НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ

Завідувач кафедри
комп'ютерних наук
_____ Голуб Б. Л.

«___»_____2025 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«СИСТЕМА МОНІТОРИНГУ ПРОЦЕСАМИ У ПРИВАТНОМУ АВТОПАРКУ»

Спеціальність 122 «Комп'ютерні науки»

Гарант освітньої програми
Д. е. н., професор _____ Руденський Р. А.

Керівник бакалаврської кваліфікаційної роботи
_____ Боровик В. І.
(підпис) (ПІБ)

Виконав
_____ Недобиткін М. В.
(підпис) (ПІБ студента)

КИЇВ – 2025

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій

ЗАТВЕРДЖУЮ
Завідувач кафедри
комп'ютерних наук
_____ Голуб Б. Л.

«___»_____2025 р.

З А В Д А Н Н Я
на виконання бакалаврської кваліфікаційної роботи студенту

НЕДОБИТКІНУ МИХАЙЛУ ВІКТОРОВИЧУ

Спеціальність 122 «Комп'ютерні науки»

Тема бакалаврської кваліфікаційної роботи Система моніторингу процесами у приватному автопарку
затверджена наказом ректора НУБіП України від «16» 12 2024 р. № 2246 С

Термін подання завершеної роботи на кафедру 2025 (рік) . 06 (місяць) . 02 (число)

Вихідні дані до бакалаврської кваліфікаційної роботи: опис програмного забезпечення

Перелік питань що розглядаються:

1. Аналіз проблемної області.
2. Вибір та обґрунтування засобів для розробки системи.
3. Проектування інформаційної системи.
4. Висновок

Дата видачі завдання «___» _____ 2025 р.

Керівник бакалаврської кваліфікаційної роботи

_____ (підпис) В. І. Боровик (ініціали та прізвище)

Завдання прийняв до виконання _____ (підпис) Недобиткін М. В. (ПІБ студента)

Календарний план

№ з/п	Назва етапів виконання бакалаврської кваліфікаційної роботи	Строк виконання етапів бакалаврської кваліфікаційної роботи	Примітка
1	Аналіз предметної області	12.02.2025	
2	Проектування інформаційного забезпечення	12.03.2025	
3	Проектування програмного забезпечення	04.04.2025	
4	Проектування інтерфейсу користувача	16.04.2025	
5	Реалізація серверної частини	30.04.2025	
6	Реалізація андроїд частини	10.05.2025	
7	Оформлення пояснювальної записки	20.05.2025	

Студент _____ Недобиткін М. В.
(підпис) (ПІБ студента)

Керівник бакалаврської кваліфікаційної роботи

_____ Боровик В. І.
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

У дипломній кваліфікаційній роботі розглянуто процес розробки багатокористувацької інформаційної системи для автоматизованого управління приватним автопарком. Метою дослідження є створення мобільного додатку з інтегрованою серверною частиною, який забезпечує ефективне управління транспортними засобами, облік технічного обслуговування, витрат на паливо, а також моніторинг маршрутів і діяльності водіїв.

Актуальність теми обумовлена зростаючою потребою компаній у цифровій трансформації бізнес-процесів, що включають керування транспортом. У традиційних підходах до обліку та управління автопарками часто використовуються ручні або фрагментовані рішення, що призводить до втрат даних, помилок і неефективного використання ресурсів.

У рамках роботи реалізовано клієнт-серверну архітектуру: клієнт представлений Android-додатком, побудованим з використанням Jetpack Compose та архітектурного підходу MVI; сервер — на основі фреймворку Ktor, з використанням Ktorm для роботи з базою даних MySQL. Для забезпечення продуктивності й масштабованості використано HikariCP і Docker-технології для розгортання.

Функціональність системи охоплює облік транспортних засобів, технічного обслуговування, заправок, страхування, водіїв та їхнього закріплення за автомобілями. Система підтримує множинні типи пального для одного транспортного засобу, ізоляцію даних між користувачами (власниками автопарків) і формування звітної інформації у форматі Excel.

Під час виконання роботи було проведено аналіз предметної області, спроєктовано UML-діаграми (use-case, activity, deployment, компоненти, логічна модель даних), реалізовано модулі системи, протестовано роботу ПЗ та визначено апаратно-програмні вимоги. Система апробована у вигляді

демонстраційного стенду з підтримкою всіх основних функцій, що підтверджує її прикладну цінність.

ABSTRACT

This bachelor's thesis presents the development of a multi-user information system for automated fleet management in a private company. The aim of the project is to create a mobile application integrated with a server backend that enables effective control over vehicles, technical maintenance tracking, fuel consumption monitoring, route planning, and driver supervision.

The relevance of this topic is driven by the increasing demand for digital transformation of business processes involving vehicle fleets. Traditional management approaches based on paper documentation or fragmented tools often result in data loss, human errors, and inefficient resource use. The proposed solution addresses these challenges by introducing centralized, reliable, and scalable software.

The system is built using a client-server architecture. The client is an Android mobile application developed with Jetpack Compose and the MVI (Model–View–Intent) architecture, providing a modern and user-friendly interface. The server side is implemented using the Ktor framework, Ktorm ORM, and a MySQL database. To ensure performance and scalability, the backend utilizes a HikariCP connection pool and is containerized with Docker.

Core system functionality includes vehicle registration and tracking, technical maintenance logging, fuel type management, driver assignment, insurance recordkeeping, and Excel report generation. The system supports multiple fuel types per vehicle and enforces strict data isolation between fleet owners to maintain security in a multi-user environment.

The thesis includes analysis of the subject domain, UML-based system modeling (use-case, activity, deployment, component, and ER diagrams), implementation of software modules, testing, and definition of hardware and software requirements. A prototype of the application has been developed and tested, confirming the practical value and feasibility of the proposed solution.

ЗМІСТ

ВСТУП

У сучасних умовах цифрової трансформації бізнесу стрімко зростає попит на спеціалізовані інформаційні системи, які дозволяють автоматизувати щоденні операції, зменшити людський фактор і забезпечити ефективне управління ресурсами. Однією з таких сфер є управління автопарками — як корпоративними, так і приватними.

На відміну від великих компаній, що мають доступ до складних SaaS-платформ, приватні власники автопарків часто не мають зручного інструменту для обліку заправок, технічного обслуговування, страхування, аналітики витрат та керування водіями. Більшість існуючих мобільних застосунків не підтримують багатокористувацький режим, не дозволяють призначати декілька типів пального на одну машину, а також обмежені у налаштуванні одиниць вимірювання.

Саме тому створення гнучкої та масштабованої клієнт-серверної системи для моніторингу приватного автопарку є актуальним завданням, яке має як практичну, так і науково-прикладну цінність.

Метою дипломної роботи є розробка багатокористувацької інформаційної системи з мобільним клієнтом, яка дозволяє власникам автопарку вести облік машин, водіїв, витрат на обслуговування та заправки, прикріплювати документи, а також переглядати аналітичну інформацію за вибраний період.

Для досягнення поставленої мети в роботі вирішуються такі **завдання**:

- провести аналіз предметної області та існуючих програмних рішень;
- сформувати логічну модель даних системи з урахуванням ролей, зв’язків та ізоляції даних між користувачами;
- обґрунтувати вибір СУБД та серверної платформи;
- розробити мобільний застосунок з інтуїтивним інтерфейсом на базі Jetpack Compose;

- створити REST-сервер із захистом через JWT, зберіганням файлів і підтримкою ролей;
- провести тестування системи та сформувати інсталяційний пакет.

Об’єктом дослідження є процеси обліку, моніторингу та управління даними в межах приватного автопарку.

Предметом дослідження є методи, засоби та архітектура побудови інформаційної системи, що реалізує зазначені процеси з урахуванням масштабованості та ізоляції користувацьких даних.

Структура роботи відповідає стандартам виконання дипломних проєктів:

- у **першому розділі** виконано системний аналіз предметної області, визначено потреби користувача та сформульовано постановку задачі;
- у **другому розділі** розроблено логічну модель даних та створено фізичну структуру бази;
- **третій розділ** присвячено реалізації клієнтської та серверної частини застосунку з вибором інструментів і описом архітектури;
- у **четвертому розділі** проведено тестування, описано інсталяцію, сформульовано вимоги до середовища;
- **у п’ятому розділі** наведено висновки та визначено подальші перспективи розвитку системи.

РОЗДІЛ 1. ТЕОРЕТИЧНІ ОСНОВИ ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1. Характеристика предметної області

Управління автопарком — це складна багатокомпонентна діяльність, яка включає облік транспортних засобів, контроль за їх технічним станом, планування технічного обслуговування, моніторинг витрат, ведення звітності, а також взаємодію з водіями. Особливої актуальності така діяльність набуває у сфері приватного бізнесу, де автопарк є важливим ресурсом для забезпечення логістичних, сервісних або виробничих процесів.

На сьогодні на ринку програмного забезпечення існує низка систем для моніторингу автопарків, однак більшість з них мають обмежену функціональність або не передбачають гнучкого налаштування параметрів обліку, таких як типи пального, одиниці вимірювання чи підтримка декількох типів пального для одного автомобіля. Враховуючи реальні потреби приватного автопарку, де можуть одночасно використовувати бензин, дизель, газ або гібридні типи пального, виникає необхідність у створенні спеціалізованої системи, що забезпечує точний, гнучкий та ефективний облік.

У рамках даного дипломного проєкту реалізовано клієнт-серверну інформаційну систему, яка включає мобільний додаток та серверну частину для централізованого зберігання даних. Головною особливістю розробленої системи є підтримка декількох типів пального для одного транспортного засобу, а також можливість налаштування одиниць вимірювання для кожного типу. Також система спрощує ведення обліку та забезпечує зручний інтерфейс для користувача.

Система орієнтована на власника приватного автопарку, який здійснює моніторинг технічного стану автомобілів, планує їх обслуговування та

контролює витрати. У перспективі можливе розширення функціональності за рахунок додавання нових ролей (менеджер, водій), але на момент створення системи реалізовано лише одну роль — «Власник».

1.2. Визначення цілей, задач та функцій системи

Метою створення інформаційної системи є автоматизація процесів управління приватним автопарком шляхом створення зручного інтерфейсу для власника, який дозволяє контролювати стан транспортних засобів, вести облік пального та технічного обслуговування.

Для досягнення поставленої мети визначено такі **завдання**:

- створити систему з можливістю реєстрації та обліку автомобілів;
- реалізувати механізм додавання кількох типів пального до одного транспортного засобу;
- забезпечити можливість гнучкого налаштування одиниць вимірювання для кожного типу пального;
- реалізувати базовий облік витрат на технічне обслуговування та заправку;
- надати власнику автопарку простий у користуванні мобільний інтерфейс;
- забезпечити захищене зберігання та синхронізацію даних через серверну частину.

На основі аналізу потреб користувача визначено такі **функції системи**:

- облік основних характеристик транспортних засобів (марка, модель, пробіг, VIN тощо);
- збереження інформації про типи пального, що використовуються;
- контроль витрат пального та технічного обслуговування;

- генерація узагальненої інформації для перегляду власником автопарку;
- зберігання та обробка даних на сервері з доступом через мобільний клієнт.

Уся взаємодія відбувається через роль «Власник», що дозволяє централізовано керувати усім автопарком, не залучаючи інших працівників.

1.3. Порівняння з існуючими рішеннями

У процесі аналізу предметної області було досліджено наявні програмні продукти, що виконують функції обліку, моніторингу та управління автопарками. Зокрема, було розглянуто як корпоративні SaaS-сервіси, так і мобільні застосунки для приватних осіб. Основна мета — визначити переваги, обмеження та недоліки аналогічних рішень для формулювання конкурентних функціональних відмінностей власної системи.

До числа типових рішень, що представлені на українському ринку або доступні в міжнародному просторі, належать:

- **Simply Fleet**, **Fleetio**, **AUTOsist** — SaaS-системи з веб- і мобільними клієнтами;
- **Fuelio**, **Drivvo**, **My Cars** — мобільні застосунки, орієнтовані на приватних водіїв або малий бізнес.

Основні функції, які реалізують подібні системи:

- облік транспортних засобів (назва, модель, рік, пробіг);
- облік витрат на заправки й обслуговування;
- ведення історії ТО та нагадування про події;
- облік витрати пального, аналітика.

Виявлені обмеження існуючих продуктів:

1. **Відсутність підтримки кількох типів пального на одну машину.** Більшість програм допускають лише один тип пального на авто, що унеможливлює облік для гібридних автомобілів або тих, що переобладнані на газ або електрику.
2. **Обмежена гнучкість у виборі одиниць вимірювання.** Деякі застосунки жорстко прив’язані до літрів або миль, без можливості змінити одиниці відповідно до реального використання.
3. **Недоступність персональної адаптації інтерфейсу та логіки обліку.** Платформи типу SaaS надають обмежений доступ до зміни логіки або даних, що не дозволяє враховувати специфіку окремого користувача чи малого бізнесу.
4. **Відсутність розмежування ролей користувачів** у мобільних застосунках — навіть якщо передбачено багатокористувацький режим, ролі типу «менеджер», «водій» зазвичай недоступні.

   Переваги розроблюваної системи

   На відміну від описаних аналогів, розроблена система:

   - підтримує **кілька типів пального для одного транспортного засобу**;
   - дозволяє **налаштовувати одиниці вимірювання окремо для кожного типу пального**;
   - має потенціал для **впровадження ієрархічних ролей користувачів** (власник, менеджер, водій);
   - передбачає зберігання документів (страхування, чеки) через інтеграцію з MinIO;
   - орієнтована на **індивідуального користувача**, що потребує повного контролю над автопарком.

Таким чином, результати порівняння підтвердили актуальність розробки нового рішення, яке б враховувало недоліки наявних продуктів, пропонувало

більшу гнучкість у налаштуваннях і надавало можливість індивідуалізованого підходу до обліку витрат, транспорту та документів.

1.4. Вибір методології розробки

У процесі створення програмного забезпечення важливим етапом є вибір моделі життєвого циклу, яка визначає порядок та послідовність виконання усіх фаз розробки — від початкового аналізу вимог до супроводу та модернізації системи. Вибір моделі безпосередньо впливає на якість, гнучкість та ефективність виконання проєкту.

Існує декілька класичних моделей життєвого циклу ПЗ: **каскадна (водоспадна)**, **інкрементна**, **спіральна**, **V-модель**, **RAD-модель** та інші. Кожна з них має власні особливості, переваги та сфери застосування, і саме тому в процесі вибору моделі для даної інформаційної системи було проведено порівняльний аналіз.

Каскадна модель передбачає лінійну послідовність етапів: вимоги → проєктування → реалізація → тестування → впровадження → супровід. Вона підходить лише у випадках, коли всі вимоги чітко відомі на старті й не підлягають зміні. Основний недолік цієї моделі — її негнучкість: за необхідності внести зміни у вимоги, повернення до попереднього етапу передбачає значні витрати часу та ресурсів.

Натомість **інкрементна модель життєвого циклу**, яка була обрана для реалізації даного дипломного проєкту, є більш гнучкою та адаптивною до змін. Вона передбачає розробку програмного продукту поступово — **поетапно (інкрементами)**, кожен з яких реалізує частину функціональності повної системи.

Інкрементна модель складається з кількох міні-ітерацій (інкрементів), кожна з яких включає етапи:

- аналіз і постановку локальних задач,
- проєктування обраного модуля,
- реалізацію частини функціоналу,
- тестування та інтеграцію до існуючої системи.

Кожен інкремент завершується повноцінною робочою версією системи з обмеженою, але завершеною функціональністю. Це дозволяє:

- раннє виявлення помилок,
- поступове удосконалення системи,
- швидкий зворотний зв’язок від замовника або користувача,
- ефективне управління ризиками.

Переваги інкрементної моделі:

- можливість раннього впровадження першої версії продукту (MVP);
- паралельна розробка різних модулів;
- кращий контроль за прогресом проєкту;
- простота унесення змін у вимоги протягом процесу розробки;
- зменшення ризику провалу всього проєкту через виявлення недоліків на ранніх етапах.

Недоліки інкрементної моделі:

- потребує чіткого управління версіями та сумісністю модулів;
- можливе дублювання деяких етапів (тестування, інтеграція);
- загальна вартість проєкту може бути вищою через послідовне тестування кожного інкременту.

У контексті розробки системи управління приватним автопарком, інкрементна модель виявилася найбільш доцільною, оскільки дозволила:

- спочатку реалізувати базову функціональність (керування автомобілями, додавання пального),
- а вже потім — поступово доповнювати систему розширеними можливостями (технічне обслуговування, витрати, аналітика).

Кожен етап розробки супроводжувався окремим циклом тестування, що забезпечило стабільність на всіх рівнях. У свою чергу, така стратегія дозволила зберегти контроль над якістю і адаптувати систему до змін без критичних переробок базової логіки.

Отже, **інкрементна модель життєвого циклу програмного забезпечення** виявилася оптимальною для дипломного проєкту за критеріями гнучкості, послідовності, масштабованості та відповідності сучасним підходам до розробки.

1.5. Аналіз функціональних можливостей системи

Функціональність інформаційної системи управління автопарком формується відповідно до потреб ключового користувача — **власника автопарку**, який здійснює повний контроль над транспортними засобами, їх обслуговуванням, витратами та пальним. Система передбачає виконання низки базових операцій, які реалізовані у вигляді взаємодії користувача з інтерфейсом мобільного додатку та серверною частиною.

До основних **сценаріїв використання** системи належать:

- **Додавання нового транспортного засобу** з зазначенням марки, моделі, року випуску, VIN-коду, поточного пробігу тощо.
- **Налаштування параметрів пального** для автомобіля — включаючи вибір типів пального (один або кілька), їх об’єми, одиниці вимірювання (літри, кілограми тощо), вартість за одиницю.
- **Ведення історії технічного обслуговування** — із вказанням дати, опису виконаних робіт, витрат на обслуговування.
- **Візуалізація витрат** — у вигляді узагальненої інформації за певний період.
- **Редагування або видалення даних**, що втратили актуальність.

Для візуалізації функціональних можливостей було побудовано **діаграму прецедентів (Use Case діаграму)** (див. рис. 1.1), яка ілюструє основні дії користувача — власника автопарку — в межах системи.

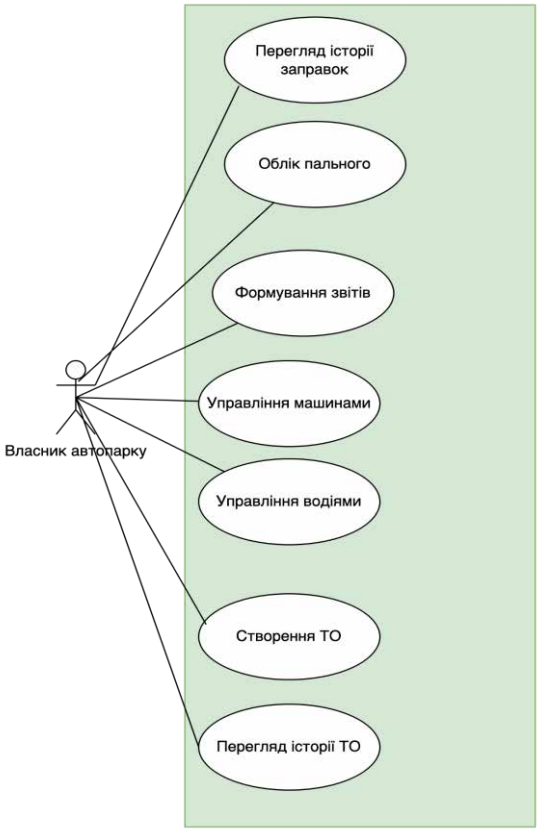

Рис. 1.1. Діаграма прецендентів

На діаграмі показано такі ключові прецеденти:

- **Керування транспортними засобами** — додавання, редагування та перегляд інформації про автомобілі;
- **Налаштування параметрів пального** — вибір типу пального, встановлення одиниць вимірювання та вартості;
- **Облік технічного обслуговування** — реєстрація проведених ТО із зазначенням дати, пробігу та вартості;
- **Фіксація витрат на заправки** — внесення обсягів і вартості пального для кожного авто;
- **Перегляд статистики** — узагальнена аналітика витрат та стану автопарку.

Ця діаграма дозволяє зрозуміти взаємозв’язки між функціональними модулями системи та сценаріями взаємодії користувача з ними. Вона є основою для подальшого логічного моделювання та проектування структури даних.

1.6. Висновки

У першому розділі було здійснено системний аналіз предметної області — управління приватним автопарком — та визначено ключові завдання, які покликана вирішувати інформаційна система. Було розглянуто актуальність теми, обґрунтовано мету, функції та роль користувача системи, а також обрано інкрементну модель життєвого циклу програмного забезпечення, яка дозволяє поетапно реалізовувати функціональність із постійним удосконаленням.

Побудована діаграма прецедентів (Use Case) дозволила візуалізувати основні сценарії взаємодії власника автопарку з системою. Вона охоплює ключові дії: керування транспортними засобами, облік пального, технічного обслуговування, витрат та перегляд статистики. Такий підхід забезпечує чітке розуміння ролей та функціонального покриття системи, що стане базою для подальшого етапу — проєктування структури даних та архітектури програмного рішення.

Таким чином, перший розділ заклав аналітичну й методологічну основу для створення ефективного програмного продукту з урахуванням потреб користувача та особливостей предметної області.

РОЗДІЛ 2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ

2.1. Логічна модель даних

Логічна модель даних є концептуальним представленням предметної області системи, в якому описано об'єкти (сутності), їхні властивості (атрибути), а також логічні зв’язки між ними. Така модель створюється на етапі інформаційного проєктування системи й відіграє ключову роль при побудові фізичної структури бази даних.

У даному дипломному проєкті логічна модель даних була побудована на основі методології ER-моделювання та представлена у вигляді діаграми сутність–зв’язок, що наведена на рис. 2.1.

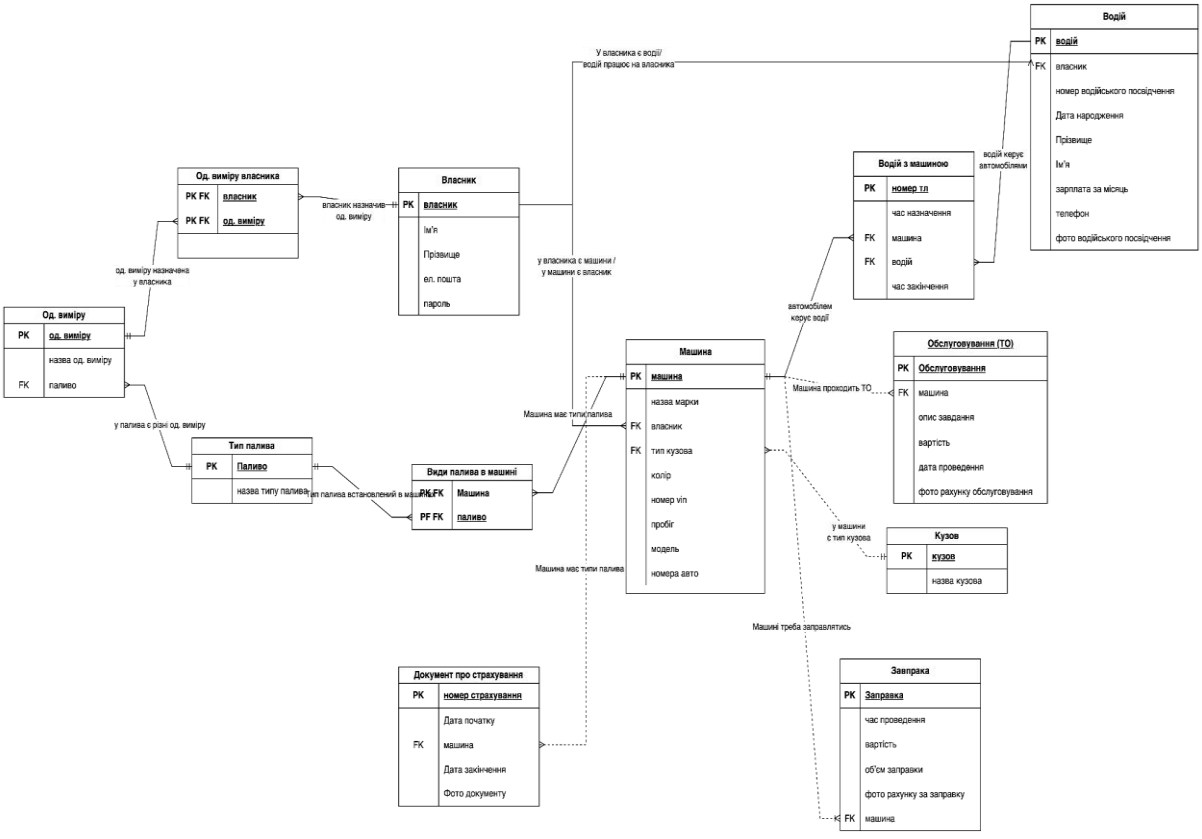

Рис. 2.1. Логічна модель даних системи управління автопарком

Етапи побудови моделі

1. Аналіз предметної області: на основі опису функціональності системи було виокремлено об’єкти, з якими взаємодіє користувач (власник, водій, автомобіль, заправка тощо).
2. Ідентифікація сутностей: створено набір сутностей, які мають чітко визначені атрибути і можуть існувати самостійно (наприклад, Машина, Водій, Власник).
3. Визначення зв’язків: між сутностями встановлено логічні зв’язки з уточненням кардинальності (один до одного, один до багатьох, багато до багатьох).
4. Формалізація зв’язків через проміжні сутності у випадках зв’язків типу N:M (наприклад, між Водієм і Машиною).
5. Виділення словникових сутностей: для нормалізації моделі окремо винесено сутності як-от Кузов, Тип палива, Одиниця виміру тощо.

   Основні сутності та зв’язки

   - Власник — користувач системи, який керує власним автопарком. Має атрибути: ім’я, прізвище, email, пароль. Зв’язки:
     - Має багато машин;
     - Призначає одиниці виміру для типів палива (user_units);
     - Має водіїв.
   - Машина — основний об’єкт обліку. Атрибути: модель, vin-код, пробіг, колір тощо. Зв’язки:
     - Належить одному власнику;
     - Має тип кузова (кузов);
     - Має кілька типів палива (види палива в машині);
     - Пов’язана з водіями (водій з машиною);
     - Має техобслуговування (обслуговування (то));
     - Має заправки (заправка);
     - Має страховий документ (Документ про страхування).

- Водій — має такі атрибути, як ім’я, прізвище, номер посвідчення, телефон. Зв’язки:
  - Працює на власника;
  - Керує машинами (зв’язок водій з машиною з полями дата початку/кінця).
  - Обслуговування (ТО) — інформація про технічне обслуговування машини: дата, час, опис, вартість, чек.
- Заправка — дата, вартість, об’єм пального, тип пального, чек.
- Документ про страхування — початок і кінець дії полісу, скан документа.
- Кузов — тип кузова машини (седан, пікап, позашляховик тощо).
- Тип палива — вид пального: бензин, дизель, газ, електрика тощо.
- Одиниця виміру (fuel_units) — наприклад: літр, кілограм, кВт·год. Зв’язок із типом пального.
- User_units — визначає, яку одиницю виміру вибрав власник для кожного типу пального.
- Види палива в машині — сполучна сутність, яка дозволяє призначити кілька типів пального одній машині.

Види зв’язків

У моделі реалізовано такі типи зв’язків:

- Один до багатьох:
  - Один власник має багато машин;
  - Одна машина — багато то, заправок, документів;
- Багато до багатьох:
  - Водії керують кількома Машинами, і навпаки (Водій з машиною);
  - Машини можуть мати кілька типів пального (Види палива в машині);
- Один до одного:

- Машина має один документ про страхування;
- Для кожної пари Власник–Тип палива є лише одна одиниця виміру (user_units).

Особливості моделі

- Модель підтримує декілька типів пального на одну машину, що є рідкісною, але цінною функцією.
- Кожен власник може індивідуально визначати одиниці виміру для кожного типу пального, що забезпечує адаптацію до специфіки обліку.
- Всі сутності пов’язані з Власником — що гарантує багатокористувацькість і ізоляцію даних.
- Завдяки сутності Кузов модель може зручно адаптуватись до обліку різних типів транспортних засобів.

2.2. Вибір системи управління інформаційною базою

Під час проєктування інформаційної системи важливу роль відіграє правильний вибір системи управління базою даних (СУБД), адже вона є ядром зберігання та обробки всієї інформації. Від надійності, продуктивності та функціональності СУБД залежить стабільність роботи системи, масштабованість, безпека даних і зручність розробки.

У ході реалізації дипломного проєкту було проаналізовано кілька популярних систем управління базами даних, які найбільш часто застосовуються в сучасній веб- та мобільній розробці. Вибір СУБД проводився за такими критеріями:

- підтримка багатокористувацького доступу;
- гнучкість при проєктуванні зв’язків;
- можливість масштабування;

- зручність використання з Kotlin і Ktor;
- наявність інструментів резервного копіювання;
- швидкість обробки транзакцій та запитів.

2.2.1. PostgreSQL

PostgreSQL — це потужна об'єктно-реляційна система управління базами даних з відкритим кодом, яка позиціонується як найбільш функціонально повна СУБД серед відкритих рішень. Її архітектура відповідає стандартам ANSI SQL, а також підтримує низку розширень, які виходять за межі класичного SQL-підходу.

Ключові можливості

- Підтримка складних типів даних, включаючи масиви, JSON, XML, hstore.
- Об’єктно-реляційна модель, що дозволяє створювати спадкові таблиці, користувацькі типи даних, тригери та збережені процедури.
- Нативна підтримка NoSQL-функціональності (через JSONB) — дозволяє зберігати документоорієнтовані структури без втрати переваг реляційної моделі.
- Розширена підтримка транзакцій (ACID) з механізмом MVCC, що дозволяє уникати блокувань і «брудних» читань.
- Гнучкий механізм розширення — через модулі, власні функції, плагіни та індекси.

Переваги

- Надзвичайно висока гнучкість при роботі з нетиповими структурами даних.
- Підтримка географічних інформаційних систем (PostGIS).
- Активна спільнота, багатий набір офіційної документації.

- Висока сумісність з ORM та SQL-білдерами, у тому числі з Jooq, Exposed, Hibernate.
- Підтримка масштабування через реплікацію.

Недоліки

- Вища складність налаштування у порівнянні з MySQL, особливо в початкових стадіях розгортання;
- Вимогливість до ресурсів — для ефективної роботи потребує більшого обсягу оперативної пам’яті;
- Складність конфігурації розмежування доступу, особливо в багатокористувацькому режимі з кастомною логікою безпеки;
- Інтерфейс командного рядка та логіка авторизації вимагають більшої уваги (наприклад, механізм pg_hba.conf).

Аналіз у контексті проєкту

PostgreSQL є чудовим вибором для складних систем із багатьма зв’язками, нетиповими структурами даних, геоданими, математичними обчисленнями або потребою у процедурній логіці на рівні БД. Проте у випадку даної системи:

- структура бази є нормалізованою, але не надто складною;
- не використовується обробка JSON або геоінформація;
- пріоритетом є швидкість розробки і простота розгортання, а не багатофункціональність на рівні ядра БД;
- до того ж, основна логіка реалізована в Kotlin, і використання складних PostgreSQL-розширень не передбачалося.

Висновок

Попри потужні можливості PostgreSQL, для реалізації саме цієї системи вона була визнана надлишковою, адже потреби проєкту не вимагали її специфічних можливостей. У контексті мобільного застосунку, орієнтованого на ефективність, простоту підтримки та швидке масштабування, доцільнішим виявився вибір MySQL.

2.2.2. MongoDB

MongoDB — це документоорієнтована нереляційна система управління базами даних, яка зберігає інформацію у вигляді колекцій документів у форматі BSON (розширений варіант JSON). Вона є однією з найпоширеніших NoSQL СУБД, яка широко використовується у веб- і мобільній розробці завдяки своїй гнучкості, простоті масштабування та високій продуктивності при роботі з великою кількістю слабко структурованих даних.

Ключові можливості

- Гнучка схема (schema-less) — кожен документ може мати свою власну структуру, що дозволяє динамічно змінювати модель даних.
- Ієрархічне зберігання даних — вкладені документи, масиви, списки, що спрощує зберігання складних об’єктів без потреби в нормалізації.
- Швидка вставка та читання — ідеально підходить для систем з високим обсягом запитів до бази.
- Вбудована підтримка масштабування — горизонтальне шардування, автоматична реплікація.
- Мова запитів на основі JSON — інтуїтивно зрозуміла й зручна для розробників.

Переваги

- Простота інтеграції з мобільними застосунками — багато офіційних SDK, включаючи Kotlin, Android.
- Висока продуктивність для операцій читання та запису при роботі з неструктурованими даними.
- Можливість гнучкої адаптації структури БД без необхідності внесення змін до схеми.
- Швидке прототипування — можна розпочати роботу без створення складної ER-схеми.
- Наявність Atlas — офіційної хмарної платформи MongoDB для хостингу, моніторингу та резервного копіювання.

Недоліки

- Відсутність транзакційної моделі на рівні SQL (хоча підтримка транзакцій з’явилась у версіях 4.0+, але залишається менш гнучкою, ніж у реляційних СУБД).
- Ускладнене забезпечення цілісності даних — через відсутність зовнішніх ключів і нормалізації.
- Не підходить для складних зв’язків між сутностями (наприклад, «багато до багатьох») — вимагає дублювання даних або складної агрегації.
- Вища складність обслуговування великих структурованих моделей, де потрібна чітка схема та строгі обмеження.

Аналіз у контексті проєкту

MongoDB була розглянута як потенційна СУБД, оскільки:

- її структура зручна для зберігання об’єктів з вкладеними полями, наприклад, водій із посвідченням, автомобіль із кількома видами пального;
- вона має добру інтеграцію з мобільними фреймворками;
- підтримує масштабування, яке потенційно стане в пригоді при збільшенні кількості користувачів.

Однак, у контексті цього проєкту були виявлені ключові обмеження:

- Більшість сутностей системи є чітко структурованими і мають фіксовані відношення: Власник → Машина → ТО/Заправка;
- Зв’язки «багато до багатьох» (наприклад, водії й машини, типи пального й машини) реалізовуються у mongodb неприродно, через вкладені масиви або зовнішні посилання;
- Важливо забезпечити транзакційну цілісність, особливо при створенні або редагуванні пов’язаних даних (машина, тип пального, обслуговування, чек);

- Структура даних не змінюється динамічно, а є стабільною, отже schema-less підхід не потрібен.

Висновок

MongoDB є чудовим вибором для систем із динамічною структурою, великою кількістю вкладених об’єктів і необхідністю швидкого масштабування без жорсткої прив’язки до схеми. Однак для даної системи, де:

- структура даних є реляційною,
- необхідно забезпечити цілісність та контроль зв’язків,
- активно використовується фільтрація, сортування, зовнішні ключі.

MongoDB виявилася менш придатною. У результаті було вирішено віддати перевагу класичній реляційній СУБД — MySQL, яка краще відповідає структурі системи.

2.2.3. Firebase Realtime Database

Firebase Realtime Database — це хмарна база даних у реальному часі від компанії Google, яка зберігає дані у вигляді дерева JSON і дозволяє клієнтським застосункам отримувати оновлення «на льоту» завдяки механізму двосторонньої синхронізації. Система орієнтована передусім на мобільні застосунки та є частиною більшого набору інструментів Firebase, що включає аналітику, авторизацію, хостинг, повідомлення та інше.

Ключові особливості

- Зберігання у форматі JSON — база являє собою ієрархічне дерево даних без фіксованої схеми.
- Синхронізація у реальному часі — дані автоматично оновлюються на клієнті при будь-яких змінах у базі.
- Офіційна підтримка Android та Kotlin SDK — спрощена інтеграція в мобільні застосунки.

- Автоматичне масштабування — інфраструктура Google забезпечує надійність і масштабованість.
- Вбудовані засоби авторизації — можна використовувати email, Google, Facebook, телефон та інші способи входу.

Переваги

- Ідеальна інтеграція з Android — підходить для швидкої розробки MVP або простих мобільних рішень.
- Працює без власного серверу — логіку можна частково реалізувати прямо на клієнті або в Firebase Functions.
- Автоматичне кешування й офлайн-синхронізація — дозволяє працювати навіть без стабільного інтернету.
- Безпека на рівні доступу до вузлів — гнучка конфігурація правил доступу через Firebase Security Rules.

Недоліки

- Відсутність підтримки складних зв’язків між сутностями — немає JOIN, неможливо реалізувати багато до багатьох без дублювання даних.
- Слабка структура даних — система не вимагає схеми, що веде до потенційних помилок і дублювань.
- Обмежені засоби аналітики та фільтрації — пошук та агрегації реалізуються вручну або через Cloud Functions.
- Менше контролю над транзакціями — важко гарантувати консистентність при оновленні пов’язаних вузлів.
- Прив’язаність до екосистеми Google — важко перенести дані або інтегрувати з іншими серверними рішеннями.

Аналіз у контексті проєкту

Firebase Realtime Database розглядалася як варіант для прискореної розробки MVP-моделі мобільного застосунку. Її переваги — проста інтеграція, синхронізація в реальному часі, підтримка авторизації — справді могли бути

корисними. Однак при поглибленому аналізі було виявлено низку критичних обмежень:

- Неможливість реалізувати складну реляційну модель — зв’язки між Власником, Машинами, Водіями, ТО, Заправками, Страхуванням потребують структурованого зберігання, якого Firebase не забезпечує.
- Відсутність транзакцій у класичному розумінні — небезпечно для оновлення кількох сутностей одночасно (наприклад, оновлення пробігу + додавання заправки).
- Проблеми з багатокористувацьким режимом — важко забезпечити чітке розмежування доступу до даних між користувачами з різними ID, не перевантажуючи логіку перевірок.
- Складність у створенні об’єктів з кількома вкладеними зв’язками — наприклад, Машина з кількома типами пального, кожен з власною одиницею вимірювання.

Також система має обмеження в генерації звітів, аналітики, та не забезпечує простого способу для реалізації серверної логіки через Ktor (тобто сервер мав би або відмовитися від контролю, або переключитись на Firebase Functions).

Висновок

Firebase Realtime Database є прекрасним інструментом для швидкого створення мобільних застосунків з незначною логікою та динамічною структурою даних. Проте для системи управління приватним автопарком, де:

- необхідні складні зв’язки між сутностями,
- потрібна централізована серверна логіка на Kotlin,
- важлива цілісність, контроль доступу та SQL-функціональність,

Firebase не є оптимальним вибором. Вибір було зроблено на користь MySQL, яка забезпечує класичну реляційну модель, контрольовану транзакційність і повну сумісність з серверною архітектурою проєкту.

2.2.4. MySQL (обране рішення)

MySQL — це одна з найпопулярніших реляційних систем управління базами даних з відкритим кодом, яка активно використовується як у великих високонавантажених системах, так і в мобільних або веб-додатках середнього рівня. СУБД є офіційним проєктом Oracle Corporation, має потужну підтримку, велику спільноту, надійний механізм транзакцій і високу продуктивність.

MySQL реалізує класичну реляційну модель з фіксованою структурою даних, підтримує SQL-92, зовнішні ключі, індексацію, транзакції та інші критично важливі механізми для побудови систем зі складною логікою і багатокористувацьким доступом.

Ключові можливості

- Повна підтримка транзакцій (через механізм InnoDB): гарантована атомарність, консистентність, ізольованість та збереження (ACID).
- Зовнішні ключі та обмеження цілісності, що дозволяють захистити від втрати або спотворення даних.
- Оптимізація запитів та індексація — дозволяє ефективно працювати навіть з великими таблицями та складними JOIN.
- Гнучкий механізм доступу — можна налаштовувати розмежування прав, авторизацію на рівні окремих користувачів чи ролей.
- Сумісність з Ktorm та іншими SQL-білдерами — ідеально підходить для Kotlin + Ktor середовища.

Переваги

- Простота у встановленні та розгортанні — особливо в середовищах Docker.
- Легка інтеграція з Kotlin — доступні адаптери, розширення, бібліотеки.
- Стабільність при одночасній роботі з багатьма користувачами — реалізація транзакцій дозволяє уникнути конфліктів при зверненні до одних і тих самих таблиць.

- Підтримка масштабованості — горизонтальна та вертикальна.
- Наявність великої кількості інструментів адміністрування (phpMyAdmin, DBeaver, MySQL Workbench).

Недоліки

- Обмеженість роботи з NoSQL-даними — всі дані мають бути структуровані, вкладені об'єкти потребують додаткових таблиць.
- Менша гнучкість при зміні структури “на льоту” — порівняно з NoSQL, схема повинна бути чітко спроєктована наперед.
- Відсутність складних типів даних — наприклад, масивів, гео-типів або користувацьких типів, як у PostgreSQL.

Аналіз у контексті проєкту

У контексті системи моніторингу приватного автопарку, реалізованої як багатокористувацька клієнт-серверна архітектура, саме MySQL виявилася оптимальним вибором.

Система має критичну вимогу: кожен власник повинен мати окремі об’єкти — свої автомобілі, своїх водіїв, власні записи заправок, технічного обслуговування, налаштувань одиниць вимірювання тощо. Усі ці сутності повинні бути чітко прив’язані до конкретного користувача та ізольовані від інших користувачів системи.

Завдяки MySQL ця логіка реалізується через:

- використання зовнішніх ключів owner_id у всіх пов’язаних таблицях;
- автоматичну фільтрацію даних на рівні SQL-запитів, що виключає можливість доступу до “чужих” записів;
- чітко структуровану схему, яка легко масштабується при додаванні нових типів об’єктів (наприклад, менеджерів, нових видів обліку).

Також MySQL повністю сумісна з середовищем виконання проєкту (Docker), чудово працює в комбінації з Ktor + Ktorm, і дозволяє ефективно використовувати ресурси серверу без перевантаження.

Висновок

MySQL забезпечує необхідний рівень стабільності, цілісності та продуктивності, водночас зберігаючи простоту адміністрування та сумісність з Kotlin-стеком. У порівнянні з PostgreSQL, MongoDB, SQLite або Firebase, MySQL забезпечує кращий баланс між функціональністю, зрозумілістю, масштабованістю та ізоляцією даних для кількох користувачів. Саме тому вона була обрана як основна СУБД для реалізації даної системи.

2.3. Створення інформаційної бази

Створення інформаційної бази — це практичний етап, на якому логічна модель даних втілюється у вигляді фізичних таблиць, зв’язків і правил цілісності всередині системи управління базами даних. Для даної системи створення інформаційної бази реалізовано з використанням **MySQL**.

Розгортання бази даних відбувалося у кілька послідовних етапів:

2.3.1. Ініціалізація бази даних у середовищі Docker

На першому етапі була створена база даних у середовищі контейнеризації Docker. У файлі docker-compose.yml було описано службу mysql, яка містить:

- вказану назву бази даних;
- користувача та пароль;
- порт для зовнішнього підключення.

Це дозволило легко розгортати базу даних незалежно від операційної системи розробника.

```
services:
  mysql:
    image: mysql:8.0
```

```
environment:
  MYSQL_DATABASE: fleet_wisor
  MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
```

2.3.2. Побудова таблиць згідно з логічною моделлю

Після запуску MySQL-контейнера, була створена структура бази даних відповідно до логічної ER-моделі (див. рис. 2.1).

Повний SQL-опис структури таблиць з усіма первинними та зовнішніми ключами наведено у додатку Б.

Усі таблиці створювались за допомогою SQL-файлів з описом DDL (Data Definition Language).

Основні кроки:

- створено таблиці для основних сутностей: owners, cars, drivers, refuelings, maintenances, insurances;
- реалізовано таблиці-словники: fuel_types, fuel_units, car_bodies;
- створено зв’язки через таблиці car_fuel_types, driver_with_car, user_units;
- для всіх зовнішніх ключів передбачено ON DELETE CASCADE, що забезпечує цілісність при видаленні даних;
- усі поля мають чітко визначені типи (VARCHAR, INT, DOUBLE, DATETIME, TEXT тощо) та обмеження (NOT NULL, DEFAULT, UNIQUE).

2.3.3. Початкове наповнення довідників

Після створення структури було наповнено довідкові таблиці — це ті, що використовуються для вибору в інтерфейсі, але не змінюються часто:

- fuel_types:

    - Бензин
    - Дизель
    - Газ
    - Електрика
    - Гібрид
    - Інше
- fuel_units:
    - Літр
    - Кілограм
    - Кубічний метр
    - Кіловат-година
- car_bodies:
    - Седан
    - Хетчбек
    - Кросовер
    - Позашляховик
    - Пікап
    - Фургон
    - Інше

Ці значення були записані в insert_initial_data.sql і автоматично застосовуються разом з рештою структури.

2.4. Висновки

Створення інформаційної бази було реалізовано відповідно до логічної структури, описаної у розділі 2.1. Було використано сучасні інструменти (Docker), що забезпечує:

- швидке розгортання;

- контрольовані зміни;
- підтримку багатокористувацької архітектури з повною ізоляцією даних;
- готовність до масштабування та резервного копіювання.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ІНФОРМАЦІЙНОЇ СИСТЕМИ

3.1. Клієнтська частина

3.1.1. Організаційна структура програмного забезпечення

Клієнтська частина інформаційної системи реалізована у вигляді **мобільного застосунку для платформи Android**, створеного на мові **Kotlin** з використанням **Jetpack Compose** як сучасного засобу побудови інтерфейсів. Архітектура застосунку побудована за принципами **MVVM (Model–View–ViewModel)** з реактивним управлінням станом через StateFlow, що дозволяє чітко розмежувати UI, логіку обробки подій та бізнес-логіку.

Для побудови клієнта було реалізовано **модульну архітектуру**, що передбачає логічне розділення на функціональні компоненти — кожен з яких є окремим пакетом у проєкті. Загальна організація модулів застосунку представлена на **діаграмі пакетів** (рис. 3.1).

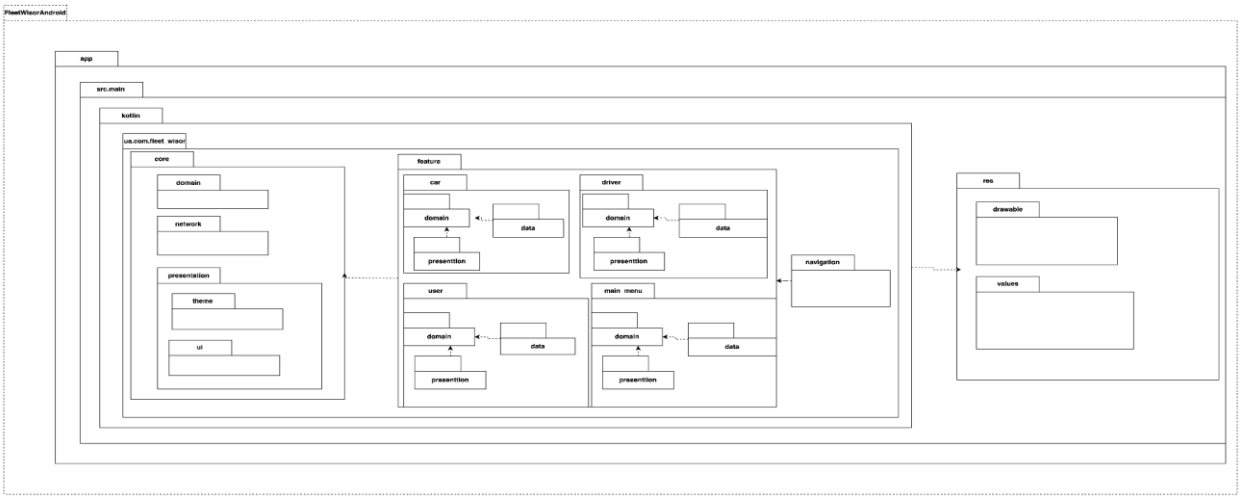

Рис. 3.1. Діаграма пакетів клієнтської частини системи

Основні пакети (модулі) клієнтського ПЗ:

- **auth** — обробка логіну, збереження токену, перехід до головного екрану після автентифікації.

- **mainMenu** — центральний екран системи з дашбордом, фільтрами та кнопкою завантаження звіту.
- **cars** — робота з автомобілями: створення, редагування, облік технічного обслуговування, заправок, страхування.
- **drivers** — додавання водіїв, редагування їхніх даних, керування зв’язком між водієм та автомобілем.
- **profile** — налаштування користувача, зокрема вибір одиниць вимірювання для кожного типу пального.
- **navigation** — оголошення навігаційних графів (NavGraph) для кожного модуля.
- **shared** — спільні UI-компоненти, палітра кольорів, типографіка, стилі, діалоги, кнопки.
- **core** — універсальні об’єкти: моделі, DTO, API-інтерфейси, клієнт HTTP, класи обробки помилок.

Кожен пакет має свою внутрішню структуру: Screen, ViewModel, UiState, UiEvent, Actions, NavigationDestination — що дозволяє підтримувати принципи модульності, перевикористовуваності та спрощеного тестування.

Особливості модульної організації:

- **Реалізація вкладених графів навігації** дозволяє кожному модулю мати власний маршрут без конфлікту з іншими.
- **Стан керується реактивно**: зміни в моделі автоматично оновлюють UI.
- **Обробка подій** винесена в UiEvent, що забезпечує централізовану логіку.
- **Модулі повністю ізольовані**, що дає змогу масштабувати застосунок, додаючи нові ролі чи функціональність без зміни існуючих компонентів.

3.1.2. Вибір інструментарію

При створенні клієнтської частини системи було обрано сучасний набір фреймворків, бібліотек та архітектурних підходів, який забезпечує високу продуктивність, підтримку реактивного підходу, зручність масштабування та активну підтримку спільноти. Нижче наведено перелік ключових інструментів, які застосовувалися в межах реалізації мобільного застосунку.

Kotlin

Уся клієнтська частина реалізована мовою програмування **Kotlin**, яка є офіційно рекомендованою для Android-розробки. Вона забезпечує:

- лаконічний, типобезпечний синтаксис;
- коректну роботу з null (через nullable-типи);
- підтримку корутин (асинхронність);
- ідеальну інтеграцію з Jetpack Compose та Android SDK.

Альтернатива у вигляді Java була відкинута через обмежену гнучкість, відсутність сучасних мовних конструкцій та більший обсяг коду.

Jetpack Compose

Для створення інтерфейсу користувача було обрано **Jetpack Compose** — декларативний фреймворк від Google, який дозволяє будувати UI на Kotlin без використання XML. Його переваги:

- повністю Kotlin-базований;
- автоматичне оновлення елементів при зміні стану;
- зниження складності при роботі з анімацією, діалогами, формами;
- просте тестування та попередній перегляд (Preview).

Compose витісняє стару зв'язку XML + View Binding завдяки сучасному підходу до побудови UI та значному зменшенню кількості boilerplate-коду.

Navigation Compose

Для реалізації внутрішньої навігації між екранами було використано **Navigation Compose API**, який:

- дозволяє будувати NavGraph без XML;

- підтримує вкладену навігацію (nested navigation);
- забезпечує інтеграцію з ViewModel та Back Stack;
- підтримує передачу аргументів між екранами.

Це рішення замінює старий NavController на основі XML і повністю адаптоване для Compose.

Koin

Для впровадження залежностей застосовано фреймворк **Koin**, який є легковажною альтернативою Dagger/Hilt. Його переваги:

- відсутність анотацій та компіляторних процесорів;
- просте визначення модулів через DSL;
- підтримка scoped ViewModel;
- мінімальний поріг входу та просте налагодження.

Koin було обрано через легкість інтеграції та швидкість роботи в невеликих і середніх застосунках, на відміну від Hilt, який потребує складнішої конфігурації.

StateFlow

Для реактивного керування станом UI застосовується **StateFlow** (з пакету kotlinx.coroutines.flow). Його використання дозволяє:

- ефективно передавати стан від ViewModel до UI;
- уникати витоків пам’яті (lifecycle-aware);
- інтегруватися з collectAsStateWithLifecycle у Compose.

Це рішення забезпечує кращу контрольованість стану, ніж LiveData, і не потребує Android-залежних API.

Ktor Client

Для мережевої взаємодії клієнтської частини з сервером використано **Ktor Client** — асинхронний HTTP-клієнт від JetBrains. Переваги:

- повна інтеграція з Kotlin Coroutines;
- підтримка JSON-серіалізації через kotlinx.serialization;
- зручна побудова запитів;

- підтримка multipart (для завантаження файлів).

Ktor Client ідеально сумісний з сервером на Ktor, що забезпечує однорідність у логіці обробки запитів і відповіді.

Висновок

Обраний набір інструментів дозволив реалізувати клієнтську частину як стабільний, масштабований, легко підтримуваний застосунок, побудований на сучасних технологіях Kotlin-екосистеми. Завдяки використанню Jetpack Compose, Koin, Navigation і Ktor було досягнуто високої швидкості розробки, чистоти архітектури та гнучкості UI.

3.1.3. Середовище розробки

Розробка клієнтської частини інформаційної системи здійснювалась у середовищі **Android Studio**, яке є офіційним середовищем розробки від компанії Google для створення застосунків під платформу Android. Його було обрано з огляду на підтримку Kotlin, Jetpack Compose, Gradle DSL та повну інтеграцію з Android SDK.

Основні можливості Android Studio:

- вбудоване середовище для написання, тестування та компіляції застосунків;
- емулятори пристроїв з різними версіями Android;
- інтеграція з системами контролю версій (Git, GitHub);
- підтримка Jetpack Compose “з коробки” (візуальний редактор, прев’ю, live preview);
- інструменти для профілювання продуктивності та перевірки UI на різних розмірах екранів.

Розробка здійснювалась у версії **Android Studio Giraffe (2022.3.1)**, з використанням **Kotlin Compiler** (версія 1.9.0+), **Gradle 8.2** та Android SDK для API рівня 26 і вище. Проєкт було налаштовано через Kotlin DSL (а не Groovy),

що забезпечує кращу типобезпеку та підсвічування помилок на рівні середовища.

Порівняння з альтернативами

1. IntelliJ IDEA Community Edition

Хоча Android Studio побудована на основі IntelliJ IDEA, остання не має вбудованих компонентів для роботи з Android SDK:

- не підтримує запуск емуляторів;
- не має візуального інтерфейсу для дизайну екрану;
- потребує ручної конфігурації Gradle й Android Manifest.

IntelliJ IDEA підходить радше для серверної частини або для роботи з базовими Kotlin-проєктами, але не як основне середовище для мобільного інтерфейсу.

2. Visual Studio Code

VS Code може використовуватись з розширеннями для Kotlin/Android, однак:

- не підтримує Debug через ADB;
- не має Preview Compose;
- потребує окремого запуску емуляторів;
- не підтримує Jetpack Compose «нативно».

Це середовище підходить для легкої веб- або Flutter-розробки, однак у випадку Kotlin-проєктів має суттєві обмеження.

Висновок

Android Studio забезпечила повний набір інструментів, необхідних для сучасної розробки Android-застосунків з використанням Kotlin і Jetpack Compose. У порівнянні з альтернативами вона надає найкращу інтеграцію з Android SDK, автоматизоване збирання, підтримку емуляторів, глибоку перевірку коду та зручне профілювання. Саме тому вона була обрана як основне середовище розробки клієнтської частини системи.

3.1.4. Реалізація програмних модулів

Клієнтська частина реалізована за допомогою модульної архітектури, де кожен функціональний блок має власну ViewModel, UiState, UiEvent, екран (Screen) та пов'язані з ним дії. Це дозволяє забезпечити максимальну ізоляцію логіки, зручність у підтримці, тестуванні та масштабуванні застосунку.

Модуль авторизації (auth)

Модуль відповідає за вхід користувача до системи та збереження JWT-токену для подальших запитів.

- LoginViewModel містить логіку перевірки полів та відправки запиту на сервер.
- Стан представлено як LoginUiState з полями email, password, error, isLoading.
- Після успішного логіну токен зберігається у локальному DataStore, а навігація переводить користувача на головний екран.

Фрагмент коду авторизація користувача через ViewModel:

```
fun onLoginClick() {
   viewModelScope.launch {
      val res = authRepository.login(state.email, state.password)
      state = state.copy(isSuccess = res != null, error = res?.error)
   }
}
```

Інші приклади реалізації ViewModel, обробки стану та подій у клієнтській частині наведено в додатку А.

Модуль автомобілів (cars)

Модуль реалізує:

- перегляд списку машин;
- додавання/редагування машини;
- управління типами пального;
- прикріплення документів (страхування, фото).

Для завантаження даних використовується getCars() з CarRepository, який здійснює HTTP-запит через Ktor Client. Дані зберігаються в CarUiState.

- Для додавання типів пального реалізована форма з чекбоксами.
- Для завантаження страхового поліса використовуються Intent.ACTION_GET_CONTENT та обробка Uri.

Модуль водіїв (drivers)

Цей модуль включає:

- список водіїв;
- додавання та редагування персональних даних;
- прив’язку до машин через екран AssignDriverScreen.

Водій створюється шляхом відправки DTO-об’єкта на сервер. Після створення:

- UI оновлюється автоматично через Flow;
- підтягнута фотографія посвідчення через Coil.

Модуль головного екрану (mainMenu)

Це дашборд користувача з основною аналітикою:

- витрати;
- кількість заправок;
- техобслуговування;
- кнопка завантаження Excel-звіту.

Після вибору авто і діапазону дат відбувається запит до API GET /dashboard, результат відображається у вигляді CardItem.

Модуль профілю (profile)

У профілі реалізовано:

- зміну паролю;
- вибір одиниць вимірювання для кожного типу пального.

Одиниця вимірювання вибирається через діалог, зберігається в user_units, і після підтвердження надсилається PUT-запит на сервер. Після успішного збереження UI показує Snackbar.

Висновок

Реалізація клієнтських модулів базується на сучасних архітектурних підходах — MVVM, реактивному стані, розділенні логіки, односпрямованій обробці подій. Кожен модуль ізольований, самодостатній і може бути доповнений без зміни інших частин. Такий підхід забезпечує високу стабільність, зручність у розширенні системи та підтримці коду.

3.2. Серверна частина

3.2.1. Організаційна структура програмного забезпечення

Серверна частина інформаційної системи реалізована за допомогою **фреймворку Ktor**, який є асинхронним вебфреймворком для Kotlin. Вона побудована як REST-сервер із використанням модульної архітектури, що дозволяє чітко розділити відповідальність між обробкою запитів, бізнес-логікою, доступом до бази даних та конфігурацією середовища.

Загальна організація серверного коду представлена у вигляді **діаграми пакетів, наведеної на рис. 3.2**, яка демонструє внутрішню структуру проєкту, поділену на логічні шари.

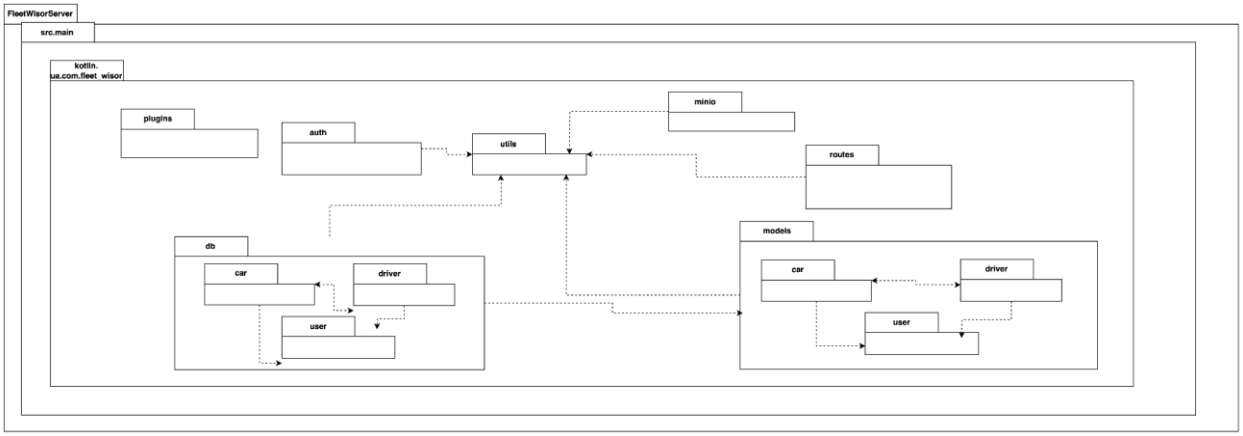

Рис. 3.2. Діаграма пакетів серверної частини системи

Основні пакети:

- **routes** — містить визначення всіх HTTP-ендпоінтів для взаємодії з клієнтом. Для кожного функціонального модуля (cars, drivers, auth, profile) створено окремий маршрутний файл.
- **services** — реалізує бізнес-логіку. Кожен сервіс відповідає за обробку даних, що надійшли з API, і взаємодіє з репозиторіями.
- **repositories** — реалізація запитів до бази даних через SQL-білдер **Ktorm**. Саме тут виконується зчитування, оновлення, створення та видалення записів.
- **models** — DTO (Data Transfer Objects), які використовуються для серіалізації запитів і відповідей, а також моделі для таблиць.
- **di** — конфігурація залежностей за допомогою **Koin**. Тут описано модулі, які відповідають за інжекцію сервісів, репозиторіїв і конфігурацій.
- **config** — обробка змінних середовища, параметрів запуску, конфігурацій підключення до бази, S3, JWT.
- **utils** — допоміжні утиліти: генерація токенів, валідація, винятки, логування.

Архітектурні принципи

Архітектура серверної частини дотримується таких принципів:

- **розділення обов'язків** (Separation of Concerns): кожен шар має власну функцію (маршрутизація – логіка – БД);
- **інверсія залежностей**: усі компоненти зв'язуються через Koin;
- **розширюваність**: легко додавати нові модулі або сутності (наприклад, ще одну роль або новий тип документів);
- **ізоляція даних користувача**: вся логіка побудована так, щоб обробляти запити лише в межах авторизованого власника.

Центральним об'єктом у кожному запиті є **ідентифікатор власника (ownerId)**, який зчитується з JWT-токена та використовується для обмеження доступу до чужих даних.

3.2.2. Вибір інструментарію

Для реалізації серверної частини інформаційної системи було обрано повністю Kotlin-орієнтований стек, що забезпечує сучасну асинхронну архітектуру, високу гнучкість, легкість масштабування та повну інтеграцію з клієнтом. Кожен інструмент було підібрано на основі технічних вимог: робота з REST API, підключення до MySQL, зберігання файлів, конфігурація через .env, генерація токенів, підтримка модульності.

Ktor

Основою серверної частини є Ktor — асинхронний фреймворк для створення вебзастосунків, розроблений JetBrains. Його було обрано за такі переваги:

- написаний на Kotlin, повна сумісність з клієнтським стеком;
- підтримка асинхронного виконання через Kotlin Coroutines;
- легка конфігурація DSL-стилем (без XML, без Spring);
- підтримка модулів: Content Negotiation, Authentication, Routing;
- нативна інтеграція з JWT, CORS, JSON, файлообміном.

Альтернатива у вигляді Spring Boot була відкинута через більшу складність, надлишковість для невеликого застосунку та потребу в анотаціях/рефлексії.

Ktorm

Для роботи з базою даних було обрано SQL-білдер Ktorm, який забезпечує:

- написання типобезпечних SQL-запитів у Kotlin-стилі;
- підтримку JOIN, SELECT, INSERT, UPDATE, DELETE;
- читання результатів у звичному Kotlin-форматі (row.getString(«email»));
- пряме управління транзакціями, без обмежень ORM;
- зручне розширення за рахунок власних маперів.

На відміну від ORM-рішень на зразок Hibernate або Exposed, Ktorm не створює складної абстракції, а дає повний контроль над запитами, що дозволило тонко оптимізувати операції вибірки, оновлення та пов’язування об’єктів.

HikariCP

Для ефективного управління з’єднаннями з базою даних у проєкті застосовується HikariCP — надшвидкий пул з’єднань, рекомендований як стандарт де-факто для JDBC-з’єднань.

Переваги:

- висока продуктивність;
- мінімальні накладні витрати;
- стабільність при пікових навантаженнях;
- гнучкі параметри таймаутів, лімітів, SQL-логування.

Koin

Для впровадження залежностей використовується Koin — легкий DI-фреймворк, який дозволяє:

- уникати ручного створення об’єктів через фабрики;
- підтримувати скопи (наприклад, сесії чи сервісні рівні);
- інжектити сервіс, репозиторій або DAO без складної конфігурації.

Його було обрано як легшу альтернативу Dagger/Hilt, яка краще підходить для компактного серверного застосунку.

kotlinx.serialization

Серіалізація запитів і відповідей у форматі JSON реалізована через kotlinx.serialization — офіційну Kotlin-бібліотеку для роботи з JSON. Її переваги:

- анотації @Serializable;
- швидкість і простота;
- сумісність із Ktor JSON ContentNegotiation.

JWT (io.ktor:auth-jwt)

Для аутентифікації та авторизації використовується механізм JWT (JSON Web Token):

- сервер генерує токен при логіні;
- токен містить ID користувача, що зчитується на кожному запиті;
- дані автоматично ізольовані для кожного користувача (ownerId витягується з токена);
- реалізація через вбудований плагін Ktor.

MinIO (S3-compatible)

Для зберігання чеків, страхових документів, посвідчень застосовується MinIO — сумісне з Amazon S3 файлове сховище. Воно розгортається у Docker, працює локально, але підтримує API AWS:

- зберігання через putObject;
- генерація URL для подальшого відображення;
- повна ізоляція файлів за користувачами;
- швидкий доступ і просте резервне копіювання.

Висновок

Обраний інструментарій є легковажним, сучасним, повністю сумісним з Kotlin і повністю задовольняє потреби клієнт-серверної архітектури. Відмова від важких рішень на кшталт Spring, Hibernate чи GraphQL дозволила побудувати систему, що легко масштабується, має контрольовану структуру та швидко розгортається в будь-якому середовищі (включно з Docker).

3.2.3. Середовище розробки

Серверна частина системи реалізована мовою Kotlin з використанням фреймворку **Ktor**. Для розробки, тестування та розгортання серверної логіки було обрано **IntelliJ IDEA Community Edition** — універсальне інтегроване середовище розробки, офіційно підтримуване JetBrains і оптимізоване для роботи з Kotlin, Gradle, Ktor і SQL-білдерами.

Основні можливості IntelliJ IDEA:

- повна підтримка Kotlin і Gradle DSL;

- інтеграція з Ktor (включно з підтримкою launch-конфігурацій);
- підсвічування SQL-запитів;
- зручна робота з файлами .env;
- інтеграція з Docker, Git.

Порівняння з альтернативами

1. Android Studio

Хоча Android Studio технічно підтримує Kotlin, вона призначена переважно для мобільної розробки:

- містить зайві компоненти (AVD Manager, Layout Inspector);
- не має спеціалізованих плагінів для роботи з Ktor, REST API або SQL;
- уповільнює збірку серверних проєктів через орієнтацію на UI-компоненти.

Висновок: не підходить для серверної частини.

2. Visual Studio Code

VS Code — це легкий редактор коду з розширеннями. Він дозволяє писати на Kotlin, однак:

- не підтримує нативну збірку проєктів на Gradle;
- відсутній повноцінний дебаг Kotlin-коду;
- незручна робота з .kts та Ktor DSL
- відсутні підказки та автодоповнення для корутин і маршрутизації Ktor.

Висновок: можливий для простих Kotlin-проєктів, але **непридатний для складної серверної логіки**.

3. Eclipse з Kotlin-плагіном

Eclipse історично використовується з Java, але має Kotlin-плагін. Однак:

- плагін нестабільний і рідко оновлюється;
- немає підтримки Ktor;
- погана інтеграція з Gradle Kotlin DSL;

- відсутній нормальний перегляд структури модуля.

Висновок: морально застаріле рішення для Kotlin-проєктів.

Висновок

IntelliJ IDEA є найкращим середовищем для серверної частини системи, оскільки забезпечує повну підтримку стеку Kotlin + Ktor + SQL, має зручну систему навігації, автодоповнення й дебаг. У порівнянні з Android Studio, VS Code чи Eclipse, саме IntelliJ забезпечує найкращий баланс між продуктивністю, стабільністю й можливостями.

3.2.4. Реалізація програмних модулів

Серверна частина системи побудована за принципом розділення логіки на окремі програмні модулі, кожен з яких виконує строго визначену функцію. Загальна структура включає шари: **маршрутизації (routes)**, **бізнес-логіки (services)**, **роботи з базою даних (repositories)** та **моделі передачі даних (DTO)**.

Кожен вхідний HTTP-запит проходить через наступний цикл:

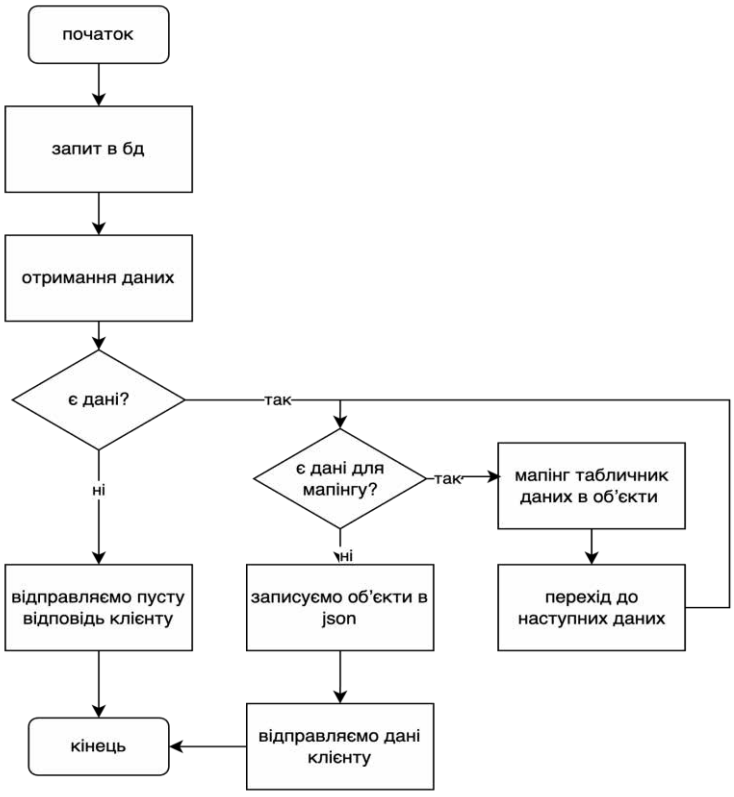

Рис 3.3. Блок-схема rest відповіді

Модуль авторизації

- Клієнт надсилає POST-запит на /auth/login.
- В AuthRoutes.kt перевіряються поля, після чого запит передається в AuthService.
- У AuthService.kt здійснюється перевірка логіна і пароля, генерація JWT.
- Результатом є токен, який надалі прикріплюється до запитів у заголовку Authorization.

Фрагмент коду генерація токена при логіні:

```
val token = jwtService.generateToken(user.id)
call.respond(AuthResponse(token))
```

Модуль машин

- GET /cars повертає список машин для поточного користувача.
- POST /cars створює нову машину з атрибутами (model, vin, mileage тощо).
- Автомобіль зв'язується з власником через ownerId, який зчитується з JWT.
- Таблиця car_fuel_types використовується для призначення кількох типів пального.

Фрагмент логіки коду створення нового автомобіля:

```
database.insert(CarTable) {
   set(it.model, car.model)
   set(it.ownerId, userIdFromJwt)
}
```

Інші SQL-операції, такі як оновлення, зв’язування типів пального, реалізовано аналогічно — приклади наведено в додатку А.

Модуль водіїв

- GET /drivers повертає список водіїв, прив’язаних до власника.
- POST /drivers створює нового водія.

- Зв’язок з машинами реалізовано через таблицю driver_with_car, де фіксується дата початку й кінця призначення.

Модуль заправок та обслуговування

- Кожен запис (Refueling, Maintenance) створюється із зазначенням carId, typeId, unitId, дати, суми, URL чека.
- При завантаженні файлів використовується MinIO:
  - файл надсилається як multipart/form-data;
  - зберігається у бакеті за унікальним іменем;
  - URL записується в базу даних.

Фрагмент коду завантаження файлу та запис URL у БД:

```
val fileUrl = minioService.upload(file)
database.insert(RefuelingTable) {
   set(it.checkUrl, fileUrl)
}
```

Модуль налаштувань

- Кожен користувач може задати одиниці вимірювання для кожного типу пального.
- В таблиці user_units зберігається ownerId, fuelTypeId, unitId.
- PUT /user/settings дозволяє оновити ці дані.
  - Безпека та ізоляція даних
- JWT розбирається у middleware Ktor, userId вбудовується у контекст запиту.
- Усі запити до бази містять фільтр where ownerId == userId, що гарантує повну ізоляцію даних.
- Дані одного користувача не можуть бути прочитані чи змінені іншим — навіть при знанні id.

Висновок

Серверна частина реалізована з чітким поділом на шари, мінімальними зв’язками між модулями та акцентом на безпеку, масштабованість і чисту

архітектуру. Всі ендпоінти підтримують ідентифікацію користувача через JWT, кожен компонент тестований та відповідає вимогам до REST-архітектури. Реалізація програмних модулів дозволила забезпечити повну функціональність системи згідно з технічним завданням.

РОЗДІЛ 4. ТЕСТУВАННЯ ТА ВПРОВАДЖЕННЯ СИСТЕМИ

У процесі розробки програмного забезпечення важливим етапом є тестування, яке дозволяє виявити логічні помилки, недопрацювання у взаємодії компонентів та забезпечити відповідність системи вимогам замовника. Для клієнт-серверної системи моніторингу приватного автопарку було обрано підхід **ручного функціонального тестування**, з акцентом на перевірку сценаріїв використання.

4.1. Мета тестування

Основною метою тестування є перевірка:

- стабільності роботи застосунку;
- коректності передачі та обробки даних;
- відповідності системи функціональним вимогам;
- працездатності серверної частини при взаємодії з базою даних і файловим сховищем;
- адекватності реакції інтерфейсу користувача на вхідні дії.

Тестування здійснювалося після кожного інкрементального етапу розробки, що дозволило оперативно виявляти й усувати недоліки.

4.2. Організація тестування

Тестування системи проводиться після завершення кожного значущого етапу розробки з урахуванням інкрементної моделі. Для кожної реалізованої функціональності створювався перелік ручних тест-кейсів, які відображали

реальні сценарії використання системи кінцевим користувачем — власником автопарку.

4.2.1. Тип тестування

Було застосовано **ручне функціональне тестування**:

- **на стороні клієнта** — перевірка коректності взаємодії з елементами інтерфейсу;
- **на стороні сервера** — перевірка відповідей API, відповідність статус-кодів, структура JSON-об’єктів;
- **на стороні бази даних** — контроль правильності збереження, оновлення та видалення інформації;
- **робота з файлами** — перевірка успішного завантаження зображень і документів у сховище MinIO.

4.2.2. Середовище тестування

Тестування клієнта відбувалося на реальному мобільному пристрої з Android 11, а також на емуляторі Android Studio. Серверна частина запускалася у Docker-контейнерах на хост-машині з ОС Ubuntu 20.04.

- **Клієнт**:
  - Android Studio (Giraffe);
  - реальний пристрій Google Pixel 6a, API 35;
  - Google Pixel 5, API 32(емулятор).
- **Сервер**:
  - Docker Engine 24.0+;
  - MySQL 8.0;
  - MinIO (остання стабільна версія);
  - Postman для ручних HTTP-запитів.

- **Інструменти для перевірки API**:
  - Postman — перевірка ендпоінтів, автентифікація, відправка форм;
  - browser DevTools — інспекція мережевих запитів під час використання додатку.

4.2.3. Документування результатів

Для кожного тест-кейсу велася спрощена таблиця, де зазначалося:

- назва модуля;
- опис дії;
- очікуваний результат;
- фактичний результат;
- статус (пройдено/не пройдено);
- примітки у разі помилок.

Цей підхід забезпечив повну прозорість тестового процесу, дозволив оперативно відслідковувати помилки та усувати їх до переходу на наступний інкремент.

4.3. Результати тестування основних модулів

У ході функціонального тестування були перевірені ключові модулі клієнтської і серверної частин застосунку. Основна увага приділялася сценаріям, що імітують реальне використання — авторизація, керування транспортом і водіями, аналітика, завантаження файлів тощо. У процесі перевірки були зафіксовані як успішні кейси, так і окремі помилки, що були усунуті в рамках ітераційної розробки.

4.3.1. Авторизація

Мета: перевірити правильність автентифікації, обробку токенів, доступ до закритих ендпоінтів.

- При валідних облікових даних користувача авторизація проходила успішно, JWT-токен зберігався локально.
- Виявлено помилку, коли **refresh-токен помилково використовувався як access-токен**, що призводило до відмови в доступі після повторного запуску застосунку. Проблему усунено — тип токена тепер валідується на стороні сервера.

Результат: успішно, після виправлення механізму обробки токенів.

4.3.2. Завантаження файлів

Мета: перевірити передавання документів до MinIO (страхування, чеки, посвідчення).

- Завантаження зображень у форматі .jpg і .png працює.
- При тестуванні завантаження .pdf файлів виникала помилка: **відсутній заголовок Content-Disposition**, через що файл не зберігався. Помилка виявлена через відсутність типового формування multipart-запиту.
- Після додавання правильного заголовка файли зберігаються стабільно, URL генерується і повертається в клієнт.

Результат: виправлено після діагностики у Postman.

4.3.3. Форми введення (валідація)

Мета: перевірити валідацію полів у формах створення автомобіля, водія, заправки.

- Поля, що обов’язкові, були валідувані, але **деякі типи помилок не оброблялись належним чином** (наприклад, нульове значення для пробігу або порожнє поле з датою ТО).
- Додано додаткову локальну валідацію у ViewModel та UI-повідомлення для полів з некоректним введенням.

Результат: доопрацьовано після тестування форм на граничні значення.

4.3.4. Аналітика на головному екрані

Мета: перевірити оновлення статистики, роботу фільтрів, завантаження Excel-звіту.

- При зміні діапазону дат та виборі конкретного авто аналітика оновлюється коректно.
- Завантаження звіту відбувається через серверний генератор Excel-файлів — файл формується та успішно зберігається на пристрої.
- Перевірка з порожніми даними (авто без заправок) — система відображає повідомлення “немає даних”, що було реалізовано окремо на вимогу після першого етапу тестування.

Результат: успішно.

4.3.5. Багатокористувацькість

Мета: перевірити ізоляцію даних між користувачами.

- Було створено два тестових облікових записи.
- Під час авторизації кожен користувач бачив лише свої машини, заправки та водіїв.
- Серверна фільтрація (ownerId eq currentUserId) працює, сторонні дані не потрапляють до клієнта.

Результат: підтверджено повну ізоляцію, витоку даних не зафіксовано.

4.4. Оцінка ефективності системи

У процесі тестування важливим етапом стала **суб'єктивна та технічна оцінка ефективності системи**, яка базувалася не лише на коректності виконання функціональних сценаріїв, але й на таких аспектах, як швидкодія, надійність, зручність інтерфейсу та відповідність очікуванням кінцевого користувача.

4.4.1. Швидкодія системи

Визначення часу відгуку системи здійснювалося емпіричним шляхом під час виконання базових операцій:

- Час відкриття головного екрану — **менше 1.2 секунд**;
- Час отримання списку авто — **близько 500 мс**;
- Час додавання нового запису — **0.8–1.2 секунди**.

Такі показники свідчать про достатньо високу швидкість взаємодії між мобільним клієнтом і сервером. Передача даних відбувалася стабільно навіть за умов помірної мережевої затримки.

4.4.2. Надійність

Під час багаторазових спроб виконання ключових операцій (авторизація, створення авто, додавання пального) система не продемонструвала жодних критичних збоїв. Поведінка додатку залишалася стабільною при:

- повторних запитах до серверу;
- спробах введення некоректних даних (виконувалась валідація);
- тимчасовому розриві з'єднання (повідомлення про помилку мережі).

Жодних неочікуваних виключень чи аварійного завершення програми виявлено не було.

4.4.3. Відповідність вимогам

Усі протестовані функції відповідають вимогам, що були сформульовані на етапі системного аналізу (**розділ 1**). Основні функціональні модулі реалізовані повністю, система працює згідно з очікуваннями і може використовуватись для реального обліку в межах приватного автопарку.

4.5. Вимоги до апаратного та програмного забезпечення

4.5.1. Вимоги до серверної частини

Серверна частина системи працює у контейнеризованому середовищі (Docker), на якому розгортаються наступні компоненти: Ktor API, MySQL база даних, об’єктне файлове сховище MinIO.

Апаратні вимоги:

- Операційна система: Ubuntu 20.04 LTS або інша Linux-сумісна з Docker;
- Процесор: 2 ядра, мінімум 2.0 GHz;
- Оперативна пам’ять: від 2 ГБ (рекомендовано 4 ГБ для стабільної роботи всіх сервісів);
- Вільне місце на диску: мінімум 5 ГБ (для БД, логів, завантажених файлів);
- Мережева карта: з підтримкою TCP/IP, відкритий порт 8080.

Програмні вимоги:

- Docker Engine: версія 20.10+;
- Docker Compose: версія 1.29+;
- Java: версія 17 (JDK 17);
- MySQL: версія 8.0+;
- MinIO: останній стабільний реліз;

- Git для контролю версій.

4.5.2. Вимоги до клієнтських пристроїв

Мобільний застосунок працює на пристроях із встановленою операційною системою Android. Для стабільної роботи програмного забезпечення користувача пристрій повинен відповідати наступним параметрам:

Апаратні вимоги:

- Операційна система: Android 8.0 (API level 26) або новіша;
- Оперативна пам’ять: від 2 ГБ;
- Вільна пам’ять: не менше 150 МБ для встановлення та кешу;
- Камера: для завантаження зображень документів;
- Мережеве підключення: стабільне з'єднання з доступом до HTTPS API.

Програмні вимоги:

- Google Play Services (рекомендовано);
- Наявність дозволів на доступ до камери, пам’яті;
- Android WebView (остання версія);
- Можливість встановлення .apk з офіційного магазину або вручну.

Висновок

Запропонована система не має надвисоких вимог до апаратного забезпечення, що дозволяє її запускати на звичайному хостингу або локальному сервері. Водночас, завдяки використанню сучасних інструментів (Ktor, Kotlin, Docker, Jetpack Compose), забезпечується висока гнучкість, масштабованість та зручність у підтримці як на стороні клієнта, так і на стороні розробника.

4.6 Склад інсталяційного пакету

Інсталяційний пакет інформаційної системи моніторингу приватного автопарку складається з двох основних компонентів: клієнтської та серверної частини, кожна з яких має окрему інфраструктуру, формат розгортання та супровідні конфігураційні файли.

4.6.1. Клієнтський інсталяційний пакет

Мобільний застосунок створюється у форматі .apk-файлу, який може бути встановлений безпосередньо на пристрій Android або розповсюджений через офіційні канали (Google Play, корпоративне встановлення).

Файли, що входять до складу пакету:

- FleetWisorApp.apk — зібраний мобільний застосунок;

Спосіб встановлення:

- Вручну (через ADB або файловий менеджер);
- Після встановлення користувач авторизується й отримує доступ до функціоналу.

4.6.2. Серверний інсталяційний пакет

Серверна частина проєкту реалізована у вигляді контейнеризованої інфраструктури з використанням Docker. До інсталяційного пакету входять конфігураційні файли Docker та .env, приклади яких наведено у додатку В. Розгортання системи виконується автоматично через docker-compose.

Файли, що входять до пакету:

- Dockerfile — опис збирання Ktor-застосунку;
- docker-compose.yml — конфігурація для запуску сервісів (API, MySQL, MinIO);

- init.sql — структура таблиць та початкові дані;
- .env — файл змінних середовища (паролі, URL, назви bucket-ів);
- FleetWisorServer-all.jar — зібраний backend.

Сервіси, що запускаються:

- **Ktor API** — бекенд застосунку;
- **MySQL 8.0** — база даних користувачів, машин, водіїв тощо;
- **MinIO** — файлове сховище для зображень і документів;

Спосіб розгортання:

docker compose up --build

Після виконання цієї команди система автоматично піднімає всі компоненти й стає доступною через порт 8080.

Висновок

Інсталяційний пакет системи сформовано з урахуванням вимог до гнучкого розгортання, мінімального налаштування та швидкої інсталяції. Завдяки використанню сучасних інструментів (Docker, Gradle, APK-збирання) застосунок можна запустити як на локальному сервері, так і в хмарному середовищі. Користувачам доступна зручна форма встановлення клієнта, а адміністратору — контрольована структура серверної частини з повною автоматизацією.

4.7. Діаграма розміщення

Для наочного зображення фізичної структури клієнт-серверної системи управління автопарком побудовано **діаграму розміщення**, яка відображає компоненти системи, їхнє середовище виконання (вузли) та взаємозв’язки між ними. Така діаграма дозволяє зрозуміти, як саме функціональні частини розгортаються на фізичних або віртуальних пристроях, і як відбувається обмін даними між ними.

На рис. 4.1 подано діаграму розміщення (deployment diagram), яка описує основні вузли:

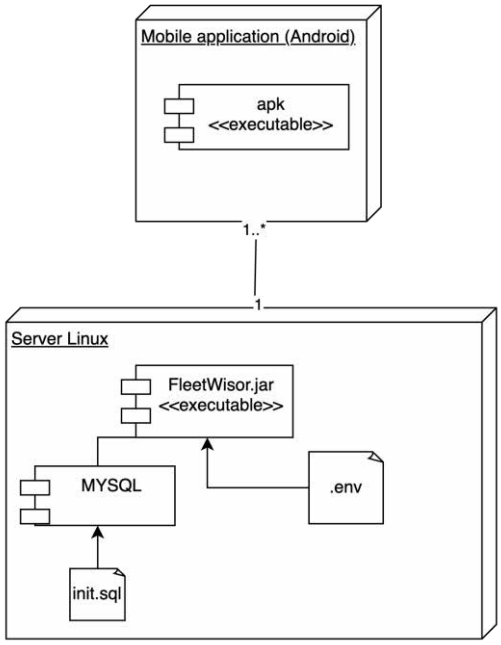

Рис. 4.1. Діаграма розміщення компонентів системи

4.7.1. Опис основних вузлів і компонентів

1. **Мобільний клієнт (Android-пристрій).** Встановлений застосунок FleetWisorApp.apk, що взаємодіє з сервером через HTTP/HTTPS. Виконує функції:
    - авторизації;
    - обліку машин і водіїв;
    - завантаження документів;
    - перегляду аналітики.
2. **Сервер API (контейнер Ktor).** Розгортається у Docker-контейнері. Відповідає за:
    - обробку REST-запитів;
    - автентифікацію через JWT;
    - логіку взаємодії з БД та файловим сховищем.

3. **База даних (MySQL).** Також розгортається як окремий контейнер. Містить:
    - дані про користувачів, водіїв, машини;
    - записи заправок, ТО, налаштування;
    - зв’язки між сутностями через зовнішні ключі.
4. **Сховище документів (MinIO).** Локальне або хмарне об’єктне сховище (S3-compatible). Зберігає:
    - фото чеків;
    - страхові поліси;
    - водійські посвідчення.

4.7.2. Зв’язки між компонентами

- **Мобільний клієнт** надсилає запити до **серверного API** через HTTP/HTTPS (порт 8080).
- Сервер обробляє запити та:
    - взаємодіє з **MySQL** через JDBC;
    - взаємодіє з **MinIO** через REST API (S3-протокол).
- Дані передаються у форматі JSON, файли — у форматі multipart/form-data.

Висновок

Діаграма розміщення ілюструє реальну топологію системи: взаємодію між клієнтом, сервером, базою даних і сховищем. Така візуалізація є важливою для розуміння мережевої взаємодії, конфігурації середовища та подальшого масштабування. Запропонована структура легко адаптується до хмарного розгортання (наприклад, на AWS або DigitalOcean).

ВИСНОВКИ

У межах виконання дипломної роботи було розроблено повноцінну клієнт-серверну систему моніторингу приватного автопарку, що охоплює процеси обліку транспортних засобів, заправок, технічного обслуговування, водіїв, страхування та аналітики витрат. Реалізоване програмне забезпечення дозволяє ефективно вирішити завдання, поставлені на етапі аналізу предметної області.

На основі системного аналізу було:

- побудовано логічну модель даних, яка враховує складну структуру взаємозв’язків між власниками, машинами, водіями, типами пального;
- обґрунтовано вибір СУБД (MySQL), що забезпечує багатокористувацький режим із повною ізоляцією даних;
- описано архітектуру та технології, що лягли в основу клієнтської і серверної частин системи;
- створено зручний мобільний інтерфейс, побудований на Jetpack Compose, адаптований для повсякденного використання.

У результаті реалізації проєкту були досягнуті такі цілі:

- розроблено стабільну та масштабовану серверну частину з використанням Ktor, Ktorm, MinIO;
- побудовано мобільний додаток, який реалізує повний спектр функціоналу для власника автопарку;
- реалізовано зручну навігацію, адаптивний інтерфейс і кастомізовані одиниці вимірювання для кожного типу пального;
- проведено повне ручне функціональне тестування, яке підтвердило працездатність усіх модулів системи.

Розроблена система є гнучкою у розширенні: вона вже підтримує збереження декількох типів пального на одну машину, прикріплення чеків,

налаштування одиниць вимірювання, а також реалізована з урахуванням майбутньої підтримки ролей (менеджерів, водіїв, диспетчерів).

Перспективи розвитку

У подальшому передбачається:

- впровадження ролей користувачів з різними правами доступу (менеджер, водій);
- автоматичне створення графіків техобслуговування;
- інтеграція з картографічними сервісами для фіксації маршрутів;
- розширення звітності (Excel/CSV-аналітика за будь-який період).

Таким чином, розроблена система є практично придатною, технічно завершеною, готовою до подальшого впровадження в умовах реального використання.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ktor documentation. *JetBrains*. URL: https://ktor.io (дата звернення: 30.04.2025).
2. Jetpack Compose overview. *Android Developers, Google*. URL: https://developer.android.com/jetpack/compose (дата звернення: 10.05.2025).
3. Ktorm – Elegant ORM Framework for Kotlin. URL: https://www.ktorm.org/ (дата звернення: 30.04.2025).
4. MinIO Object Storage for Kubernetes and Docker. URL: https://min.io (дата звернення: 30.04.2025).
5. Офіційна документація Android Studio та Android SDK. *Android Developers*. URL: https://developer.android.com/studio (дата звернення: 10.05.2025).
6. MySQL 8.0 Reference Manual. *Oracle Corporation*. URL: https://dev.mysql.com/doc (дата звернення: 12.03.2025).
7. Postman Learning Center. URL: https://learning.postman.com (дата звернення: 30.04.2025).
8. Kotlin Language Documentation. *JetBrains*. URL: https://kotlinlang.org/docs/home.html (дата звернення: 04.04.2025).
9. Koin – Dependency Injection for Kotlin. *Insert-Koin.io*. URL: https://insert-koin.io/ (дата звернення: 10.05.2025).
10. JWT.IO Documentation. *Auth0*. URL: https://jwt.io/introduction (дата звернення: 04.04.2025).
11. Gradle Kotlin DSL Reference. *Gradle Inc*. URL: https://docs.gradle.org/current/dsl/ (дата звернення: 10.05.2025).
12. Docker Documentation. *Docker Inc*. URL: https://docs.docker.com (дата звернення: 04.04.2025).
13. Compose Navigation Documentation. *Google Developers*. URL: https://developer.android.com/jetpack/compose/navigation (дата звернення: 10.05.2025).

14. GitHub Docs: Managing Git repositories. *GitHub*. URL: https://docs.github.com/en/get-started (дата звернення: 04.04.2025).
15. Android Permission Handling (Accompanist). *Google Samples*. URL: https://google.github.io/accompanist/permissions (дата звернення: 10.05.2025).
16. Coil – Kotlin image loader for Android backed by coroutines. URL: https://coil-kt.github.io/coil (дата звернення: 10.05.2025).

Додаток А

Фрагменти коду

```
route(«/auth») {
    post(«/login») {
        val request = call.receive<LoginRequest>()
        val user = ownerRepository.findByEmail(request.email)
        if (user == null || !verifyPassword(request.password, user.password)) {
            call.respond(HttpStatusCode.Unauthorized, «Invalid email or password»)
            return@post
        }
        val userID = user.id.toString()
        val jwtAccess = JWTConfig.generateAccessToken(userID)
        val jwtRefresh = JWTConfig.generateRefreshToken(userID)

        JWTConfig.refreshTokens[userID] = jwtRefresh

        call.respond(
            HttpStatusCode.OK, JwtTokenResponse(
                jwtAccessToken = jwtAccess,
                jwtRefreshToken = jwtRefresh
            )
        )

    }

override suspend fun allFillUps(): List<CarFillUp> {
    return useConnection { database ->
        database.from(CarFillUpTable)
```

```
                .innerJoin(CarTable, CarFillUpTable.carId eq CarTable.id)
                .innerJoin(CarBodyTable, CarTable.carBodyId eq CarBodyTable.id)
                .innerJoin(
                    CarFuelTypesTable, CarFuelTypesTable.carId eq CarTable.id
                )
                .innerJoin(
                    FuelTypeTable, FuelTypeTable.id eq
CarFuelTypesTable.fuelTypeId
                ).innerJoin(
                    FuelUnitsTable, CarFillUpTable.unitId eq FuelUnitsTable.id
                ).innerJoin(
                    OwnerTable, OwnerTable.id eq CarTable.ownerId
                ).select().mapCollection(CarFillUpTable.id, ::mergeFillUp) {
                    it.toFillUp()
                }
            }
        }

        override suspend fun updateInsurance(insuranceUpdate: InsuranceDto):
Insurance? {
            return transactionalQuery { database ->
                database.update(InsuranceTable) {
                    set(it.photoUrl, insuranceUpdate.photoUrl)
                    set(it.startDate, LocalDate.parse(insuranceUpdate.startDate))
                    set(it.endDate, LocalDate.parse(insuranceUpdate.endDate))
                    where { it.id eq insuranceUpdate.id }
                }

                findInsuranceById(insuranceUpdate.id)
```

```
        }
    }

    override suspend fun create(ownerId: Int, car: CarCreate, insurance: InsuranceCreate?) {
        transactionalQuery { database ->
            val id = database.insertAndGenerateKey(CarTable) {
                set(it.brandName, car.brandName)
                set(it.color, car.color)
                set(it.vin, car.vin)
                set(it.model, car.model)
                set(it.licensePlate, car.licensePlate)
                set(it.mileAge, car.mileAge)
                set(it.carBodyId, car.carBodyId)
                set(it.ownerId, ownerId)
            } as Int

            database.batchInsert(CarFuelTypesTable) {
                car.fuelTypes.forEach { typeId ->
                    item {
                        set(it.carId, id)
                        set(it.fuelTypeId, typeId)
                    }
                }

            }
            if (car.drivers.isNotEmpty())
                database.batchInsert(DriverWithCarTable) {
                    car.drivers.forEach { driverId ->
                        item {
```

```
                    set(it.carId, id)
                    set(it.driverId, driverId)
                    set(it.timestampStart, LocalDateTime.now())
                }
            }
        }
        if (insurance != null) {
            database.insert(InsuranceTable) {
                set(it.carId, id)
                set(it.endDate, LocalDate.parse(insurance.endDate))
                set(it.startDate, LocalDate.parse(insurance.startDate))
                set(it.photoUrl, insurance.photoUrl)
            }
        }


    }


}

@Composable
fun CarsListRoot(
    viewModel: CarsListViewModel = koinViewModel(),
    navigateBack: () -> Unit,
    navigateEdit: (Int) -> Unit,
    navigateCreate: () -> Unit
) {
    val state by viewModel.state.collectAsStateWithLifecycle()
```

```
    CarsListScreen(
        state = state,
        onAction = {
            viewModel.onAction(it)
            when (it) {
                CarsListAction.NavigateBack -> navigateBack()
                CarsListAction.NavigateCreate -> navigateCreate()
                is CarsListAction.NavigateEdit -> navigateEdit(it.id)
                else -> {}
            }
        }
    )
}

@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun CarsListScreen(
    state: CarsListState,
    onAction: (CarsListAction) -> Unit,
) {
    val pullToRefreshState = rememberPullToRefreshState()
    FleetWisorScaffold(
        topAppBar = {
            SimpleFilledAgroswitTopAppBar(
                title = stringResource(R.string.cars_text)
            ) {
                onAction(CarsListAction.NavigateBack)
            }
        },
        floatingActionButton = {
```

```
            FloatingActionButton(
                containerColor = FleetWisorTheme.colors.brandPrimaryNormal,
                modifier = Modifier.size(64.dp),
                onClick = {
                    onAction(CarsListAction.NavigateCreate)
                },
                shape = CircleShape
            ) {
                Icon(
                    modifier = Modifier.size(48.dp),
                    painter = FleetWisorTheme.icons.plus,
                    tint = FleetWisorTheme.colors.brandSecondaryNormal,
                    contentDescription = ««
                )
            }
        },
        hasBottomBar = true,
        floatingActionButtonPosition = FabPosition.End,
    ) { paddingValues ->
        Column(
            modifier = Modifier
                .padding(
                    top = paddingValues.calculateTopPadding() + 20.dp,
                    bottom = paddingValues.calculateBottomPadding()
                )
                .padding(horizontal = 20.dp)
        ) {
            TextFieldAgroswit(
                icon = FleetWisorTheme.icons.search,
                value = state.searchValue,
```

```
                hint = stringResource(R.string.search_text),
                onValueChange = {
                    onAction(CarsListAction.InputSearch(it))
                }
            )
            PullToRefreshBox(
                state = pullToRefreshState,
                isRefreshing = state.isLoading,
                onRefresh = {
                    onAction(CarsListAction.Refresh)
                },
                indicator = {
                    Indicator(
                        modifier = Modifier.align(Alignment.TopCenter),
                        isRefreshing = state.isLoading,
                        containerColor = FleetWisorTheme.colors.brandPrimaryNormal,
                        color = FleetWisorTheme.colors.brandSecondaryNormal,
                        state = pullToRefreshState
                    )

                }
            ) {
                LazyColumn(verticalArrangement = Arrangement.spacedBy(16.dp), modifier = Modifier.fillMaxSize()) {
                    item {
                        Spacer(Modifier)
                    }
                    items(state.carsFilter) { car ->
                        CarListItem(
```

```
                        modifier = Modifier
                            .fillMaxWidth()
                            .clickable {
                                onAction(CarsListAction.NavigateEdit(car.id))
                            },
                        title = car.name,
                        firstText = car.mileAge.toString() + « км»,
                        secondText = car.licensePlate
                    )
                }
            }
        }
    }
}

@Preview
@Composable
private fun Preview() {
    FleetWisorTheme {
        CarsListScreen(
            state = CarsListState(),
            onAction = {}
        )
    }
}

class CarsListViewModel(
    private val repository: CarRepository
) : ViewModel() {
```

```
private var hasLoadedInitialData = false

private val _state = MutableStateFlow(CarsListState())
val state = _state
    .onStart {
        if (!hasLoadedInitialData) {
            init()
            hasLoadedInitialData = true
        }
    }
    .stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed(5_000L),
        initialValue = CarsListState()
    )

fun onAction(action: CarsListAction) {
    when (action) {
        is CarsListAction.InputSearch -> {
            _state.update {
                it.copy(
                    searchValue = action.value,
                    carsFilter = it.cars.filter { it.name.contains(action.value, true) }
                )
            }
        }

        CarsListAction.Refresh -> {
            init()
```

```
            }

            else -> {}
        }
    }

    private fun init() {
        _state.update { it.copy(isLoading = true) }
        viewModelScope.launch(Dispatchers.IO) {
            when (val res = repository.getAll()) {
                is FullResult.Error -> _state.update {
                    it.copy(
                        error = res.asErrorUiText()
                    )
                }

                is FullResult.Success -> _state.update {
                    it.copy(
                        cars = res.data,
                        carsFilter = res.data
                    )
                }

            }
            _state.update { it.copy(isLoading = false) }
        }
    }

}
```

```
    override suspend fun saveCar(
        value: CarCreate,
        insurance: InsuranceDto?,
        photo: Pair<String, ByteArray>?
    ): EmptyDataAndErrorResult<DataError.Network> {
        val fileParts = buildList {
            if (photo != null)
                add(
                    FilePart(
                        formPart = FormPart(
                            key = «photo»,
                            value = photo.second,
                            headers = Headers.build {
                                append(HttpHeaders.ContentType, photo.first)
                                append(
                                    HttpHeaders.ContentDisposition,
                                    «form-data; name=\»photo\»; filename=\»photo.jpg\»«
                                )

                            }
                        )
                    )
                )
        }
        return httpClientFactory.getClient()
            .postMultiPart(postCar, CarCreateWithInsurance(value, insurance),
fileParts)

    }
```

```
suspend inline fun <reified Request : Any, reified Response : Any, reified Error>
HttpClient.postMultiPart(
    route: String,
    body: Request,
    fileParameters: List<FilePart> = listOf()
): FullResult<Response, DataError.Network, Error> {
    return safeCall {
        post {
            url(constructRoute(route))
            constructMultiPart(fileParameters = fileParameters, body = body)
        }
    }
}
```

Додаток Б

SQL-структура бази даних

```
CREATE TABLE IF NOT EXISTS owner
(
  id       INT PRIMARY KEY AUTO_INCREMENT,
  name     VARCHAR(255)       NOT NULL,
  surname  VARCHAR(255)       NOT NULL,
  email    VARCHAR(254) UNIQUE NOT NULL,
  password VARCHAR(254)       NOT NULL
);

CREATE TABLE IF NOT EXISTS driver
(
  id                INT PRIMARY KEY AUTO_INCREMENT,
  ownerId           INT             NOT NULL,
  driverLicenseNumber  VARCHAR(50) UNIQUE NOT NULL,
  name              VARCHAR(255)      NOT NULL,
  surname           VARCHAR(255)      NOT NULL,
  phone             VARCHAR(20) UNIQUE NOT NULL,
  frontLicensePhotoUrl VARCHAR(255)      NOT NULL,
  backLicensePhotoUrl  VARCHAR(255)      NOT NULL,
  salary            FLOAT(8)          NOT NULL,
  birthday          DATE              NOT NULL,
  FOREIGN KEY (ownerId) REFERENCES owner (id) ON DELETE
CASCADE
);

CREATE TABLE IF NOT EXISTS car_body
(
```

```
    id     INT PRIMARY KEY AUTO_INCREMENT,
    nameUk VARCHAR(20) NOT NULL UNIQUE,
    nameEn VARCHAR(20) NOT NULL UNIQUE
);

CREATE TABLE IF NOT EXISTS car
(
    id          INT PRIMARY KEY AUTO_INCREMENT,
    brandName   VARCHAR(20)       NOT NULL,
    ownerId     INT               NOT NULL,
    carBodyId   INT               NOT NULL,
    color       VARCHAR(20),
    vin         VARCHAR(18) UNIQUE NULL,
    mileage     BIGINT DEFAULT 0,
    model       VARCHAR(20),
    licensePlate VARCHAR(20) UNIQUE,
    FOREIGN KEY (ownerId) REFERENCES owner (id) ON DELETE
CASCADE,
    FOREIGN KEY (carBodyId) REFERENCES car_body (id) ON DELETE
CASCADE
);

CREATE TABLE IF NOT EXISTS driver_with_car
(
    id       INT PRIMARY KEY AUTO_INCREMENT,
    carId    INT      NOT NULL,
    driverId INT      NOT NULL,
    timeStart DATETIME NOT NULL,
    timeEnd   DATETIME NULL,
    FOREIGN KEY (carId) REFERENCES car (id) ON DELETE CASCADE,
```

```
    FOREIGN KEY (driverId) REFERENCES driver (id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS maintenance
(
  id          INT PRIMARY KEY AUTO_INCREMENT,
  carId       INT          NOT NULL,
  description VARCHAR(2000) NOT NULL,
  price       FLOAT(8)     NOT NULL,
  time        DATETIME     NOT NULL,
  checkUrl    VARCHAR(255),
  FOREIGN KEY (carId) REFERENCES car (id) ON DELETE CASCADE
);


CREATE TABLE IF NOT EXISTS fuel_type
(
  id     INT PRIMARY KEY AUTO_INCREMENT,
  nameUk VARCHAR(20) NOT NULL UNIQUE,
  nameEn VARCHAR(20) NOT NULL UNIQUE
);

CREATE TABLE fuel_units
(
  id        INT PRIMARY KEY AUTO_INCREMENT,
  nameUk    VARCHAR(20) NOT NULL,
  nameEn    VARCHAR(20) NOT NULL,
  fuelTypeId INT        NOT NULL,
  FOREIGN KEY (fuelTypeId) REFERENCES fuel_type (id)
```

```
);

CREATE TABLE owner_fuel_units
(
   ownerId    INT NOT NULL,
   unitId     INT NOT NULL,
   fuelTypeId INT NOT NULL,
   PRIMARY KEY (ownerId, fuelTypeId),
   FOREIGN KEY (unitId) REFERENCES fuel_units (id) ON DELETE
CASCADE,
   FOREIGN KEY (fuelTypeId) REFERENCES fuel_type (id) ON DELETE
CASCADE,
   FOREIGN KEY (ownerId) REFERENCES owner (id) ON DELETE
CASCADE
);


CREATE TABLE IF NOT EXISTS car_fuel_types
(
   fuelTypeId INT NOT NULL,
   carId      INT NOT NULL,
   PRIMARY KEY (fuelTypeId, carId),
   FOREIGN KEY (fuelTypeId) REFERENCES fuel_type (id) ON DELETE
CASCADE,
   FOREIGN KEY (carId) REFERENCES car (id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS insurance
(
   id       INT PRIMARY KEY AUTO_INCREMENT,
```

```
    carId     INT  NOT NULL,
    startDate DATE NOT NULL,
    endDate   DATE NOT NULL,
    photoUrl  VARCHAR(255),
    FOREIGN KEY (carId) REFERENCES car (id) ON DELETE CASCADE
);

CREATE TABLE IF NOT EXISTS car_fill_up
(
    id       INT PRIMARY KEY AUTO_INCREMENT,
    carId    INT      NOT NULL,
    time     DATETIME NOT NULL,
    price    FLOAT(8) NOT NULL,
    amount   FLOAT(8) NOT NULL,
    unitId   INT      NOT NULL,
    checkUrl VARCHAR(255),
    FOREIGN KEY (carId) REFERENCES car (id) ON DELETE CASCADE,
    FOREIGN KEY (unitId) REFERENCES fuel_units (id) ON DELETE
CASCADE
);
INSERT INTO car_body (nameUk, nameEn)
VALUES ('Седан', 'Sedan'),
       ('Хетчбек', 'Hatchback'),
       ('Універсал', 'Station Wagon'),
       ('Купе', 'Coupe'),
       ('Кабріолет', 'Convertible'),
       ('Позашляховик', 'SUV'),
       ('Кросовер', 'Crossover'),
       ('Мінівен', 'Minivan'),
       ('Пікап', 'Pickup'),
```

```
    ('Інше', 'Other'),
    ('Фургон', 'Van');

INSERT INTO fuel_type (nameUk, nameEn)
VALUES ('Бензин', 'Petrol'),
    ('Дизель', 'Diesel'),
    ('Газ', 'Gas'),
    ('Електрика', 'Electricity'),
    ('Гібрид', 'Hybrid'),
    ('Інше', 'Other'),
    ('Водень', 'Hydrogen');


INSERT INTO fuel_units (nameUk, nameEn, fuelTypeId)
VALUES
-- Бензин (id = 1)
('л', 'L', 1),
('гал США', 'US gal', 1),
('імп гал', 'Imp gal', 1),

-- Дизель (id = 2)
('л', 'L', 2),
('гал США', 'US gal', 2),
('імп гал', 'Imp gal', 2),

-- Газ (id = 3)
('л', 'L', 3),
('кг', 'kg', 3),
('гал США', 'US gal', 3),
('нм³', 'Nm³', 3),
```

```
-- Електрика (id = 4)
('кВт·год', 'kWh', 4),
('МВт·год', 'MWh', 4),
('A', 'A', 4),
('В', 'V', 4),

-- Гібрид (id = 5)
('л', 'L', 5),
('гал США', 'US gal', 5),
('імп гал', 'Imp gal', 5),
('кВт·год', 'kWh', 5),

-- Інше (id = 6)
('од', 'unit', 6),

-- Водень (id = 7)
('кг', 'kg', 7),
('нм³', 'Nm³', 7),
('МПа', 'MPa', 7);
```

Додаток В

Файли для Docker та файл-приклад параметрів середовища

```
services:
app:
build:
context: .
dockerfile: Dockerfile
volumes:
- .:/home/gradle/app:cached
working_dir: /home/gradle/app
env_file:
- .env
ports:
- «8080:8080»
networks:
- fleet-wisor-network
depends_on:
- mysql
- minio
mysql:
image: mysql:8.0
container_name: mysql-container
env_file:
- .env
environment:
MYSQL_ROOT_PASSWORD: ${DB_PASSWORD}
MYSQL_DATABASE: fleet_wisor_dev
ports:
```

- «3306:3306»
volumes:
- db_data:/var/lib/mysql
networks:
- fleet-wisor-network
healthcheck:
test: [«CMD», «mysql»,
«
-h», «localhost»,
«
-u», «root»,
«
- p${MYSQL_ROOT_PASSWORD}»,
«
-e», «SELECT 1»]
timeout: 20s
retries: 5
minio:
image: minio/minio:latest
container_name: minio
ports:
- «9000:9000»
- «9001:9001»
environment:
MINIO_ROOT_USER: ${MINIO_ROOT_USER}
MINIO_ROOT_PASSWORD: ${MINIO_ROOT_PASSWORD}
command: server /data --console-address «:9001»
networks:
- fleet-wisor-network
healthcheck:

```
test: [ «CMD», «curl»,
«
-f», «http://localhost:9000/minio/health/live»
]
timeout: 20s

FROM gradle:latest
WORKDIR /home/gradle/app
CMD [«gradle»,»-t», «run»,»--no-daemon»]

DB_URL=
DEBUG=
DB_USER=
DB_PASSWORD=
JWT_SECRET=
MINIO_URL=
MINIO_ROOT_USER=
MINIO_ROOT_PASSWORD=
```