану HTTP та тіла відповіді для деталізації	Помилки інкапсулюються у структурованих об'єктах у відповіді запиту	Використання стандартизованого елемента `fault` у SOAP-конверті
Засоби безпеки	Базується на механізмах безпеки HTTP/TLS, а також реалізується через токени (Bearer тощо) або комплексні схеми аутентифікації (OAuth 2, OpenID Connect)	Використовує механізми безпеки транспортного рівня та автентифікації (часто сумісні з REST)	Має вбудовані розширення безпеки (WS-Security), що включають шифрування та цифрові підписи

Джерело: [створено автором]

Останнім критерієм порівняння є загальний рівень розповсюдженості кожного архітектурного стилю в сучасній розробці.

Таблиця 1.2.4.

Порівняння архітектурних стилів API за рівнем розповсюдженості

Назва аспекту	**REST**	**GraphQL**	**SOAP**
Рівень розповсюдженості	Найбільш поширений стиль API для веб-сервісів та мобільних додатків	Набуває значної популярності, особливо в сегментах, що потребують гнучкого отримання даних	Широко використовується у корпоративних та legacy-системах

Джерело: [створено автором]

Обґрунтування вибору архітектурних рішень для серверної частини "Kolesko"

На підставі проведеного аналізу архітектурних стилів API, а також з урахуванням специфічних вимог до функціональності, масштабованості та керованості системи онлайн-маркетплейсу вживаних автомобілів "Kolesko", було прийнято архітектурне рішення щодо використання стилю **REST для проектування API** серверної частини.

Таке рішення базується на ряді обґрунтованих переваг REST, які є вирішальними для даного типу застосунку:

- **Відповідність принципам простоти та керованості:** REST є відносно простим для розуміння, проектування та реалізації, що знижує складність розробки та подальшої підтримки системи.
- **Широка підтримка галузевих стандартів та інструментарію:** Наявність зрілої екосистеми інструментів, бібліотек та широкої спільноти розробників для RESTful API, що включає підтримку в обраному стеку технологій (.NET), значно прискорює процес імплементації.
- **Ефективність для взаємодії з ресурсними моделями:** Природа маркетплейсу, що оперує чітко визначеними ресурсами (оголошення, користувачі, автомобілі), ідеально відповідає ресурсно-орієнтованому підходу REST.
- **Природна підтримка масштабування:** Принцип відсутності стану (statelessness) в REST є фундаментальним для забезпечення горизонтального масштабування серверної частини під зростаюче навантаження без необхідності управління станом сесії на сервері.

Загальна архітектура серверної частини та її компоненти

Крім вибору стилю API, архітектура серверної частини "Kolesko" включає ряд принципових рішень щодо внутрішньої організації системи та вибору технологічного стеку, спрямованих на досягнення високої продуктивності, надійності, безпеки та зручності розробки:

- **Технологічна платформа:** Використання .NET забезпечує високий рівень продуктивності, стабільності та доступ до потужних бібліотек.
- **Організація коду:** Архітектура реалізована з урахуванням принципів чистої архітектури (Clean Architecture), модульності та розділення відповідальності, часто з використанням патернів, таких як Mediator, Command, CQRS тощо, що сприяють чистоті та керованості коду.
- **Шар доступу до даних:** Використання Entity Framework Core як об'єктно-реляційного мапера (ORM) для ефективної та безпечної взаємодії з реляційною базою даних PostgreSQL.
- **Система управління ідентифікацією та доступом:** Інтеграція з Keycloak як централізованим сервісом ідентифікації забезпечує надійну аутентифікацію користувачів на основі сучасних протоколів (OpenID Connect/OAuth 2.0).
- **Валідація даних:** Застосування FluentValidation для декларативного визначення правил валідації вхідних даних, що підвищує надійність системи.
- **Документування API:** Використання бібліотеки Swagger для автоматичної генерації інтерактивної документації API, що спрощує процес інтеграції для споживачів API.
- **Стратегія тестування:** Впровадження багаторівневої стратегії тестування, що включає модульне (unit-testing) та інтеграційне тестування (integration-testing), з використанням фреймворку xUnit та допоміжних бібліотек: NSubstitute, FluentAssertions, Testcontainers та Bogus для забезпечення коректності та стабільності функціонування системи.

Таким чином, обраний архітектурний підхід для серверної частини маркетплейсу "Kolesko", що ґрунтується на принципах REST для зовнішнього інтерфейсу та реалізований з використанням сучасного стеку технологій та передових практик проектування, забезпечує необхідну основу для побудови масштабованої, надійної та безпечної платформи, здатної ефективно вирішувати завдання пошуку, продажу та управління оголошеннями вживаних автомобілів.

1.3. Порівняльний аналіз функціональних та технічних характеристик застосунку з відповідними аналогами на ринку на основі SWOT-аналізу

Сучасний український ринок онлайн-платформ, спеціалізованих на торгівлі вживаними та новими автомобілями, характеризується значною насиченістю. Специфіка цих платформ варіюється: деякі переважно сфокусовані на наданні детальної інформації про історію та технічні характеристики транспортних засобів, їхній стан та особливості, в той час як інші акцентують увагу на підвищенні зручності та ефективності процесу пошуку для користувачів, пропонуючи розширений функціонал для швидкого знаходження найбільш підходящих варіантів, що часто є визначальним фактором для потенційних клієнтів.

При проектуванні онлайн-маркетплейсу вживаних автомобілів "Kolesko" було визначено ключові аспекти, на яких зосереджується розробка з метою забезпечення конкурентоспроможності та задоволення потреб цільової аудиторії:

- Надання чіткого та повного опису загальних характеристик та специфічних особливостей автомобілів.
- Забезпечення швидкого та інтуїтивно зрозумілого доступу до всіх функціональних можливостей застосунку.
- Імплементація передових функцій, що спрощують користувачам процес пошуку, зокрема, розробка гнучкої та просунутої системи фільтрації оголошень, здатної суттєво скоротити час вибору.

Для проведення комплексного аналізу поточного стану ринку та позиціонування майбутнього застосунку було ідентифіковано ключових конкурентів на українському ринку онлайн-платформ для торгівлі автомобілями. До таких платформ віднесено RST та avtobazar.ua. Після ідентифікації аналогів було проведено детальний аналіз їхніх характеристик, зосередивши особливу увагу на технічних аспектах реалізації та функціональних можливостях з точки зору користувача.

Аналіз конкурентних платформ

RST: Ця онлайн-платформа є одним з ключових гравців на українському ринку, функціонуючи з 2006 року та позиціонуючи себе як популярний онлайн-автобазар. За даними платформи, вона містить понад 400 000 оголошень, надаючи користувачам можливості розміщення, редагування та просування своїх пропозицій[6]. Візуальне представлення головної сторінки платформи RST:

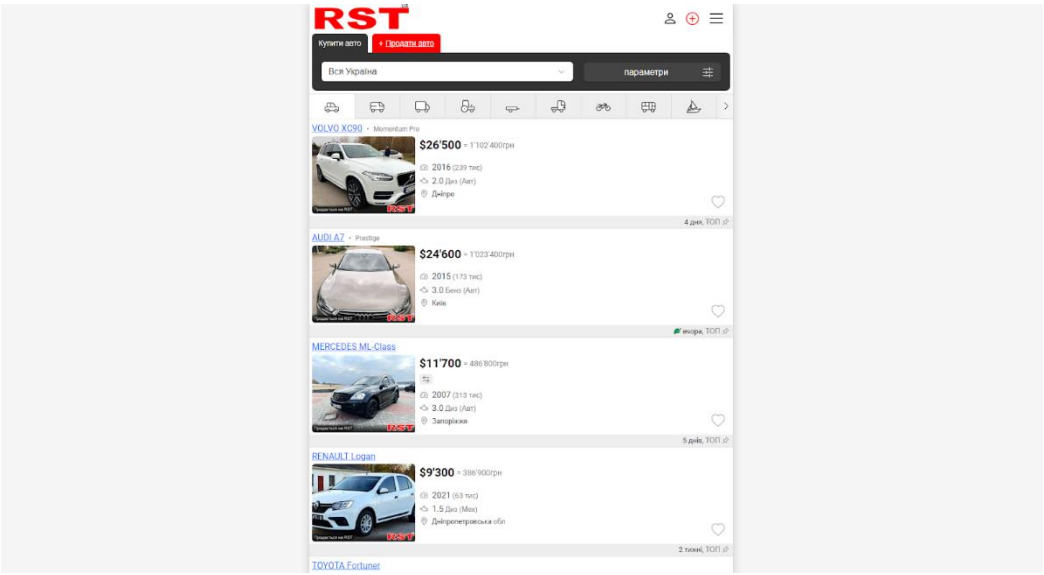

Рис. 1.3.1. Зовнішній вигляд головної сторінки онлайн-платформи rst.ua. *Джерело: [6]*

З технічної та функціональної точки зору, платформа RST має наступні позитивні аспекти:

- **Чітка структурна організація та навігація:** Інтерфейс є зрозумілим для користувача, що спрощує пошук необхідних розділів та інформації.
- **Адаптивний дизайн:** Реалізовано динамічне пристосування інтерфейсу користувача (UI) до розміру екрану пристрою, з якого здійснюється доступ до ресурсу.
- **Наявність базових фільтрів:** Платформа містить функціонал для базової фільтрації оголошень за певними критеріями.

Проте, було ідентифіковано низку негативних технічних аспектів:

- **Архітектурні обмеження:** За наявною інформацією, використання архітектурного патерну MVC як основного для реалізації може обмежувати потенціал масштабованості системи та ускладнювати процес інтеграції нових сервісів у сучасному контексті розробки.
- **Обмеження підсистеми аутентифікації та авторизації:** Виявлено, що реалізація механізмів аутентифікації та авторизації є примітивною, що становить потенційні ризики для безпеки даних користувачів та цілісності платформи.
- **Низький рівень швидкодії:** Спостерігається недостатній рівень продуктивності застосунку, що призводить до погіршення користувацького досвіду (UX) та вказує на необхідність оптимізації внутрішньої логіки та взаємодії з даними.

Avtobazar.ua: Цей онлайн-сервіс активно функціонує на українському ринку вживаних автомобілів з 2010 року. Платформа агрегує оголошення від приватних осіб, автосалонів та дилерів, а також надає унікальну серед конкурентів можливість публікації оголошень про продаж автомобільних запчастин[7]. Зовнішній вигляд головної сторінки платформи avtobazar.ua:

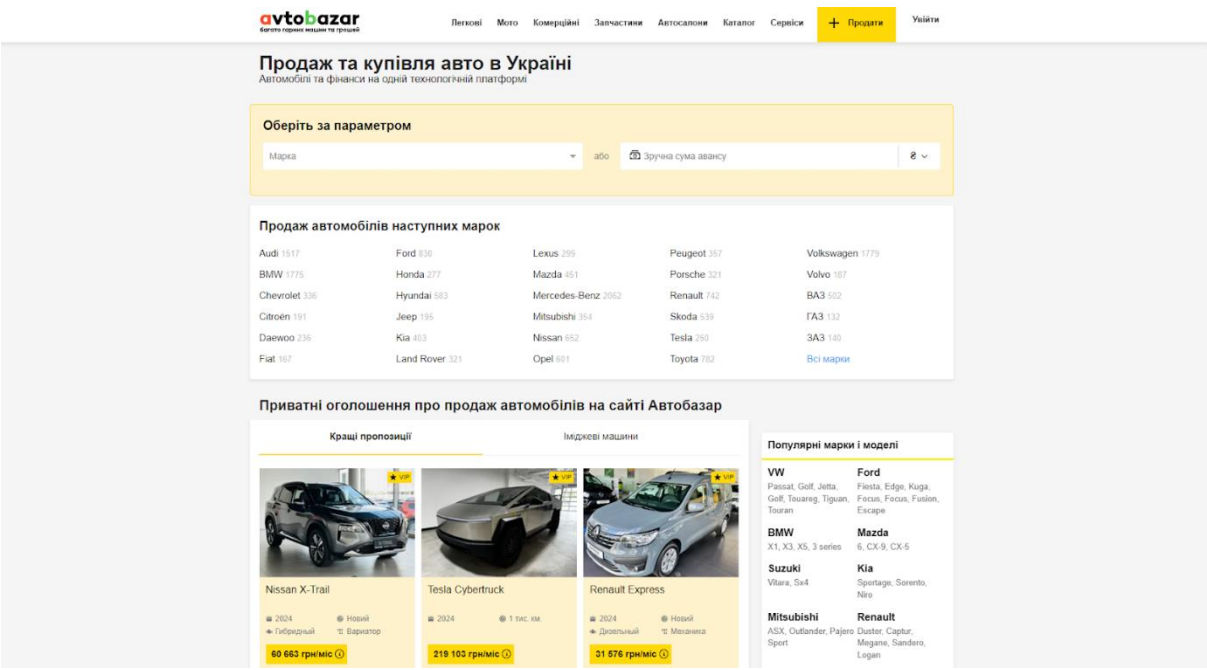

Рис. 1.3.2. Зовнішній вигляд головної сторінки онлайн-платформи avtobazar.ua.

Джерело: [7]

Серед позитивних технічних та функціональних аспектів Avtobazar.ua можна виділити:

- **Чітка структура та навігація:** Платформа має добре організовану структуру та зрозумілу навігацію, що сприяє зручності використання.
- **Прийнятний рівень швидкодії:** Застосунок демонструє досить хорошу продуктивність, забезпечуючи відносно швидке завантаження сторінок.

Однак, існують суттєві негативні аспекти:

- **Відсутність функціоналу фільтрації:** Платформа не надає користувачам можливості використовувати фільтри для пошуку оголошень, що значно ускладнює процес знаходження релевантних пропозицій.
- **Обмеження підсистеми аутентифікації та авторизації:** Подібно до RST, ідентифіковано примітивність реалізації механізмів аутентифікації та авторизації, що створює потенційні виклики для забезпечення безпеки даних користувачів.

Результати аналізу конкурентних платформ, зокрема ідентифіковані позитивні та негативні технічні аспекти їх реалізації, були враховані при проектуванні архітектури та функціоналу "Kolesko" з метою уникнення аналогічних недоліків у власній системі та інтеграції найкращих практик.

SWOT-аналіз онлайн-платформи "Kolesko"

Для комплексної оцінки стратегічного позиціонування онлайн-платформи "Kolesko" на ринку було застосовано методологію SWOT-аналізу. Цей метод дозволяє систематизувати внутрішні фактори (сильні та слабкі сторони) та зовнішні фактори (можливості та загрози), що впливають на успіх проєкту.

Таблиця 1.3.1.

SWOT-аналіз онлайн-платформи "Kolesko"

Сильні сторони (Strengths)	**Слабкі сторони (Weaknesses)**
Розробка на основі сучасних архітектурних рішень, що забезпечує масштабованість та гнучкість.	Обмеженість початкового функціоналу порівняно з усталеними гравцями ринку (наприклад, відсутність соціальних функцій, інтеграції з усіма можливими сторонніми сервісами).
Імплементація просунутої та гнучкої системи фільтрації оголошень, орієнтованої на користувача.	Невелика початкова кількість інтеграцій зі сторонніми сервісами.
Забезпечення високого рівня продуктивності (швидкодії) системи.	Потреба у залученні початкової аудиторії (продавців та покупців).
Надання чіткого та детального опису характеристик автомобілів.	

Використання надійної системи аутентифікації та авторизації на базі Keycloak.	
Можливості (Opportunities)	**Загрози (Threats)**
Потенціал для інтеграції з широким спектром сторонніх сервісів (наприклад, для перевірки історії автомобілів, оформлення страхових полісів, фінансування).	Високий рівень конкуренції з боку існуючих, усталених платформ.
Можливість залучення значної користувацької бази через цільові маркетингові кампанії.	Вплив макроекономічних факторів (наприклад, зміни на ринку вживаних автомобілів, купівельної спроможності населення) на зацікавленість користувачів.
Перспектива міжнародного розширення платформи на сусідні або глобальні ринки.	Ризик виникнення непередбачуваних технічних проблем, пов'язаних з апаратними збоями або спробами зовнішнього несанкціонованого втручання.
Розширення функціоналу шляхом додавання супутніх сервісів (наприклад, оцінка авто, консультації експертів).	Зміни у законодавстві, що регулює онлайн-торгівлю та захист персональних даних.

Джерело: [створено автором]

Результати проведеного SWOT-аналізу слугують основою для формування стратегії розвитку онлайн-платформи "Kolesko", дозволяючи сфокусувати зусилля на використанні існуючих сильних сторін, мінімізації впливу слабких, реалізації потенційних можливостей та розробці планів реагування на можливі зовнішні загрози.

Висновки до розділу №1

У першому розділі роботи було проведено комплексне дослідження теоретичних основ та факторів, що визначають розробку серверної частини для онлайн-маркетплейсу вживаних автомобілів "Kolesko".

Проаналізовано ключові тенденції українського вторинного автомобільного ринку на основі актуальних аналітичних досліджень, зокрема диверсифікацію платформ продажу транспортних засобів, динаміку зміни кількості та середньої вартості розміщених оголошень, а також зростання інтересу до екологічних транспортних засобів (гібридних та електричних авто). На підставі виявлених трендів, зумовлених макроекономічними викликами та зміною споживчих пріоритетів, обґрунтовано актуальність створення сучасної онлайн-платформи, яка відповідатиме актуальним потребам користувачів у зручному пошуку доступних та, за необхідності, екологічних варіантів.

Значну увагу приділено архітектурним рішенням для проектування серверної частини застосунку та її програмного інтерфейсу (API) як ключового інтеграційного рівня. Проведено порівняльний аналіз провідних архітектурних стилів API – REST, GraphQL, SOAP – систематизовано їхні характеристики за механізмами взаємодії з даними, гнучкістю запитів, аспектами розробки та підтримки, операційними характеристиками, безпекою та рівнем розповсюдженості у порівняльних таблицях. На підставі цього аналізу обґрунтовано вибір REST як оптимального архітектурного стилю для реалізації API маркетплейсу "Kolesko", зважаючи на його відповідність принципам простоти, масштабованості, ефективність для ресурсних моделей та широку підтримку в обраному технологічному стеку (.NET).

Визначено ключові компоненти загальної архітектури серверної частини, включаючи використання .NET, принципів чистої архітектури (Clean Architecture) та патернів CQRS для організації коду, Entity Framework Core як ORM, Keycloak для централізованої ідентифікації та доступу, FluentValidation для валідації даних, Swagger для документування API та багаторівневої стратегії тестування.

Також здійснено аналіз ринку онлайн-платформ вживаних автомобілів, ідентифіковано ключових конкурентів та проаналізовано їхні функціональні й технічні характеристики з метою визначення сильних сторін та недоліків існуючих рішень на ринку. Проведено SWOT-аналіз онлайн-платформи "Kolesko" для комплексної оцінки її стратегічного позиціонування, виявлення внутрішніх факторів успіху/обмежень та зовнішніх можливостей/загроз.

Результати цього комплексного теоретичного та аналітичного дослідження створили необхідну науково-практичну базу для подальшого проектування та реалізації серверної частини онлайн-маркетплейсу "Kolesko", дозволивши сформувати обґрунтовані архітектурні рішення та стратегію розвитку проєкту з урахуванням актуальних ринкових умов, потреб користувачів та конкурентного середовища.

РОЗДІЛ 2
ПРИКЛАДНА ЧАСТИНА РОЗРОБКИ СЕРВЕРНОЇ ЧАСТИНИ МАРКЕТПЛЕЙСУ ВЖИВАНИХ АВТОМОБІЛІВ

2.1. Вибір технологій, архітектурних підходів для розробки

У даному підрозділі представлено обґрунтування вибору ключового технологічного стеку та архітектурних патернів, які були використані при проектуванні та реалізації серверної частини онлайн-маркетплейсу вживаних автомобілів "Kolesko". Головною метою було визначення оптимального набору інструментів та підходів, що забезпечують високу продуктивність, масштабованість, безпеку та підтримуваність системи.

2.1.1. Вибір технологічної платформи для реалізації API

Для побудови REST API серверної частини було прийнято рішення базувати реалізацію на технологічній платформі **ASP.NET Core Web API**, яка є частиною екосистеми .NET, розробленої компанією Microsoft. Починаючи з 5-ї версії, фреймворк ASP.NET Core став частиною уніфікованої платформи .NET, що розширила його застосування за межі суто веб-додатків. На момент створення REST API для онлайн-платформи Kolesko було обрано **.NET 8**, як актуальну версію з довгостроковою підтримкою.

Вибір даної версії та платформи загалом обумовлений комплексом факторів[8]:

- **Кросплатформна сумісність:** Можливість розгортання застосунку на різних операційних системах (Windows, Linux, macOS) завдяки використанню .NET Runtime, що забезпечує гнучкість при виборі середовища експлуатації.
- **Висока продуктивність:** Платформа .NET відома своєю високою швидкістю виконання коду та ефективним управлінням ресурсами, що критично важливо для обробки значної кількості одночасних запитів, характерних для маркетплейсу.

- **Вбудовані механізми безпеки та регулярні оновлення:** Платформа надає вбудовані засоби захисту від поширених веб-атак (наприклад, XSS, CSRF), а регулярні оновлення безпеки від Microsoft забезпечують актуальний рівень захисту.
- **Сприяння масштабованості:** Підтримка контейнеризації (Docker), інтеграція з хмарними сервісами та можливість реалізації як горизонтального, так і вертикального масштабування роблять .NET вдалим вибором для зростаючих проєктів.
- **Розвинена екосистема та інструментарій:** Доступ до великої кількості готових бібліотек та пакетів через NuGet, активна спільнота розробників та детальна документація від Microsoft значно спрощують та прискорюють розробку.
- **Статус LTS та гарантована підтримка:** .NET 8 має статус Long Term Support (LTS), що гарантує отримання технічної підтримки та оновлень безпеки протягом трьох років, забезпечуючи стабільність проєкту.
- **Високий рівень зрілості технології:** Платформа .NET є перевіреною часом, з великою кількістю успішно реалізованих проєктів та усталеними практиками розробки.
- **Модульна структура:** Можливість включати у застосунок лише необхідні компоненти завдяки модульній архітектурі дозволяє оптимізувати розмір та ресурсоспоживання.

2.1.2. Вибір загального архітектурного стилю

Після визначення базової технологічної платформи було здійснено аналіз та вибір загального архітектурного стилю для побудови серверної частини застосунку. На основі критеріїв масштабованості, тестованості, підтримуваності та гнучкості перевагу було надано принципам **Clean**

Architecture, запропонованим Р. Мартіном[9]. Дана архітектура передбачає чітке розділення відповідальності між шарами, де зовнішні шари залежать від внутрішніх, але не навпаки, забезпечуючи незалежність бізнес-логіки від інфраструктурних деталей.

Структура Clean Architecture типово включає наступні шари:

- Domain Layer (Core) – інкапсулює бізнес-логіку та правила.
- Application Layer – містить логіку застосунку та визначає сценарії використання.
- Infrastructure Layer – реалізує взаємодію із зовнішніми ресурсами (файлова система, мережа тощо).
- Persistence Layer – відповідає за взаємодію зі сховищами даних.
- Presentation Layer – забезпечує взаємодію з користувачем або іншими системами (наприклад, API).

Аргументами на користь вибору Clean Architecture слугували наступні її переваги:

- **Незалежність від інфраструктури:** Бізнес-логіка, розташована у центральних шарах, не залежить від конкретних фреймворків, баз даних чи зовнішніх сервісів, що спрощує їхню потенційну заміну.
- **Висока тестованість:** Чіткий поділ на шари та ізольованість бізнес-логіки дозволяють легко писати юніт-тести для кожного шару окремо без складних зовнішніх залежностей.
- **Сприяння масштабованості:** Модульність та чітка структура полегшують розподіл роботи між командами та незалежне масштабування окремих компонентів.

- **Висока підтримуваність:** Ізольованість шарів та стандартизована структура коду спрощують внесення змін, пошук помилок та адаптацію нових розробників до проєкту.
- **Відповідність принципам SOLID:** Архітектура природно сприяє дотриманню принципів об'єктно-орієнтованого проектування SOLID, що підвищує якість та підтримуваність коду.
- **Гнучкість до змін:** Чіткі межі між шарами роблять систему стійкішою до змін вимог бізнесу або технологічних платформ.

2.1.3. Вибір підсистеми управління ідентифікацією та доступом (IAM)

Критичним аспектом для будь-якої онлайн-платформи, що оперує користувацькими даними та транзакціями, є забезпечення надійних механізмів аутентифікації та авторизації. Для реалізації функціоналу управління ідентифікацією та доступом (Identity and Access Management – IAM) у проєкті "Kolesko" було обрано відкриту систему **Keycloak**. Keycloak виступає як окремий Identity Provider, що централізовано керує життєвим циклом користувачів, їхніми ролями та дозволами.

Вибір Keycloak обумовлений наступними факторами[10]:

- **Дотримання стандартів:** Підтримка сучасних галузевих стандартів безпеки (OpenID Connect, OAuth 2.0), що забезпечує сумісність, взаємодію та високий рівень захисту.
- **Багатий функціонал "з коробки":** Надання широкого спектру готових до використання можливостей, таких як управління користувачами та групами, федерація ідентифікацій, Single Sign-On (SSO), багатофакторна аутентифікація (MFA), без необхідності розробки цієї складної логіки в рамках основного бекенду.
- **Зниження навантаження та спрощення архітектури основного бекенду:** Винесення комплексної логіки аутентифікації та

авторизації в окремий, спеціалізований сервіс, дозволяє серверній частині "Kolesko" сфокусуватися виключно на бізнес-логіці маркетплейсу.

- **Масштабованість та надійність:** Keycloak є зрілим, високодоступним рішенням, призначеним для роботи з великою кількістю користувачів та запитів.
- **Спрощення процесів комплаєнсу:** Використання відомого, стандартизованого та добре документованого рішення для управління ідентифікацією полегшує дотримання нормативних вимог щодо захисту персональних даних та доступу.

Інтеграція Keycloak з серверною частиною проєкту реалізується з використанням стандартних бібліотек для взаємодії за протоколами OIDC/OAuth2. Цей процес включає налаштування клієнта в Keycloak, конфігурацію серверної частини для перевірки та валідації токенів доступу, отриманих від Keycloak, а також використання інформації про ролі та скоупи (scopes) для реалізації механізмів авторизації на рівні API ендпоінтів.

2.1.4. Застосування архітектурного патерну CQRS та бібліотеки MediatR

Для оптимізації обробки операцій, пов'язаних з маніпуляцією та отриманням даних через API, було прийнято рішення застосувати архітектурний патерн **Command Query Responsibility Segregation (CQRS)**. CQRS передбачає розділення моделей або об'єктів для операцій читання (Queries) та операцій запису/зміни стану (Commands) даних[11]. Такий підхід дозволяє незалежно оптимізувати шляхи виконання для операцій, які змінюють дані, та для операцій, що лише їх отримують. Для спрощення імплементації CQRS та реалізації патерну Посередника (Mediator pattern), який зменшує прямі залежності між відправником і отримувачем запиту/команди, було обрано бібліотеку **MediatR**.

Вибір CQRS та MediatR обґрунтований наступними перевагами:

- **Чітке розділення відповідальності:** Явно відокремлює логіку читання від логіки запису, що покращує структуру коду та його підтримку.
- **Можливість незалежної оптимізації:** Дозволяє використовувати різні моделі даних та технології для читання (оптимізовані для швидкості отримання) та запису (оптимізовані для цілісності транзакцій).
- **Спрощення процесу тестування:** Ізольовані команди та запити легше тестувати в ізоляції.
- **Зменшення зв'язності:** Патерн Mediator, реалізований у MediatR, знижує прямі залежності між компонентами системи.
- **Підтримка пайплайну обробки:** MediatR надає вбудовану підтримку для створення пайплайнів, через які проходять команди та запити, дозволяючи зручно інтегрувати крос-наскрізні аспекти, такі як валідація запитів, управління транзакціями, логування чи обробка подій.
- **Покращена читабельність та зменшення дублювання:** Стандартизований підхід до обробки команд та запитів робить код більш зрозумілим та зменшує необхідність дублювання логіки.
- **Легка інтеграція:** MediatR добре інтегрується з системою Dependency Injection у .NET та іншими компонентами екосистеми.
- **Активна спільнота та підтримка:** MediatR є популярною бібліотекою з активною спільнотою та постійною підтримкою.

В результаті проведеного аналізу та обґрунтування було визначено ключові технології та архітектурні підходи для реалізації серверної частини онлайн-маркетплейсу "Kolesko". Обраний технологічний стек на базі **.NET** забезпечує необхідний рівень продуктивності, безпеки та кросплатформності. Загальна

архітектура, що ґрунтується на принципах **Clean Architecture**, разом з використанням патерну **CQRS** та бібліотеки **MediatR** для обробки запитів, а також інтеграцією **Keycloak** для управління ідентифікацією та доступом, формує надійний, масштабований, тестований та легко підтримуваний фундамент для розробки програмного забезпечення, що відповідає сучасним вимогам до веб-застосунків.

2.2. Створення та налаштування проєкту

Після обрання технологічного стеку та визначення основного архітектурного стилю для серверної частини застосунку, було розроблено та імплементовано структуру програмного рішення. З метою забезпечення високого рівня організованості коду, керованості залежностями та відповідності принципам обраної Чистої архітектури, було прийнято рішення про створення ієрархії директорій, яка базується на принципах функціональної декомпозиції або feature-driven підходу[12]. Такий підхід передбачає групування файлів та компонентів, що стосуються певної функціональності або сутності домену (наприклад, VehicleModel, яка представляє модель автомобіля), в єдиному логічному місці. Ця організація спрямована на полегшення навігації та розуміння структури проєкту, забезпечення масштабованості, зниження зв'язності між ними, а також на підтримку принципу єдиної відповідальності (Single Responsibility Principle – SRP).

Відповідно до шарів Чистої архітектури, програмне рішення було розділено на п'ять окремих проєктів (.NET проєктів), назва яких відповідає назві відповідного шару: Application, Domain, Infrastructure, Persistence та Presentation. Далі представлено детальний опис структури та призначення кожного з цих проєктів.

1. Проєкт Application

Проєкт Application відповідає за реалізацію бізнес-логіки застосунку та координацію взаємодії між доменним шаром та зовнішніми інтерфейсами. Він містить визначення команд (Commands) та запитів (Queries) у рамках патерну

CQRS, а також обробники (Handlers) для них. Структура шару Application включає:

- Behaviors – компоненти, що реалізують крос-наскрізні пайплайни обробки запитів (наприклад, валідація, транзакційність, логування).
- LocationRegions – директорія, що містить логіку (команди, запити, обробники), пов’язану з управлінням регіонами локацій продажу автомобілів.
- LocationTowns – директорія, що містить логіку, пов’язану з управлінням населеними пунктами локацій продажу автомобілів.
- VehicleBodyTypes, VehicleBrands, VehicleColors, VehicleDrivetrainTypes, VehicleEngineTypes, VehicleModels, VehicleTransmissionTypes, VehicleTypes – директорії, що містять компоненти логіки (команди, запити, обробники, DTO), пов'язаної з різними класифікаторами та деталями транспортних засобів.
- Vehicles – директорія, що містить основну логіку роботи з сутностями транспортних засобів (оголошення про продаж).
- Images – директорія, що містить логіку, пов’язану з обробкою та управлінням зображеннями транспортних засобів.
- Users – директорія, що містить логіку (команди, запити, обробники), пов’язану з управлінням користувачами та пов’язаними з ними аспектами (оновлення особистих даних, отримання оголошень, збережених у «Бажане» тощо).
- Shared – директорія, що містить спільні ресурси, такі як допоміжні класи, методи розширення, профілі маппінгу (AutoMapper) тощо.
- DependencyInjection.cs – файл для конфігурації контейнера інверсії управління (IoC) на рівні шару Application.

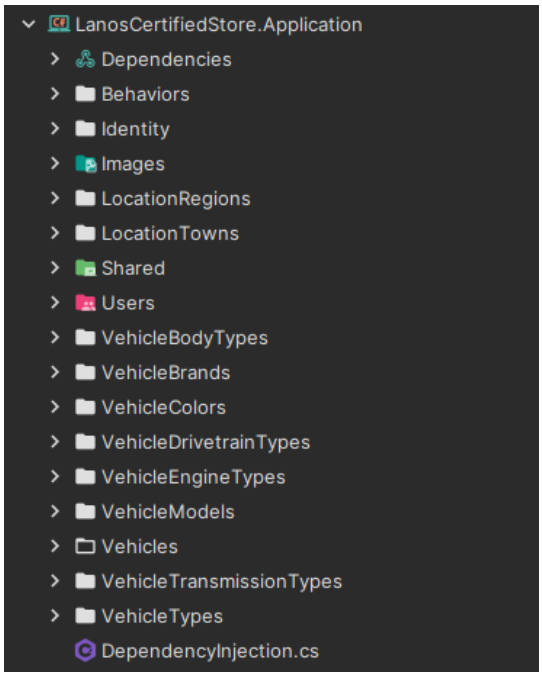

Рис. 2.2.1. Структура шару Application. *Джерело: [створено автором]*

2. Проєкт Domain

Проєкт Domain є ядром програмного рішення і містить основні сутності доменної моделі та бізнес-правила, які не залежать від деталей реалізації або інфраструктури. Він реалізує найвнутрішній шар Чистої архітектури. Структура шару Domain включає:

- Contracts – директорія, що містить інтерфейси та контракти, які визначають взаємодію в межах доменного шару (наприклад, репозиторії) і використовуються іншими шарами за принципом інверсії залежностей.
- Entities – директорія, що містить визначення основних доменних сутностей (наприклад, Vehicle, Location, Image тощо).

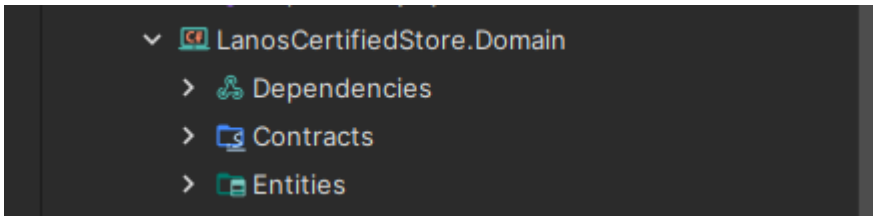

Рис. 2.2.2. Структура шару Domain. *Джерело: [створено автором]*

3. Проєкт Infrastructure

Проєкт Infrastructure відповідає за реалізацію технічних аспектів та взаємодію із зовнішніми сервісами та ресурсами, які знаходяться за межами ядра застосунку (наприклад, сторонні API, файлові системи). Він містить імплементації інтерфейсів, визначених у доменному або прикладному шарах. Структура шару Infrastructure:

- SeedingData – компоненти для початкового наповнення бази даних тестовими або конфігураційними даними.
- Images – сервіси, що забезпечують взаємодію зі стороннім API сховища для обробки та зберігання зображень.
- Locations – сервіси для роботи із зовнішніми даними або сервісами, пов'язаними з локаціями (наприклад, геокодування).
- Users – сервіси для роботи з користувачами та Identity Provider`ом маркетплейсу – Keycloak.
- Vehicles – сервіси, що надають функціональність, пов'язану з автомобілями.
- DependencyInjection.cs – файл для конфігурації залежностей шару Infrastructure, реєструючи імплементації зовнішніх сервісів.

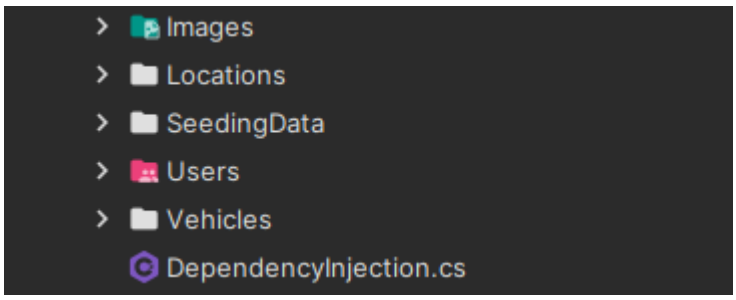

Рис. 2.2.3. Структура шару Infrastructure. *Джерело: [створено автором]*

4. Проєкт Persistence

Проєкт Persistence відповідає виключно за логіку взаємодії зі сховищем даних, зокрема з реляційною базою даних PostgreSQL, використовуючи Entity Framework Core як основний інструмент. Він містить:

- Commands – реалізації команд для модифікації даних у сховищі (наприклад, додавання нового оголошення).
- Contexts – налаштування контексту бази даних (DbContext) для Entity Framework Core, включаючи конфігурацію моделей та зв'язків.
- Data – директорія для зберігання SQL-сценаріїв, файлів міграцій бази даних або інших ресурсів, пов'язаних з управлінням даними.
- Queries – реалізації запитів для ефективного отримання даних з бази даних.
- Services – сервіси, що інкапсулюють складнішу логіку взаємодії з базою даних або репозиторії.
- DependencyInjection.cs – файл для конфігурації залежностей шару Persistence.

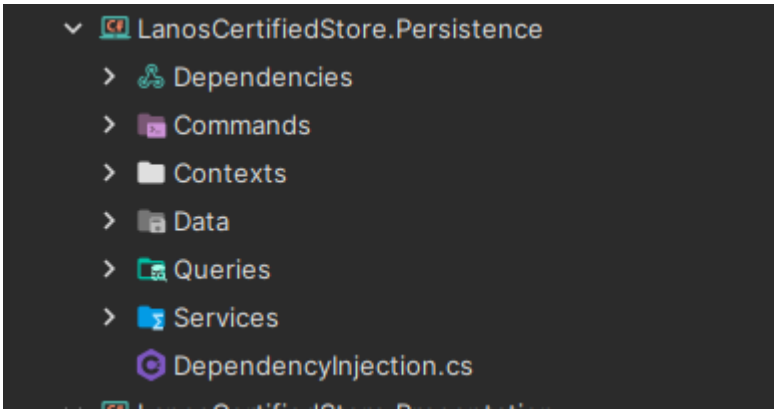

Рис. 2.2.4. Структура шару Persistence. *Джерело: [створено автором]*

5. Проєкт Presentation

Проєкт Presentation є зовнішнім шаром Чистої архітектури, що відповідає за точку входу до застосунку (у даному випадку – API) та взаємодію з клієнтськими застосунками. Він містить компоненти, необхідні для обробки вхідних HTTP-запитів, їхньої маршрутизації до прикладного шару та формування HTTP-відповідей. Структура шару Presentation:

- Properties – директорія для конфігураційних файлів налаштувань проєкту.

- wwwroot – директорія для статичних файлів.
- Controllers – реалізації контролерів, які обробляють вхідні HTTP-запити, виконують базову валідацію та делегують виконання логіці прикладного шару, відправляючи команди або запити MediatR.
- Extensions – компоненти для розширення функціональності.
- Middlewares – реалізації проміжних обробників для конвеєра обробки HTTP-запитів (наприклад, для обробки помилок).
- appsettings.json – конфігураційні файли для налаштувань застосунку.
- Dockerfile – файл для конфігурації збірки та розгортання застосунку в контейнері Docker.
- Program.cs – точка входу до програми, що ініціалізує хост застосунку, налаштовує Dependency Injection та будує конвеєр обробки HTTP-запитів.

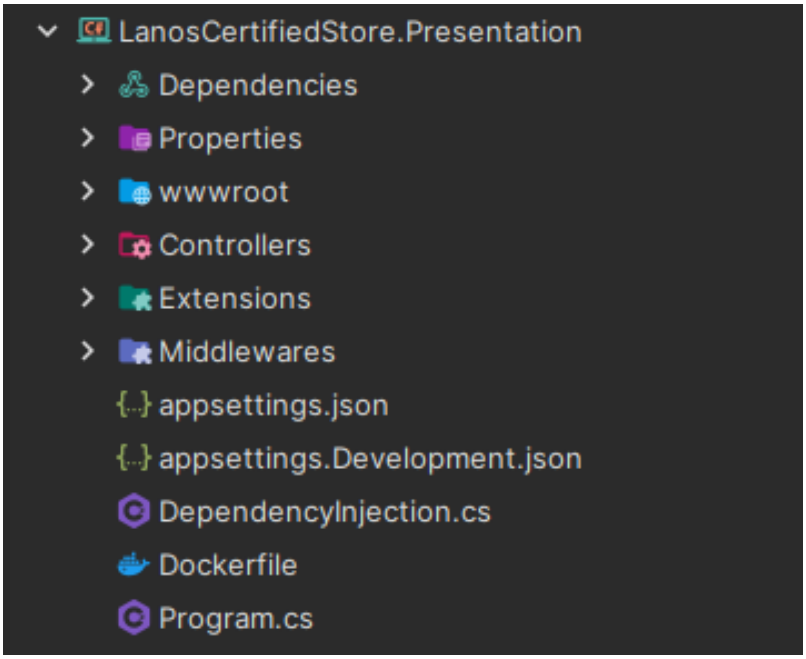

Рис. 2.2.5. Структура шару Presentation. *Джерело: [створено автором]*

Така структура програмного рішення, що відповідає принципам Clean Architecture та використовує функціональну декомпозицію всередині шарів, сприяє створенню добре організованого, підтримуваного, тестованого та масштабованого коду серверної частини онлайн-маркетплейсу "Kolesko".

2.3. Конфігурація базової інфраструктури API

Перед початком безпосередньої реалізації кінцевих точок (ендпоінтів) API для онлайн-маркетплейсу вживаних автомобілів "Kolesko" було проведено аналіз та вибір базового підходу до їх побудови. У .NET існують два основні підходи до реалізації HTTP API: класичні контролери (Controller-based API) та Minimal API[13].

Розглянемо характеристики кожного з цих підходів:

- **Controller-based API:** Це традиційний підхід до побудови Web API в .NET, що базується на використанні класів контролерів, які успадковуються від ControllerBase[13]. Даний підхід є доцільним у випадках, коли:
  - Проєкт має потенціал до значного масштабування та зростання комплексності.
  - Вимагається чітка, стандартизована структура та високий рівень організації коду.
  - Необхідна реалізація складних сценаріїв валідації та авторизації запитів.
- **Minimal API:** Цей підхід, представлений у .NET 6, пропонує більш лаконічний спосіб створення API без необхідності визначення класів контролерів [13]. Його ключові особливості:
  - Відсутність явно визначених класів контролерів.
  - Зменшення обсягу шаблонного коду.
  - Потенційно трохи вища продуктивність порівняно з Controller-based API.

Виконавши детальний аналіз цих підходів та зіставивши їх із специфічними вимогами до API маркетплейсу "Kolesko" (зокрема, необхідність забезпечення масштабованості, чіткої організації коду та підтримки складних механізмів безпеки та валідації), перевагу було надано підходу **Controller-based API**. Таке рішення

обґрунтовано тим, що класичні контролери краще відповідають потребам проєкту зі значним потенціалом зростання та складною бізнес-логікою.

Після вибору підходу до реалізації ендпоінтів, було розроблено базові компоненти інфраструктури API. Фундаментальним компонентом архітектури Presentation шару став **базовий контролер BaseApiController**, який успадковується всіма специфічними контролерами API.

Лістинг 2.3.1. Код базового контроллера BaseApiController.cs

```
namespace LanosCertifiedStore.Presentation.Controllers.Common;

[ApiController]
public abstract class BaseApiController : ControllerBase
{
   private ISender? _sender;

   protected ISender Sender => (_sender ??=
       HttpContext.RequestServices.GetService<ISender>())!;

   private protected static ProblemDetails CreateValidationProblemDetails(Error error,
       Error[]? errors) => new()
   {
       Title = "ValidationError",
       Status = (int)HttpStatusCode.BadRequest,
       Detail = error.Message,
       Extensions = { { nameof(errors), errors } }
   };

   private protected static ProblemDetails CreateNotFoundProblemDetails(Error error) =>
new()
   {
       Title = error.Code,
       Status = (int)HttpStatusCode.NotFound,
       Detail = error.Message
   };
}
```

Джерело: [створено автором]

Даний клас агрегує спільну функціональність та інтегрується з механізмом маршрутизації запитів (ISender від бібліотеки MediatR), що дозволяє делегувати обробку запитів та команд з рівня контролерів до відповідних обробників у шарі Application. Це сприяє мінімізації бізнес-логіки безпосередньо в контролерах,

підвищуючи чистоту та підтримуваність коду. Базовий контролер також надає стандартизовані методи (CreateValidationProblemDetails, CreateNotFoundProblemDetails) для формування об'єктів ProblemDetails у відповідь на типові помилки, такі як помилки валідації даних (HTTP статус 400 BadRequest) або відсутність запитуваного ресурсу (HTTP статус 404 NotFound).

Конвеєр обробки запитів та крос-наскрізні аспекти

Крос-наскрізні функціональні аспекти обробки запитів, такі як логування, валідація та управління транзакціями, реалізовано за допомогою механізму пайплайнів (Pipeline Behaviors) бібліотеки MediatR. Ці пайплайни формують конвеєр, через який проходить кожен запит або команда перед тим, як потрапити до кінцевого обробника (Handler).

1. **Пайплайн логування:** Першим у послідовності виконання пайплайнів є обробник логування (RequestLoggingPipelineBehavior). Він відповідає за запис інформації про процес обробки запитів. Для логування використано бібліотеку Serilog [14] у поєднанні із сервером збору та візуалізації логів Seq [15]. Пайплайн фіксує ключові етапи: початок обробки запиту, успішне завершення або завершення з помилкою, надаючи деталі для подальшого аналізу.

Лістинг 2.3.2. Пайплайн для проведення логування RequestLoggingPipelineBehavior.cs

```
namespace LanosCertifiedStore.Application.Behaviors;

internal sealed class RequestLoggingPipelineBehavior<TRequest, TResponse>(
   ILogger<RequestLoggingPipelineBehavior<TRequest, TResponse>> logger)
   : IPipelineBehavior<TRequest, TResponse>
   where TRequest : notnull
   where TResponse : Result
{
   public async Task<TResponse> Handle(TRequest request,
       RequestHandlerDelegate<TResponse> next,
       CancellationToken cancellationToken)
   {
       var requestName = typeof(TRequest).Name;

       logger.LogInformation("Processing request {requestName}", requestName);
```

```
        var result = await next();

        if (result.IsSuccess)
        {
            logger.LogInformation("Completed request {RequestName}", requestName);
        }
        else
        {
            using (LogContext.PushProperty("Error", result.Error, true))
            {
                logger.LogError("Completed request {RequestName} with error", requestName);
            }
        }

        return result;
    }
}
```

Джерело: [створено автором]

2. **Пайплайн валідації:** Наступним етапом у конвеєрі обробки є валідація вхідних даних запиту. Цей процес реалізовано за допомогою бібліотеки FluentValidation [16], яка дозволяє декларативно визначати правила валідації для об'єктів команд та запитів. Кожен запит або команда може мати відповідний валідатор. Структура валідатора для команди створення моделі автомобіля (CreateVehicleModelCommandRequestValidator) та приклади визначення правил валідації наведено у лістингах 2.3.3 та 2.3.4 відповідно.

Лістинг 2.3.9. Валідатор запиту створення моделі автомобіля

```
internal sealed class CreateVehicleModelCommandRequestValidator :
AbstractValidator<CreateVehicleModelCommandRequest>
{
    public CreateVehicleModelCommandRequestValidator()
    {
        GetNameValidationRules();
        GetBrandValidationRules();
        GetTypeValidationRules();
        GetBodyTypesValidationRules();
        GetTransmissionTypesValidationRules();
        GetDrivetrainTypesValidationRules();
        GetEngineTypesValidationRules();
        GetProductionYearValidationRules();
    }
```

Джерело: [створено автором]

Лістинг 2.3.4. Приклади правил валідації у CreateVehicleModelCommandRequestValidator

```
private void GetBodyTypesValidationRules()
{
   Expression<Func<CreateVehicleModelCommandRequest, IEnumerable<Guid>>>
expression =
       x => x.AvailableBodyTypesIds;

   RuleFor(expression)
       .NotNull()
       .NotEmpty()
       .WithMessage(VehicleModelValidatorMessages.EmptyBodyTypeCollection);
}

private void GetEngineTypesValidationRules()
{
   Expression<Func<CreateVehicleModelCommandRequest, IEnumerable<Guid>>>
expression =
       x => x.AvailableEngineTypesIds;

   RuleFor(expression)
       .NotNull()
       .NotEmpty()
       .WithMessage(VehicleModelValidatorMessages.EmptyEngineTypeCollection);
}
```

Джерело: [створено автором]

Логіка валідаційного пайплайну (ValidationPipelineBehavior) перевіряє наявність валідаторів для поточного запиту. Якщо валідатори існують, пайплайн виконує їх та збирає список помилок. У випадку наявності помилок, обробка запиту переривається, формується негативний результат валідації, який потім конвертується у відповідь зі статусом 400 BadRequest на рівні контролера. Якщо помилок валідації немає, виконання передається наступному обробнику в пайплайні.

Лістинг 2.3.5. Основна логіка валідаційного пайплайну ValidationPipelineBehavior.cs

```
public class ValidationPipelineBehavior<TRequest, TResponse>(
   ILogger<ValidationPipelineBehavior<TRequest, TResponse>> logger,
   IEnumerable<IValidator<TRequest>> validators) : IPipelineBehavior<TRequest, TResponse>
```

```
    where TRequest : ICommandRequestBase
    where TResponse : Result
{
    public async Task<TResponse> Handle(
        TRequest request,
        RequestHandlerDelegate<TResponse> next,
        CancellationToken cancellationToken)
    {
        if (!validators.Any())
        {
            return await next();
        }

        var errors = await GetValidationErrors(request, cancellationToken);

        if (!AreErrorsPresent(errors))
        {
            return await next();
        }

        var validationResult = CreateValidationResult<TResponse>(errors);

        logger.LogError("Validation failures have occured: {@ValidationResult}",
validationResult);
        return validationResult;
    }
```

Джерело: [створено автором]

3. **Пайплайн транзакційності:** Для забезпечення атомарності та консистентності операцій, що передбачають модифікацію стану даних у сховищі (тип запиту Command), застосовано пайплайн управління транзакціями (TransactionPipelineBehavior).

Лістинг 2.3.6. Пайплайн TransactionPipelineBehavior.cs

```
public class TransactionPipelineBehavior<TRequest, TResponse>(
    ITransactionService transactionService) : IPipelineBehavior<TRequest, TResponse>
    where TRequest : ICommandRequestBase
    where TResponse : Result
{
    public async Task<TResponse> Handle(
        TRequest request,
        RequestHandlerDelegate<TResponse> executedAction,
        CancellationToken cancellationToken)
    {
        await transactionService.BeginTransaction(cancellationToken);

        var response = await executedAction();
```

```
        if (!response.IsSuccess)
        {
            await transactionService.RollbackTransaction(cancellationToken);

            return response;
        }

        await transactionService.CommitTransaction(cancellationToken);

        return response;
    }
}
```

Джерело: [створено автором]

Даний пайплайн забезпечує, що всі операції запису в рамках виконання команди будуть або повністю успішними (з фіксацією транзакції), або повністю скасованими (з відкатом транзакції) у випадку будь-якої помилки під час виконання executedAction, що представляє основну логіку обробника команди.

Проміжне програмне забезпечення для обробки виключень (Exception Handling Middleware)

Для забезпечення стійкості застосунку до непередбачуваних помилок та уніфікації формату відповідей у разі виникнення виключень, у конвеєрі обробки HTTP-запитів реалізовано проміжне програмне забезпечення (middleware) для обробки виключень. Цей механізм перехоплює необроблені виключення, логує їх та генерує стандартизовану HTTP-відповідь зі статусом помилки та деталями (у форматі Problem Details), запобігаючи краху застосунку.

Була розроблена абстрактна база ExceptionHandlerBase, яка слугує основою для створення специфічних обробників для різних типів виключень.

Лістинг 2.3.7. Базовий клас Middleware ExceptionHandlerBase.cs

```
namespace LanosCertifiedStore.Presentation.Middlewares.ExceptionRelated.Common.Classes;

internal abstract class ExceptionHandlerBase<TException>(
    ILogger<ExceptionHandlerBase<TException>> logger,
```

```
    HttpStatusCode statusCode,
    string title,
    string? detail = null!) : IExceptionHandler
    where TException : Exception
{
    public async ValueTask<bool> TryHandleAsync(
        HttpContext httpContext,
        Exception exception,
        CancellationToken cancellationToken)
    {
        if (exception is not TException)
        {
            return false;
        }

        logger.LogError(
            exception, "Exception occurred: {Message}", exception.Message);

        var problemDetails = new ProblemDetails
        {
            Title = title,
            Status = (int)statusCode,
            Detail = detail ?? exception.Message
        };

        httpContext.Response.StatusCode = (int)statusCode;

        await httpContext.Response.WriteAsJsonAsync(problemDetails, cancellationToken);

        return true;
    }
}
```

Джерело: [створено автором]

Цей абстрактний клас інкапсулює спільну логіку обробки виключень: визначення відповідного HTTP статусу та заголовка помилки, логування деталей виключення та форматування відповіді у Problem Details. Специфічні обробники успадковують цей клас для перехоплення конкретних типів виключень (наприклад, пов'язаних з базою даних, бізнес-логікою, або загальних необроблених виключень). Такий підхід підвищує стабільність та безпеку API, оскільки помилки не призводять до неочікуваного завершення роботи застосунку, а стандартизовані логи та відповіді спрощують моніторинг та усунення проблем.

2.4. Впровадження Keycloak в якості системи управління ідентифікацією та доступом

Забезпечення надійної системи управління ідентифікацією та доступом (Identity and Access Management – IAM) є критично важливим компонентом архітектури будь-якої сучасної онлайн-платформи, зокрема такої, що оперує користувацькими даними та забезпечує контроль доступу до ресурсів, як онлайн-маркетплейс "Kolesko". Як було зазначено раніше, для реалізації цих функцій обрано відкриту платформу **Keycloak**, яка виступає як централізований Identity Provider. Таке архітектурне рішення дозволяє винести відповідальність за управління користувачами, аутентифікацію та авторизацію за межі основного бізнес-логіки застосунку, підвищуючи його сфокусованість, безпеку та масштабованість. Даний підрозділ деталізує процес впровадження та конфігурації Keycloak, а також описує реалізовані кастомні розширення для задоволення специфічних потреб проєкту.

2.4.1. Доступні механізми аутентифікації та реєстрації користувачів

Платформа Keycloak розроблена з урахуванням необхідності підтримки різноманітних сценаріїв аутентифікації, надаючи гнучкі механізми для входу користувачів до системи. У застосунку "Kolesko" реалізовано можливість реєстрації та аутентифікації користувачів з використанням декількох методів для максимальної зручності[17]:

1. **Традиційна аутентифікація за логіном/email та паролем:** Користувачі мають можливість зареєструвати новий обліковий запис, вказавши адресу електронної пошти (або унікальний логін) та створивши пароль. Надалі вхід до системи здійснюється шляхом введення цих облікових даних. Цей метод є стандартним і зрозумілим для більшості користувачів.
2. **Аутентифікація через соціальні провайдери:** Для спрощення процесу реєстрації та входу інтегровано можливість аутентифікації через зовнішні соціальні провайдери, зокрема **Google (Google Social Login)**. Цей механізм дозволяє користувачам використовувати свої існуючі облікові

записи Google для швидкого доступу до маркетплейсу без необхідності створення нових облікових даних.

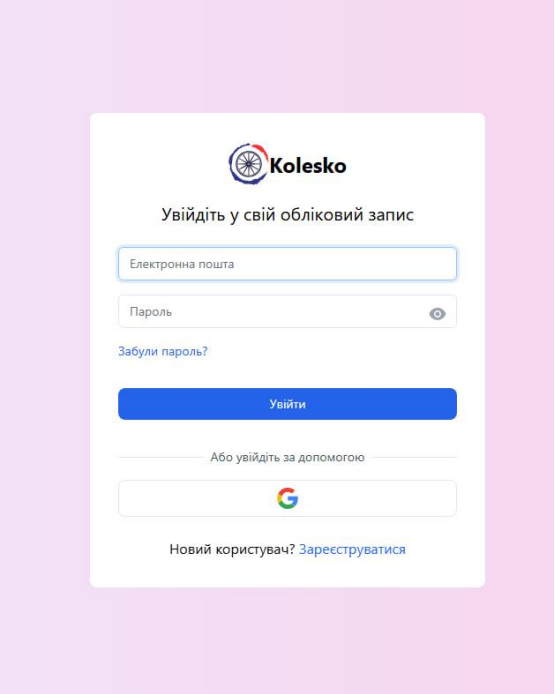

Рис. 2.4.1. Форма входу до застосунку, реалізована через Keycloak. *Джерело: [створено автором]*

Система Keycloak налаштована таким чином, що при першому використанні соціальної аутентифікації користувачем, відбувається автоматична спроба **зв'язування облікового запису соціального провайдера з існуючим користувачем** у Keycloak. Цей процес базується на співставленні адреси електронної пошти, наданої соціальним провайдером, з email, зареєстрованим в Keycloak. У випадку виявлення співпадіння, обліковий запис соціального провайдера автоматично асоціюється з існуючим користувачем Keycloak. Якщо користувач з такою електронною поштою відсутній, Keycloak створює новий обліковий запис у своєму сховищі, асоційований із соціальним провайдером та отриманими від нього даними про користувача (email, ім'я, прізвище). Це забезпечує єдину ідентифікацію користувача незалежно від методу входу.

2.4.2. Процес верифікації облікових записів електронної пошти

З метою підвищення безпеки та підтвердження достовірності контактних даних користувачів, а також для можливості використання функції відновлення паролю та надсилання сповіщень, після успішної реєстрації нового облікового запису (незалежно від обраного методу аутентифікації), аккаунт вимагає підтвердження адреси електронної пошти.

Рис. 2.4.2. Сповіщення про необхідність підтвердження електронної пошти після реєстрації. *Джерело: [створено автором]*

Keycloak автоматично ініціює відправлення верифікаційного повідомлення на вказану електронну адресу користувача. Це повідомлення містить унікальне посилання, перехід за яким підтверджує належність електронної пошти користувачеві та активує його обліковий запис у Keycloak. До моменту успішного підтвердження email, функціональність облікового запису користувача може бути обмежена відповідно до політик реалму, наприклад, забороною на створення оголошень. Цей стандартний потік верифікації, наданий Keycloak, є критично важливим для підтримки цілісності даних користувачів та забезпечення базового рівня безпеки.

Рис. 2.4.3. Повідомлення про необхідність підтвердження електронної пошти.

Джерело: [створено автором]

2.4.3. Розширення функціоналу Keycloak кастомними компонентами

Незважаючи на широкий спектр можливостей, які надає Keycloak "з коробки", у процесі реалізації виникла потреба в специфічній логіці, що вимагала розробки кастомних розширень. На щастя, Keycloak підтримує інтеграції різноманітних доповнень, що дозволяє чудово налаштовувати систему під конкретні потреби[18]. Ці розширення реалізовано як стандартні провайдери Keycloak на мові Java.

1. **Кастомний слухач подій (EventListenerProvider):** Для забезпечення синхронізації даних про нового користувача між Keycloak та внутрішньою базою даних застосунку "Kolesko" було розроблено кастомний слухач подій. Він реалізує інтерфейс EventListenerProvider і підписується на певні події життєвого циклу користувача.

   Цей слухач реагує на події успішної реєстрації користувача (EventType.REGISTER) та створення користувача через адміністративний інтерфейс або API (AdminEvent з ResourceType.USER та OperationType.CREATE). У відповідь на ці події, він отримує об'єкт UserModel для новоствореного користувача та викликає приватний метод sendUserData. Метод sendUserData

призначений для ініціації процесу синхронізації – наприклад, викликаючи відповідну кінцеву точку API серверної частини "Kolesko", передаючи ідентифікатор користувача Keycloak та інші необхідні дані (email, username). Це дозволяє створити або оновити відповідний запис користувача у внутрішній базі даних застосунку, зв'язавши його з ідентифікатором Keycloak, що є необхідним для зберігання додаткових даних про користувача, які не зберігаються в Keycloak, та для реалізації бізнес-логіки, прив'язаної до користувача (наприклад, оголошення користувача).

2. **Кастомний валідатор атрибутів:** Для забезпечення унікальності певних атрибутів користувача, які можуть бути додані як кастомні поля (наприклад, номер телефону), було розроблено кастомний валідатор Keycloak. Цей валідатор призначений для забезпечення унікальності певних атрибутів користувача (наприклад, номера телефону, який може бути доданий як кастомний атрибут користувача через User Profile SPI або прямо в Admin Console) в межах реалму "lsc". Валідатор імплементує інтерфейс AbstractStringValidator та реалізує метод doValidate, який викликається Keycloak у процесі валідації даних користувача (наприклад, при реєстрації або редагуванні профілю). У методі doValidate викликається приватний метод isAttributeUnique, який виконує пошук існуючих користувачів у Keycloak за заданим атрибутом (attributeName) та його значенням (attributeValue) за допомогою UserProvider.searchForUserByUserAttributeStream. Логіка методу isAttributeUnique враховує сценарій редагування профілю існуючого користувача, виключаючи його самого з результатів пошуку дублікатів. Якщо знайдено іншого користувача з таким самим значенням атрибута (або будь-якого користувача при створенні), валідація вважається неуспішною, і валідатор додає відповідну

помилку (ValidationError) до контексту валідації. При виявленні дубліката генерується помилка з кодом, який може бути локалізований на стороні клієнта (як показано у фрагменті конфігурації реалму в localizationTexts). Цей механізм дозволяє запобігти створенню або оновленню користувача з неунікальними даними, що є важливим для підтримки цілісності даних у системі маркетплейсу.

2.5. Реалізація основного функціоналу серверної частини

Детальний опис імплементації ключових бізнес-функціональних модулів серверної частини онлайн-маркетплейсу вживаних автомобілів "Kolesko" охоплює структуру API ендпоінтів для взаємодії з основними сутностями, організацію обробки запитів у прикладному шарі відповідно до патерну Command Query Responsibility Segregation (CQRS), а також реалізацію специфічних функціональних можливостей, таких як механізм пошуку оголошень з оцінкою релевантності та управління списками бажаних оголошень користувачів.

2.5.1. Структура API ендпоінтів та делегування обробки запитів

Відповідно до архітектурного рішення про використання підходу Controller-based API (обґрунтування представлено у підрозділі 2.3), реалізація кінцевих точок API здійснена за допомогою класів контролерів, що успадковуються від базового класу BaseApiController. Цей підхід забезпечує стандартизований каркас для обробки HTTP-запитів. Використання базового контролера та механізму маршрутизації запитів бібліотеки MediatR дозволяє суттєво мінімізувати бізнес-логіку безпосередньо в контролерах, делегуючи обробку команд (що змінюють стан системи) та запитів (що отримують дані) до відповідних обробників (Handlers) у шарі Application. Контролери, таким чином, виступають, по суті, точками входу до API, відповідальними за прийом та початкову обробку запиту (парсинг, базова валідація), делегування його до відповідного обробника через Sender.Send(), та форматування фінальної HTTP-відповіді на основі об'єкта Result, отриманого від обробника.

Як приклади реалізації контролерів, що відповідають за взаємодію з основними сутностями системи, розглянемо VehicleModelController та UserController.

Контролер VehicleModelController містить набір методів для виконання операцій над ресурсами моделей автомобілів. Кожен екшен використовує відповідні атрибути HTTP-методів ([HttpGet], [HttpPost], [HttpPut]), атрибути для документування можливих типів відповідей ([ProducesResponseType]), а також атрибути авторизації ([HasAccessPermission], де це необхідно).

Лістинг 2.5.1. Фрагмент VehicleModelController.cs - Приклад ендпоінту отримання колекції моделей

```
    [AllowAnonymous] // Дозволяє неавторизований доступ (якщо логіка дозволяє публічний
доступ)
    [HttpGet] // HTTP GET метод для отримання ресурсів
    [ProducesResponseType(typeof(PaginationResult<VehicleModelDto>),
StatusCodes.Status200OK)] // Документування успішної відповіді (пагінований список DTO)
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)] //
Документування відповіді з помилкою клієнта (наприклад, невірні параметри)
    public async Task<ActionResult<PaginationResult<VehicleModelDto>>> GetModels(
        [FromQuery] VehicleModelFilteringRequestParameters requestParameters) // Параметри
фільтрації та пагінації з рядка запиту
    {
        // Делегування обробки запиту на отримання колекції до MediatR Sender
        var result = await Sender.Send(new
CollectionVehicleModelsQueryRequest(requestParameters));

        // Обробка результату: якщо успіх, повернути дані з OK (200) статусом
        // Обробка помилок (наприклад, валідації) може бути зроблена в базовому контролері
або пайплайні
        return Ok(result.Value);
    }

    // ... інші ендпоінти VehicleModelController
```

Джерело: [створено автором]

Аналогічно, UserController відповідає за функціонал, пов'язаний з користувачами, включаючи доступ до їхніх персональних даних та списків (наприклад, списку бажаних оголошень).

Лістинг 2.5.2. Фрагмент UserController.cs - Приклад ендпоінту отримання списку бажаних оголошень користувача

```
[Route("api/user")] // Маршрут контролера для ресурсів користувачів
[ApiController] // Атрибут для контролерів API
public sealed class UserController : BaseApiController // Успадкування від базового
контролера
{
	// Приклад ендпоінту отримання списку бажаних оголошень
	[HasAccessPermission("users:read")] // Атрибут авторизації: вимагає дозволу
"users:read"
	[HttpGet("wishlist")] // HTTP GET метод для ресурсу "wishlist"
	[ProducesResponseType(typeof(PaginationResult<VehicleDto>),
StatusCodes.Status200OK)] // Документування успішної відповіді
	[ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status404NotFound)] //
Документування відповіді з помилкою "Не знайдено" (наприклад, користувача)
	public async Task<ActionResult<PaginationResult<VehicleDto>>> GetUserWishlist(
	[FromQuery] UserWishlistFilteringRequestParameters parameters) // Параметри
фільтрації/пагінації для списку бажаного
	{
	// Делегування обробки запиту на отримання списку бажаного до MediatR Sender
	var result = await Sender.Send(new GetUserWishlistQueryRequest(parameters));

	// Обробка результату: якщо успіх, повернути дані; якщо помилка, використати базовий
метод для ProblemDetails
	if (!result.IsSuccess)
	{
		// Використання методу базового контролера для форматування помилки
ProblemDetails
		return NotFound(CreateNotFoundProblemDetails(result.Error!));
	}

	// Форматування успішної відповіді
	return Ok(result.Value);
	}
}
```

Джерело: [створено автором]

Як видно з прикладів (Лістинг 2.5.1. та 2.5.2.), контролери мають чисту структуру, а їхня основна роль зводиться до прийому запиту, виконання необхідних перевірок (наприклад, авторизації за допомогою атрибута [HasAccessPermission]), делегування його обробки до відповідного обробника через MediatR Sender та форматування фінальної відповіді на основі об'єкта Result, отриманого від обробника.

2.5.2. Реалізація бізнес-логіки: Хендлери та Сервіси (CQRS)

Обробка бізнес-логіки, ініційована запитами з рівня контролерів (через Sender.Send()), здійснюється у шарі Application за допомогою обробників (Handlers) у відповідності до патерну CQRS. Кожен об'єкт запиту або команди, що передається через MediatR (IRequest, ICommandRequest, IQueryRequest), має свій відповідний обробник, який реалізує інтерфейс IRequestHandler<TRequest, TResponse>.

Лістинг 2.5.3. Хендлер обробки команди створення моделі автомобіля CreateVehicleModelCommandRequestHandler.cs

```
[Route("api/user")] // Маршрут контролера для ресурсів користувачів
[ApiController] // Атрибут для контролерів API
public sealed class UserController : BaseApiController // Успадкування від базового
контролера
{
	// Приклад ендпоінту отримання списку бажаних оголошень
	[HasAccessPermission("users:read")] // Атрибут авторизації: вимагає дозволу
"users:read"
	[HttpGet("wishlist")] // HTTP GET метод для ресурсу "wishlist"
	[ProducesResponseType(typeof(PaginationResult<VehicleDto>),
StatusCodes.Status200OK)] // Документування успішної відповіді
	[ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status404NotFound)] //
Документування відповіді з помилкою "Не знайдено" (наприклад, користувача)
	public async Task<ActionResult<PaginationResult<VehicleDto>>> GetUserWishlist(
```

Джерело: [створено автором]

Аналогічно, обробники запитів (Queries) відповідають за отримання даних, часто делегуючи виконання до відповідних сервісів. Наприклад, GetUserWishlistQueryRequestHandler обробляє запит на отримання списку бажаних оголошень користувача.

Лістинг 2.5.4. Хендлер обробки запиту отримання списку бажаних оголошень GetUserWishlistQueryRequestHandler.cs

```
// Хендлер запиту отримання списку бажаного, залежить від IUserContext та IUserService
internal sealed class GetUserWishlistQueryRequestHandler(IUserContext userContext,
IUserService userService) :
    IRequestHandler<GetUserWishlistQueryRequest, Result<PaginationResult<VehicleDto>>> //
Обробляє GetUserWishlistQueryRequest і повертає Result<PaginationResult<VehicleDto>>
{
    // Метод обробки запиту
    public async Task<Result<PaginationResult<VehicleDto>>> Handle(
```

```
        GetUserWishlistQueryRequest request,
        CancellationToken cancellationToken)
    {
        // Отримання ID поточного користувача з контексту безпеки та встановлення його у
параметри фільтрації запиту
        // Це робиться тут, щоб не виставляти UserId назовні в запиті, а отримувати його з
довіреного джерела
        (request.FilteringParameters as IUserWishlistFilteringRequestParameters)!.UserId =
userContext.UserId;

        // Делегування бізнес-логіки отримання списку бажаного до сервісу
        var wishlistVehicles = await userService.GetUserWishlist(request,
cancellationToken);

        // Формування та повернення успішного результату з даними та параметрами пагінації
        // Успішний результат обертається в об'єкт Result
        return Result<PaginationResult<VehicleDto>>.Success(
            new PaginationResult<VehicleDto>(wishlistVehicles,
request.FilteringParameters.PageIndex));
    }
}
```

Джерело: [створено автором]

Як ілюструють приклади (Лістинг 2.5.3. та 2.5.4.), хендлери, як правило, мають невеликий розмір і їхнє основне завдання – координація виконання бізнес-логіки, часто шляхом делегування до спеціалізованих сервісів або безпосереднього використання команд/запитів шару Persistence.

Більш комплексна бізнес-логіка, що стосується певної сутності домену або функціонального аспекту (наприклад, вся логіка, пов'язана з користувачами, включаючи їхні списки оголошень), та оркестрація взаємодії зі сховищем даних інкапсульована у спеціалізованих сервісах (наприклад, IUserService). Ці сервіси реалізують інтерфейси, визначені у шарі Application, та використовують Command/Query об'єкти, визначені у шарі Persistence, для доступу до даних. Такий підхід відповідає принципу єдиної відповідальності (Single Responsibility Principle – SRP) та сприяє кращій підтримуваності коду.

Лістинг 2.5.5. Фрагмент реалізації сервісу UserService.cs (методи, пов'язані зі списком бажаного)

```
// Реалізація сервісу користувачів, що інкапсулює бізнес-логіку та використовує
Command/Query об'єкти
```

```
internal sealed class UserService(
    AddUserCommand addUserCommand, // Команда для додавання користувача
    ChangeUserRoleCommand changeUserRoleCommand, // Команда для зміни ролі користувача
    SaveChangesCommand saveChangesCommand, // Команда для збереження змін
    GetUserWishlistQuery getUserWishlistQuery, // Запит для отримання списку бажаного
    CountUserWishlistVehiclesQuery countUserWishlistVehiclesQuery) : IUserService //
Реалізація інтерфейсу сервісу користувачів
{
    // ... інші методи сервісу (наприклад, AddAsync, ChangeUserRole), які використовують
відповідні команди та SaveChangesCommand ...

    // Метод отримання списку бажаних оголошень користувача
    // Делегує виконання до об'єкта GetUserWishlistQuery шару Persistence
    public async Task<IReadOnlyCollection<VehicleDto>> GetUserWishlist(
        GetUserWishlistQueryRequest request,
        CancellationToken cancellationToken = default)
    {
        return await getUserWishlistQuery.Execute(request, cancellationToken);
    }

    // Метод отримання кількості оголошень у списку бажаного користувача
    // Делегує виконання до об'єкта CountUserWishlistVehiclesQuery шару Persistence
    public async Task<ItemsCountDto>
GetWishlistItemsCount(CountUserWishlistVehiclesQueryRequest request,
        CancellationToken cancellationToken = default)
    {
        return await countUserWishlistVehiclesQuery.Execute(request, cancellationToken);
    }
}
```

Джерело: [створено автором]

Сервіс UserService (фрагмент у Лістинг 2.5.5.), ін'єктує об'єкти команд та запитів з шару Persistence (GetUserWishlistQuery, CountUserWishlistVehiclesQuery тощо) та використовує їх для виконання необхідних операцій, інкапсулюючи бізнес-логіку управління користувачами та їхніми списками. Це є класичним застосуванням патерну CQRS у шарі Application, де сервіси оркеструють взаємодію з шаром даних. Кожна команда та запит у шарі Persistence реалізує логіку, необхідну для однієї, атомарної операції з даними, повністю відповідаючи принципу єдиної відповідальності (SRP).

2.5.3. Імплементація механізму пошуку оголошень та оцінки релевантності

Одним із критично важливих функціональних модулів маркетплейсу є система пошуку оголошень, яка має забезпечити швидке та релевантне знаходження автомобілів за запитами користувачів. Ефективна система пошуку є ключовим фактором для забезпечення позитивного користувацького досвіду.

Аналіз підходів до реалізації пошукового механізму: При проектуванні системи пошуку для маркетплейсу Kolesko було проведено аналіз двох основних стратегій: використання вбудованого механізму повнотекстового пошуку (Full-Text Search - FTS) та розробка власної логіки пошуку з використанням нормалізованих даних та індексів. Вбудований механізм FTS (наприклад, PostgreSQL[19]) пропонує переваги у морфологічному аналізі та оптимізовану продуктивність, але має обмеження у кастомізації ранжування та складність інтеграції зі специфічною бізнес-логікою. Кастомна логіка з нормалізованими даними, хоча й вимагає власної розробки алгоритмів токенізації та нормалізації, надає повний контроль над процесом пошуку та ранжування, що є критично важливим для точного співставлення результатів з потребами користувачів на автомобільному маркетплейсі.

На основі проведеного аналізу було прийнято рішення про використання власної логіки пошуку з нормалізованими даними, що забезпечує максимальну гнучкість та повний контроль над процесом обробки пошукових запитів та ранжування результатів відповідно до специфічних вимог бізнес-логіки маркетплейсу.

Реалізований механізм пошуку включає декілька ключових компонентів:

1. **Токенізація та нормалізація пошукових запитів:** Процес токенізації вхідного пошукового запиту реалізовано з урахуванням можливих розділових знаків та регістру, перетворюючи запит на набір нормалізованих токенів.

Лістинг 2.5.6. Логіка розбиття запиту на токени (ключові слова)

```
// Вхідний параметр searchTerm - рядок пошукового запиту від користувача
// Необхідні using-директиви: using System.Text.RegularExpressions; using System.Linq;
using System.Collections.Generic;

var tokens = Regex.Split(searchTerm.ToLower(), @"[\s,.;:\-\|]+") // Розбиття рядка за
регулярним виразом (пробіли, пунктуація)
      .Where(token => !string.IsNullOrEmpty(token)) // Фільтрація порожніх токенів
      .ToList(); // Перетворення результату у список токенів
```

Джерело: [створено автором]

2. **Формування пошукових предикатів:** Система використовує динамічне формування предикатів (умов фільтрації) для запиту до бази даних за допомогою пакету PredicateBuilder. Це дозволяє створювати складні WHERE умови, що перевіряють наявність пошукових токенів у різних полях оголошення (марка, модель, колір тощо) з використанням регістронезалежного пошуку (EF.Functions.ILike) та підтримкою часткових збігів (%token%).

Лістинг 2.5.7. Логіка створення пошукового предикату для запиту до бази даних

```
// Вхідний параметр tokens - список нормалізованих токенів пошукового запиту
// Ініціалізація нового предикату з умовою true (для подальшого об'єднання за допомогою
OR)
var predicate = PredicateBuilder.New<Vehicle>(true); // Починаємо з true, щоб перший OR
працював коректно

// Побудова предикату шляхом об'єднання умов для кожного токена за допомогою оператора OR
predicate = tokens.Select(token => $"%{token}%").Aggregate(predicate,
    (current, tokenPattern) =>
        current.Or(vehicle => // Додавання нової умови через OR
            // Перевірка на наявність токена (частковий збіг) у різних полях оголошення
            EF.Functions.ILike(vehicle.Brand.Name, tokenPattern) || // Регістронезалежний
пошук за назвою марки
            EF.Functions.ILike(vehicle.Model.Name, tokenPattern) || // за назвою моделі
            EF.Functions.ILike(vehicle.Color.Name, tokenPattern) || // за назвою кольору
            EF.Functions.ILike(vehicle.BodyType.Name, tokenPattern) || // за типом кузову
            EF.Functions.ILike(vehicle.TransmissionType.Name, tokenPattern) || // за типом
трансмісії
            EF.Functions.ILike(vehicle.EngineType.Name, tokenPattern) || // за типом
двигуна
            EF.Functions.ILike(vehicle.ProductionYear.ToString(), tokenPattern) // за
роком випуску (як рядок)
        )
);
```

Джерело: [створено автором]

3. **Система оцінки релевантності результатів:** Після отримання оголошень, що відповідають предикату, виконується їх ранжування. Впроваджено багаторівневу систему оцінки релевантності, яка присвоює вагові бали оголошенням на основі того, наскільки точно та в яких полях були знайдені пошукові токени. Це забезпечує відображення найбільш релевантних результатів вище у списку.

Лістинг 2.5.8. Логіка оцінки релевантності результатів

```
// Вхідний параметр queryable - IQueryable<Vehicle> з оголошеннями, що вже відфільтровані за предикатом пошуку
// Вхідний параметр tokens - список нормалізованих токенів пошукового запиту

return queryable
       // .Where(predicate) // Застосування сформованого предикату фільтрації (може бути застосовано раніше)
       .OrderByDescending(v => // Сортування результатів за спаданням оцінки релевантності
       (tokens.Any(t => v.Brand.Name.ToLower() == t) ? 100 : 0) + // Точний збіг токена з назвою марки (вага 100)
       (tokens.Any(t => v.Model.Name.ToLower() == t) ? 100 : 0) + // Точний збіг токена з назвою моделі (вага 100)
       tokens.Count(t => v.Brand.Name.ToLower().Contains(t)) * 20 + // Частковий збіг токена в назві марки (вага 20 за кожен збіг)
       tokens.Count(t => v.Model.Name.ToLower().Contains(t)) * 20 + // Частковий збіг токена в назві моделі (вага 20 за кожен збіг)
       tokens.Count(t => v.Color.Name.ToLower().Contains(t)) * 10 + // Частковий збіг токена в назві кольору (вага 10)
       tokens.Count(t => v.BodyType.Name.ToLower().Contains(t)) * 10 + // Частковий збіг токена в назві типу кузову (вага 10)
       (tokens.Any(t => v.ProductionYear.ToString().Contains(t)) ? 15 : 0) // Частковий збіг токена в році випуску (вага 15, якщо є хоча б один)
       );
```

Джерело: [створено автором]

2.5.4. Управління списками бажаних оголошень користувачів

Функціонал управління списками бажаних оголошень дозволяє користувачам зберігати оголошення, які їх зацікавили, для подальшого перегляду та керування ними, включаючи додавання та видалення оголошень зі свого списку. Реалізація цього функціоналу відповідає загальній архітектурі

застосунку з використанням патерну CQRS та делегуванням логіки між шарами Application та Persistence.

Взаємодія з цим функціоналом відбувається через контролери UserController (для перегляду списку та його кількості) та VehiclesController (для додавання/видалення оголошень зі списку). Контролери приймають запити від клієнта та делегують їх обробку до відповідних обробників у шарі Application через MediatR Sender.

Операції отримання списку бажаного та його кількості реалізуються як запити (Queries), що обробляються відповідними обробниками та компонентами сервісу IUserService. Наприклад, GetUserWishlistQueryRequestHandler обробляє запит на отримання списку бажаних оголошень.

Лістинг 2.5.9. Хендлер обробки запиту отримання списку бажаних оголошень GetUserWishlistQueryRequestHandler.cs

```
// Хендлер запиту отримання списку бажаного, залежить від IUserContext та IUserService
internal sealed class GetUserWishlistQueryRequestHandler(IUserContext userContext,
IUserService userService) :
    IRequestHandler<GetUserWishlistQueryRequest, Result<PaginationResult<VehicleDto>>> //
Обробляє GetUserWishlistQueryRequest і повертає Result<PaginationResult<VehicleDto>>
{
    // Метод обробки запиту
    public async Task<Result<PaginationResult<VehicleDto>>> Handle(
        GetUserWishlistQueryRequest request,
        CancellationToken cancellationToken)
    {
        // Отримання ID поточного користувача та встановлення його у параметри фільтрації
запиту
        // Це робиться тут, щоб не виставляти UserId назовні в запиті, а отримувати його з
довіреного джерела
        (request.FilteringParameters as IUserWishlistFilteringRequestParameters)!.UserId =
userContext.UserId;

        // Делегування бізнес-логіки отримання списку бажаного до сервісу
        var wishlistVehicles = await userService.GetUserWishlist(request,
cancellationToken);

        // Формування та повернення успішного результату з даними та параметрами пагінації
        // Успішний результат обертається в об'єкт Result
        return Result<PaginationResult<VehicleDto>>.Success(
            new PaginationResult<VehicleDto>(wishlistVehicles,
request.FilteringParameters.PageIndex));
```

```
    }
}
```

Джерело: [створено автором]

Операції додавання та видалення оголошення зі списку бажаного реалізуються як команди (Commands), що обробляються відповідними обробниками: AddVehicleToWishlistCommandRequestHandler та RemoveVehicleFromWishlistCommandRequestHandler.

Лістинг 2.5.10. Хендлер команди додавання оголошення до списку бажаного AddVehicleToWishlistCommandRequestHandler.cs

```
// Хендлер запиту отримання списку бажаного, залежить від IUserContext та IUserService
internal sealed class GetUserWishlistQueryRequestHandler(IUserContext userContext,
IUserService userService) :
    IRequestHandler<GetUserWishlistQueryRequest, Result<PaginationResult<VehicleDto>>> //
Обробляє GetUserWishlistQueryRequest і повертає Result<PaginationResult<VehicleDto>>
{
    // Метод обробки запиту
    public async Task<Result<PaginationResult<VehicleDto>>> Handle(
        GetUserWishlistQueryRequest request,
        CancellationToken cancellationToken)
    {
        // Отримання ID поточного користувача з контексту та встановлення його у параметри
фільтрації запиту
        // Це робиться тут, щоб не виставляти UserId назовні в запиті, а отримувати його з
довіреного джерела
        (request.FilteringParameters as IUserWishlistFilteringRequestParameters)!.UserId =
userContext.UserId;

        // Делегування бізнес-логіки отримання списку бажаного до сервісу
        var wishlistVehicles = await userService.GetUserWishlist(request,
cancellationToken);

        // Формування та повернення успішного результату з даними та параметрами пагінації
        // Успішний результат обертається в об'єкт Result
        return Result<PaginationResult<VehicleDto>>.Success(
            new PaginationResult<VehicleDto>(wishlistVehicles,
request.FilteringParameters.PageIndex));
    }
}
```

Джерело: [створено автором]

Лістинг 2.5.11. Хендлер команди видалення оголошення зі списку бажаного RemoveVehicleFromWishlistCommandRequestHandler.cs

```
// Хендлер команди видалення оголошення зі списку бажаного, залежить від IVehicleService
та IUserContext
internal sealed class RemoveVehicleFromWishlistCommandRequestHandler(
    IVehicleService vehicleService, // Сервіс для роботи з оголошеннями
    IUserContext userContext) : IRequestHandler<RemoveVehicleFromWishlistCommandRequest,
Result> // Обробляє команду і повертає Result (без даних)
{
    // Метод обробки команди
    public async Task<Result> Handle(RemoveVehicleFromWishlistCommandRequest request,
CancellationToken cancellationToken)
    {
        try
        {
            // Отримання ID поточного користувача з контексту безпеки
            var requestingUserId = userContext.UserId;

            // Делегування бізнес-логіки видалення зі списку бажаного до сервісу
            await vehicleService.RemoveVehicleFromWishlist(request.VehicleId,
requestingUserId, cancellationToken);

            // Повернення успішного результату
            return Result.Create(Error.None); // Error.None означає успіх у Result об'єкті
        }
        catch (KeyNotFoundException) // Обробка випадку, коли оголошення не знайдено (або
користувача, якщо сервіс кине)
        {
            // Повернення результату з помилкою "Не знайдено"
            return Result.Create(Error.NotFound(request.VehicleId));
        }
    }
}
```

Джерело: [створено автором]

Обробники команд додавання (Лістинг 2.5.10.) та видалення (Лістинг 2.5.11.) оголошення зі списку бажаного отримують ID оголошення від контролера. Вони використовують IUserContext для безпечного отримання ідентифікатора поточного авторизованого користувача та делегують подальшу обробку до сервісу IVehicleService, передаючи йому обидва ідентифікатори.

Сервіси (IUserService, IVehicleService) інкапсулюють складнішу бізнес-логіку та оркестрацію взаємодії з шаром Persistence. Сервіс IUserService містить методи для отримання списку бажаного та його кількості, делегуючи їх виконання до об'єктів запитів шару Persistence.

Оскільки операції додавання та видалення оголошення зі списку бажаного є діями над *оголошенням* (з боку користувача), логіка цих операцій

інкапсульована у сервісі VehicleService, який відповідає за всі аспекти роботи з оголошеннями. Цей сервіс ін'єктує відповідні команди шару Persistence для модифікації даних.

Реалізація основного функціоналу серверної частини онлайн-маркетплейсу "Kolesko" демонструє послідовне та ефективне застосування обраних архітектурних патернів та технологій. Структура API на базі Controller-based підходу забезпечує чітке розділення відповідальності, де контролери делегують обробку запитів та команд шару Application через MediatR. Бізнес-логіка інкапсульована у хендлерах та сервісах, які, у свою чергу, взаємодіють з шаром Persistence для доступу до даних за допомогою Command/Query об'єктів. Така організація дозволила ефективно реалізувати ключові функціональні модулі, включаючи складний механізм пошуку оголошень з детальною токенізацією, формуванням предикатів та оцінкою релевантності, а також функціонал управління списками бажаних оголошень користувачів (перегляд, додавання, видалення). Кожен компонент реалізовано з урахуванням принципів чистої архітектури, що забезпечує високу підтримуваність, тестованість та масштабованість серверної частини застосунку.

2.6. Впровадження модульного та інтеграційного тестування серверної частини

Забезпечення високої якості програмного забезпечення є наріжним каменем у розробці надійних та підтримуваних систем. У контексті розробки серверної частини онлайн-маркетплейсу "Kolesko" цьому аспекту було приділено значну увагу шляхом впровадження комплексної системи тестування, що охоплює як модульний (unit), так і інтеграційний (integrational) рівні. Такий підхід дозволив гарантувати коректність роботи окремих компонентів системи та ефективність їх взаємодії, що є критично важливим для стабільного функціонування платформи.

2.6.1. Вибір інструментарію для тестування

При проектуванні та реалізації системи тестування серверної частини застосунку "Kolesko" було обрано набір сучасних інструментів, кожен з яких виконує специфічну роль у процесі забезпечення якості коду. Для покриття потреб як модульного, так і інтеграційного тестування було інтегровано наступні технології:

- **xUnit:** Виступає як основний фреймворк для написання та організації тестових сценаріїв обох рівнів.
- **NSubstitute:** Використовується для створення замінників (мок-об'єктів та стабів) зовнішніх залежностей, що є особливо важливим для ізоляції компонентів під час модульного тестування.
- **Fluent Assertions:** Надає гнучкий та виразний синтаксис для формулювання тверджень (assertions) у тестових методах, підвищуючи їх читабельність.
- **Faker (Bogus):** Застосовується для генерації різноманітних реалістичних тестових даних, що дозволяє моделювати різні сценарії використання системи.
- **Testcontainers:** Ключовий інструмент для інтеграційного тестування, що дозволяє створювати ізольовані та реалістичні тестові середовища на базі контейнерів Docker.

2.6.2. Реалізація модульного тестування

Модульне тестування є базовим рівнем тестування, спрямованим на перевірку коректності функціонування окремих, ізольованих компонентів або модулів коду (наприклад, класів, методів). У проєкті "Kolesko" значний обсяг unit-тестів було розроблено для перевірки бізнес-логіки, реалізованої у шарі Application (зокрема, обробників команд та запитів).

В якості основного фреймворку для модульного тестування обрано **xUnit**. Його переваги, такі як[20]: лаконічний синтаксис, підтримка паралельного виконання тестів, можливість створення параметризованих тестів (за допомогою атрибута [Theory] у поєднанні з джерелами даних) та відсутність глобального стану між тестами (забезпечується створенням нового екземпляра тестового класу для кожного тестового методу), зробили його ефективним інструментом для розробки масштабованої системи unit-тестів.

Лістинг 2.6.1. Приклад unit-тесту обробника запиту з використанням xUnit

```
public sealed class CountVehicleBrandsQueryRequestHandlerTests
{
    // Оголошення мок-об'єкта для залежного сервісу та обробника запиту
    private readonly IVehicleBrandService _vehicleBrandService =
Substitute.For<IVehicleBrandService>();
    private readonly CountVehicleBrandsQueryRequestHandler _handler;
    // Об'єкт запиту для тестування
    private readonly CountVehicleBrandsQueryRequest _request = new(new
VehicleBrandFilteringRequestParameters());

    // Конструктор тестового класу для ініціалізації об'єктів
    public CountVehicleBrandsQueryRequestHandlerTests()
    {
        _handler = new CountVehicleBrandsQueryRequestHandler(_vehicleBrandService);
    }

    // Тестовий метод (Fact) для перевірки виклику залежного сервісу
    [Fact]
    public async Task Handler_ShouldInvokeGetVehicleBrandsCount_FromVehicleBrandService()
    {
        // Arrange (підготовка) - об'єкти вже ініціалізовано в конструкторі

        // Act (виконання дії) - виклик методу обробника запиту
        await _handler.Handle(_request, default);

        // Assert (перевірка) - перевірка, чи був викликаний відповідний метод мок-об'єкта
        await _vehicleBrandService
            .Received() // Перевірка, що метод був викликаний
            .GetVehicleBrandsCount(_request, default!); // Вказання методу та аргументів
    }
}
```

Джерело: [створено автором]

Для ізоляції тестованих компонентів від їхніх зовнішніх залежностей (сервісів, репозиторіїв) активно використовується бібліотека **NSubstitute**. Вона дозволяє легко створювати замінники (substitution instances) для інтерфейсів та абстрактних класів. Ключові переваги NSubstitute[21]: інтуїтивно зрозумілий синтаксис для налаштування поведінки замінників (.Returns()) та перевірки викликів (.Received()), підтримка мокування асинхронних методів (Task, Task<T>) та зручні засоби для перевірки кількості та аргументів викликів.

Лістинг 2.6.2. Приклад unit-тесту з використанням NSubstitute та Fluent Assertions

```
public sealed class CollectionVehicleTypesQueryRequestHandlerTests
{
    // Оголошення мок-об'єкта сервісу типів автомобілів
    private readonly IVehicleTypeService _vehicleTypeService =
```

```
Substitute.For<IVehicleTypeService>();
    // Обробник запиту та об'єкт запиту
    private readonly CollectionVehicleTypesQueryRequestHandler _handler;
    private readonly CollectionVehicleTypesQueryRequest _request = new(new
VehicleTypeFilteringRequestParameters());

    // Конструктор для ініціалізації
    public CollectionVehicleTypesQueryRequestHandlerTests()
    {
        _handler = new CollectionVehicleTypesQueryRequestHandler(_vehicleTypeService);
    }

    // Тестовий метод
    [Fact]
    public async Task Handler_ShouldReturnCollectionOfVehicleTypes()
    {
        // Arrange (підготовка) - створення очікуваних даних
        List<VehicleTypeDto> expectedTypes =
        [
            new VehicleTypeDto { Id = Guid.NewGuid(), Name = "Легковик" },
            new VehicleTypeDto { Id = Guid.NewGuid(), Name = "Вантажівка" },
            new VehicleTypeDto { Id = Guid.NewGuid(), Name = "Автобус" }
        ];
        // Формування очікуваного результату пагінації
        var expectedResult = new PaginationResult<VehicleTypeDto>(
            expectedTypes,
            _request.FilteringParameters.PageIndex);

        // Налаштування поведінки мок-об'єкта: що повернути при виклику певного методу
        _vehicleTypeService.GetVehicleTypeCollection(_request, default)
            .Returns(expectedTypes);

        // Act (виконання дії) - виклик методу обробника
        var result = await _handler.Handle(_request, default);

        // Assert (перевірка) - використання Fluent Assertions для перевірки результату
        result.IsSuccess // Перевірка, що результат успішний
            .Should().BeTrue(); // Має бути True

        result.Value // Значення результату
            .Should() // Має
            .BeEquivalentTo(expectedResult); // Бути еквівалентним очікуваному результату
(для об'єктів/колекцій)
    }
}
```

Джерело: [створено автором]

Для забезпечення чіткої та зрозумілої перевірки результатів виконання тестів інтегровано бібліотеку **Fluent Assertions 7**. Вона пропонує декларативний підхід до формулювання тверджень, що робить код тестів більш читабельним та виразним завдяки[22]: виразному ланцюжковому синтаксису (.Should().Be(...)), детальним повідомленням про помилки при невдалих

перевірках, спеціалізованим методам для різних типів даних та підтримці перевірки асинхронних операцій.

Лістинг 2.6.3. Приклад unit-тесту з використанням Fluent Assertions та NSubstitute (перевірка виклику)

```
public sealed class CollectionVehicleEngineTypesQueryRequestHandlerTests
{
    // Оголошення мок-об'єкта сервісу типів двигунів
    private readonly IVehicleEngineTypeService _vehicleEngineTypeService =
        Substitute.For<IVehicleEngineTypeService>();
    // Обробник запиту та об'єкт запиту
    private readonly CollectionVehicleEngineTypesQueryRequestHandler _handler;
    private readonly CollectionVehicleEngineTypesQueryRequest _request = new(new
VehicleEngineTypeFilteringRequestParameters());

    // Конструктор для ініціалізації
    public CollectionVehicleEngineTypesQueryRequestHandlerTests()
    {
        _handler = new
CollectionVehicleEngineTypesQueryRequestHandler(_vehicleEngineTypeService);
    }

    // Тестовий метод (Fact) для перевірки виклику залежного сервісу
    [Fact]
    public async Task
Handler_ShouldInvokeGetVehicleEngineTypeCollection_FromVehicleEngineTypeService()
    {
        // Arrange (підготовка) - об'єкти вже ініціалізовано в конструкторі

        // Act (виконання дії) - виклик методу обробника запиту
        await _handler.Handle(_request, default);

        // Assert (перевірка) - використання Fluent Assertions та NSubstitute для
перевірки виклику методу
        await _vehicleEngineTypeService // Мок-об'єкт
            .Received() // Має бути викликаний
            .GetVehicleEngineTypeCollection(_request, default!); // Вказання методу та
аргументів
    }
}
```

Джерело: [створено автором]

Для генерації різноманітних тестових даних, що відображали б різні сценарії використання системи, було впроваджено бібліотеку **Faker** або ж **Bogus**. Вона дозволяє генерувати реалістичні дані різних типів (імена, email, номери телефонів тощо), що робить тестування більш наближеним до реальних умов[23]. Крім того, Faker підтримує відтворюваність тестів шляхом встановлення початкового значення для генератора випадкових чисел (seed).

Лістинг 2.6.4. Приклад unit-тесту з використанням Faker для генерації тестових даних

```
public async Task Send_Should_ReturnSuccessResult_IfUserExists()
{
    // Arrange (підготовка) - генерація тестових даних користувача за допомогою Faker
    // RegisterUserOnKeycloakAndAddToDb - допоміжний метод для створення тестового
користувача у Keycloak та БД
    var userRepresentation = await RegisterUserOnKeycloakAndAddToDb(
        Faker.Internet.Email(), // Генерація випадкового email
        Faker.Internet.Password(), // Генерація випадкового пароля
        Faker.Phone.UkrainianPhoneNumber(), // Генерація випадкового українського номера
телефону
        UserRole.User); // Вказання ролі

    // Створення об'єкта запиту з використанням згенерованого ID користувача
    var queryRequest = new GetUserDataQueryRequest(userRepresentation.Id!);

    // Act (виконання дії) - відправка запиту через Sender (імітація виклику від
контролера)
    var result = await Sender.Send(queryRequest);

    // Assert (перевірка) - використання Fluent Assertions для перевірки результату
    result.Error // Перевірка об'єкта помилки
        .Should().BeEquivalentTo(Error.None); // Має бути еквівалентним Error.None
(означає успіх)

    result.IsSuccess.Should().BeTrue(); // Явна перевірка успіху

    result.Value // Перевірка значення результату (отриманих даних користувача)
        .Should().NotBeNull(); // Не має бути Null
    result.Value!.Email.Equals(userRepresentation.Email,
StringComparison.OrdinalIgnoreCase) // Перевірка email
        .Should().BeTrue(); // Має відповідати email, що використовувався при створенні
}
```

Джерело: [створено автором]

Впровадження зазначеного інструментарію дозволило створити ефективну систему модульного тестування, що сприяла підвищенню якості коду, спрощенню процесу рефакторингу та полегшенню подальшої підтримки окремих компонентів системи.

2.6.3. Реалізація інтеграційного тестування

Інтеграційне тестування є наступним рівнем тестування після модульного і спрямоване на перевірку коректності взаємодії між різними компонентами системи або між системою та зовнішніми сервісами. У складній екосистемі маркетплейсу "Kolesko", де численні шари та сервіси взаємодіють між собою та

із зовнішніми залежностями (наприклад, база даних), інтеграційне тестування стало критично важливим для забезпечення стабільної роботи всієї платформи та виявлення проблем, які не можуть бути виявлені на рівні unit-тестів (наприклад, некоректна робота з базою даних, помилки конфігурації). Інтеграційні тести дозволили перевірити реальну взаємодію, включаючи: взаємодію із системою управління базами даних (PostgreSQL), взаємодію зі сторонніми сервісами (через імітацію їх відповідей або запуск тестових екземплярів) та обробку реальних HTTP-запитів API із формуванням відповідей.

Ключовою технологією, що забезпечила ефективність інтеграційного тестування, стала бібліотека **Testcontainers**. Ця технологія дозволяє створювати ізольовані контейнери Docker (наприклад, з базою даних PostgreSQL або іншими сервісами) безпосередньо з коду тестів, що забезпечує повністю самодостатнє, реалістичне та відтворюване тестове середовище[24]. Основними перевагами використання Testcontainers є: повна ізоляція тестового середовища для кожного тестового запуску, реалістичне тестування завдяки використанню реальних екземплярів залежностей (а не моків), декларативне налаштування необхідної інфраструктури та підтримка паралельного виконання тестів.

Фабрика IntegrationTestsWebApplicationFactory використовується для створення ізольованого екземпляра веб-додатку ASP.NET Core для кожного тестового класу (або колекції тестів), підключаючи його до тимчасового контейнера бази даних PostgreSQL, запущеного за допомогою Testcontainers. Це забезпечує реалістичне тестове середовище, максимально наближене до продуктивного, але повністю ізольоване від інших тестів та локальних ресурсів розробника.

Інтеграційні тести пишуться з використанням фреймворку xUnit та фабрики IntegrationTestsWebApplicationFactory. Вони взаємодіють із застосунком через

HTTP-клієнт або безпосередньо викликаючи обробники запитів/команд через MediatR (якщо це внутрішні інтеграційні тести між шарами).

Лістинг 2.6.6. Приклад інтеграційного тесту обробника запиту з використанням WebApplicationFactory та Testcontainers

```
// Тестовий клас інтеграційних тестів для запитів оголошень
// Успадковується від IntegrationTestBase, який надає доступ до Sender та фабрики
public sealed class VehicleQueriesIntegrationTests(
    IntegrationTestsWebApplicationFactory factory) : IntegrationTestBase(factory) //
Ін'єкція фабрики
{
    // Тестовий метод для перевірки отримання фільтрованих оголошень
    [Fact]
    public async Task Send_CollectionQueryRequest_Should_ReturnFilteredVehicles()
    {
        // Arrange (підготовка) - створення параметрів фільтрації
        var filteringParameters = new VehicleFilteringRequestParameters();
        // Створення об'єкта запиту
        var queryRequest = new CollectionVehiclesQueryRequest(filteringParameters);

        // Act (виконання дії) - відправка запиту через MediatR Sender тестового хоста
        var response = await Sender.Send(queryRequest);

        // Assert (перевірка) - використання Fluent Assertions для перевірки результату
        response.Error // Перевірка об'єкта помилки
            .Should().Be(Error.None); // Має бути Error.None (успіх)
        response.IsSuccess // Перевірка прапора успіху
            .Should().BeTrue(); // Має бути True
        response.Value.Items // Перевірка колекції оголошень у результаті
            .Should().NotBeEmpty(); // Не має бути порожньою
    }
}
```

Джерело: [створено автором]

Інтеграційні тест (Лістинг 2.6.6) перевіряє роботу компонентів у їх взаємодії, використовуючи реальну базу даних (запущену в контейнері Testcontainers) та реальну імплементацію залежностей (якщо вони не були спеціально замінені для тестування конкретного сценарію інтеграції).

2.6.4. Забезпечення високого рівня покриття коду

Одним із ключових показників якості тестування є покриття коду тестами (Code Coverage). У проєкті "Kolesko" завдяки впровадженню як модульних, так і інтеграційних тестів, вдалося досягти надзвичайно високого рівня покриття

коду — 95%, що було підтверджено аналізом за допомогою інструменту dotCover від JetBrains.

Такий високий рівень покриття було досягнуто завдяки комплексному підходу, що включав:

1. **Систематичний підхід до тестування:** Тести розроблялися перед створенням нового функціоналу, слідуючи методології Test-Driven Development
2. **Комбінування різних типів тестів:** Модульні тести забезпечували глибоку перевірку окремих компонентів в ізоляції, тоді як інтеграційні тести верифікували коректність їх взаємодії в умовах, наближених до реальних.
3. **Автоматизація процесу тестування:** CI/CD пайплайни включали автоматичне виконання всіх тестів та аналіз покриття коду при кожній зміні, забезпечуючи постійний контроль якості.
4. **Культура якості в команді:** Впроваджені стандарти розробки та політики code review вимагали обов'язкової наявності тестів для нового або зміненого коду.

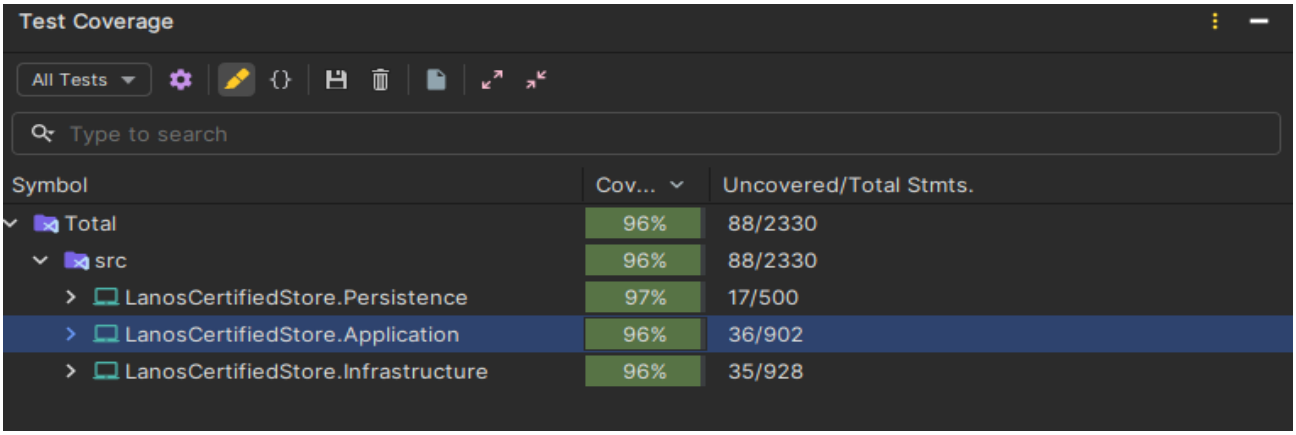

Рис. 2.6.1. Візуалізація відсотку покриття коду підпроектів застосунку, отримана за допомогою JetBrains dotCover. *Джерело: [створено автором]*

Досягнення покриття коду на рівні 95% є свідченням високої зрілості процесу розробки та забезпечує значну впевненість у стабільності та надійності функціонування серверної частини маркетплейсу "Kolesko". Такий високий рівень покриття не тільки зменшує кількість дефектів у продуктивному середовищі, але й суттєво спрощує процес рефакторингу та подальшого розвитку системи.

Висновки до розділу №2

Другий розділ роботи детально висвітлює прикладну частину розробки серверної частини онлайн-маркетплейсу вживаних автомобілів "Kolesko", втілюючи в життя теоретичні засади та архітектурні рішення, визначені у першому розділі.

На початковому етапі реалізації (підпункт 2.1) було обґрунтовано та здійснено вибір конкретної технологічної платформи для реалізації API (.NET) та обрано підсистему управління ідентифікацією та доступом (IAM) на базі Keycloak. Також підтверджено застосування архітектурного патерну CQRS у поєднанні з бібліотекою MediatR як основи для організації бізнес-логіки. Далі описано процес створення та базового налаштування проєкту (підпункт 2.2), а також конфігурації необхідної інфраструктури API (підпункт 2.3), що заклало технічний фундамент для подальшої розробки.

Значну увагу (підпункт 2.4) приділено практичному впровадженню системи управління ідентифікацією та доступом на базі Keycloak в якості обраного IAM рішення. Деталізовано доступні механізми аутентифікації та реєстрації користувачів, розглянуто процес верифікації облікових записів електронної пошти, а також висвітлено можливості розширення стандартного функціоналу Keycloak за допомогою кастомних компонентів. Цей аспект є критично важливим для забезпечення безпеки та ефективного управління користувачами платформи.

Представлено реалізацію основного бізнес-функціоналу серверної частини (підпункт 2.5). Детально описано структуру API ендпоінтів та механізм делегування обробки запитів у відповідності до принципів CQRS (підпункти 2.5.1, 2.5.2). Окремо розглянуто реалізацію специфічних функціональних модулів: імплементацію механізму пошуку оголошень з оцінкою релевантності (підпункт 2.5.3) та функціонал управління списками бажаних оголошень користувачів, що охоплює операції перегляду, додавання та видалення (підпункт 2.5.4).

Завершальною частиною розділу є опис впровадження системи забезпечення якості коду через модульне та інтеграційне тестування (підпункт 2.6). Обґрунтовано вибір інструментарію тестування (підпункт 2.6.1), деталізовано підходи до реалізації модульних (підпункт 2.6.2) та інтеграційних тестів (підпункт 2.6.3), а також продемонстровано досягнення високого рівня покриття коду як показника ефективності розробленої системи тестування (підпункт 2.6.4).

Таким чином, у другому розділі продемонстровано комплексний процес практичної реалізації серверної частини онлайн-маркетплейсу "Kolesko". Це включає вибір конкретних технологій та архітектурних підходів на етапі імплементації, базове налаштування інфраструктури, детальне впровадження системи управління ідентифікацією та доступом на базі Keycloak, реалізацію ключових бізнес-функцій та розробку системи тестування, що забезпечує верифікацію розробленого функціоналу. Результати розділу підтверджують створення функціональної, безпечної, надійної та добре протестованої серверної основи для платформи, готової до подальшого розвитку.

ВИСНОВКИ

Підсумовуючи результати виконаної кваліфікаційної роботи, слід зазначити, що було досягнуто основної мети – розробки серверної частини онлайн-маркетплейсу вживаних автомобілів "Kolesko" з урахуванням сучасних архітектурних підходів, вимог до функціональності, безпеки та якості програмного забезпечення.

У першому розділі роботи проведено комплексне дослідження теоретичних засад та факторів, що визначають розробку сучасної онлайн-платформи для торгівлі автомобілями. Проаналізовано актуальні тренди українського вторинного автомобільного ринку, визначено ключові потреби користувачів та обґрунтовано актуальність створення маркетплейсу. На основі аналізу провідних архітектурних стилів API обґрунтовано вибір REST як найбільш відповідного стилю для даного проєкту та визначено ключові компоненти загальної архітектури серверної частини, включаючи стек технологій та основні патерни проектування. Також здійснено аналіз ринкового середовища та конкурентів для формування стратегічного позиціонування "Kolesko".

У другому розділі роботи здійснено практичну реалізацію визначених архітектурних рішень та функціональних вимог. Детально описано процес вибору та застосування конкретних технологій та інструментарію, що відповідають обраній архітектурі (.NET, Entity Framework Core, MediatR, Swagger, FluentValidation тощо). Виконано налаштування базової інфраструктури проєкту та API. Впроваджено ключовий бізнес-функціонал серверної частини, який включає ефективні механізми пошуку оголошень з оцінкою релевантності, що є критично важливим для зручності користувачів, а також функціонал управління списками бажаних оголошень користувачів. Особливу увагу приділено реалізації надійної системи управління ідентифікацією та доступом, яка була успішно впроваджена на базі централізованого сервісу Keycloak, забезпечуючи безпеку та розмежування прав доступу.

Для забезпечення високої якості розробленого програмного рішення, впроваджено комплексну багаторівневу систему тестування, що охоплює як модульне (unit-testing), так і інтеграційне (integration-testing) тестування. Продемонстровано застосування сучасного інструментарію (xUnit, NSubstitute, Fluent Assertions, Faker) для модульного тестування та використання Testcontainers для створення ізольованих та реалістичних тестових середовищ для інтеграційних тестів. Як результат системного підходу до розробки та тестування, досягнуто високого рівня покриття коду застосунку тестами (95%), що підтверджує його надійність та стабільність.

Таким чином, виконана робота демонструє спроможність застосування сучасних методологій, архітектурних підходів (REST, Clean Architecture, CQRS) та технологій (.NET, Keycloak, Testcontainers) для створення масштабованих, безпечних, функціональних та надійних серверних рішень в умовах реальних вимог ринку. Розроблена серверна частина маркетплейсу "Kolesko" успішно реалізує визначений ключовий бізнес-функціонал, відповідає вимогам до якості та безпеки, що створює міцну технічну основу для подальшого розвитку проєкту та його потенційного розгортання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Аналітичне дослідження вторинного авторинку України станом на жовтень 2024 року. [Електронний ресурс] – URL: Що з цінами на вторинному авторинку? Підсумки жовтня 2024 — Eauto (дата звернення: 05.12.2024 р.).
2. RESTful web API design. [Електронний ресурс] – URL: Web API design best practices - Azure Architecture Center | Microsoft Learn (дата звернення: 05.12.2025 р.).
3. GraphQL | A query language for your API. [Електронний ресурс] – URL: GraphQL | A query language for your API (дата звернення: 05.12.2025 р.).
4. Introduction to GraphQL | GraphQL. [Електронний ресурс] – URL: Introduction to GraphQL | GraphQL (дата звернення: 05.12.2025 р.).
5. What is SOAP API: Formats, Protocols, and Architecture. [Електронний ресурс] – URL: What is SOAP API: Formats, Protocols, and Architecture (дата звернення: 05.12.2025 р.).
6. RST: онлайн-автобазар України. [Електронний ресурс] – URL: Продається на RST — Купити авто в Україні — авторинок RST, автобазар України - автопродаж на РСТ, продаж бу авто (дата звернення: 08.12.2025 р.).
7. Автобазар Україна: продаж та купівля авто. [Електронний ресурс] – URL: Автобазар Україна. Продаж та купівля авто. Сайт оголошень з купля продажу автомобілів (дата звернення: 08.12.2025 р.).
8. Introduction to .NET - .NET | Microsoft Learn. [Електронний ресурс] – URL: Introduction to .NET - .NET | Microsoft Learn (дата звернення: 12.01.2025 р.).
9. Complete Guide to Clean Architecture - GeeksforGeeks. [Електронний ресурс] – URL: Complete Guide to Clean Architecture - GeeksforGeeks (дата звернення: 13.01.2025 р.).
10. Keycloak | Open Source Identity and Access Management [Електронний ресурс] – URL: Keycloak | Open Source Identity and Access Management (дата звернення: 02.02.2025 р.).
11. CQRS pattern - Azure Architecture Center | Microsoft Learn. [Електронний ресурс] – URL: CQRS pattern - Azure Architecture Center | Microsoft Learn (дата звернення: 13.01.2025 р.).
12. Simplifying Projects with a Folder-by-Feature Structure | by The C2 Group | The C2 Group | Medium [Електронний ресурс] – URL: Simplifying Projects with a Folder-by-Feature Structure | by The C2 Group | The C2 Group | Medium (дата звернення: 14.01.2025 р.).
13. APIs overview | Microsoft Learn. [Електронний ресурс] – URL: APIs overview | Microsoft Learn (дата звернення: 13.01.2025 р.).
14. Serilog — simple .NET logging with fully-structured events [Електронний ресурс] – URL: Serilog — simple .NET logging with fully-structured events (дата звернення: 27.02.2025 р.).
15. Seq — centralized structured logs [Електронний ресурс] – URL: Seq — centralized structured logs (дата звернення: 04.03.2025 р.).

16. FluentValidation — FluentValidation documentation [Електронний ресурс] – URL: FluentValidation — FluentValidation documentation (дата звернення: 10.03.2025р.).
17. Authorization Services Guide | Keycloak [Електронний ресурс] – URL: Authorization Services Guide | Keycloak (дата звернення: 18.02.2025 р.).
18. Custom extensions on Keycloack | Cloud-IAM | DOCS [Електронний ресурс] – URL: Custom extensions on Keycloack | Cloud-IAM | DOCS (дата звернення: 24.02.2025 р.).
19. PostgreSQL: Documentation: 17: 12.1. Introduction. [Електронний ресурс] – URL: PostgreSQL: Documentation: 17: 12.1. Introduction (дата звернення: 16.03.2025 р.).
20. Getting started with xUnit.net v3 (command line) > xUnit.net [Електронний ресурс] – URL: Getting started with xUnit.net v3 (command line) > xUnit.net (дата звернення: 18.12.2024р.).
21. NSubstitute: Getting started [Електронний ресурс] – URL: NSubstitute: Getting started (дата звернення: 21.12.2024 р.).
22. Introduction - Fluent Assertions [Електронний ресурс] – URL: Introduction - Fluent Assertions (дата звернення: 23.12.2024 р.).
23. Faker.net by Kuree [Електронний ресурс] – URL: Faker.net by Kuree (дата звернення: 26.12.2024 р.).
24. Getting Started - Testcontainers [Електронний ресурс] – URL: Getting Started-Testcontainers (дата звернення: 26.12.2024 р.).

ДОДАТКИ

ДОДАТОК А.

Приклад кастомного провайдера для прослуховування подій Keycloak

```
package com.cevher.keycloak;

import java.util.Map;
import org.jboss.logging.Logger;
import org.keycloak.events.Event;
import org.keycloak.events.EventListenerProvider;
import org.keycloak.events.EventType;
import org.keycloak.events.admin.AdminEvent;
import org.keycloak.events.admin.OperationType;
import org.keycloak.events.admin.ResourceType;
import org.keycloak.models.KeycloakSession;
import org.keycloak.models.RealmModel;
import org.keycloak.models.RealmProvider;
import org.keycloak.models.UserModel;

// Кастомний провайдер для прослуховування подій Keycloak
public class CustomEventListenerProvider implements EventListenerProvider {
  private static final Logger log = Logger.getLogger(CustomEventListenerProvider.class);

  private final KeycloakSession session;
  private final RealmProvider model;

  // Конструктор для отримання KeycloakSession
  public CustomEventListenerProvider(KeycloakSession session) {
      this.session = session;
      this.model = session.realms();
  }

  // Обробка подій користувача (реєстрація, вхід тощо)
  @Override // Явне позначення перевантаження стандартного інтерфейсу
  public void onEvent(Event event) {
      log.debugf("New %s Event", event.getType());
      // Виклик методу обробки даних користувача при реєстрації
      if (EventType.REGISTER.equals(event.getType())) {
      log.debugf("Processing REGISTER event for user ID: %s", event.getUserId());
      RealmModel realm = this.model.getRealm(event.getRealmId());
      UserModel user = this.session.users().getUserById(realm, event.getUserId());
      if (user != null) {
      sendUserData(user); // Передача даних користувача для подальшої обробки
      } else {
      log.errorf("User not found for REGISTER event with ID: %s", event.getUserId());
      }
      }
      // Можна додати обробку інших типів подій користувача, якщо потрібно (наприклад,
LOGIN, LOGOUT)
  }

  // Обробка адміністративних подій
```

```
  @Override // Явне позначення перевантаження стандартного інтерфейсу
  public void onEvent(AdminEvent adminEvent, boolean b) {
       log.debug("onEvent(AdminEvent)");
       // Виклик методу обробки даних користувача при створенні користувача адміністратором
через Admin API
       if (ResourceType.USER.equals(adminEvent.getResourceType()) &&
         OperationType.CREATE.equals(adminEvent.getOperationType())) {
       log.debugf("Processing Admin CREATE USER event for resource path: %s",
adminEvent.getResourcePath());
       RealmModel realm = this.model.getRealm(adminEvent.getRealmId());
       // Отримання ID користувача з resourcePath та пошук користувача
       // ResourcePath для створення користувача Admin API часто має формат users/<userID>
       String userId = adminEvent.getResourcePath().substring("users/".length());
       UserModel user = this.session.users().getUserById(realm, userId);
       if (user != null) {
       sendUserData(user); // Передача даних користувача для подальшої обробки
       } else {
       log.errorf("User not found for Admin CREATE USER event with ID: %s from path %s",
userId, adminEvent.getResourcePath());
       }
       }
       // Можна додати обробку інших типів адміністративних подій, якщо потрібно
(наприклад, DELETE USER, UPDATE USER)
  }

  // Метод для відправлення даних користувача до зовнішньої системи (наприклад, бекенд
API)
  private void sendUserData(UserModel user) {
       // Отримання необхідних даних користувача (наприклад, ID, email, username)
       String userId = user.getId();
       String userEmail = user.getEmail();
       String userName = user.getUsername();

       log.debugf("Sending data for user ID: %s, Email: %s, Username: %s to external
service", userId, userEmail, userName);

       try {
       // Приклад: Виклик зовнішнього API або сервісу бекенду для синхронізації даних
       // Припустимо, існує клас Client для взаємодії з бекендом
       // Client.postService(userId, userEmail, userName);
       log.debug("User data successfully sent to external service.");
       } catch (Exception e) {
       // Логування помилки у випадку невдачі
       log.errorf("Failed to call external service for user ID %s: %s", userId, e);
       }
  }

  // Метод, що викликається при закритті сесії провайдера
  @Override
  public void close() {
       // Логіка очищення ресурсів, якщо потрібно
  }

  // Приклади методів toString для логування подій (не є основною логікою)
  // @Override public String toString(Event event) { /* ... */ return ""; }
  // @Override public String toString(AdminEvent event) { /* ... */ return ""; }
```

ДОДАТОК Б.

Приклад кастомного валідатора даних у Keycloak

```
package com.dawidgora.provider;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;
import org.keycloak.models.KeycloakSession;
import org.keycloak.models.RealmModel;
import org.keycloak.models.UserModel;
import org.keycloak.models.UserProvider;
import org.keycloak.provider.ConfiguredProvider;
import org.keycloak.provider.ProviderConfigProperty;
import org.keycloak.validate.AbstractStringValidator;
import org.keycloak.validate.ValidationContext;
import org.keycloak.validate.ValidationError;
import org.keycloak.validate.ValidatorConfig;

public class UniqueAttributeValidatorProvider extends AbstractStringValidator implements
ConfiguredProvider {
  public static final String ID = "unique-attribute";
  public static final String MESSAGE_ATTRIBUTE_NOT_UNIQUE = "error.duplicated.attribute";

  private static final List<ProviderConfigProperty> configProperties = new ArrayList<>();

  public String getId() {
    return ID; // Використання константи для ID валідатора
  }

  @Override // Перевизначення методу валідації рядкового атрибута
  protected void doValidate(String attributeValue, String attributeName, ValidationContext
context, ValidatorConfig config) {
    KeycloakSession session = context.getSession();
    // Отримання поточного користувача, якщо контекст пов'язаний з існуючим користувачем
(наприклад, при редагуванні профілю)
    UserModel currentUser =
(UserModel)context.getAttributes().get(UserModel.class.getName());

    // Виклик логіки перевірки унікальності
    if (!isAttributeUnique(attributeValue, attributeName, session, currentUser))
      // Додавання помилки валідації, якщо атрибут не унікальний
      context.addError(new ValidationError(ID, attributeName, MESSAGE_ATTRIBUTE_NOT_UNIQUE
+ "." + attributeName));
  }

  // Метод перевірки унікальності атрибута
  public boolean isAttributeUnique(String attributeValue, String attributeName,
KeycloakSession session, UserModel currentUser) {
    UserProvider userProvider = session.getProvider(UserProvider.class); // Отримання
провайдера користувачів Keycloak
    RealmModel realm = getRealm(session); // Отримання поточного реалму

    // Пошук користувачів з таким же значенням атрибута
    Stream<UserModel> usersWithSameAttributeValue =
userProvider.searchForUserByUserAttributeStream(realm, attributeName, attributeValue);

    // Фільтрація поточного користувача (для коректної валідації при оновленні) та
перевірка наявності будь-якого іншого користувача з таким атрибутом
```

```
    return (currentUser != null) ?
      usersWithSameAttributeValue.filter(user ->
!user.getId().equals(currentUser.getId())).findAny().isEmpty() : // Якщо користувач існує
(оновлення), перевірити, чи є інші користувачі з таким атрибутом
      usersWithSameAttributeValue.findAny().isEmpty(); // Якщо користувача немає
(створення), перевірити, чи є будь-який користувач з таким атрибутом
  }

  public String getHelpText() {
    return "Validator to ensure attribute uniqueness across users in the realm."; // Більш
описовий текст допомоги
  }

  // Метод для отримання об'єкта RealmModel (може бути опущений у фінальному документі)
  RealmModel getRealm(KeycloakSession session) { /* ... */ return null; }

  public List<ProviderConfigProperty> getConfigProperties() {
    return configProperties;
  }
}
```

ДОДАТОК В.

Приклад фабрики для ініціалізації інтеграційних тестів за допомогою Testcontainers та WebApplicationFactory

```
public sealed class IntegrationTestsWebApplicationFactory :
WebApplicationFactory<Program>, IAsyncLifetime
{
    // Оголошення контейнера PostgreSQL
    private readonly PostgreSqlContainer _dbContainer = new PostgreSqlBuilder()
        .WithImage("postgres") // Використання стандартного образу PostgreSQL
        .WithDatabase("LanosCertifiedStore") // Назва бази даних
        .WithUsername("postgres") // Користувач
        .WithPassword("postgres") // Пароль
        .Build(); // Побудова конфігурації контейнера

    // Метод для конфігурації веб-хоста тестового застосунку
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        // Виклик базової реалізації
        base.ConfigureWebHost(builder);

        // Отримання рядка підключення до контейнера бази даних Testcontainers
        var connectionString = _dbContainer.GetConnectionString() + "; " + "Include Error
Detail=true";

        // Встановлення рядка підключення як змінної середовища, яку прочитає тестовий
застосунок
        Environment.SetEnvironmentVariable(
            "ConnectionStrings:PostgreSqlConnection",
            connectionString
        );

        // Додаткова конфігурація сервісів тестового хоста (наприклад, відключення
```

```
логування)
        builder.ConfigureTestServices(services =>
        {
            // Silence logging for integration tests
            services.AddSingleton<ILoggerFactory, NullLoggerFactory>();
        });
    }

    // Асинхронний метод ініціалізації (запускається перед виконанням тестів)
    public async Task InitializeAsync()
    {
        // Запуск контейнера бази даних
        await _dbContainer.StartAsync();
    }

    // Асинхронний метод очищення (запускається після виконання всіх тестів)
    public new async Task DisposeAsync()
    {
        // Зупинка контейнера бази даних
        await _dbContainer.StopAsync();
        // Виклик базової реалізації DisposeAsync (рекомендовано)
        await base.DisposeAsync();
    }
}
```

ДОДАТОК Г.

Приклад повної реалізації VehicleController

```
using LanosCertifiedStore.Application.Images;
using
LanosCertifiedStore.Application.Images.Commands.AddImageToVehicleCommandRequestRelated;
using
LanosCertifiedStore.Application.Images.Commands.RemoveImageFromVehicleCommandRequestRelate
d;
using
LanosCertifiedStore.Application.Images.Commands.SetVehicleMainImageCommandRequestRelated;
using LanosCertifiedStore.Application.Shared.DtosRelated;
using LanosCertifiedStore.Application.Shared.RequestParamsRelated;
using LanosCertifiedStore.Application.Shared.ResultRelated;
using LanosCertifiedStore.Application.Shared.ValidationRelated;
using LanosCertifiedStore.Application.Vehicles;
using
LanosCertifiedStore.Application.Vehicles.Commands.AddVehicleToWishlistCommandRequestRelate
d;
using
LanosCertifiedStore.Application.Vehicles.Commands.CreateVehicleCommandRequestRelated;
using
LanosCertifiedStore.Application.Vehicles.Commands.DeleteVehicleCommandRequestRelated;
using
LanosCertifiedStore.Application.Vehicles.Commands.RemoveVehicleFromWishlistCommandRequestR
elated;
using
LanosCertifiedStore.Application.Vehicles.Commands.UpdateVehicleCommandRequestRelated;
using LanosCertifiedStore.Application.Vehicles.Dtos;
using LanosCertifiedStore.Application.Vehicles.Queries.CollectionVehiclesQueryRelated;
```

```
using LanosCertifiedStore.Application.Vehicles.Queries.CountVehiclesQueryRelated;
using LanosCertifiedStore.Application.Vehicles.Queries.SearchVehiclesQueryRelated;
using LanosCertifiedStore.Application.Vehicles.Queries.SingleVehicleQueryRequestRelated;
using LanosCertifiedStore.Application.Vehicles.Queries.VehiclePriceRangeQueryRelated;
using LanosCertifiedStore.Infrastructure.Authorization;
using LanosCertifiedStore.Persistence.Commands.VehicleRelated;
using LanosCertifiedStore.Presentation.Controllers.Common;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;

namespace LanosCertifiedStore.Presentation.Controllers.VehicleRelated;

[Route("api/vehicles")]
public sealed class VehiclesController : BaseApiController
{
    [AllowAnonymous]
    [HttpGet]
    [ProducesResponseType(typeof(PaginationResult<VehicleDto>), StatusCodes.Status200OK)]
    public async Task<ActionResult<PaginationResult<VehicleDto>>> GetVehicles(
        [FromQuery] VehicleFilteringRequestParameters requestParameters)
    {
        var result = await Sender.Send(new
CollectionVehiclesQueryRequest(requestParameters));

        return Ok(result.Value);
    }

    [AllowAnonymous]
    [HttpGet("find")]
    [ProducesResponseType(typeof(PaginationResult<SearchVehicleDto>),
StatusCodes.Status200OK)]
    public async Task<ActionResult<PaginationResult<SearchVehicleDto>>> SearchVehicles(
        [FromQuery] string searchTerm,
        ItemQuantitySelection itemQuantity = ItemQuantitySelection.Ten,
        int pageIndex = 1)
    {
        var requestParameters = new SearchVehicleFilteringRequestParameters
        {
            ItemQuantity = itemQuantity,
            PageIndex = pageIndex,
            SearchTerm = searchTerm,
        };

        var result = await Sender.Send(new SearchVehiclesQueryRequest(requestParameters));

        return Ok(result.Value);
    }

    [AllowAnonymous]
    [HttpGet("{id:guid}", Name = "GetVehicleById")]
    [ProducesResponseType(typeof(SingleVehicleDto), StatusCodes.Status200OK)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status404NotFound)]
    public async Task<ActionResult<SingleVehicleDto>> GetVehicle(Guid id)
    {
        var result = await Sender.Send(new SingleVehicleQueryRequest(id));

        if (!result.IsSuccess)
        {
            return NotFound(CreateNotFoundProblemDetails(result.Error!));
        }
```

```
        return Ok(result.Value);
    }

    [AllowAnonymous]
    [HttpGet("count")]
    [ProducesResponseType(typeof(ItemsCountDto), StatusCodes.Status200OK)]
    public async Task<ActionResult<ItemsCountDto>> GetItemsCount(
        [FromQuery] VehicleFilteringRequestParameters requestParameters)
    {
        var result = await Sender.Send(new CountVehiclesQueryRequest(requestParameters));

        return Ok(result.Value);
    }

    [AllowAnonymous]
    [HttpGet("price-range")]
    [ProducesResponseType(typeof(PriceRangeDto), StatusCodes.Status200OK)]
    public async Task<ActionResult<PriceRangeDto>> GetPriceRange(
        [FromQuery] VehicleFilteringRequestParameters requestParameters)
    {
        var result = await Sender.Send(new
VehiclePriceRangeQueryRequest(requestParameters));

        return Ok(result.Value);
    }

    [HasAccessPermission("vehicles:create")]
    [HttpPost]
    [ProducesResponseType(StatusCodes.Status201Created)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)]
    public async Task<ActionResult<Guid>> CreateVehicle(CreateVehicleCommandRequest
request)
    {
        var result = await Sender.Send(request);

        if (result is IValidationResult validationResult)
        {
            return BadRequest(CreateValidationProblemDetails(result.Error!,
validationResult.Errors));
        }

        return CreatedAtRoute("GetVehicleById", new { id = result.Value }, result.Value);
    }

    [HasAccessPermission("users:read")]
    [HttpPost("{vehicleId:guid}/add-to-wishlist")]
    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)]
    public async Task<ActionResult> AddVehicleToWishlist(Guid vehicleId)
    {
        var request = new AddVehicleToWishlistCommandRequest(vehicleId);
        var result = await Sender.Send(request);

        if (result is IValidationResult validationResult)
        {
            return BadRequest(CreateValidationProblemDetails(result.Error!,
validationResult.Errors));
        }
```

```
        return NoContent();
    }

    [HasAccessPermission("users:read")]
    [HttpPost("{vehicleId:guid}/remove-from-wishlist")]
    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)]
    public async Task<ActionResult> RemoveVehicleFromWishlist(Guid vehicleId)
    {
        var request = new RemoveVehicleFromWishlistCommandRequest(vehicleId);
        var result = await Sender.Send(request);

        if (result is IValidationResult validationResult)
        {
            return BadRequest(CreateValidationProblemDetails(result.Error!,
validationResult.Errors));
        }

        return NoContent();
    }

    [HttpPut("{id:guid}")]
    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status404NotFound)]
    public async Task<ActionResult> UpdateVehicle(Guid id, [FromBody]
UpdateVehicleCommandRequest request)
    {
        request = request with { Id = id };

        var result = await Sender.Send(request);

        if (result is IValidationResult validationResult)
        {
            return BadRequest(CreateValidationProblemDetails(result.Error!,
validationResult.Errors));
        }

        if (!result.IsSuccess)
        {
            return NotFound(CreateNotFoundProblemDetails(result.Error!));
        }

        return NoContent();
    }

    [HttpDelete("{id:guid}")]
    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)]
    public async Task<ActionResult> DeleteVehicle(Guid id)
    {
        var result = await Sender.Send(new DeleteVehicleCommandRequest(id));

        if (!result.IsSuccess)
        {
            return NotFound(CreateNotFoundProblemDetails(result.Error!));
        }

        return NoContent();
    }
```

```
    [ProducesResponseType(StatusCodes.Status200OK)]
    [ProducesResponseType(typeof(Error), StatusCodes.Status200OK)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status400BadRequest)]
    [HttpPost("{id:guid}/images")]
    public async Task<ActionResult> AddImageToVehicle(Guid id, List<IFormFile> images)
    {
        var request = new AddImagesToVehicleCommandRequest(id, images);

        var result = await Sender.Send(request);

        if (result is IValidationResult validationResult)
        {
            return BadRequest(CreateValidationProblemDetails(result.Error!,
validationResult.Errors));
        }

        if (result.Error == Error.None)
        {
            return Ok();
        }

        return Ok(result.Error);
    }

    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status404NotFound)]
    [HttpDelete("{vehicleId:guid}/images")]
    public async Task<ActionResult> DeleteImageFromVehicle(Guid vehicleId, [FromQuery]
string imageId)
    {
        var result = await Sender.Send(new RemoveImageFromVehicleCommandRequest(vehicleId,
imageId));

        if (result.IsSuccess)
        {
            return NoContent();
        }

        if (result.Error == Error.NotFound(vehicleId) || result.Error ==
ImageErrors.NotFound(imageId))
        {
            return NotFound(CreateNotFoundProblemDetails(result.Error!));
        }

        return BadRequest(result.Error);
    }

    [ProducesResponseType(StatusCodes.Status204NoContent)]
    [ProducesResponseType(typeof(ProblemDetails), StatusCodes.Status404NotFound)]
    [HttpPost("{vehicleId:guid}/images/main")]
    public async Task<ActionResult> SetVehicleMainImage(Guid vehicleId, [FromQuery] string
imageId)
    {
        var result = await Sender.Send(new SetVehicleMainImageCommandRequest(vehicleId,
imageId));

        if (result.IsSuccess)
        {
            return NoContent();
```

```
        }

        if (result.Error == Error.NotFound(vehicleId) || result.Error ==
ImageErrors.NotFound(imageId))
        {
            return NotFound(CreateNotFoundProblemDetails(result.Error!));
        }

        return BadRequest(result.Error);
    }
}
```