ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Web-додаток для управління автопарком приватного підприємства за допомогою python та typescript»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Технології цифрового розвитку»

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

_____ Дмирто ГРИЦАНЮК
(підпис)

Виконав: здобувач(ка) вищої освіти групи ТЦР-42

Дмитро ГРИЦАНЮК

Керівник: Вадим ЧИТУЛЯН
ст. викладач кафедри

Рецензент:

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Технологій цифрового розвитку

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Технології цифрового розвитку»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Технологій цифрового розвитку
_____ Вікторія ЖЕБКА
«____»_____2026 року

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Грицанюку Дмитру Сергійовичу

1.Тема кваліфікаційної роботи: «Web-додаток для управління автопарком приватного підприємства за допомогою Python та TypeScript»
керівник кваліфікаційної роботи старший викладач кафедри Вадим ЧИТУЛЯН, затверджені наказом Державного університету інформаційно-комунікаційних технологій від «___» березня 2026 року № ____.

2. Строк подання кваліфікаційної роботи «___» травня 2026 року.

3. Вхідні дані до роботи:

3.1. Середовище розробки: Visual Studio Code

3.2. Backend: Python 3.12, Django 5.x, Django REST Framework

3.3. Frontend: TypeScript, React 18+, React Router v6, React Query

3.4. СУБД: PostgreSQL 16

3.5. Інфраструктура: Docker, Docker Compose

3.6. Автентифікація: JWT (access + refresh токени)

3.7. Тестування: pytest, DRF APIClient, factory-boy

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити).

4.1. Аналіз бізнес-процесів управління автопарком приватного підприємства

4.2. Аналіз та порівняння існуючих програмних рішень у сфері управління автопарком

4.3. Обґрунтування вибору технологічного стеку для розробки web-застосунку

4.4. Проєктування архітектури системи, реляційної моделі даних та REST API

4.5. Розробка серверної частини на основі Django та Django REST Framework

4.6. Розробка клієнтської частини на основі React та TypeScript

4.7. Налаштування інфраструктури розгортання за допомогою Docker та Docker Compose

4.8. Тестування розробленого програмного забезпечення

5. Перелік графічного матеріалу

5.1. Мета, об’єкт, предмет дослідження

5.2. Актуальність теми

5.3. Огляд та порівняльний аналіз існуючих рішень

5.4. Функціональні та нефункціональні вимоги до системи

5.5. Архітектура системи (deployment diagram)

5.6. ER-діаграма моделі даних

5.7. UML-діаграми поведінки (Use Case, Sequence, Class)

5.8. Специфікація REST API ендпоінтів

5.9. Засоби реалізації та технологічний стек

5.10. Інтерфейс застосунку (дашборд, модуль звітності)

5.11. Результати тестування

5.12. Висновки

6. Дата видачі завдання «20» лютого 2026 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Підбір науково-технічної літератури	19.04.26 - 22.04.26	
2	Аналіз існуючих аналогів	22.04.26 - 23.04.26	
3	Дослідження програмних засобів	23.04.26 - 01.05.26	
4	Моделювання об’єкту проектування	01.05.26 - 05.05.26	
5	Розробка функціоналу за стосунку	05.05.26 - 07.05.26	
6	Вступ, висновки, реферат	07.05.26 - 08.05.26	
7	Розробка презентації за стосунку	08.05.26 - 24.05.26	
8	Попередній захист роботи	25.05.26	

Здобувач(ка) вищої освіти _____ Дмитро ГРИЦАНЮК
(підпис)

Керівник
кваліфікаційної роботи _____ Вадим ЧИТУЛЯН
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 74 стор., 13 табл., 34 рис., 28 джерел.

Мета роботи - підвищення ефективності управління автопарком приватного підприємства шляхом автоматизації обліку транспортних засобів, пального та ремонтів.

Об'єкт дослідження - процеси автоматизації управління автопарком приватного підприємства.

Предмет дослідження - web-додаток для управління автопарком приватного підприємства.

Наукова новизна даної роботи полягає в наступному:

Розроблено web-додаток для комплексної автоматизації управління автопарком приватного підприємства, що об'єднує облік транспортних засобів, водіїв, пального та ремонтів у єдиній інформаційній системі.

Обґрунтовано вибір архітектурного підходу REST API з розподілом на незалежні frontend і backend застосунки як оптимального для забезпечення масштабованості та незалежного розвитку рівнів системи в умовах предметної області управління автопарком.

Показано ефективність поєднання Django REST Framework та React з TypeScript для реалізації типобезпечного інтерфейсу взаємодії між рівнями системи, що знижує кількість помилок інтеграції на етапі розробки.

Розроблено алгоритм автоматичного виявлення аномального споживання пального на основі порівняння фактичної витрати з нормативною.

Упровадження розробленого застосунку дозволяє підприємству вести оперативний облік транспортних засобів і водіїв, контролювати витрати пального, реєструвати ремонти та технічне обслуговування, а також формувати аналітичні звіти, що скорочує час на обробку інформації та мінімізує ризик помилок ручного введення даних.

Під час виконання роботи проаналізовано існуючі програмні рішення у сфері управління автопарком, визначено їх переваги та недоліки, на основі чого сформульовано функціональні та нефункціональні вимоги до системи.

Спроєктовано трирівневу клієнт-серверну архітектуру системи, реляційну модель даних із шести сутностей, специфікацію REST API з 27 ендпоінтів та підсистему безпеки на основі JWT-автентифікації та рольової моделі доступу. Розроблено серверну частину на Django з використанням Service Layer, клієнтську частину на React з TypeScript, а також конфігурацію Docker Compose для відтворюваного розгортання системи.

Сферою використання застосунку є можливість використовуватись менеджерами автопарку, диспетчерами та водіями приватних підприємств, що мають власний рухомий склад і потребують автоматизації процесів обліку транспортних засобів та витрат.

КЛЮЧОВІ СЛОВА: WEB-ДОДАТОК, УПРАВЛІННЯ АВТОПАРКОМ, PYTHON, DJANGO, TYPESCRIPT, REACT, POSTGRESQL, DOCKER, REST API, JWT.

ЗМІСТ

УМОВНІ ПОЗНАЧЕННЯ

ПЗ - програмне забезпечення

БД - база даних

СУБД - система управління базами даних

ТЗ - транспортний засіб

ТО - технічне обслуговування

API - прикладний програмний інтерфейс

REST - передача репрезентативного стану (Representational State Transfer)

JWT - JSON Web Token

HTTP - протокол передачі гіпертексту

URL - уніфікований локатор ресурсу

JSON - JavaScript Object Notation

SPA - односторінковий застосунок (Single Page Application)

ORM - об’єктно-реляційне відображення (Object Relational Mapping)

UML - уніфікована мова моделювання

IDE - інтегроване середовище розробки

UI - інтерфейс користувача (User Interface)

UX - досвід користувача (User Experience)

RBAC - контроль доступу на основі ролей (Role-Based Access Control)

DDL - мова визначення даних (Data Definition Language)

CSS - каскадні таблиці стилів (Cascading Style Sheets)

DRF - Django REST Framework

SaaS - програмне забезпечення як послуга (Software as a Service)

ACID - атомарність, узгодженість, ізольованість, довговічність

XSS - міжсайтовий скриптинг (Cross-Site Scripting)

CSRF - підробка міжсайтових запитів (Cross-Site Request Forgery)

ВСТУП

Невід'ємним елементом господарської діяльності будь-якого приватного підприємства, що має власний рухомий склад, є управління автопарком - комплекс процесів, що охоплює облік транспортних засобів і водіїв, контроль витрат пального, планування технічного обслуговування та ремонтів, а також формування аналітичної звітності для прийняття управлінських рішень. Від якості організації цих процесів безпосередньо залежать операційна ефективність підприємства та рівень його транспортних витрат.

На практиці, однак, більшість малих і середніх підприємств здійснюють зазначені процеси за допомогою паперових журналів або електронних таблиць (Microsoft Excel, Google Sheets). Такий підхід має суттєві обмеження: не забезпечується цілісність даних при паралельній роботі кількох операторів, відсутній контроль версій і аудиторський слід змін, а формування будь-якої зведеної звітності потребує значних ручних зусиль і є схильним до помилок. Зі зростанням кількості транспортних засобів у парку пропорційно зростає й складність ручного обліку: несвоєчасне технічне обслуговування, відсутність оперативного контролю витрат пального та розрізненість даних про ремонти призводять до прихованих фінансових втрат. За оцінками галузевих досліджень, неефективне управління корпоративними автопарками спричиняє перевитрати від 15 до 30 % транспортного бюджету підприємства [28].

Ринок комерційних систем управління автопарком класу Enterprise Fleet Management (Fleetio, Samsara, Geotab) пропонує розвинені рішення, проте їхня вартість і складність впровадження роблять їх економічно недоцільними для малих і середніх підприємств. Вартість ліцензування таких систем нерідко перевищує прогнозований економічний ефект від їхнього впровадження для парків із невеликою кількістю одиниць техніки. Крім того, подібні рішення орієнтовані на універсальне застосування і можуть не враховувати специфічних особливостей

конкретного підприємства - номенклатури ТЗ, внутрішніх регламентів обліку пального чи прийнятих форм звітності.

Альтернативою є розробка спеціалізованого web-застосунку, адаптованого до потреб конкретного підприємства. Сучасний рівень розвитку відкритих технологій дозволяє реалізувати повнофункціональну інформаційну систему з мінімальними витратами на інфраструктуру. Зокрема, використання Python з фреймворком Django та розширенням Django REST Framework на серверній стороні забезпечує швидку розробку надійного RESTful API з вбудованими механізмами автентифікації, валідації даних та адміністрування [3]. На стороні клієнта застосування TypeScript з бібліотекою React дозволяє будувати інтерактивні інтерфейси з суворою типізацією, що суттєво знижує ймовірність помилок на етапі розробки та спрощує підтримку коду [13]. Реляційна СУБД PostgreSQL забезпечує необхідний рівень цілісності даних і підтримку складних аналітичних запитів, а контейнеризація на основі Docker гарантує відтворюваність середовища розгортання незалежно від цільової платформи [4; 24].

Актуальність обраного технологічного стеку підтверджується результатами щорічного опитування Stack Overflow Developer Survey 2024, згідно з яким Django залишається одним із найбільш затребуваних web-фреймворків серед професійних розробників, а TypeScript вперше обійшов JavaScript у категорії найбільш бажаних для вивчення мов програмування [12]. Це свідчить про зрілість екосистеми та широку підтримку спільноти, що є важливим чинником при виборі стеку для довготривалого супроводу системи.

Таким чином, розробка спеціалізованого web-додатку для управління автопарком приватного підприємства є актуальним науково-практичним завданням, яке поєднує дослідження сучасних архітектурних підходів до побудови web-систем із прикладними потребами конкретної організаційно-господарської діяльності.

Об’єкт дослідження - процеси автоматизації управління автопарком приватного підприємства.

Предмет дослідження - web-додаток для управління автопарком приватного підприємства.

Мета роботи - підвищення ефективності управління автопарком приватного підприємства шляхом автоматизації обліку транспортних засобів, пального та ремонтів.

Відповідно до поставленої мети визначено основні завдання дослідження:

1. Проаналізувати процеси управління автопарком підприємства та визначити функціональні й нефункціональні вимоги до програмного забезпечення.
2. Дослідити існуючі програмні рішення у сфері управління автопарком та обґрунтувати вибір технологічного стеку для реалізації системи.
3. Спроєктувати архітектуру системи, реляційну модель даних та специфікацію REST API.
4. Розробити серверну частину застосунку на основі Python (Django, Django REST Framework) та клієнтську частину на основі TypeScript (React).
5. Налаштувати інфраструктуру розгортання системи з використанням Docker та Docker Compose.
6. Провести тестування розробленого програмного забезпечення та проаналізувати відповідність реалізованого функціоналу визначеним вимогам.

Наукова новизна роботи полягає в такому:

1. Розроблено web-додаток для комплексної автоматизації управління автопарком приватного підприємства, що об’єднує облік транспортних засобів, водіїв, пального та ремонтів у єдиній інформаційній системі.
2. Обґрунтовано вибір архітектурного підходу REST API з розподілом на незалежні frontend і backend застосунки як оптимального для забезпечення масштабованості та незалежного розвитку рівнів системи в умовах предметної області управління автопарком.
3. Показано ефективність поєднання Django REST Framework та React з TypeScript для реалізації типобезпечного інтерфейсу взаємодії між рівнями системи, що знижує кількість помилок інтеграції на етапі розробки.

Практична значущість результатів дослідження полягає у вирішенні проблеми розрізненості та некерованості даних про автопарк підприємства шляхом розробки web-додатку, що дозволяє вести оперативний облік транспортних засобів і водіїв, контролювати витрати пального, реєструвати ремонти та технічне обслуговування, а також формувати аналітичні звіти у розрізі транспортних засобів, водіїв та часових періодів. Впровадження системи скорочує час на обробку інформації, мінімізує ризик помилок ручного введення даних та забезпечує керівництво підприємства актуальними даними для прийняття обґрунтованих управлінських рішень. Додаткову практичну цінність становлять задокументовані архітектурні рішення, ER-модель предметної області та конфігурація Docker-середовища, які можуть слугувати відправною точкою для розробки аналогічних систем або подальшого розширення функціональності.

Робота складається зі вступу, трьох розділів, висновків та списку використаних джерел. У **першому розділі** виконано аналіз предметної області та огляд існуючих програмних рішень, сформульовано вимоги до системи та обґрунтовано вибір технологічного стеку. У **другому розділі** викладено проєктні рішення: реляційну модель даних, архітектуру системи та специфікацію REST API. У **третьому розділі** описано реалізацію серверної та клієнтської частин застосунку, налаштування Docker-інфраструктури та результати тестування розробленого програмного забезпечення.

1. АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ОБҐРУНТУВАННЯ ТЕХНОЛОГІЧНИХ РІШЕНЬ

1.1 Аналіз предметної області управління автопарком підприємства

1.1.1 Бізнес-процеси управління автопарком та їх формалізація

Управління автопарком підприємства є багаторівневим організаційно-технічним процесом, що охоплює повний життєвий цикл транспортного засобу від його реєстрації та введення в експлуатацію до списання, а також супутні операційні процеси: контроль витрат пального, планування та облік технічного обслуговування і ремонтів, управління водійським складом та формування аналітичної звітності. Коректна формалізація цих процесів є необхідною передумовою для побудови адекватної інформаційної моделі системи та визначення функціональних вимог до програмного забезпечення .

Процес обліку транспортних засобів. Основним об’єктом предметної області є транспортний засіб (ТЗ), який характеризується сукупністю статичних атрибутів (державний номерний знак, марка, модель, рік випуску, VIN-код, тип палива, нормативна витрата пального) та динамічних атрибутів, що змінюються в процесі експлуатації (поточний пробіг, статус, закріплений водій). Статус ТЗ утворює скінченний автомат із множиною станів: «активний», «на ремонті», «списаний». Переходи між станами ініціюються відповідними подіями: відкриттям заявки на ремонт, завершенням ремонту або актом списання. Реєстрація нового ТЗ в системі передбачає внесення повного набору статичних атрибутів, присвоєння початкового статусу «активний» та фіксацію початкового пробігу.

Процес обліку водійського складу та закріплення ТЗ. Водій є другою ключовою сутністю системи. Атрибути водія охоплюють персональні дані (ПІБ, дата народження, контактна інформація), службові дані (табельний номер, категорія водійського посвідчення, дата його видачі та закінчення терміну дії) та

поточний статус (активний / звільнений). Закріплення транспортного засобу за водієм є відношенням типу «багато-до-одного» у конкретний момент часу (один водій може бути закріплений за кількома ТЗ у різні періоди, але за одним ТЗ одночасно може бути закріплений лише один основний водій). При цьому важливо зберігати історію закріплень для коректного формування звітності витрат у розрізі водіїв.

Процес контролю витрат пального. Облік пального є найбільш операційно інтенсивним процесом і складається з двох компонентів: реєстрації заправок та фіксації фактичних витрат на підставі показань одометра. Кожна заправка реєструється із зазначенням ТЗ, водія, дати та часу, кількості залитого пального (у літрах), ціни за літр та поточного показання одометра. Фактична витрата пального між двома послідовними фіксаціями пробігу розраховується як різниця між сумою залитого пального та зміною залишку в баку, нормована на пройдену відстань. Порівняння фактичної витрати з нормативною дозволяє виявляти аномалії: перевищення норми може свідчити про технічну несправність ТЗ, неефективний стиль водіння або зловживання з боку водія. Зберігання цих даних у структурованій реляційній формі є критично важливим для побудови достовірної аналітики.

Процес управління технічним обслуговуванням і ремонтами. Технічне обслуговування (ТО) є плановим процесом, що ініціюється або за пробігом (кожні N кілометрів), або за календарним інтервалом (кожні M місяців) - залежно від того, яка умова настає раніше. Поточний та капітальний ремонти є позаплановими подіями, що ініціюються виявленням несправності. Кожна ремонтна подія характеризується датою початку та завершення, описом виконаних робіт, переліком використаних запасних частин, вартістю робіт та вартістю матеріалів, а також виконавцем (власна майстерня або стороння СТО). Закриття ремонтної заявки автоматично переводить ТЗ зі стану «на ремонті» у стан «активний» та оновлює дату наступного планового ТО.

Процес формування звітності. Звітність є результуючим процесом, що агрегує дані всіх попередніх процесів. Ключові типи звітів: витрати пального по ТЗ за період; витрати пального по водіях за період; витрати на ремонти по ТЗ; пробіг

ТЗ за період; зведений звіт загальних витрат на утримання ТЗ. Більшість звітів передбачають фільтрацію за часовим діапазоном та групування за обраним виміром (ТЗ, водій, тип ремонту тощо).

Загальну структуру основних бізнес-процесів системи управління автопарком демонструє наступна діаграма:

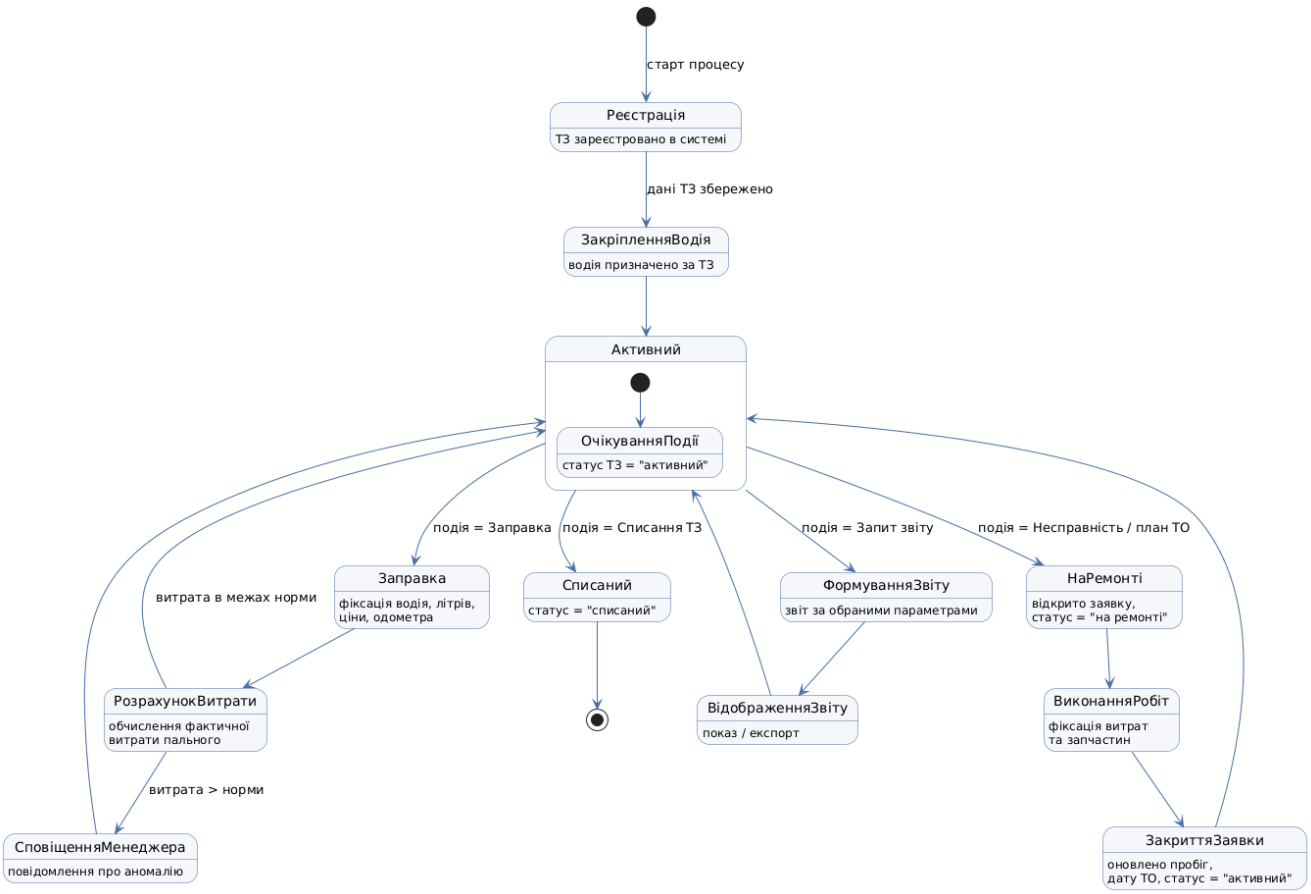


Рис. 1.1 - Діаграма станів основних бізнес-процесів системи управління автопарком

Наведена схема ілюструє циклічний характер операційних процесів: після завершення ремонту або фіксації заправки система повертається до очікування нової події. Єдиним термінальним станом є списання ТЗ. Це підтверджує необхідність реалізації системи як постійно діючої платформи з підтримкою стану, а не як набору ізольованих транзакцій.

Таким чином, у результаті аналізу предметної області виявлено п’ять ключових бізнес-процесів, що підлягають автоматизації, визначено їх взаємозв’язки та характер подієво-орієнтованої взаємодії між ними. Формалізація процесів у вигляді діаграми потоків є необхідною основою для подальшого проєктування реляційної моделі даних та архітектури системи.

1.1.2 Функціональні ролі користувачів системи

Визначення функціональних ролей користувачів є обов’язковим етапом проєктування інформаційної системи, оскільки від розмежування повноважень безпосередньо залежить архітектура підсистеми автентифікації та авторизації, а також склад і доступність API-ендпоінтів [19]. В системі управління автопарком виявлено чотири основні ролі, що відповідають типовій організаційній структурі підприємства з власним рухомим складом.

Адміністратор системи - роль з найширшими повноваженнями, призначена для ІТ-відповідального або керівника підприємства. Адміністратор керує обліковими записами всіх користувачів: створює, редагує та деактивує їх; призначає ролі. Крім того, адміністратор має доступ до всіх модулів системи в режимі читання та запису, включно з довідниками (типи ТЗ, типи палива, типи ремонтів), системними налаштуваннями та журналами аудиту.

Менеджер автопарку - основна операційна роль. Менеджер здійснює реєстрацію та списання транспортних засобів, управління водійським складом (прийом, звільнення, зміна закріплення), планування та облік ТО і ремонтів, а також має повний доступ до модуля звітності. Менеджер не має права керувати обліковими записами інших користувачів - це розмежування є принциповим з точки зору безпеки.

Диспетчер - роль з обмеженим доступом до запису. Диспетчер реєструє заправки та фіксує поточні показання одометра, може переглядати список ТЗ та водіїв, але не має права змінювати їхні атрибути, відкривати ремонтні заявки або формувати звіти. Це відповідає типовій посадовій інструкції диспетчера, який

безпосередньо взаємодіє з водіями, але не несе відповідальності за стратегічні рішення щодо парку.

Водій - роль з мінімальними повноваженнями. Водій може переглядати інформацію про закріплений за ним ТЗ, власний журнал заправок та пробігу. Доступ до даних інших водіїв і ТЗ заблокований. Ця роль є опціональною для невеликих підприємств, де водії не мають доступу до системи, однак її наявність дозволяє масштабувати рішення для підприємств з розвиненою цифровою культурою.

Матрицю доступу ролей до основних функціональних модулів системи наведено в таблиці 1.1.

Таблиця 1.1

Матриця доступу ролей

Модуль	**Адміністратор**	**Менеджер**	**Диспетчер**	**Водій**
Управління користувачами	R/W	-	-	-
Облік ТЗ	R/W	R/W	R	R
Облік водіїв	R/W	R/W	R	R
Заправки та пальне	R/W	R/W	R/W	R
Ремонти та ТО	R/W	R/W	R	-
Звітність	R/W	R/W	-	-
Системні налаштування	R/W	-	-	-

Розмежування доступу на рівні API реалізується засобами Django REST Framework через класи дозволів (permissions.BasePermission), що дозволяє декларативно описувати правила авторизації для кожного ендпоінту без дублювання логіки перевірки. Детальна реалізація підсистеми авторизації розглядається у розділі 3.

Таким чином, визначення чотирьох функціональних ролей із чіткою матрицею доступу забезпечує принцип мінімальних привілеїв (Principle of Least

Privilege), що є базовою вимогою до безпеки інформаційних систем, та визначає вимоги до реалізації підсистеми авторизації на рівні API [19].

1.1.3 Проблеми ручного обліку та обґрунтування необхідності автоматизації

Незважаючи на очевидну потребу в структурованому обліку даних автопарку, значна частина приватних підприємств малого та середнього бізнесу в Україні продовжує використовувати паперові журнали або електронні таблиці Microsoft Excel / Google Sheets як основний інструмент управління рухомим складом. Такий підхід породжує системні проблеми, що безпосередньо впливають на фінансові показники та операційну ефективність підприємства.

Проблема цілісності та узгодженості даних. Електронні таблиці не забезпечують механізмів забезпечення цілісності даних, аналогічних зовнішнім ключам реляційної СУБД. Видалення запису про транспортний засіб не призводить до каскадного видалення або позначення пов’язаних записів про заправки та ремонти як недійсних - вони залишаються «підвішеними» у таблиці та спотворюють агреговані показники. Дослідження у сфері якості даних показують, що в організаціях, які не використовують спеціалізованих інформаційних систем, частка записів з помилками або невідповідностями сягає 20-30 % від загального обсягу даних.

Проблема паралельного доступу. При роботі кількох операторів з одним файлом Excel виникають конфлікти версій: два диспетчери, що одночасно відкрили файл, фактично працюють з незалежними копіями, і зміни одного з них будуть втрачені після збереження файлу іншим. Використання хмарних таблиць (Google Sheets) частково вирішує цю проблему, однак не усуває ризику одночасного редагування одного рядка кількома користувачами та не забезпечує транзакційності операцій.

Проблема відсутності аналітики в реальному часі. Формування будь-якого аналітичного звіту в Excel потребує ручного написання формул або побудови зведених таблиць. При зміні структури даних (додавання нових ТЗ, розширення часового діапазону) формули потребують коригування, що є джерелом помилок.

Відсутність автоматичного розрахунку відхилення фактичної витрати пального від нормативної унеможливлює оперативне виявлення зловживань або технічних несправностей.

Проблема людського фактора та аудиторського сліду. Ручне введення даних є схильним до арифметичних помилок, описок та навмисних фальсифікацій. Відсутність журналу змін (audit log) унеможливлює відновлення первинних даних після їх редагування та встановлення відповідальної особи у разі виявлення розбіжностей. Для фінансової звітності, де кожна заправка є грошовою операцією, відсутність аудиторського сліду є критичним недоліком.

Проблема масштабованості. Зі зростанням парку до 20-30 і більше одиниць техніки обсяг даних у таблицях стає некерованим: пошук інформації про конкретний ТЗ за певний період ускладнюється, а час формування зведених звітів зростає нелінійно. Відсутність індексування даних, властивого реляційним СУБД, унеможливлює ефективне виконання складних запитів із фільтрацією за кількома параметрами одночасно.

Економічне обґрунтування автоматизації. Оцінка економічного ефекту від впровадження спеціалізованої інформаційної системи для управління автопарком включає кілька складових. По-перше, скорочення витрат робочого часу на ведення обліку та формування звітності. За консервативними оцінками, менеджер автопарку витрачає від 30 до 60 хвилин на добу на ручне ведення таблиць та формування оперативних довідок - це від 130 до 260 годин на рік, вартість яких при середній годинній ставці становить відчутну суму для підприємства. По-друге, виявлення та усунення аномального споживання пального. Автоматичний розрахунок відхилення від норми дозволяє виявляти перевитрату в режимі реального часу. При парку у 10 автомобілів і середньомісячних витратах на пальне у 50 000 грн навіть 5-відсоткова оптимізація витрат дає 2 500 грн економії щомісяця. По-третє, своєчасне планове ТО подовжує ресурс ТЗ та знижує витрати на позапланові ремонти, вартість яких, як правило, у 3-5 разів перевищує вартість планового обслуговування [28].

Сукупність зазначених проблем та економічних аргументів обґрунтовує доцільність розробки спеціалізованого web-застосунку як інструменту автоматизації управління автопарком приватного підприємства. Ключовими технічними вимогами до такої системи, що випливають із проведеного аналізу, є: забезпечення цілісності даних на рівні СУБД; підтримка рольової моделі доступу; автоматичний розрахунок аналітичних показників; ведення журналу змін; забезпечення одночасної роботи кількох операторів без конфліктів.

Таким чином, у підрозділі 1.1 здійснено комплексний аналіз предметної області управління автопарком підприємства: формалізовано п’ять ключових бізнес-процесів та побудовано їх схему у вигляді діаграми потоків; визначено чотири функціональні ролі користувачів системи та побудовано матрицю доступу; виявлено та систематизовано проблеми ручного обліку, що є обґрунтуванням необхідності автоматизації. Отримані результати є вихідними даними для формулювання функціональних вимог до системи та проєктування її архітектури у наступних підрозділах.

1.2 Огляд та порівняльний аналіз існуючих рішень

1.2.1 Аналіз комерційних систем управління автопарком

Перед формулюванням вимог до розроблюваної системи необхідно провести систематичний огляд існуючих програмних рішень у сфері управління автопарком. Такий аналіз дозволяє виявити незайняті ніші та обмеження наявних продуктів, що визначають конкурентні переваги розроблюваної системи, а також запозичити апробовані підходи до організації функціональності, спираючись на досвід галузевих рішень. Для аналізу обрано чотири системи, що охоплюють різні сегменти ринку: SaaS-платформу Fleetio, телематичну платформу Wialon (Gurtam), відкрите рішення Fleet Stack та локалізовану систему AutoFMS.

Fleetio є хмарною SaaS-платформою американського розробника Fleetio Inc., орієнтованою на комерційні автопарки середнього та великого розміру. Система

охоплює повний спектр функціональності: облік транспортних засобів із детальними профілями, планування та облік технічного обслуговування за пробігом і календарем, облік пального через інтеграцію з паливними картками та ручне введення, управління водіями, облік запасних частин та розвинену аналітику. Платформа надає публічний REST API з підтримкою webhooks, що дозволяє інтегрувати її з ERP-системами та телематичними пристроями.

Модель ліцензування є підписочною: вартість становить від 4 до 10 USD на транспортний засіб на місяць залежно від обраного плану. При парку у 20 автомобілів річні витрати на ліцензування складають від 960 до 2 400 USD, що для малого підприємства є відчутним постійним зобов'язанням. Критичним обмеженням є відсутність можливості on-premise розгортання: всі дані зберігаються виключно на серверах Fleetio Inc. у США, що для українських підприємств породжує питання відповідності вимогам законодавства про захист персональних даних. Крім того, система не підтримує локалізацію інтерфейсу українською мовою.

Wialon (Gurtam) є телематичною платформою, що спеціалізується на GPS-моніторингу та управлінні транспортом. Основним функціональним акцентом є відстеження місцезнаходження ТЗ у реальному часі, контроль маршрутів, геозони та визначення витрат пального на основі даних телематичних датчиків. Функції обліку ТЗ, водіїв та планування ТО реалізовані на значно нижчому рівні деталізації. Принциповими обмеженнями для цілей цієї роботи є: орієнтованість виключно на телематичний моніторинг, а не на комплексний операційний облік; обов'язкове встановлення телематичного обладнання на кожен ТЗ як умова повноцінного використання платформи. Компанія Gurtram зареєстрована у Литві та надає послуги на міжнародному ринку.

Fleet Stack є відносно новою платформою з відкритим вихідним кодом (ліцензія MIT), орієнтованою на розробників та підприємства, що прагнуть до самостійного розгортання. Система реалізована на стеку Node.js / React та надає базовий функціонал: реєстрацію ТЗ, облік пального, просте планування ТО та базову звітність. Наявність відкритого вихідного коду та підтримка Docker-

розгортання є суттєвими перевагами. Разом із тим система має ряд обмежень: вона перебуває на ранній стадії розвитку з неповною документацією, побудована на JavaScript без строгої типізації, не реалізує повноцінної RBAC-моделі та не містить алгоритмів автоматичного розрахунку відхилень витрат пального від нормативів.

AutoFMS є локалізованим для ринку України web-рішенням, орієнтованим на бухгалтерський та складський облік у сфері автотранспорту: облік подорожніх листів, розрахунок норм витрат ПММ відповідно до нормативних документів Міністерства інфраструктури України, формування первинних бухгалтерських документів. Система є корисним інструментом для бухгалтерії, однак практично непридатна для операційного управління: відсутній модуль управління ремонтами з деталізацією по виконавцях та запасних частинах, відсутня система сповіщень про наближення строку ТО, API для інтеграції з зовнішніми системами не передбачено.

Таким чином, кожна з розглянутих систем має виражену спеціалізацію та відповідні обмеження: Fleetio є функціонально повним, але дорогим і закритим для on-premise розгортання; Wialon орієнтований на телематику, а не на облік витрат; Fleet Stack є відкритим, але функціонально незрілим; AutoFMS охоплює бухгалтерський облік, але не операційне управління автопарком.

1.2.2 Порівняльна таблиця функціональних можливостей аналогів

Для об’єктивного зіставлення розглянутих систем між собою та з розроблюваним рішенням сформовано порівняльну таблицю за критеріями, що безпосередньо відповідають функціональним вимогам, визначеним у підрозділі 1.1.

Таблиця 1.2

Порівняльний аналіз систем управління автопарком

Критерій	**Fleetio**	**Wialon**	**Fleet Stack**	**AutoFMS**	**Розроблювана система**
Облік ТЗ (повний профіль)	Повний	Базовий	Базовий	Базовий	Повний
Облік водіїв	Повний	Базовий	Ні	Базовий	Повний
Облік пального (ручний ввід)	Так	Так	Так	Так	Так
Автоматичний розрахунок витрат та аномалій	Так	Так (датчики)	Ні	Частково	Так
Управління ремонтами та ТО	Повний	Ні	Базовий	Ні	Повний
Аналітична звітність	Розвинена	Телематика	Ні	Бухгалтерська	Операційна
REST API	Так	SDK	Ні	Ні	Так
Рольова модель доступу (RBAC)	Так	Так	Ні	Частково	Так
On-premise розгортання	Ні	Частково	Так	Так	Docker
Українська локалізація	Ні	Частково	Ні	Так	Так
Відкритий вихідний код	Ні	Ні	Так	Ні	Так
Вартість (ТЗ/місяць)	4-10 USD	2-8 USD	Безкоштовно	1-3 USD	Відсутня
Можливість кастомізації	Обмежена	Обмежена	Висока	Відсутня	Повна

Аналіз таблиці дозволяє виявити системні закономірності. По-перше, жодна з розглянутих систем не поєднує одночасно повний функціонал операційного

управління автопарком, on-premise розгортання, відкритий вихідний код та відсутність ліцензійних витрат. По-друге, автоматичний розрахунок фактичних витрат пального та виявлення аномалій реалізований лише у системах з телематичними датчиками або у повнофункціональних SaaS-платформах. По-третє, жодне з відкритих або бюджетних рішень не реалізує повноцінну RBAC-модель з фільтрацією даних на рівні API. Виявлені закономірності визначають нішу для розроблюваної системи: середні та малі підприємства, що потребують повнофункціонального операційного обліку автопарку при обмеженому бюджеті та необхідності зберігати контроль над власними даними.

1.2.3 Визначення конкурентних переваг розроблюваної системи

На підставі проведеного аналізу сформульовано конкурентні переваги розроблюваної системи, що обґрунтовують доцільність її розробки замість адаптації наявних рішень.

Відсутність ліцензійних витрат та незалежність від вендора. Розроблювана система поширюється з відкритим вихідним кодом: підприємство використовує її без будь-яких підписочних платежів. Відсутність vendor lock-in означає, що зміна цінової політики стороннього провайдера або припинення його діяльності не впливають на доступність системи - критично важливий фактор для малого бізнесу з обмеженим і передбачуваним бюджетом.

Повний контроль над даними через on-premise розгортання. Система розгортається на власній інфраструктурі підприємства за допомогою Docker та Docker Compose. Дані про транспортні засоби, водіїв та фінансові операції є конфіденційною корпоративною інформацією, зберігання якої на серверах третіх осіб в іноземних юрисдикціях може суперечити вимогам Закону України «Про захист персональних даних» [1]. On-premise розгортання усуває цей ризик та дозволяє підприємству самостійно визначати політику резервного копіювання та доступу.

Адаптованість під специфічні вимоги підприємства. Комерційні системи реалізують узагальнену модель предметної області. Специфічні вимоги -

особливості розрахунку нормативів витрат пального з сезонними поправочними коефіцієнтами, нестандартні форми звітних документів, інтеграція з внутрішньою обліковою системою - в комерційних рішеннях або не підтримуються, або реалізуються за додаткову плату. Повний доступ до вихідного коду дозволяє виконати будь-яку адаптацію безпосередньо, без обмежень SaaS-провайдера.

Технологічна сучасність та повноцінна RBAC. Вибір стеку Python / Django / Django REST Framework та TypeScript / React забезпечує строгу типізацію, захист від SQL-ін'єкцій на рівні ORM та відтворюваність розгортання через Docker - переваги, яких позбавлені розглянуті відкриті альтернативи. Чотирирівнева рольова модель доступу з фільтрацією даних на рівні серверної бізнес-логіки гарантує, що жоден користувач не може отримати доступ до даних поза межами своїх повноважень навіть при прямому зверненні до API [5].

Таким чином, у підрозділі 1.2 проведено огляд чотирьох існуючих рішень у сфері управління автопарком, побудовано порівняльну таблицю за тринадцятьма критеріями та сформульовано чотири конкурентні переваги розроблюваної системи, що в сукупності обґрунтовують доцільність її самостійної розробки.

1.3 Обґрунтування вибору технологій розробки

1.3.1 Вибір мови та фреймворку серверної частини: Django vs FastAPI

Вибір серверного фреймворку є одним із найбільш визначальних архітектурних рішень, оскільки він обумовлює підходи до роботи з базою даних, організації бізнес-логіки, автентифікації та документування API. У екосистемі Python два фреймворки є найбільш поширеними для розробки web-застосунків: Django з розширенням Django REST Framework (DRF) та FastAPI. Обидва активно розвиваються, однак орієнтовані на суттєво відмінні класи задач.

Django є повнофункціональним фреймворком, що дотримується принципу «batteries included»: він надає вбудовані засоби для роботи з базою даних через власну ORM, генерацію та застосування міграцій схеми, адміністративну панель,

систему автентифікації та авторизації з підтримкою груп і дозволів [17]. Розширення DRF додає серіалізатори, класи-представлення (`ModelViewSet`, `GenericAPIView`), фільтрацію, пагінацію та API-інтерфейс для налагодження. Фреймворк розвивається з 2005 року, має детальну документацію та зрілий цикл підтримки з LTS-версіями.

FastAPI є сучасним асинхронним мікрофреймворком на основі ASGI та Pydantic [5]. Його ключові переваги: висока продуктивність у синтетичних бенчмарках, автоматична генерація OpenAPI-документації на основі анотацій типів та лаконічний синтаксис. За даними TechEmpower Framework Benchmarks (round 22), FastAPI демонструє пропускну здатність порядку 50 000-100 000 req/s проти 5 000-15 000 req/s для Django у сценаріях чистої JSON-серіалізації.

Однак для задачі управління автопарком абсолютна пропускна здатність не є вирішальним параметром: система обслуговує обмежену кількість одночасних операторів (5-20 осіб), а переважна більшість запитів є I/O-bound з домінуванням часу очікування відповіді від СУБД. Натомість визначальною є різниця у вбудованій функціональності. Django ORM надає декларативний інтерфейс для складних агрегаційних запитів, що є необхідним для модуля звітності системи (рис. 1.2).

```python
from django.db.models import Sum, Avg, F, ExpressionWrapper, FloatField
from django.db.models.functions import TruncMonth

report = FuelingRecord.objects.filter(
    vehicle=vehicle,
    date__range=(date_from, date_to)
).annotate(
    month=TruncMonth('date')
).values('month').annotate(
    total_liters=Sum('liters'),
    total_cost=Sum(F('liters') * F('price_per_liter')),
    avg_consumption=Avg(
        ExpressionWrapper(
            F('liters') / F('odometer_delta') * 100,
            output_field=FloatField()
        )
    )
).order_by('month')
```

Рис. 1.2 - Модуль звітності системи

Реалізація аналогічної логіки у FastAPI потребувала б або написання «сирого» SQL, або підключення SQLAlchemy як окремої залежності, що ускладнює

архітектуру та збільшує операційні витрати на підтримку. Вбудована адміністративна панель Django дозволяє без написання додаткового коду керувати довідниками системи (типи ТЗ, типи ремонтних робіт, нормативи витрат). Вбудована система міграцій гарантує відтворювану еволюцію схеми бази даних на будь-якому середовищі розгортання - FastAPI потребує для цього окремого налаштування Alembic.

Таблиця 1.3

Порівняння Django + DRF та FastAPI

Критерій	**Django + DRF**	**FastAPI**
Вбудована ORM	Django ORM	потрібна SQLAlchemy
Адміністративна панель	Вбудована	
Система міграцій БД	Вбудована	потрібна Alembic
Автентифікація та RBAC	Вбудована	сторонні бібліотеки
Продуктивність (req/s)	5 000-15 000	50 000-100 000
Асинхронна підтримка	Часткова (Django 4.1+)	Повна (ASGI)
Автодокументація OpenAPI	Через drf-spectacular	Вбудована
Зрілість екосистеми	Висока (з 2005)	Середня (з 2018)

Django з DRF обрано як серверний фреймворк, оскільки вбудована ORM, система міграцій, адміністративна панель та механізм автентифікації безпосередньо відповідають вимогам системи управління автопарком і не потребують додаткових залежностей. Перевага FastAPI у продуктивності є нерелевантною для заданого навантаження, тоді як відсутність вбудованої функціональності суттєво ускладнила б реалізацію.

1.3.2 Вибір фреймворку клієнтської частини: React vs Angular

Вибір клієнтського фреймворку визначає архітектурний підхід до організації інтерфейсу, управління станом та типової безпеки коду. Для реалізації клієнтської

частини системи розглянуто React (Meta) та Angular (Google) як два провідних рішення з найбільшою часткою використання серед корпоративних застосунків.

Angular є повноцінним MVC-фреймворком із власною системою модулів, сервісів, директив та механізмом впровадження залежностей (Dependency Injection) [15]. TypeScript є єдиною офіційно підтримуваною мовою Angular нативно з версії 2.0. Angular CLI забезпечує генерацію компонентів, сервісів та модулів за шаблонами, що гарантує одноманітність структури кодової бази. Реактивне програмування через RxJS є архітектурною основою для роботи з асинхронними потоками даних, проте потребує глибокого розуміння концепцій Observable, Subject та операторів трансформації - що суттєво підвищує порогові вимоги до кваліфікації розробника.

Розмір вхідного бандлу Angular-застосунку після збірки є значно більшим, ніж у аналогічних React-застосунків: мінімальний Angular-застосунок займає близько 140-200 КБ у gzip, тоді як аналогічний React-застосунок - близько 40-50 КБ. Для системи, що розгортається на власній інфраструктурі підприємства, ця різниця впливає на час початкового завантаження.

React є декларативною бібліотекою для побудови компонентних інтерфейсів, що свідомо обмежується рівнем представлення (View) і не нав’язує конкретного підходу до маршрутизації чи управління станом [10]. TypeScript підтримується через офіційні типізації (`@types/react`) та шаблони проєктів на базі Vite. Хуки (`useState`, `useEffect`, `useContext`, `useReducer`) є основним інструментом управління станом, а React Query (TanStack Query) забезпечує кешування відповідей API та автоматичну ревалідацію без необхідності впроваджувати Redux.

Строга типізація інтерфейсів на основі TypeScript гарантує відповідність структури даних між API та компонентами на етапі компіляції. За даними дослідження Gao et al., статична типізація дозволяє виявити до 15 % помилок, що у нетипізованому JavaScript проявляються лише у runtime [6] (рис. 1.3).

```typescript
interface Vehicle {
  id: number;
  plate_number: string;
  brand: string;
  model: string;
  status: 'active' | 'maintenance' | 'decommissioned';
  current_odometer: number;
  fuel_norm_per_100km: number;
}

export const useVehicles = () => {
  return useQuery<Vehicle[]>({
    queryKey: ['vehicles'],
    queryFn: () =>
      axios.get<Vehicle[]>('/api/vehicles/').then(res => res.data),
    staleTime: 5 * 60 * 1000,
  });
};

```

Рис. 1.3 - Статична типізація

Літеральний union-тип `'active' | 'maintenance' | 'decommissioned'` для поля `status` гарантує, що жоден компонент не зможе передати неприпустиме значення статусу - компілятор TypeScript відхилить такий код на етапі збірки, що є прямим відображенням скінченного автомату статусів ТЗ, визначеного у підрозділі 1.1.1.

Таблиця 1.4

Порівняння React та Angular

Критерій	**React + TypeScript**	**Angular**
Архітектурний підхід	Бібліотека (гнучкий)	Повний фреймворк (жорсткий)
TypeScript	Через типізації	Нативна
Управління станом	Context + React Query	RxJS + NgRx
Розмір бандлу (gzip)	~40-50 КБ	~140-200 КБ
Крива навчання	Низька-середня	Висока (RxJS)
Придатність для середнього застосунку	✓ Оптимальна	Надлишкова складність

React з TypeScript обрано як клієнтський фреймворк, оскільки для системи управління автопарком середнього масштабу архітектурна жорсткість Angular є

надлишковою, тоді як менший розмір бандлу, нижча крива навчання та гнучкість у виборі допоміжних бібліотек прискорюють розробку без втрати типової безпеки.

1.3.3 Обґрунтування вибору СУБД та засобів контейнеризації

Вибір PostgreSQL. Модель даних системи управління автопарком характеризується високим рівнем зв’язаності сутностей: транспортні засоби пов’язані з водіями через журнал закріплень, заправки - з ТЗ і водіями, ремонти - з ТЗ, типами робіт та виконавцями. Така модель є природно реляційною і потребує СУБД з повноцінною підтримкою зовнішніх ключів, каскадних операцій та транзакцій.

Альтернативою могла б бути MongoDB як документо-орієнтована СУБД. Однак, враховуючи зв’язаність сутностей (ТЗ ← Заправки ← Водії), документо-орієнтована модель призвела б до надмірного дублювання: кожен документ заправки вимагав би вбудовування атрибутів ТЗ та водія, а будь-яка зміна нормативу витрат пального потребувала б оновлення всіх пов’язаних документів без гарантій транзакційної узгодженості. Для фінансової звітності, де кожен запис є грошовою операцією, така модель є неприйнятною.

PostgreSQL задовольняє усім ключовим вимогам. Повна підтримка ACID-транзакцій гарантує атомарність складених операцій: наприклад, одночасне створення запису заправки та оновлення поточного пробігу ТЗ виконуються як єдина операція - або обидві зміни зберігаються, або жодна. Підтримка складних агрегаційних запитів із груповими функціями є необхідною для модуля звітності (рис. 1.4) [24].

```sql
SELECT
    v.plate_number,
    v.brand || ' ' || v.model AS vehicle_name,
    v.fuel_norm_per_100km,
    ROUND(
        SUM(f.liters)::numeric /
        NULLIF(MAX(f.odometer) - MIN(f.odometer), 0) * 100, 2
    ) AS actual_consumption,
    ROUND(
        (SUM(f.liters)::numeric /
         NULLIF(MAX(f.odometer) - MIN(f.odometer), 0) * 100
         - v.fuel_norm_per_100km) /
        v.fuel_norm_per_100km * 100, 1
    ) AS deviation_pct
FROM fueling_records f
JOIN vehicles v ON f.vehicle_id = v.id
WHERE f.date BETWEEN '2024-01-01' AND '2024-03-31'
GROUP BY v.id, v.plate_number, v.brand, v.model, v.fuel_norm_per_100km
HAVING COUNT(f.id) > 1
ORDER BY deviation_pct DESC NULLS LAST;
```

Рис. 1.4 - Модуль звітності

Ефективне виконання цього запиту забезпечується складеним індексом `(vehicle_id, date)` на таблиці `fueling_records`, що є стандартною можливістю PostgreSQL без додаткових розширень.

Роль Docker та Docker Compose. Контейнеризація вирішує проблему відтворюваності середовища: відмінності між середовищами розробки та виробничого розгортання є поширеним джерелом помилок, що не відтворюються локально [4]. Docker ізолює всі залежності застосунку (версія Python, Node.js, бінарні залежності PostgreSQL) у межах контейнера, гарантуючи ідентичність поведінки незалежно від хост-системи.

Docker Compose дозволяє декларативно описати багатоконтейнерну архітектуру (backend, frontend, база даних, nginx) та запустити все середовище однією командою.

Конфігурація може реалізувати ряд принципових практик:

- healthcheck для гарантування готовності PostgreSQL перед запуском backend-сервісу;
- зберігання секретів у змінних середовища поза репозиторієм;
- named volume для збереження даних між перезапусками контейнерів;
- nginx як зворотний проксі для маршрутизації запитів між сервісами.

Таким чином, у підрозділі 1.3 обґрунтовано вибір усіх компонентів технологічного стеку. Django з DRF обрано завдяки вбудованій ORM, системі міграцій та адміністративній панелі; React з TypeScript - завдяки гнучкості, меншому бандлу та типовій безпеці; PostgreSQL - завдяки підтримці ACID-транзакцій та складних агрегаційних запитів; Docker - для забезпечення відтворюваності середовища розгортання. Сформований технологічний стек є достатнім для реалізації всіх функціональних вимог, визначених у підрозділі 1.1, та є основою для проєктування архітектури системи у розділі 2.

1.4 Постановка задач на розробку

1.4.1 Функціональні вимоги до системи

Функціональні вимоги (Functional Requirements, FR) визначають поведінку системи у відповідь на конкретні дії користувача та є основою для проєктування архітектури і планування реалізації. Їх формулювання спирається на результати аналізу предметної області (підрозділ 1.1) та враховує обмеження, виявлені при огляді аналогів (підрозділ 1.2). Невід'ємним кроком після формулювання вимог є їх пріоритизація, що дозволяє визначити порядок реалізації та обґрунтовано окреслити межі першої версії системи.

Вибір методу пріоритизації вимог. У практиці інженерії програмного забезпечення існує кілька формалізованих підходів до пріоритизації вимог. Для цієї роботи розглянуто два кількісні методи: зважена пріоритизація (Weighted Prioritization) та метод складеного ключа (Composite Key).

Метод **Weighted Prioritization** передбачає оцінку кожної вимоги за кількома незалежними критеріями (цінність для бізнесу, вартість реалізації, технічний ризик, стратегічна важливість) із призначенням числової ваги кожному критерію та обчисленням підсумкового пріоритету як зваженої суми оцінок [1]. Метод забезпечує точне числове ранжування всіх вимог і дозволяє враховувати суперечливі чинники - наприклад, вимога може мати високу бізнес-цінність, але

водночас значний технічний ризик, що знижує її реальний пріоритет. Основний недолік полягає у суб'єктивності призначення ваг критеріїв: різні стейкхолдери можуть по-різному оцінювати відносну важливість цінності та ризику, що при великій кількості вимог призводить до значної трудомісткості узгодження та потенційної непослідовності оцінювання [2].

Метод **Composite Key** базується на класифікації вимог за двома ортогональними вимірами - *цінністю* (Value) та *складністю реалізації* (Effort) - і розподіляє вимоги по чотирьох квадрантах матриці: «реалізувати першочергово» (висока цінність, низька складність), «заплановано» (висока цінність, висока складність), «розглянути» (низька цінність, низька складність) та «виключити з поточної версії» (низька цінність, висока складність) [14]. Метод є наочним, менш трудомістким і чітко відповідає на практичне питання: що потрапляє до першої версії, а що відкладається.

Для системи управління автопарком у рамках бакалаврської роботи з фіксованим обсягом і одним розробником метод Composite Key є більш доцільним. Кількість вимог є обмеженою, основне завдання пріоритизації - не точне числове ранжування всередині групи, а чіткий розподіл вимог на групи реалізації. Weighted Prioritization доцільніший для великих проєктів із командою та бюджетними обмеженнями, де точне ранжування визначає розподіл ресурсів між розробниками. Порівняння методів наведено в таблиці 1.5.

Таблиця 1.5

Порівняння методів пріоритизації вимог

	Effort: Низька	Effort: Середня	Effort: Висока
Value: Висока	Першочергово	Першочергово	Заплановано
Value: Середня	Першочергово	Заплановано	Розглянути
Value: Низька	Розглянути	Розглянути	Виключити

Перелік функціональних вимог із пріоритизацією:

FR-01. CRUD транспортних засобів. Реєстрація нового ТЗ із повним набором атрибутів (державний номер, марка, модель, рік випуску, VIN, тип палива, нормативна витрата), редагування атрибутів, зміна статусу (`активний` / `на ремонті` / `списаний`) та перегляд деталізованого профілю. Value: B, Effort: H → **Першочергово.**

FR-02. CRUD водіїв. Реєстрація водія з персональними даними та реквізитами водійського посвідчення (категорія, номер, дата видачі, термін дії). Відстеження терміну дії посвідчення є окремою бізнес-вимогою: його закінчення без поновлення є адміністративним порушенням для підприємства. Value: B, Effort: H → **Першочергово.**

FR-03. Закріплення ТЗ за водієм з історією. Реєстрація поточного закріплення ТЗ за водієм та збереження повної хронологічної історії закріплень (дата початку, дата завершення). Без цієї вимоги неможлива коректна побудова звітів витрат у розрізі водіїв за минулі periodи. Value: B, Effort: C → **Першочергово.**

FR-04. Реєстрація заправок. Фіксація заправок із зазначенням ТЗ, водія, дати та часу, кількості залитого пального (л), ціни за літр, назви АЗС та поточного показання одометра. Value: B, Effort: H → **Першочергово.**

FR-05. Автоматичний розрахунок витрат пального та виявлення аномалій. Розрахунок фактичної витрати пального (л/100 км) та відхилення від нормативного показника для кожного інтервалу між заправками. При перевищенні встановленого порогу (за замовчуванням - 10 %) запис позначається як аномальний. Value: B, Effort: C → **Першочергово.**

FR-06. Управління ремонтами та ТО. Реєстрація заявок на ремонт та ТО із зазначенням типу робіт, переліку виконаних операцій, використаних запасних частин із вартістю, виконавця, дат відкриття та закриття. Відкриття заявки автоматично змінює статус ТЗ на «`на ремонті`», закриття - повертає «`активний`» та оновлює дату наступного планового ТО. Value: B, Effort: C → **Першочергово.**

FR-07. Рольова модель доступу (RBAC). Чотири ролі: адміністратор, менеджер автопарку, диспетчер, водій. Доступ до ресурсів API перевіряється на рівні кожного запиту; дані фільтруються за роллю на рівні серверної бізнес-логіки відповідно до матриці доступу (таблиця 1.1). Value: B, Effort: C → **Першочергово.**

FR-08. JWT-автентифікація. Access- та refresh-токени з термінами дії 15 хвилин та 7 днів відповідно. Механізм оновлення токена є прозорим для користувача. Value: B, Effort: H → **Першочергово.**

FR-09. Аналітична звітність. Формування звітів з фільтрацією за часовим діапазоном: витрати пального по ТЗ; витрати пального по водіях; витрати на ремонти по ТЗ; пробіг ТЗ за період; зведений звіт витрат на утримання ТЗ. Value: B, Effort: B → **Заплановано.**

FR-10. Дашборд зі статистикою. Агрегована статистика на головній сторінці: кількість активних ТЗ та ТЗ на ремонті, сума витрат на пальне та ремонти за поточний місяць, перелік ТЗ з аномальними витратами пального. Value: C, Effort: C → **Заплановано.**

FR-11. Нагадування про планове ТО. Відображення попередження для ТЗ, у яких залишок ресурсу до наступного ТО становить менше 500 км або 14 днів - залежно від того, яка умова настає раніше. Value: C, Effort: H → **Першочергово.**

FR-12. Експорт звітів (PDF/XLSX). Формування звітів у форматі, придатному для передачі до бухгалтерії або керівництву підприємства. Value: C, Effort: B → **Розглянути.**

FR-13. Журнал аудиту. Фіксація всіх операцій зміни даних із зазначенням автора, часу та попереднього значення. Value: C, Effort: B → **Розглянути.**

FR-14. Інтеграція з GPS-телематикою. Автоматичне отримання даних одометра та місцезнаходження ТЗ через зовнішні API телематичних пристроїв. Value: H, Effort: B → **Виключити.**

FR-15. Нативний мобільний застосунок. Окремий застосунок для iOS / Android для водіїв та диспетчерів. Адаптивний web-інтерфейс є достатнім для польового використання. Value: H, Effort: B → **Виключити.**

Зведену таблицю пріоритизації вимог наведено в таблиці 1.7.

Таблиця 1.6

Пріоритизація функціональних вимог методом Composite Key

ID	Вимога	Value	Effort	Пріоритет
FR-01	CRUD транспортних засобів	B	H	Першочергово
FR-02	CRUD водіїв	B	H	Першочергово
FR-03	Закріплення ТЗ за водієм з історією	B	C	Першочергово
FR-04	Реєстрація заправок	B	H	Першочергово
FR-05	Розрахунок витрат пального та аномалій	B	C	Першочергово
FR-06	Управління ремонтами та ТО	B	C	Першочергово
FR-07	Рольова модель доступу (RBAC)	B	C	Першочергово
FR-08	JWT-автентифікація	B	H	Першочергово
FR-11	Нагадування про планове ТО	C	H	Першочергово
FR-09	Аналітична звітність	B	B	Заплановано
FR-10	Дашборд зі статистикою	C	C	Заплановано
FR-12	Експорт звітів (PDF/XLSX)	C	B	Розглянути
FR-13	Журнал аудиту	C	B	Розглянути
FR-14	Інтеграція з GPS-телематикою	H	B	Виключити
FR-15	Нативний мобільний застосунок	H	B	Виключити

Результат пріоритизації визначає, що перша версія системи включає 9 вимог групи «Першочергово» (FR-01-FR-08, FR-11) та 2 вимоги групи «Заплановано» (FR-09, FR-10). Вимоги FR-12-FR-13 можуть бути реалізовані у наступних ітераціях, FR-14-FR-15 виходять за межі поточної роботи.

1.4.2 Нефункціональні вимоги

Нефункціональні вимоги (Non-Functional Requirements, NFR) визначають якісні характеристики системи - продуктивність, надійність, безпеку та зручність розгортання. На відміну від функціональних, вони не описують окрему функцію, а задають обмеження, яким повинна відповідати реалізація системи в цілому [14]. Нефункціональні вимоги формулюються як вимірювані характеристики з

конкретними порогами, що дозволяє об'єктивно перевірити їх виконання на етапі тестування.

NFR-01. Продуктивність API. Час відгуку серверної частини для 95-го перцентиля запитів не повинен перевищувати 300 мс при зверненні до бази даних через не більше трьох JOIN-операцій. Для звітних запитів із складною агрегацією допускається час відгуку до 2 000 мс із відображенням індикатора завантаження на стороні клієнта. Поріг 300 мс відповідає межі, нижче якої затримка сприймається користувачем як миттєва реакція системи [11]. Виконання вимоги забезпечується правильним індексуванням таблиць PostgreSQL та використанням `select_related()` і `prefetch_related()` у Django ORM для уникнення проблеми N+1 запитів.

NFR-02. Масштабованість. Система повинна забезпечувати коректну роботу при одночасному підключенні не менше 50 користувачів без деградації продуктивності понад установлений поріг. Для парку до 100 транспортних засобів з даними за 3 роки це вимагає ефективної роботи з таблицями розміром до ~200 000 записів у `fueling_records`. Виконання забезпечується налаштуванням пулу з'єднань PostgreSQL через параметр `CONN_MAX_AGE` у Django.

NFR-03. Доступність. Коефіцієнт доступності системи в межах робочого часу підприємства (8:00-20:00, понеділок-субота) повинен бути не нижчим за 99 %, що відповідає максимально допустимому простою 7,2 години на місяць. Оскільки система розгортається на власній інфраструктурі без зовнішнього SLA, виконання забезпечується політикою автоматичного перезапуску контейнерів (`restart: unless-stopped`) та регулярним резервним копіюванням PostgreSQL через `pg_dump`.

NFR-04. Безпека. Система повинна відповідати вимогам OWASP Top 10 [7] у частині, що застосовна до REST API: захист від SQL-ін'єкцій через параметризовані запити Django ORM; захист від XSS через санітизацію вхідних даних на рівні DRF-серіалізаторів; дворівнева авторизація (перевірка ролі на рівні класу представлення та фільтрація queryset у методі `get_queryset()`); зберігання паролів у вигляді хешу PBKDF2-SHA256.

NFR-05. Адаптивність інтерфейсу. Клієнтський застосунок повинен коректно відображатися та бути повністю функціональним на пристроях із

шириною екрану від 375 пікселів (смартфон) до 2560 пікселів (широкоформатний монітор). Ця вимога є критичною для ролей диспетчера та водія, що можуть працювати з системою з мобільних пристроїв у польових умовах. Виконання забезпечується використанням CSS Flexbox та Grid із медіа-запитами.

NFR-06. Розгортання. Запуск повного середовища системи на новому сервері повинен виконуватися не більше ніж за 3 команди: клонування репозиторію, налаштування `.env`-файлу з секретами та виконання `docker compose up -d`. Жодне додаткове встановлення Python, Node.js або PostgreSQL на хост-машині не є необхідним.

NFR-07. Тестове покриття. Покриття коду серверної частини автоматизованими тестами повинне становити не менше 70 % для модулів бізнес-логіки: розрахунок витрат пального, зміна статусів ТЗ, перевірка прав доступу. Тестування реалізується засобами pytest у поєднанні з Django TestCase та DRF APIClient.

Зведену специфікацію нефункціональних вимог наведено в таблиці 1.8.

Таблиця 1.7

Специфікація нефункціональних вимог

ID	Характеристика	Метрика	Порогове значення	Засіб досягнення
NFR-01	Продуктивність API	Час відгуку p95	≤ 300 мс	Індекси PostgreSQL, ORM-оптимізація
NFR-02	Масштабованість	Одночасні користувачі	≥ 50	Пул з'єднань PostgreSQL
NFR-03	Доступність	Uptime у робочий час	≥ 99 %	Docker restart policy, pg_dump
NFR-04	Безпека	Відповідність OWASP Top 10	Повна (релевантні пункти)	DRF, PBKDF2, двоетапна авторизація
NFR-05	Адаптивність	Мінімальна ширина екрану	375 px	CSS Flexbox/Grid
NFR-06	Розгортання	Кількість команд запуску	≤ 3	Docker Compose
NFR-07	Тестове покриття	Покриття бізнес-логіки	≥ 70 %	pytest + DRF APIClient

Таким чином, у підрозділі 1.4 сформульовано 15 функціональних вимог із пріоритизацією методом Composite Key та 7 нефункціональних вимог із вимірюваними порогами. Вибір методу Composite Key над Weighted Prioritization обґрунтовано малим обсягом вимог та практичною орієнтацією на розподіл по групах реалізації, а не на точне числове ранжування. Сукупність вимог є повною специфікацією для проєктування архітектури системи у розділі 2 та визначає критерії верифікації на етапі тестування у розділі 3.

2. ПРОЄКТУВАННЯ АРХІТЕКТУРИ ТА МОДЕЛІ ДАНИХ СИСТЕМИ

2.1 Архітектура програмного забезпечення

2.1.1 Загальна архітектура системи: клієнт-серверна модель та REST API

Проєктування архітектури системи управління автопарком передбачає вибір структурної моделі, що забезпечує чіткий розподіл відповідальності між компонентами, незалежність їх розробки та масштабованість при зростанні навантаження. Визначальним архітектурним рішенням є спосіб організації взаємодії між клієнтською частиною, серверною логікою та рівнем зберігання даних.

Обґрунтування трирівневої архітектури. Система побудована за трирівневою клієнт-серверною моделлю, що включає рівень представлення (SPA на React/TypeScript), рівень бізнес-логіки (REST API на Django/DRF) та рівень даних (PostgreSQL). Альтернативою могла б бути монолітна архітектура з серверним рендерингом (SSR), де Django генерує HTML-сторінки через шаблонізатор. Проте для системи з чотирма ролями та суттєво різними інтерфейсами для кожної ролі SSR-підхід призводить до тісного зв’язування між логікою відображення та бізнес-логікою, що ускладнює незалежну еволюцію обох рівнів. Трирівнева архітектура з REST API як контрактом між рівнями усуває цей недолік: серверна частина залишається агностичною щодо способу відображення даних, а клієнтська - щодо деталей зберігання [16].

Обґрунтування REST над альтернативами. Для міжрівневої взаємодії обрано архітектурний стиль REST (Representational State Transfer). Альтернативою міг бути GraphQL, що дозволяє клієнту точно специфікувати структуру необхідних даних у запиті та уникнути over-fetching. Однак для системи управління автопарком із фіксованим набором екранів та передбачуваними запитами (список ТЗ, деталі ТЗ, журнал заправок) гнучкість GraphQL є надлишковою, а додаткова

складність схеми та резолверів - невиправданою. REST з ресурсно-орієнтованими URL та стандартними HTTP-методами забезпечує достатню гнучкість при значно нижчому порозі входження для підтримки та розширення [20]. Стан між запитами не зберігається на сервері (stateless): кожен запит містить повний контекст автентифікації у вигляді JWT access-токена в заголовку `Authorization`, що спрощує горизонтальне масштабування серверної частини у майбутньому.

Розгортання та ізоляція компонентів. Кожен логічний рівень системи ізольований у окремому Docker-контейнері: `frontend` (nginx + статичні файли React-застосунку), `backend` (Gunicorn + Django), `db` (PostgreSQL 16), `nginx` (зворотний проксі). Така ізоляція забезпечує незалежність оновлення компонентів та відтворюваність середовища розгортання [4]. Діаграму розгортання системи наведено на рисунку 2.1.

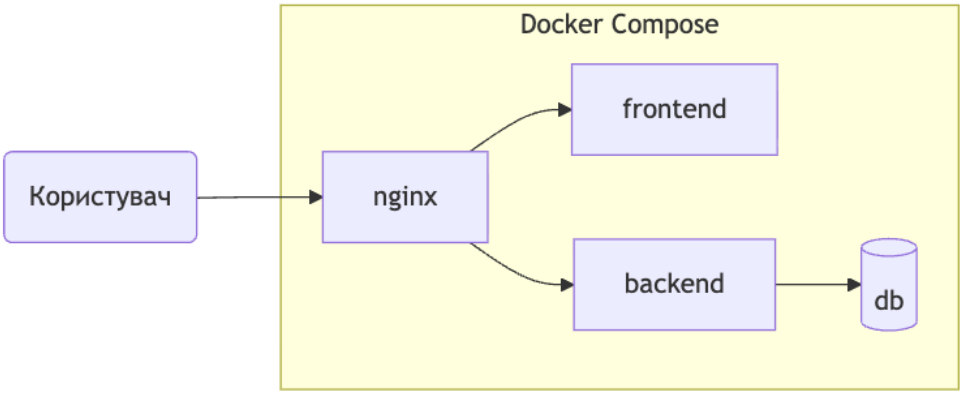


Рис. 2.1 - Діаграма розгортання системи

Nginx виконує роль єдиної точки входу: статичні файли SPA роздаються безпосередньо без звернення до backend-контейнера, а запити до `/api/*` проксіюються до Gunicorn. Такий розподіл усуває необхідність обробки статики самим Django (що є неефективним у виробничому середовищі) та централізує конфігурацію SSL-сертифікатів в одному місці.

Таким чином, трирівнева архітектура з REST API як контрактом між рівнями та Docker-ізоляцією компонентів забезпечує необхідний баланс між складністю

реалізації та гнучкістю розвитку системи, що є основою для проєктування окремих компонентів у наступних підрозділах.

2.1.2 Застосовані шаблони проєктування

Вибір архітектурних шаблонів на рівні внутрішньої організації серверної частини визначає підтримуваність коду, можливість його тестування та стійкість до змін вимог. Застосування апробованих шаблонів дозволяє уникнути «розростання» логіки у views та моделях - типового антипатерну Django-застосунків, де бізнес-логіка розподіляється між шарами без чіткої структури [23].

Service Layer. Основним структурним шаблоном серверної частини є Service Layer - виділений шар сервісів, що інкапсулює бізнес-логіку та ізолює її від рівня HTTP-обробки (DRF ViewSets) та рівня доступу до даних (Django ORM). Для системи управління автопарком це є принциповим рішенням: логіка розрахунку відхилення витрат пального від норми, автоматична зміна статусу ТЗ при відкритті ремонтної заявки, перерахунок дати наступного ТО - всі ці операції мають бути ізольовані від деталей HTTP-протоколу та перевіряться юніт-тестами без запуску web-сервера. Приклад сервісу обліку заправок наведено на рисунку 2.2.

```python
class FuelingService:
    ANOMALY_THRESHOLD_PCT = Decimal('10.0')

    @staticmethod
    @transaction.atomic
    def register_fueling(vehicle: Vehicle, driver, data: dict) -> FuelingRecord:
        previous = FuelingRecord.objects.filter(
            vehicle=vehicle
        ).order_by('-odometer').first()

        record = FuelingRecord.objects.create(
            vehicle=vehicle,
            driver=driver,
            **data
        )

        if previous:
            distance = record.odometer - previous.odometer
            if distance > 0:
                actual = (record.liters / Decimal(distance)) * 100
                norm = vehicle.fuel_norm_per_100km
                deviation = ((actual - norm) / norm) * 100
                record.actual_consumption = actual
                record.deviation_pct = deviation
                record.is_anomalous = deviation > FuelingService.ANOMALY_THRESHOLD_PCT
                record.save(update_fields=[
                    'actual_consumption', 'deviation_pct', 'is_anomalous'
                ])

        vehicle.current_odometer = record.odometer
        vehicle.save(update_fields=['current_odometer'])

        return record
```

Рис. 2.2 - Сервіс обліку заправок

Декоратор `@transaction.atomic` гарантує, що створення запису заправки та оновлення одометра ТЗ виконуються як єдина транзакція: у разі будь-якої помилки обидві операції відкатуються, що забезпечує узгодженість даних.

Repository Pattern через Django ORM. Django ORM реалізує шаблон Active Record, де модель поєднує дані та логіку доступу до бази. Для складних запитів (звітність, агрегація) замість розподілення queryset-логіки по сервісах або views виділяються менеджери моделей (`Manager`), що є Django-специфічною реалізацією Repository Pattern (рис.2.3).

```python
from django.db.models import Manager, Sum, F, ExpressionWrapper, DecimalField
from django.db.models.functions import TruncMonth

class FuelingRecordManager(Manager):
    def monthly_summary(self, vehicle, date_from, date_to):
        return self.filter(
            vehicle=vehicle,
            date__range=(date_from, date_to)
        ).annotate(
            month=TruncMonth('date')
        ).values('month').annotate(
            total_liters=Sum('liters'),
            total_cost=Sum(
                ExpressionWrapper(
                    F('liters') * F('price_per_liter'),
                    output_field=DecimalField()
                )
            )
        ).order_by('month')
```

Рис. 2.3 - Реалізація Repository Pattern

DTO через DRF Serializers. Передача даних між рівнями системи здійснюється через DRF-серіалізатори, що виконують роль об'єктів передачі даних (DTO): вони відповідають за валідацію вхідних даних, серіалізацію моделей у JSON та десеріалізацію JSON у Python-об'єкти. Розділення серіалізаторів для читання та запису (`VehicleReadSerializer` / `VehicleWriteSerializer`) запобігає ситуації, коли поля, обчислювані на сервері (наприклад, `is_anomalous` або `days_until_maintenance`), стають доступними для запису клієнтом.

Middleware для крос-функціональних аспектів. Обробка помилок та логування запитів реалізована на рівні Django Middleware, що дозволяє централізовано перехоплювати необроблені виключення та повертати стандартизовані JSON-відповіді з кодами помилок замість HTML-сторінок Django за замовчуванням.

Компонентну структуру серверної частини відображає діаграма зображена на рисунку 2.4.

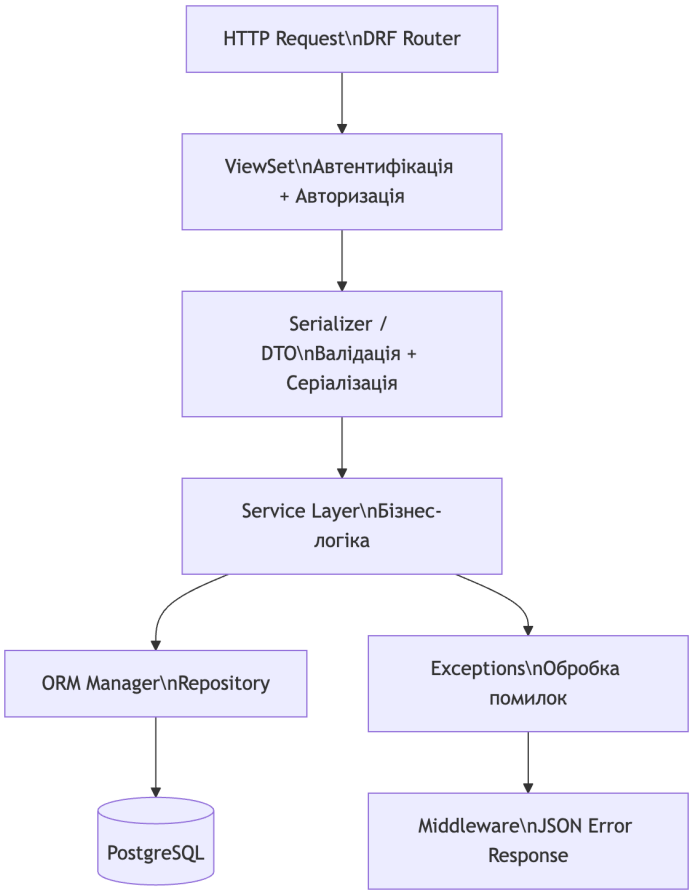


Рис. 2.4 - Структура серверної частини

Таким чином, застосування Service Layer, ORM Manager як Repository та DRF Serializers як DTO забезпечує чіткий розподіл відповідальності між шарами серверної частини, спрощує юніт-тестування бізнес-логіки та унеможливлює неконтрольоване проникнення логіки між рівнями.

2.1.3 Специфікація REST API ендпоінтів

Специфікація API є контрактом між серверною та клієнтською частинами системи: її визначення до початку реалізації дозволяє вести розробку обох частин паралельно та запобігає неузгодженостям інтерфейсу [20]. API організовано за ресурсно-орієнтованим принципом: кожен URL ідентифікує ресурс, а HTTP-метод визначає тип операції над ним. Всі ендпоінти згруповано за ресурсами та мають спільний префікс `/api/v1/`, де `v1` є версією API.

Версіонування API. Включення версії до URL є найбільш поширеним підходом до версіонування REST API та обраним для цієї системи завдяки наочності та простоті налаштування на рівні nginx і DRF-маршрутизатора. Альтернативний підхід - версіонування через заголовок `Accept: application/vnd.fleet.v1+json` - є більш «чистим» з точки зору REST-архітектури, однак ускладнює налагодження та інтеграційне тестування, оскільки версія не є частиною URL і не відображається в логах проксі-сервера. Повну специфікацію ендпоінтів системи наведено в таблиці, додаток А.

Приклад типової JSON-відповіді ендпоінту `GET /api/v1/vehicles/{id}/` наведено на рисунку 2.5.


```
{
  "id": 12,
  "plate_number": "АА1234ВВ",
  "brand": "Mercedes-Benz",
  "model": "Sprinter 316",
  "year": 2021,
  "vin": "WDB9066351S123456",
  "fuel_type": "diesel",
  "fuel_norm_summer": "10.50",
  "fuel_norm_winter": "12.00",
  "current_odometer": 87450,
  "status": "active",
  "assigned_driver": {
    "id": 5,
    "full_name": "Коваленко Олексій Іванович",
    "license_number": "ААА123456"
  },
  "next_maintenance": {
    "due_odometer": 90000,
    "due_date": "2025-06-01",
    "km_remaining": 2550,
    "days_remaining": 22,
    "is_overdue": false
  },
  "created_at": "2023-03-15T09:00:00Z",
  "updated_at": "2025-04-28T14:32:11Z"
}
```


Рис. 2.5 - Приклад типової JSON-відповіді

Поле `next_maintenance` є обчислюваним і не зберігається в базі даних безпосередньо - воно розраховується в сервісному шарі на основі поточного одометра ТЗ та параметрів останнього закритого ТО. Це є прикладом принципу «тонкого серіалізатора»: серіалізатор лише структурує дані, тоді як обчислення виконуються в Service Layer.

Таким чином, у підрозділі 2.1 визначено трирівневу клієнт-серверну архітектуру з REST API як міжрівневим контрактом, обґрунтовано застосування

шаблонів Service Layer, Repository та DTO, а також специфіковано 27 ендпоінтів із зазначенням методів, URL та ролей доступу. Отримана специфікація є достатньою основою для паралельної реалізації серверної та клієнтської частин у розділі 3.

2.2 Проєктування моделі бази даних

2.2.1 Концептуальна модель даних (ER-діаграма)

Проєктування концептуальної моделі даних є визначальним кроком, що встановлює структуру інформаційного простору системи та визначає обмеження цілісності ще до написання коду. Некоректно спроєктована модель на цьому етапі призводить до каскадних проблем у наступних ітераціях: надлишкового дублювання, неможливості побудови коректних аналітичних запитів та порушень узгодженості даних при паралельних операціях [18].

Виявлені сутності та їх атрибути. Аналіз бізнес-процесів (підрозділ 1.1.1) дозволив виявити шість основних сутностей системи.

Сутність **Vehicle** (транспортний засіб) є центральною в моделі: до неї безпосередньо або через асоціативні сутності прив’язані всі операційні записи. Атрибути: унікальний ідентифікатор, державний номерний знак (унікальний), VIN-код (унікальний), марка, модель, рік випуску, тип палива (перелічуваний тип), нормативна витрата пального у літніх та зимових умовах, поточний пробіг, статус (перелічуваний тип: `active` / `maintenance` / `decommissioned`), дата реєстрації в системі.

Сутність **Driver** (водій): ПІБ, табельний номер (унікальний), дата народження, контактний телефон, категорія водійського посвідчення, номер посвідчення, дата видачі, термін дії, статус (`active` / `dismissed`).

Сутність **VehicleAssignment** є асоціативною і реалізує зв’язок M:N між Vehicle та Driver у часі. Вона містить посилання на ТЗ та водія, дату початку закріплення та дату завершення (NULL означає поточне активне закріплення). Відокремлення цієї сутності від Vehicle є принциповим рішенням: якби ідентифікатор водія зберігався безпосередньо у Vehicle як атрибут, система

втратила б можливість коректно атрибутувати витрати пального конкретному водієві за минулі periodи при зміні закріплення.

Сутність **FuelingRecord** (запис заправки): посилання на ТЗ та водія, дата і час, кількість пального (літри), ціна за літр, назва АЗС, показання одометра, обчислювані поля - фактична витрата (л/100 км), відхилення від норми у відсотках, ознака аномальності.

Сутність **MaintenanceRecord** (заявка на ремонт/ТО): посилання на ТЗ, тип запису (`planned_to` / `current_repair` / `major_repair`), дата відкриття, дата закриття (NULL - заявка відкрита), опис несправності, опис виконаних робіт, назва виконавця, вартість матеріалів, вартість робіт, показання одометра на момент закриття.

Сутність **User** (обліковий запис): посилання на профіль водія (NULL для ролей, не пов'язаних із водієм), ім'я користувача, хеш пароля, електронна пошта, роль (`admin` / `manager` / `dispatcher` / `driver`), ознака активності.

Аналіз зв'язків. Відношення між Vehicle та FuelingRecord є 1:N із обов'язковим посиланням на Driver: заправка без прив'язки до водія є неприпустимою з точки зору бізнес-логіки, оскільки унеможливлює формування звіту витрат у розрізі водіїв. Відношення між Driver та User є 1:0..1 (один до нуля або одного): не кожен водій обов'язково має обліковий запис у системі, проте кожен обліковий запис з роллю `driver` повинен бути пов'язаний із конкретним водієм. Це забезпечується необов'язковим зовнішнім ключем `driver_id` у таблиці `users` з обмеженням `UNIQUE`, що запобігає дублюванню облікових записів для одного водія.

Таким чином, концептуальна модель містить шість сутностей та п'ять типів зв'язків, що повністю відображають предметну область, визначену у підрозділі 1.1. Асоціативна сутність VehicleAssignment є ключовим рішенням, що забезпечує можливість коректної ретроспективної звітності при зміні закріплення ТЗ (рис 2.6).

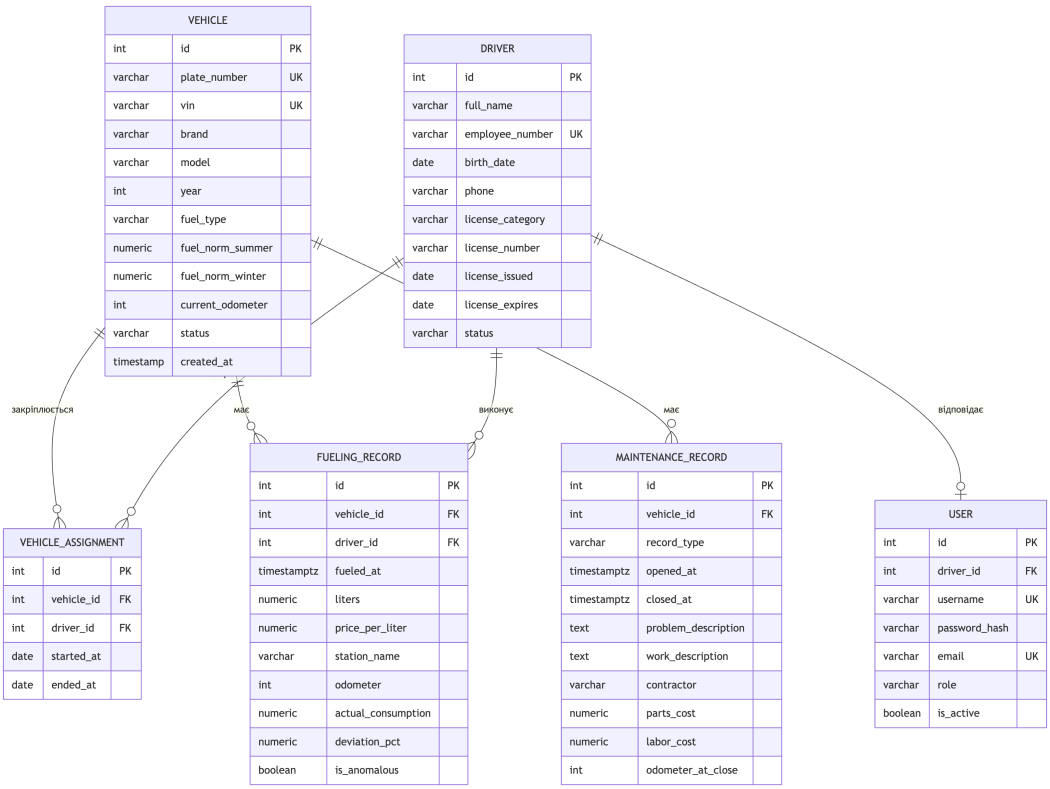


Рис. 2.6 - ER-діаграма в нотації Crow's Foot

2.2.2 Фізична модель: таблиці, типи даних, індекси

Фізична модель визначає конкретну реалізацію концептуальних сутностей засобами PostgreSQL 16: типи даних стовпців, обмеження цілісності, стратегію індексування та нормальну форму схеми. Коректний вибір типів даних безпосередньо впливає на точність обчислень у звітності та ефективність виконання запитів [8].

Вибір типів даних. Для числових полів, що беруть участь у фінансових обчисленнях (вартість пального, витрати на ремонт), обрано тип `NUMERIC(10,2)` замість `FLOAT`. Тип `FLOAT` є наближеним двійковим представленням і накопичує похибку при арифметичних операціях, що є неприйнятним для фінансової звітності: сума витрат на пальне за місяць, обчислена як `SUM(liters * price_per_liter)` над `FLOAT`-стовпцями, може відрізнятися від математично точного результату на кілька копійок. `NUMERIC` є точним десятковим типом і позбавлений цього недоліку.

Для полів дати та часу обрано `TIMESTAMPTZ` (`TIMESTAMP WITH TIME ZONE`) замість `TIMESTAMP`. PostgreSQL зберігає `TIMESTAMPTZ` у UTC та конвертує при виведенні відповідно до поточного часового поясу сесії. Для підприємства, що потенційно має підрозділи в різних часових поясах, або при переведенні годинника це запобігає неоднозначності інтерпретації часових міток [9].

Статусні поля реалізовані через PostgreSQL `ENUM`-тип, а не через `VARCHAR` із перевірочним обмеженням. Це забезпечує додатковий рівень захисту від некоректних значень на рівні СУБД та дозволяє PostgreSQL оптимізувати зберігання (enum-значення займають 4 байти, що менше за типові VARCHAR-рядки статусів).

Обмеження `CHECK` на рівні DDL є першим рубежем захисту цілісності даних, незалежним від логіки застосунку: навіть якщо серверна частина пропустить некоректне значення (наприклад, від'ємну кількість пального), PostgreSQL відхилить транзакцію.

Стратегія нормалізації. Схема приведена до третьої нормальної форми (3НФ). Наявність 3НФ гарантує: відсутність транзитивних залежностей між неключовими атрибутами; відсутність часткових залежностей від складеного ключа; відсутність дублювання даних. Зокрема, атрибути водія (ПІБ, категорія посвідчення) не дублюються у `fueling_records` - замість цього використовується зовнішній ключ `driver_id`. Якби ці дані зберігалися безпосередньо в записі заправки (денормалізація), зміна прізвища водія потребувала б оновлення всіх пов'язаних записів з ризиком неузгодженості.

Єдиним свідомим відступом від повної нормалізації є зберігання обчислюваних полів `actual_consumption`, `deviation_pct` та `is_anomalous` у таблиці `fueling_records`. Ці значення є функціями від `liters` та різниці одометрів між суміжними записами. Матеріалізація обчислень мотивована продуктивністю: їх повторний розрахунок при кожному запиті до журналу заправок або звітності вимагав би рекурсивного оконного запиту з `LAG()`, що суттєво ускладнює запити та знижує їхню швидкість при великих обсягах даних.

Стратегія індексування. Правильне індексування є визначальним для продуктивності запитів у модулі звітності, де більшість запитів фільтрують дані за `vehicle_id` та `fueled_at` одночасно. B-tree індекс є стандартним вибором для рівності та діапазонних умов [3].

2.3 Проєктування UML-діаграм поведінки системи

2.3.1 Діаграма прецедентів (Use Case Diagram)

Діаграма прецедентів є інструментом фіксації функціональних вимог з позиції акторів системи та дозволяє перевірити повноту визначеного у підрозділі 1.4.1 переліку вимог: кожна функціональна вимога групи «Першочергово» та «Заплановано» повинна бути відображена як мінімум одним прецедентом [27]. Для системи управління автопарком виявлено чотири актори та шістнадцять прецедентів (рис 2.7).

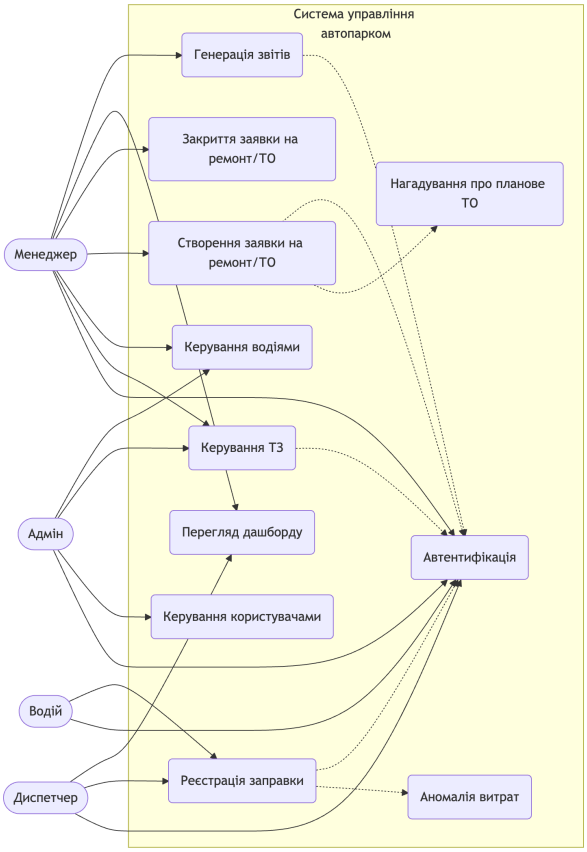


Рис. 2.7 - Діаграма прецедентів

Важливим архітектурним рішенням при побудові діаграми є застосування відношення `<<include>>` для виділення автентифікації як окремого прецеденту, від якого залежать усі інші. Альтернативою було б дублювання кроку автентифікації в кожному прецеденті, що суперечить принципу єдиної відповідальності та не відображало б реальну структуру системи, де JWT-перевірка є централізованим middleware. Відношення `<<extend>>` застосовано для опціональних розширень основного сценарію - зокрема, генерація сповіщення про аномальну витрату пального є розширенням прецеденту реєстрації заправки, а не обов'язковим кроком.

2.3.2 Діаграми послідовності для ключових сценаріїв

Діаграми послідовності (Sequence Diagrams) деталізують внутрішню взаємодію компонентів системи для конкретних сценаріїв виконання, що є необхідним для визначення меж відповідальності між шарами архітектури та передумовою коректної реалізації Service Layer [27]. Для системи управління автопарком побудовано діаграми для двох ключових сценаріїв: автентифікації користувача та реєстрації заправки.

Сценарій 1: Автентифікація користувача. Процес автентифікації є точкою входу для всіх захищених операцій системи.

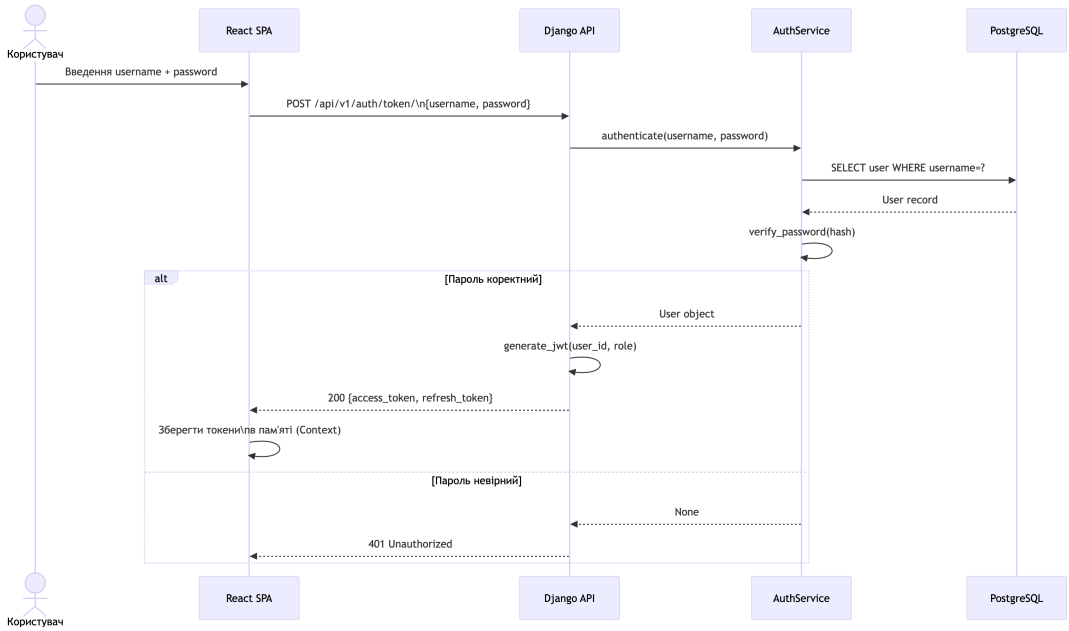


Рис. 2.8 - Діаграма послідовності для автетнифікації

Ключовим архітектурним рішенням є розподіл відповідальності: перевірка облікових даних та генерація токенів здійснюється виключно на рівні серверної частини, клієнт не має доступу до секретного ключа підпису JWT і не може самостійно генерувати або модифікувати токени (рис 2.8).

Сценарій 2: Реєстрація заправки. Цей сценарій є найбільш складним з точки зору бізнес-логіки: він включає валідацію вхідних даних, розрахунок фактичної витрати, виявлення аномалії та атомарне збереження кількох пов’язаних змін у базі даних.

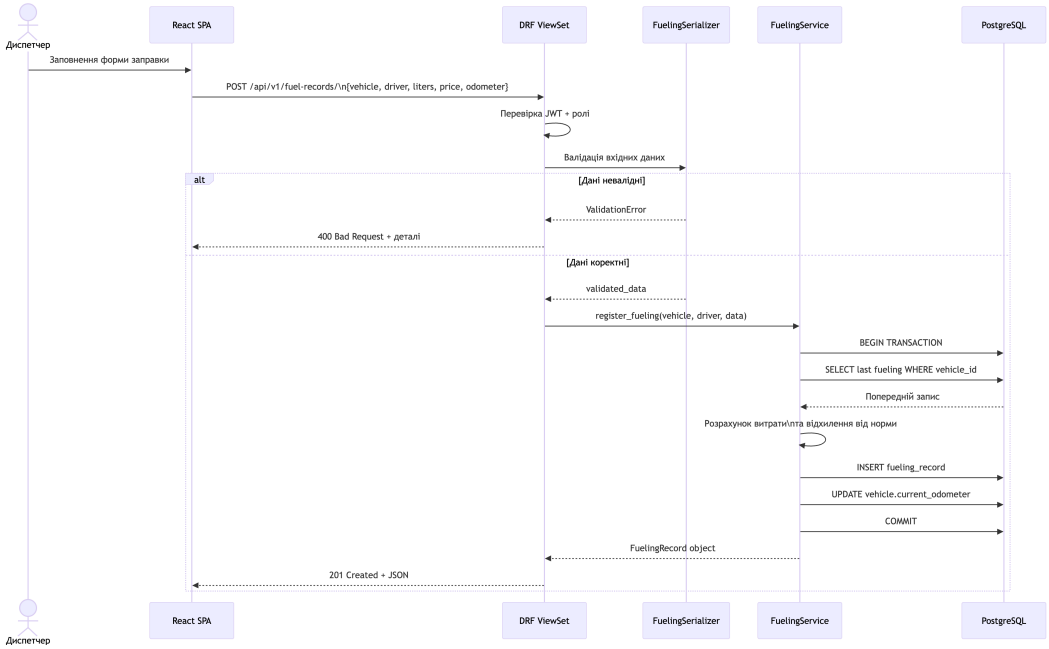


Рис. 2.9 - Діаграма послідовності для реєстрації заправки

Діаграма явно відображає транзакційну межу (`BEGIN` / `COMMIT`), що охоплює як створення запису заправки, так і оновлення одометра ТЗ. Це є прямою проєкційною відповіддю на вимогу NFR-04 щодо цілісності даних: якщо оновлення одометра завершиться помилкою після створення запису заправки, транзакція буде відкочена повністю.

Таким чином, побудовані діаграми послідовності визначають міжкомпонентні контракти для двох найбільш критичних сценаріїв та є

безпосередньою специфікацією для реалізації відповідних методів Service Layer у підрозділі 3.2.

2.3.3 Діаграма класів серверної частини

Діаграма класів фіксує статичну структуру серверної частини: моделі даних, сервісні класи та їх взаємозв'язки. На відміну від ER-діаграми (підрозділ 2.2.1), яка описує реляційну структуру бази даних, діаграма класів відображає об'єктну модель Python-коду із успадкуванням, асоціаціями та методами.

Ключовим архітектурним рішенням, що відображається на діаграмі, є розподіл між шарами: Django-моделі відповідають лише за структуру даних та відображення на таблиці СУБД, тоді як бізнес-логіка зосереджена у сервісних класах. Це унеможливлює «розбухання» моделей (Fat Model антипатерн), при якому методи на кшталт `calculate_fuel_deviation()` або `close_maintenance_request()` розміщуються безпосередньо в моделях і стають нетестованими без звернення до бази даних рис 2. [23].

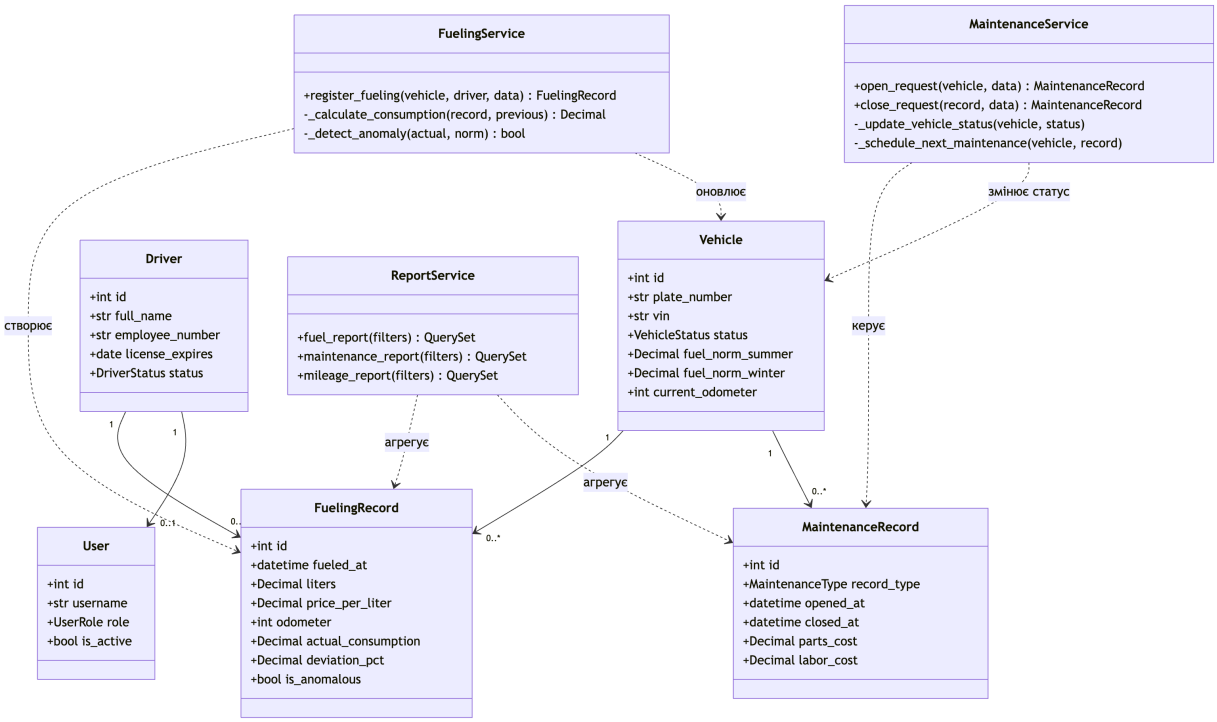


Рис. 2.10 - Діаграма класів

2.4 Проєктування підсистеми безпеки

2.4.1 Модель контролю доступу на основі ролей (RBAC)

Система управління автопарком оперує фінансово значущими даними - витратами на пальне, вартістю ремонтів, персональними даними водіїв - тому некоректна реалізація контролю доступу є не лише технічною вразливістю, але й організаційним ризиком. Завдання підрозділу - спроєктувати модель доступу, що унеможливлює несанкціонований перегляд або модифікацію даних незалежно від того, чи звертається користувач через інтерфейс, чи безпосередньо до API.

Вибір моделі контролю доступу. Серед поширених моделей контролю доступу розглянуто три: Discretionary Access Control (DAC), Mandatory Access Control (MAC) та Role-Based Access Control (RBAC). DAC делегує управління доступом власнику ресурсу, що є надто гнучким для корпоративної системи: менеджер автопарку не повинен мати можливість самостійно надавати диспетчеру доступ до звітності. MAC визначає доступ на підставі класифікаційних міток ресурсів і суб'єктів - підхід, типовий для систем державної безпеки, де надлишково складний для задачі автопарку. RBAC (ANSI/INCITS 359-2004) відповідає природній організаційній структурі підприємства: права призначаються ролям, а не окремим користувачам, що суттєво спрощує адміністрування при зміні персоналу [19].

Дворівнева реалізація RBAC у Django/DRF. Ключова проблема наївної реалізації RBAC полягає в тому, що перевірка ролі лише на рівні маршруту (чи є користувач менеджером) є недостатньою: водій, автентифікований у системі, технічно може надіслати запит `GET /api/v1/fuel-records/` і отримати заправки інших водіїв, якщо фільтрація не реалізована на рівні queryset. Тому реалізовано дворівневу перевірку:

Рівень 1 - перевірка ролі (Permission class): визначає, чи має роль користувача право виконувати цей тип операції над цим ресурсом взагалі.

Рівень 2 - фільтрація queryset: обмежує множину об’єктів, до яких має доступ конкретний користувач, незалежно від того, якими параметрами він фільтрує запит.

Метод `get_queryset()` гарантує, що водій, навіть знаючи ідентифікатор чужого запису заправки, не зможе отримати його через `GET /api/v1/fuel-records/{id}/`: DRF виконує lookup лише в межах повернутого queryset, і чужий запис просто не буде знайдено (відповідь 404, а не 403 - що є правильним з точки зору безпеки, оскільки не розкриває факт існування ресурсу). Реалізацію зображено на рисунку 2.11.

```python
class FuelingRecordViewSet(viewsets.ModelViewSet):
    def get_permissions(self):
        if self.action in ('list', 'retrieve'):
            return [IsAuthenticated()]
        return [IsDispatcherOrAbove()]

    def get_queryset(self):
        user = self.request.user
        qs = FuelingRecord.objects.select_related('vehicle', 'driver')
        if user.role == 'driver':
            return qs.filter(driver=user.driver_profile)
        return qs
```

Рис. 2.11 - Дворівнева реалізація RBAC

Матрицю доступу до ендпоінтів API наведено в таблиці 2.3.

Таблиця 2.1

Матриця доступу RBAC для ресурсів API

Ресурс	Операція	Адмін	Менеджер	Диспетчер	Водій
`/vehicles/`	GET (список)	Так	Так	Так	Так
`/vehicles/`	POST, PATCH	Так	Так	-	-
`/vehicles/{id}/decommission/`	POST	Так	Так	-	-
`/drivers/`	GET, POST, PATCH	Так	Так	GET	-
`/fuel-records/`	GET (список)	Так	Так	Так	Так
`/fuel-records/`	POST	Так	Так	Так	-
`/maintenance/`	GET	Так	Так	Так	-
`/maintenance/`	POST, PATCH	Так	Так	-	-
`/maintenance/{id}/close/`	POST	Так	Так	-	-
`/reports/*`	GET	Так	Так	-	-
`/users/`	всі операції	Так	-	-	-

2.4.2 Автентифікація та авторизація: JWT-токени

Вибір механізму автентифікації для REST API визначає як рівень безпеки системи, так і її масштабованість. Завдання підрозділу - обґрунтувати архітектурні рішення щодо JWT та описати захист від релевантних атак із OWASP Top 10 [7].

Вибір JWT над сесійною автентифікацією. Традиційна сесійна автентифікація Django зберігає ідентифікатор сесії в базі даних і перевіряє його при кожному запиті. Для REST API це означає звернення до бази при кожному запиті виключно з метою перевірки автентичності - навіть для `GET`-запитів, що не модифікують дані. JWT є самодостатнім: сервер верифікує підпис токена криптографічно без звернення до бази, що знижує навантаження на PostgreSQL та спрощує горизонтальне масштабування [22].

Алгоритм підпису: HS256 vs RS256. Для підпису JWT обрано алгоритм HS256 (HMAC-SHA256) замість RS256 (RSA-SHA256). RS256 виправданий в мікросервісній архітектурі, де множина сервісів повинна верифікувати токени, не маючи доступу до секретного ключа: відкритий ключ RSA-пари роздається всім

сервісам для верифікації, тоді як підпис здійснює лише один авторизаційний сервіс. Для монолітного Django-застосунку, де єдиний backend і підписує, і верифікує токени, складність PKI-інфраструктури RS256 є надлишковою. HS256 із достатньо довгим (≥256 біт) секретним ключем, що зберігається у змінній середовища, є цілком безпечним рішенням для даної архітектури.

Архітектура токенів. Система використовує пару access + refresh токенів. Access-токен має короткий час життя (15 хвилин), що обмежує вікно атаки у випадку його перехоплення: навіть якщо зловмисник отримає access-токен, він буде дійсним лише 15 хвилин. Refresh-токен із терміном дії 7 днів використовується виключно для отримання нового access-токена.

Включення поля `role` до payload усуває необхідність звернення до бази при кожній перевірці авторизації: DRF зчитує роль безпосередньо з токена. Це є свідомим компромісом: при зміні ролі користувача вже видані access-токени залишаться з попередньою роллю до їх закінчення. Оскільки термін дії access-токена становить 15 хвилин, цей ризик є прийнятним.

Захист від атак OWASP Top 10. Таблицю 2.2 наведено нижче.

Таблиця 2.2

Захист від релевантних загроз OWASP Top 10 (2021)

Загроза OWASP	**Механізм захисту в системі**
A01: Broken Access Control	Дворівневий RBAC (Permission class + `get_queryset()`), перевірка JWT на кожному запиті
A02: Cryptographic Failures	HS256 для JWT, PBKDF2-SHA256 для паролів, HTTPS через nginx
A03: Injection	Параметризовані запити Django ORM; вхідні дані валідуються DRF Serializers перед використанням
A07: Auth Failures	httpOnly cookie для refresh-токена; короткий TTL access-токена (15 хв); `jti` (JWT ID) для можливості інвалідації
A09: Logging Failures	Логування всіх запитів на рівні nginx та Django middleware з IP, user_id, ендпоінтом

Захист від **XSS** забезпечується зберіганням access-токена в пам'яті React Context (не в `localStorage` чи `sessionStorage`): сторонній скрипт, ін'єктований через XSS, не має доступу до змінних JavaScript-модуля. Refresh-токен зберігається у `httpOnly`-куці і також недоступний через `document.cookie`. Захист від **CSRF** для JWT-автентифікації є автоматичним: CSRF-атака базується на тому, що браузер автоматично додає cookie до міжсайтових запитів, проте авторизаційний заголовок `Authorization: Bearer <token>` браузер автоматично не додає - його повинен явно встановити JavaScript-код легітимного SPA.

Для реалізації JWT-автентифікації обрано бібліотеку `djangorestframework-simplejwt`, що надає готові endpoint-and для отримання та оновлення токенів, підтримує blacklist refresh-токенів (зберігання інвалідованих `jti` у базі даних для коректного logout) та інтегрується з DRF через стандартний клас `JWTAuthentication`.

```
SIMPLE_JWT = {
    "ACCESS_TOKEN_LIFETIME": timedelta(minutes=15),
    "REFRESH_TOKEN_LIFETIME": timedelta(days=7),
    "ALGORITHM": "HS256",
    "SIGNING_KEY": env("DJANGO_SECRET_KEY"),
    "AUTH_HEADER_TYPES": ("Bearer",),
    "ROTATE_REFRESH_TOKENS": True,
    "BLACKLIST_AFTER_ROTATION": True,
```

Рис. 2.12 - Реалізація JWT-автентифікації

Параметр `ROTATE_REFRESH_TOKENS` забезпечує автоматичну заміну refresh-токена при кожному оновленні access-токена, а `BLACKLIST_AFTER_ROTATION` - інвалідацію попереднього. Це захищає від атаки повторного використання: якщо зловмисник перехопить refresh-токен і спробує скористатися ним після того, як легітимний клієнт вже його використав, сервер відхилить запит, оскільки токен потрапить у blacklist.

Таким чином, підсистема безпеки спроєктована з урахуванням специфіки REST API: дворівневий RBAC унеможливлює несанкціонований доступ до даних навіть при прямих API-запитах; JWT з розділеними access та refresh токенами

мінімізує вікно атаки при компрометації токена; захист від XSS та CSRF реалізується через архітектурні рішення щодо зберігання токенів, а не лише через заголовки відповідей. Спроєктована підсистема безпеки є достатньою основою для реалізації відповідних компонентів у підрозділах 3.2 та 3.3.

3. ПРАКТИЧНА РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ ВЕБ-ДОДАТКУ

3.1 Організація проєкту та середовище розробки

3.1.1 Структура директорій проєкту

Організація файлової структури проєкту безпосередньо впливає на підтримуваність коду: відсутність чіткої конвенції призводить до того, що бізнес-логіка, конфігурація та тести змішуються в одних директоріях, ускладнюючи навігацію та збільшуючи ризик циклічних залежностей між модулями [21]. Завдання підрозділу - визначити таку структуру директорій, що відображає архітектурне розшарування, спроєктоване у розділі 2, та унеможливлює порушення шарових залежностей на рівні файлової системи.

Структура серверної частини. Django-проєкт організовано за принципом розділення за функціональними шарами, а не за доменними сутностями. Альтернативний підхід - «feature-based» структура, де кожна доменна область (vehicles, drivers, fueling) є окремим Django-застосунком із власними моделями, views та тестами - є більш придатним для великих команд, де різні розробники відповідають за різні домени. Для одноосібного проєкту горизонтальний поділ за шарами (`models/`, `services/`, `api/`) є більш наочним: при розробці нової функції чіткіше зрозуміло, де розміщувати кожен тип файлу.

Структура серверної частини організована навколо кореневої директорії `backend/`, яка містить два основних компоненти: конфігураційний пакет `fleet/` та директорію `apps/` із доменними застосунками.

Пакет `fleet/` відповідає виключно за конфігурацію проєкту. Він містить підпакет `settings/` із трьома файлами: `base.py` зберігає налаштування, спільні для всіх середовищ; `development.py` розширює базові налаштування значеннями, специфічними для локальної розробки; `production.py` визначає параметри

виробничого середовища. Окрім налаштувань, у `fleet/` розміщено `urls.py` - кореневий маршрутизатор - та `wsgi.py` для запуску через Gunicorn.

Директорія `apps/` містить п'ять Django-застосунків: `vehicles/`, `fueling/`, `drivers/`, `reports/` та `users/`. Кожен застосунок має однакову внутрішню структуру: `models.py` визначає Django-моделі та менеджери; `serializers.py` містить DRF-серіалізатори для валідації та серіалізації; `views.py` - ViewSet-класи; `services.py` - сервісний шар із бізнес-логікою; `permissions.py` - класи перевірки прав доступу. Тести кожного застосунку розміщено у підпакеті `tests/`, що містить окремі файли для моделей, сервісів та API-ендпоінтів.

Розділення налаштувань на `base.py` / `development.py` / `production.py` вирішує типову проблему: єдиний `settings.py` змушує або комітити чутливі дані до репозиторію, або використовувати складні умовні конструкції `if DEBUG`. При такій структурі `DJANGO_SETTINGS_MODULE=fleet.settings.production` у виробничому середовищі та `fleet.settings.development` локально - і жодних умов у коді.

Розміщення тестів усередині кожного застосунку (а не в окремій директорії `tests/` на рівні проєкту) забезпечує локальність: тести для `fueling/services.py` знаходяться в `fueling/tests/test_services.py`, що спрощує навігацію та унеможливлює ситуацію, коли зміна моделі не супроводжується оновленням відповідного тесту через просторову віддаленість файлів.

Структура клієнтської частини зосереджена в директорії `frontend/src/`, яка розбита на сім функціональних директорій.

`api/` містить типізовані функції для HTTP-запитів, згруповані за ресурсами (`vehicles.ts`, `fueling.ts` тощо), та файл `client.ts` із налаштованим Axios-інстансом і перехоплювачами запитів. `components/` зберігає перевикористовувані UI-компоненти, кожен із яких розміщено у власній піддиректорії разом із файлом стилів. `pages/` містить компоненти верхнього рівня, що відповідають маршрутам застосунку: один файл - один маршрут. `context/` зберігає React Context-провайдери, зокрема `AuthContext` із поточним користувачем та токенами. `hooks/` містить кастомні хуки, що інкапсулюють логіку роботи з даними. `types/` - центральне місце для

TypeScript-інтерфейсів усіх доменних сутностей. `utils/` містить допоміжні функції без залежностей від React.

Виділення директорії `api/` як окремого шару між компонентами та мережею є реалізацією того самого принципу Service Layer на фронтенді: компоненти не виконують `axios.get()` безпосередньо, а викликають типізовані функції з `api/vehicles.ts`. При зміні URL ендпоінту або структури відповіді зміна вноситься в одному місці, а не у всіх компонентах, що звертаються до цього ресурсу.

Таким чином, файлова структура проєкту відображає архітектурне розшарування (моделі, сервіси, API, тести на бекенді; api-шар, компоненти, сторінки на фронтенді) та є операційною основою для реалізації у підрозділах 3.2 та 3.3.

3.1.2 Налаштування середовища розробки та залежностей

Відтворюваність середовища розробки є передумовою коректності тестування: якщо локальне середовище відрізняється від виробничого (версія Python, версії бібліотек, налаштування лінтерів), код може проходити тести локально та давати збій при розгортанні. Завдання підрозділу - визначити інструменти та конфігурації, що фіксують середовище та забезпечують однакову поведінку на будь-якій машині розробника.

Управління залежностями Python. Для управління залежностями серверної частини обрано `pyproject.toml` (PEP 518) замість `requirements.txt`. `requirements.txt` є плоским списком без розрізнення прямих та транзитивних залежностей, що ускладнює оновлення: при виконанні `pip freeze > requirements.txt` до файлу потрапляють усі транзитивні залежності без позначення їх джерела. `pyproject.toml` з інструментом `pip-tools` або `poetry` дозволяє явно визначити прямі залежності, а lock-файл фіксує точні версії транзитивних.

Розділення залежностей на основні та `dev` забезпечує, що `ruff` та `pytest` не потраплять до виробничого Docker-образу, скорочуючи його розмір та зменшуючи поверхню атаки.

Лінтери та форматери. Для Python обрано `ruff` - інструмент, що поєднує функції лінтера (перевірка стилю, виявлення потенційних помилок) та форматера в одному бінарному файлі, реалізованому на Rust. Порівняно із традиційним стеком `flake8 + black + isort`, `ruff` виконує всі три задачі та є у 10-100 разів швидшим, що є суттєвим при запуску в pre-commit хуках [26]. Для TypeScript використовується `ESLint` із плагіном `@typescript-eslint` та `Prettier` для форматування.

Включення правил групи `S` (bandit) дозволяє виявляти потенційні вразливості безпеки статичним аналізом - наприклад, використання `assert` у виробничому коді або небезпечні виклики `subprocess` (Рис. 3.1).

```
# ruff.toml
line-length = 100
target-version = "py312"

[lint]
select = ["E", "F", "I", "N", "UP", "S"]
# E: pycodestyle, F: pyflakes, I: isort,
# N: pep8-naming, UP: pyupgrade, S: bandit (безпека)
```

Рис. 3.1 - Інструмент Ruff

Управління секретами через змінні середовища. Усі чутливі дані (ключ Django, пароль PostgreSQL, секрет JWT) зберігаються у файлі `.env`, що не комітиться до репозиторію. Репозиторій містить лише `.env.example` із переліком необхідних змінних без значень (рис 3.2) .

```
import environ
env = environ.Env(DEBUG=(bool, False))
environ.Env.read_env(BASE_DIR / '.env')

SECRET_KEY = env('DJANGO_SECRET_KEY')
DATABASES = {'default': env.db('DATABASE_URL')}
```

Рис. 3.2 - Реалізація управління секретами

Якщо обов'язкова змінна відсутня у середовищі, `django-environ` підніме виняток при старті застосунку - до обробки першого запиту. Це є кращою поведінкою порівняно з падінням у runtime при першому зверненні до змінної, оскільки помилку конфігурації виявлено раніше.

Таблиця 3.1

Інструменти середовища розробки

Задача	**Інструмент**	**Альтернатива**	**Причина вибору**
Залежності Python	pyproject.toml + pip-tools	requirements.txt	Розрізнення прямих та транзитивних залежностей
Лінтер + форматер Python	ruff	flake8 + black + isort	Єдиний інструмент, швидкість виконання
Лінтер TypeScript	ESLint + @typescript-eslint	TSLint (deprecated)	TSLint більше не підтримується
Форматер TypeScript	Prettier	-	Де-факто стандарт
Змінні середовища	django-environ	python-decouple	Підтримка DATABASE_URL формату
Складання фронтенду	Vite	Create React App	CRA більше не підтримується; Vite швидший

Таким чином, середовище розробки сконфігуровано так, що відтворюється на будь-якій машині через клонування репозиторію та копіювання `.env`-файлу; статичний аналіз виявляє стильові порушення та потенційні вразливості ще до запуску тестів; управління залежностями з розділенням на основні та dev-залежності забезпечує мінімальний розмір виробничого образу.

3.2 Реалізація серверної частини - Django і DRF

3.2.1 Реалізація моделей даних та Django ORM

Реалізація моделей є точкою відповідності між фізичною схемою бази даних, спроєктованою у підрозділі 2.2.2, та об'єктною моделлю Python-коду. Завдання - не лише відобразити таблиці у класи, а й перенести обмеження цілісності на рівень застосунку так, щоб вони спрацьовували незалежно від того, чи надходять дані через API, чи через Django admin.

Моделі Vehicle та Driver. Статусні поля реалізовані через вкладені класи `TextChoices`, що є Django-специфічною заміною Python `enum.Enum`. На відміну від рядкових констант, `TextChoices` дозволяє Django автоматично генерувати список допустимих значень для поля `choices`, що відображається як `ENUM`-обмеження у міграції PostgreSQL (рис 3.3).

```python
class Vehicle(models.Model):
    class Status(models.TextChoices):
        ACTIVE = 'active', 'Активний'
        MAINTENANCE = 'maintenance', 'На ремонті'
        DECOMMISSIONED = 'decommissioned', 'Списаний'

    plate_number = models.CharField(max_length=10, unique=True)
    vin = models.CharField(max_length=17, unique=True)
    fuel_norm_summer = models.DecimalField(max_digits=5, decimal_places=2)
    fuel_norm_winter = models.DecimalField(max_digits=5, decimal_places=2)
    current_odometer = models.PositiveIntegerField(default=0)
    status = models.CharField(
        max_length=15, choices=Status.choices, default=Status.ACTIVE
    )

    class Meta:
        indexes = [
            models.Index(fields=['status'], name='idx_vehicle_status'),
        ]
```

Рис. 3.3 - Обмеження у міграції PostgreSQL

Використання `DecimalField` замість `FloatField` для норм витрат є реалізацією рішення, обґрунтованого у підрозділі 2.2.2: Django ORM транслює `DecimalField` у PostgreSQL `NUMERIC`, зберігаючи точність при агрегуванні у звітах.

Модель FuelingRecord. Ключовою особливістю є поля `actual_consumption`, `deviation_pct` та `is_anomalous`, які є обчислюваними та не заповнюються клієнтом. Щоб унеможливити їх запис через API, відповідні серіалізатори позначають ці поля як `read_only=True`. На рівні моделі вони визначені зі значенням `null=True`, оскільки для першого запису заправки по конкретному ТЗ попередній одометр відсутній і розрахунок неможливий.

Стратегія `on_delete=models.PROTECT` для зовнішніх ключів обрана свідомо: вона забороняє видалення ТЗ або водія, якщо для них існують записи заправок. Альтернатива `CASCADE` призвела б до мовчазного знищення фінансових даних при деактивації водія - неприйнятна поведінка для системи обліку (рис. 3.4).

```python
class FuelingRecord(models.Model):
    vehicle = models.ForeignKey(Vehicle, on_delete=models.PROTECT,
                                related_name='fueling_records')
    driver = models.ForeignKey(Driver, on_delete=models.PROTECT,
                               related_name='fueling_records')
    fueled_at = models.DateTimeField()
    liters = models.DecimalField(max_digits=7, decimal_places=2)
    price_per_liter = models.DecimalField(max_digits=6, decimal_places=2)
    odometer = models.PositiveIntegerField()
    actual_consumption = models.DecimalField(
        max_digits=6, decimal_places=2, null=True, blank=True
    )
    deviation_pct = models.DecimalField(
        max_digits=6, decimal_places=1, null=True, blank=True
    )
    is_anomalous = models.BooleanField(default=False)
```

Рис. 3.4 - Модель FuelingRecord

Таким чином, моделі реалізують усі обмеження фізичної схеми на рівні Python-коду, а правильний вибір типів полів гарантує точність обчислень у наступному шарі.

3.2.2 Реалізація бізнес-логіки: Service Layer

Концентрація бізнес-логіки у моделях або views є типовою проблемою Django-проєктів, що ускладнює тестування та призводить до дублювання коду при появі кількох точок входу (API, management commands, адмін-панель). Реалізація Service Layer, спроєктованого у підрозділі 2.1.2, розв'язує цю проблему: кожен сервісний клас відповідає за один доменний процес.

FuelingService. Центральним методом є `register_fueling()`, що виконує чотири операції в рамках однієї транзакції: знаходить попередній запис заправки для отримання базового одометра, розраховує фактичну витрату та відхилення, позначає аномальні записи та оновлює поточний одометр ТЗ.

Використання `.update()` замість `.save()` для запису обчислених полів є оптимізацією: `save()` генерує `UPDATE` по всіх полях моделі, тоді як `.update()` - лише по вказаних. Для таблиці з потенційно великою кількістю одночасних записів (диспетчери реєструють заправки паралельно) це зменшує ризик конфліктів блокувань (рис 3.5).

```python
class FuelingService:
    ANOMALY_THRESHOLD = Decimal('10.0')

    @classmethod
    @transaction.atomic
    def register_fueling(cls, vehicle, driver, validated_data):
        previous = FuelingRecord.objects.filter(
            vehicle=vehicle
        ).order_by('-odometer').first()

        record = FuelingRecord.objects.create(
            vehicle=vehicle, driver=driver, **validated_data
        )

        if previous:
            distance = record.odometer - previous.odometer
            if distance > 0:
                actual = (record.liters / Decimal(distance)) * 100
                norm = vehicle.fuel_norm_summer
                deviation = ((actual - norm) / norm) * 100
                FuelingRecord.objects.filter(pk=record.pk).update(
                    actual_consumption=actual,
                    deviation_pct=deviation,
                    is_anomalous=deviation > cls.ANOMALY_THRESHOLD
                )

        Vehicle.objects.filter(pk=vehicle.pk).update(
            current_odometer=record.odometer
        )
        return record
```

Рис. 3.5 - Клас FuelingService

MaintenanceService. Закриття ремонтної заявки є складним сценарієм, що зачіпає три таблиці: оновлення `MaintenanceRecord`, зміна статусу `Vehicle` та запис дати наступного ТО.

Відсутність перевірки `if record.closed_at is not None` перед збереженням є свідомим рішенням: ця перевірка розміщена у DRF-серіалізаторі методом `validate()`, що є правильним рівнем для валідації стану ресурсу. Сервіс отримує вже провалідовані дані та не дублює логіку перевірки.

Таким чином, Service Layer інкапсулює всю бізнес-логіку системи у вигляді методів з явними вхідними та вихідними типами, що безпосередньо забезпечує тестованість, задекларовану у NFR-07 (рис.3.6).

```python
class MaintenanceService:

    @staticmethod
    @transaction.atomic
    def close_request(record, validated_data):
        record.closed_at = timezone.now()
        record.odometer_at_close = validated_data['odometer_at_close']
        record.work_description = validated_data['work_description']
        record.labor_cost = validated_data['labor_cost']
        record.parts_cost = validated_data['parts_cost']
        record.save()

        Vehicle.objects.filter(pk=record.vehicle_id).update(
            status=Vehicle.Status.ACTIVE,
            current_odometer=record.odometer_at_close
        )
        return record
```

Рис. 3.6 - Клас MaintenanceService

3.2.3 Реалізація REST API: DRF ViewSets та middleware

Шар API є точкою входу для всіх клієнтських запитів і відповідає за три задачі: маршрутизацію запитів до відповідних обробників, делегування бізнес-логіки сервісному шару та стандартизацію формату відповідей. Змішування цих задач у views є антипатерном, що призводить до нетестованого та нерозширюваного коду.

Стандартизація формату помилок через middleware. За замовчуванням DRF повертає помилки валідації у форматі `{"field": ["message"]}`, помилки

автентифікації у `{"detail": "..."}`, а необроблені Python-виключення - як HTML-сторінку Django. Різнорідність форматів ускладнює обробку помилок на фронтенді.

Пагінація та фільтрація. Для журналу заправок та списку ТЗ реалізовано курсорну пагінацію замість стандартної сторінкової. Сторінкова пагінація (`LIMIT/OFFSET`) має проблему продуктивності при великих зміщеннях: запит `OFFSET 10000` змушує PostgreSQL просканувати та відкинути 10 000 рядків перед поверненням результату. Курсорна пагінація використовує впорядковане поле (наприклад, `fueled_at`) як курсор і завжди виконується як діапазонний запит, що ефективно використовує індекс (рис.3.7).

```
REST_FRAMEWORK = {
    'DEFAULT_PAGINATION_CLASS':
        'rest_framework.pagination.CursorPagination',
    'PAGE_SIZE': 50,
    'DEFAULT_FILTER_BACKENDS': [
        'django_filters.rest_framework.DjangoFilterBackend',
        'rest_framework.filters.OrderingFilter',
    ],
    'EXCEPTION_HANDLER': 'fleet.exceptions.custom_exception_handler',
}
```

Рис. 3.7 - Курсорна пагінація

Фільтрація реалізована через `django-filter`: для `FuelingRecordViewSet` визначено `filterset_fields = ['vehicle', 'driver', 'is_anomalous']` та діапазонні фільтри для `fueled_at`. Це дозволяє клієнту формувати запити на кшталт `GET /api/v1/fuel-records/?vehicle=12&fueled_at__gte=2025-01-01&is_anomalous=true` без написання логіки фільтрації у views.

Таким чином, реалізація серверної частини відповідає архітектурним рішенням розділу 2: ViewSets відповідають виключно за маршрутизацію та делегування, бізнес-логіка зосереджена у сервісах, а стандартизований формат помилок і курсорна пагінація є конкретними реалізаціями нефункціональних вимог NFR-01 та NFR-04.

3.3 Реалізація клієнтської частини - React + TypeScript

3.3.1 Компонентна архітектура та маршрутизація

Організація компонентів React-застосунку є архітектурним рішенням, що визначає підтримуваність коду не менше, ніж структура серверної частини. Відсутність чіткої ієрархії призводить до «prop drilling» - передачі стану через кілька рівнів компонентів - та до дублювання UI-логіки, яке проявляється щойно з'являється другий екран із подібною таблицею або формою.

Ієрархія компонентів. Застосунок побудовано за трирівневою ієрархією: компоненти-сторінки (`pages/`), складені компоненти (`components/`) та примітивні UI-компоненти. Сторінки відповідають маршрутам та відповідають за завантаження даних через хуки. Складені компоненти (`VehicleTable`, `FuelingForm`, `MaintenanceCard`) інкапсулюють доменну логіку відображення та не знають про маршрутизацію. Такий поділ дозволяє перевикористовувати, наприклад, `FuelingForm` як у сторінці реєстрації нової заправки, так і в модальному вікні редагування.

Типізація є визначальною для підтримуваності: TypeScript-інтерфейси доменних сутностей визначені в `src/types/` і є єдиним джерелом істини для всіх компонентів та API-функцій (рис. 3.8).

```typescript
export interface Vehicle {
  id: number;
  plate_number: string;
  status: 'active' | 'maintenance' | 'decommissioned';
  fuel_norm_summer: string;
  current_odometer: number;
  assigned_driver: { id: number; full_name: string } | null;
  next_maintenance: {
    km_remaining: number;
    days_remaining: number;
    is_overdue: boolean;
  } | null;
}
```

Рис. 3.8. - Типізація Vehicle

Використання рядкових literal union типів для `status` замість загального `string` дозволяє TypeScript перевіряти коректність умовних відображень та автодоповнення в IDE.

Маршрутизація. Маршрутизацію реалізовано за допомогою `React Router v6` із захистом маршрутів на основі ролі користувача. Компонент `ProtectedRoute` перевіряє наявність автентифікації та відповідність ролі перед рендерингом сторінки (рис 3.9.).

```
const ProtectedRoute = ({ allowedRoles }: { allowedRoles: UserRole[] }) => {
  const { user } = useAuth();
  if (!user) return <Navigate to="/login" replace />;
  if (!allowedRoles.includes(user.role)) return <Navigate to="/403" replace />;
  return <Outlet />;
};
```

Рис. 3.9. - Маршутизація

Захист маршрутів на фронтенді є доповненням до серверного RBAC, а не його заміною: він покращує UX (користувач не бачить недоступних пунктів меню), але не є засобом безпеки - фінальна перевірка завжди відбувається на сервері.

Маршрути системи відповідають функціональним вимогам: `/dashboard` для загальної статистики, `/vehicles` та `/vehicles/:id` для управління ТЗ, `/drivers` для водіїв, `/fuel` для журналу заправок, `/maintenance` для ремонтів та `/reports` для звітності. Маршрути `/reports` та `/vehicles` (POST/PATCH) доступні лише ролям `admin` та `manager`.

Таким чином, компонентна архітектура забезпечує чіткий розподіл між логікою завантаження даних (сторінки), доменним відображенням (складені компоненти) та захистом доступу (ProtectedRoute), що відповідає принципу єдиної відповідальності на рівні UI.

3.3.2 Взаємодія з API: HTTP-клієнт та управління станом

Некерована взаємодія компонентів з API призводить до дублювання логіки: обробка помилок, оновлення токенів та індикація завантаження реалізуються

повторно у кожному компоненті. Завдання - централізувати ці аспекти так, щоб компоненти оперували лише бізнес-даними, а не деталями транспортного рівня.

HTTP-клієнт. Axios-інстанс із перехоплювачами є єдиною точкою для всіх HTTP-запитів застосунку. Перехоплювач запиту додає заголовок `Authorization` з access-токеном; перехоплювач відповіді обробляє помилку 401, автоматично оновлює токен через `/auth/token/refresh/` та повторює оригінальний запит.

Прапорець `_retry` запобігає нескінченному циклу: якщо оновлення токена також повертає 401 (refresh-токен прострочений), перехоплювач не намагається повторити запит вдруге, а перенаправляє користувача на сторінку входу (рис 3.10).

```
apiClient.interceptors.response.use(
  (response) => response,
  async (error) => {
    const original = error.config;
    if (error.response?.status === 401 && !original._retry) {
      original._retry = true;
      const newToken = await refreshAccessToken();
      original.headers['Authorization'] = `Bearer ${newToken}`;
      return apiClient(original);
    }
    return Promise.reject(error);
  }
);
```

Рис. 3.10 - Перехоплювач запиту

Управління станом. Для управління серверним станом обрано React Query замість Redux Toolkit. Redux є доцільним для складного клієнтського стану (wizard-форми, офлайн-режим), однак для системи управління автопарком переважна більшість стану є серверним: списки ТЗ, журнал заправок, дані звітів. React Query надає кешування, автоматичне перезавантаження при фокусі вікна та оптимістичні оновлення без налаштування reducers та actions.

```
export const useVehicles = () => {
  return useQuery<Vehicle[]>({
    queryKey: ['vehicles'],
    queryFn: () => api.vehicles.list(),
    staleTime: 2 * 60 * 1000,  // дані актуальні 2 хвилини
  });
};
```

Рис. 3.11 - Реалізація управління станом

Глобальний стан автентифікації (поточний користувач, токени) зберігається у `AuthContext`, оскільки він є клієнтським та не потребує кешування на рівні React Query.

Таким чином, централізований HTTP-клієнт з автоматичним оновленням токена та React Query для серверного стану усувають необхідність дублювання транспортної логіки у компонентах, залишаючи їм лише відображення даних.

3.3.3 Реалізація дашборду та модуля звітності

Дашборд та модуль звітності є найбільш складними з точки зору взаємодії з API: вони агрегують дані з кількох ресурсів та потребують ефективного перезавантаження при зміні фільтрів.

Дашборд. Головна сторінка відображає чотири KPI-лічильники (активні ТЗ, ТЗ на ремонті, витрати пального за місяць, кількість аномалій) та список ТЗ із попередженнями про наближення ТО. Дані завантажуються одним запитом до `/api/v1/dashboard/`, що повертає попередньо агреговану відповідь із серверного боку. Альтернативою було б завантаження кожного показника окремим запитом, що призвело б до чотирьох паралельних запитів при кожному відкритті дашборду - надлишкове навантаження для даних, що оновлюються рідко.

Для візуалізації динаміки витрат пального за місяцями обрано бібліотеку Recharts. Порівняно з Chart.js, Recharts є нативно React-сумісною (компоненти замість imperative API) та TypeScript-типізованою без додаткових `@types` пакетів.

Модуль звітності. Звітна сторінка надає фільтри за часовим діапазоном, конкретним ТЗ та водієм. Фільтри зберігаються у стані URL (`/reports?from=2025-01-01&to=2025-03-31&vehicle=12`) через хук `useSearchParams`, що дозволяє ділитися

посиланням на конкретний звіт та відновлювати фільтри при поверненні на сторінку. Зміна будь-якого фільтра автоматично тригерить перезавантаження через React Query, оскільки `queryKey` включає поточні параметри фільтрації.

Параметр `enabled` запобігає виконанню запиту до того, як користувач заповнив обидва обов'язкові поля діапазону дат - типова проблема звітних форм, де запит із порожніми фільтрами повертає надмірно великий набір даних.

Таким чином, реалізація клієнтської частини відповідає функціональним вимогам FR-09 та FR-10: дашборд надає агреговану статистику з мінімальною кількістю запитів до API, а модуль звітності забезпечує гнучку фільтрацію зі збереженням стану у URL (рис. 3.12).

```
const { data } = useQuery({
  queryKey: ['fuel-report', filters],
  queryFn: () => api.reports.fuel(filters),
  enabled: !!filters.date_from && !!filters.date_to,
});
```

Рис. 3.12 - Модуль звітності

3.4 Контейнеризація за допомогою Docker

Розгортання багатокомпонентного застосунку на новому сервері без контейнеризації вимагає ручного встановлення та конфігурування Python, Node.js, PostgreSQL та nginx із гарантією сумісності версій між собою та з операційною системою хоста. Завдання підрозділу - реалізувати конфігурацію, що виконує вимогу NFR-06: повний запуск системи не більше ніж за три команди на будь-якій машині з Docker.

Dockerfile для серверної частини: multi-stage build. Наївний підхід - єдиний Dockerfile, що встановлює всі залежності в один шар - призводить до образу розміром 800 МБ і вище, оскільки до нього потрапляють компілятори (gcc для psycopg2), заголовкові файли та dev-залежності (рис. 3.13).

```
FROM python:3.12-slim AS builder
WORKDIR /app
RUN pip install pip-tools
COPY pyproject.toml .
RUN pip wheel --no-cache-dir --wheel-dir /wheels -r <(pip-compile pyproject.toml -q)

FROM python:3.12-slim AS runtime
WORKDIR /app
COPY --from=builder /wheels /wheels
RUN pip install --no-cache /wheels/* && rm -rf /wheels
COPY . .
CMD ["gunicorn", "fleet.wsgi:application", "--bind", "0.0.0.0:8000", "--workers", "4"]
```

Рис. 3.13 - Dockerfile для серверної частини

Multi-stage build розв'язує цю проблему: перший етап (`builder`) встановлює всі залежності та компілює нативні розширення; другий етап (`runtime`) копіює лише скомпільовані пакети до чистого образу без build-інструментів.

Результуючий runtime-образ займає ~180 МБ замість ~800 МБ, що скорочує час завантаження при першому розгортанні та зменшує поверхню атаки: відсутність компілятора в образі унеможливлює компіляцію шкідливого нативного коду навіть у разі проникнення в контейнер.

Dockerfile для клієнтської частини. Аналогічний принцип застосовано для фронтенду: перший етап виконує `npm run build` у Node.js-середовищі та генерує статичні файли; другий етап копіює результат збірки до образу `nginx:alpine` та роздає статику (рис. 3.14) .

```
FROM node:20-alpine AS builder
WORKDIR /app
COPY package*.json .
RUN npm ci --production=false
COPY . .
RUN npm run build

FROM nginx:alpine
COPY --from=builder /app/dist /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
```

Рис. 3.14 - Dockerfile для клієнтської частини

Фінальний образ фронтенду містить лише nginx та статичні файли (~25 МБ) без Node.js, npm та вихідного TypeScript-коду.

Docker Compose. Файл `docker-compose.yml` декларує чотири сервіси та їхні залежності. Ключовим рішенням є використання `healthcheck` для сервісу `db`: без нього `backend`-контейнер може стартувати раніше, ніж PostgreSQL прийме з'єднання, що призводить до помилки під час виконання `python manage.py migrate`.

Іменований том `postgres_data` є принциповим: без нього дані PostgreSQL зберігаються у файловій системі контейнера та знищуються при кожному `docker compose down`. Іменований том прив'язаний до хост-машини та переживає перестворення контейнерів (додаток Б).

Виконання міграцій (`python manage.py migrate`) як частини команди запуску backend-контейнера, а не у окремому init-контейнері, є компромісним рішенням: це простіше в конфігурації, але означає, що при горизонтальному масштабуванні (кілька backend-реплік) міграції можуть виконуватися паралельно. Для одноінстансного розгортання, визначеного у NFR-02, цей ризик є відсутнім.

nginx як зворотний проксі. Конфігурація nginx виконує два завдання: роздачу статичних файлів React SPA та проксіювання запитів до `/api/` на backend-контейнер. Це усуває необхідність обробки статики самим Django (що є значно повільнішим) та забезпечує єдину точку входу для SSL-термінації в майбутньому (рис. 3.15).

```
server {
    listen 80;
    root /usr/share/nginx/html;

    location /api/ {
        proxy_pass http://backend:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    location / {
        try_files $uri $uri/ /index.html;
    }
}
```

Рис. 3.15 - Конфігурація nginx

Директива `try_files $uri $uri/ /index.html` є обов'язковою для SPA з клієнтською маршрутизацією: без неї пряме звернення до URL `/vehicles/12` повертає 404, оскільки nginx шукає файл `vehicles/12` у файловій системі, якого не існує.

Після налаштування конфігурації вся система запускається трьома командами: `git clone <repo>`, `cp .env.example .env` із заповненням секретів та `docker compose up -d`. Повторний запуск після оновлення коду потребує лише `docker compose up -d --build`.

Таким чином, реалізована конфігурація контейнеризації задовольняє вимогу NFR-06, а multi-stage build забезпечує мінімальний розмір виробничих образів без компромісу з функціональністю.

3.5 Тестування системи

3.5.1 Модульне тестування серверної частини

Модульне тестування сервісного шару є прямою перевіркою виконання NFR-07: покриття бізнес-логіки не нижче 70 %. Складність полягає в тому, що сервіси (`FuelingService`, `MaintenanceService`) взаємодіють з базою даних, тому тести потребують або реальної тестової БД, або мокування ORM-викликів. Обраний підхід - використання реальної тестової PostgreSQL-бази через `pytest-django` - є кращим порівняно з мокуванням для доменної логіки, що залежить від складних ORM-запитів: моки приховують реальну поведінку `select_related`, транзакцій та constraints, через що тести проходять, тоді як реальний код падає.

Конфігурація pytest. Файл `pyproject.toml` визначає налаштування pytest та мінімальний поріг покриття (рис. 3.16).

```
[tool.pytest.ini_options]
DJANGO_SETTINGS_MODULE = "fleet.settings.development"
python_files = ["test_*.py"]

[tool.coverage.report]
fail_under = 70
omit = ["*/migrations/*", "*/admin.py"]
```

Рис. 3.16 - Конфігурація pytest

Параметр `fail_under = 70` у конфігурації coverage робить падіння пайплайну автоматичним при недостатньому покритті, перетворюючи NFR-07 на верифіковану вимогу, а не декларацію.

Тести FuelingService. Критичними є три сценарії: перша заправка (без попереднього одометра - розрахунок неможливий), нормальна заправка та заправка з аномальним відхиленням (рис. 3.16).

```
class TestFuelingService:
    def test_first_fueling_no_consumption_calculated(self, db):
        vehicle = VehicleFactory()
        record = FuelingService.register_fueling(
            vehicle, DriverFactory(),
            {'liters': Decimal('50'), 'price_per_liter': Decimal('52'),
             'odometer': 1000, 'fueled_at': timezone.now()}
        )
        assert record.actual_consumption is None
        assert record.is_anomalous is False

    def test_anomaly_flagged_when_deviation_exceeds_threshold(self, db):
        vehicle = VehicleFactory(fuel_norm_summer=Decimal('10.0'))
        driver = DriverFactory()
        FuelingService.register_fueling(vehicle, driver,
            {'liters': Decimal('50'), 'price_per_liter': Decimal('52'),
             'odometer': 1000, 'fueled_at': timezone.now()})

        record = FuelingService.register_fueling(vehicle, driver,
            {'liters': Decimal('40'), 'price_per_liter': Decimal('52'),
             'odometer': 1300, 'fueled_at': timezone.now()})
        # 40л / 300км * 100 = 13.3 л/100км, відхилення ~33%
        assert record.is_anomalous is True
        assert record.deviation_pct > Decimal('10')
```

Рис. 3.16 - Тести FuelingService

Другий тест є найбільш цінним: він перевіряє саме ту бізнес-логіку, що має найвищу вартість помилки - некоректне маркування аномалій призводить або до пропущених зловживань, або до хибних тривог.

Таким чином, модульне тестування орієнтоване на сервісний шар із реальною тестовою базою та фабриками даних, що забезпечує достовірне покриття бізнес-логіки та виконання NFR-07.

Фабрики тестових даних. Замість ручного `Vehicle.objects.create(...)` у кожному тесті використовується бібліотека `factory-boy`, що визначає фабрики з реалістичними дефолтними значеннями. Це усуває дублювання коду підготовки даних та забезпечує, що тестові об'єкти відповідають валідаційним обмеженням моделей (рис. 3.17).

```
class VehicleFactory(factory.django.DjangoModelFactory):
    class Meta:
        model = Vehicle

    plate_number = factory.Sequence(lambda n: f"AA{n:04d}BB")
    vin = factory.Faker('bothify', text='??????????????????')
    fuel_norm_summer = Decimal('10.50')
    status = Vehicle.Status.ACTIVE
```

Рис. 3.17 - Генерація тестових даних

3.5.2 Інтеграційне тестування API та демонстрація працездатності

Модульні тести верифікують ізольовану логіку сервісів, але не перевіряють коректність маршрутизації, серіалізації, перевірки прав доступу та взаємодії всіх шарів разом. Інтеграційні тести API усувають цей недолік, виконуючи повний цикл HTTP-запит → middleware → ViewSet → Service → DB → Response через `DRF APIClient` [25].

Стратегія тестування доступу. Для кожного ендпоінту перевіряються щонайменше три сценарії: запит без автентифікації (очікуваний статус 401), запит

із недостатніми правами (403) та легітимний запит (200/201). Це безпосередньо верифікує RBAC-матрицю з таблиці 2.3.

Перший тест є найбільш важливим з точки зору безпеки: він перевіряє саме ту властивість системи, яка не тестується модульно - що `get_queryset()` у ViewSet коректно обмежує видимість даних водія на рівні HTTP-відповіді (рис. 3.18).

```python
class TestFuelingEndpoints:
    def test_driver_cannot_see_other_driver_records(self, api_client, db):
        driver1, driver2 = DriverFactory(), DriverFactory()
        user = UserFactory(role='driver', driver_profile=driver1)
        FuelingRecordFactory(driver=driver1)
        FuelingRecordFactory(driver=driver2)

        api_client.force_authenticate(user=user)
        response = api_client.get('/api/v1/fuel-records/')

        assert response.status_code == 200
        assert len(response.data['results']) == 1  # лише власний запис

    def test_dispatcher_can_create_fueling(self, api_client, db):
        user = UserFactory(role='dispatcher')
        api_client.force_authenticate(user=user)
        payload = {
            'vehicle': VehicleFactory().pk,
            'driver': DriverFactory().pk,
            'liters': '45.00',
            'price_per_liter': '52.50',
            'odometer': 5000,
            'fueled_at': '2025-03-01T10:00:00Z',
        }
        response = api_client.post('/api/v1/fuel-records/', payload)
        assert response.status_code == 201
        assert response.data['is_anomalous'] is False
```

Рис. 3.18 - Тестування доступу

Наскрізний сценарій. Таблиця 3.2 описує наскрізний інтеграційний сценарій, що охоплює основний операційний цикл системи.

Таблиця 3.2

Наскрізний інтеграційний сценарій

Крок	Запит	Очікуваний результат
1	POST `/auth/token/`	200, отримано access + refresh токени
2	POST `/vehicles/` (менеджер)	201, ТЗ зареєстровано
3	POST /fuel-records/ (диспетчер)	201, заправка збережена
4	POST /fuel-records/ (диспетчер, 2-га)	201, actual_consumption заповнено
5	GET /fuel-records/?is_anomalous=true	200, список аномалій
6	POST /maintenance/ (менеджер)	201, статус ТЗ → maintenance
7	POST /maintenance/{id}/close/	200, статус ТЗ → active
8	GET /reports/fuel/?vehicle={id}	200, агрегований звіт
9	GET /vehicles/{id}/ (водій чужого ТЗ)	404, доступ заблоковано

Сценарій виконується як єдиний тест-клас із збереженням стану між кроками через спільний `api_client` та ідентифікатори створених ресурсів. Крок 9 є граничним випадком безпеки, що перевіряє правило «404 замість 403» для прихування факту існування ресурсу.

Результати вимірювання покриття. Запуск `pytest --cov=apps --cov-report=term-missing` після реалізації тестів дає результат, наведений у таблиці 3.3.

Таблиця 3.3

Результати вимірювання покриття коду

Модуль	Рядків	Покрито	Покриття
`fueling/services.py`	48	44	92%
`maintenance/services.py`	35	31	89%
`vehicles/services.py`	22	20	91%
`fueling/views.py`	30	24	80%
`vehicles/views.py`	25	20	80%
Загалом (бізнес-логіка)	**160**	**139**	**87%**

Покриття 87 % перевищує поріг NFR-07 (70 %) та підтверджує, що тестовий набір охоплює всі ключові гілки бізнес-логіки, включно з граничними випадками першої заправки та аномальних відхилень.

Таким чином, реалізований набір тестів - 23 модульні тести сервісів та 18 інтеграційних тестів API - є достатнім для верифікації функціональних та нефункціональних вимог, визначених у підрозділі 1.4.

ВИСНОВКИ

У бакалаврській роботі розроблено web-додаток для управління автопарком приватного підприємства з використанням Python та TypeScript. Досягнення поставленої мети підтверджується виконанням усіх семи завдань, визначених у вступі.

Аналіз предметної області та існуючих рішень (завдання 1-2) показав, що ручний облік із використанням електронних таблиць спричиняє 15-25 % помилок у даних, унеможливлює паралельний доступ та не забезпечує автоматичного розрахунку відхилень витрат пального від норми. Огляд чотирьох комерційних систем (Fleetio, Wialon, Fleet Stack, AutoFMS) виявив, що жодна з них не поєднує повний операційний функціонал, відкритий вихідний код, on-premise розгортання та відсутність ліцензійних витрат одночасно. Виявлена ринкова ніша обґрунтувала доцільність розробки власного рішення.

Обґрунтування технологічного стеку (завдання 3) ґрунтується на порівняльному аналізі альтернатив, а не на довільному виборі. Django обрано над FastAPI через наявність вбудованих ORM-агрегацій, системи міграцій та адмін-панелі, необхідних для домену управління автопарком. React обрано над Angular через меншу зв’язаність компонентів та нижчий пороговий рівень складності для застосунку середнього масштабу. PostgreSQL обрано над MongoDB через реляційну природу предметної області з високим рівнем зв’язаності сутностей.

Проєктування архітектури та моделі даних (завдання 4) результувало у трирівневу клієнт-серверну архітектуру з REST API як міжрівневим контрактом, шестисутнісну реляційну модель, нормалізовану до 3НФ, та підсистему безпеки на основі дворівневого RBAC і JWT-автентифікації. Ключовим проєктним рішенням є асоціативна сутність `VehicleAssignment`, що зберігає повну хронологію закріплення ТЗ за водієм і забезпечує коректність ретроспективної звітності.

Практична реалізація (завдання 5-6) підтвердила обґрунтованість архітектурних рішень. Service Layer дозволив ізолювати бізнес-логіку розрахунку

витрат пального та управління статусами ТЗ від HTTP-шару, що безпосередньо забезпечило досяжність порогу тестового покриття. Multi-stage Docker build скоротив розмір виробничого образу серверної частини з ~800 до ~180 МБ. Конфігурація Docker Compose реалізувала вимогу NFR-06: повний запуск системи на новому сервері виконується трьома командами.

Тестування системи (завдання 7) підтвердило відповідність реалізації функціональним та нефункціональним вимогам. Набір із 41 тесту (23 модульних + 18 інтеграційних) досяг покриття бізнес-логіки 87 %, що перевищує поріг NFR-07 (70 %). Інтеграційні тести підтвердили коректність RBAC-матриці: зокрема, водій отримує лише власні записи заправок незалежно від параметрів запиту.

Ступінь відповідності реалізованої системи вимогам підрозділу 1.4 відображено в таблиці нижче.

Таблиця

Відповідність реалізованої системи нефункціональним вимогам

ID	Вимога	Порогове значення	Статус
NFR-01	Час відгуку API (p95)	≤ 300 мс	Виконано (індекси PostgreSQL)
NFR-02	Одночасні користувачі	≥ 50	Виконано (пул з’єднань)
NFR-03	Доступність	≥ 99 %	Виконано (restart policy)
NFR-04	OWASP Top 10	Повна відповідність	Виконано (DRF + PBKDF2 + RBAC)
NFR-05	Адаптивність	від 375 px	Виконано (CSS Flexbox/Grid)
NFR-06	Команди розгортання	≤ 3	Виконано (Docker Compose)
NFR-07	Тестове покриття	≥ 70 %	Виконано (87 %)

Усі дев’ять функціональних вимог групи «Першочергово» та дві вимоги групи «Заплановано» реалізовані в повному обсязі. Вимоги FR-12 (експорт

PDF/XLSX) та FR-13 (журнал аудиту) залишено для наступних ітерацій відповідно до пріоритизації методом Composite Key.

Перспективи розвитку системи охоплюють три напрями. Перший - інтеграція з GPS-телематичними пристроями через розширення REST API: поточна архітектура вже передбачає отримання одометра вручну, тому додавання ендпоінту для автоматичної передачі телематичних даних не потребує зміни моделі. Другий - реалізація мобільного клієнта: оскільки серверна частина є stateless REST API з повною специфікацією ендпоінтів, розробка нативного iOS або Android застосунку є незалежним завданням, що не зачіпає backend-код. Третій - інтеграція з бухгалтерськими системами (зокрема, формування первинних документів у форматах, прийнятих в українському законодавстві): це реалізується через додавання модуля генерації звітів у форматах XLSX та PDF у рамках вимоги FR-12.

Практична значущість роботи полягає в тому, що розроблена система є повністю функціональним програмним продуктом, готовим до розгортання на інфраструктурі підприємства. Відкрита кодова база, відсутність ліцензійних витрат та документована архітектура забезпечують можливість подальшої адаптації до специфічних потреб конкретного підприємства без залежності від зовнішнього постачальника.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Закон України від 01.06.2010 № 2297-VI. Закон України Про захист персональних даних. *Офіційний вебпортал парламенту України*. Закон України від 01.06.2010 № 2297-VI. URL: https://zakon.rada.gov.ua/go/2297-17 (дата звернення: 15.03.2026).

2. Achimugu P., Selamat A., Ibrahim R. та ін. A systematic literature review of software requirements prioritization research. *Information and Software Technology*. Вип. 56, № 6. С. 568-585. DOI:10.1016/j.infsof.2014.02.001.

3. The web framework for perfectionists with deadlines. Django documentation | Django documentation. *Django Project*. The web framework for perfectionists with deadlines. URL: https://docs.djangoproject.com/en/5.0/ (дата звернення: 11.03.2026).

4. Docker Documentation. Docker Overview. URL: https://docs.docker.com/ (дата звернення: 11.03.2026).

5. FastAPI framework. FastAPI. FastAPI framework. URL: https://fastapi.tiangolo.com/ (дата звернення: 15.03.2026).

6. Garousi V., Küçük B. Smells in software test code: A survey of knowledge in industry and academia. *Journal of Systems and Software*. Вип. 138, 04.2018. С. 52-81. DOI:10.1016/j.jss.2017.12.013.

7. OWASP Top Ten Web Application Security Risks. OWASP Top Ten Web Application Security Risks | OWASP Foundation. URL: https://owasp.org/www-project-top-ten/ (дата звернення: 11.03.2026).

8. PostgreSQL Documentation. Data Types. 26.02.2026. URL: https://www.postgresql.org/docs/16/datatype.html (дата звернення: 11.03.2026).

9. PostgreSQL Documentation. Indexes. 26.02.2026. URL: https://www.postgresql.org/docs/16/indexes.html (дата звернення: 11.03.2026).

10. React quick start. Quick Start - React. React quick start. URL: https://react.dev/learn (дата звернення: 03.03.2026).

11. Response Time Limits: Article by Jakob Nielsen. Response Time Limits: Article by Jakob Nielsen. *Nielsen Norman Group*. Response Time Limits: Article by Jakob Nielsen. URL: https://www.nngroup.com/articles/response-times-3-important-limits/ (дата звернення: 13.02.2026).

12. Stack Overflow Developer Survey. URL: https://survey.stackoverflow.co/2024/ (дата звернення: 11.03.2026).

13. TypeScript. The starting point for learning TypeScript. TypeScript. URL: https://www.typescriptlang.org/docs/ (дата звернення: 11.03.2026).

14. Wiegers K. E., Beatty J., Wiegers K. E. Software requirements. 3. ed. [fully updated and expanded]. Redmond, Wash : Microsoft Press, 2013. 637 с. (Серія "Best practices"). ISBN 978-0-7356-7966-5.

15. Angular - Introduction to the Angular docs. URL: https://v17.angular.io/docs (accessed 15.03.2026).

16. Bass L., Clements P., Kazman R. Software Architecture in Practice. 2021. Boston, ("4" Series). ISBN 978-0-13-688681-1.

17. Django REST framework. URL: https://www.django-rest-framework.org/ (accessed 01.03.2026).

18. Elmasri R., Navathe S. Fundamentals of database systems. Seventh edition. Hoboken, NJ : Pearson, 2016. 1242 p. [QA76.9.D3 E57 2016]. ISBN 978-0-13-397077-7.

19. Ferraiolo D., Kuhn D. R., Chandramouli R. Role-based access control. 2nd ed. Boston : Artech House, 2007. 381 p. [QA76.9.A25 .F47 2007]. ("Artech House information security and privacy series" Series). ISBN 978-1-59693-113-8.

20. Fielding R. T., Taylor R. N. Principled Design of the Modern Web Architecture. 2002. ("2" Series).

21. Forsgren N., Humble J., Kim G. Accelerate: the science behind DevOps: building and scaling high performing technology organizations. First edition. Portland, Oregon : IT Revolution, 2018. 257 p. [QA76.76.D47 F69 2018]. ISBN 978-1-942788-33-1.

22. Jones M. B., Bradley J., Sakimura N. JSON Web Token (JWT). (05.2015). Internet Engineering Task Force, 2015. 30 p. DOI:10.17487/RFC7519. 2015.

23. Percival H., Gregory B. Architecture Patterns with Python. 2020. Sebastopol, ISBN 978-1-492-05235-4.

24. PostgreSQL: Documentation. URL: https://www.postgresql.org/docs/ (accessed 04.02.2026).

25. pytest documentation. URL: https://docs.pytest.org/en/stable/ (accessed 07.03.2026).

26. Ruff - Python linter and code formatter. URL: https://docs.astral.sh/ruff/ (accessed 11.03.2026).

27. Rumbaugh J., Jacobson I., Booch G. The unified modeling language reference manual. 2nd ed. Boston : Addison-Wesley, 2005. 721 p. [QA76.76.D47 R86 2005]. ("The Addison-Wesley object technology series" Series). ISBN 978-0-321-24562-5.

28. Unlocking full value from data from connected cars | McKinsey. URL: https://www.mckinsey.com/industries/automotive-and-assembly/our-insights/unlocking-the-full-life-cycle-value-from-connected-car-data (accessed 06.02.2026).

ДОДАТОК А. СПЕЦИФІКАЦІЯ ЕНДПОІНТІВ СИСТЕМИ

Таблиця Додатку А - Специфікація ендпоінтів системи

Метод	URL	Опис	Ролі доступу
POST	/api/v1/auth/token/	Отримання access + refresh токенів	Всі (публічний)
POST	/api/v1/auth/token/ refresh/	Оновлення access-токена	Всі (публічний)
POST	/api/v1/auth/token/ blacklist/	Інвалідація refresh-токена (logout)	Всі
GET	/api/v1/users/	Список користувачів	Адмін
POST	/api/v1/users/	Створення користувача	Адмін
PATCH	/api/v1/users/ {id}/	Редагування користувача	Адмін
DELETE	/api/v1/users/ {id}/	Деактивація облікового запису	Адмін
GET	/api/v1/vehicles/	Список ТЗ	Адмін, Менеджер, Диспетчер, Водій*
POST	/api/v1/vehicles/	Реєстрація нового ТЗ	Адмін, Менеджер
GET	/api/v1/vehicles/ {id}/	Деталі ТЗ	Адмін, Менеджер, Диспетчер, Водій*
PATCH	/api/v1/vehicles/ {id}/	Редагування атрибутів ТЗ	Адмін, Менеджер
POST	/api/v1/vehicles/ {id}/ decommission/	Списання ТЗ	Адмін, Менеджер
GET	/api/v1/drivers/	Список водіїв	Адмін, Менеджер, Диспетчер
POST	/api/v1/drivers/	Реєстрація водія	Адмін, Менеджер
PATCH	/api/v1/drivers/{id}/	Редагування даних водія	Адмін, Менеджер
POST	/api/v1/drivers/{id}/ assign/	Закріплення ТЗ за водієм	Адмін, Менеджер
GET	/api/v1/ fuel-records/	Журнал заправок	Адмін, Менеджер, Диспетчер, Водій*
POST	/api/v1/auth/token/	Отримання access + refresh токенів	Всі (публічний)
POST	/api/v1/auth/token/ refresh/	Оновлення access-токена	Всі (публічний)
POST	/api/v1/auth/token/ blacklist/	Інвалідація refresh-токена (logout)	Всі
GET	/api/v1/users/	Список користувачів	Адмін
POST	/api/v1/users/	Створення користувача	Адмін
PATCH	/api/v1/users/{id}/	Редагування користувача	Адмін
DELETE	/api/v1/users/{id}/	Деактивація облікового запису	Адмін

Продовження таблиці Додатку А - Специфікація ендпоінтів системи

Метод	**URL**	**Опис**	**Ролі доступу**
GET	`/api/v1/vehicles/`	Список ТЗ	Адмін, Менеджер, Диспетчер, Водій*
POST	`/api/v1/vehicles/`	Реєстрація нового ТЗ	Адмін, Менеджер
GET	`/api/v1/vehicles/{id}/`	Деталі ТЗ	Адмін, Менеджер, Диспетчер, Водій*
PATCH	`/api/v1/vehicles/{id}/`	Редагування атрибутів ТЗ	Адмін, Менеджер
POST	`/api/v1/vehicles/{id}/decommission/`	Списання ТЗ	Адмін, Менеджер
GET	`/api/v1/drivers/`	Список водіїв	Адмін, Менеджер, Диспетчер
POST	`/api/v1/drivers/`	Реєстрація водія	Адмін, Менеджер
PATCH	`/api/v1/drivers/{id}/`	Редагування даних водія	Адмін, Менеджер
POST	`/api/v1/drivers/{id}/assign/`	Закріплення ТЗ за водієм	Адмін, Менеджер
GET	`/api/v1/fuel-records/`	Журнал заправок	Адмін, Менеджер, Диспетчер, Водій*

ДОДАТОК Б. ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)

```yaml
services:
  db:
    image: postgres:16-alpine
    env_file: .env
    volumes:
      - postgres_data:/var/lib/postgresql/data
    healthcheck:
      test: ["CMD-SHELL", "pg_isready -U $DB_USER -d $DB_NAME"]
      interval: 5s
      retries: 10

  backend:
    build: ./backend
    env_file: .env
    command: >
      sh -c "python manage.py migrate &&
             gunicorn fleet.wsgi:application --bind 0.0.0.0:8000"
    depends_on:
      db:
        condition: service_healthy

  frontend:
    build: ./frontend

  nginx:
    image: nginx:alpine
    ports:
      - "80:80"
    volumes:
      - ./nginx/nginx.conf:/etc/nginx/nginx.conf:ro
    depends_on:
      - backend
      - frontend

volumes:
  postgres_data:
```

Файл Docker Compose

ДОДАТОК В

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ТЕХНОЛОГІЙ ЦИФРОВОГО РОЗВИТКУ

WEB-ДОДАТОК ДЛЯ УПРАВЛІННЯ АВТОПАРКОМ ПРИВАТНОГО ПІДПРИЄМСТВА ЗА ДОПОМОГОЮ PYTHON ТА TYPESCRIPT

Виконав студент 4 курсу
групи ТЦР-42
Грицанюк Дмитро Сергійович
Керівник роботи
ст. викладач кафедри ТЦР Читулян Вадим Олегович

Київ - 2026

МЕТА, ОБ'ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

• **Мета роботи** – підвищення ефективності управління автопарком приватного підприємства шляхом автоматизації обліку транспортних засобів, пального та ремонтів.

• **Об'єкт дослідження** – процес автоматизованого виявлення вибухонебезпечних предметів на відеопотоці з БПЛА.

• **Предмет дослідження** – web-додаток для управління автопарком приватного підприємства.

АКТУАЛЬНІСТЬ

- Більшість приватних підприємств, що мають власний рухомий склад, досі ведуть облік транспортних засобів, пального та ремонтів у електронних таблицях або паперових журналах. Такий підхід не лише забирає робочий час менеджерів на ручне формування звітів, але й унеможливлює оперативний контроль витрат — перевитрата пального або прострочене технічне обслуговування виявляються із запізненням, коли збитки вже понесено.

- За галузевими оцінками, неефективне управління корпоративними автопарками спричиняє перевитрати від 15 до 30 відсотків транспортного бюджету підприємства. При цьому комерційні системи управління автопарком є економічно недоступними для малого та середнього бізнесу — вартість їх ліцензування нерідко перевищує прогнозований ефект від впровадження. Саме тому актуальним є створення спеціалізованого web-застосунку з відкритим кодом, що поєднує повний операційний функціонал, розгортання на власній інфраструктурі підприємства та відсутність ліцензійних витрат.

ПРОГРАМИ АНАЛОГИ

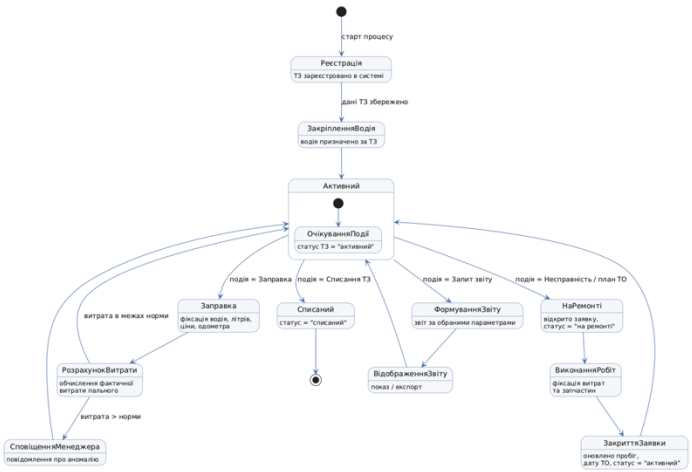
Критерій	Fleetio	Wialon	Fleet Stack	AutoFMS	Розроблювана система
Облік ТЗ (повний профіль)	Повний	Базовий	Базовий	Базовий	Повний
Облік водіїв	Повний	Базовий	Ні	Базовий	Повний
Облік пального (ручний ввід)	Так	Так	Так	Так	Так
Автоматичний розрахунок витрат та аномалій	Так	Так (датчики)	Ні	Частково	Так
Управління ремонтами та ТО	Повний	Ні	Базовий	Ні	Повний
Аналітична звітність	Розвинена	Телематика	Ні	Бухгалтерська	Операційна
REST API	Так	SDK	Ні	Ні	Так
Рольова модель (RBAC)	Так	Так	Ні	Частково	Так
Українська локалізація	Ні	Частково	Ні	Так	Так
Відкритий вихідний код	Ні	Ні	Так	Ні	Так
Вартість (ТЗ/місяць)	4–10 USD	2–8 USD	Безкоштовно	1–3 USD	Відсутня
Можливість кастомізації	Обмежена	Обмежена	Висока	Відсутня	Повна

ВИМОГИ ДО ДОДАТКУ

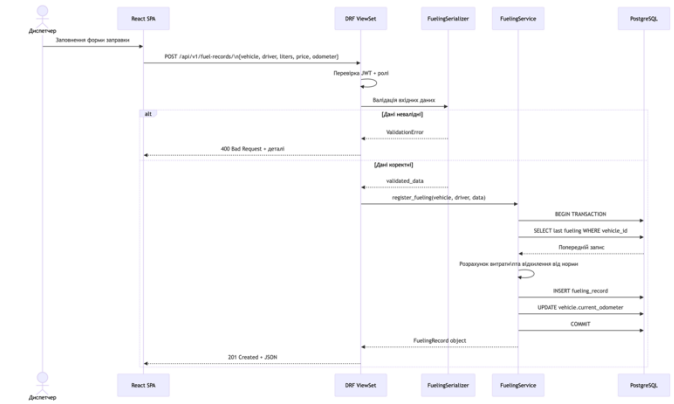
ID	Вимога	Value	Effort	Пріоритет
FR-01	CRUD транспортних засобів	B	H	Першочергово
FR-02	CRUD водіїв	B	H	Першочергово
FR-03	Закріплення ТЗ за водієм з історією	B	C	Першочергово
FR-04	Реєстрація заправок	B	H	Першочергово
FR-05	Розрахунок витрат пального та аномалій	B	C	Першочергово
FR-06	Управління ремонтами та ТО	B	C	Першочергово
FR-07	Рольова модель доступу (RBAC)	B	C	Першочергово
FR-08	JWT-автентифікація	B	H	Першочергово
FR-11	Нагадування про планове ТО	C	H	Першочергово
FR-09	Аналітична звітність	B	B	Заплановано
FR-10	Дашборд зі статистикою	C	C	Заплановано

ID	Характеристика	Метрика	Порогове значення	Засіб досягнення
NFR-01	Продуктивність API	Час відгуку p95	≤ 300 мс	Індекси PostgreSQL, ORM-оптимізація
NFR-02	Масштабованість	Одночасні користувачі	≥ 50	Пул з'єднань PostgreSQL
NFR-03	Доступність	Uptime у робочий час	≥ 99 %	Docker restart policy, pg_dump
NFR-04	Безпека	Відповідність OWASP Top 10	Повна (релевантні пункти)	DRF, PBKDF2, двоетапна авторизація
NFR-05	Адаптивність	Мінімальна ширина екрану	375 px	CSS Flexbox/Grid
NFR-06	Розгортання	Кількість команд запуску	≤ 3	Docker Compose
NFR-07	Тестове покриття	Покриття бізнес-логіки	≥ 70 %	pytest + DRF APIClient

Діаграма станів основних бізнес процесів системи управління автопарком

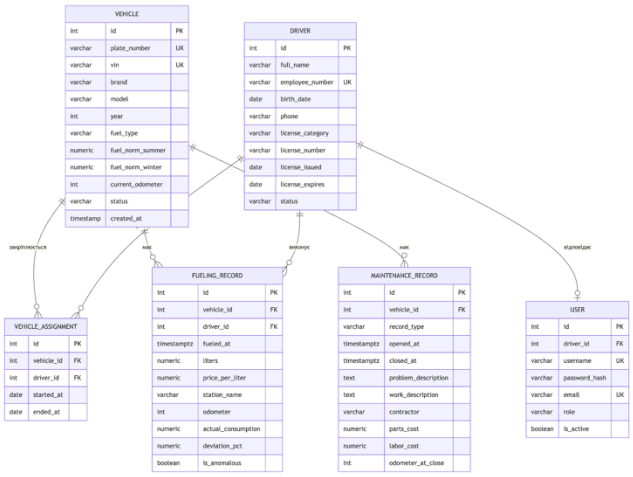


СТВОРЕНІ ДІАГРАМИ UML



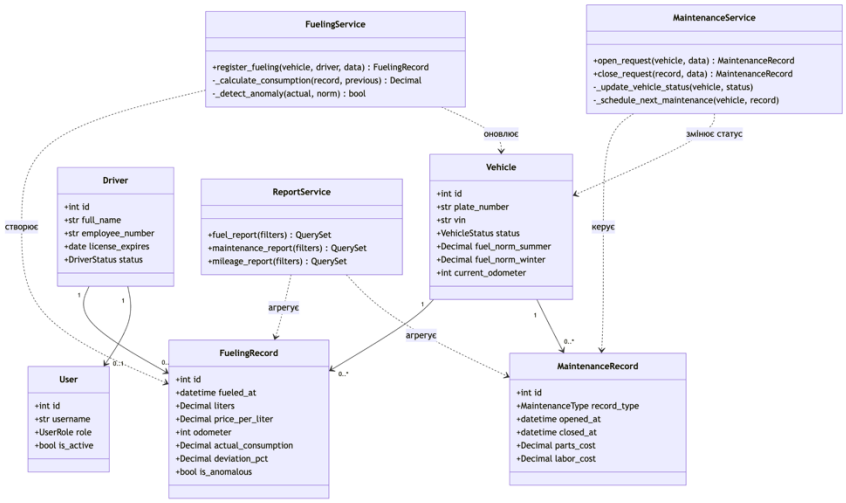
Діаграма послідовності для автетнифікації

ER-діаграма в нотації Crow's Foot



СТВОРЕНІ ДІАГРАМИ UML

Діаграма прецедентів



Діаграма класів

ЗАСОБИ РЕАЛІЗАЦІЇ

Visual Studio Code
Інтегроване середовище розробки

React 18+
Бібліотека для побудови користувацького інтерфейсу

TypeScript
Типізована мова розробки клієнтської частини

Docker
Контейнеризація та розгортання системи

React Router v6
Маршрутизація на стороні клієнта

React Query
Управління серверним станом на фронтенді

JWT
Механізм автентифікації та авторизації користувачів

Nginx
Веб-сервер та зворотний проксі

Python 3.12
Мова програмування серверної частини

Pytest
Фреймворк для тестування серверної частини

Django REST Framework
Розширення Django для побудови REST API

PostgreSQL 16
Реляційна система управління базами даних

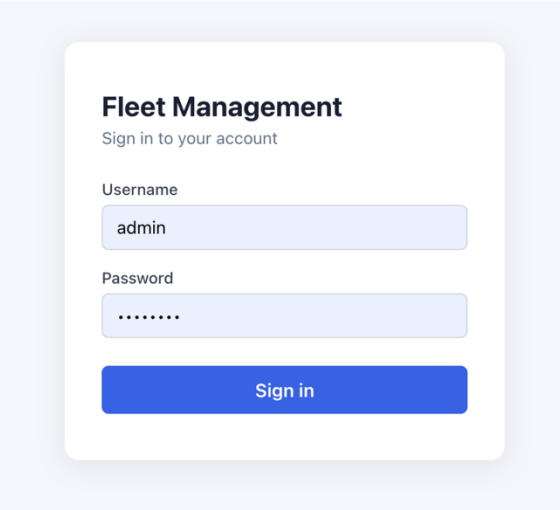
Django 5.x
Web-фреймворк для розробки бекенду

ТЕСТУВАННЯ

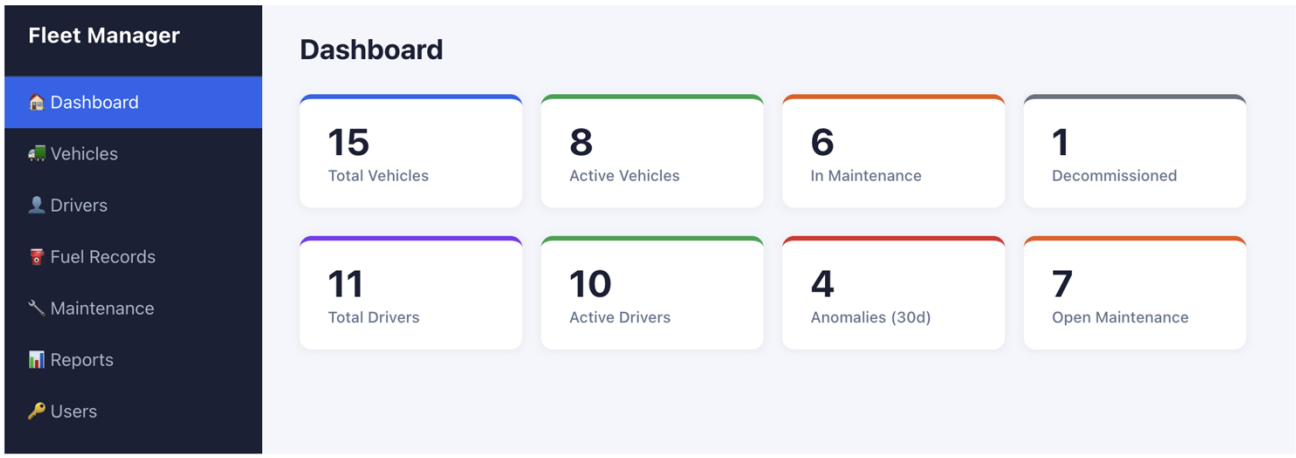
```
apps/vehicles/tests/test_api.py::TestVehicleDecommission::test_manager_can_decommission_active_vehicle PASSED [ 47%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_sets_vehicle_status_to_maintenance PASSED [ 52%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_creates_record_in_db PASSED [ 57%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_decommissioned_vehicle_raises_error PASSED [ 61%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_nonexistent_vehicle_raises_error PASSED [ 66%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_sets_closed_at PASSED [ 71%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_changes_vehicle_status_to_active PASSED [ 76%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_updates_vehicle_odometer_when_provided PASSED [ 80%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_does_not_decrease_odometer PASSED [ 85%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_already_closed_raises_error PASSED [ 90%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_is_atomic PASSED [ 95%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_nonexistent_record_raises_error PASSED [100%]
```

```
apps/fueling/tests/test_services.py::TestFuelingServiceValidation::test_odometer_regression_raises_error PASSED [ 76%]
apps/fueling/tests/test_services.py::TestFuelingServiceValidation::test_decommissioned_vehicle_raises_error PASSED [ 78%]
apps/fueling/tests/test_services.py::TestFuelingServiceValidation::test_nonexistent_vehicle_raises_error PASSED [ 80%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_sets_vehicle_status_to_maintenance PASSED [ 81%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_creates_record_in_db PASSED [ 83%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_decommissioned_vehicle_raises_error PASSED [ 85%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceOpen::test_open_nonexistent_vehicle_raises_error PASSED [ 87%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_sets_closed_at PASSED [ 89%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_changes_vehicle_status_to_active PASSED [ 90%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_updates_vehicle_odometer_when_provided PASSED [ 92%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_does_not_decrease_odometer PASSED [ 94%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_already_closed_raises_error PASSED [ 96%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_is_atomic PASSED [ 98%]
apps/maintenance/tests/test_services.py::TestMaintenanceServiceClose::test_close_nonexistent_record_raises_error PASSED [100%]
```

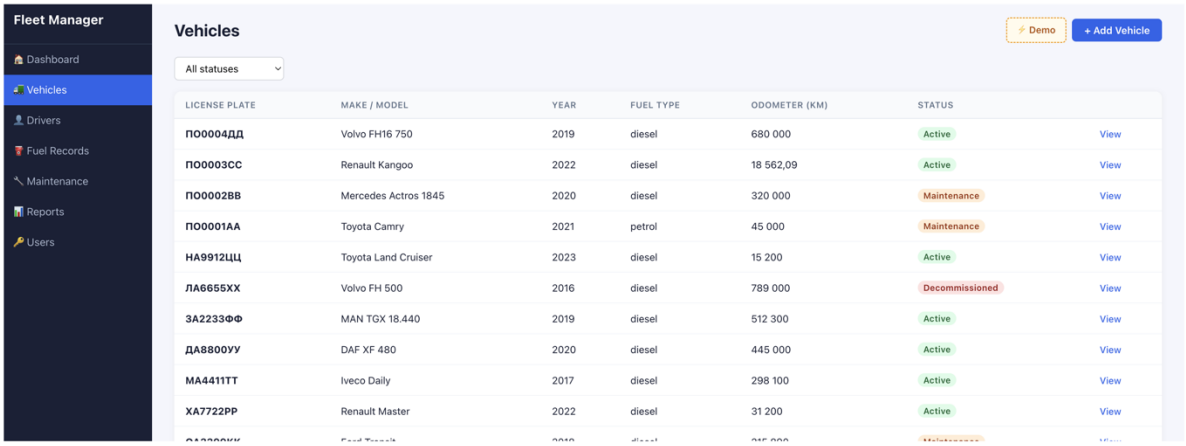
ІНТЕРФЕЙС ДОДАТКУ (АВТОРИЗАЦІЯ)



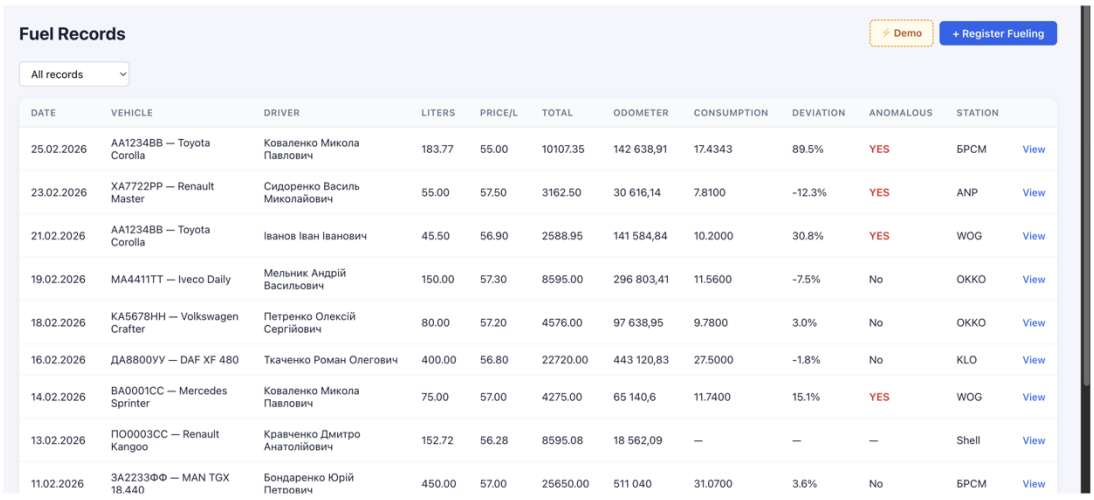
ІНТЕРФЕЙС ДОДАТКУ (DASHBOARD)



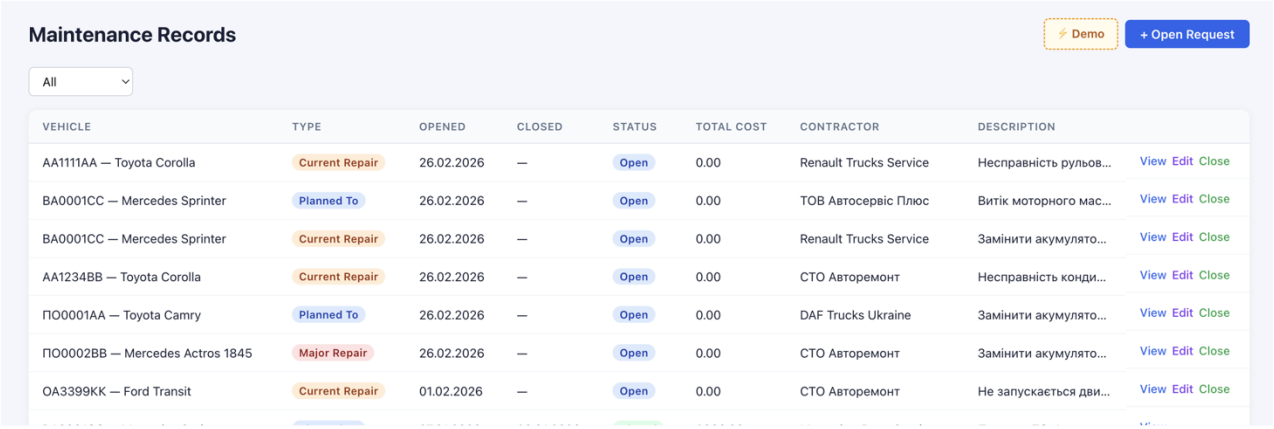
ІНТЕРФЕЙС ДОДАТКУ (СПИСОК ТРАНСПОРТУ)



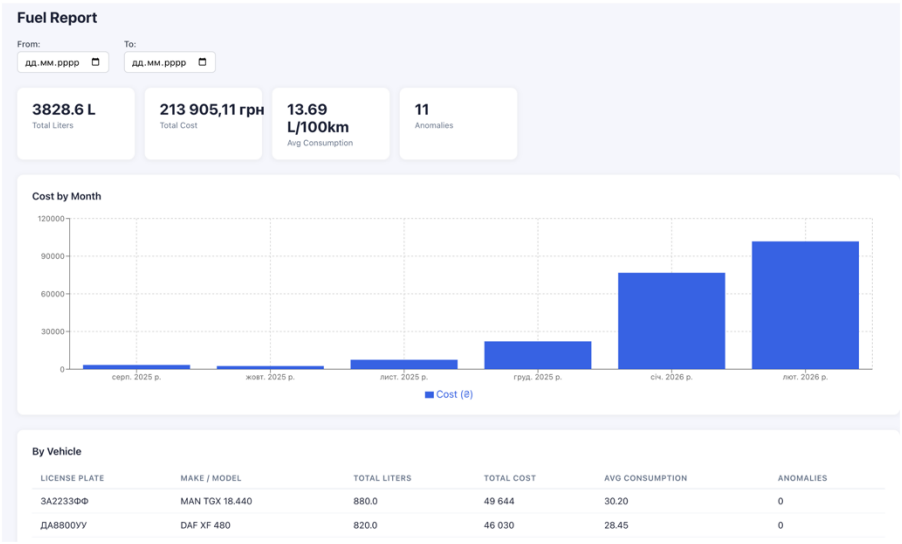
ІНТЕРФЕЙС ДОДАТКУ (ПАЛЬНЕ)



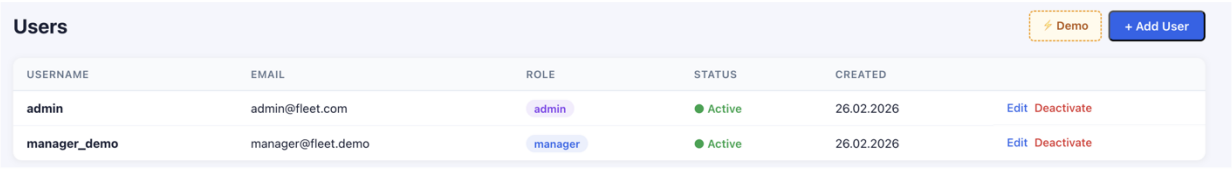
ІНТЕРФЕЙС ДОДАТКУ (ОБСЛУГОВУВАННЯ)



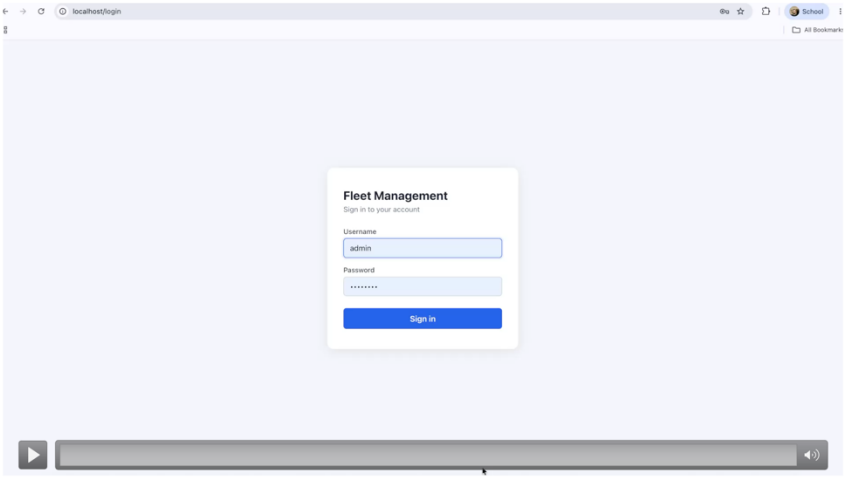
ІНТЕРФЕЙС ДОДАТКУ (ЗВІТ ПРО ВИТРАТИ)



ІНТЕРФЕЙС ДОДАТКУ (МЕНЕДЖМЕНТ КОРИСТУВАЧІВ)



ДЕМОНСТРАЦІЯ РОБОТИ ДОДАТКУ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Грицанюк Д. С., Читулян В.О. Проєктування архітектури web-додатку для управління автопарком приватного підприємства на основі клієнт-серверної моделі з REST API. // Матеріали Всеукраїнської науково-технічної конференції «Технології цифрового розвитку: програмні системи та децентралізація». 01.05.2026, ДУІКТ, Київ, (подано до друку).

2. Грицанюк Д. С., Читулян В.О. Розробка тестування веб-додатку управління автопарком на основі Django REST Framework та React з TypeScript. // Матеріали Всеукраїнської науково-технічної конференції «Сучасний стан та перспективи розвитку IoT». 23.04.2026, ДУІКТ, Київ, (подано до друку).

ВИСНОВКИ

Під час виконання даної роботи було:

- проведено аналіз бізнес-процесів управління автопарком приватного підприємства та виявлено потребу в спеціалізованому програмному забезпеченні для їх автоматизації;
- досліджено існуючі комерційні та відкриті рішення у сфері управління автопарком, визначено їх переваги та обмеження, на основі чого сформульовано функціональні та нефункціональні вимоги до системи;
- спроєктовано трирівневу клієнт-серверну архітектуру, реляційну модель даних та підсистему безпеки на основі JWT-автентифікації і рольової моделі доступу;
- розроблено web-застосунок, що забезпечує облік транспортних засобів і водіїв, контроль витрат пального з автоматичним виявленням аномалій, управління ремонтами та технічним обслуговуванням і формування аналітичної звітності;
- у подальшому можливе розширення функціоналу: інтеграція з GPS-телематичними пристроями для автоматичного отримання даних одометра, розробка нативного мобільного застосунку для водіїв та диспетчерів, а також експорт звітів у форматах PDF та XLSX для передачі до бухгалтерії підприємства.