Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет інформаційних систем та програмної інженерії
(повна назва факультету)
Кафедра програмної інженерії
(повна назва кафедри)

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня

магістр
(назва освітнього ступеня)

на тему: “Розробка багатоплатформеного застосунку для моніторингу стану та витрат обслуговування автомобіля”

Виконав(ла): студент(ка) 6 курсу групи СПм-61
спеціальності 121
«Інженерія програмного забезпечення»
(шифр і назва спеціальності)

(підпис) Тихович Б. П. (прізвище та ініціали)

Керівник (підпис) Петрик М. Р. (прізвище та ініціали)

Нормоконтроль (підпис) Стоянов Ю. А. (прізвище та ініціали)

Завідувач кафедри (підпис) Петрик М. Р. (прізвище та ініціали)

Рецензент (підпис) (прізвище та ініціали)

Тернопіль
2025

Міністерство освіти і науки України
Тернопільський національний технічний університет імені Івана Пулюя

Факультет комп'ютерно-інформаційних систем і програмної інженерії
(повна назва факультету)

Кафедра програмної інженерії
(повна назва кафедри)

		ЗАТВЕРДЖУЮ		
		Завідувач кафедри		
		(підпис)		(прізвище та ініціали)
		« »		20___ р.

З А В Д А Н Н Я
НА КВАЛІФІКАЦІЙНУ РОБОТУ

на здобуття освітнього ступеня магістр
(назва освітнього ступеня)

за спеціальністю 121 Інженерія програмного забезпечення
(шифр і назва спеціальності)

студенту Тихович Богдан Петрович
(прізвище, ім'я, по батькові)

1. Тема роботи Розробка багатоплатформенного застосунку для моніторингу стану та витрат обслуговування автомобіля

Керівник роботи Петрик Михайло Романович
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

Затверджені наказом ректора від «___» _____ 20___ року № _____

2. Термін подання студентом завершеної роботи

3. Вихідні дані до роботи

4. Зміст роботи (перелік питань, які потрібно розробити)

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень, слайдів)

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання _____

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1.	Вибір та погодження теми з керівником.	16.06.2025-	виконано
	Створення та затвердження технічного завдання.	24.06.2025	
2.	Аналіз технічного завдання, підбір та аналіз	25.06.2025-	виконано
	бібліографічних матеріалів необхідних для виконання	01.07.2025	
	кваліфікаційної роботи.		
3.	Формування аналізу вимог.	28.09.2025	виконано
4.	Проєктування та розробка програмного забезпечення.	18.11.2025	виконано
5.	Тестування та дослідна експлуатація програмного	01.12.2025	виконано
	забезпечення.		
6.	Апробація результатів роботи (участь у конференції).	17-18.12.2025	виконано
7.	Перевірка програмою антиплагіат.	03.12.2025	виконано
8.	Охорона праці.	.12.2025	виконано
9.	Безпека у надзвичайних ситуаціях.	.12.2025	виконано
10.	Формування записки кваліфікаційної роботи.	.12.2025	виконано
11.	Нормоконтроль.	.12.2025	виконано
12.	Попередній захист.	.12.2025	виконано
13.	Рецензування кваліфікаційної роботи.	.12.2025	виконано
14.	Захист кваліфікаційної роботи.	.12.2025	виконано

Студент _____ Тихович Б.П
(підпис) (прізвище та ініціали)

Керівник роботи _____ _____
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Тихович Б. П. Розробка багатоплатформеного застосунку для моніторингу стану та витрат обслуговування автомобіля. Магістерська кваліфікаційна робота Тернопільський національний технічний університет імені Івана Пулюя. Факультет комп'ютерно-інформаційних систем та програмної інженерії, група СПм-61 Тернопіль, ТНТУ – 2025 р., сторінок – 101 , джерел – 45, рисунків – 33 , таблиць – 1.

Ключові слова: багатоплатформенний застосунок, автомобіль, моніторинг витрат, технічне обслуговування, Ionic, React, NestJS, PostgreSQL, PWA, UML, тестування.

Магістерська кваліфікаційна робота присвячена розробці багатоплатформенного програмного застосунку для моніторингу стану автомобіля та обліку витрат на його обслуговування. Актуальність теми зумовлена необхідністю автоматизації процесів ведення історії експлуатації транспортних засобів, контролю фінансових витрат і підвищення зручності користування подібними системами в умовах зростаючої мобільності користувачів.

Метою роботи є проєктування та реалізація програмної системи, яка забезпечує зручний облік витрат, технічного обслуговування, пробігу автомобіля, а також надає аналічну інформацію та статистику для підтримки прийняття рішень. У процесі виконання роботи проведено аналіз предметної області, сформульовано функціональні та нефункціональні вимоги, визначено основних акторів і сценарії використання системи.

У роботі спроєктовано архітектуру клієнтської та серверної частин застосунку з використанням сучасних підходів і технологій. Серверна частина реалізована на базі Node.js та NestJS із використанням реляційної бази даних PostgreSQL, а клієнтська частина з використанням Ionic React, що забезпечує кросплатформенність та можливість розгортання у вигляді PWA. Для моделювання системи застосовано UML-діаграми класів і діаграми послідовностей.

Окрему увагу приділено реалізації та тестуванню основного функціоналу системи. Проведено ручне та модульне тестування серверної частини, що

підтвердило коректність роботи реалізованих модулів і відповідність системи початково визначеним вимогам. Отримані результати можуть бути використані для подальшого розвитку програмного продукту та його впровадження у практичну експлуатацію.

ABSTRACT

Tykhovych B. P. Development of a cross-platform application for monitoring vehicle condition and maintenance costs. Master's qualification thesis. Ternopil Ivan Puluj National Technical University. Faculty of Computer and Information Systems and Software Engineering, group SPm-61, Ternopil, TNTU – 2025, pages – 101, references – 45 , figures – 33, tables – 1.

Keywords: cross-platform application, vehicle, cost monitoring, maintenance, Ionic, React, NestJS, PostgreSQL, PWA, UML, testing.

The master's qualification thesis is devoted to the development of a cross-platform software application for monitoring the technical condition of a vehicle and tracking maintenance-related expenses. The relevance of the topic is determined by the need to automate the processes of maintaining vehicle operation history, controlling financial expenses, and improving user convenience in conditions of increasing user mobility.

The aim of the work is to design and implement a software system that provides convenient accounting of expenses, maintenance activities, and vehicle mileage, as well as analytical information and statistics to support decision-making. In the course of the work, the subject area was analyzed, functional and non-functional requirements were defined, and the main actors and system use cases were identified.

The architecture of the client and server parts of the application was designed using modern approaches and technologies. The server side was implemented using Node.js and NestJS with a relational PostgreSQL database, while the client side was developed using Ionic React, which ensures cross-platform compatibility and the possibility of deployment as a Progressive Web Application (PWA). UML class diagrams and sequence diagrams were used to model the system.

Special attention was paid to the implementation and testing of the core system functionality. Manual and unit testing of the server-side components was conducted, confirming the correctness of the implemented modules and compliance of the system with the initially defined requirements. The obtained results can be used for further development of the software product and its practical deployment.

ЗМІСТ

ВСТУП

Стрімкий розвиток інформаційних технологій та широке поширення мобільних пристроїв суттєво впливають на способи взаємодії користувачів з цифровими сервісами у повсякденному житті. Однією зі сфер, що активно потребує сучасних програмних рішень, є експлуатація та обслуговування автомобілів. Зростання кількості транспортних засобів, підвищення вартості їх утримання та необхідність регулярного технічного контролю зумовлюють потребу у систематизованому підході до обліку витрат, технічного обслуговування та аналізу стану автомобіля.

Актуальність теми полягає в тому, що більшість наявних рішень не забезпечують комплексного підходу до ведення історії автомобіля, обліку витрат і надання аналітичної інформації в зручному та доступному форматі. Часто користувачі змушені використовувати різні інструменти або вести облік вручну, що ускладнює аналіз даних та підвищує ймовірність помилок. У зв’язку з цим виникає необхідність у створенні багатоплатформенного застосунку, який поєднує функції обліку, аналізу та моніторингу в єдиній системі.

Кваліфікаційна робота магістра на тему «Розробка багатоплатформеного застосунку для моніторингу стану та витрат обслуговування автомобіля» присвячена створенню програмного продукту, що дозволяє користувачеві вести детальну історію витрат і обслуговування автомобіля, аналізувати фінансові показники за різні періоди та отримувати узагальнену статистичну інформацію. Особливу увагу приділено зручності користування, доступності з різних пристроїв та можливості подальшого масштабування системи.

У межах роботи реалізовано клієнтську частину застосунку з використанням фреймворку Ionic React, що забезпечує кросплатформенність і можливість запуску як на мобільних пристроях, так і у вебсередовищі. Серверна частина розроблена на основі Node.js та NestJS, що дозволяє побудувати модульну, масштабовану та легко підтримувану архітектуру. Для зберігання даних використовується реляційна база

даних PostgreSQL, а для взаємодії з нею застосовано ORM-інструменти, що спрощують роботу з даними та підвищують надійність системи.

Окремим аспектом роботи є інтеграція елементів штучного інтелекту, зокрема використання ШІ-сервісу для розпізнавання інформації з фотографій чеків. Це дозволяє автоматизувати процес введення витрат, зменшити навантаження на користувача та підвищити точність даних. Реалізований підхід поєднує обробку зображень, аналіз тексту та валідацію отриманої інформації на серверному рівні.

Метою магістерської роботи є проєктування та реалізація багатоплатформенного застосунку для моніторингу стану автомобіля та витрат на його обслуговування, який відповідає функціональним і нефункціональним вимогам, забезпечує зручний користувацький інтерфейс, надійне зберігання даних і можливості аналітичної обробки інформації. Для досягнення поставленої мети виконано аналіз предметної області, сформовано вимоги до системи, спроєктовано її архітектуру, реалізовано основний функціонал та проведено тестування.

Об’єктом дослідження є процеси розробки програмних систем для моніторингу та обліку експлуатаційних даних автомобіля. Предметом дослідження є методи та технології створення багатоплатформенних застосунків із використанням сучасних клієнтських і серверних фреймворків, баз даних та засобів автоматизованого і ручного тестування.

Таким чином, дана магістерська робота спрямована на створення сучасного, функціонального та масштабованого програмного рішення, що підвищує ефективність обліку та аналізу витрат на обслуговування автомобіля. У подальших розділах роботи детально розглянуто аналіз вимог і предметної області, процес проєктування системи, реалізацію основного функціоналу та результати тестування розробленого застосунку

1 АНАЛІЗ ВИМОГ ТА ПРЕДМЕТНОЇ ОБЛАСТІ

У даному розділі проводиться аналіз предметної області, пов’язаної з експлуатацією автомобілів та обліком витрат на їх обслуговування. Визначаються основні проблеми, які виникають у користувачів під час контролю технічного стану автомобіля, а також формулюються цілі та вимоги до програмної системи. Результати аналізу слугують основою для подальшого проєктування застосунку.

1.1 Аналіз предметної області

Сучасна експлуатація автомобіля супроводжується значною кількістю інформації, пов’язаної з його технічним станом, пробігом, витратами на обслуговування та фінансовими операціями протягом усього життєвого циклу транспортного засобу. Власники автомобілів змушені зберігати ці дані у різних формах, зокрема у паперових документах, електронних таблицях або розрізнених мобільних застосунках, що ускладнює цілісне бачення історії автомобіля та прийняття обґрунтованих рішень щодо його обслуговування або продажу.

Аналіз предметної області показує, що на ринку програмного забезпечення існують рішення, орієнтовані на окремі аспекти експлуатації автомобіля, зокрема облік витрат пального, сервісні нагадування або базову діагностику. Проте більшість таких застосунків не забезпечують комплексного підходу до ведення повної історії автомобіля, не підтримують можливість роботи з декількома транспортними засобами та обмежено враховують фінансову складову володіння автомобілем.

Особливо актуальною є проблема централізованого зберігання історії автомобіля, яка включає дані про пробіг, витрати, виконані роботи з обслуговування, а також інформацію про купівлю та продаж транспортного засобу. Відсутність структурованої цифрової історії ускладнює аналіз фактичних витрат, знижує прозорість стану автомобіля та ускладнює процес передавання інформації

іншому власнику або кільком користувачам, які спільно експлуатують транспортний засіб.

Окремою складовою предметної області є процес введення та обробки даних. Ручне внесення інформації є трудомістким і схильним до помилок, що знижує мотивацію користувачів вести повний облік. У зв’язку з цим актуальним є застосування інструментів автоматизації збору даних, зокрема використання технологій розпізнавання інформації з фотографій чеків і документів із подальшим перетворенням отриманих результатів у структуровані записи [41].

Аналіз також показує зростаючу потребу користувачів у багатоплатформенних програмних рішеннях, які забезпечують доступ до даних незалежно від типу пристрою або операційної системи. Використання веб- та мобільних застосунків дозволяє забезпечити безперервний доступ до інформації про автомобіль, що є важливим фактором зручності та практичності системи.

Таким чином, предметна область розроблюваної системи охоплює процеси збирання, зберігання та аналізу інформації про експлуатацію автомобіля, фінансовий облік витрат, управління історією одного або декількох транспортних засобів, а також забезпечення спільного доступу до даних. Виявлені особливості предметної області підтверджують актуальність створення багатоплатформенного застосунку, який поєднує зазначені функції в єдиному програмному рішенні та створює основу для подальшого розширення функціональних можливостей системи.

1.2 Постановка задачі та цілей розробки

У сучасних умовах експлуатації автомобілів власники стикаються з великою кількістю інформації, що відображає технічний стан транспортного засобу, історію обслуговування, пробіг, витрати на експлуатацію та фінансові операції. Відсутність централізованого цифрового інструменту для збирання, зберігання та аналізу таких даних призводить до розпорошеності інформації,

низької прозорості витрат, неможливості комплексного контролю технічного стану та складнощів при передаванні історії автомобіля іншим користувачам.

Основна задача цієї роботи полягає у створенні багатоплатформенного застосунку, який забезпечує інтегрований підхід до ведення цифрової історії автомобіля. Система повинна дозволяти зберігати дані про пробіг, витрати та обслуговування автомобіля, вести фінансовий облік, надавати можливість роботи з декількома автомобілями одночасно та забезпечувати спільний доступ до даних різним користувачам у безпечному середовищі [20].

Метою цієї роботи є створення багатоплатформенного програмного рішення, яке:

- централізує інформацію про стан та експлуатацію автомобіля;
- забезпечує структуроване зберігання даних про пробіг, витрати, обслуговування та фінанси;
- дозволяє користувачу вести історію одного або декількох автомобілів;
- забезпечує безпечний доступ і спільну роботу з даними;
- створює основу для подальшого розширення функціоналу, включаючи інтеграцію з додатковими сервісами, наприклад маркетплейсом автотоварів.

Для досягнення поставленої мети необхідно виконати такі завдання:

- провести аналіз предметної області та потреб користувачів у веденні історії автомобіля, визначити проблемні аспекти існуючих рішень;
- визначити функціональні та нефункціональні вимоги до програмної системи з урахуванням багатоплатформенності, зручності використання та безпеки даних;
- виявити ключових акторів системи та сценарії використання для забезпечення повного охоплення функціоналу;
- спроєктувати архітектуру застосунку, включаючи клієнтську та серверну частини, з можливістю подальшого розширення та інтеграції додаткових модулів;

- розробити клієнтську частину системи з використанням Ionic React для забезпечення сумісності з веб-браузерами та мобільними платформами Android і iOS;
- реалізувати механізми введення даних про автомобіль вручну та з використанням завантажених зображень документів з подальшим аналізом ШІ;
- забезпечити функціонал фінансового обліку, включаючи реєстрацію вартості придбання, витрат та даних про продаж автомобіля;
- реалізувати безпеку системи: автентифікацію користувачів, контроль доступу до даних, захист передавання інформації між клієнтом та сервером;
- реалізувати комплексні механізми безпеки системи, зокрема автентифікацію користувачів із використанням сервісів Firebase, контроль доступу до даних автомобіля та захист передавання інформації між клієнтською і серверною частинами за допомогою захищених API [33, 34, 35, 36];
- провести тестування системи на відповідність функціональним та нефункціональним вимогам, перевірити коректність роботи, стабільність та зручність використання.

Очікувані результати

- Розроблений багатоплатформений застосунок, що дозволяє вести цифрову історію автомобіля;
- Можливість роботи з декількома автомобілями та спільного доступу до даних;
- Структуроване зберігання даних про пробіг, витрати та обслуговування автомобіля;
- Безпечний облік фінансових операцій та контроль доступу користувачів;
- Основа для подальшого розвитку системи та інтеграції додаткових аналітичних і сервісних модулів.

Створення такого застосунку дозволяє підвищити ефективність ведення обліку експлуатації автомобіля, зменшити ризик упущення важливих даних,

забезпечити прозорість фінансових витрат та спростити процес передачі історії автомобіля. Багатоплатформовість і інтеграція сучасних технологій обробки даних забезпечує широку доступність системи та її практичну цінність для користувачів.

1.3 Визначення функціональних та нефункціональних вимог

Функціональні вимоги:

- забезпечувати реєстрацію користувачів у застосунку;
- підтримувати автентифікацію користувачів та вхід до облікового запису;
- зберігати персональні налаштування користувача;
- надавати можливість додавання одного або декількох автомобілів;
- зберігати базові характеристики автомобіля (марка, модель, рік випуску);
- забезпечувати ручне введення пробігу автомобіля;
- забезпечувати ручне введення витрат на експлуатацію автомобіля;
- забезпечувати ручне введення даних про технічне обслуговування;
- зберігати повну історію експлуатації автомобіля;
- надавати можливість завантаження фотографій чеків та документів;
- виконувати автоматичну обробку зображень із використанням алгоритмів штучного інтелекту;
- формувати структуровані записи на основі результатів аналізу зображень;
- дозволяти користувачу редагувати автоматично згенеровані дані;
- вести облік вартості придбання автомобіля;
- здійснювати фінансовий облік витрат за категоріями;
- зберігати інформацію про продаж автомобіля;
- обчислювати сумарні витрати на експлуатацію автомобіля;
- надавати можливість спільного доступу до автомобіля для кількох користувачів;
- забезпечувати передавання історії автомобіля іншому користувачу;

- забезпечувати розмежування прав доступу користувачів;
- підтримувати можливість подальшого розширення функціоналу системи.

Нефункціональні вимоги

- бути багатоплатформенною та працювати у веб-середовищі;
- забезпечувати роботу на мобільних пристроях з Android та iOS;
- використовувати фреймворк Ionic React для клієнтської частини [19, 37];
- мати інтуїтивно зрозумілий інтерфейс користувача;
- забезпечувати зручну навігацію між основними розділами;
- забезпечувати захист персональних даних користувачів;
- підтримувати автентифікацію та контроль доступу;
- використовувати захищені канали передавання даних;
- забезпечувати стабільну роботу при збільшенні обсягу даних;
- вебінтерфейс повинен коректно функціонувати в основних браузерах (Google Chrome, Edge, Opera, Firefox тощо) ;
- система має підтримувати розгортання на різних серверах або у хмарних середовищах (Docker, Kubernetes, cloud-hosting) без втрати функціональності чи даних [42];
- підтримувати масштабування без втрати продуктивності.

Сформульовані функціональні та нефункціональні вимоги визначають основні можливості, обмеження та якісні характеристики розроблюваної програмної системи. Вони є основою для подальшого проєктування архітектури, розробки програмних компонентів та проведення тестування з метою перевірки відповідності реалізованого функціоналу поставленим вимогам.

1.4 Визначення акторів та варіантів використання системи

Для коректного проєктування програмної системи та визначення її функціональних можливостей важливим етапом є ідентифікація акторів і варіантів використання. Актори відображають основні ролі користувачів і зовнішніх

сервісів, які взаємодіють із системою, тоді як варіанти використання описують типові сценарії взаємодії з функціоналом застосунку. Визначення акторів та їх ролей дозволяє формалізувати вимоги до системи, забезпечити чітке розмежування прав доступу та слугує основою для подальшого проєктування архітектури і UML-діаграм.

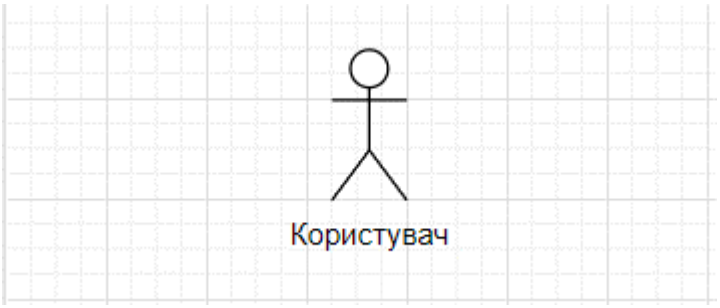


Рисунок 1.1 – Основні актори системи

Короткий опис акторів представлений в таблиці 1.1.

Таблиця 1.1 – Визначення акторів

Актор	Короткий опис
Користувач	Має можливість реєструватися та проходити аутентифікацію у системі. Також має можливість завантажувати файли у сховище та зі сховища, переглядати елементи файлової системи, переглядати детальну інформацію про елементи файлової системи, редагувати дані елементів файлової системи, видаляти елементи файлової системи.
Сервіс штучного інтелекту	Виконує аналіз завантажених користувачем зображень чеків та документів з метою автоматичного вилучення структурованих даних, які передаються до системи для подальшої обробки та збереження.

Визначення варіантів використання представлені таблиці 1.2.

Таблиця 1.2 – Варіанти використання

Актор	Варіант використання	Опис
Користувач	Реєстрація	Користувач має можливість створити особистий обліковий запис у системі шляхом введення електронної адреси та пароля. Після успішної реєстрації створюється профіль користувача та ініціалізуються базові налаштування системи.
Користувач	Авторизація	Користувач виконує вхід до системи з використанням власних облікових даних для отримання доступу до персонального функціоналу та збережених даних.
Користувач	Додавання автомобіля	Користувач має можливість додати новий автомобіль до системи, вказавши його основні характеристики, такі як марка, модель, рік випуску та початковий пробіг.
Користувач	Перегляд списку автомобілів	Користувач може переглядати перелік усіх автомобілів, які прив’язані до його облікового запису.
Користувач	Введення пробігу	Користувач має можливість вручну додавати актуальні показники пробігу автомобіля з фіксацією дати внесення даних.
Користувач	Облік витрат	Користувач може вносити інформацію про витрати на автомобіль, зазначаючи суму, категорію витрат та дату.

Продовження таблиці 1.2

1	2	3
Користувач	Облік технічного обслуговування	Користувач має можливість фіксувати проведені роботи з технічного обслуговування автомобіля із зазначенням типу робіт та пробігу.
Користувач	Завантаження фото документів	Користувач може додавати фотографії чеків або документів, пов’язаних з обслуговуванням автомобіля.
Користувач	Перегляд історії автомобіля	Користувач має доступ до повної історії експлуатації автомобіля, включаючи пробіг, витрати та обслуговування.
Користувач	Надання спільного доступу	Користувач може надати іншому користувачу доступ до даних конкретного автомобіля з визначеним рівнем прав.
Користувач	Передавання історії автомобіля	Користувач має можливість передати повну історію автомобіля іншому користувачу системи.
Користувач	Перегляд фінансової статистики	Користувач може переглядати узагальнену інформацію про загальні витрати на автомобіль та їх розподіл за категоріями.
Система штучного інтелекту	Аналіз зображень	Зовнішній сервіс штучного інтелекту виконує аналіз завантажених зображень чеків та документів з метою вилучення структурованих даних.

За допомогою діаграми варіантів використання (Рис. 1.2) формалізовано взаємодію користувача з програмною системою та визначено основні сценарії її експлуатації. Дана діаграма дозволяє на концептуальному рівні відобразити функціональні можливості системи з точки зору кінцевого користувача, не заглиблюючись у деталі реалізації. Використання діаграми варіантів використання

сприяє чіткому структуруванню функціональних вимог, визначенню меж системи та взаємозв’язків між окремими сценаріями роботи [10].

Розробка варіантів використання системи.

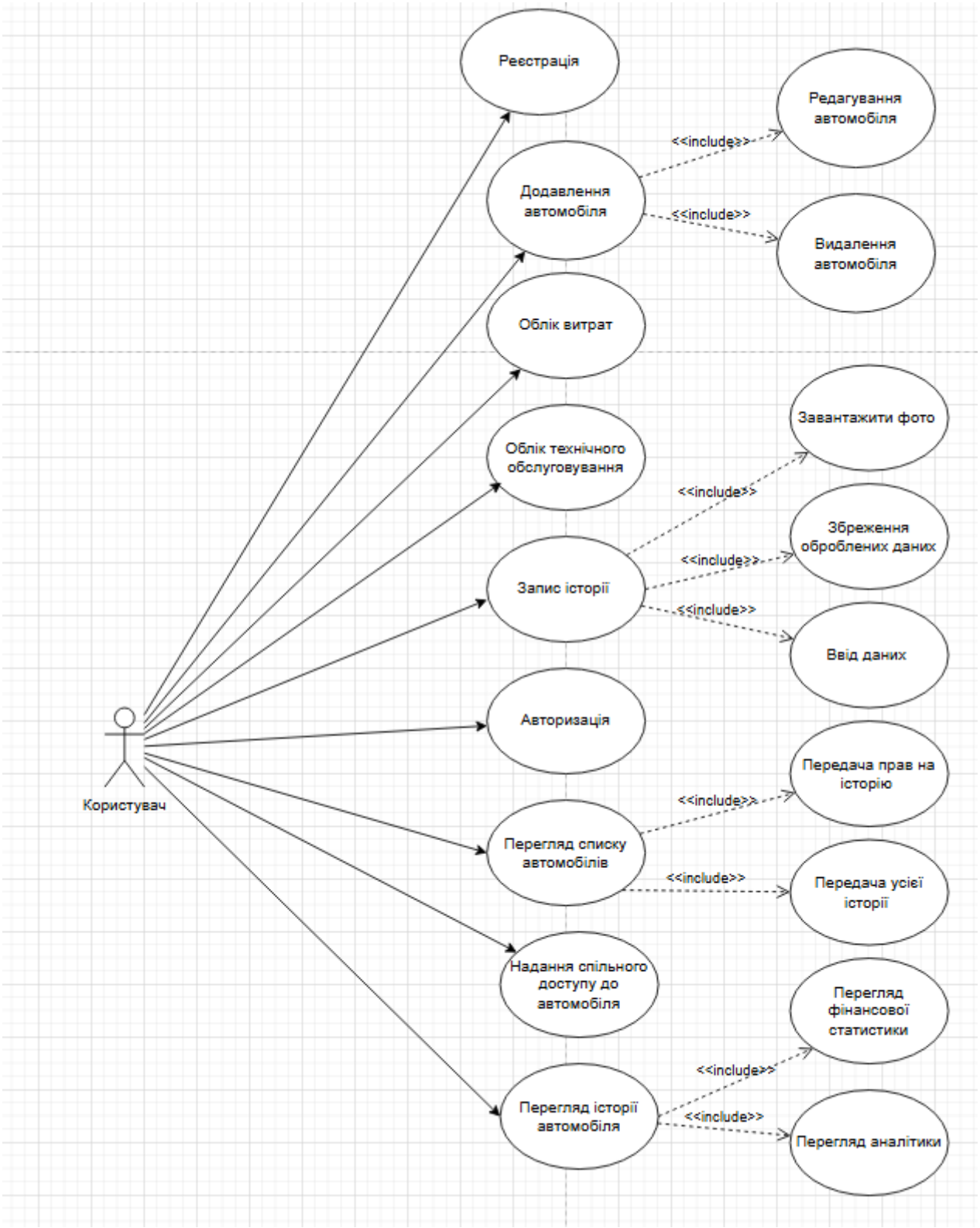


Рисунок 1.2 – Діаграма варіантів використання

Побудована діаграма забезпечує узгодженість між вимогами користувачів і подальшими етапами проєктування програмної системи, зокрема розробкою архітектури та UML-діаграм. Вона дає змогу ідентифікувати обов’язкові та додаткові сценарії взаємодії, а також визначити залежності між ними за допомогою відношень include та extend. Це дозволяє зменшити ризик логічних помилок на етапі проєктування, підвищити цілісність системи та забезпечити її масштабованість.

Таким чином, діаграма варіантів використання слугує основою для подальшої деталізації програмної системи, розробки користувацького інтерфейсу та формування тестових сценаріїв, забезпечуючи відповідність реалізованого функціоналу визначеним вимогам.

2 ПРОЄКТУВАННЯ ТА КОНСТРУЮВАННЯ ПРОГРАМНОЇ СИСТЕМИ

У цьому розділі розглядаються питання проєктування архітектури багатоплатформенного застосунку для моніторингу стану та витрат обслуговування автомобіля. Обґрунтовується вибір процесу розробки, технологічних рішень і структури бази даних. Також виконано моделювання системи за допомогою UML-діаграм.

2.1 Вибір процесу розробки та технологій

Для розробки багатоплатформенного застосунку для моніторингу стану та витрат обслуговування автомобіля обрано гнучкий (Agile) підхід із використанням Scrum як основної моделі управління процесом розробки. Такий вибір обумовлений кількома ключовими факторами:

– Ітеративна розробка та швидка адаптація

Agile дозволяє розбивати розробку на короткі ітерації (спринти) тривалістю 1–2 тижні, що забезпечує можливість оперативно реагувати на зміни вимог користувачів або уточнення функціоналу. Для MVP (Minimum Viable Product) це критично важливо, оскільки дає змогу швидко протестувати базовий функціонал та отримати зворотний зв’язок [39].

– Інтеграція з сучасними сервісами управління розробкою

Використовується GitHub для контролю версій, організації репозиторіїв та ведення історії комітів. Для автоматизації процесу інтеграції та деплою налаштовано CI/CD-пайплайн (наприклад, GitHub Actions), який дозволяє автоматично запускати тести, збирати фронтенд (Ionic React) та бекенд (Node.js/Express), а також розгортати застосунок на тестових середовищах [40, 18, 2, 3].

– Ролі та відповідальність у команді

Scrum передбачає чітке розмежування ролей: Product Owner визначає пріоритети та функціональні вимоги, Scrum Master контролює процес спринтів, а

розробники реалізують функціонал. Така структура дозволяє узгоджено координувати роботу між фронтенд-розробниками, бекенд-розробниками та фахівцями з інтеграції сервісів штучного інтелекту.

– Взаємодія фронтенду та бекенду

Процес передбачає безперервну інтеграцію змін, що дозволяє тестувати взаємодію клієнтської частини (Ionic React) із серверною логікою, REST API, базою даних PostgreSQL та сервісами Firebase у реальному часі. Це зменшує ризик виникнення критичних помилок на етапі інтеграції та забезпечує стабільність системи.

– Відстеження прогресу та якості

Завдяки Agile практикуються щоденні стендапи, використання дошок завдань (GitHub Projects) і планування спринтів. Усі зміни відслідковуються через коміти, pull request-и та автоматичне тестування, що гарантує високу якість коду та своєчасне виявлення помилок.

Клієнтська частина багатоплатформенного застосунку розробляється з використанням фреймворку Ionic React, що дозволяє створювати PWA (Progressive Web App) та мобільні застосунки для Android та iOS. Використання Ionic React забезпечує кросплатформенність, доступ до нативних функцій пристрою, а також є основою для гнучкого і модульного підходу до розробки інтерфейсу користувача [20].

Для візуалізації даних застосовуються спеціалізовані бібліотеки UI-компонентів, що дозволяють створювати таблиці, графіки, індикатори та дашборди, які відображають інформацію про пробіг, витрати та стан автомобіля у зручній та наочній формі. Крім того, інтеграція з бібліотеками для роботи з формами та завантаженням файлів забезпечує можливість введення даних вручну та завантаження фото документів для подальшого аналізу системою штучного інтелекту [35, 32].

Для організації структури проекту використовується підхід Feature-Sliced Design (FSD) (Рис. 2.1), який передбачає модульне та ієрархічне розділення

функціоналу на логічні «шари» та «фічі». Основні принципи FSD для даного застосунку включають:

- Виділення глобальних і локальних фіч: функціонал поділяється на великі блоки (наприклад, облік автомобіля, фінанси, сповіщення) і дрібніші компоненти (введення пробігу, завантаження фото, перегляд аналітики), що дозволяє легше масштабувати систему та додавати нові можливості.
- Ієрархія шарів: шари організовані від найбільш загальних (app, pages) до більш специфічних (widgets, features, entities), що забезпечує зрозумілу структуру та легку навігацію по коду.
- Ізоляція залежностей: FSD дозволяє уникнути «плутанини» у зв’язках між компонентами, забезпечуючи чіткі точки інтеграції між модулями та мінімізуючи ризик побічних ефектів при зміні функціоналу.
- Простота тестування: завдяки розділенню на ізольовані фічі можна проводити юніт-тестування окремих компонентів без необхідності піднімати всю систему, що значно підвищує стабільність і надійність клієнтської частини.
- Можливість підключення додаткових сервісів: м спрощує інтеграцію з зовнішніми API, такими як Firebase для автентифікації та push-сповіщень, або ШІ-сервіси для обробки завантажених фото документів [17].

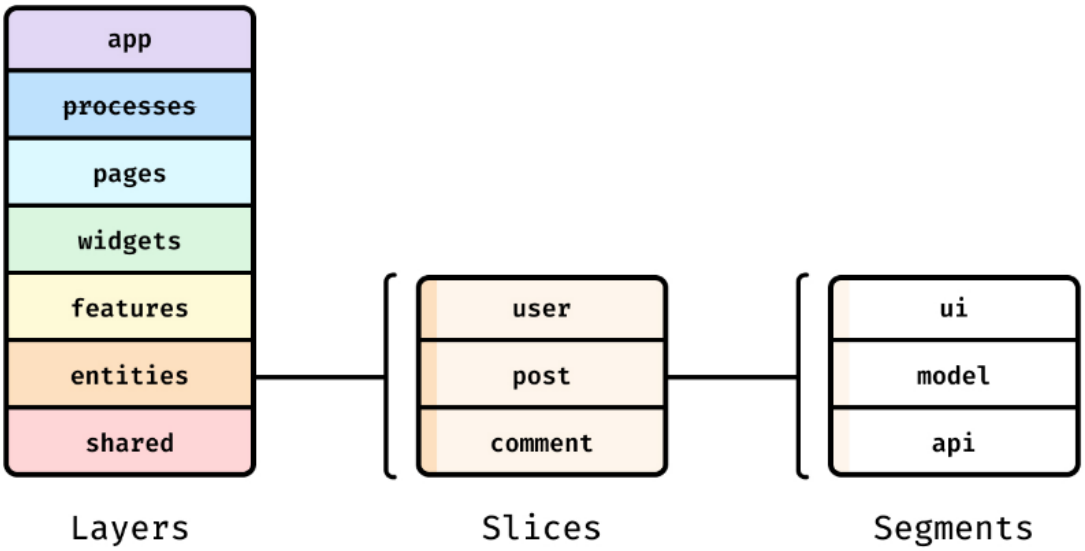


Рисунок 2.1 – Діграма розподілення програми з Feature-Sliced Design

Застосування Feature-Sliced Design у комбінації з Ionic React дозволяє не лише забезпечити модульність, масштабованість та підтримуваність коду, а й спрощує процес розробки фронтенду з можливістю вставки зображень, форм, графіків і таблиць, що є критично важливим для функціоналу та інтерфейсу.

Зважаючи на те, що програмний продукт є багатоплатформенним застосунком із клієнтською частиною на Ionic React, а також передбачає інтенсивну взаємодію з базою даних, зовнішніми сервісами та модулями аналізу даних, для реалізації серверної частини було обрано платформу Node.js з використанням фреймворку Nest.js [4].

Використання Node.js обґрунтоване його асинхронною моделлю обробки запитів, яка є ефективною для систем із великою кількістю паралельних HTTP-запитів від мобільних і вебклієнтів. Це є особливо важливим для даного застосунку, оскільки сервер повинен одночасно обробляти операції автентифікації, запити на отримання історії автомобіля, збереження фінансових даних та обробку мультимедійних файлів [15].

Для побудови серверної архітектури було використано Nest.js, оскільки цей фреймворк надає структурований підхід до розробки бекенду на основі

модульності, інверсії залежностей та чіткої ієрархії компонентів. Такий підхід є доцільним для середніх і великих проєктів, де важливо забезпечити:

- масштабованість серверної логіки;
- зручну підтримку та розширення функціоналу;
- розділення відповідальності між модулями (користувачі, автомобілі, фінанси, доступи, аналітика) [4].

Для зберігання даних було обрано реляційну систему управління базами даних PostgreSQL, що зумовлено необхідністю роботи зі структурованими та взаємопов'язаними даними. У межах даного застосунку це включає:

- облік користувачів і ролей доступу;
- зберігання інформації про декілька автомобілів одного користувача;
- ведення історії пробігу, витрат та технічного обслуговування;
- фінансові операції та аналітичні розрахунки [5].

PostgreSQL забезпечує підтримку складних SQL-запитів, транзакцій, зовнішніх ключів та обмежень цілісності, що є критично важливим для збереження коректності фінансових і технічних даних автомобіля. Крім того, дана СУБД добре масштабується та широко використовується у промислових вебзастосунках [6, 11].

Для взаємодії між клієнтською та серверною частинами застосунку реалізовано REST API, яке забезпечує стандартизований та зрозумілий механізм обміну даними. Використання REST-підходу дозволяє:

- уніфікувати доступ до серверних ресурсів з різних платформ (PWA, Android, iOS);
- чітко розділити відповідальність між фронтендом і бекендом;
- спростити тестування API та інтеграцію з зовнішніми сервісами.

REST API реалізує основні операції керування ресурсами, такі як створення, отримання, оновлення та видалення даних (CRUD), а також забезпечує захищену передачу інформації між клієнтом і сервером з використанням механізмів автентифікації та авторизації [7].

Таким чином, використання Node.js у поєднанні з Nest.js та PostgreSQL є технічно обґрунтованим вибором для серверної частини даного застосунку, оскільки ці технології забезпечують продуктивність, надійність, масштабованість і тісну інтеграцію з клієнтською частиною, реалізованою на Ionic React.

Для зберігання структурованих даних системи використовується реляційна база даних PostgreSQL, яка забезпечує цілісність, надійність та підтримку складних зв’язків між сутностями, зокрема користувачами, автомобілями, витратами та історією обслуговування. Такий вибір є доцільним з огляду на необхідність обробки транзакційних та аналітичних даних.

Для реалізації автентифікації користувачів, управління сесіями та надсилання push-сповіщень застосовується Firebase, що дозволяє забезпечити безпечний доступ до системи та стабільну роботу на платформах Android і iOS. Хмарні сервіси Firebase також зменшують навантаження на серверну частину застосунку.

Зберігання документів і фотографій, що завантажуються користувачами, реалізується за допомогою хмарного файлового сховища (Firebase Storage або Amazon S3), що забезпечує масштабованість і контроль доступу до мультимедійних даних.

Для автоматизації введення даних передбачено використання сервісів розпізнавання тексту на зображеннях (OCR). Це дозволяє автоматично вилучати інформацію з чеків та документів, зменшуючи кількість ручного введення та підвищуючи точність даних, що зберігаються у системі [41].

Безпека системи забезпечується шляхом використання Firebase Authentication, розмежування прав доступу на серверному рівні та захищеного передавання даних через HTTPS/SSL. Такий підхід дозволяє гарантувати конфіденційність і цілісність персональних та фінансових даних користувачів [14].

Якість програмного забезпечення забезпечується застосуванням юніт- та інтеграційного тестування, а також використанням Firebase Test Lab для перевірки роботи мобільних застосунків на різних пристроях і версіях операційних систем [13].

2.2 Проєктування архітектури системи

Проєктування архітектури програмної системи визначає логічну та фізичну структуру застосунку, принципи взаємодії між його компонентами та забезпечує можливість масштабування, підтримки і розвитку системи. Для реалізації багатоплатформенного застосунку моніторингу стану та витрат обслуговування автомобіля було обрано клієнт-серверну архітектуру, доповнену використанням хмарних сервісів та зовнішніх інтеграцій.

Клієнтська частина системи реалізована на базі Ionic React із застосуванням архітектурного підходу Feature-Sliced Design (FSD). Даний підхід передбачає структурування фронтенд-коду за функціональними ознаками (features), що дозволяє ізолювати бізнес-логіку, компоненти інтерфейсу та модулі стану. Використання FSD підвищує масштабованість клієнтської частини, спрощує підтримку коду та забезпечує можливість незалежного розвитку окремих функціональних модулів, таких як управління автомобілями, облік витрат, історія обслуговування та сповіщення [17].

Застосунок реалізується як Progressive Web Application (PWA) (Рис. 2.2), що дозволяє поєднати переваги веб- та мобільних застосунків. PWA забезпечує встановлення застосунку на пристрій користувача, роботу в офлайн-режимі, швидке завантаження інтерфейсу та підтримку push-сповіщень. Завдяки використанню PWA забезпечується єдиний код для вебверсії, а також спрощується розгортання та оновлення застосунку без необхідності повторної публікації в магазинах застосунків [20].

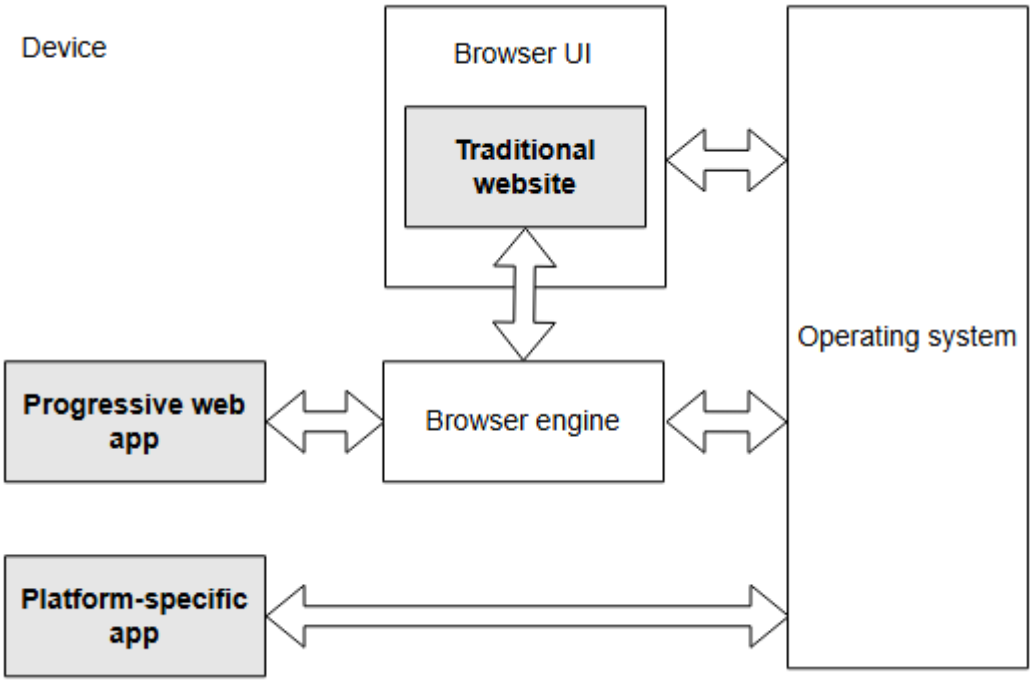


Рисунок 2.2 – Схема роботи застосунку у PWA [20]

Серверна частина системи реалізована з використанням Nest.js, який базується на архітектурному патерні MVC (Рис. 2.3) (Model–View–Controller). У межах даного підходу контролери відповідають за обробку HTTP-запитів, сервіси реалізують бізнес-логіку, а моделі описують структуру та взаємодію даних із базою PostgreSQL. Така архітектура забезпечує чітке розмежування відповідальності між компонентами, підвищує тестованість коду та спрощує розширення функціоналу серверної частини [44].

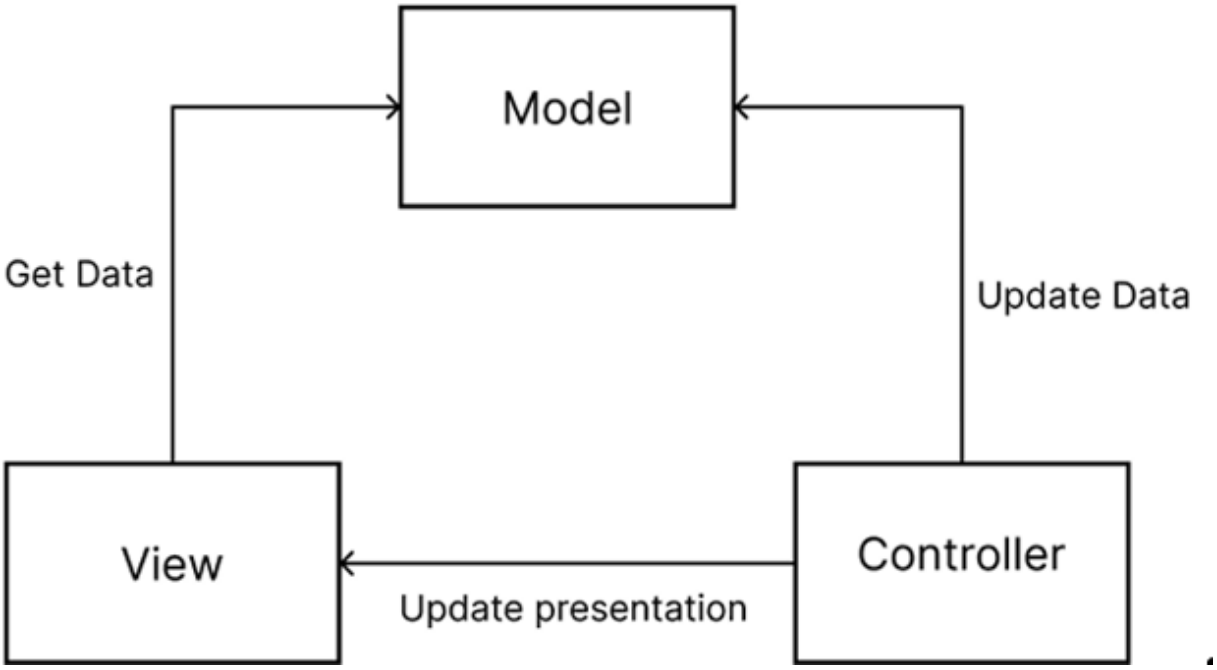


Рисунок 2.3 – Архітектурний патерн MVC

Взаємодія між клієнтською та серверною частинами здійснюється через REST API з використанням захищеного протоколу HTTPS. Для зберігання структурованих даних застосовується реляційна база даних PostgreSQL, що дозволяє ефективно працювати з транзакціями, зв’язками між сутностями та історичними даними.

Архітектура системи також включає інтеграцію з Firebase, який використовується для автентифікації користувачів, управління сесіями, надсилання push-сповіщень та зберігання мультимедійних файлів. Це забезпечує високий рівень безпеки, масштабованість і стабільну роботу на мобільних платформах Android та iOS.

Окремим рівнем архітектури є інтеграція сервісів штучного інтелекту, які використовуються для розпізнавання тексту на зображеннях документів і автоматичного вилучення даних. Такі сервіси взаємодіють із серверною частиною, що дозволяє централізовано обробляти результати аналізу та зберігати їх у базі даних [12].

Запропонована архітектура поєднує модульність, безпеку та гнучкість і створює надійну основу для подальшого розвитку системи, зокрема розширення

аналітичного функціоналу, інтеграції маркетплейсу та впровадження нових сервісів підтримки експлуатації автомобіля.

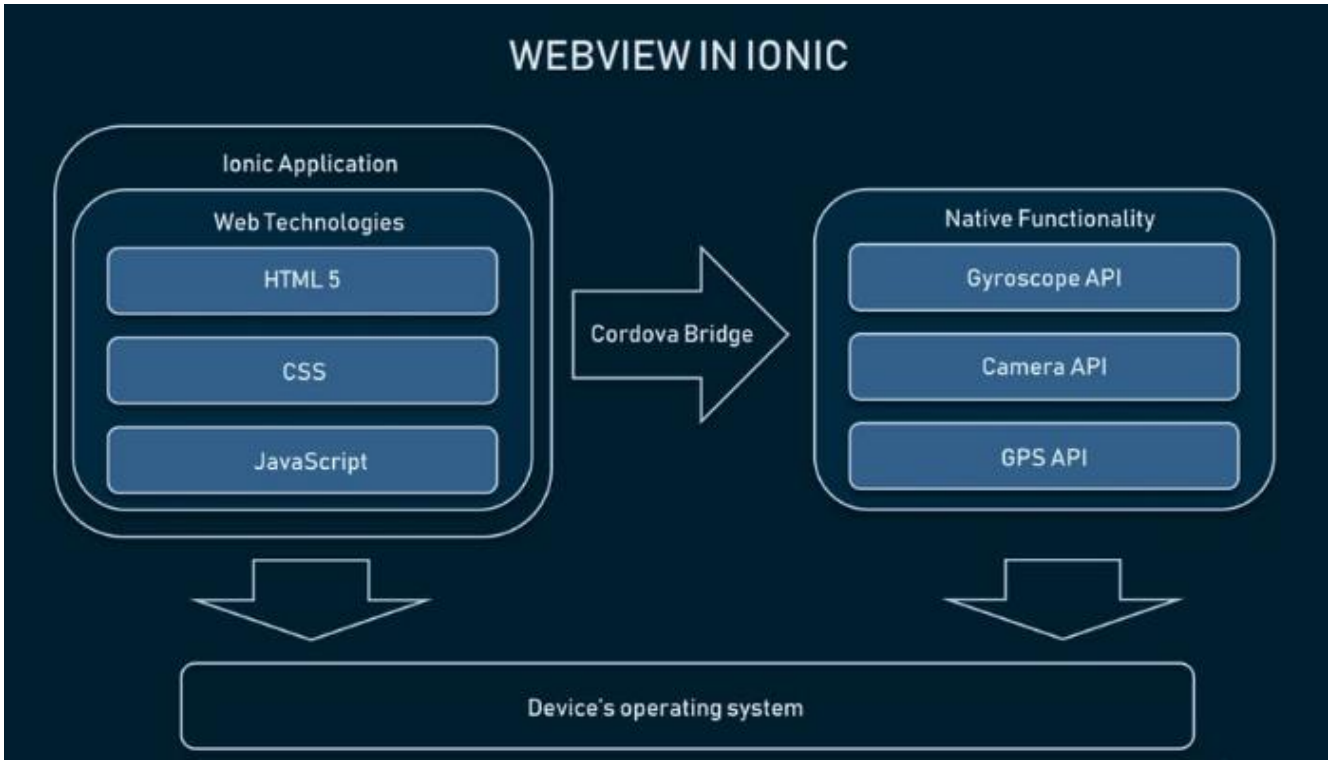


Рисунок 2.4 – Схема роботи WEBView в Ionic

Окрему роль в архітектурі системи відіграє використання фреймворку (Рис. 2.4) Ionic, який забезпечує ефективну реалізацію багатоплатформенного клієнтського застосунку. Ionic надає набір готових UI-компонентів, оптимізованих для мобільних пристроїв, що дозволяє створювати інтерфейс, адаптований до особливостей сенсорного керування та різних розмірів екранів.

Завдяки інтеграції з React, Ionic дозволяє використовувати сучасний компонентний підхід до побудови інтерфейсу, ефективно керувати станом застосунку та повторно використовувати логіку між різними модулями. Це спрощує реалізацію складних сценаріїв взаємодії користувача з системою, зокрема роботи з формами введення даних, візуалізацією історії автомобіля та переглядом аналітичної інформації.

Важливою перевагою Ionic є можливість збірки застосунку у вигляді Progressive Web Application, а також нативних мобільних застосунків для Android

та iOS із використанням одного кодового базису. Це зменшує витрати на розробку та підтримку, забезпечує однакову бізнес-логіку на всіх платформах і пришвидшує оновлення функціоналу.

Крім того, Ionic забезпечує інтеграцію з нативними можливостями мобільних пристроїв через Capacitor, зокрема доступ до камери, файлової системи та push-сповіщень. Це є критично важливим для реалізації функцій завантаження фото документів, обробки зображень та оперативного інформування користувача про стан автомобіля [43, 19].

Таким чином, використання Ionic у поєднанні з React є доцільним з архітектурної точки зору та забезпечує гнучкість, масштабованість і ефективність розробки багатоплатформенного застосунку.

2.3 Проєктування структури бази даних

Проєктування структури бази даних є одним з базових етапів створення програмної системи, оскільки саме на цьому рівні визначається спосіб зберігання інформації, взаємозв’язок між основними сутностями та можливість подальшого масштабування системи. Для багатоплатформенного застосунку було обрано реляційну модель бази даних, яка дозволяє забезпечити цілісність даних, підтримку складних зв’язків та ефективну обробку історичних і фінансових записів.

Процес проєктування бази даних включав кілька етапів: визначення ключових сутностей, нормалізацію даних, встановлення типів зв’язків між таблицями та вибір оптимальних типів даних. Особлива увага приділялася збереженню історичних даних, фінансової інформації та забезпеченню цілісності даних при виконанні операцій додавання, оновлення та видалення записів.

Для реалізації серверної частини системи було обрано PostgreSQL як основну систему управління базами даних. Даний вибір обумовлений рядом технічних і архітектурних переваг. PostgreSQL є потужною реляційною СУБД з відкритим вихідним кодом, яка забезпечує високий рівень надійності, підтримку транзакцій, складних запитів та розширених типів даних [5].

Використання PostgreSQL є доцільним з огляду на необхідність роботи з взаємопов’язаними даними, такими як користувачі, автомобілі, витрати та історія обслуговування. СУБД підтримує строгі обмеження цілісності, зовнішні ключі та механізми каскадних операцій, що є критично важливим для коректної роботи системи. Крім того, PostgreSQL забезпечує точну роботу з фінансовими даними завдяки підтримці типу DECIMAL, що дозволяє уникнути похибок при збереженні грошових значень [16].

Ще однією перевагою PostgreSQL є можливість масштабування, оптимізації запитів та використання індексів для роботи з великими обсягами історичних даних. Це особливо важливо для системи, яка передбачає накопичення інформації протягом тривалого часу [6].

До ключових сутностей належать таблиці, що безпосередньо відображають основні об’єкти предметної області та формують ядро інформаційної моделі системи. Такими таблицями є users, cars, mileages, expenses та maintenances.

Таблиця (Рис. 2.5) users є центральною в системі та призначена для зберігання інформації про користувачів застосунку. Вона містить унікальний ідентифікатор користувача, облікові дані для автентифікації, контактну інформацію, аватар та роль у системі. Унікальність електронної адреси забезпечує коректну ідентифікацію користувача та запобігає створенню дублюючих облікових записів. Саме з цією таблицею пов’язані всі основні функціональні сценарії системи.

```
@Entity('users')  Show usages  Bogdan *
export class User {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'varchar', unique: true })
  email: string;

  @Column({ type: 'varchar' })
  password: string;

  @Column({ type: 'varchar', nullable: true })
  name: string | null;

  @Column({ type: 'varchar', nullable: true })
  phone: string | null;

  @Column({ type: 'varchar', nullable: true })
  avatar: string | null;

  @Column({ type: 'varchar', nullable: true })
  role: string | null;

  @OneToMany(() => RefreshToken, (refreshToken) => refreshToken.user)
  refreshTokens: RefreshToken[];

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}
```

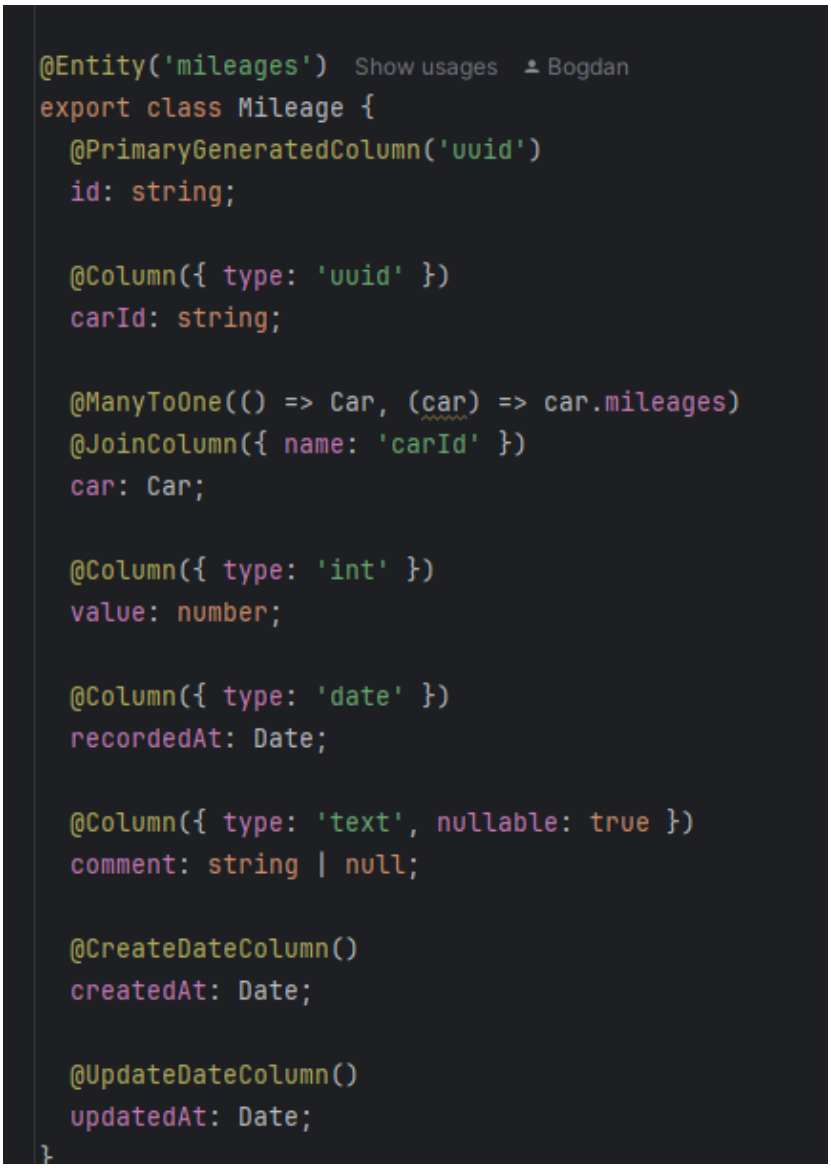
Рисунок 2.5 – Сутність UserEntity.

Таблиця (Рис. 2.6) cars відображає транспортні засоби, які обліковуються в системі. Кожен автомобіль прив’язаний до конкретного користувача, що дозволяє одному користувачу вести облік декількох транспортних засобів. Таблиця містить основні характеристики автомобіля, зокрема марку, модель, рік випуску, початковий пробіг та статус активності. Поле активності дозволяє зберігати історичні дані навіть після продажу автомобіля, не видаляючи його з системи.

```
@Entity('cars')  Show usages  Bogdan
export class Car {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'uuid' })
  userId: string;

  @ManyToOne(() => User)
  @JoinColumn({ name: 'userId' })
  user: User;

  @Column({ type: 'varchar' })
  make: string;

  @Column({ type: 'varchar' })
  model: string;

  @Column({ type: 'int' })
  year: number;

  @Column({ type: 'int' })
  initialMileage: number;

  @Column({ type: 'boolean', default: false })
  isActive: boolean;

  @OneToMany(() => Mileage, (mileage) => mileage.car)
  mileages: Mileage[];

  @OneToMany(() => Expense, (expense) => expense.car)
  expenses: Expense[];

  @OneToMany(() => Maintenance, (maintenance) => maintenance.car)
  maintenances: Maintenance[];
```

Рисунок 2.6 – Сутність CarEntity.

Для фіксації змін пробігу використовується таблиця (Рис. 2.7) mileages, яка реалізує історичний облік експлуатації автомобіля. Кожен запис містить значення пробігу, дату внесення та додатковий коментар. Такий підхід дозволяє відслідковувати динаміку використання автомобіля та формувати аналітичні звіти на основі часових даних.



```
@Entity('mileages')  Show usages  Bogdan
export class Mileage {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'uuid' })
  carId: string;

  @ManyToOne(() => Car, (car) => car.mileages)
  @JoinColumn({ name: 'carId' })
  car: Car;

  @Column({ type: 'int' })
  value: number;

  @Column({ type: 'date' })
  recordedAt: Date;

  @Column({ type: 'text', nullable: true })
  comment: string | null;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}
```

Рисунок 2.7 – Сутність MileageEntity.

Таблиця (Рис. 2.8) expenses відповідає за фінансовий облік витрат, пов’язаних з експлуатацією автомобіля. Вона зберігає інформацію про суму витрат, дату здійснення, опис та категорію витрат. Прив’язка витрат до конкретного автомобіля забезпечує можливість аналізу загальних та категорійних витрат протягом усього життєвого циклу транспортного засобу.

```
@Entity('expenses')  Show usages  Bogdan
export class Expense {
  @PrimaryGeneratedColumn('uuid')
  id: string;

  @Column({ type: 'uuid' })
  carId: string;

  @ManyToOne(() => Car, (car) => car.expenses)
  @JoinColumn({ name: 'carId' })
  car: Car;

  @Column({ type: 'uuid' })
  categoryId: string;

  @ManyToOne(() => ExpenseCategory, (category) => category.expenses)
  @JoinColumn({ name: 'categoryId' })
  category: ExpenseCategory;

  @Column({ type: 'decimal', precision: 10, scale: 2 })
  amount: number;

  @Column({ type: 'date' })
  expenseDate: Date;

  @Column({ type: 'text', nullable: true })
  description: string | null;

  @CreateDateColumn()
  createdAt: Date;

  @UpdateDateColumn()
  updatedAt: Date;
}
```

Рисунок 2.8 – Сутність ExpenseEntity.

Для обліку технічного обслуговування передбачена таблиця maintenances, яка містить інформацію про тип виконаних робіт, пробіг на момент обслуговування, дату, вартість та додаткові пояснення. Наявність даної таблиці дозволяє формувати повну історію сервісного обслуговування автомобіля та використовувати ці дані для побудови рекомендацій щодо майбутнього технічного обслуговування.

Допоміжні таблиці виконують сервісну та довідкову функцію та забезпечують коректну роботу ключових сутностей системи. До них належать таблиці refresh_tokens та expense_categories.

Таблиця refresh_tokens використовується для реалізації механізму безпечної автентифікації та підтримки сесій користувачів. Вона зберігає токени оновлення, пов’язані з користувачами, та час їх дії. Зв’язок між таблицями users і refresh_tokens реалізовано за принципом «один до багатьох» із застосуванням каскадного видалення, що дозволяє автоматично очищувати токени при видаленні користувача та підвищує рівень безпеки системи.

Таблиця expense_categories є довідковою та призначена для класифікації витрат. Вона містить унікальні назви категорій і відповідні іконки для відображення у клієнтському застосунку. Використання окремої таблиці категорій спрощує аналітику, уніфікує дані та зменшує ризик помилок при введенні інформації користувачем.

У базі даних реалізовано чітку систему зв’язків між таблицями, що відображає логіку предметної області. Зв’язки між users і cars, а також між cars і підлеглими таблицями реалізовані за схемою «один до багатьох». Категорії витрат пов’язані з таблицею expenses аналогічним чином.

Для всіх таблиць використовуються UUID як первинні ключі, що забезпечує унікальність записів, спрощує інтеграцію з клієнтською частиною та підвищує безпеку. Фінансові значення зберігаються з використанням типу DECIMAL(10,2), що гарантує точність обліку грошових операцій. Також передбачено автоматичне створення часових міток для фіксації моментів створення та оновлення записів [16].

Запропонована структура бази даних є логічно завершеною, відповідає вимогам предметної області та забезпечує надійну основу для подальшого розвитку застосунку, зокрема інтеграції аналітичних модулів, сервісів штучного інтелекту та маркетплейсу автотоварів.

2.4 Побудова UML-діаграм системи

За допомогою UML-діаграм класів здійснюється моделювання структури програмної системи, взаємозв’язків між її сутностями та сервісами. Діаграма дозволяє наочно відобразити, які об’єкти існують у системі, які дані вони зберігають, а також як ці об’єкти взаємодіють між собою та з бізнес-логікою.

Мета цього підрозділу показати архітектуру застосунку на рівні об’єктів, включаючи основні сутності, їхні атрибути, ключові зв’язки та інтеграцію з сервісними класами, що реалізують бізнес-функціонал.

На основі аналізу вимог та логіки функціонування системи було сформовано набір програмних класів, що відображають структурні та поведінкові особливості її компонентів. Ці класи та їхні взаємозв’язки представлені у вигляді діаграми класів, яка слугує концептуальною моделлю внутрішньої архітектури програмного забезпечення. Діаграма надає візуальне відображення основних сутностей, їхніх атрибутів і залежностей між ними, демонструючи, як дані переходять між компонентами та які ролі виконують окремі елементи у системі. Кожен клас у цій моделі має власну відповідальність і містить властивості та методи, необхідні для реалізації конкретного функціоналу — починаючи від обробки користувацьких даних і управління файлами, закінчуючи підтримкою ієрархії папок та механізмів доступу. Такий підхід дозволяє забезпечити структурованість, розширюваність і зрозумілість програмного коду, а також створює підґрунтя для подальшої реалізації, тестування та масштабування системи.

UML-діаграма класів демонструє структуру програмного забезпечення застосунку (Рис. 2.9) та взаємозв’язки між його основними компонентами. Вона включає ключові сутності, такі як користувачі, автомобілі, історія пробігу, витрати, категорії витрат, обслуговування та токени автентифікації, а також сервіси, що реалізують бізнес-логіку системи. Діаграма наочно показує атрибути кожного класу, їх типи даних і ключові властивості, включаючи UUID-ідентифікатори, timestamps і nullable поля [10].

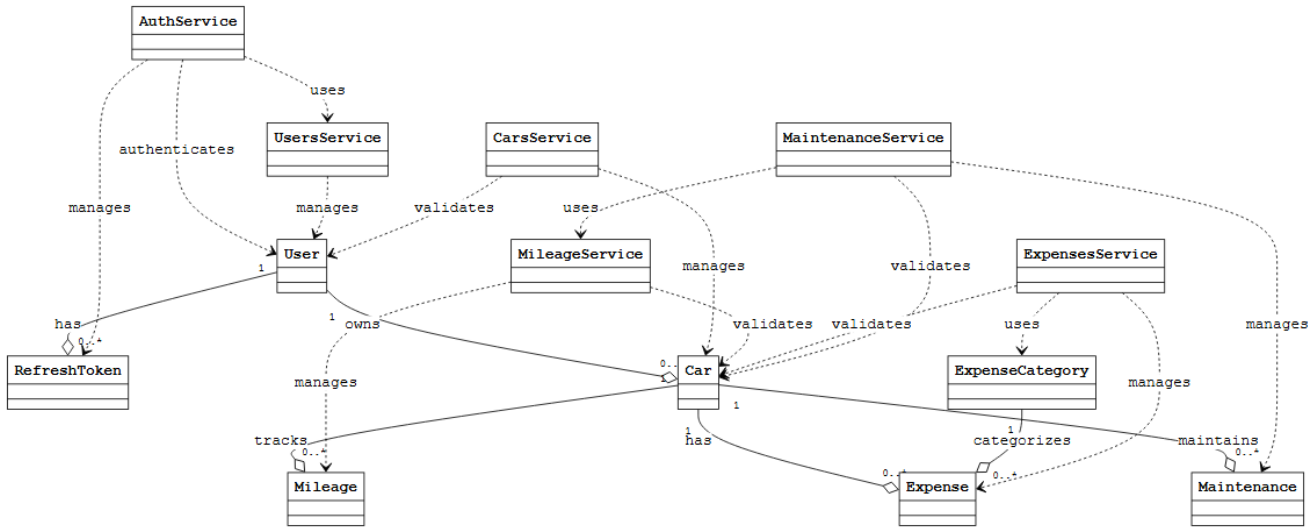


Рисунок 2.9 – Концептуальна схема UML діаграми класів системи

Зв’язки між сутностями представлені у вигляді агрегованих відносин (has-a), що демонструють ієрархію та належність об’єктів, а також залежностей (dependency), які відображають тимчасове використання одних класів іншими сервісами для реалізації функціоналу. Наприклад, AuthService залежить від UsersService для перевірки та автентифікації користувачів, а MaintenanceService взаємодіє з MileageService для визначення рекомендацій щодо обслуговування.

Діаграма також групує функціональні області системи: автентифікацію та управління користувачами, керування автомобілями, облік пробігу, витрат та обслуговування, а також організацію бази даних. Вона демонструє, як сервіси виконують операції CRUD, перевірку прав доступу, валідацію даних і формування статистики.

Завдяки такій моделі розробники можуть чітко бачити структуру системи, розмежування відповідальностей класів та шляхи обміну даними між компонентами. Це дозволяє забезпечити масштабованість, розширюваність і підтримуваність коду, а також полегшує інтеграцію нових функцій та сервісів у майбутньому. UML-діаграма слугує основою для подальшої реалізації, тестування та модифікації системи, забезпечуючи узгоджене і зрозуміле бачення архітектури на всіх етапах розробки.

Клас User (рис. 2.10) відображає користувачів системи та містить їхні основні дані для автентифікації, ідентифікації та управління доступом до функцій застосунку. Він забезпечує основу для персоналізації даних, контролю за правами доступу і зберігання інформації про профіль користувача, включаючи контактні дані та роль у системі.

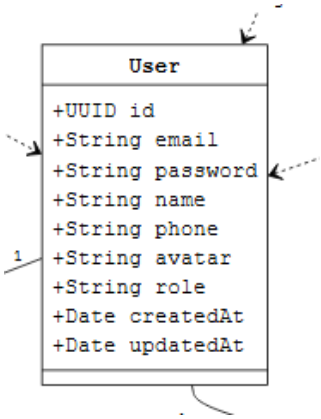


Рисунок 2.10 – Клас User

Клас RefreshToken (рис. 2.11) призначений для зберігання токенів автентифікації, підтримки безпечних сесій користувачів та оновлення токенів для забезпечення безперервного доступу до системи без повторного входу.

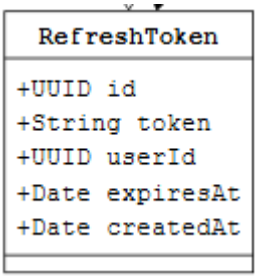


Рисунок 2.11 – Клас RefreshToken

Клас Car (рис. 2.12) представляє автомобілі користувачів, включаючи марку, модель, рік випуску, початковий пробіг та активний стан. Він слугує основою для зберігання історії транспортного засобу та управління кількома машинами для одного користувача.

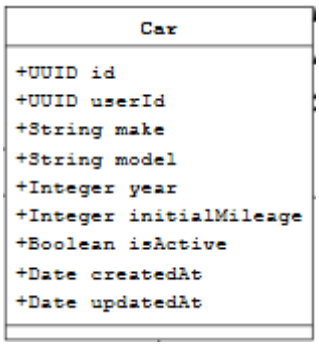


Рисунок 2.12 – Клас Car

Клас Mileage (рис. 2.13) відображає записи про пробіг автомобіля, включаючи дати фіксації та коментарі. Він дозволяє відстежувати історію експлуатації транспортного засобу та забезпечує дані для аналізу стану автомобіля та планування обслуговування.

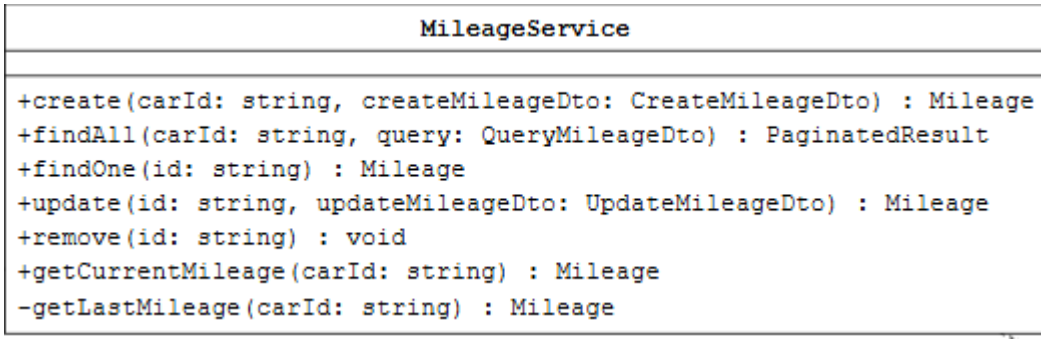


Рисунок 2.13 – Клас Mileage

Клас ExpenseCategory (рис. 2.14) представляє категорії витрат, наприклад паливо, ремонт, ТО, страховка, мийка, парковка. Це дозволяє систематизувати витрати, формувати звіти та забезпечувати зручну фільтрацію даних за категоріями.

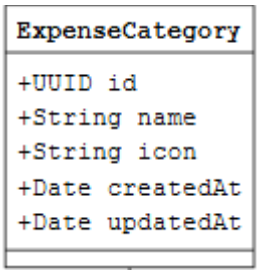


Рисунок 2.14 – Клас ExpenseCategory

Клас Maintenance (рис. 2.15) відображає записи про технічне обслуговування автомобіля, включаючи тип сервісу, пробіг, дату та вартість. Він дозволяє відстежувати історію ТО, формувати рекомендації щодо майбутніх перевірок та аналізувати стан транспортного засобу.

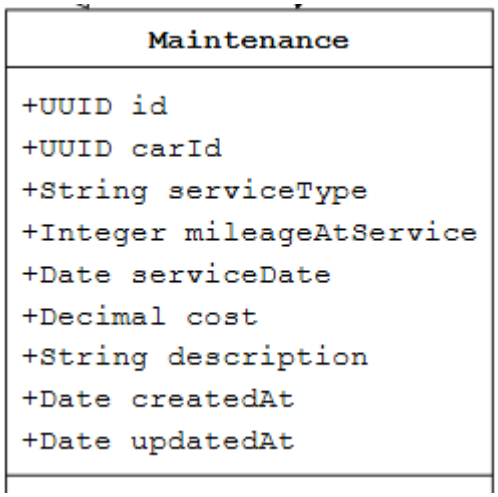


Рисунок 2.15 – Клас Maintenance

Клас UsersService (рис. 2.16) керує даними користувачів, забезпечує їх валідацію, збереження та інтеграцію з іншими компонентами системи, забезпечуючи безпеку та коректну роботу профілів.

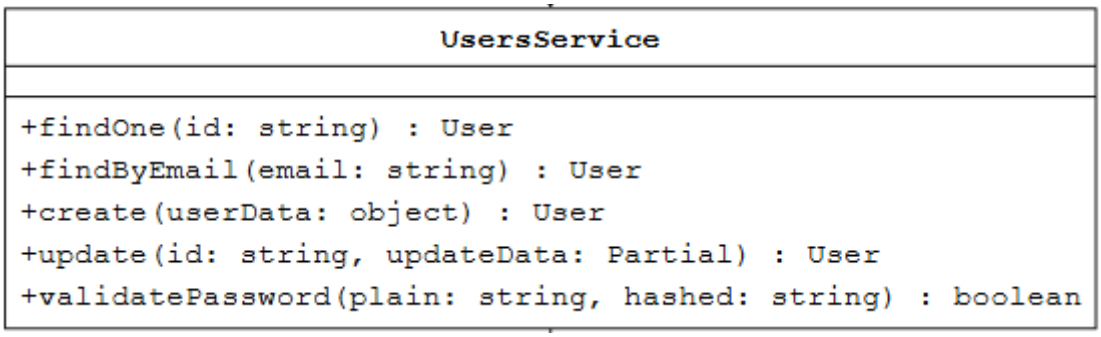


Рисунок 2.16 – Клас UsersService

Клас AuthService (рис. 2.17) реалізує автентифікацію, управління токенами та контроль доступу користувачів. Він гарантує безпечний вхід до системи, управління сесіями та авторизацію для різних ролей.

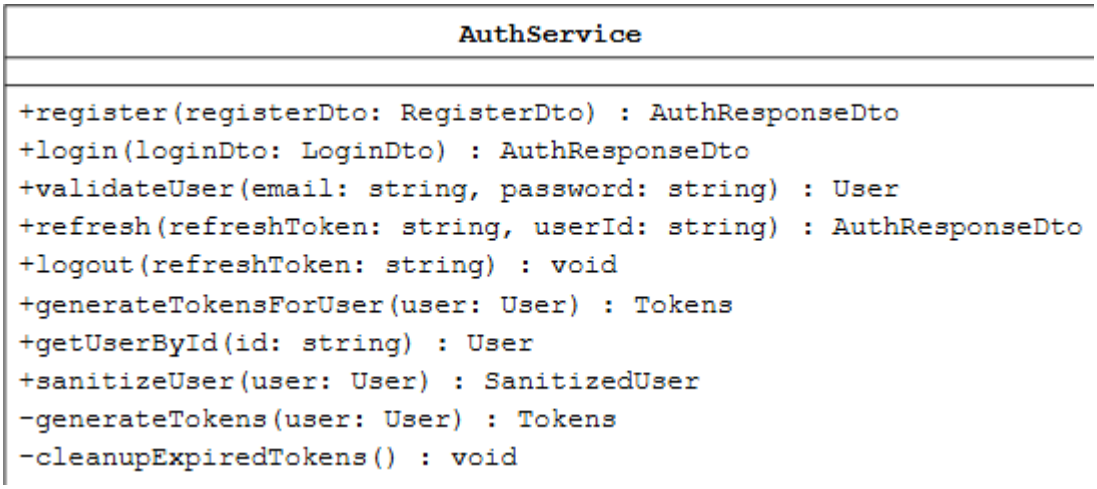


Рисунок 2.17 – Клас AuthService

Клас CarsService (рис. 2.18) відповідає за управління інформацією про автомобілі, включаючи створення, редагування, активацію та перевірку власності. Він є центральним компонентом для роботи з транспортними засобами та інтегрується з усіма сервісами, що з ними пов’язані.

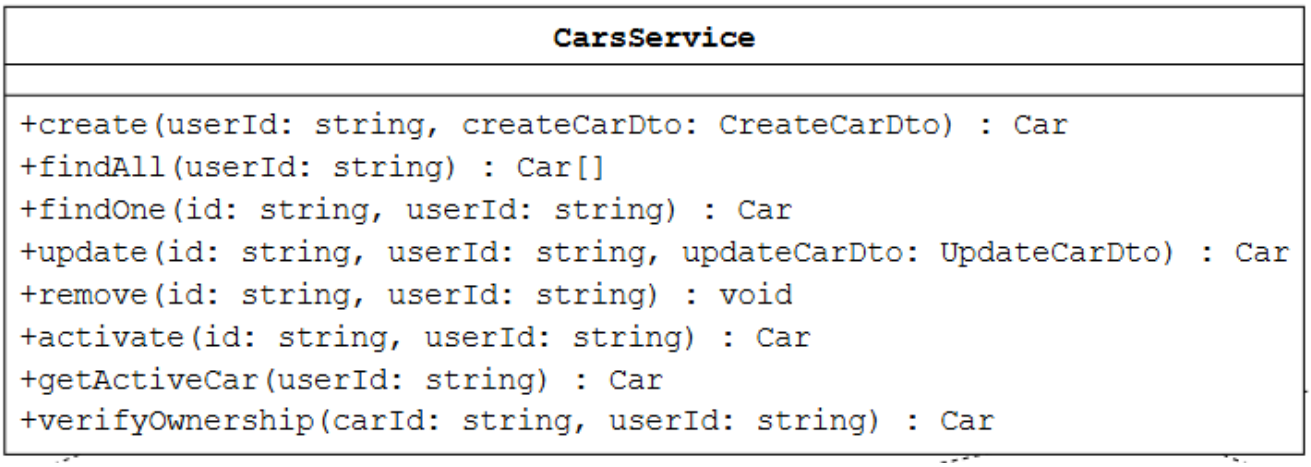


Рисунок 2.18 – Клас CarsService

Клас MileageService (рис. 2.19) обробляє дані про пробіг автомобілів, веде історію та забезпечує актуальні показники. Він необхідний для аналітики, формування звітів та планування технічного обслуговування.

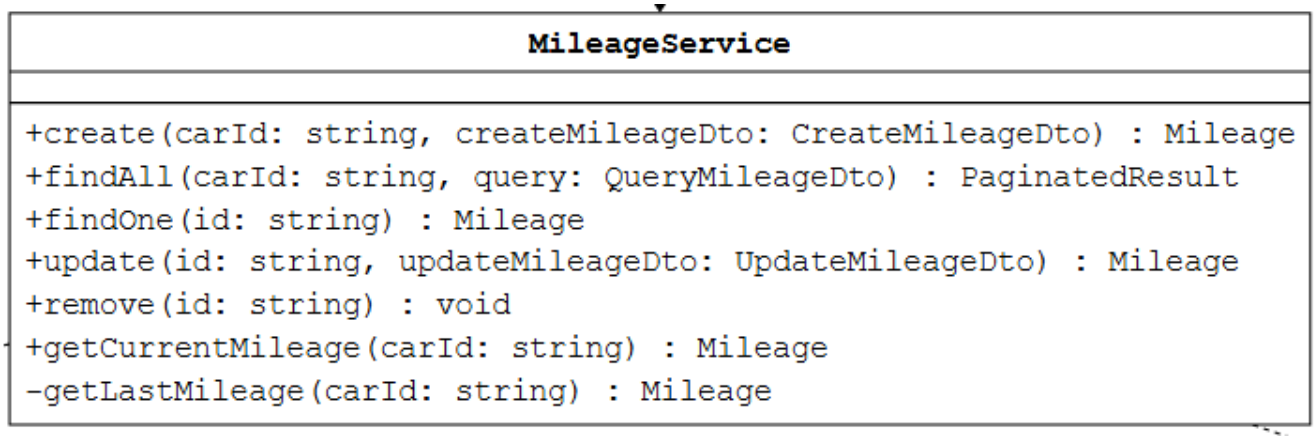


Рисунок 2.19 – Клас MileageService

Клас ExpensesService (рис. 2.20) керує витратами автомобілів, включаючи категоризацію, статистику та облік фінансових операцій. Це дозволяє користувачу контролювати бюджет, вести історію витрат і отримувати звіти для планування подальших витрат.

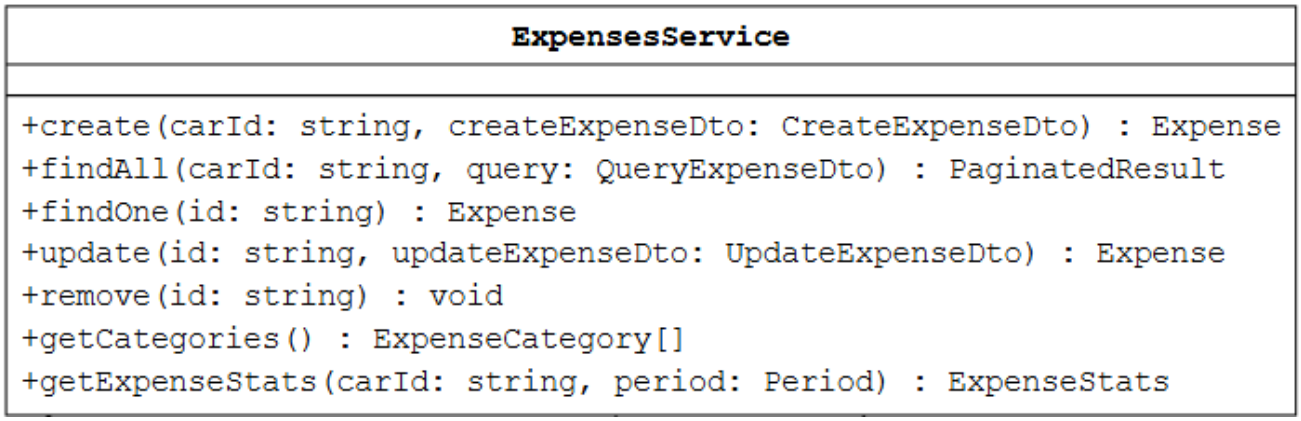


Рисунок 2.20 – Клас ExpensesService

Клас MaintenanceService (рис. 2.21) відповідає за ведення обслуговування автомобілів, відстежує минулі роботи та генерує рекомендації щодо майбутніх технічних перевірок. Він інтегрується з сервісами пробігу і витрат для комплексного контролю стану автомобіля.

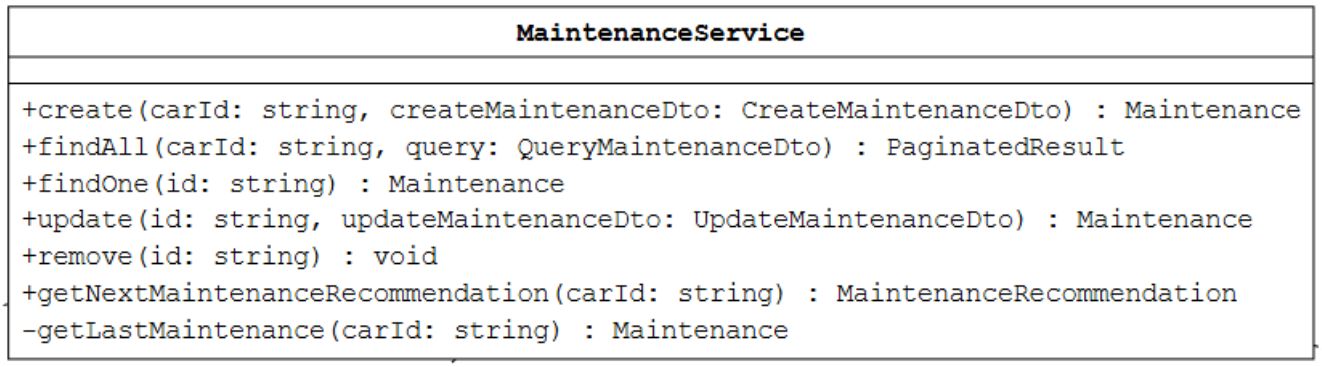


Рисунок 2.21 – Клас MaintenanceService

Клас Expense (рис. 2.22) зберігає інформацію про витрати на автомобіль, такі як категорія, сума, дата та опис витрати. Він дозволяє вести структурований фінансовий облік та отримувати статистичні дані для аналізу витрат за різними параметрами.

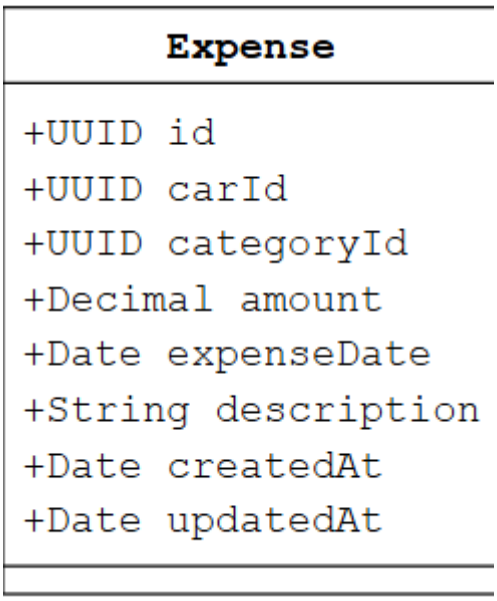


Рисунок 2.22 – Клас Expense

UML-діаграми класів системи наочно демонструють структуру основних сутностей та сервісів, їхні атрибути, зв’язки та залежності, що дозволяє зрозуміти внутрішню архітектуру застосунку. Такий підхід забезпечує наочність моделі, спрощує розробку, тестування та подальше масштабування системи, а також слугує основою для планування інтеграції нових функцій і сервісів.

За допомогою діаграми послідовностей можна наочно відобразити взаємодію об’єктів системи у часі, показати порядок викликів методів та обмін повідомленнями між класами під час виконання конкретного сценарію. Вона дозволяє зрозуміти логіку обробки даних, визначити послідовність дій компонентів і сервісів, а також спрощує аналіз динаміки системи, виявлення вузьких місць у функціоналі та планування тестування і подальшого розширення програмного продукту [10].

За допомогою діаграми послідовностей (Рис 2.23) демонструється процес обробки чека користувачем у системі та взаємодія між ключовими компонентами програмного продукту. Діаграма відображає послідовність дій від моменту створення фотографії чека до збереження структурованих даних про витрати в базі даних.

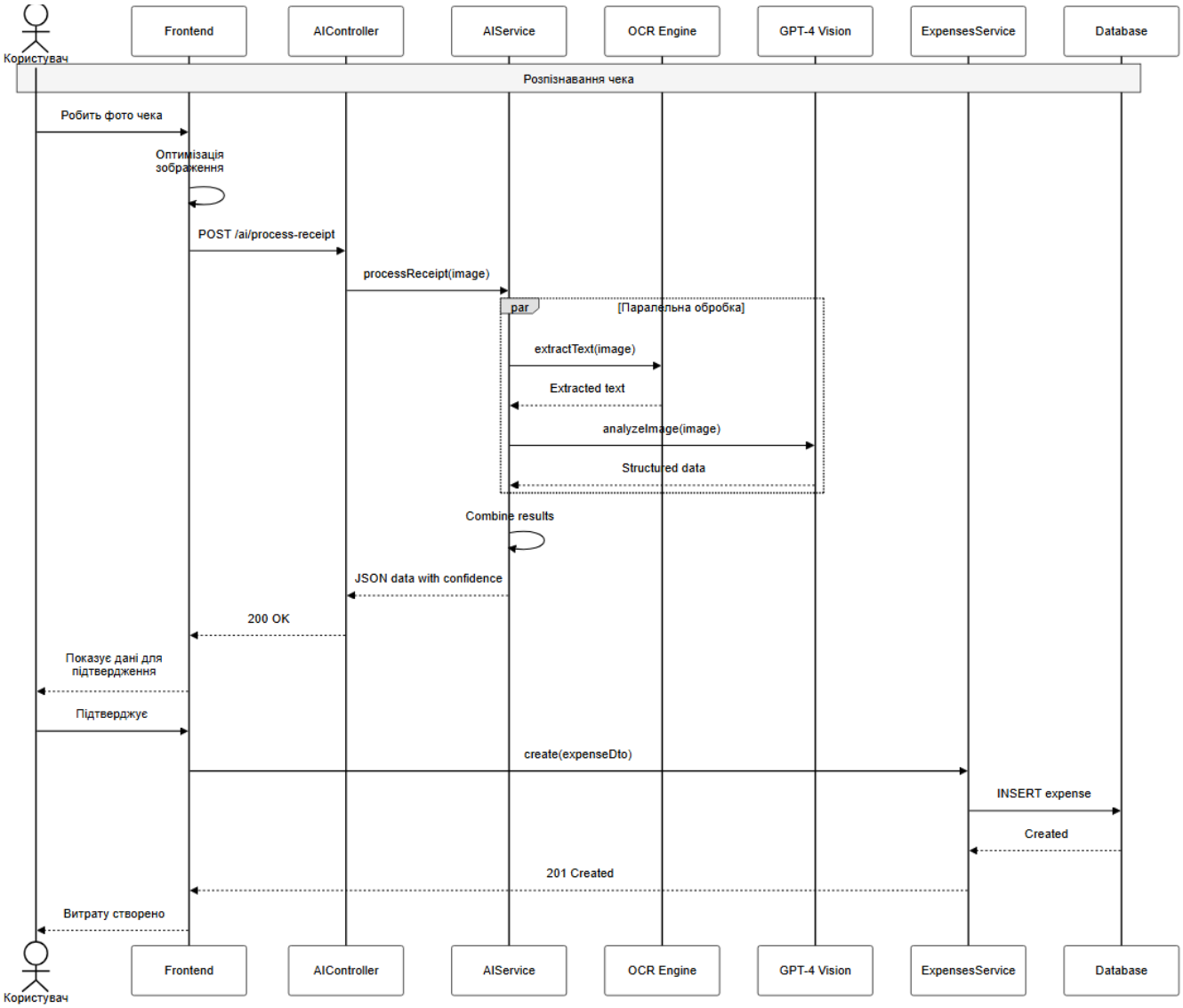


Рисунок 2.23 – Діаграма послідовності, створення запису за допомогою ШІ.

На початку користувач робить фото чека, яке обробляється на фронтенді для оптимізації зображення. Далі запит надсилається на серверний контролер AIController, який передає його на сервіс AIService. У паралельних потоках відбувається розпізнавання тексту за допомогою OCR та аналіз зображення через GPT-4 Vision для отримання структурованих даних. Результати об’єднуються і повертаються контролеру у вигляді JSON із вказанням рівня впевненості [12, 41].

Після цього дані відображаються користувачу для підтвердження. Після підтвердження фронтенд передає інформацію у сервіс ExpensesService, який здійснює збереження даних у базу. База повертає результат створення витрати, і користувач отримує підтвердження успішного збереження.

Діаграма дозволяє наочно продемонструвати взаємодію фронтенду, бекенду, сервісів штучного інтелекту та бази даних, показати паралельну обробку інформації та чітко визначити послідовність дій для реалізації функціоналу обробки чеків у системі.

За допомогою діаграми послідовностей на (Рис 2.24) показано процес додавання витрати та перегляду статистики користувачем у застосунку. Діаграма демонструє взаємодію між фронтендом, контролером, сервісом, механізмом перевірки прав доступу та базою даних.

На етапі додавання витрати користувач вводить дані через інтерфейс фронтенду, який надсилає запит на контролер ExpensesController. Контролер перевіряє права доступу користувача за допомогою CarOwnershipGuard, що звертається до бази даних для підтвердження володіння автомобілем. Після отримання дозволу контролер передає дані у сервіс ExpensesService, який зберігає інформацію у базі даних і повертає підтвердження створення витрати на фронтенд, де користувач бачить повідомлення про успішне додавання.

На етапі перегляду статистики користувач відкриває відповідний розділ у фронтенді, який надсилає GET-запит на контролер. Контролер викликає сервіс для отримання агрегованих даних із бази (сума, середнє тощо), після чого результати повертаються на фронтенд і відображаються у вигляді графіків.

Діаграма дозволяє візуально відобразити порядок обробки даних, перевірку прав доступу, взаємодію між компонентами та логіку побудови статистичних даних, забезпечуючи чітке розуміння процесів додавання витрат та аналітики у системі.

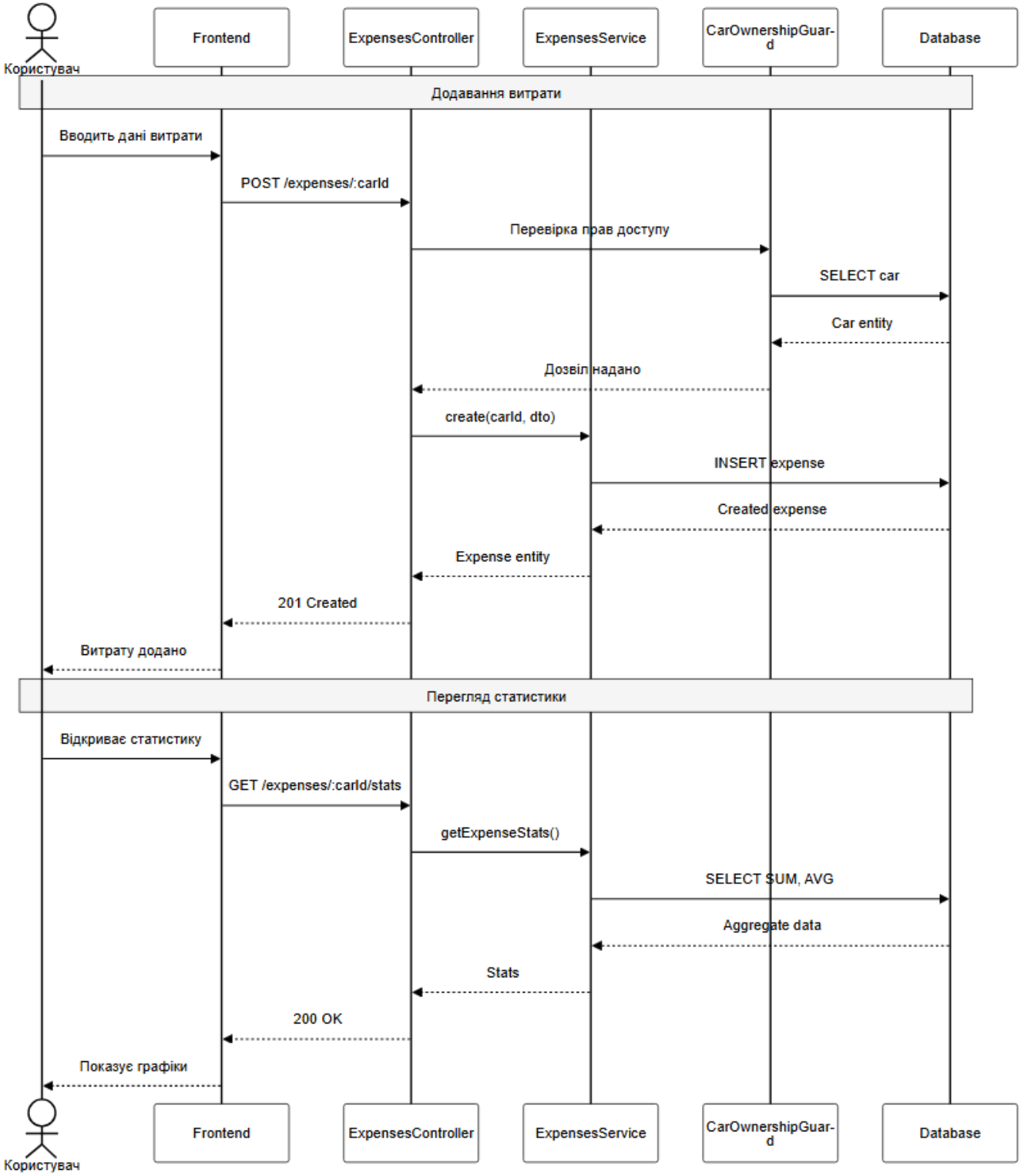


Рисунок 2.24 – Діаграма послідовності, створення витрат та перегляд статистики.

2.5 Оцінка технологічних рішень і структури даних

У результаті проєктування та аналізу вимог до застосунку було обрано оптимальний набір технологій і підходів, які забезпечують ефективну реалізацію багатоплатформового застосунку для моніторингу стану та витрат автомобіля. Для клієнтської частини використано Ionic React, що дозволяє створювати PWA та мобільні застосунки для Android і iOS, забезпечуючи єдиний кодовий базис і високу швидкість розробки. Архітектура фронтенду побудована за принципами Feature-Sliced Design, що полегшує масштабування, підтримку та інтеграцію нових функціональних модулів.

Серверна частина реалізована на Node.js/Nest.js, що забезпечує модульність, підтримку MVC-патерну та гнучку роботу з REST API для взаємодії з фронтендом. Для зберігання структурованих даних обрана PostgreSQL із підтримкою UUID, автоматичних timestamps та каскадного видалення, що гарантує надійність і цілісність інформації. Для автентифікації користувачів, зберігання сесій застосовується Firebase, а для роботи з документами і зображеннями хмарне сховище Firebase Storage.

Сутності системи були детально змодельовані у вигляді UML-діаграм, що дозволяє наочно представити ключові класи, їхні атрибути та зв’язки між ними. Моделювання сутностей та сервісів показало взаємозв’язки між користувачами, автомобілями, витратами, обслуговуванням та пробігом, що створює основу для логічної та безпечної обробки даних.

Вибір технологій і моделювання сутностей забезпечують масштабованість, розширюваність та надійність системи, дозволяючи в майбутньому інтегрувати сервіси штучного інтелекту, додаткову аналітику та маркетплейс автотоварів. Цей підхід створює міцну основу для подальшої розробки, тестування та впровадження застосунку.

3 РОЗРОБКА ТА ТЕСТУВАННЯ СИСТЕМИ

Розділ присвячений практичній реалізації функціоналу розробленого програмного застосунку. Описуються основні етапи реалізації, використані інструменти та підходи до розробки. Окрема увага приділяється тестуванню системи та аналізу результатів з метою перевірки відповідності сформульованим вимогам.

3.1 Реалізація функціоналу системи

Реалізація програмного продукту виконана з урахуванням вимог багатоплатформенності, модульності та безпеки. Клієнтська частина застосунку розроблена за допомогою Ionic React, що дозволяє створювати PWA, а також мобільні застосунки для Android та iOS без необхідності дублювати код для кожної платформи. Структура фронтенду організована за принципами Feature-Sliced Design (FSD), що передбачає поділ на функціональні модулі: управління автомобілями, облік пробігу, витрат та обслуговування, аналітика та рекомендації, фінансовий облік та інтеграція з ШІ-сервісами. Така архітектура забезпечує масштабованість, легкість розширення функціоналу та повторне використання компонентів.

Візуалізація даних реалізована з використанням графіків, таблиць та індикаторів, що дозволяє користувачу швидко оцінювати стан автомобіля та переглядати історичні дані. Для інтерактивності застосовується динамічна генерація контенту на основі введених даних, а також підтримка підтвердження даних після обробки ШІ-модулем, що підвищує точність автоматизованих рішень.

Серверна частина побудована на Node.js/Nest.js, що забезпечує модульну архітектуру, дотримання патерну MVC та інтеграцію з REST API. Це дозволяє фронтенду виконувати запити на створення, оновлення та отримання даних у стандартизованому форматі. PostgreSQL обрана для зберігання структурованих даних завдяки її надійності, підтримці складних зв’язків між сутностями,

транзакційності та масштабованості, а також можливості легкого розширення бази даних у майбутньому. Для автентифікації користувачів та управління сесіями застосовується Firebase Auth, а для зберігання зображень документів – Firebase Storage, що гарантує безпечне та ефективне управління мультимедійними файлами.

Основний функціонал системи реалізований через окремі модульні блоки:

Введення та обробка даних про автомобіль – користувач може вводити інформацію вручну або завантажувати фотографії документів, які обробляються через OCR-модулі та ШІ-сервіси для автоматичного вилучення даних з високою точністю.

Історія автомобіля – забезпечується ведення детальної історії пробігу, витрат, обслуговування та фінансових операцій, що дозволяє здійснювати аналітику та планування ТО.

Керування декількома автомобілями – користувач може вести історію декількох автомобілів, а система підтримує мультидоступ та передачу історії іншому користувачу, що забезпечує колективну роботу або передачу даних при зміні власника.

Аналітичні функції та рекомендації – система формує поради щодо оптимального часу проведення технічного обслуговування та контролю стану автомобіля на основі введених даних, що підвищує безпеку експлуатації.

Фінансовий облік – реалізовано ведення інформації про придбання, витрати та продаж автомобіля з можливістю формування звітів за категоріями витрат, що дозволяє користувачу аналізувати економічну ефективність експлуатації транспортного засобу.

Безпека та контроль доступу – автентифікація користувачів, розмежування прав доступу до функціоналу, захист передавання даних між фронтендом і сервером через HTTPS/SSL, а також контроль доступу на рівні окремих сутностей у базі даних.

Завдяки інтеграції ШІ-модулів, зокрема для розпізнавання тексту на чеках і документах, система автоматично формує структуровані дані для фінансового

обліку, зменшуючи ручну роботу користувача та підвищуючи точність введених даних.

Реалізація функціоналу системи демонструє поєднання сучасних технологій фронтенду, бекенду, баз даних та ШІ-сервісів, що дозволяє створити масштабований, безпечний та інтуїтивно зрозумілий програмний продукт. Архітектурні та технологічні рішення забезпечують основу для подальшого розвитку системи, інтеграції нових аналітичних модулів, ШІ-сервісів та маркетплейсу автотоварів без необхідності суттєвих змін в існуючій структурі.

Клас ExpensesService управляє витратами автомобілів, працюючи з базою даних через репозиторії Expense та ExpenseCategory. Він створює стандартні категорії витрат, перевіряє їх наявність та обчислює статистику витрат за автомобілем або користувачем, включно з сумою, розподілом по категоріях і по місяцях, забезпечуючи централізоване управління фінансами та зручне відображення на фронтенді.

Лістинг 3.1 файлу expenses.service.ts

```
export class ExpensesService {
  constructor(
    @InjectRepository(Expense)
    private expenseRepository: Repository<Expense>,
    @InjectRepository(ExpenseCategory)
    private categoryRepository: Repository<ExpenseCategory>,
  ) {}
   async getExpenseStatsForUser(
    userId: string,
    period?: { from: Date; to: Date },
    carId?: string,
  ): Promise<{
    totalAmount: number;
    byCategory: Array<{
      category: string;
      amount: number;
      percentage: number;
    }>;
    byMonth: Array<{ month: string; amount: number }>;
  }> {
    const queryBuilder = this.expenseRepository
      .createQueryBuilder('expense')
      .leftJoinAndSelect('expense.category', 'category')
      .leftJoin('expense.car', 'car')
```

```
      .where('car.userId = :userId', { userId });
    if (carId) {
      queryBuilder.andWhere('expense.carId = :carId', { carId });
    }
    if (period) {
      queryBuilder.andWhere('expense.expenseDate BETWEEN :from AND
:to', {
        from: period.from,
        to: period.to,
      });
    }
    const expenses = await queryBuilder.getMany();
    const totalAmount = expenses.reduce(
      (sum, expense) => sum + Number(expense.amount),
      0,
    );
    // Group by category
    const categoryMap = new Map<string, number>();
    expenses.forEach((expense) => {
      const categoryName = expense.category.name;
      const current = categoryMap.get(categoryName) || 0;
      categoryMap.set(categoryName, current +
Number(expense.amount));
    });
    const byCategory = Array.from(categoryMap.entries()).map(
      ([category, amount]) => ({
        category,
        amount,
        percentage: totalAmount > 0 ? (amount / totalAmount) * 100 :
      }),
    );
    // Group by month
    const monthMap = new Map<string, number>();
    expenses.forEach((expense) => {
      const date = new Date(expense.expenseDate);
      const month = `${date.getFullYear()}-${String(date.getMonth()
+ 1).padStart(2, '0')}`;
      const current = monthMap.get(month) || 0;
      monthMap.set(month, current + Number(expense.amount));
    });
    const byMonth = Array.from(monthMap.entries())
      .map(([month, amount]) => ({
        month,
        amount,
      }))
      .sort((a, b) => a.month.localeCompare(b.month));
    return {
      totalAmount,
      byCategory,
      byMonth,
    };
  }
}
```

Компонент AllExpensesPage реалізує сторінку перегляду всіх фінансових записів користувача у застосунку. Він дозволяє перемикатися між переглядом витрат та обслуговування автомобілів за допомогою сегментованого контролу (IonSegment), а також обирати період відображення даних (тиждень, місяць, квартали, рік або всі записи). Компонент динамічно обчислює діапазон дат на основі вибраного періоду і передає його відповідним спискам (AllExpensesList або AllMaintenanceList) для відображення відфільтрованих записів. Таким чином, сторінка забезпечує інтерактивний та зручний перегляд історії фінансових операцій та сервісних заходів у мобільному застосунку.

Лістинг 3.2 файлу AllExpensesPage.tsx

```
const AllExpensesPage = () => {
  const [segment, setSegment] = useState<"expenses" |
"maintenance">(
    "expenses"
  );
  const [period, setPeriod] = useState<string>("month");
  const getDateRange = (period: string) => {};
  const { from, to } = getDateRange(period);
  return (
    <IonPage>
      <IonHeader>
        <IonToolbar>
          <IonButtons slot="start">
            <IonBackButton defaultHref="/tabs/home" />
          </IonButtons>
          <IonTitle>Всі записи</IonTitle>
        </IonToolbar>
      </IonHeader>
      <IonContent>
        <div className="ion-padding">
          <IonSegment
            value={segment}
            onIonChange={(e) =>
              setSegment(e.detail.value as "expenses" |
"maintenance")
            }
          >
            <IonSegmentButton value="expenses">
              <IonLabel>Витрати</IonLabel>
            </IonSegmentButton>
            <IonSegmentButton value="maintenance">
              <IonLabel>Обслуговування</IonLabel>
```

```
            </IonSegmentButton>
          </IonSegment>

          <IonItem>
            <IonLabel>Період</IonLabel>
            <IonSelect
              value={period}
              onIonChange={(e) => setPeriod(e.detail.value)}
              interface="popover"
            >
              <IonSelectOption value="week">Останній
тиждень</IonSelectOption>
              <IonSelectOption value="month">Останній
місяць</IonSelectOption>
              <IonSelectOption value="3months">3
місяці</IonSelectOption>
              <IonSelectOption value="6months">6
місяців</IonSelectOption>
              <IonSelectOption value="year">Рік</IonSelectOption>
              <IonSelectOption value="all">Всі</IonSelectOption>
            </IonSelect>
          </IonItem>
        </div>
        {segment === "expenses" ? (
          <AllExpensesList from={from} to={to} />
        ) : (
          <AllMaintenanceList from={from} to={to} />
        )}
      </IonContent>
    </IonPage>
  );
};
```

Сервіс AiService у Nest.js призначений для автоматичної обробки зображень чеків та вилучення з них структурованих даних про витрати користувачів за допомогою GPT-4 Vision та OCR. Він перевіряє формат зображення, розпізнає текст, визначає суму, дату, продавця та категорію витрат, оцінює довіру до даних і позначає записи для ручної перевірки при потребі. Завдяки цьому сервіс інтегрується у систему, як повторно використовувана та масштабована компонента, забезпечуючи точність, автоматизацію та простоту тестування системи.

Лістинг 3.3 файл ai.service.ts

```
export class AiService {
  private async analyzeReceiptWithGPT4Vision(
    base64Image: string,
  ): Promise<AIResponse> {
    const prompt = `Analyze this receipt image and extract the
following information in JSON format:
{"amount": number (total amount),
  "date": "YYYY-MM-DD" format,
  "merchant": string (store/company name),
  "category": string (one of: "fuel", "maintenance", "carwash",
"parts", "insurance", "other"),
  "description": string (brief description),
  "items": array of items if visible
}
Important:
- Return only valid JSON
- If you can't find a field, use null
- For amount, use the total (not including change)
- Guess the most likely category based on merchant/items
- Be confident in your extraction`;
    try {
      const response = await fetch(this.openaiApiUrl, {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
          Authorization: `Bearer ${this.openaiApiKey}`,
        },
        body: JSON.stringify({
          model: 'gpt-4o-mini', // Using gpt-4o-mini for vision
(cheaper and faster)
          messages: [
            {
              role: 'user',
              content: [
                {
                  type: 'text',
                  text: prompt,
                },
                {
                  type: 'image_url',
                  image_url: {
                    url:
`${this.cleanBase64(base64Image)}`,
                  },
                },
              ],
            },
          ],
          max_tokens: 500,
          temperature: 0.2, // Low temperature for more consistent
results
```

```
        }),
      });
      if (!response.ok) {
        const errorData = await response.json().catch(() => ({}));
        this.logger.error('OpenAI API error:', errorData);
        throw new Error(`OpenAI API error: ${response.statusText}`);
      }
      const result = await response.json();
      const content = result.choices[0]?.message?.content;
      if (!content) {
        throw new Error('No content in OpenAI response');
      }
      // Extract JSON from response (GPT might add markdown code
blocks)
      const jsonMatch = content.match(/\{[\s\S]*\}/);
      if (!jsonMatch) {
        throw new Error('Could not extract JSON from response');
      }
      const extractedData: ExtractedReceiptData =
JSON.parse(jsonMatch[0]);
      // Calculate confidence based on completeness
      const confidence = this.calculateConfidence(extractedData);
      return {
        data: extractedData,
        confidence,
        rawText: content,
      };
    } catch (error) {
      this.logger.error(`GPT-4 Vision API error: ${error.message}`);
      throw error;
    }
  }
}
```

Наведений код демонструє реалізацію ключових сервісів бекенду та фронтенду застосунку, показуючи основний функціонал системи облік витрат та обслуговування автомобілів, фільтрацію даних за періодами, управління категоріями витрат та інтеграцію з сервісами штучного інтелекту для автоматичного розпізнавання чеків. Він ілюструє принципи роботи системи: структурованість через класи та сервіси, використання ін’єкції залежностей у Nest.js, обробку даних на сервері та їх відображення на фронтенді, автоматизацію аналізу та перевірки інформації, а також забезпечення розширюваності, масштабованості та контролю точності даних у межах програмного продукту.

3.2 Тестування системи та оцінка якості реалізації

Метою тестування розробленої системи є комплексна оцінка її працездатності, надійності та відповідності встановленим функціональним і нефункціональним вимогам. Тестування дозволяє визначити правильність роботи окремих модулів, стабільність взаємодії компонентів, коректність обробки даних користувача, а також ефективність алгоритмів і сервісів, що використовуються у системі. Особливу увагу приділено перевірці механізмів безпеки, включаючи автентифікацію користувачів, контроль доступу до даних, захист від несанкціонованих дій та забезпечення цілісності інформації при взаємодії клієнтської та серверної частин.

Завдання тестування передбачають виявлення помилок у роботі окремих модулів та інтеграції між ними, перевірку відповідності реалізованого функціоналу вимогам користувача та технічним специфікаціям, оцінку безпеки системи, стабільності та надійності роботи серверної частини під навантаженням, а також аналіз зручності та інтуїтивності користувацького інтерфейсу і оцінку продуктивності та швидкодії системи при обробці великих обсягів даних.

Реалізація цих завдань забезпечує високу якість програмного продукту, знижує ризик виникнення критичних помилок під час експлуатації, підвищує безпеку та надійність системи, а також створює підґрунтя для подальшого масштабування, інтеграції додаткових сервісів і впровадження нових функціональних модулів. Комплексний підхід до тестування забезпечує контроль якості на всіх рівнях розробки, від клієнтської частини з PWA та мобільними додатками на платформі Ionic React до серверної частини з використанням Nest.js,

PostgreSQL та інтеграції з сервісами штучного інтелекту. Цілі тестування полягали у забезпеченні надійності системи, своєчасного виявлення помилок у логіці обробки даних, перевірки безпеки та точності взаємодії між сервісами бекенду. Завдання тестування включали контроль коректності запису та читання даних із бази, перевірку дотримання правил автентифікації та авторизації, а також оцінку стійкості системи до некоректних запитів.

Для досягнення цих цілей використовувались юніт-тести, що охоплюють основні сервіси: UsersService, AuthService, CarsService, MileageService, ExpensesService та MaintenanceService. Тести перевіряли правильність обробки CRUD-операцій, зв’язки між сутностями, валідацію даних і обробку виключень. Крім того, проведено інтеграційне тестування REST API для впевненості у коректності обміну даними між фронтендом та серверною частиною, з перевіркою поведінки при різних сценаріях використання [8].

Функціональне тестування проводилось вручну (manual testing) з метою підтвердження логіки роботи бекенду під час виконання реальних сценаріїв користувача, таких як додавання витрати, реєстрація нового автомобіля, оновлення пробігу та формування рекомендацій щодо обслуговування. Цей підхід дозволив оцінити не лише технічну коректність, а й відповідність функціоналу очікуванням кінцевого користувача [9].

На головній сторінці користувач може обрати активний автомобіль та переглянути його ключову статистику, включаючи поточний пробіг та витрати за місяць, а також отримати попередження про наближення технічного обслуговування. За допомогою плаваючого меню швидких дій користувач має змогу оперативно додавати нові записи про пробіг, витрати та технічне обслуговування обраного автомобіля, а також переглядати повну історію всіх витрат (рис. 3.1).

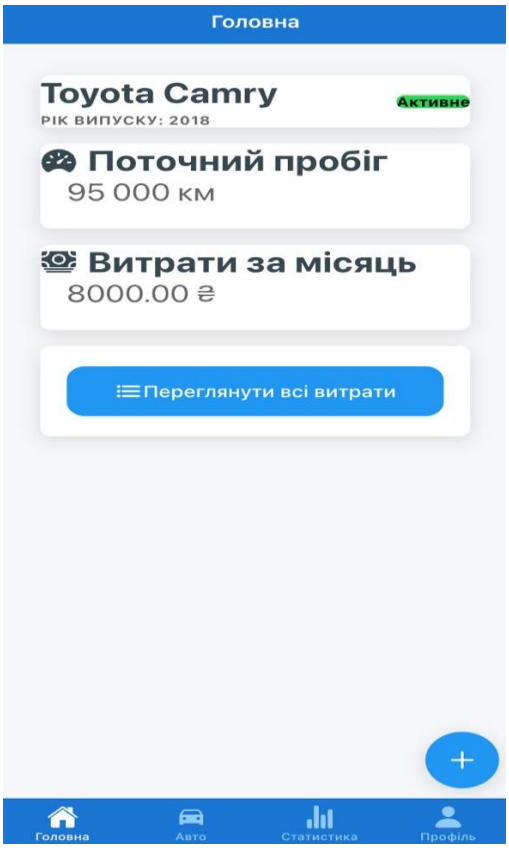


Рисунок 3.1 – Головна сторінка для користувачів

На сторінці статистики користувач має можливість фільтрувати дані про витрати за вибраним автомобілем або переглядати зведену інформацію по всіх автомобілях, а також обирати часовий період аналізу (місяць, три місяці, шість місяців або рік). Система відображає загальну суму витрат за обраний період, детальний розподіл витрат по категоріях з процентним співвідношенням та візуальними індикаторами, а також помісячну динаміку витрат для аналізу тенденцій (рис. 3.2).

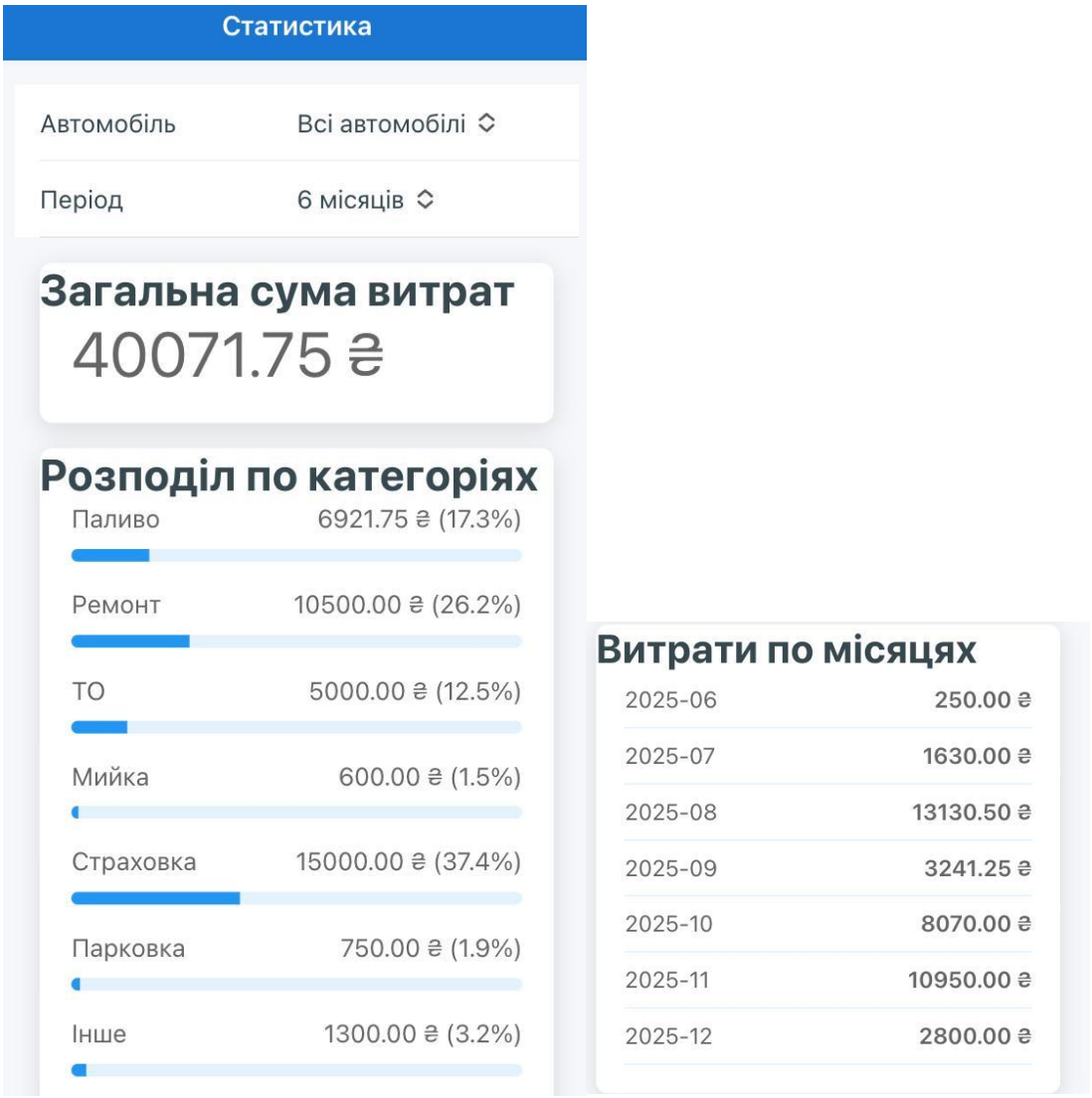


Рисунок 3.2 – Сторінка статистики

На сторінці додавання витрат користувач заповнює форму з обов'язковими полями (категорія витрати, сума та дата) і може додати опціональний опис, а також завантажити до 5 фотографій чеків через камеру або галерею з автоматичною валідацією якості зображень, яка блокує збереження у разі наявності помилкових фото. Сторінка додавання пробігу дозволяє ввести нове значення одометра з автоматичним відображенням поточного пробігу для контроля, обрати дату запису та додати необов'язковий коментар, що забезпечує зручне ведення історії експлуатації автомобіля (рис. 3.3).

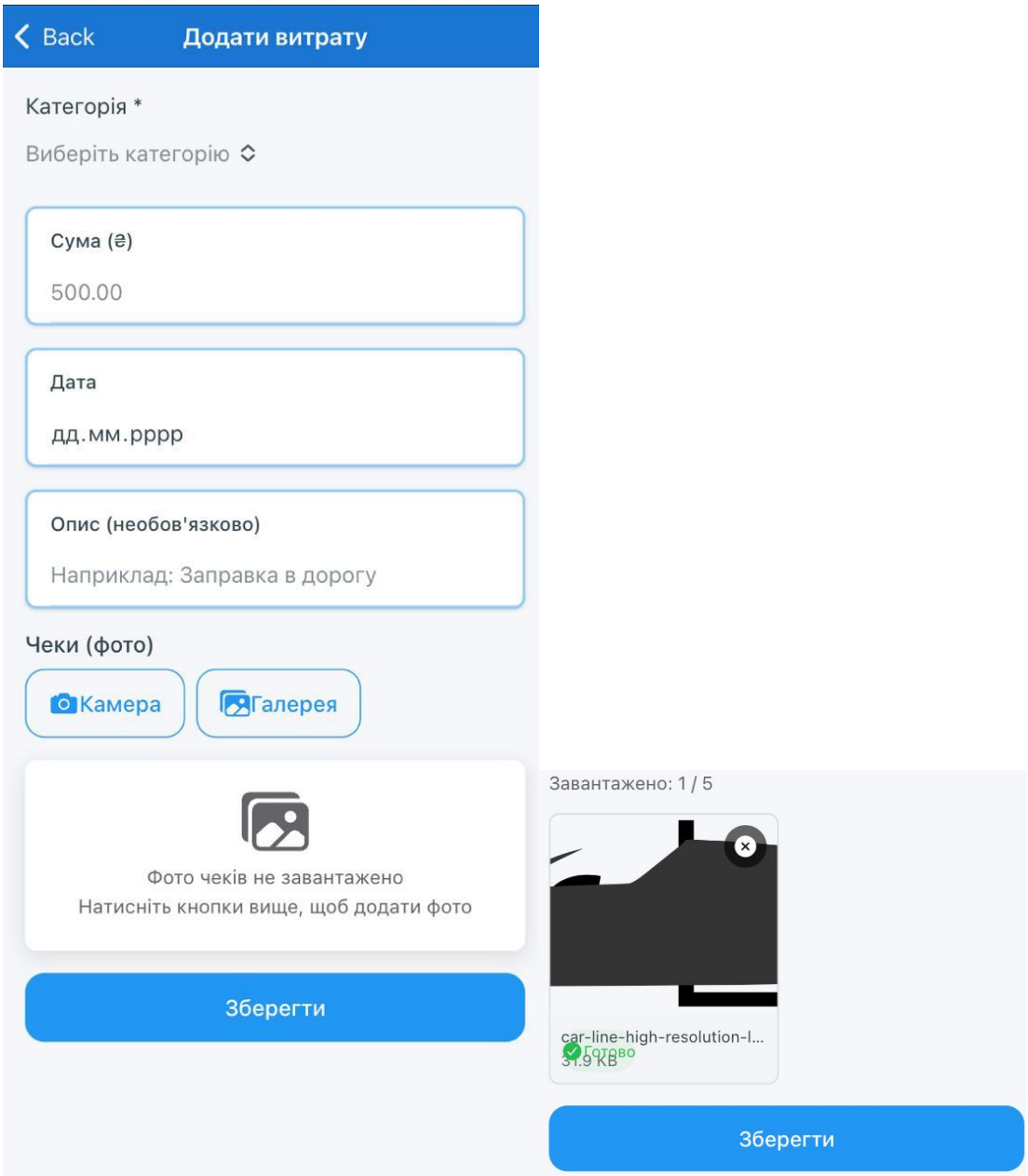


Рисунок 3.3 – Сторінка додавання витрат

Компонент завантаження чеків дозволяє користувачу додавати фотографії через камеру пристрою або вибирати з галереї (до 5 фото), після чого кожне зображення автоматично проходить симуляцію валідації протягом 1-3 секунд У разі виявлення проблем з якістю фото (розмитість, погане освітлення, неповне зображення чеку, неможливість розпізнати суму тощо) система відображає детальне повідомлення про помилку і блокує збереження форми до видалення проблемних фотографій, забезпечуючи таким чином контроль якості даних (рис. 3.4).

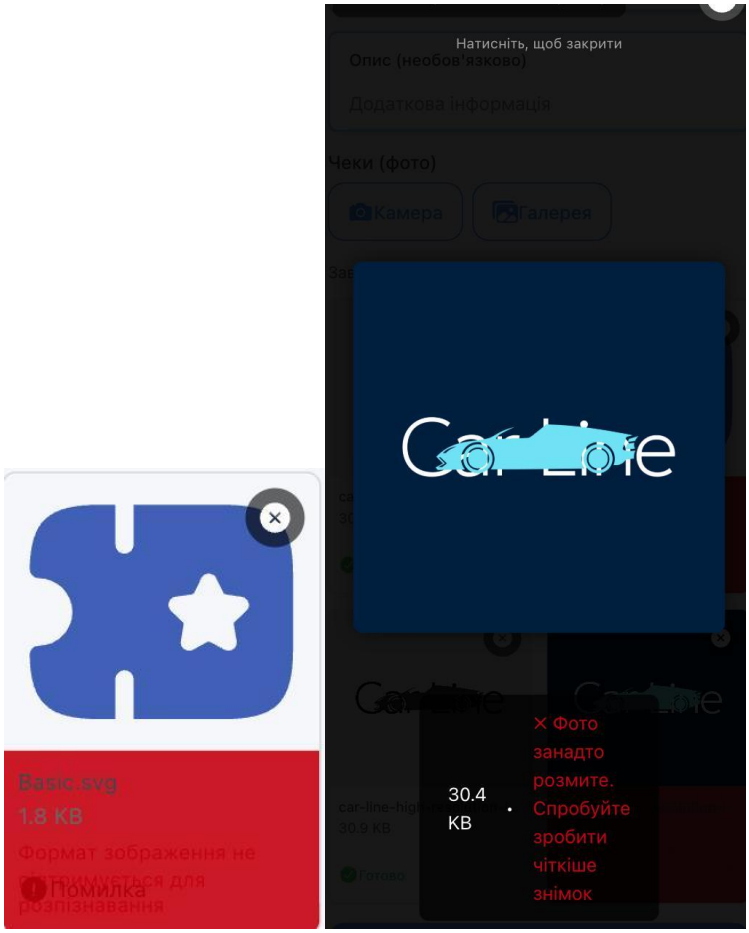


Рисунок 3.4 – Вигляд завантаження чеків

На сторінці всіх записів користувач може перемикатися між переглядом витрат та технічного обслуговування через сегментоване меню, а також фільтрувати дані за часовим періодом (тиждень, місяць, 3/6/12 місяців або всі записи), конкретним автомобілем та категорією витрат. Система автоматично групує записи за датами для зручності сприйняття і надає можливість видаляти окремі записи через свайп-жест, що забезпечує швидкий доступ до повної історії операцій з можливістю детальної фільтрації (рис. 3.5).

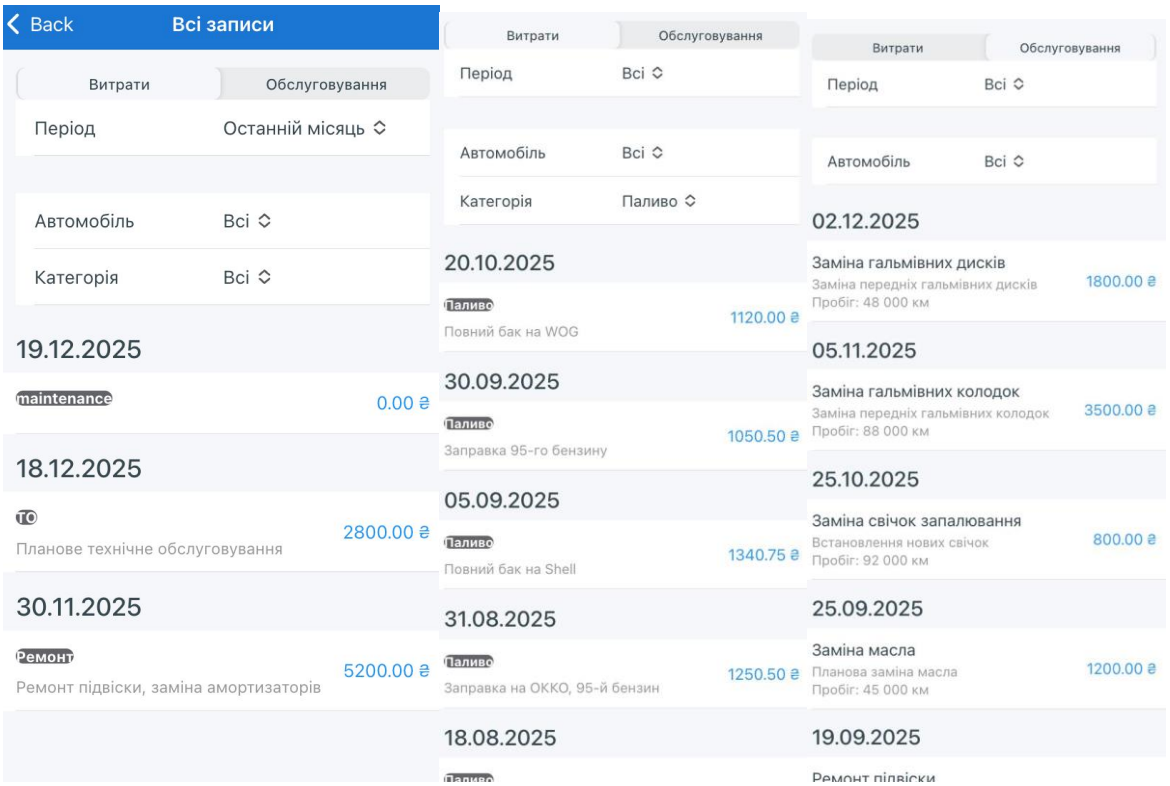


Рисунок 3.5 – Сторінка записів

На сторінці деталей автомобіля відображається повна інформація про обраний транспортний засіб, включаючи марку, модель, рік випуску, початковий пробіг та дату додавання до системи, а також статус активності (активний автомобіль позначається спеціальним бейджем). Користувач має можливість встановити автомобіль як активний для відображення на головній сторінці або видалити його з системи через відповідні кнопки управління, при цьому всі операції супроводжуються діалоговими вікнами підтвердження та сповіщеннями про результат виконання (рис. 3.6).

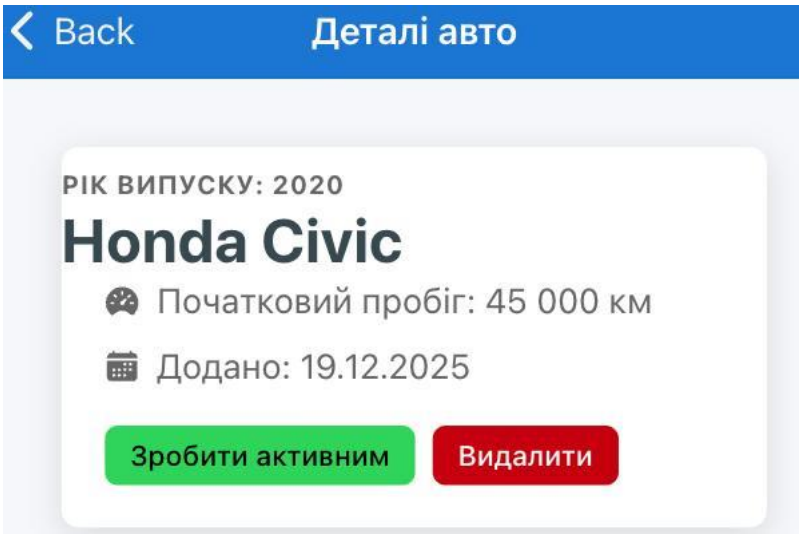


Рисунок 3.6 – Вигляд завантаження чеків

Протестовано основні функціональні можливості веб-застосунку для управління автомобілями. Головна сторінка забезпечує швидкий доступ до ключової статистики активного автомобіля з можливістю оперативного додавання записів через плаваюче меню. Сторінка статистики дозволяє проводити аналіз витрат за різними критеріями (автомобіль, період, категорія) з візуалізацією розподілу по категоріях та помісячною динамікою.

Проведення комплексного тестування є обов’язковою складовою процесу розробки програмного забезпечення, оскільки саме на цьому етапі забезпечується перевірка коректності та стабільності реалізованих рішень. Unit-тестування відіграє ключову роль у виявленні помилок на ранніх стадіях розробки, підвищенні надійності системи, забезпеченні її безпеки та підтвердженні відповідності функціоналу поставленим вимогам. Застосування такого підходу дозволяє суттєво знизити ризики виникнення критичних збоїв під час експлуатації та сприяє створенню якісного програмного продукту, орієнтованого на потреби кінцевих користувачів на (Рис. 3.7) наведено приклад візуалізації роботи Unit-тестів у межах розробленої системи.

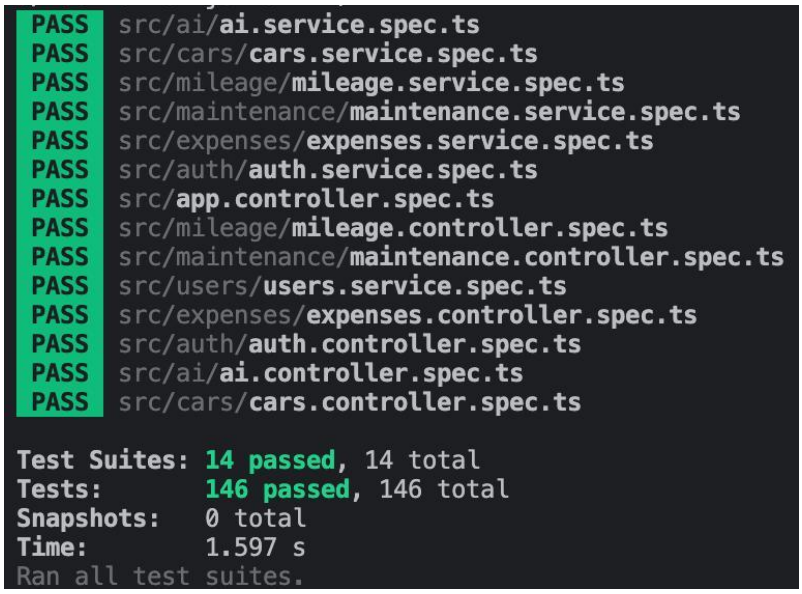


Рисунок 3.7 – Unit-теси

3.3 Аналіз результатів тестування та верифікація вимог

У межах даного розділу було проведено узагальнений аналіз результатів тестування розробленої інформаційної системи та виконано верифікацію її відповідності початково сформульованим функціональним і нефункціональним вимогам. Тестування виступало завершальним і водночас одним із найважливіших етапів розробки, оскільки саме на цьому кроці підтверджується коректність реалізації бізнес-логіки, стабільність роботи системи та готовність програмного продукту до практичного використання.

У процесі розробки застосунку було використано поєднання unit-тестування та ручне-тестування, що дозволило комплексно оцінити якість реалізації як окремих програмних компонентів, так і системи в цілому. Unit-тести застосовувалися переважно на рівні серверної частини (backend) та були спрямовані на перевірку коректності роботи сервісів, методів обробки даних, бізнес-логіки, а також взаємодії з базою даних. Особливу увагу було приділено таким модулям, як облік витрат, формування статистики, робота з категоріями, фільтрація даних за періодами та обробка запитів користувача. Результати unit-тестування підтвердили, що основні функції системи працюють стабільно,

повертають очікувані результати та коректно обробляють граничні й нестандартні ситуації.

Ручне-тестування, у свою чергу, застосовувалося для перевірки цілісного функціонування системи з точки зору кінцевого користувача. У ході ручного тестування перевірялася логіка навігації, коректність взаємодії між клієнтською та серверною частинами, правильність відображення даних, а також відповідність реалізованого функціоналу сценаріям використання, визначеним на етапі аналізу вимог. Особлива увага приділялася перевірці сценаріїв додавання, перегляду та фільтрації витрат і записів обслуговування, роботі з періодами часу, а також використанню допоміжного функціоналу, зокрема аналізу чеків за допомогою AI-сервісу.

Окремо було здійснено верифікацію вимог, визначених у технічному завданні. У результаті проведеного тестування встановлено, що всі ключові функціональні вимоги реалізовані в повному обсязі. Система забезпечує можливість ведення історії витрат та обслуговування автомобіля, формування статистичних зведень, фільтрацію даних за різними періодами, а також підтримує розширений функціонал аналізу даних із використанням інтелектуальних алгоритмів. Нефункціональні вимоги, зокрема вимоги до стабільності, логічної узгодженості та зручності використання, також були дотримані, що підтверджується результатами ручного тестування та відсутністю критичних помилок під час експлуатаційних сценаріїв.

За результатами аналізу тестування можна зробити висновок, що розроблена система є працездатною, стабільною та відповідає початковим вимогам, сформульованим на етапі проєктування. Поєднання unit-тестування та manual-тестування дозволило виявити та усунути потенційні недоліки на ранніх етапах, що позитивно вплинуло на загальну якість програмного продукту. Таким чином, проведене тестування підтверджує готовність системи до подальшого використання та можливого розширення функціональних можливостей у майбутньому.

4 БЕЗПЕКА ЖИТТЄДІЯЛЬНОСТІ ТА ОСНОВИ ОХОРОНИ ПРАЦІ

У даному розділі розглядаються питання охорони праці та безпеки життєдіяльності під час виконання робіт у сфері інформаційних технологій. Аналізуються потенційні небезпечні та шкідливі фактори, що можуть виникати під час розробки та експлуатації програмного забезпечення. Також висвітлюються заходи цивільного захисту та запобігання надзвичайним ситуаціям техногенного характеру.

4.1 Охорона праці

Охорона праці під час розробки багатоплатформеного застосунку для моніторингу стану та витрат обслуговування автомобіля є важливою складовою професійної діяльності фахівця з програмної інженерії та спрямована на створення безпечних, комфортних і нешкідливих умов праці, а також на збереження здоров’я і працездатності розробника. У процесі виконання магістерської кваліфікаційної роботи основні завдання були пов’язані з проєктуванням, розробкою та тестуванням програмного застосунку з використанням персонального комп’ютера, ноутбука та мобільних пристроїв, що зумовлює необхідність дотримання вимог охорони праці, електробезпеки, пожежної безпеки й ергономіки робочого місця.

Правові та організаційні засади забезпечення охорони праці в Україні визначаються Законом України «Про охорону праці» [21], який регламентує обов’язки щодо створення безпечних умов праці, запобігання професійним захворюванням і мінімізації впливу шкідливих та небезпечних виробничих факторів. Відповідно до вимог цього закону, під час організації роботи розробника програмного забезпечення мають враховуватися особливості інтелектуальної праці, тривале використання комп’ютерної техніки, статичні навантаження та психоемоційне напруження.

Під час розробки багатоплатформеного застосунку основними потенційно небезпечними та шкідливими факторами є: тривале статичне навантаження на

опорно-руховий апарат; підвищене навантаження на органи зору внаслідок роботи з екранними пристроями; психоемоційна напруга, пов’язана з програмуванням і тестуванням функціональності; ризик ураження електричним струмом під час експлуатації електронного обладнання; пожежна небезпека, обумовлена використанням електромережі та комп’ютерної техніки.

Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями встановлюються нормативним документом НПАОП 0.00-7.15-18 [22]. Даний документ визначає обов’язкові вимоги до організації робочого місця, параметрів екранних пристроїв, режимів праці та відпочинку, а також заходів щодо зменшення негативного впливу тривалої роботи за комп’ютером на здоров’я працівників.

Згідно з вимогами НПАОП 0.00-7.15-18, робота з екранними пристроями повинна бути організована таким чином, щоб забезпечити оптимальні умови зорового сприйняття інформації, зменшити м’язове напруження та запобігти перевтомі. Це досягається шляхом раціонального планування робочого часу, дотримання регламентованих перерв і правильного розміщення технічних засобів [22].

Організація робочого місця розробника багатоплатформеного застосунку має відповідати будівельним і санітарно-гігієнічним вимогам. Відповідно до ДБН В.2.2-28:2010 [23], робоче місце з персональним комп’ютером повинно забезпечувати достатню площу для розміщення обладнання, можливість вільного переміщення працівника та комфортні умови для виконання професійних обов’язків.

Ергономічні вимоги до взаємодії людини з комп’ютерними системами визначаються стандартами серії ДСТУ ISO 9241 [24]. Вони регламентують розміщення монітора, клавіатури, маніпулятора типу «миша», робочого столу та крісла. Елементи робочого місця повинні бути налаштовані так, щоб забезпечувати природне положення тіла та зменшувати навантаження на хребет і верхні кінцівки.

Важливим чинником безпеки та працездатності є освітлення робочого місця. Вимоги до природного і штучного освітлення визначені ДБН В.2.5-28:2018 [25],

який встановлює нормативні показники освітленості, рівномірність освітлення та обмеження засліплюючої дії світильників. Дотримання цих вимог сприяє зниженню зорової втоми та підвищенню ефективності праці.

Для зменшення негативного впливу тривалої роботи з екранними пристроями необхідно дотримуватися раціонального режиму праці та відпочинку. Згідно з вимогами НПАОП 0.00-7.15-18, робота за комп'ютером повинна чергуватися з перервами, під час яких рекомендовано виконувати вправи для очей і короткі фізичні розминки [22].

Під час експлуатації комп'ютерної та мобільної техніки необхідно дотримуватися вимог електробезпеки, визначених Правилами улаштування електроустановок [26] та Правилами технічної експлуатації електроустановок споживачів [27]. Усі електричні пристрої повинні бути справними, мати надійну ізоляцію та заземлення. Забороняється використання пошкоджених кабелів і несправних розеток.

Вимоги пожежної безпеки під час виконання робіт з розробки програмного забезпечення регламентуються Правилами пожежної безпеки в Україні [28]. Для запобігання пожежам необхідно використовувати сертифіковане електрообладнання, не допускати перевантаження електромережі та забезпечувати наявність первинних засобів пожежогасіння.

Таким чином, дотримання вимог чинних нормативно-правових актів України у сфері охорони праці під час розробки багатоплатформеного застосунку для моніторингу стану та витрат обслуговування автомобіля забезпечує безпечні умови праці, знижує вплив шкідливих факторів і сприяє збереженню здоров'я та працездатності розробника.

4.2 Організація цивільного захисту на об’єктах промисловості та виконання заходів щодо запобігання виникненню надзвичайних ситуацій техногенного походження

У сучасних умовах ефективне управління технічним станом транспортних засобів і контроль витрат на їх обслуговування є важливою складовою безпеки на промислових об’єктах. Тема магістерської роботи «Розробка багатоплатформеного застосунку для моніторингу стану та витрат обслуговування автомобіля» передбачає створення інструментів, які дозволяють забезпечити постійний контроль за автопарком підприємств, прогнозувати можливі несправності та оптимізувати витрати на обслуговування. Інтеграція таких систем у загальну організацію цивільного захисту на об’єктах промисловості дозволяє підвищити рівень безпеки персоналу, запобігати надзвичайним ситуаціям техногенного характеру та забезпечити безперервне функціонування виробничих процесів.

Цивільний захист на промислових об’єктах, включно з підприємствами та сервісними центрами автотранспорту, є ключовим елементом забезпечення безпеки держави та безперебійного функціонування критичної інфраструктури. Основною метою ЦЗ є збереження життя і здоров’я персоналу, населення та майна, а також мінімізація ризику виникнення надзвичайних ситуацій техногенного характеру, які можуть впливати на роботу автопарків і підприємств, що обслуговують транспортні засоби. Ефективна організація ЦЗ передбачає комплекс заходів: підготовку персоналу, створення спеціалізованих формувань, впровадження інженерно-технічних рішень і систем раннього оповіщення, а також планування евакуаційних заходів і оперативне реагування на надзвичайні ситуації. У цьому контексті особливе значення має моніторинг стану автомобільного транспорту та витрат на його обслуговування, що дозволяє прогнозувати можливі відмови техніки, уникати аварій та забезпечувати безперебійну роботу об’єкта [30].

Організація цивільного захисту на об’єктах промисловості, включаючи автосервіси та логістичні підприємства, здійснюється за типовою структурою з урахуванням специфіки виробництва та обслуговування транспортних засобів.

Керівник об’єкта несе повну відповідальність за стан ЦЗ, управління силами та засобами, а також організацію аварійно-рятувальних і невідкладних робіт. Він забезпечує персонал засобами індивідуального та колективного захисту, організовує евакуаційні заходи, формує спеціалізовані сили для ліквідації наслідків надзвичайних ситуацій і підтримує їх готовність до практичних дій. Додатково створюються диспетчерські служби та впроваджуються системи моніторингу технічного стану автомобілів, що дозволяє своєчасно визначати несправності, оптимізувати витрати на обслуговування та підвищувати безпеку руху техніки на підприємстві [30].

На потенційно небезпечних підприємствах, які обслуговують автотранспорт або працюють із паливно-мастильними матеріалами, впроваджуються локальні автоматизовані системи раннього виявлення загрози та оповіщення персоналу. Додатково здійснюються інженерно-технічні заходи для зменшення ризику аварій і пожеж, а власники об’єктів відповідають за захист населення, що перебуває в зоні можливого ураження. Використання систем моніторингу стану автомобілів дозволяє контролювати технічний стан автопарку, запобігати поломкам і знижувати ризики аварій на території підприємства [30].

Для виконання спеціальних заходів ЦЗ створюються служби на базі структурних підрозділів підприємства. У кризових ситуаціях можуть формуватися невоєнізовані формування, включаючи рятувальні команди, аварійно-технічні бригади, розвідувальні групи, протипожежні та санітарні дружини. Основними формуваннями загального призначення є рятувальні та зведені рятувальні загони, що можуть оперативно реагувати на аварії, у тому числі пов’язані з технічним станом автомобілів [30].

Запобігання надзвичайним ситуаціям забезпечується через інженерно-технічні заходи, постійний моніторинг стану об’єктів і навколишнього середовища, а також прогнозування соціально-економічних наслідків можливих НС. Системи оповіщення персоналу й населення підтримуються у готовності до негайного застосування, створюються локальні системи сповіщення про загрози. Спеціалізовані формування забезпечуються засобами індивідуального захисту,

проводиться будівництво захисних споруд. Одночасно сучасні багатоплатформені застосунки для моніторингу стану автомобілів дозволяють оцінювати витрати на обслуговування, планувати технічні перевірки та уникати ризиків, що можуть стати причиною надзвичайних ситуацій. Підтримується готовність органів влади та підпорядкованих сил до дій, спрямованих на запобігання і реагування на НС [30].

Евакуаційні заходи є одним із ключових способів захисту персоналу та населення. Вони передбачають завчасне вивезення людей із зон можливого ураження у безпечні райони. Евакуація може бути загальною або частковою, обов'язковою або тимчасовою. Органи з евакуації формуються на державному, регіональному, місцевому та об'єктовому рівнях, плануються маршрути, визначаються безпечні райони, організовується збір, реєстрація та розселення населення. Переміщення здійснюється транспортом або пішки, з урахуванням пунктів медичної допомоги, привалів та контролю за рухом колон. Водночас системи моніторингу автотранспорту дозволяють забезпечувати безпечне використання техніки під час евакуаційних заходів, координувати рух автоколон та підтримувати працездатність критичного автопарку підприємства [30].

Отже, організація цивільного захисту на об'єктах промисловості, що включає підприємства автосервісу та логістики, є комплексним процесом, який інтегрує підготовку персоналу, створення спеціалізованих формувань, впровадження інженерно-технічних заходів і систем раннього оповіщення, а також планування і проведення евакуаційних заходів. Використання багатоплатформеного моніторингу стану автомобілів і витрат на їх обслуговування дозволяє мінімізувати ризики надзвичайних ситуацій, забезпечити безпеку персоналу та ефективне функціонування об'єктів промисловості навіть у складних умовах [30].

ВИСНОВКИ

У результаті виконання магістерської кваліфікаційної роботи було розроблено багатоплатформенний програмний застосунок для моніторингу стану автомобіля та обліку витрат на його обслуговування. Розроблена система орієнтована на підвищення зручності користування, автоматизацію процесів ведення історії експлуатації автомобіля та надання аналічної інформації щодо витрат і технічного обслуговування.

У першому розділі виконано детальний аналіз предметної області, пов’язаної з експлуатацією та обслуговуванням автомобілів. Було досліджено існуючі підходи до обліку витрат і технічних робіт, визначено основні проблеми та обмеження наявних рішень. На основі проведеного аналізу сформульовано мету та завдання роботи, визначено функціональні й нефункціональні вимоги до програмної системи, а також описано основних акторів і варіанти використання. Це дозволило сформувати чітке уявлення про очікувану поведінку системи та вимоги до її якості.

У другому розділі було виконано проєктування програмної системи. Обґрунтовано вибір процесу розробки та архітектурних рішень, спроєктовано клієнтську і серверну частини застосунку з використанням сучасних підходів до побудови програмних систем. Особливу увагу приділено архітектурі бекенд-частини на основі NestJS із використанням принципів MVC, а також організації фронтенду відповідно до підходу Feature-Sliced Design. Було спроєктовано структуру бази даних на базі PostgreSQL, визначено основні сутності та їх взаємозв’язки, а також побудовано UML-діаграми класів і діаграми послідовностей, що відображають логіку взаємодії компонентів системи. У розділі також здійснено оцінку обраних технологічних рішень і їх доцільність для реалізації поставлених завдань.

У третьому розділі реалізовано основний функціонал системи, включаючи облік витрат, ведення історії технічного обслуговування, формування статистики та аналітичних показників, а також інтеграцію ШІ-модуля для розпізнавання інформації з фотографій чеків. Клієнтська частина була реалізована з

використанням Ionic React, що забезпечило кросплатформенність та можливість використання застосунку як у мобільному, так і у вебсередовищі. Проведено тестування серверної частини системи із застосуванням як ручного, так і модульного (unit) тестування, що дозволило перевірити коректність роботи основних бізнес-логік, стабільність системи та її відповідність початково визначеним вимогам. Аналіз результатів тестування підтвердив правильність реалізації функціоналу та відповідність системи функціональним і нефункціональним вимогам.

У результаті виконання кваліфікаційної роботи магістра було створено сучасний багатоплатформенний застосунок для моніторингу стану та витрат обслуговування автомобіля, що поєднує зручний користувацький інтерфейс, надійну серверну архітектуру та аналічні можливості. Система реалізована з використанням технологій Ionic React, Node.js, NestJS та PostgreSQL, що забезпечує її масштабованість, продуктивність і готовність до подальшого розвитку. Отримані результати підтверджують доцільність обраних технологічних рішень і можуть бути використані як основа для впровадження програмного продукту в реальних умовах експлуатації, а також для подальшого розширення функціональних можливостей системи

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. JavaScript [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Web/JavaScript.
2. TypeScript is JavaScript with syntax for types [Електронний ресурс]. – Режим доступу: https://www.typescriptlang.org.
3. About Node.js [Електронний ресурс]. – Режим доступу: https://nodejs.org/en/about.
4. NestJS Documentation [Електронний ресурс]. – Режим доступу: https://docs.nestjs.com/.
5. PostgreSQL Documentation [Електронний ресурс]. – Режим доступу: https://www.postgresql.org/docs/current/.
6. TypeORM Documentation [Електронний ресурс]. – Режим доступу: https://typeorm.io.
7. REST API як спосіб спілкування компонент веб-додатків [Електронний ресурс]. – 2023. – Режим доступу: https://foxminded.ua/shcho-take-rest-api/.
8. Jest Documentation [Електронний ресурс]. – Режим доступу: https://jestjs.io/docs/getting-started.
9. Manual Testing in Software Testing [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/software-testing/software-testing-manual-testing/.
10. UML Documentation [Електронний ресурс]. – Режим доступу: http://www.uml.org/.
11. Relationships in SQL [Електронний ресурс]. – Режим доступу: https://blog.devart.com/types-of-relationships-in-sql-server-database.html
12. OpenAI. GPT-4 Vision Guide [Електронний ресурс]. – Режим доступу: https://platform.openai.com/docs/guides/images-vision
13. Unit Testing in Software Testing [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/software-testing/unit-testing-software-testing/

14. SSL [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/SSL

15. HTTPS [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/HTTPS

16. Decimal (Numerical Representation) [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%9E%D0%BF%D1%82%D0%B8%D1%87%D0%BD%D0%B5_%D1%80%D0%BE%D0%B7%D0%BF%D1%96%D0%B7%D0%BD%D0%B0%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F_%D1%81%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%96%D0%B2

17. Feature-Sliced Design: Overview [Електронний ресурс]. – Режим доступу: https://feature-sliced.design/docs/get-started/overview

18. CI/CD Concepts [Електронний ресурс]. – Режим доступу: https://itedu.center/ua/blog/review/what-is-ci-cd/?srsltid=AfmBOopCDy0mQhD4DhiOGTwsbfl-Dtdk5LbLG8z69Em8kJWYrmIhQ4XK

19. Ionic Framework React Quickstart [Електронний ресурс]. – Режим доступу: https://ionicframework.com/docs/react/quickstart

20. Progressive Web Apps (PWA) [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/enUS/docs/Web/Progressive_web_apps

21. Закон України «Про охорону праці» від 14.10.1992 № 2694-XII (зі змінами). URL: https://zakon.rada.gov.ua/laws/show/2694-12 (дата звернення: 16.12.2025).

22. НПАОП 0.00-7.15-18. Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями : затв. наказом Мінсоцполітики України від 14.02.2018 № 207. URL: https://zakon.rada.gov.ua/laws/show/z0508-18 (дата звернення: 16.12.2025).

23. ДБН В.2.2-28:2010. Будинки і споруди. Будинки адміністративного побутового призначення. https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=27263 (дата звернення: 16.12.2025).

24. ДСТУ ISO 9241 (серія). Ергономіка взаємодії людина-система (різні частини). URL: https://online.budstandart.com/ua (пошук за серією стандартів) (дата звернення: 16.12.2025).

25. ДБН В.2.5-28:2018. Природне і штучне освітлення : затв. наказом Мінрегіону України від 03.10.2018 № 264. Київ : Укрархбудінформ, 2018. URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=79885 звернення: 16.12.2025)

26. Правила улаштування електроустановок (ПУЕ) : затв. наказом Міненерговугілля України від 21.07.2017 № 476 (нова редакція). URL: https://zakon.rada.gov.ua/laws/show/v0476732-17 (дата звернення: 16.12.2025).

27. Правила технічної експлуатації електроустановок споживачів (ПТЕЕС) : затв. наказом Мінпаливенерго України від 25.07.2006 № 258. URL: https://zakon.rada.gov.ua/laws/show/z1143-06 (дата звернення: 16.12.2025).

28. НАПБ А.01.001-2014. Правила пожежної безпеки в Україні : затв. наказом МВС України від 30.12.2014 № 1417. URL: https://online.budstandart.com/ua/catalog/doc-page.html?id_doc=60541 (дата звернення: 16.12.2025).

29. Стручок В. С. Безпека в надзвичайних ситуаціях : методичний посібник для здобувачів освітнього ступеня «магістр» всіх спеціальностей денної та заочної (дистанційної) форм навчання. Тернопіль : ФОП Паляниця В. А., 2022. 156 с. URL: https://elartu.tntu.edu.ua/handle/lib/39196 (дата звернення: 19.12.2025).

30. Стручок В. С. Техноекологія та цивільна безпека. Частина «Цивільна безпека» : навчальний посібник. Тернопіль : ФОП Паляниця В. А., 2022. 156 с. URL: http://elartu.tntu.edu.ua/handle/lib/39424 (дата звернення: 19.12.2025).

31. Guide to the Software Engineering Body of Knowledge (SWEBOK Guide). Version 4.0 / ed. H. Washizaki. IEEE Computer Society, 2024. 411 p. https://ieeecs-media.computer.org/media/education/swebok/swebok-v4.pdf

32. Tailwind CSS Installation using Vite [Електронний ресурс]. – Режим доступу: https://tailwindcss.com/docs/installation/using-vite.

33. Firebase Hosting Documentation [Електронний ресурс]. – Режим доступу: https://firebase.google.com/docs/hosting.

34. Firebase Authentication Documentation [Електронний ресурс]. – Режим доступу: https://firebase.google.com/docs/auth.

35. FirebaseUI for iOS Authentication Documentation [Електронний ресурс]. – Режим доступу: https://firebase.google.com/docs/auth/ios/firebaseui.

36. Cloud Firestore Documentation [Електронний ресурс]. – Режим доступу: https://firebase.google.com/docs/firestore.

37. React Reference Documentation [Електронний ресурс]. – Режим доступу: https://react.dev/reference/react.

38. Zustand Getting Started Documentation [Електронний ресурс]. – Режим доступу: https://zustand.docs.pmnd.rs/getting-started/introduction.

39. Мінімально життєздатний продукт [Електронний ресурс]. – Режим доступу: https://uk.wikipedia.org/wiki/%D0%9C%D1%96%D0%BD%D1%96%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE_%D0%B6%D0%B8%D1%82%D1%82%D1%94%D0%B7%D0%B4%D0%B0%D1%82%D0%BD%D0%B8%D0%B9_%D0%BF%D1%80%D0%BE%D0%B4%D1%83%D0%BA%D1%82.

40. GitHub Actions Documentation [Електронний ресурс]. – Режим доступу: https://docs.github.com/en/actions.

41. Optical Character Recognition Using Node.js and Tesseract OCR Engine [Електронний ресурс]. – Режим доступу: https://dev.to/oluwatobi2001/optical-character-recognition-using-node-js-and-tesseract-ocr-engine-1ab.

42. Docker Guides Documentation [Електронний ресурс]. – Режим доступу: https://docs.docker.com/guides/.

43. Capacitor Documentation [Електронний ресурс]. – Режим доступу: https://capacitorjs.com/docs.

44. Model View Controller docs [Електронний ресурс]. – Режим доступу: https://developer.mozilla.org/en-US/docs/Glossary/MVC#mvc_on_the_web

45. Firebase Storage [Електронний ресурс]. – Режим доступу: https://firebase.google.com/docs/storage

ДОДАТКИ

Лістинг коду 1 – файл main.ts

```
import { NestFactory } from '@nestjs/core';
import { ValidationPipe } from '@nestjs/common';
import { AppModule } from './app.module';
import { ExpensesService } from './expenses/expenses.service';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);

  app.useGlobalPipes(
    new ValidationPipe({
      whitelist: true,
      forbidNonWhitelisted: true,
      transform: true,
    }),
  );

  const frontendUrl = process.env.FRONTEND_URL ||
'http://localhost:3000';
  app.enableCors({
    origin: frontendUrl,
    credentials: true,
    methods: ['GET', 'POST', 'PUT', 'DELETE', 'PATCH', 'OPTIONS'],
    allowedHeaders: ['Content-Type', 'Authorization'],
  });

  // Seed expense categories
  const expensesService = app.get(ExpensesService);
  await expensesService.seedCategories();
  console.log('✅ Expense categories seeded');

  const port = process.env.PORT ?? 3001;
  await app.listen(port);
  console.log(`🚀 Application is running on:
http://localhost:${port}`);
  console.log(`📡 CORS enabled for: ${frontendUrl}`);
}

void bootstrap();
```

Лістинг коду 2 – файл app.module.ts

```
import { Module } from '@nestjs/common';
import { ConfigModule, ConfigService } from '@nestjs/config';
import { TypeOrmModule, TypeOrmModuleOptions } from
'@nestjs/typeorm';
import { APP_GUARD } from '@nestjs/core';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { AuthModule } from './auth/auth.module';
```

```
import { UsersModule } from './users/users.module';
import { CarsModule } from './cars/cars.module';
import { MileageModule } from './mileage/mileage.module';
import { ExpensesModule } from './expenses/expenses.module';
import { MaintenanceModule } from
'./maintenance/maintenance.module';
import { AiModule } from './ai/ai.module';
import { JwtAuthGuard } from './auth/guards/jwt-auth.guard';
import typeormConfig from './config/typeorm.config';

@Module({
  imports: [
    ConfigModule.forRoot({
      isGlobal: true,
      load: [typeormConfig],
    }),
    TypeOrmModule.forRootAsync({
      inject: [ConfigService],
      useFactory: (configService: ConfigService) => {
        const config =
configService.get<TypeOrmModuleOptions>('typeorm');
        if (!config) {
          throw new Error('TypeORM configuration not found');
        }
        return config;
      },
    }),
    UsersModule,
    AuthModule,
    CarsModule,
    MileageModule,
    ExpensesModule,
    MaintenanceModule,
    AiModule,
  ],
  controllers: [AppController],
  providers: [
    AppService,
    {
      provide: APP_GUARD,
      useClass: JwtAuthGuard,
    },
  ],
})
export class AppModule {}
```

Лістинг коду 3 – файл auth.controller.ts

```
import {
  Controller,
  Post,
  Get,
```

```
  Body,
  UseGuards,
  HttpCode,
  HttpStatus,
} from '@nestjs/common';
import { AuthService } from './auth.service';
import { RegisterDto } from './dto/register.dto';
import { LoginDto } from './dto/login.dto';
import { RefreshTokenDto } from './dto/refresh-token.dto';
import { LocalAuthGuard } from './guards/local-auth.guard';
import { JwtAuthGuard } from './guards/jwt-auth.guard';
import { JwtRefreshGuard } from './guards/jwt-refresh.guard';
import { CurrentUser } from './decorators/current-user.decorator';
import { Public } from './decorators/public.decorator';
import { User } from '../users/entities/user.entity';
import type { RefreshTokenPayload } from './interfaces/jwt-
payload.interface';

@Controller('auth')
export class AuthController {
  constructor(private readonly authService: AuthService) {}

  @Public()
  @Post('register')
  async register(@Body() registerDto: RegisterDto) {
    return this.authService.register(registerDto);
  }

  @Public()
  @UseGuards(LocalAuthGuard)
  @Post('login')
  @HttpCode(HttpStatus.OK)
  async login(@CurrentUser() user: User) {
    const tokens = await
this.authService.generateTokensForUser(user);
    return {
      ...tokens,
      user: this.authService.sanitizeUser(user),
    };
  }

  @Public()
  @UseGuards(JwtRefreshGuard)
  @Post('refresh')
  @HttpCode(HttpStatus.OK)
  async refresh(@CurrentUser() payload: RefreshTokenPayload) {
    return this.authService.refresh(payload.refreshToken,
payload.userId);
  }

  @UseGuards(JwtAuthGuard)
  @Post('logout')
  @HttpCode(HttpStatus.OK)
```

```
  async logout(@Body() refreshTokenDto: RefreshTokenDto) {
    await this.authService.logout(refreshTokenDto.refreshToken);
    return { message: 'Logged out successfully' };
  }

  @UseGuards(JwtAuthGuard)
  @Get('me')
  async getProfile(@CurrentUser() user: User) {
    // User already loaded by JWT strategy, but we need full user
data
    const fullUser = await this.authService.getUserById(user.id);
    return this.authService.sanitizeUser(fullUser);
  }
}
```

Лістинг коду 4 – файл auth.service.ts

```
import {
  Injectable,
  UnauthorizedException,
  ConflictException,
} from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository, LessThan } from 'typeorm';
import { ConfigService } from '@nestjs/config';
import type { SignOptions } from 'jsonwebtoken';
import { UsersService } from '../users/users.service';
import { User } from '../users/entities/user.entity';
import { RefreshToken } from '../users/entities/refresh-
token.entity';
import { RegisterDto } from './dto/register.dto';
import { LoginDto } from './dto/login.dto';
import { AuthResponseDto } from './dto/auth-response.dto';
import { JwtPayload } from './interfaces/jwt-payload.interface';

@Injectable()
export class AuthService {
  constructor(
    private readonly usersService: UsersService,
    private readonly jwtService: JwtService,
    private readonly configService: ConfigService,
    @InjectRepository(RefreshToken)
    private readonly refreshTokenRepository:
Repository<RefreshToken>,
  ) {}

  async register(registerDto: RegisterDto): Promise<AuthResponseDto>
{
    const existingUser = await
this.usersService.findByEmail(registerDto.email);
    if (existingUser) {
```

```
      throw new ConflictException('User with this email already
exists');
    }

    const user = await this.usersService.create(registerDto);
    const tokens = await this.generateTokens(user);

    return {
      ...tokens,
      user: this.sanitizeUser(user),
    };
  }

  async validateUser(email: string, password: string): Promise<User
| null> {
    const user = await this.usersService.findByEmail(email);
    if (!user) {
      return null;
    }

    const isPasswordValid = await
this.usersService.validatePassword(
      password,
      user.password,
    );
    if (!isPasswordValid) {
      return null;
    }

    return user;
  }

  async login(loginDto: LoginDto): Promise<AuthResponseDto> {
    const user = await this.validateUser(loginDto.email,
loginDto.password);
    if (!user) {
      throw new UnauthorizedException('Invalid credentials');
    }

    const tokens = await this.generateTokens(user);

    return {
      ...tokens,
      user: this.sanitizeUser(user),
    };
  }

  async refresh(
    refreshToken: string,
    userId: string,
  ): Promise<AuthResponseDto> {
    const tokenEntity = await this.refreshTokenRepository.findOne({
      where: { token: refreshToken },
```

```
      relations: ['user'],
    });

    if (!tokenEntity || tokenEntity.expiresAt < new Date()) {
      throw new UnauthorizedException('Invalid or expired refresh
token');
    }

    if (tokenEntity.userId !== userId) {
      throw new UnauthorizedException('Token does not match user');
    }

    const user = await this.usersService.findOne(userId);
    if (!user) {
      throw new UnauthorizedException('User not found');
    }

    await this.refreshTokenRepository.remove(tokenEntity);

    const tokens = await this.generateTokens(user);

    return {
      ...tokens,
      user: this.sanitizeUser(user),
    };
  }

  async logout(refreshToken: string): Promise<void> {
    const tokenEntity = await this.refreshTokenRepository.findOne({
      where: { token: refreshToken },
    });

    if (tokenEntity) {
      await this.refreshTokenRepository.remove(tokenEntity);
    }
  }

  private async generateTokens(user: User): Promise<{
    accessToken: string;
    refreshToken: string;
  }> {
    const payload: JwtPayload = { email: user.email, sub: user.id };

    const jwtSecret = this.configService.get<string>('JWT_SECRET');
    const jwtRefreshSecret =
      this.configService.get<string>('JWT_REFRESH_SECRET');
    const accessExpiresIn =
      this.configService.get<string>('JWT_EXPIRES_IN') || '15m';
    const refreshExpiresIn =
      this.configService.get<string>('JWT_REFRESH_EXPIRES_IN') ||
'7d';

    if (!jwtSecret || !jwtRefreshSecret) {
```

```
      throw new Error('JWT secrets are not configured');
    }

    const accessTokenOptions: SignOptions = {
      secret: jwtSecret,
      // @ts-expect-error - expiresIn accepts string in jsonwebtoken
      expiresIn: accessExpiresIn,
    };

    const refreshTokenOptions: SignOptions = {
      secret: jwtRefreshSecret,
      // @ts-expect-error - expiresIn accepts string in jsonwebtoken
      expiresIn: refreshExpiresIn,
    };

    const accessToken = this.jwtService.sign(payload,
accessTokenOptions);
    const refreshToken = this.jwtService.sign(payload,
refreshTokenOptions);

    // Зберігаємо refresh token в БД
    const expiresAt = new Date();
    const days = parseInt(refreshExpiresIn.replace('d', ''), 10) ||
7;
    expiresAt.setDate(expiresAt.getDate() + days);

    const refreshTokenEntity = this.refreshTokenRepository.create({
      token: refreshToken,
      userId: user.id,
      expiresAt,
    });
    await this.refreshTokenRepository.save(refreshTokenEntity);

    await this.cleanupExpiredTokens();

    return { accessToken, refreshToken };
  }

  async generateTokensForUser(user: User): Promise<{
    accessToken: string;
    refreshToken: string;
  }> {
    return this.generateTokens(user);
  }

  async getUserById(id: string): Promise<User> {
    const user = await this.usersService.findOne(id);
    if (!user) {
      throw new UnauthorizedException('User not found');
    }
    return user;
  }
```

```
  sanitizeUser(user: User): Omit<User, 'password' | 'refreshTokens'>
{
    const { password, refreshTokens, ...sanitized } = user;
    return sanitized;
  }

  private async cleanupExpiredTokens(): Promise<void> {
    await this.refreshTokenRepository.delete({
      expiresAt: LessThan(new Date()),
    });
  }
}
```

Лістинг коду 5 – файл App.tsx

```
* ---------------------------------------------------
 * For more info, please see:
 * https://ionicframework.com/docs/theming/dark-mode
 */

/* import '@ionic/react/css/palettes/dark.always.css'; */
/* import '@ionic/react/css/palettes/dark.class.css'; */
/* import "@ionic/react/css/palettes/dark.system.css"; */ /*
Disabled for light theme */

/* Theme variables */
import "./theme/variables.css";

/* Tailwind CSS */
import "./index.css";

setupIonicReact();

const App: React.FC = () => (
  <IonApp>
    <IonReactRouter>
      <AppProviders>
        <AppRouter />
      </AppProviders>
    </IonReactRouter>
  </IonApp>
);

export default App;
```

Лістинг коду 6 – файл all/index.tsx

```
import {
  IonContent,
  IonHeader,
  IonPage,
  IonTitle,
```

```
  IonToolbar,
  IonButtons,
  IonBackButton,
  IonSegment,
  IonSegmentButton,
  IonLabel,
  IonItem,
  IonSelect,
  IonSelectOption,
} from "@ionic/react";
import { useState } from "react";
import { AllExpensesList } from "../../../features/expenses/ui/all-
expenses-list";
import { AllMaintenanceList } from
"../../../features/maintenance/ui/all-maintenance-list";

const AllExpensesPage = () => {
  const [segment, setSegment] = useState<"expenses" |
"maintenance">(
    "expenses"
  );
  const [period, setPeriod] = useState<string>("month");

  const getDateRange = (period: string) => {
    const now = new Date();
    let from: string;
    let to: string = now.toISOString().split("T")[0];

    switch (period) {
      case "week":
        from = new Date(now.getTime() - 7 * 24 * 60 * 60 * 1000)
          .toISOString()
          .split("T")[0];
        break;
      case "month":
        from = new Date(now.getFullYear(), now.getMonth(), 1)
          .toISOString()
          .split("T")[0];
        break;
      case "3months":
        from = new Date(now.getFullYear(), now.getMonth() - 3, 1)
          .toISOString()
          .split("T")[0];
        break;
      case "6months":
        from = new Date(now.getFullYear(), now.getMonth() - 6, 1)
          .toISOString()
          .split("T")[0];
        break;
      case "year":
        from = new Date(now.getFullYear(), 0,
1).toISOString().split("T")[0];
        break;
```

```
      case "all":
        from = "";
        to = "";
        break;
      default:
        from = new Date(now.getFullYear(), now.getMonth(), 1)
          .toISOString()
          .split("T")[0];
    }

    return { from, to };
  };

  const { from, to } = getDateRange(period);

  return (
    <IonPage>
      <IonHeader>
        <IonToolbar>
          <IonButtons slot="start">
            <IonBackButton defaultHref="/tabs/home" />
          </IonButtons>
          <IonTitle>Всі записи</IonTitle>
        </IonToolbar>
      </IonHeader>
      <IonContent>
        <div className="ion-padding">
          <IonSegment
            value={segment}
            onIonChange={(e) =>
              setSegment(e.detail.value as "expenses" |
"maintenance")
            }
          >
            <IonSegmentButton value="expenses">
              <IonLabel>Витрати</IonLabel>
            </IonSegmentButton>
            <IonSegmentButton value="maintenance">
              <IonLabel>Обслуговування</IonLabel>
            </IonSegmentButton>
          </IonSegment>

          <IonItem>
            <IonLabel>Період</IonLabel>
            <IonSelect
              value={period}
              onIonChange={(e) => setPeriod(e.detail.value)}
              interface="popover"
            >
              <IonSelectOption value="week">Останній
тиждень</IonSelectOption>
              <IonSelectOption value="month">Останній
місяць</IonSelectOption>
```

```
              <IonSelectOption value="3months">3
місяці</IonSelectOption>
              <IonSelectOption value="6months">6
місяців</IonSelectOption>
              <IonSelectOption value="year">Рік</IonSelectOption>
              <IonSelectOption value="all">Всі</IonSelectOption>
            </IonSelect>
          </IonItem>
        </div>

        {segment === "expenses" ? (
          <AllExpensesList from={from} to={to} />
        ) : (
          <AllMaintenanceList from={from} to={to} />
        )}
      </IonContent>
    </IonPage>
  );
};

export default AllExpensesPage;
```

Лістинг коду 7 – файл statistics/index.tsx

```
import {
  IonContent,
  IonHeader,
  IonPage,
  IonTitle,
  IonToolbar,
  IonCard,
  IonCardHeader,
  IonCardTitle,
  IonCardContent,
  IonSpinner,
  IonSelect,
  IonSelectOption,
  IonItem,
  IonLabel,
} from "@ionic/react";
import { useState } from "react";
import { useCars } from "../../../features/cars/api/cars.queries";
import { useAllExpenseStats } from
"../../../features/expenses/api/expenses.queries";
import { useExpenseCategories } from
"../../../features/expenses/api/expenses.queries";

const StatisticsPage = () => {
  const [period, setPeriod] = useState<string>("month");
  const [selectedCarId, setSelectedCarId] = useState<string>("");
  const { data: cars, isLoading: carsLoading } = useCars();
  const { data: categories } = useExpenseCategories();
```

```
  const getDateRange = (period: string) => {
    const now = new Date();
    let from: string;
    let to: string = now.toISOString().split("T")[0];

    switch (period) {
      case "month":
        from = new Date(now.getFullYear(), now.getMonth(), 1)
          .toISOString()
          .split("T")[0];
        break;
      case "3months":
        from = new Date(now.getFullYear(), now.getMonth() - 3, 1)
          .toISOString()
          .split("T")[0];
        break;
      case "6months":
        from = new Date(now.getFullYear(), now.getMonth() - 6, 1)
          .toISOString()
          .split("T")[0];
        break;
      case "year":
        from = new Date(now.getFullYear(), 0,
1).toISOString().split("T")[0];
        break;
      default:
        from = new Date(now.getFullYear(), now.getMonth(), 1)
          .toISOString()
          .split("T")[0];
    }

    return { from, to };
  };

  const { from, to } = getDateRange(period);
  const { data: stats, isLoading: statsLoading } =
useAllExpenseStats(
    from,
    to,
    selectedCarId || undefined
  );

  if (carsLoading) {
    return (
      <IonPage>
        <IonHeader>
          <IonToolbar>
            <IonTitle>Статистика</IonTitle>
          </IonToolbar>
        </IonHeader>
        <IonContent className="ion-padding">
          <div className="ion-text-center">
```

```
            <IonSpinner />
          </div>
        </IonContent>
      </IonPage>
    );
  }

  return (
    <IonPage>
      <IonHeader>
        <IonToolbar>
          <IonTitle className="text-white">Статистика</IonTitle>
        </IonToolbar>
      </IonHeader>
      <IonContent>
        <div className="ion-padding">
          <IonItem>
            <IonLabel>Автомобіль</IonLabel>
            <IonSelect
              value={selectedCarId}
              onIonChange={(e) => setSelectedCarId(e.detail.value)}
              interface="popover"
            >
              <IonSelectOption value="">Всі
автомобілі</IonSelectOption>
              {cars?.map((car) => (
                <IonSelectOption key={car.id} value={car.id}>
                  {car.brand} {car.model}
                </IonSelectOption>
              ))}
            </IonSelect>
          </IonItem>

          <IonItem>
            <IonLabel>Період</IonLabel>
            <IonSelect
              value={period}
              onIonChange={(e) => setPeriod(e.detail.value)}
              interface="popover"
            >
              <IonSelectOption value="month">Останній
місяць</IonSelectOption>
              <IonSelectOption value="3months">3
місяці</IonSelectOption>
              <IonSelectOption value="6months">6
місяців</IonSelectOption>
              <IonSelectOption value="year">Рік</IonSelectOption>
            </IonSelect>
          </IonItem>

          {statsLoading ? (
            <div className="ion-text-center ion-padding">
              <IonSpinner />
```

```
            </div>
          ) : stats ? (
            <>
              <IonCard style={{ margin: "16px", overflow: "hidden"
}}>
                <IonCardHeader>
                  <IonCardTitle style={{ wordBreak: "break-word" }}>
                    Загальна сума витрат
                  </IonCardTitle>
                </IonCardHeader>
                <IonCardContent>
                  <h1
                    style={{
                      fontSize: "2.5rem",
                      margin: 0,
                      wordBreak: "break-word",
                    }}
                  >
                    {stats.totalAmount.toFixed(2)} ₴
                  </h1>
                </IonCardContent>
              </IonCard>

              <IonCard style={{ margin: "16px", overflow: "hidden"
}}>
                <IonCardHeader>
                  <IonCardTitle style={{ wordBreak: "break-word" }}>
                    Розподіл по категоріях
                  </IonCardTitle>
                </IonCardHeader>
                <IonCardContent>
                  {stats.byCategory.length > 0 ? (
                    stats.byCategory.map((cat) => (
                      <div key={cat.category} style={{ marginBottom:
"1rem" }}>
                        <div
                          style={{
                            display: "flex",
                            justifyContent: "space-between",
                            alignItems: "flex-start",
                            marginBottom: "0.5rem",
                            gap: "8px",
                          }}
                        >
                          <span style={{ wordBreak: "break-word",
flex: 1 }}>
                            {cat.category}
                          </span>
                          <span style={{ flexShrink: 0, whiteSpace:
"nowrap" }}>
                            {cat.amount.toFixed(2)} ₴ (
                            {cat.percentage.toFixed(1)}%)
                          </span>
```

```
                        </div>
                        <div
                          style={{
                            width: "100%",
                            height: "8px",
                            backgroundColor: "#E3F2FD",
                            borderRadius: "4px",
                            overflow: "hidden",
                          }}
                        >
                          <div
                            style={{
                              width: `${cat.percentage}%`,
                              height: "100%",
                              backgroundColor: "#2196F3",
                              transition: "width 0.3s ease",
                            }}
                          />
                        </div>
                      </div>
                    ))
                  ) : (
                    <p>Немає даних</p>
                  )}
                </IonCardContent>
              </IonCard>

              <IonCard style={{ margin: "16px", overflow: "hidden"
}}>
                <IonCardHeader>
                  <IonCardTitle style={{ wordBreak: "break-word" }}>
                    Витрати по місяцях
                  </IonCardTitle>
                </IonCardHeader>
                <IonCardContent>
                  {stats.byMonth.length > 0 ? (
                    stats.byMonth.map((month) => (
                      <div
                        key={month.month}
                        style={{
                          display: "flex",
                          justifyContent: "space-between",
                          alignItems: "center",
                          padding: "0.5rem 0",
                          borderBottom: "1px solid #E3F2FD",
                          gap: "8px",
                        }}
                      >
                        <span style={{ wordBreak: "break-word",
flex: 1 }}>
                          {month.month}
                        </span>
                        <span
```

```
                          className="font-semibold text-medium-blue"
                          style={{ flexShrink: 0, whiteSpace:
"nowrap" }}
                        >
                          {month.amount.toFixed(2)} ₴
                        </span>
                      </div>
                    ))
                  ) : (
                    <p>Немає даних</p>
                  )}
                </IonCardContent>
              </IonCard>
            </>
          ) : (
            <p>Немає даних про витрати</p>
          )}
        </div>
      </IonContent>
    </IonPage>
  );
};

export default StatisticsPage;
```

Додаток Б Публікація у науковій конференції

УДК 004.41

Тихович Б.П. – ст. гр. СП-61

Тернопільський національний технічний університет імені Івана Пулюя

БАГАТОПЛАТФОРМЕННИЙ ЗАСТОСУНОК ДЛЯ МОНІТОРИНГУ СТАНУ ТА ВИТРА ОБСЛУГОВУВАННЯ АВТОМОБІЛЯ

Науковий керівник: д. ф.-м. н., професор Петрик М. Р.

Tykhovych B.

Ternopil Ivan Puluj National Technical University

MULTIPLATFORM APPLICATION FOR MONITORING THE CONDITION AND MAINTENANCE COSTS OF A VEHICLE

Supervisor: PhD in Technical Sciences, Professor Petryk M.

Ключові слова: моніторинг автомобіля, багатоплатформенний застосунок, PWA, Ionic, діагностика, телеметрія, API, мобільний додаток, Firebase.

Keywords: vehicle monitoring, multiplatform application, PWA, Ionic, diagnostics, telemetry, API, mobile application, Firebase.

Науково-дослідна робота присвячена розробці багатоплатформенного застосунку для моніторингу стану та витрат обслуговування автомобіля. Метою роботи є створення програмного рішення, яке забезпечує централізоване зберігання, аналіз та управління інформацією про експлуатацію автомобіля протягом усього періоду його використання. Застосунок орієнтований на підвищення зручності ведення історії автомобіля, контролю витрат та підтримки своєчасного технічного обслуговування.

На початковому етапі виконано аналіз предметної області та визначено потреби користувачів у систематизації даних про автомобіль. Особливу увагу приділено проблемам відсутності єдиного цифрового простору для зберігання історії пробігу, витрат, обслуговування та фінансових операцій, пов’язаних з автомобілем. На основі цього сформульовано функціональні та нефункціональні вимоги до програмної системи, а також визначено основні варіанти використання та акторів.

Розроблювана система надає користувачу можливість додавати один або декілька автомобілів і вести для кожного з них повну хронологічну історію експлуатації. Дані про пробіг, витрати та обслуговування можуть вводитися вручну або за допомогою завантаження

фотографій чеків і документів, які обробляються з використанням алгоритмів штучного інтелекту з подальшим перетворенням у структуровані записи. Передбачена можливість перевірки та коригування автоматично згенерованих даних користувачем.

На основі накопиченої інформації система забезпечує фінансовий облік автомобіля, включаючи фіксацію вартості придбання, витрат за категоріями та даних про продаж. Окрему увагу приділено можливості спільного доступу до автомобіля, що дозволяє кільком користувачам працювати з однією історією, а також передавати історію автомобіля іншому власнику.

Для реалізації клієнтської частини застосунку використано фреймворк Ionic React, що забезпечує багатоплатформенність та можливість використання системи як у веб-браузері, так і на мобільних пристроях. Архітектура програмної системи спроєктована з урахуванням можливості подальшого розширення функціоналу, зокрема інтеграції додаткових сервісів та модулів у майбутньому такі, як Firebase.

Під час розробки враховано нефункціональні вимоги, зокрема вимоги до зручності використання, продуктивності та безпеки. Реалізовано механізми автентифікації користувачів, розмежування доступу до даних автомобілів та захист передавання інформації між клієнтською і серверною частинами системи.

Завершальним етапом роботи стало тестування програмної системи, яке включало перевірку коректності реалізації основного функціоналу, стабільності роботи застосунку та відповідності реалізованих можливостей сформульованим вимогам. Результати тестування підтвердили працездатність системи та доцільність її використання для ведення цифрової історії автомобіля.

Отримані результати свідчать про те, що розроблений багатоплатформенний застосунок може бути ефективним інструментом для обліку та аналізу експлуатації автомобіля. Подальший розвиток системи можливий у напрямку розширення аналітичних можливостей, рекомендаційних сервісів та інтеграції з маркетплейсом автотоварів.

Література

1. Official Documentation of the Ionic Framework [Електронний ресурс] – Режим доступу до ресурсу: https://ionicframework.com/docs/v3/.
2. Official Firebase Developer Documentation [Електронний ресурс] – Режим доступу до ресурсу: https://firebase.google.com/docs.