МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра програмної інженерії

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

на тему: **«РОЗРОБКА СИСТЕМИ ДЛЯ ВІДСТЕЖЕННЯ ТЕХНІЧНОГО СТАНУ АВТОМОБІЛІВ»**

Виконав: студент 2 курсу, групи 8.1213-з-дн
спеціальності 121 інженерія програмного забезпечення
(шифр і назва спеціальності)
освітньої програми інженерія програмного забезпечення
(назва освітньої програми)
А.І. Туша
(ініціали та прізвище)

Керівник доцент кафедри програмної інженерії, доцент, к.т.н. Мухін В.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Рецензент завідувач кафедри фундаментальної та прикладної математики, професор, д.т.н. Гребенюк С.М.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя – 2025

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра програмної інженерії

Рівень вищої освіти магістр

Спеціальність 121 інженерія програмного забезпечення
(шифр і назва)

Освітня програма інженерія програмного забезпечення

ЗАТВЕРДЖУЮ
Завідувач кафедри програмної інженерії, к.ф.-м.н., доцент

_____ Лісняк А.О.
(підпис)

“ ____ ” _____ 2024 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Туші Андрію Івановичу
(прізвище, ім’я та по-батькові)

1. Тема роботи Розробка системи для відстеження технічного стану автомобілів

керівник роботи Мухін Віталій Вікторович, к.т.н., доцент
(прізвище, ім’я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 01 » березня 2024 року № 384-с

2. Строк подання студентом роботи 17.02.2025 р.

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.
3. Розробка додатку та тестування.

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)
презентація за темою доповіді

6. Консультанти розділів роботи

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

7. Дата видачі завдання 07.03.2024 р.

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	29.03.2024	
2.	Збір вихідних даних.	03.05.2024	
3.	Обробка методичних та теоретичних	28.06.2024	
	джерел.		
4.	Розробка першого та другого розділу.	18.10.2024	
5.	Розробка третього розділу.	31.01.2025	
6.	Оформлення та нормоконтроль	10.02.2025	
	кваліфікаційної роботи магістра.		
7.	Захист кваліфікаційної роботи.	05.03.2025	

Студент _____ А.І. Туша
(підпис) (ініціали та прізвище)

Керівник роботи _____ В.В. Мухін
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ А.В. Столярова
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота магістра «Розробка системи для відстеження технічного стану автомобілів»: 75 с., 60 рис., 16 джерел.

СИСТЕМА, ФРЕЙМВОРК, ADAPTER, BOOTSTRAP, CAN, EXPRESS JS, NODE, POSTGRE SQL, TOKEN.

Об’єкт дослідження – процес розробки програмного продукту для менеджменту автосервісного підприємства, що інтегрує функції організації робочих процесів, обліку клієнтів і транспортних засобів, обробки платежів та забезпечення систем нагадувань.

Мета роботи: створення програмного забезпечення для менеджменту автосервісного підприємства, яке спрямовано на підвищення рівня обслуговування клієнтів, оптимізацію внутрішніх процесів підприємства, забезпечення можливостей для подальшого масштабування та інтеграції.

Предмет дослідження – підходи та сучасні технології для розробки додатків, що мають клієнтську та серверну частину.

Методи дослідження: комплексний огляд технічної літератури та документації на обрані технології, бізнес-аналіз процесів, моделювання архітектури та оцінка ефективності продукту.

Ключовою особливістю майбутнього продукту є адаптивність до потреб конкретного бізнесу та можливістю інтеграції з апаратними системами для діагностики транспортних засобів. Він автоматизує ключові процеси: запис клієнтів, управління базою даних транспортних засобів, обробку платежів і планування технічного обслуговування, забезпечуючи зручність і прозорість для всіх учасників.

SUMMARY

Master’s qualifying paper “Development of a System for Tracking the Technical Condition of Cars”: 75 pages, 60 figures, 16 references.

ADAPTER, BOOTSTRAP, CAN, FRAMEWORK, NODE EXPRESS JS, POSTGRE SQL, SYSTEM, TOKEN.

The object of study is the process of developing software for managing a car service enterprise, integrating workflow organization, customer and vehicle accounting, payment processing, and reminder systems.

The aim of the study is to create adaptable software tailored to specific business needs, ensuring efficient workflow organization, improved customer service, optimized internal processes, and reduced costs, with scalability and integration capabilities.

The subject of research is the technologies and architectural solutions for developing car service management systems, including workflow automation, structured data processing, payment transaction handling, and system scalability for future integration with vehicle diagnostics hardware.

The research methods include a comprehensive review of technical literature, analysis of business processes, architectural system modeling, software development methodologies, functional validation, and performance assessment.

Developed software should be suitable for both small businesses and large enterprises to optimize operations, increase customer loyalty, reduce costs, and automate routine tasks while minimizing human errors.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Вступ 7
1 Постановка задачі 9
1.1 Огляд проблеми вибору системи для менеджменту автосервіса 9
1.2 Аналіз існуючих рішень 11
1.3 Висновки за результатами аналізу інформаційно-управлінських систем для автосервісних підприємств 16
1.4 Аналіз функціональних вимог до програмного забезпечення автосервісу 17
1.5 Визначення цільової аудиторії 18
2 Проєктування та архітектура програмного продукту 20
2.1 Планування структури програмного продукту 20
2.2 Детальний опис обраного підходу 22
2.3 Діаграма прецедентів 24
2.4 Рекомендовані технології 26
3 Розробка, тестування та розгортання продукту 28
3.1 Розробка та написання коду клієнтської частини 28
3.2 Проєктування структури бази даних 39
3.3 Розробка серверної частини застосунку 43
3.3.1 Структура та принципи діагностики автомобільних електронних систем за стандартом OBD-II 57
3.3.2 Налаштування взаємодії з OBD-II адаптером 66
3.4 Тестування та відлагодження 68
3.5 Розгортання програмного продукту. 70
Висновки 72
Перелік посилань 74

ВСТУП

Інформаційно-управлінська система для автосервісів призначена для забезпечення ефективної організації робочих процесів, таких як запис клієнтів, планування технічного обслуговування, облік транспортних засобів, обробка платежів та реалізація системи нагадувань. Система спрямована на оптимізацію бізнес-процесів, зниження витрат та підвищення рівня обслуговування клієнтів. Важливою особливістю є можливість масштабування та інтеграції з додатковими модулями, такими як апаратні системи діагностики автомобілів.

Система автоматизує запис клієнтів, формування графіків обслуговування та облік транспортних засобів. Завдяки цьому підприємство може оптимально розподіляти ресурси та зменшувати час очікування клієнтів. Крім того, забезпечується автоматичне нагадування клієнтам про планове технічне обслуговування або виконання рутинних завдань, таких як заміна масла чи перевірка гальм.

Моніторинг та аналітика є важливими складовими системи. Вона дозволяє збирати й аналізувати дані про обслуговування транспортних засобів, виявляти тенденції та формувати звіти. Це сприяє підвищенню ефективності роботи підприємства, допомагає приймати обґрунтовані рішення та покращує загальну якість обслуговування.

Система підтримує інтеграцію з платіжними платформами, що дозволяє здійснювати оплату послуг безпосередньо через інтерфейс. Вона автоматизує створення квитанцій і забезпечує зручність як для клієнтів, так і для бухгалтерії автосервісу. Це скорочує час, необхідний для обробки платежів, та зменшує ймовірність помилок.

Технічні вимоги до системи передбачають її доступність через браузер на будь-якому пристрої з підтримкою Інтернету. У розробці використовуються сучасні технології, такі як Node.js для серверної частини,

React і Bootstrap для клієнтської частини, а також реляційна база даних, наприклад, PostgreSQL або MySQL, для зберігання інформації про клієнтів, транспортні засоби та записи обслуговування. Особлива увага приділяється забезпеченню безпеки даних, включаючи шифрування, багаторівневу автентифікацію та захист конфіденційної інформації.

Очікується, що створення цієї інформаційно-управлінської системи надасть підприємствам зручний інструмент для організації роботи, підвищить ефективність процесів, скоротить витрати та покращить якість обслуговування клієнтів. Система адаптована для використання як у невеликих бізнесах, так і на великих підприємствах, що забезпечує її універсальність та перспективність.

Документування розробки охоплює всі етапи створення системи. Описуються вибрані технології, архітектура, проведені тести та отримані результати. Також передбачається створення керівництва користувача, що дозволить швидко адаптувати систему до потреб підприємства та навчити персонал ефективно нею користуватися.

Перш ніж інтегрувати систему у робочий процес підприємства, проводиться всебічне тестування всіх компонентів. Воно включає модульне тестування окремих функцій, інтеграційне тестування для перевірки взаємодії між компонентами та системне тестування для оцінки роботи системи як цілісного продукту. Тестування забезпечує стабільну й надійну роботу системи.

Після завершення тестування передбачається поетапне впровадження системи у роботу підприємства. Цей процес включає підготовку інфраструктури, інтеграцію з існуючими бізнес-процесами та навчання персоналу. Завдяки такому підходу система зможе максимально швидко почати функціонувати та приносити користь бізнесу.

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Огляд проблеми вибору системи для менеджменту автосервіса

У сучасній автомобільній індустрії системи управління автосервісами відіграють важливу роль у забезпеченні ефективності роботи підприємств, підвищенні рівня обслуговування клієнтів та оптимізації робочих процесів. Вибір правильного програмного забезпечення для автосервісу є ключовим завданням, яке впливає на загальну продуктивність підприємства. Цей огляд розгляне основні аспекти цієї проблеми, включаючи вимоги до систем менеджменту автосервісів, типи доступних рішень, а також їх переваги та недоліки.

Вимоги до систем менеджменту автосервісу.

Перед вибором програмного забезпечення для автосервісу необхідно визначити основні вимоги, які система має виконувати. Ці вимоги включають:

- масштабованість та адаптивність: система повинна легко налаштовуватися під потреби конкретного підприємства та мати можливість додавання нових функцій або модулів;
- зручність використання: інтерфейс повинен бути інтуїтивно зрозумілим для персоналу різного рівня підготовки, з мінімальною потребою у додатковому навчанні;
- автоматизація: програмне забезпечення має забезпечувати автоматизацію ключових процесів, таких як запис клієнтів, ведення бази даних транспортних засобів, обробка платежів та створення нагадувань;
- безпека: дані клієнтів і підприємства повинні бути захищені за допомогою сучасних технологій шифрування та багаторівневої автентифікації.

Перейдемо до розгляду типів систем менеджменту автосервісів.

Універсальні CRM-системи. Ці системи пропонують широкий набір функцій для управління бізнесом, але часто містять зайві функції, які не використовуються автосервісами.

Спеціалізовані рішення для автосервісів. Такі системи розроблені з урахуванням потреб автосервісного бізнесу, що дозволяє легко інтегрувати функції запису клієнтів, планування технічного обслуговування та обліку транспортних засобів.

Інтегровані рішення з апаратними системами. Ці системи можуть об’єднувати функції управління з можливостями діагностики автомобілів у реальному часі через підключення до діагностичного обладнання.

Переваги та недоліки різних систем:

– універсальні CRM-системи: мають широкий функціонал, підходять для багатьох бізнесів, але ж завдяки цьому є складними у налаштуванні під конкретну специфіку та містять багато зайвих функцій;
– спеціалізовані рішення для автосервісів є оптимізованими для роботи з автосервісами, включають потрібні функції; у той же час їх інтеграція з іншими системами може бути недостатньою;
– інтегровані рішення з апаратними системами: зручні для моніторингу технічного стану транспортних засобів у реальному часті, є можливість автоматизації процесів діагностики; суттєвим недоліком є залежніть від надійності інфраструктури та стороннього програмного забезпечення.

Отже, вибір програмного забезпечення для автосервісу залежить від потреб конкретного підприємства. Для невеликих автосервісів ідеально підходять спеціалізовані рішення з базовими функціями. Для великих підприємств або сервісів із високими вимогами до діагностики доцільно використовувати інтегровані системи, які поєднують управління робочими процесами з апаратною діагностикою. Кожен тип системи має свої переваги, і вибір повинен базуватися на аналізі вимог бізнесу та його специфіки.

1.2 Аналіз існуючих рішень

RemOnline [1] – це комплексне програмне рішення для станцій технічного обслуговування (СТО), яке забезпечує ведення обліку виконаних робіт, використаних запчастин та матеріалів, а також допомагає контролювати процеси ремонту автомобілів клієнтів. Програмний продукт надає широкий функціонал, що включає нормування часу виконання робіт, автоматичне формування звітів та розрахунок кінцевої вартості замовлення. Загальний вигляд інтерфейсу наведено на рисунку 1.1.

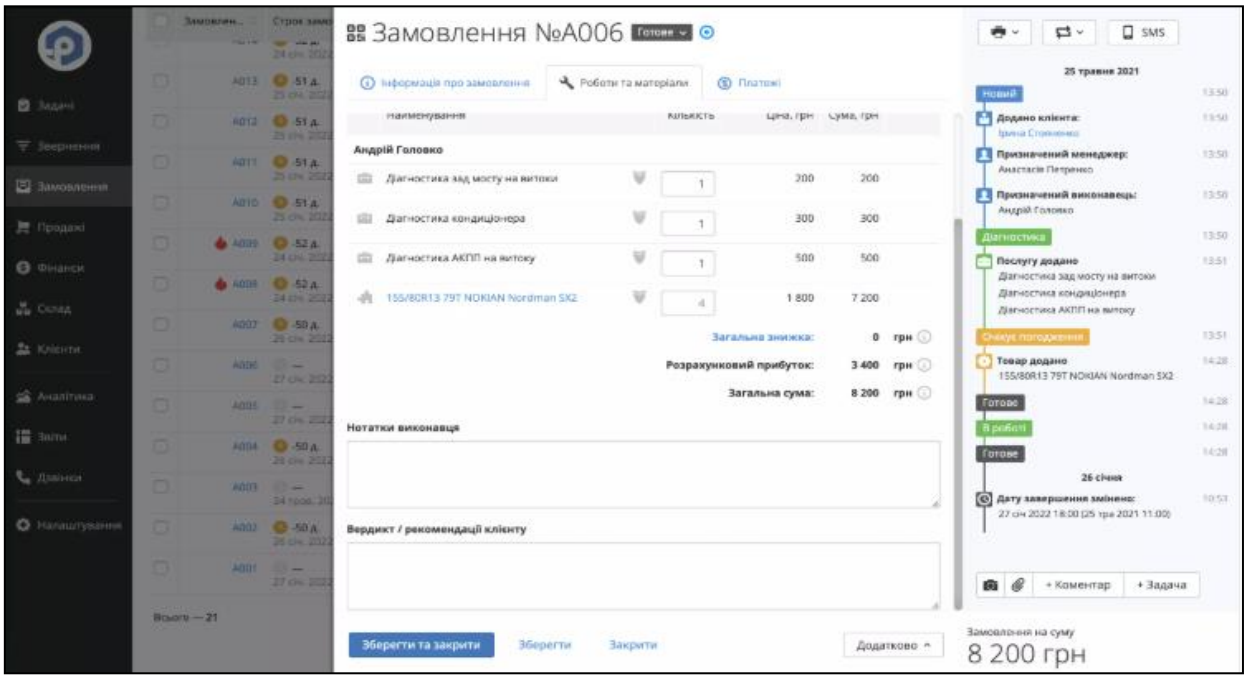


Рисунок 1.1 – Приклад інтерфейсу программного продукту RemOnline

Крім того, програма має функціонал для проведення розрахунку продуктивності персоналу (див. рис. 1.2), формування системи мотивації, аналіз витрат робочого часу.

Розглянемо переваги та недоліки цього програмного рішення.

Переваги RemOnline. Програма забезпечує комплексний облік та контроль усіх операцій на СТО. Вона автоматизує рутинні процеси, такі як облік запчастин, виставлення рахунків та підготовка звітів. Це допомагає уникнути помилок і покращити точність планування.

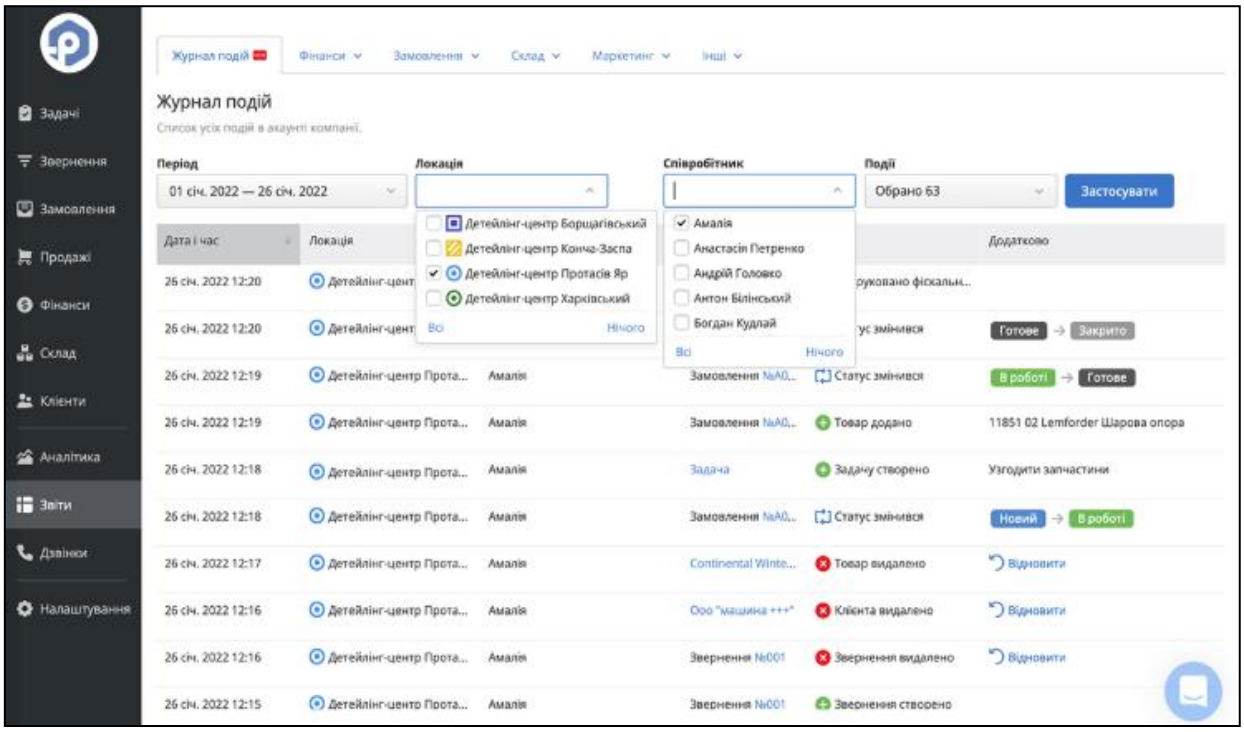


Рисунок 1.2 – Вікно «Журнал подій» программного продукту RemOnline

RemOnline пропонує потужні аналітичні інструменти, які дозволяють відстежувати продуктивність співробітників, витрати та доходи. Це сприяє ухваленню обґрунтованих управлінських рішень і підвищенню ефективності роботи.

Система також містить функціонал для оцінки роботи персоналу та формування системи мотивації, що сприяє підвищенню продуктивності. Завдяки хмарній архітектурі програма доступна з будь-якого пристрою, що дозволяє керівникам контролювати робочі процеси віддалено.

Інтерфейс RemOnline інтуїтивно зрозумілий, що забезпечує швидке навчання персоналу та мінімізує ризики помилок під час використання.

Недоліки RemOnline. Незважаючи на переваги, програма має деякі обмеження. Відсутність клієнтського порталу ускладнює для клієнтів можливість самостійно відстежувати статус ремонту, що може створювати додаткове навантаження на менеджерів.

Також система не інтегрується з діагностичним обладнанням, що ускладнює збереження історії технічного стану автомобілів. Це може бути критично для автосервісів, які використовують сучасні методи діагностики.

Ще одним мінусом є залежність від інтернет-з’єднання, що може ускладнити роботу у разі перебоїв з мережею. Крім того, RemOnline є платним продуктом, що може бути фінансовим навантаженням для малих підприємств.

FixIQ Pro [2] – це сучасна платформа для управління транспортним бізнесом, яка надає широкий спектр інструментів (див. рис. 1.3) для оптимізації робочих процесів в автосервісах, мийках, шиномонтажах та інших суміжних сферах.

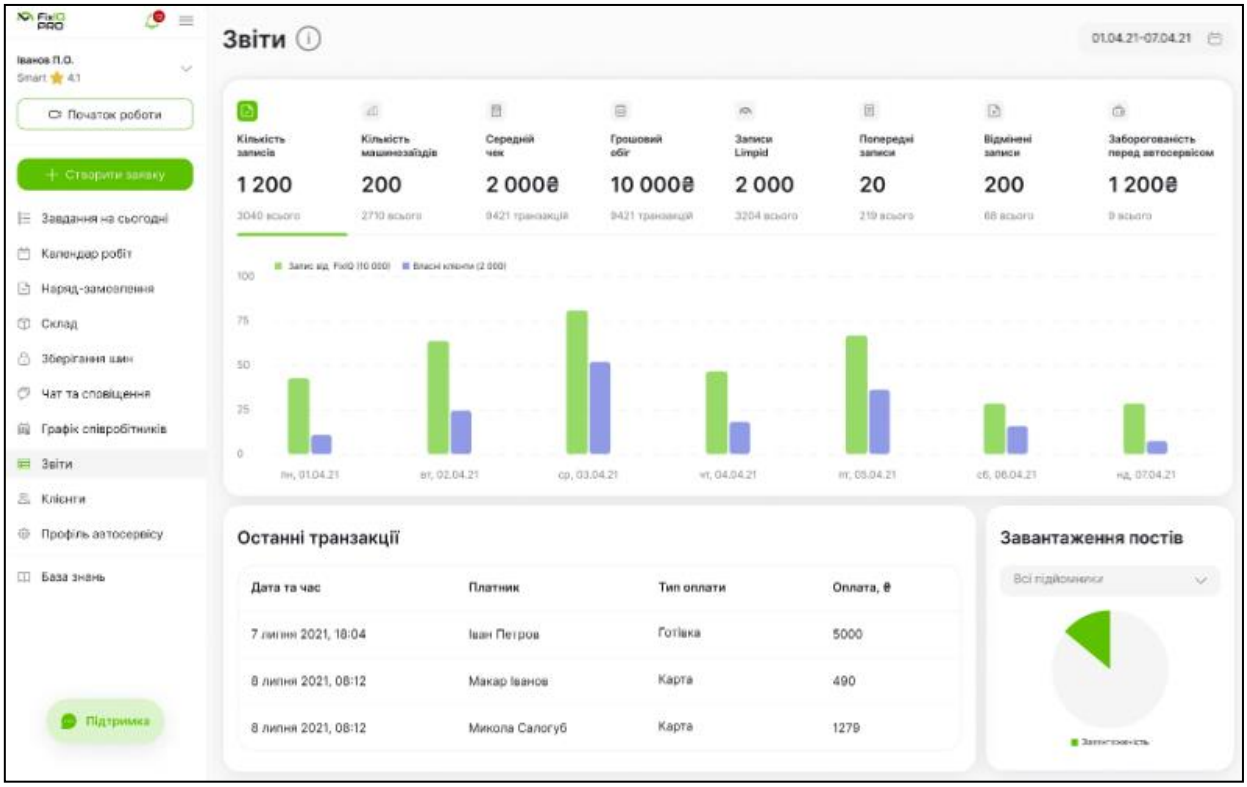


Рисунок 1.3 – Приклад інтерфейсу программного продукту FixIQ Pro

Переваги FixIQ Pro:

– автоматизація процесів: система дозволяє ефективно керувати наряд-замовленнями, зберігати історію обслуговування клієнтів та автомобілів, а також оптимізувати графік роботи співробітників;

– інтеграція з Opendatabot: автоматичне підтягування даних про автомобіль за допомогою Opendatabot спрощує процес введення інформації та підвищує точність даних;

– довідники нормо-годин: наявність бази нормо-годин для популярних марок автомобілів дозволяє швидше планувати роботи та забезпечує стандартизацію процесів;
– цифровий документообіг: автоматичне формування документів у картці замовлення забезпечує зручний доступ до необхідної інформації як для співробітників, так і для клієнтів;
– комунікація з клієнтами: вбудований чат дозволяє зберігати всю історію спілкування з клієнтами, що спрощує узгодження робіт та підвищує рівень обслуговування.

Недоліки FixIQ Pro:

– обмежений безкоштовний функціонал: для доступу до повного спектру функцій знадобиться платна підписка, вартість якої залежить від кількості обслуговуваних автомобілів;
– відсутня інтеграція із діагностичним обладнанням.

Програма для автосервісу та СТО від ***ExpressSoft*** [3] – це спеціалізоване програмне забезпечення, розроблене для автоматизації та оптимізації бізнес-процесів станцій технічного обслуговування. Базовий функціонал наведено на рисунку 1.4.

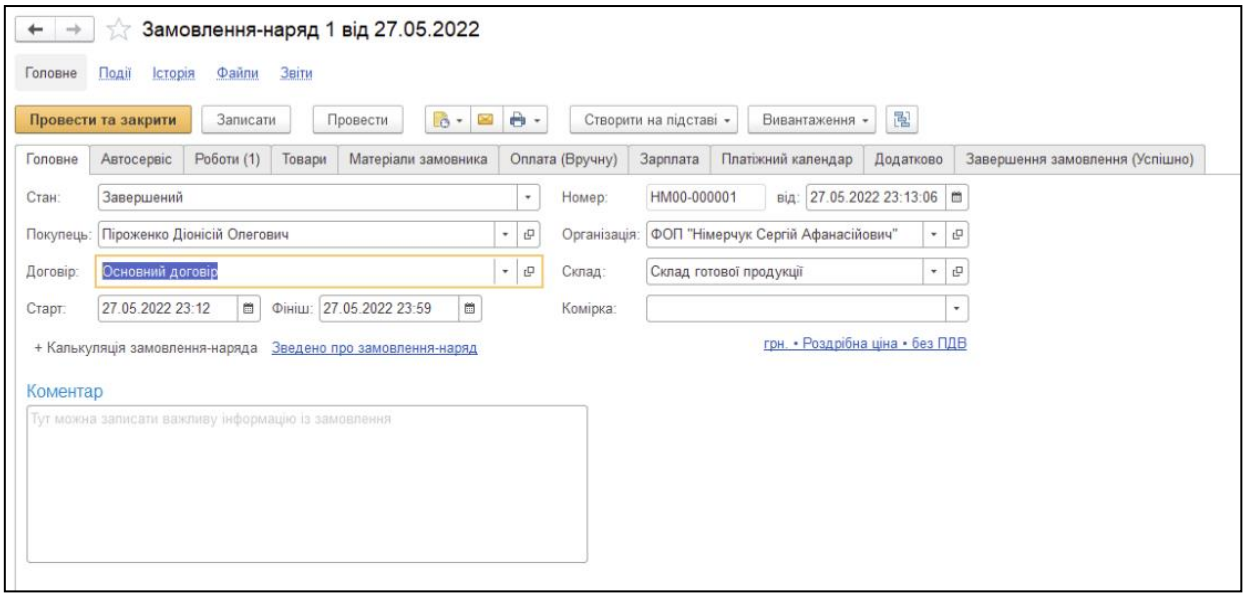


Рисунок 1.4 – Приклад інтерфейсу для оформлення заказ-наряду у програмі для менеджменту автосервісного підприємства від ExpressSoft

Програма базується на платформі «BAS Малий бізнес» і адаптована під специфічні потреби автосервісів.

Переваги:

– комплексний облік: система забезпечує детальний облік автомобілів клієнтів, включаючи марку, модель, рік випуску, об’єм двигуна та VIN-код – це дозволяє зберігати повну історію обслуговування кожного транспортного засобу;
– управління замовленнями: програма дозволяє формувати наряди на сервісні роботи, в яких зазначаються виконані роботи, використані матеріали, виконавці та витрачений час – це спрощує процес контролю та планування робіт;
– автоматизація документообігу: автоматичне створення друкованих форм документів, таких як рахунки, акти виконаних робіт та інші, зменшує ймовірність помилок і прискорює обробку документації;
– фінансовий облік: система підтримує оформлення як готівкових, так і безготівкових розрахунків, а також автоматизований розрахунок заробітної плати працівників на основі фіксованих тарифів або погодинної оплати;
– управління запасами: програма допомагає планувати та обліковувати закупівлі необхідних запчастин у постачальників, а також вести облік розрахунків з ними;
– CRM-функціонал: інтегровані інструменти для управління відносинами з клієнтами, включаючи SMS-розсилки та чат-боти, сприяють покращенню комунікації та підвищенню лояльності клієнтів.

Недоліки:

– складність впровадження: для повноцінного використання всіх можливостей програми може знадобитися час на налаштування та навчання персоналу, особливо якщо раніше не використовувалися подібні системи;

– вартість: хоча програма пропонує широкий функціонал, її вартість може бути високою для невеликих автосервісів, однак, компанія надає безкоштовну демо-версію для ознайомлення з можливостями продукту;
– залежність від платформи: оскільки програма базується на «BAS Малий бізнес», можливості її розширення та інтеграції з іншими системами можуть бути обмежені рамками цієї платформи.

1.3 Висновки за результатами аналізу інформаційно-управлінських систем для автосервісних підприємств

Автоматизація бізнес-процесів у сфері автосервісу є однією з ключових умов ефективного функціонування підприємства. Розглянуті програмні продукти, такі як RemOnline, FixIQ, та ExpressSoft, демонструють широкий спектр можливостей для оптимізації роботи автосервісів, що включають управління клієнтською базою, облік транспортних засобів, формування звітності та аналіз продуктивності персоналу.

Загалом, усі розглянуті рішення мають суттєві переваги, серед яких можна виділити автоматизацію рутинних процесів, зменшення адміністративного навантаження, підтримку бухгалтерського обліку, управління запасами та надання клієнтам інструментів для взаємодії з сервісом, таких як портали чи мобільні додатки. Це забезпечує підвищення продуктивності роботи підприємства, зменшення впливу людського фактору та покращення взаємодії з клієнтами.

Разом із тим, усі аналізовані системи мають спільний недолік: відсутність інтеграції із сучасними діагностичними інструментами, такими як CAN-сканери та інше обладнання для автоматичного збору технічних даних автомобіля. Це є важливою прогалиною, яка обмежує можливості систем у сфері технічного обслуговування та діагностики транспортних засобів. У сучасних умовах стрімкого розвитку технологій така інтеграція могла б

значно підвищити якість послуг, зокрема шляхом автоматичного збирання даних про технічний стан автомобіля, виключення помилок, пов’язаних із людським фактором, та надання клієнтам детальної інформації про необхідні роботи.

Ще одним аспектом, який варто зазначити, є те, що жодна з аналізованих систем не пропонує функціонал, спрямований на формування рекомендацій щодо майбутнього обслуговування автомобіля на основі зібраних даних. Такий інструмент міг би стати корисним для підвищення лояльності клієнтів та забезпечення їхньої довіри до автосервісу.

Таким чином, для створення сучасного рішення, яке відповідало б усім потребам автосервісного бізнесу, необхідно поєднати найкращі практики існуючих продуктів з новими функціями, що включають інтеграцію з діагностичним обладнанням, автоматизацію аналізу отриманих даних, формування рекомендацій для технічного обслуговування та вдосконалення системи взаємодії з клієнтами. Таке рішення стане важливим кроком до впровадження інноваційних технологій у сферу обслуговування автомобілів та дозволить автосервісним підприємствам бути конкурентоспроможними на сучасному ринку.

1.4 Аналіз функціональних вимог до програмного забезпечення автосервісу

Інформаційно-управлінська система для автосервісного підприємства має поєднувати кращі практики сучасних рішень із додатковими унікальними функціями:

– інтеграція з діагностичним обладнанням: система підтримує підключення CAN-сканерів для збору технічних даних автомобіля в реальному часі – це дозволяє виявляти можливі несправності, формувати рекомендації щодо технічного обслуговування та зберігати результати діагностики для подальшого аналізу;

– автоматизація бізнес-процесів: система автоматизує запис клієнтів, ведення бази даних транспортних засобів, формування графіків обслуговування та облік платежів – це знижує навантаження на адміністративний персонал і мінімізує ймовірність людських помилок;
– інтеграція з системами нагадувань і лояльності: автоматичні повідомлення нагадують клієнтам про необхідність технічного обслуговування, а система лояльності дозволяє накопичувати бонуси за виконані послуги, стимулюючи повторні звернення та підвищуючи лояльність клієнтів.

Розроблена з використанням сучасних технологій, таких як Node.js для серверної частини, React і Bootstrap для клієнтської частини та PostgreSQL для зберігання даних, система поєднує ефективність і зручність використання. Її багаторівнева архітектура забезпечує масштабованість, можливість розширення функціоналу та адаптацію під потреби конкретного бізнесу.

Завдяки унікальному функціоналу система є важливим інструментом для автосервісів, які прагнуть підвищити рівень обслуговування клієнтів, оптимізувати свої робочі процеси та інтегрувати інноваційні технології для покращення якості послуг.

1.5 Визначення цільової аудиторії

Основні аудиторії програмного продукту для менеджменту автосервісного підприємства:

– власники автосервісів та СТО: включає власників невеликих майстерень, середніх і великих автосервісів – вони прагнуть автоматизувати бізнес-процеси, знизити витрати, підвищити ефективність роботи підприємства та впровадити сучасні інструменти управління;

- адміністратори та менеджери автосервісів: користувачі, які відповідають за запис клієнтів, планування графіків, облік платежів та управління щоденними операціями (важливі аспекти для них – простота використання системи, швидкий доступ до даних і автоматизація рутинних завдань);
- технічні спеціалісти: до цієї категорії входять механіки та діагности, які працюють із транспортними засобами – система надає їм можливість інтеграції з діагностичним обладнанням (CAN-сканери) для збору та аналізу технічних даних у реальному часі, а також зберігання історії сервісу для покращення точності діагностики;
- клієнти автосервісів: власники автомобілів, які прагнуть отримати якісний сервіс із прозорим обліком історії обслуговування, персоналізованими знижками, нагадуваннями про ТО та зручною системою взаємодії із сервісом;
- мережі автосервісів: підприємства з декількома локаціями, які потребують централізованої системи управління – такий продукт дозволяє ефективно керувати операціями між філіями, координувати роботу персоналу та підтримувати єдиний стандарт обслуговування.

2 ПРОЄКТУВАННЯ ТА АРХІТЕКТУРА ПРОГРАМНОГО ПРОДУКТУ

2.1 Планування структури програмного продукту

Розробка програмного продукту для менеджменту автосервісного підприємства є інноваційним рішенням, яке має на меті вдосконалити управління бізнес-процесами, оптимізувати взаємодію з клієнтами та забезпечити високу якість обслуговування. Цей додаток дозволить створити сучасну платформу, яка адаптується під специфічні потреби автосервісів, незалежно від їхнього масштабу. Завдяки зручному інтерфейсу користувачі зможуть легко взаємодіяти з системою, записуватись на послуги, отримувати нагадування про планові ТО, а також переглядати історію обслуговування транспортного засобу.

Основна структура програмного продукту складається з трьох ключових компонентів: інтерфейсу користувача, серверної частини та бази даних (див. рис. 2.1).

Інтерфейс користувача поділяється на панелі адміністратора і клієнта, що дозволяє кожному користувачеві мати доступ до потрібного функціоналу. Адміністратор може керувати записами, моніторити виконання робіт та аналізувати дані, а клієнт – оформлювати заявки на діагностику чи ремонт, отримувати інформацію про послуги та здійснювати оплату. Серверна частина забезпечує обробку даних і взаємодію між інтерфейсом користувача та базою даних, що дозволяє гарантувати швидкодію та надійність системи.

Окрему увагу приділено безпеці даних та інтеграції з іншими системами.

Продукт підтримуватиме сучасні методи аутентифікації користувачів, резервне копіювання даних, а також інтеграцію з платіжними платформами та системами сповіщень. Це забезпечить не лише надійний захист персональної інформації, але й автоматизацію процесів, таких як надсилання повідомлень про заплановані обслуговування чи стан виконання ремонту.



Рисунок 2.1 – Розробка програмного продукту для менеджменту автосервісного підприємства

Також передбачається можливості інтеграції з апаратним діагностичним обладнанням, що дозволить не лише значно прискорити процеси діагностики, але й забезпечити автоматичне збереження отриманих даних у системі.

2.2 Детальний опис обраного підходу

Розробляємий проєкт має чітку, добре організовану структуру, яка дозволяє легко розширювати функціонал, підтримувати код та забезпечує зрозумілий розподіл обов'язків між різними компонентами системи. Основні елементи проєкту розподілені по папках, таких як models, controllers, routes, middleware, що сприяє модульності і спрощує інтеграцію нових функцій.

У папці models розташовані моделі бази даних, які є ключовим елементом у збереженні даних про клієнтів, транспортні засоби, історію обслуговування, платежі та діагностичні записи. Завдяки використанню ORM-бібліотеки, такої як Sequelize [4] , забезпечується зручність взаємодії з базою даних та підтримка реляційних зв'язків між таблицями.

Папка controllers містить логіку обробки запитів. Наприклад, контролери users.js, clients.js та vehicles.js відповідають за управління даними про користувачів, клієнтів та транспортні засоби відповідно. Контролери, пов'язані з сервісними операціями (diagnostics.js, maintenances.js, repairs.js), дозволяють обробляти дані щодо технічного обслуговування, зберігати історію виконаних робіт та пов'язувати ці дані з конкретними клієнтами та автомобілями.

Роутінг представлено у папці routes. Ці файли відповідають за обробку HTTP-запитів, спрямовуючи їх до відповідних контролерів. Наприклад, файл services.js об'єднує маршрути для роботи з діагностикою, ремонтами та технічним обслуговуванням, забезпечуючи єдиний інтерфейс для роботи з цими функціями.

Файли в папці middleware забезпечують додаткову функціональність для обробки запитів. Наприклад, auth.js і checkRole.js відповідають за перевірку прав доступу та автентифікацію, що забезпечує захист даних та контрольовану взаємодію користувачів із системою. errorHandling.js реалізує централізовану обробку помилок, що підвищує стабільність роботи додатку.

Інтеграція CAN-систем додає новий рівень функціональності до проєкту. Ці системи, працюючи через стандартний OBD-II інтерфейс, дозволяють зчитувати параметри автомобіля в реальному часі, такі як оберти двигуна, температура рідин, тиск у шинах, стан гальм та інші технічні характеристики. Для цього розробляється окремий модуль, який взаємодіє з CAN-сканером, отримує дані та передає їх у серверну частину для обробки.

Зібрані дані аналізуються сервером та зберігаються в базі даних. На основі цих даних система може автоматично створювати рекомендації щодо необхідності технічного обслуговування, виявляти можливі несправності та попереджати користувачів про ризики. Інтеграція з CAN-системами значно підвищує рівень сервісу, роблячи його більш точним та інформативним.

Система нагадувань є важливим інструментом для підвищення лояльності клієнтів і забезпечення безпеки автомобілів. Вона автоматично формує та надсилає сповіщення клієнтам на основі даних із бази, наприклад, про необхідність заміни масла, проведення діагностики чи планового технічного обслуговування. Нагадування можуть надсилатися через SMS, електронну пошту або вбудовані повідомлення у вебінтерфейсі.

Ця система працює в тісному зв’язку з контролерами сервісних операцій, такими як maintenances.js та repairs.js, а також використовує дані з бази для налаштування точних умов, наприклад, залежно від пробігу, часу чи стану транспортного засобу. Завдяки цим можливостям автосервіс може підтримувати постійний контакт із клієнтами, покращуючи якість обслуговування та сприяючи повторним зверненням.

Така структура забезпечує комплексний підхід до розробки програмного продукту для автосервісів. Компоненти взаємодіють у єдиній

системі, де кожна частина виконує свою роль: клієнтська частина забезпечує зручний інтерфейс, серверна обробляє запити та забезпечує бізнес-логіку, база даних зберігає всю необхідну інформацію, а CAN-системи й нагадування додають інноваційності та підвищують рівень сервісу. Цей підхід забезпечує надійність, масштабованість і гнучкість продукту, який може адаптуватися до потреб будь-якого автосервісного підприємства.

2.3 Діаграма прецедентів

Діаграма прецедентів (Use Case Diagram) – це графічне представлення взаємодії користувачів (акторів) із системою, яке демонструє, які функціональні можливості (прецеденти) система надає. Це важливий інструмент у процесі розробки програмного забезпечення, оскільки дозволяє формалізувати вимоги до системи та показати взаємодію користувачів із ключовими функціями.

Розглянемо деякі особливості діаграми прецедентів (див. рис. 2.2) для нашого проєкту. Почнемо з Акторів.

У цьому проєкті акторами є:

- менеджер/адміністратор: він має повний доступ до системи, володіє правами на створення та редагування будь-яких записів, створює та активує/деактивує налаштування для розсилки, програм лояльності та інше;
- працівник автосервісу: додає відповідну інформацію до системи, створює записи із клієнтами, автомобілями, платежами;
- авторизований користувач: отримує інформацію про історію обслуговування, нагадування про ТО, має доступ із правами читача до записів про своє авто та платежі;
- неавторизований користувач: доступ обмежено формами входу, нагадування та реєстрації нового клієнта.

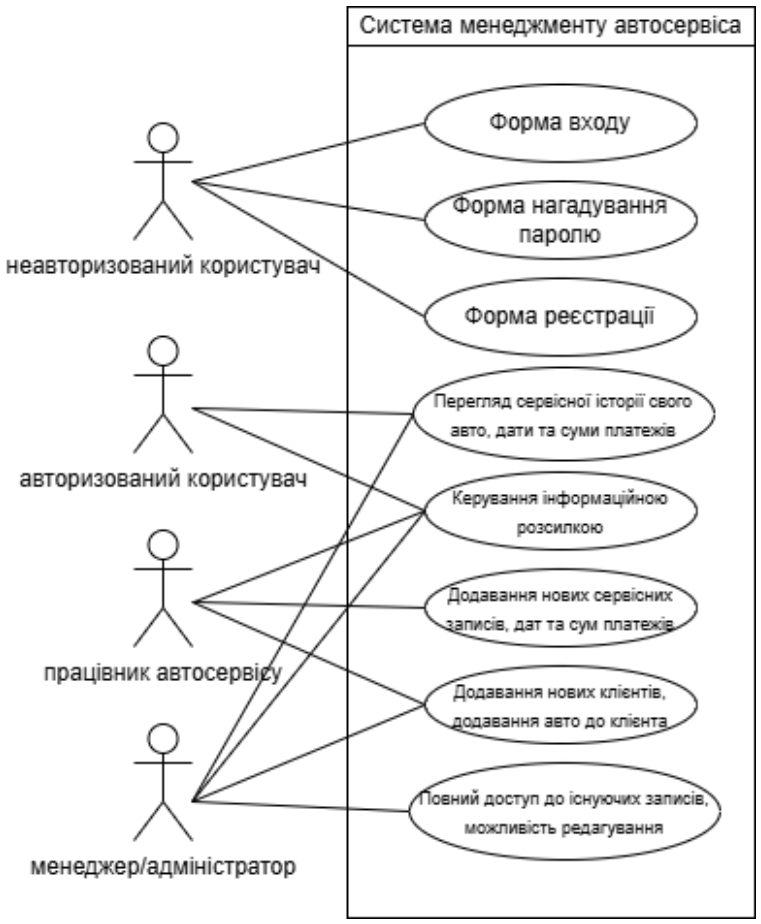


Рисунок 2.2 – Діаграма прецедентів для застосунку

Наступним етапом йде розгляд прецедентів. Діаграма включає такі ключові прецеденти:

– вхід до системи: відповідає за здійснення входу до інформаційної системи;
– реєстрація у системі: за допомогою форми відбувається додавання клієнта до системи;
– нагадування паролю: дозволяє користувачу нагадати втрачений пароль;
– перегляд сервісної інформації: надає доступ авторизованому користувачу до записів у системі стосовно належного йому автомобіля, перегляд сервісної історії, перегляд платежів, перегляд особистих даних;
– керування розсилкою: адміністратор або клієнт має можливість додати/скасувати розсилку інформаційних повідомлень;

– додавання сервісних записів: працівник автосервісу має право додавання нових записів, внесення фінансової інформації (сума оплати, статус платежа);
– додавання нових клієнтів, зв’язок клієнт-авто: працівник автосервісу має доступ до створення нових записів про клієнта, має права додавання автомобіля до створеного клієнтського запису;
– адміністративний доступ: менеджер/адміністратор має повний доступ до всіх записів, має права на редагування та видалення.

Діаграма прецедентів для проєкту є важливим інструментом для візуалізації функціональності системи та взаємодії користувачів, що дозволяє забезпечити чітке розуміння вимог і полегшити етапи розробки та впровадження.

2.4 Рекомендовані технології

Frontend:
– фреймворк – React.js;
– UI-компоненти – Bootstrap;
– інструменти для форм – React Hook Form;
– роутинг – React Router.

Backend:
– мова програмування – Node.js (Express.js);
– архітектура – REST API;
– бібліотеки – JSON Web Token (JWT) для аутентифікації, Nodemailer для Email.

База даних:
– реляційна БД – PostgreSQL.

Системи інтеграції:
– пакет NPM noble для CAN діагностики.

Система безпеки:

– аутентифікація – JWT;
– захист даних – HTTPS, шифрування паролів (bcrypt);
– резервне копіювання – Supabase.

3 РОЗРОБКА, ТЕСТУВАННЯ ТА РОЗГОРТАННЯ ПРОДУКТУ

3.1 Розробка та написання коду клієнтської частини

Для розробки продукту плануємо використовувати React [5] – відкриту JavaScript бібліотеку для створення інтерфейсів користувача.

Для створення нового проєкту React використаємо команду з рисунку 3.1, де diploma – назва нашого проєкту. Це створить структуру, яка наведена на рисунку 3.2.

```
npx create-react-app diploma
```

Рисунок 3.1

```
├── node_modules/          # Залежності
├── public/                # Статичні файли
├── src/                   # Джерельний код
│   ├── App.css            # Стили для App компонента
│   ├── App.js             # Головний компонент
│   ├── index.js           # Точка входу додатка
│   └── ...                # Інші файли
├── package.json           # Інформація про проєкт та залежності
└── README.md              # Інструкції
```

Рисунок 3.2

Для спрощення створення та стилізації інтерфейсів користувача також встановимо та підключимо (у файлі index.js) до нашого проєкту фреймворк Bootstrap за допомогою команд з рисунку 3.3.

```
npm install react-bootstrap bootstrap
import 'bootstrap/dist/css/bootstrap.min.css'
```

Рисунок 3.3

Також встановимо бібліотеку для маршрутизації React Router [6]. Виконаємо відповідно команду з рисунку 3.4.

```
npm install react-router-dom
```

Рисунок 3.4

Напишемо компонент для ідентифікації входу користувача. Інтерфейс компонента наведено на рисунку 3.5. Для перевірки корректності вводу будемо використовувати регулярні вирази. Код функції для перевірки паролю наведено на рисунку 3.6.

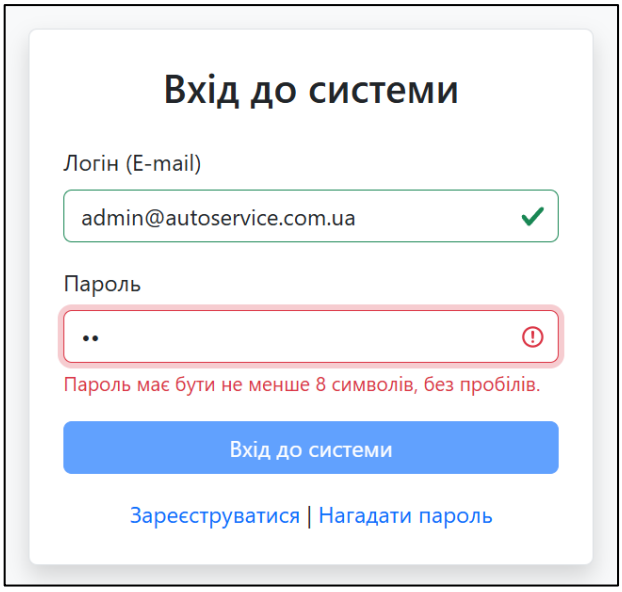


Рисунок 3.5 – Інтерфейс для введення логіну та пароля користувача

```
// Перевірка логіна (електронна пошта)
  const validateLogin = (value) => {
    const emailPattern = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    setIsLoginValid(emailPattern.test(value));
    setLogin(value);
  };
  // Перевірка пароля
  const validatePassword = (value) => {
    const passwordPattern = /^[A-Za-z0-9][A-Za-z0-9!@#$%^&*()_+=\-
`~<>?,./:;»'[\]{}|]{7,}$/;
    const isValid = passwordPattern.test(value) && !value.includes(««);
    setIsPasswordValid(isValid);
    setPassword(value);
  };
```

Рисунок 3.6 – Приклад використання регулярних виразів для валідації даних

Наступним етапом розробки є створення інтерфейсу для реєстрації нового користувача (див. рис. 3.7) та форми для відновлення доступу до облікового запису (див. рис. 3.8).

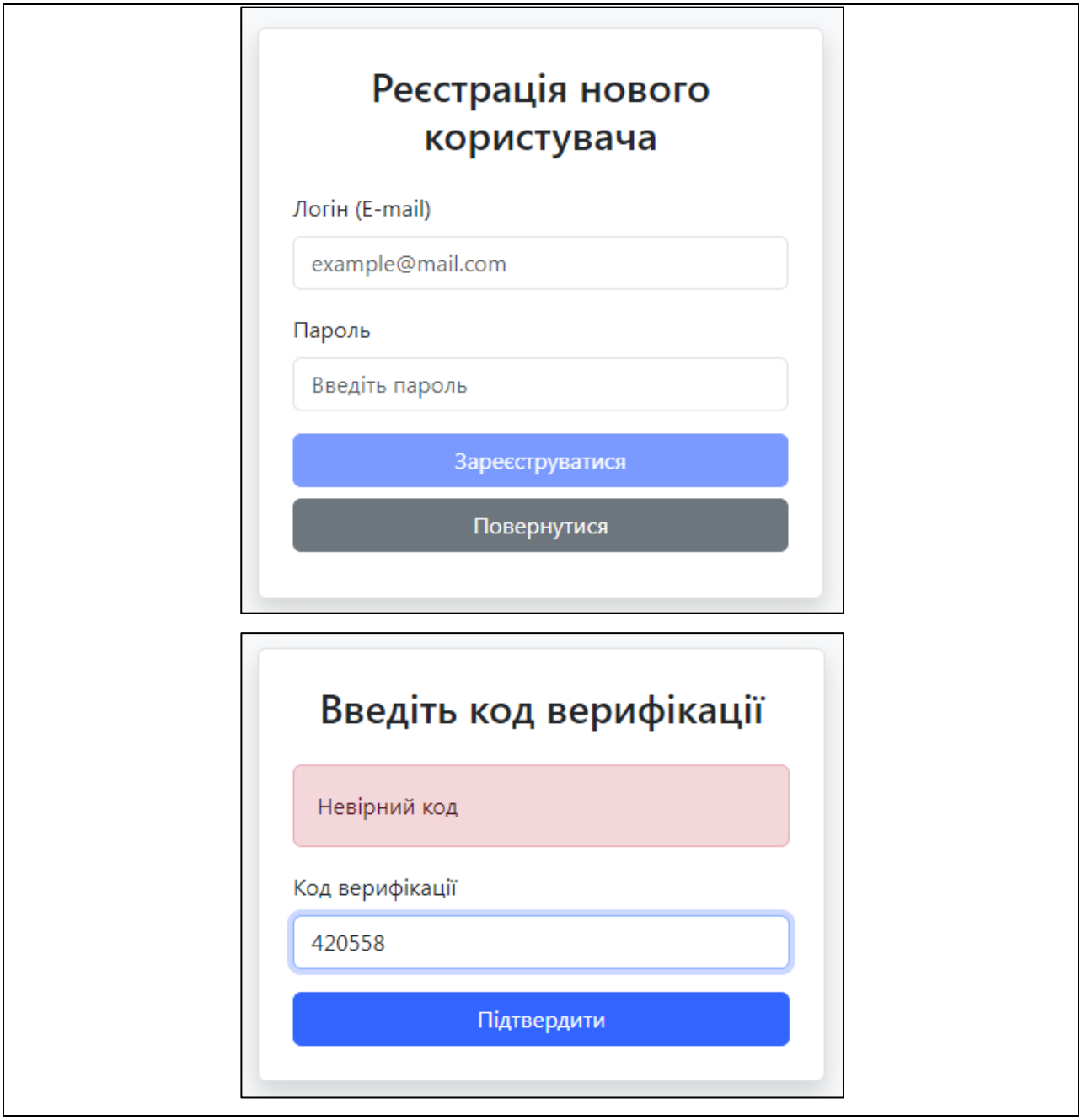


Рисунок 3.7 – Інтерфейс для реєстрації нового користувача

Дані, що вводяться у поля форм, перевіряються на відповідність встановленим вимогам за допомогою регулярних виразів, використовується двофакторна автентифікація. Використання цього підходу є ефективним і раціональним, оскільки дозволяє суттєво зменшити обсяг коду, необхідного для реалізації перевірки введених даних.

Аналогічним чином побудований і наступний компонент, що відповідає за надання втраченого чи забутого паролю. Для ідентифікації користувача використовується його електронна адреса.

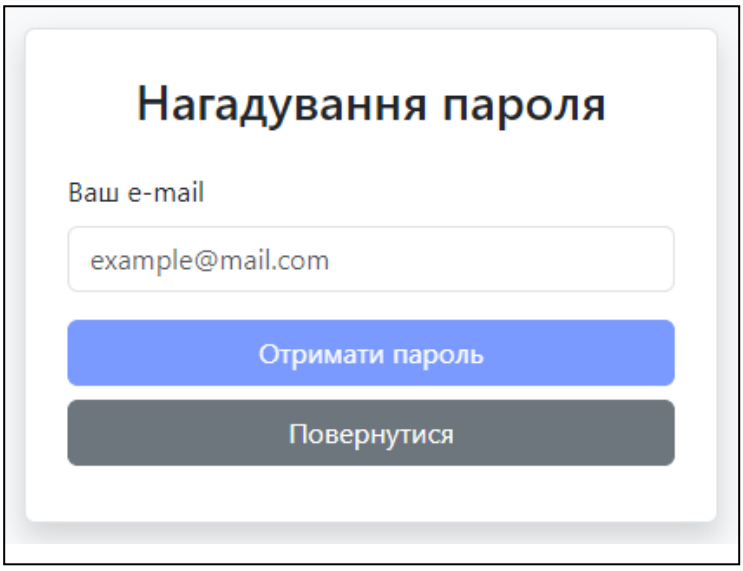


Рисунок 3.8 – Інтерфейс для нагадування пароля користувача

Після вдалого доступу до системи нам потрібно відображати отримані від сервера дані. Насамперед, передбачимо передачу отриманих даних між компонентами нашого программного продукту.

Ми будемо використовувати React Context – це вбудований механізм управління станом у React, який дозволяє передавати дані через дерево компонентів без необхідності вручну пропускати пропси через кожен рівень. Він використовується для створення глобальних змінних, які можуть бути доступними в будь-якому місці програми, наприклад, для тем, мови інтерфейсу або автентифікації. `React Context` складається з двох основних частин: `Context Provider` (компонент, що постачає дані) і `Context Consumer` (компонент, що отримує дані).

Значення, надані через `Provider`, доступні всім дочірнім компонентам, які використовують Consumer або хук `useContext`. Це дозволяє спростити управління станом у програмах, уникаючи надмірного підйому стану або використання бібліотек типу Redux для простих задач.

Створюємо файл контексту для клієнтів. Структура показана на рисунку 3.9.

Імпортуємо створений файл контексту та огорнемо наш основний компонент у <ClientsProvider> як показано на рисунку 3.10.

```
import React, { createContext, useState } from 'react';

export const ClientsContext = createContext();

export const ClientsProvider = ({ children }) => {
    const [clients, setClients] = useState([]);

    return (
        <ClientsContext.Provider value={{ clients, setClients }}>
            {children}
        </ClientsContext.Provider>
    );
};
```

Рисунок 3.9 – Файл контексту для передачі стану

```
import { ClientsProvider } from './context/ClientsContext.js'
root.render(
        <Router>
            <ClientsProvider>
                <App />
            </ClientsProvider>
        </Router>
);
……. Файл LoginForm.jsx…….
const { setClients } = useContext(ClientsContext);
if (clientsResponse.ok) {
setClients(clientsData); // Зберігаємо клієнтів у контекст
      navigate('/clients'); // Переходимо на сторінку клієнтів
}
…….Файл Clients.js……
const { clients } = useContext(ClientsContext);  // Використовуємо у
Clients
```

Рисунок 3.10 – Використання контексту для передачі стану

Таким чином ми передаємо отримані від сервера дані у таблицю Клієнти. Результат зображено на рисунку 3.11. Код обробки події наступний (див. рис. 3.12). Як видно, ми за допомогою стану передаємо клієнтський id, що потім під час обробки запиту сервером повертає нам дані пов’язаного із цим клієнтом автомобіля.



Рисунок 3.11 – Візуальне відображення інформації у таблиці «Клієнти»

```
<button
    className="btn btn-primary btn-sm m-1"
    onClick={() => {
        setSelectedClientId(client.id);
        setActiveTab('vehicles');
    }}
>
    Транспорт
</button>
```

Рисунок 3.12 – Отримання клієнтського id для подальшого запиту

Загальний інтерфейс сторінки із пов’язаними автомобілями наведено на рисунку 3.13. Додана також можливість завантаження фотографії визначеного автомобіля та можливість переглянути платежі, які були пов’язані із обраним автомобілем. Також передбачена можливість вийти до таблиці «Транспортні засоби».

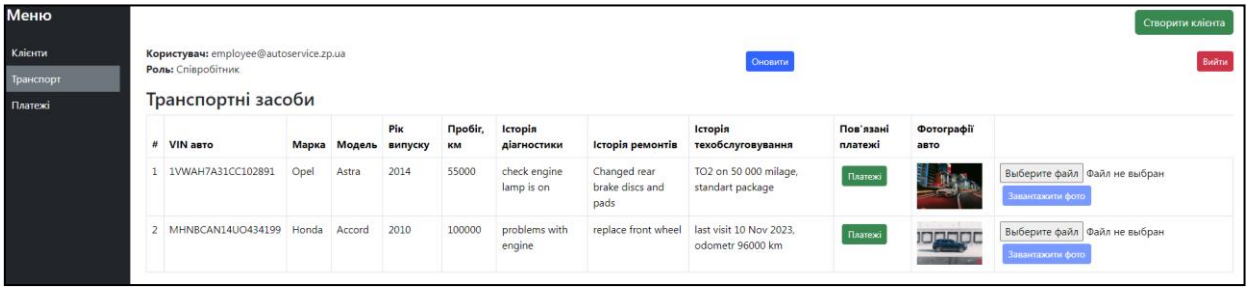


Рисунок 3.13 – Візуальне відображення інформації у таблиці

Додавання нового користувача має наступну логіку. Після натискання кнопки «Створити клієнта», яка доступна лише користувачам із рівнем доступу «Адмін» чи «Співробітник», система перенаправляє нас на вкладку

/new-client, за якою доступний компонент <NewClientForm />, інтерфейс якого наведено на рисунку 3.14. Валідація значень, що вводяться у поля, проводиться за допомогою регулярних виразів, як і у попередніх формах. Після натискання кнопки «Зареєструватися» відбувається створення об’єкту із даними, отриманими з полів форми, та виконується запит до серверу, під час якого передається в заголовці `Authorization` токен авторизації та створений об’єкт у тілі запиту, завчасно перетворений на строку за допомогою функції `JSON.stringify(clientData)`. Нарешті, сервер обробляє запит, створює клієнта в базі даних і повертає відповідь, яку можна далі використовувати для підтвердження успіху або обробки помилок.

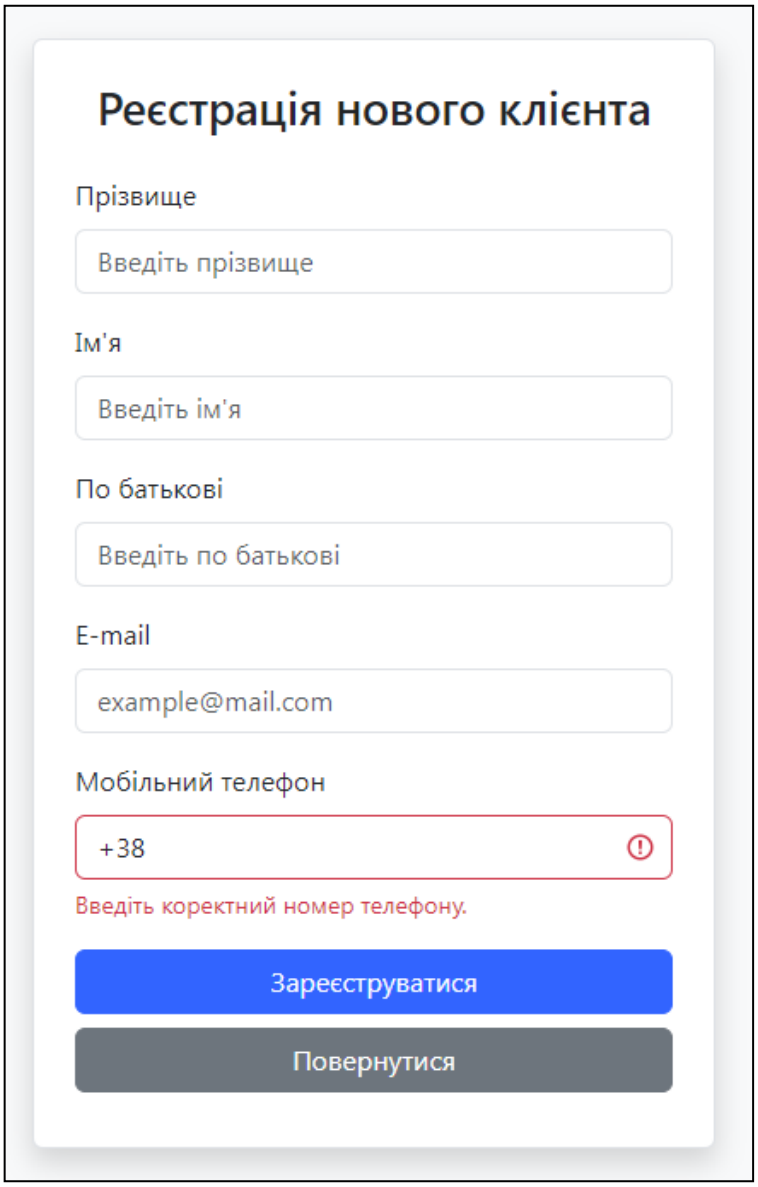


Рисунок 3.14 – Інтерфейс для реєстрації нового клієнта

Аналогічно виглядає інтерфейс для додавання автомобіля та платежів. Марки та моделі автомобілів завчасно сформовані у відповідному файлі та доступні у клієнтській частині у вигляді випадаючих списків. Це дозволяє підтримувати чистоту введених даних та уникати синтаксичних помилок при заповленні. Для платежів передбачено статус «Оплачено» та «Очікується», а також розмір дисконту у відсотках.

Загалом, в нашій системі дані передаються між компонентами через пропси, стани (`useState`), контекст (`useContext`) і навігацію (`navigate`). Головний компонент Dashboard керує активною вкладкою (`activeTab`) та зберігає вибраних об'єктів, таких як клієнт (`selectedClientId`) чи автомобіль (`selectedVehicleId`). Він передає ці дані в дочірні компоненти, наприклад, у `ClientsTable`, `VehiclesTable` або `PaymentsTable`, щоб вони завантажували й відображали відповідну інформацію. `ClientsTable` відповідає за показ списку клієнтів, використовуючи контекст `ClientsContext` для завантаження даних. За потреби цей компонент може змінити вкладку та передати `clientId` у `VehiclesTable`, де відображаються транспортні засоби, пов'язані з клієнтом. У `VehiclesTable` є можливість вибору автомобіля, що змінює вкладку на «Платежі» і передає vehicleId до `PaymentsTable`.

`PaymentsTable` завантажує список платежів для конкретного автомобіля, використовуючи запит до сервера. Усі ці таблиці мають механізм оновлення через виклик запитів у `useEffect`, залежно від змінених пропсів, таких як `clientId` чи `vehicleId`. Форми створення нових записів, такі як `NewClientForm`, `NewAutoForm` чи `NewPaymentForm`, відповідають за відправлення даних на сервер для створення нового клієнта, автомобіля чи платежу. Після успішного створення форми використовують navigate для повернення в Dashboard, передаючи стан, щоб встановити відповідну активну вкладку та передати потрібні ідентифікатори (`clientId` чи `vehicleId`).

Крім того, контекст ClientsContext використовується для зберігання даних клієнтів, що дозволяє зменшити кількість запитів до сервера. Це забезпечує ефективну передачу даних між компонентами без дублювання

запитів. Уся система працює так, щоб кожен компонент оновлювався автоматично при зміні даних або станів, переданих через пропси, контекст чи навігацію. Це створює динамічну й інтерактивну архітектуру, яка дозволяє ефективно обробляти дані та забезпечувати зручність для користувача.

Наступним етапом вдосконалення функціоналу є розширення можливостей редагування та видалення записів у таблицях клієнтів, транспортних засобів та платежів. Ми створили функціонал, який дозволяє користувачеві з роллю «Адмін» виконувати редагування даних через модальне вікно, а також видаляти записи. Для користувачів із ролями «Співробітник» або «Клієнт» ці дії недоступні, що забезпечує обмеження доступу на основі ролей.

Одним із ключових елементів функціоналу є універсальний компонент EditModal (див. рис. 3.15). Він відображає форму для редагування записів, використовуючи поля, які налаштовуються залежно від контексту (таблиця клієнтів, транспортних засобів чи платежів). Поля форми можуть бути як текстовими, так і у вигляді випадаючих списків.

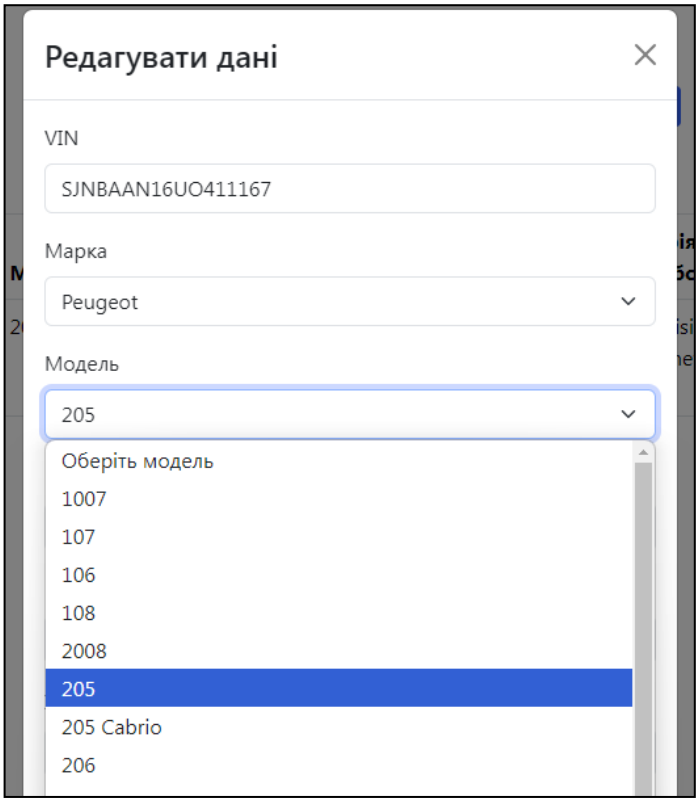


Рисунок 3.15 – Модальне вікно для редагування введених даних

При зміні даних у формі стан editData автоматично оновлюється через функцію onChange, що передається до EditModal. Це дозволяє зберігати всі зміни до моменту натискання кнопки «Зберегти». У таблиці транспортних засобів випадаючі списки для вибору марки та моделі реалізовані динамічно. Наприклад, при зміні марки (mark) список моделей автоматично оновлюється відповідно до доступних моделей цієї марки. Це забезпечує коректний вибір моделі, яка відповідає обраній марці. Код функції наведено на рисунку 3.16.

```
const handleEditDataChange = (updatedData) => {

if (updatedData.mark !== editData.mark) {
updatedData.model = ""; // Скидаємо модель, якщо змінено марку
}
setEditData(updatedData);
};
```

Рисунок 3.16 – Функція динамічного оновлення списків

Функція видалення записів реалізована через кнопку «Видалити» в кожному рядку таблиці. Натискання кнопки викликає функцію, яка надсилає запит на сервер для видалення відповідного запису. Наприклад, для таблиці платежів (див. рис. 3.17).

Редагування та видалення доступні лише для користувачів із роллю «Адмін». Для скорочення кількості коду перевірка доступу реалізована на сервері за допомогою middleware CheckRole.

```
const handleDeletePayment = async (paymentId) => {
 if (window.confirm("Ви впевнені, що хочете видалити цей платіж?")) {
 try {
const response = await fetch(
`http://localhost:5000/api/payments/deletePayment/${paymentId}`, {
 method: "DELETE",
 headers: {
Authorization: `Bearer ${localStorage.getItem("token")}` },
} );
if (response.ok) {
alert("Платіж успішно видалено");
fetchPayments(); // Оновлення таблиці після видалення
}
```

Рисунок 3.17 – Функція для видалення записів із таблиці платежів

Переваги реалізованого функціоналу:

– зручність редагування: універсальний компонент EditModal дозволяє редагувати дані для різних таблиць із мінімальними змінами в конфігурації;
– динамічні списки: забезпечують зручність і точність вибору значень;
– оновлення даних: дані таблиць завжди актуальні завдяки автоматичному завантаженню після змін;
– безпека: функції обмежені на основі ролей на сервері за допомогою middleware.

Перейдемо до реалізації функціоналу для відправлення нагадувань клієнтам. Система нагадувань побудована таким чином, щоб забезпечити автоматизоване інформування клієнтів електронною поштою у визначений час. Її функціонал поділений на дві частини: клієнтську, де адміністратор або співробітник може створити нагадування, та серверну, де відбувається збереження даних, їх обробка і автоматичне надсилання листів.

На клієнтській стороні адміністратор або співробітник має доступ до таблиці клієнтів, у якій кожен рядок містить чекбокс для активації нагадування. Коли чекбокс увімкнено, відображаються два додаткові поля: для вибору дати нагадування та тексту повідомлення. Користувач обирає дату через інтерактивний календар у полі типу date та вводить текст у текстове поле. Як тільки текстове поле втрачає фокус, система автоматично зберігає нагадування, надсилаючи запит на сервер. У разі вимкнення чекбокса нагадування видаляється з бази даних через відповідний API-запит. Уся взаємодія з сервером відбувається через функції fetch, які надсилають PUT чи DELETE запити залежно від дії.

Наступним кроком реалізуємо функціонал для отримання кодів помилок електронних систем автомобіля за допомогою адаптера ELM-327. Додамо кнопку для відправлення запитів у таблицю транспортних засобів (див. рис. 3.18).

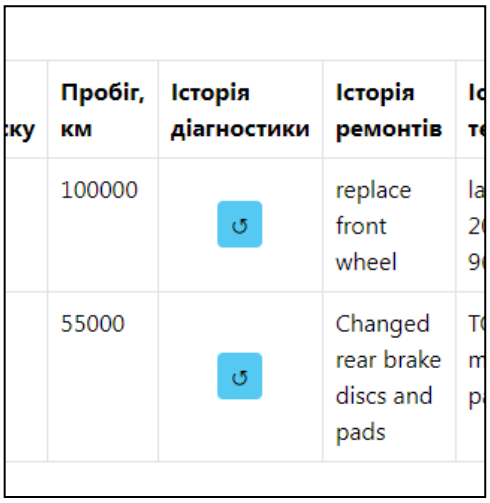


Рисунок 3.18 – Кнопка для опитування ЕБУ автомобіля

Після натискання кнопки виконується fetch-запит до маршруту http://localhost:5000/api/vehicles/readErrors/:vehicleId. У разі успішного виконання сервер повертає список кодів помилок, який додається у базу даних та відображається у відповідному компоненті.

3.2 Проєктування структури бази даних

Для цього ми використовуватимемо PostgreSQL [7]. За допомогою утиліти pgAdmin4, яка входить до пакету PostgreSQL, створимо базу даних та назвемо її ,наприклад, autoservice_dp (див. рис. 3.19).

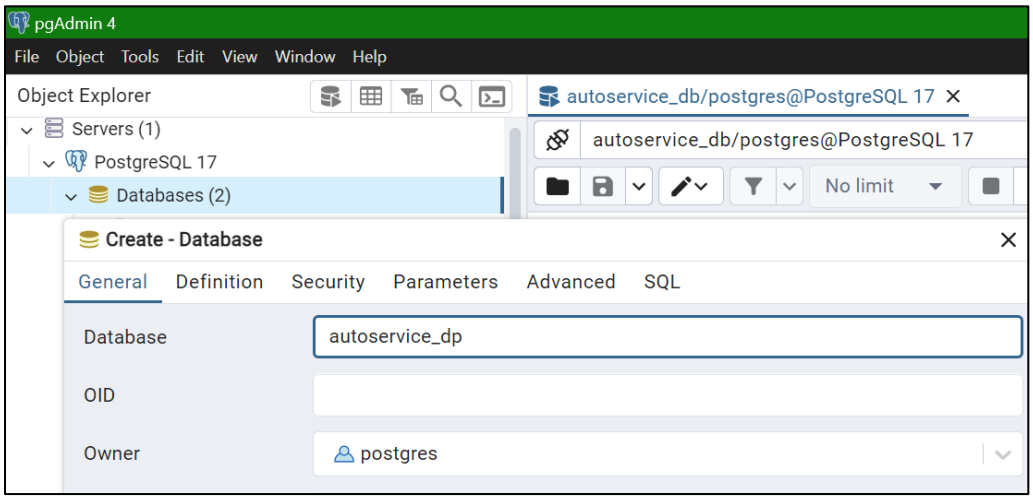


Рисунок 3.19 – Інтерфейс для створення бази даних

Застосуємо наступний код для створення та додавання таблиць із даними до бази даних. Для зручного управління базою у Node.js використовуємо Sequelize – ORM-бібліотеку, що спрощує взаємодію з реляційними базами.

Sequelize дозволяє визначати моделі, які відповідають таблицям бази, та встановлювати між ними зв’язки: «один до одного», «один до багатьох» та «багато до багатьох». Це забезпечує структурований підхід до роботи з даними, автоматичну валідацію, підтримку міграцій та оптимізує запити.

Використання Sequelize спрощує керування базою, зменшує потребу у складних SQL-запитах та підвищує продуктивність додатка.

Загалом, схема нашої бази даних зображена на рисунку 3.20.

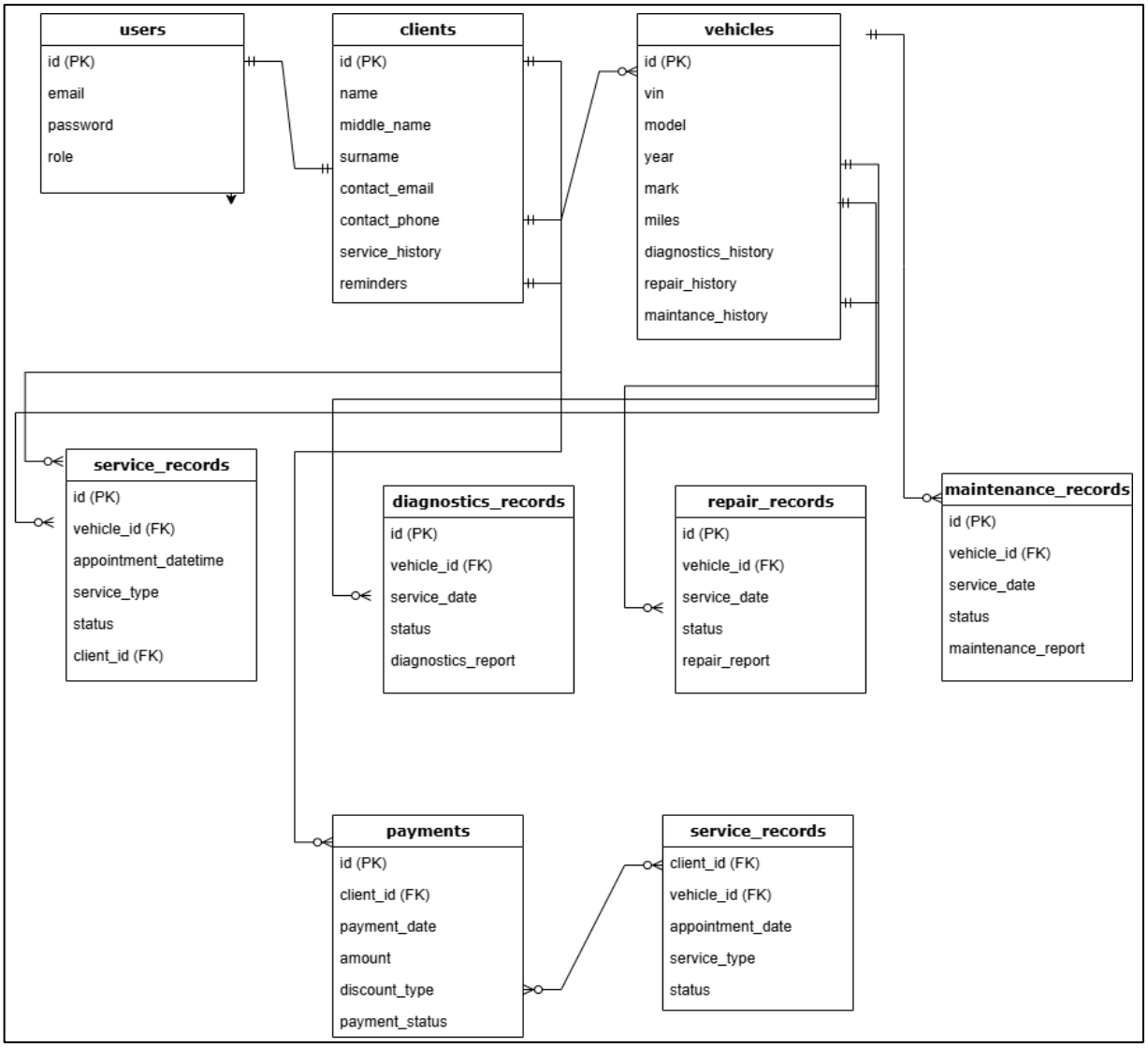


Рисунок 3.20 – Схема бази даних додатку для менеджменту автосервісу

Нижче, на рисунку 3.21, наведено приклад створення моделі для транспортного засобу.

```
export const Vehicle = sequelize.define('vehicle', {
    id: { type: DataTypes.INTEGER, primaryKey: true, autoIncrement: true },
    vin: { type: DataTypes.STRING, unique: true, allowNull: false },
    mark: { type: DataTypes.STRING, allowNull: false },
    model: { type: DataTypes.STRING, allowNull: false },
    year: {type: DataTypes.INTEGER, allowNull: false},
    miles: {type: DataTypes.INTEGER, allowNull: false},
    diagnostics_history: { type: DataTypes.TEXT },
    repair_history: { type: DataTypes.TEXT },
    maintenance_history: { type: DataTypes.TEXT },
});
```

Рисунок 3.21 – Код для створення ORM-об’єкту Sequelize

У реляційних базах даних зв’язки між таблицями відіграють важливу роль у забезпеченні цілісності даних та ефективному управлінні інформацією. У Sequelize для визначення взаємозв’язків між моделями використовуються спеціальні методи, що дозволяють встановлювати такі асоціації, як «один до одного» (One-to-One), «один до багатьох» (One-to-Many) та «багато до багатьох» (Many-to-Many).

Більш поширеним є зв’язок «один до багатьох» (див. рис. 3.22), коли один запис може мати багато пов’язаних записів. Наприклад, один клієнт автосервісу може володіти кількома автомобілями. У такому випадку встановлюється зв’язок hasMany на стороні основної таблиці та belongsTo у підлеглій таблиці, що дозволяє чітко визначити, які записи взаємопов’язані.

Зв’язок «один до одного» застосовується в тих випадках, коли один запис у першій таблиці відповідає лише одному запису у другій. Наприклад, якщо кожен автомобіль має лише одну картку технічного обслуговування, можна використати hasOne та belongsTo для встановлення такого зв’язку.

При цьому hasOne визначає, що один запис може мати лише один відповідний запис у зв’язаній таблиці, а belongsTo вказує, що запис у цій таблиці належить до певного запису в іншій.

```
// Клієнт має багато автомобілів
Client.hasMany(Vehicle, { foreignKey: 'client_id' });
Vehicle.belongsTo(Client, { foreignKey: 'client_id' });

// Клієнт має багато платежів
Client.hasMany(Payment, { foreignKey: 'client_id' });
Payment.belongsTo(Client, { foreignKey: 'client_id' });

// Транспортний засіб пов'язаний із записами діагностики
Vehicle.hasMany(DiagnosticsRecord, { foreignKey: 'vehicle_id' });
DiagnosticsRecord.belongsTo(Vehicle, { foreignKey: 'vehicle_id' });

// Транспортний засіб пов'язаний із записами ремонту
Vehicle.hasMany(RepairRecord, { foreignKey: 'vehicle_id' });
RepairRecord.belongsTo(Vehicle, { foreignKey: 'vehicle_id' });

// Транспортний засіб пов'язаний із записами технічного обслуговування
Vehicle.hasMany(MaintenanceRecord, { foreignKey: 'vehicle_id' });
MaintenanceRecord.belongsTo(Vehicle, { foreignKey: 'vehicle_id' });

// Транспортний засіб має багато платежів
Vehicle.hasMany(Payment, { foreignKey: 'vehicle_id' });
Payment.belongsTo(Vehicle, { foreignKey: 'vehicle_id' });

// Клієнт може мати багато записів по діагностиці, ремонту і технічному
обслуговуванню
Client.hasMany(DiagnosticsRecord, { foreignKey: 'client_id' });
DiagnosticsRecord.belongsTo(Client, { foreignKey: 'client_id' });

Client.hasMany(RepairRecord, { foreignKey: 'client_id' });
RepairRecord.belongsTo(Client, { foreignKey: 'client_id' });

Client.hasMany(MaintenanceRecord, { foreignKey: 'client_id' });
MaintenanceRecord.belongsTo(Client, { foreignKey: 'client_id' });
```

Рисунок 3.22 – Встановлення зв’язків між таблицями бази даних

Для складніших сценаріїв використовується зв’язок «багато до багатьох», коли записи обох таблиць можуть мати багато зв’язків між собою. Наприклад, автомобіль може проходити кілька типів сервісного обслуговування, і кожен тип обслуговування застосовується до багатьох

автомобілів. У таких випадках створюється проміжна таблиця, яка містить ідентифікатори обох основних таблиць. Це реалізується за допомогою belongsToMany, що дозволяє Sequelize автоматично керувати багатозв’язковими відношеннями.

Використання зв’язків у Sequelize значно спрощує управління даними, дозволяє автоматизувати складні запити без написання сирого SQL-коду, а також забезпечує гнучкість та масштабованість системи. Завдяки цьому можна легко інтегрувати додаткові функції, підтримувати розширення бази даних та забезпечувати швидке отримання інформації без зайвих витрат ресурсів.

3.3 Розробка серверної частини застосунку

Перейдемо до обробки логіки запитів до бази даних, таблиці якої будуть містити дані про користувачів, їх контактну інформацію, деталі візитів до сервісного центру, інформацію про лояльність (знижки, нагадування).

Для серверної частини нашого додатка, яка буде займатися обробкою даних, взаємодією з базами даних і виконанням бізнес-логіки використаємо платформу Node (Express) JS [8].

Ініціалізуємо проєкт за допомогою команд, наведених на рисунку 3.23.

```
mkdir server
cd server
npm init -y
npm install express pg bcrypt jsonwebtoken dotenv cors
```

Рисунок 3.23

Створимо структуру папок для нашої серверної частини додатку, яка узагальнено буде виглядати так, як на рисунку 3.24.

```
server/
├── models/
│   └── models.js
├── error/
│   └── apiError.js
├── uploads/
├── routes/
│   ├── main.js
│   ├── users.js
│   ├── clients.js
│   ├── vehicles.js
│   ├── services.js
│   └── payments.js
├── controllers/
│   ├── users.js
│   ├── clients.js
│   ├── vehicles.js
│   ├── diagnostics.js
│   ├── maintenances.js
│   ├── repairs.js
│   └── payments.js
├── utils/
│   └── email.js
│   └── emailReminder.js
│   └── initializeUser.js
├── middleware/
│   └── auth.js
│   └── checkRole.js
│   └── errorHandling.js
├── .env
├──db.js
├── server.js
```

Рисунок 3.24

Для додавання на сторінки проєкту графічних зображень ми будемо використовувати пакет `express-fileupload` , який встановлюємо за допомогою команди `npm install express-upload`. Для формування унікальних ідентифікаторів встановимо також пакет `uuid` за допомогою команди `npm i uuid`.

Додавання файлів (див. рис. 3.25) у контролері `vehicle` працює через завантаження зображення транспортного засобу на сервер із подальшим

збереженням його шляху в базі даних. Спочатку функція перевіряє, чи файл передано разом із запитом, і чи є він зображенням. Це здійснюється через перевірку об’єкта req.files та його MIME-типу.

```
if (!req.files || !req.files.img) {
return next(ApiError.badRequest("Image file is required"));
}

if (!img.mimetype.startsWith("image/")) {
return next(ApiError.badRequest("Invalid file type. Only image files are
allowed."));
}
```

Рисунок 3.25 – Функція для додавання зображення

Далі генерується (див. рис. 3.26) унікальне ім’я файлу за допомогою бібліотеки uuid та додається розширення оригінального файлу. Файл зберігається у папку uploads/vehicles, яка створюється, якщо її не існує.

```
let fileName = uuidv4() + path.extname(img.name);
const uploadPath = path.resolve(__dirname, "../uploads/vehicles");
await img.mv(path.join(uploadPath, fileName));
```

Рисунок 3.26 – Генерація унікального ім’я файлу

Після цього створюється запис у базі даних (див. рис. 3.27) з усіма переданими полями та шляхом до завантаженого зображення. Таким чином, функція забезпечує перевірку файлів, їх збереження на сервері та інтеграцію з базою даних, щоб дані автомобіля були доступними для подальшої обробки. Успішний результат повертається у форматі JSON із деталями створеного автомобіля.

Таким чином, контролери забезпечують чітку структуру коду, ізоляцію логіки додатку та гнучкість для розширення функціональності.

Контролер `VehiclesController` забезпечує взаємодію з автомобілями у системі через функції для створення, редагування, отримання всіх автомобілів та отримання автомобілів, прив’язаних до конкретного клієнта. Ось як працюють його основні функції.

```
const vehicle = await Vehicle.create({
  vin,
    mark,
    model,
    year,
    miles,
    diagnostics_history,
    repair_history,
    maintenance_history,
    client_id,
    img: `uploads/vehicles/${fileName}`,
});
```

Рисунок 3.27 – Додавання створеного запису у таблицю Vehicle

Функція `createVehicle` відповідає за додавання нового автомобіля до бази даних. Вона приймає дані про автомобіль із тіла запиту, а також зображення, яке перевіряється на коректність типу. Унікальне ім’я генерується для файлу зображення, і він зберігається в папку на сервері. Усі дані, включаючи шлях до зображення, додаються до бази, після чого повертається створений запис. Ключові рядки коду функції зображено на рисунку 3.28.

```
let fileName = uuidv4() + path.extname(img.name);
await img.mv(path.join(uploadPath, fileName));
const vehicle = await Vehicle.create({ vin, mark, img:
`uploads/vehicles/${fileName}` });
return res.json(vehicle);
```

Рисунок 3.28 – Основні фрагменти коду функції для створення авто

Функція `getAllVehicles` (див. рис. 3.29) реалізує пагінацію для отримання списку всіх автомобілів. Вона обчислює кількість записів на сторінку (`limit`) і номер поточної сторінки (`page`), щоб забезпечити зручну навігацію. Дані повертаються у вигляді структури, яка включає загальну кількість записів, кількість сторінок, номер поточної сторінки та список автомобілів.

Функція `editVehicle` (див. рис. 3.30) дозволяє оновлювати інформацію про автомобіль. Вона приймає `vehicleID` через параметри запиту та перевіряє, чи існує відповідний запис у базі. Дані з тіла запиту використовуються для оновлення полів автомобіля. Якщо оновлення успішне, функція повертає оновлений запис.

```
const vehicles = await Vehicle.findAndCountAll({ limit, offset });
const response = { totalItems: vehicles.count, totalPages, currentPage,
data: vehicles.rows };
return res.status(200).json(response);
```

Рисунок 3.29 – Основні фрагменти коду функції для отримання списку автомобілів

```
const vehicles = await Vehicle.findAndCountAll({ limit, offset });
const response = { totalItems: vehicles.count, totalPages, currentPage,
data: vehicles.rows };
return res.status(200).json(response);
```

Рисунок 3.30 – Основні фрагменти коду функції для редагування авто

Функція `getAllVehiclesForClient` (див. рис. 3.31) дозволяє отримати всі автомобілі, які прив’язані до конкретного клієнта.

```
const vehicles = await Vehicle.findAndCountAll({ limit, offset });
const response = { totalItems: vehicles.count, totalPages, currentPage,
data: vehicles.rows };
return res.status(200).json(response);
```

Рисунок 3.31 – Основні фрагменти коду функції для належного клієнту автомобіля

Вона перевіряє, чи передано clientId, і виконує пошук у базі. Якщо для клієнта немає автомобілів, повертається повідомлення про помилку. У разі успіху повертається список транспортних засобів, пов’язаних із вказаним клієнтом.

Усі функції мають обробку помилок і повертають відповідні повідомлення, якщо виникають проблеми, наприклад, відсутність автомобіля чи некоректний запит. Це забезпечує стабільну роботу контролера та інтеграцію із системою.

Функції інших контролерів реалізовані аналогічним чином, з урахуванням специфіки обробки відповідних даних і бізнес-логіки. Вони також забезпечують прийом, перевірку та обробку вхідних даних, взаємодію з базою даних і формування відповідей клієнту.

Обробка помилок у проєкті реалізована за допомогою спеціального `middleware`, яке централізовано перехоплює всі помилки, що виникають під час роботи контролерів або запитів. Це дозволяє забезпечити стабільність системи, формувати зрозумілі відповіді для клієнтів і підтримувати єдиний підхід до обробки виняткових ситуацій.

Коли в контролері виникає помилка, вона передається через функцію `next()` з переданим об’єктом помилки. Наприклад, у разі некоректного запиту контролер може передати помилку (рис. 3.32).

```
if (!client) {
    return next(ApiError.notFound("Client not found"));
}
```

Рисунок 3.32

Передана помилка потрапляє до `middleware`, яке визначає тип помилки. Якщо це екземпляр ApiError, з нього береться статус код і повідомлення, які відправляються у відповідь клієнту (рис. 3.33).

```
if (err instanceof ApiError) {
    return res.status(err.status).json({ message: err.message });
}
```

Рисунок 3.33

Якщо помилка не є об’єктом ApiError, сервер повертає стандартну відповідь із кодом 500 і повідомленням “Unexpected error”.

Клас ApiError містить стандартні методи для створення типових помилок, наприклад:

- `badRequest` для помилок 400, коли запит некоректний;
- `notFound` для ситуацій, коли ресурс не знайдено;
- `internalError` для внутрішніх проблем сервера.

Такий підхід дозволяє централізовано обробляти помилки, підтримувати єдиний формат відповідей клієнту і легко розширювати логіку у разі потреби. Це робить систему стійкою до помилок і зручною для обслуговування.

Роутінг у проєкті реалізований через розподіл маршрутів по окремих модулях, що відповідають різним частинам функціональності. Основна мета роутінгу – зв’язати HTTP-запити клієнтів із відповідними функціями контролерів. Логіка побудована на використанні бібліотеки Express, яка забезпечує зручність і гнучкість при організації маршрутів.

У проєкті маршрути поділено на кілька модулів, кожен з яких відповідає за обробку певної групи функцій. Наприклад, існують маршрути для роботи з клієнтами, автомобілями, сервісними операціями (ремонт, технічне обслуговування, діагностика) та платежами. Усі маршрути зібрані у відповідній папці, де для кожного модуля створений окремий файл, що полегшує підтримку та розширення проєкту.

Основний файл маршрутизації (див. рис. 3.34) імпортує всі ці модулі та додає їх до головного об’єкта Express.

```
const mainRouter = express.Router();
mainRouter.use('/clients', clientsRouter);
mainRouter.use('/vehicles', vehiclesRouter);
mainRouter.use('/services', servicesRouter);
mainRouter.use('/payments', paymentsRouter);
module.exports = mainRouter;
```

Рисунок 3.34 – Зміст файлу із маршрутами

У кожному з модулів визначені маршрути для роботи з відповідними ресурсами (див. рис. 3.35). Наприклад, у файлі clientsRouter.js реалізовано маршрути для створення, редагування, отримання одного клієнта чи списку клієнтів.

```
      router.post('/createClient', clientsController.createClient);
      router.put('/editClient/:clientId', clientsController.editClient);
      router.get('/getAllClients', clientsController.getAllClients);
      router.get('/getOneClient/:clientId',
clientsController.getOneClient);
```

Рисунок 3.35 – Маршрутизація запитів згідно дій

Подібним чином організовано й інші маршрути, наприклад, для роботи з автомобілями в vehiclesRouter.js (див. рис. 3.36).

```
      router.post('/createVehicle', vehiclesController.createVehicle);
      router.put('/editVehicle/:vehicleId',
vehiclesController.editVehicle);
      router.get('/getAllVehicles', vehiclesController.getAllVehicles);
      router.get('/getAllVehiclesForClient/:clientId',
vehiclesController.getAllVehiclesForClient);
```

Рисунок 3.36 – Маршрутизація запитів для транспортних засобів

Усі маршрути відповідають RESTful-принципам, що забезпечує логічність і передбачуваність взаємодії з API. Наприклад:

– POST – для створення ресурсів;
– GET – для отримання даних;
– PUT – для оновлення даних;
– DELETE – для видалення ресурсів (якщо потрібно реалізувати).

Централізована структура роутінгу дозволяє зручно інтегрувати нові модулі чи розширювати функціональність, додаючи нові маршрути у відповідні файли. Це забезпечує масштабованість та зручність підтримки всього проєкту.

Авторизація належного користувача робиться на основі JWT токена [9]. Структура токена показана на рисунку 3.37.

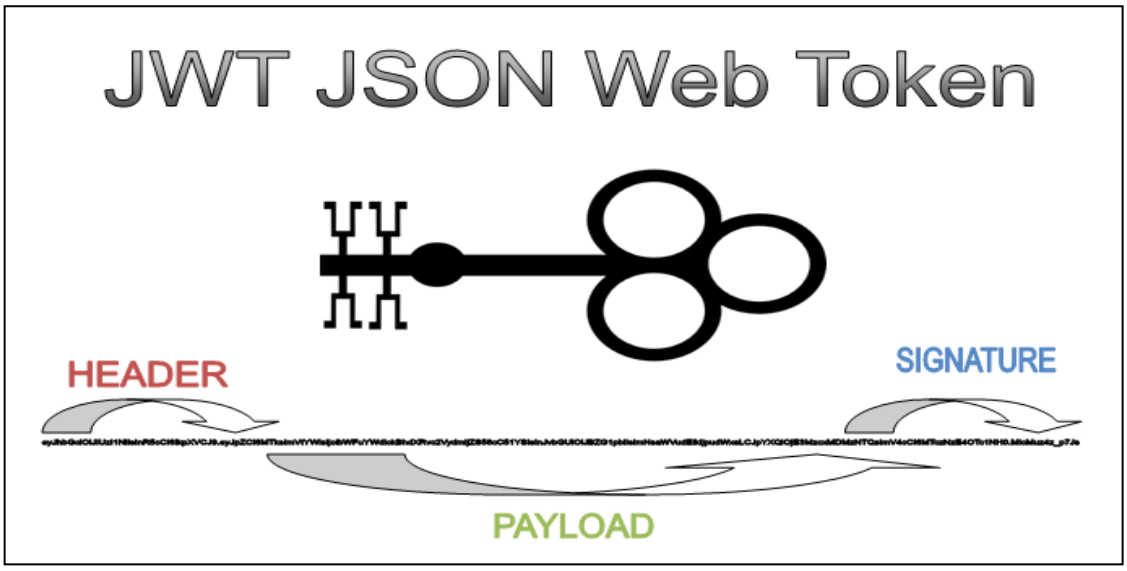


Рисунок 3.37 – Основні елементи JSON Web токену

Основна ідея роботи токену наступна: при спробі запиту закритої для неавторизованого користувача інформації, відбувається запит логіну та пароля. Після введення яких сервер генерує токен для авторизації та повертає його у клієнтську частину. Після спроби перейти на потрібну сторінку, клієнтська частина надсилає запит серверу, заголовок якого включає частину токену. Сервер валідує його по секретному ключу і якщо він є справжнім, то користувач отримує доступ до потрібної сторінки. Структура взаємодії клієнт-сервер наведена на рисунку 3.38.

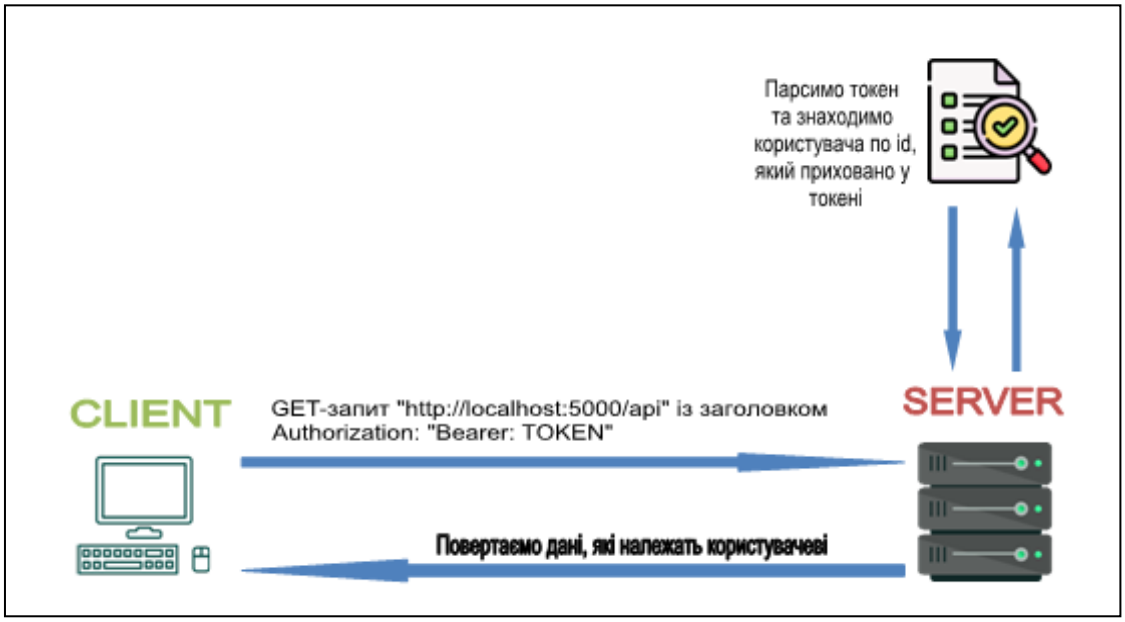


Рисунок 3.38 – Взаємодія клієнт-сервер при проведенні авторизації

Для реалізації цього функціоналу встановимо пакети `jsonwebtoketn` та `bcrypt` за допомогою команди `npm i jsonwebtoken`.

Це дозволяє створювати токени та не зберігати паролі у відкритому вигляді у базі даних.

Розглянемо детальніше логіку роботи. В нашій системі процес первинної реєстрації користувача починається з того, що новий користувач передає електронну пошту та пароль у запиті до сервера. Сервер перевіряє, чи є в базі даних запис із таким email, і якщо його немає, то створює нового користувача з роллю `Client`. Для цього пароль хешується за допомогою бібліотеки `bcrypt`, щоб забезпечити безпеку зберігання, а також генерується унікальний токен JWT. Токен містить закодовану інформацію про ID користувача, його email та роль, що дозволяє системі ідентифікувати користувача при наступних запитах. Після успішної реєстрації токен повертається користувачеві, щоб той міг використовувати його для авторизації.

Для захищених маршрутів система використовує `middleware` для обробки токена. `authMiddleware` відповідає за перевірку, чи був наданий токен у заголовку запиту, і чи є він дійсним. Якщо токен пройшов перевірку, дані з нього додаються до запиту (`req.user`), що дозволяє іншим компонентам системи знати, який користувач виконує запит. Наприклад, це можуть бути ID, email або роль користувача. Якщо токен недійсний або відсутній, `middleware` повертає помилку `401 Unauthorized`, і запит не доходить до основної логіки.

Система розділяє рівні доступу користувачів за допомогою іншого middleware – `checkRole`. Цей компонент перевіряє, чи відповідає роль користувача дозволеним для конкретного маршруту. Наприклад, маршрути для додавання нового клієнта або редагування даних клієнтів доступні лише користувачам із ролями Admin та Employee. Якщо роль не входить до списку дозволених, запит завершується з помилкою 403 Forbidden. Таким чином, навіть якщо користувач має валідний токен, він не зможе виконувати дії, які не відповідають його рівню доступу.

Коли `Client` здійснює запит, наприклад, на перегляд своїх даних, система перевіряє, чи збігається email із токена з email у базі даних для цього клієнта. Це гарантує, що користувачі не отримають доступ до чужих даних. Для ролей `Admin` та `Employee` таких обмежень немає, і вони можуть отримати доступ до будь-якого запису, оскільки їх роль передбачає ширший функціонал.

Загалом, система базується на централізованій логіці контролю доступу. Middleware виконує основні перевірки безпеки до того, як запит досягне контролера. Це дозволяє зберігати контролери максимально чистими та зосередженими на бізнес-логіці. Наприклад, у випадку маршруту редагування клієнта, роль користувача перевіряється в middleware перед тим, як основна логіка оновлення даних виконується в контролері. Такий підхід забезпечує як гнучкість, так і безпеку, дозволяючи легко додавати нові маршрути та ролі в майбутньому.

Під час реєстрації нового користувача важливо підтвердити його електронну пошту, щоб гарантувати достовірність наданої інформації та забезпечити безпеку системи. Процес відправлення електронного листа для підтвердження реєстрації зазвичай складається з наступних кроків:

– реєстрація користувача: користувач заповнює форму реєстрації, надаючи необхідні дані, такі як електронна пошта та пароль;
– створення запису в базі даних: після подання форми система створює новий запис у базі даних з інформацією про користувача – на цьому етапі обліковий запис зазвичай позначається як неактивний, одночасно створюється та додається до бази даних 6-ти значний код підтвердження та час його створення;
– відправлення електронного листа: за допомогою функції `sendVerificationEmail` та бібліотеки `nodemailer` на вказану електронну адресу надсилається лист із кодом підтвердження;
– користувач переходить за посиланням: отримавши лист, користувач вводить ориманий код у поле для підтвердження;

– обробка підтвердження на сервері: сервер отримує запит із кодом та перевіряє на відповідність запису в базі даних – якщо токен дійсний, а термін його дії не перевищує ліміту з моменту створення, обліковий запис активується, змінюючи статус на «Активний» (кількість спроб також обмежується);
– підтвердження активації: після успішної активації система може перенаправити користувача на сторінку входу або інформувати його про успішне підтвердження реєстрації.

Приклад функції наведено на рисунку 3.39.

Генеруємо код підтвердження (див. рис. 3.40) за допомогою функції генерації випадкових чисел, задаємо час створення коду та період його дії. Додаємо до моделі User необхідні дані.

```
      // Функція для відправлення email
      export const sendVerificationEmail = async (email, code) => {
      const transporter = nodemailer.createTransport({
      host: 'smtp.ukr.net', // Замість 'smtp.example.com' вкажіть свій SMTP
сервер
      port: 465, // 465 для SSL або 587 для TLS
      secure: true, // true для SSL
      auth: {
      user: process.env.SERVER_EMAIL, // Серверна пошта
      pass: process.env.SERVER_PASS, // Пароль від пошти
      },
      });
      const mailOptions = {
      from: `"AutoService" <${process.env.SERVER_EMAIL}>`,
      to: email, // Email отримувача
      subject: 'Підтвердження реєстрації',
      text: `Ваш код підтвердження: ${code}. Код дійсний протягом 10
хвилин.`,
      };
      await transporter.sendMail(mailOptions);
      };
```

Рисунок 3.39 – Функція для відправлення листа із кодом підтвердження

```
      const generateVerificationCode = () => {
    return Math.floor(100000 + Math.random() * 900000).toString();
};
      …
      const currentTime = moment().utc();
const codeExpiry = currentTime.clone().add(10, 'minutes');
      …
      verification_code: verificationCode,
code_expiry: codeExpiry.utc().format(), // Зберігаємо в UTC
createdAt: currentTime.utc().format(), // Зберігаємо в UTC
```

Рисунок 3.40 – Генерація та збереження коду підтвердження

Таким чином, підтвердження реєстрації за допомогою коду, надісланого на електронну пошту, забезпечує верифікацію користувача та підтверджує дійсність наданої електронної адреси, що підвищує безпеку облікового запису.

Реалізуємо також логіку для нагадування втраченого пароля. Логіка функції нагадування пароля працює наступним чином. Користувач надсилає запит із електронною поштою та VIN-кодом автомобіля. Система спочатку перевіряє, чи існує користувач із вказаною електронною поштою у моделі User. Далі перевіряється (рис. 3.41), чи ця електронна пошта співпадає з `contact_email` у моделі `Client`, тобто чи цей email дійсно прив'язаний до клієнта. Після цього здійснюється перевірка, чи VIN-код автомобіля з моделі `Vehicle` належить цьому клієнту через поле `client_id`. Якщо всі перевірки успішні, генерується новий пароль, який хешується за допомогою `bcrypt` і зберігається в базу даних. Потім система надсилає новий пароль на вказану електронну адресу за допомогою функції відправки листів `nodemailer`. У разі успішного виконання запиту користувач отримує повідомлення про те, що новий пароль надіслано на його електронну пошту. Ця логіка забезпечує безпечне та просте відновлення пароля, перевіряючи, що електронна пошта та VIN-код належать одному й тому ж клієнту.

```
      const client = await Client.findOne({ where: { contact_email: email } 
});

if (!client) {
    return next(ApiError.badRequest('Email не привязаний до жодного 
клієнта'));
}
```

Рисунок 3.41 – Приклад коду для перевірки наявності email

Логіка відправлення нагадувань виконана наступним чином. На серверній стороні запит обробляється маршрутом, де виконується аутентифікація та перевірка ролі користувача, щоб упевнитися, що тільки адміністратор або співробітник може створювати чи видаляти нагадування.

Контролер setReminder приймає дані нагадування, такі як текст і дата, знаходить відповідного клієнта в базі даних, і зберігає ці дані в полі reminders. Якщо ж позначку з чекбокса було знято, контролер removeReminder видаляє це нагадування, встановлюючи значення null для відповідного запису.

Відправлення нагадувань реалізовано через бібліотеку node-schedule, яка дозволяє налаштувати регулярну перевірку бази даних. Щодня о 08:00 запускається функція перевірки нагадувань. Вона визначає, чи є в базі записи, де дата нагадування збігається з поточною, і якщо такі записи є, вони обробляються. Для кожного клієнта з активним нагадуванням викликається функція надсилання листа через nodemailer. Електронна пошта відправляється на адресу клієнта з текстом нагадування. Після успішного надсилання нагадування позначається як виконане шляхом очищення відповідного поля у базі даних.

Функція надсилання листів працює через SMTP-сервер, наприклад, Gmail або Ukr.net. У налаштуваннях вказується електронна адреса сервера, порт, а також облікові дані для аутентифікації. Це дозволяє забезпечити надійну доставку листів клієнтам. Крім того, для зручності реалізовано ручний запуск перевірки нагадувань через окремий API-роут, що дозволяє адміністраторам або розробникам перевіряти функціонал без необхідності чекати часу, зазначеного у розкладі.

Таким чином, додавання нагадувань починається на клієнтській стороні з інтуїтивного інтерфейсу таблиці. Дані автоматично зберігаються на сервері та регулярно перевіряються. Це дозволяє гарантувати, що клієнт отримає потрібне повідомлення вчасно, а адміністратори можуть ефективно керувати нагадуваннями. Система є зручною, безпечною та легко розширюваною, наприклад, для додавання інших каналів сповіщень, таких як SMS або push-нотифікації.

Ключова відмінність розроблюваного продукту від існуючих аналогів полягає в можливості підключення Bluetooth-модуля, побудованого на

контролері ELM-327. Це забезпечує зчитування кодів помилок електронних систем автомобіля за протоколом OBD-II з подальшим збереженням отриманих даних у базі даних.

Для розуміння принципу та методів діагностування автомобілів за допомогою згаданого вище адаптера пропонується ознайомитися із матеріалом наступного розділу.

3.3.1 Структура та принципи діагностики автомобільних електронних систем за стандартом OBD-II

Бурхливий прогрес в області електроніки і електротехніки за останні роки і десятиліття призвів до різкого збільшення кількості електронних компонентів в автомобілі. Поряд з гідравлікою і пневматикою електроніка проникла в усі частини сучасного автомобіля. Окремі електронні компоненти і комплексні електронні системи стають все компактнішими, дешевшими і, разом з тим, ефективнішими. В результаті з’являються нові можливості використання електроніки в автомобілі, що дозволяють постійно розширювати обсяг вже існуючих функцій.

Але наявність все більшої кількості електронних пристроїв викликала нову проблему – наявність величезної кількості проводів, що з’єднують блоки та датчики між собою. Стало зрозуміло, що потрібно вживати заходів щодо скорочення сполучних елементів та раціоналізувати схемотехніку, тим самим спростивши та здешевивши виробництво. Було ухвалено рішення відмовитися від принципу «один провід для одного пристрою» та впроваджено принцип передачі даних в інформаційну шину, де вже кожен датчик та блок зчитуватиме необхідну саме йому команду. За такого підходу кратно зменшується кількість з’єднань та значно підвищується загальна надійність системи. Крім того, подібна архітектура дозволяє порівняно легко проводити діагностику комплексних систем із безліччю електронних блоків.

Описаний вище підхід був прийнятий за основу. І, починаючи з 1983 року, коли компанія Bosch почала вести розробки в цьому напрямку, отримав назву CAN (Controller Area Network). Результат не забарився і вже в 1986 році в Детройті на конференції Асоціації Автомобільних Інженерів Bosch представила працюючий прототип системи.

CAN є послідовною шиною, що підтримує одночасну роботу багатьох ведучих пристроїв. Це означає, що всі вузли CAN-мережі мають можливість передавати дані і декілька вузлів одночасно можуть давати запит на шину (див. рис. 3.42).

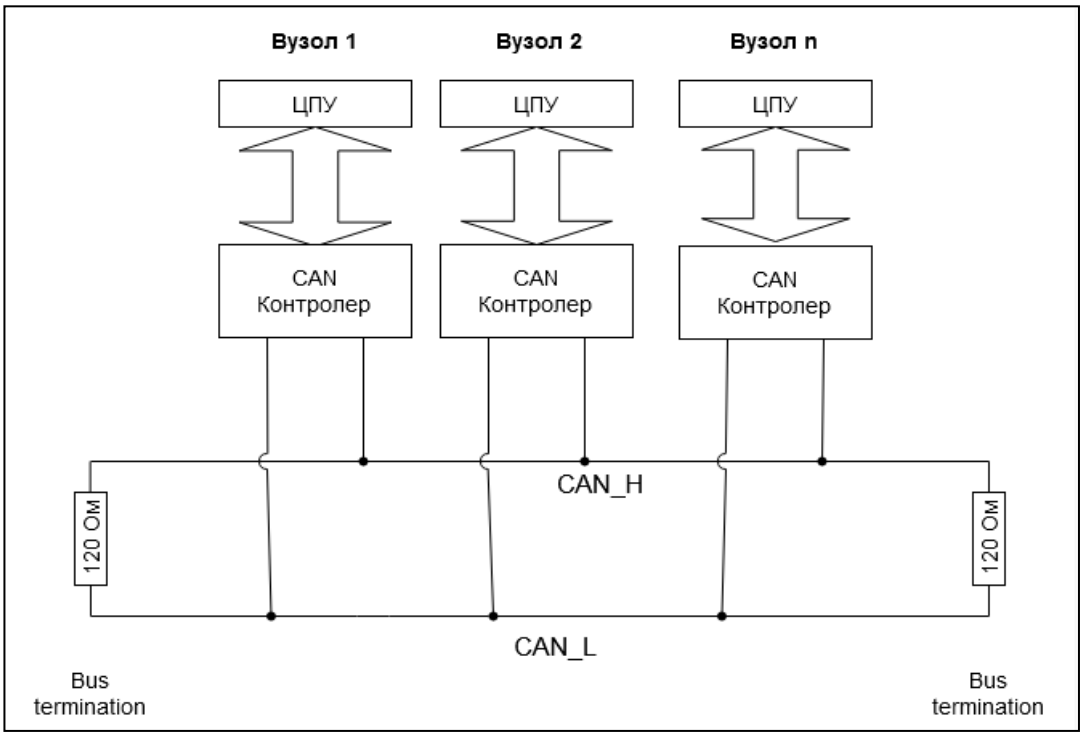


Рисунок 3.42 – Топологія мережі CAN

На ринку CAN присутній у двох версіях: версія 2.0A задає 11-бітову ідентифікацію повідомлень (тобто в системі може бути 2048 учасників), версія 2.0B – 29-бітову (536 млн. учасників). Слід відзначити, що версія 2.0B, яку часто називають FullCAN, поступово витісняє версію 2.0A, яку називають, також, BasicCAN.

Магістраль CAN використовує двопровідникову звиту пару і працює з максимальною швидкістю трансляції 1 Мбіт/с на довжині до 40 м. Із

збільшенням відстані зменшується максимальна швидкість трансляції (напр. 250 кбіт/с до 250 м).

Повернемося до стандартів OBD, які на сьогодні фактично стали синонімом «діагностики». Але чи насправді вони такими завжди були?

Історія формування стандарту OBD II (On Board Diagnostic) має свої корені в минулому столітті, у 1950-х роках, коли уряд США раптово побачив негативний вплив автомобільної промисловості на екологію [10]. Початково влада стояла перед проблемою, не знаючи, як її вирішити. Згодом було створено різні комітети для аналізу ситуації, робота яких і багато оцінок призвели до ухвалення законодавчих актів. Проте, виробники автомобілів, хоч і видавалися, що дотримуються цих актів, насправді не завжди дотримувалися їх, ігноруючи обов'язкові тестові процедури та стандарти.

На початку 1970-х законодавці спробували знову впровадити зміни, проте їхні зусилля були ігноровані. Тільки у 1977 році ситуація почала змінюватися. Енергетична криза і зменшення виробництва змусили виробників прийняти рішучі заходи, щоб зберегти себе. Це спонукав Департамент контролю за повітряним середовищем (ARB) і Агентство з охорони довкілля (EPA) до серйозних дій.

На цьому тлі розвивалася концепція стандарту OBD II. Раніше кожен виробник використовував власні системи і методи контролю викидів. Щоб змінити це, Асоціація автомобільних інженерів (SAE) запропонувала декілька стандартів. Можна сказати, що OBD виникла, коли ARB зробило обов'язковими багато стандартів SAE для автомобілів, що випускалися починаючи з 1988 року в Каліфорнії. Спочатку система OBD II була простою і обмежувалася деякими аспектами, такими як датчик кисню, система рециркуляції вихлопних газів (EGR), система подачі палива та блок управління двигуном (ECM) щодо викидів.

Уряд США був під постійним тиском з боку автосервісів та захисників довкілля, які всі вимагали втручання EPA. У результаті було розроблено широкий набір процедур і стандартів, використовуючи ідеї ARB та стандарти

SAE. До 1996 року всі виробники автомобілів, що продаються в США, повинні були відповідати цим вимогам. Це призвело до створення другого покоління системи бортового діагностування – On-Board Diagnostics II або OBD II.

Варто зауважити, що концепція OBD II виникла не відразу, а розвивалася протягом багатьох років. OBD II – це не система управління двигуном, а набір правил і вимог, які повинні дотримуватися виробниками для відповідності федеральним нормам з викидами. OBD II визначає тип діагностичного роз'єму (див. рис. 3.43), його протоколи зв'язку та формати повідомлень.

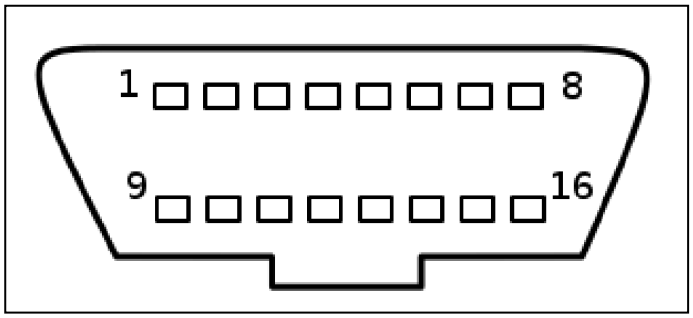


Рисунок 3.43 – Схема розташування виводів порту OBD2

У Європейському Союзі було прийнято EOBD – варіант автодіагностики, заснований на OBD-II, який став обов'язковим для всіх автомобілів з січня 2001 року. Також існує японський стандарт – JOBD. Попередня версія автодіагностики OBD-I існувала до 1989 року, але не отримала широкого поширення. Наразі ведуться розробки нової версії автодіагностики – OBD-III.

Розглянемо підтримувані діагностичні протоколи OBD- II.

SAE J1850 PWM (Pulse Width Modulation – модуляція ширини імпульсу). Високошвидкісний протокол, забезпечує продуктивність в 41,6 Кбайт/с. Він використовується в марках Ford, Jaguar і Mazda. Відповідно до протоколу PWM сигнали передаються по двох проводах, приєднаним до контактів 2 і 10 діагностичного роз'єму.

SAE J1850 VPW (Variable Pulse Width – змінна ширина імпульсу). Протокол VPW підтримує передачу даних зі швидкістю 10,4 Кбайт/с і застосовується в автомобілях марок General Motors (GM) і Chrysler. Протокол VPW передбачає передачу даних по одному дроту, приєднаному до контакту 2 діагностичного роз'єму.

ISO 9141-2 розроблений ISO і застосовується в більшості європейських і азійських автомобілів, а також в деяких автомобілях Chrysler. Використовує контакт 7 (K-Line) і опціонально контакт 15 (L-Line).

ISO 14230 KWP2000 (Keyword Protocol 2000) на фізичному рівні ідентичний ISO 9141. Також використовує контакт 7 (K-Line) і опціонально 15 (L-Line).

ISO 15765 CAN. У рамках OBDII використовує 2 контакти: 6 і 14. Є найбільш швидкісним і сучасним.

Щоб ефективно пояснити різні типи сигналів, корисно спочатку зрозуміти типове застосування CAN. Кожен вузол CAN складається з мікроконтролера з вбудованим контролером CAN і трансивера, який прив'язаний до шини. Дискретна реалізація цього показана на рисунку 3.44 [11].

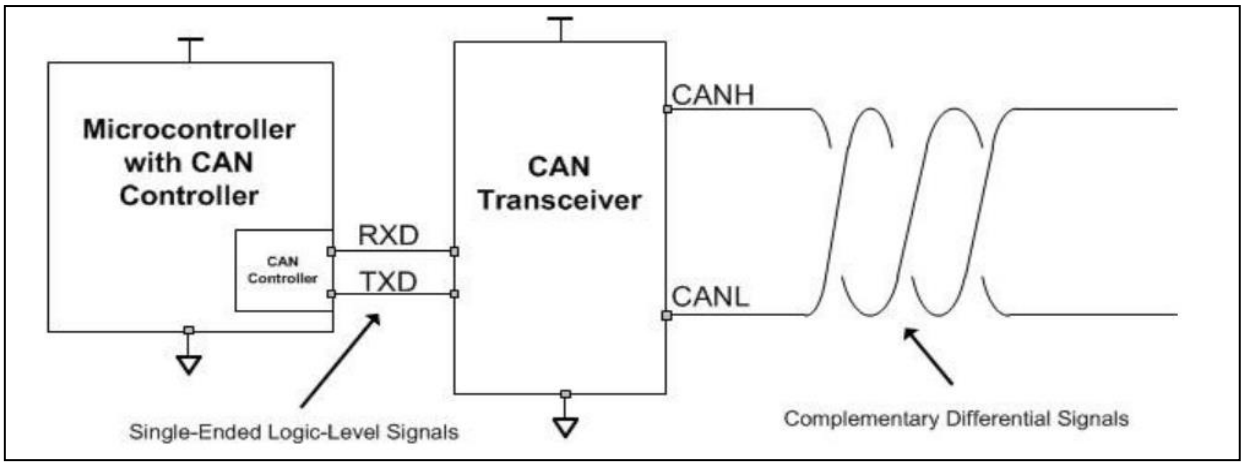


Рисунок 3.44 – Контролер шини CAN та трансивер

Трансивер CAN обробляє два типи сигналів: односторонні сигнали (TXD і RXD) і диференціальні сигнали (CANH і CANL). Під час нормальної

роботи трансивер CAN перетворює односторонній вихідний сигнал логічного рівня (TXD) від контролера CAN на диференціальний сигнал. Він також перетворює диференціальний сигнал на шині назад у односторонній логічний сигнал (RXD) для введення в контролер CAN. По суті, трансивер забезпечує диференціальне керування та можливість диференціального прийому до та від шини CAN.

Шина CAN має два логічних стани: домінантне і рецесивне. Домінуючий стан виникає, коли логічний низький рівень застосовується до вхідного контакту передачі (зазвичай називається TXD) трансивера. Рецесивний стан відповідає логічному високому рівню на вхідному контакті передачі трансивера. Схему станів наведено на рисунку 3.45.

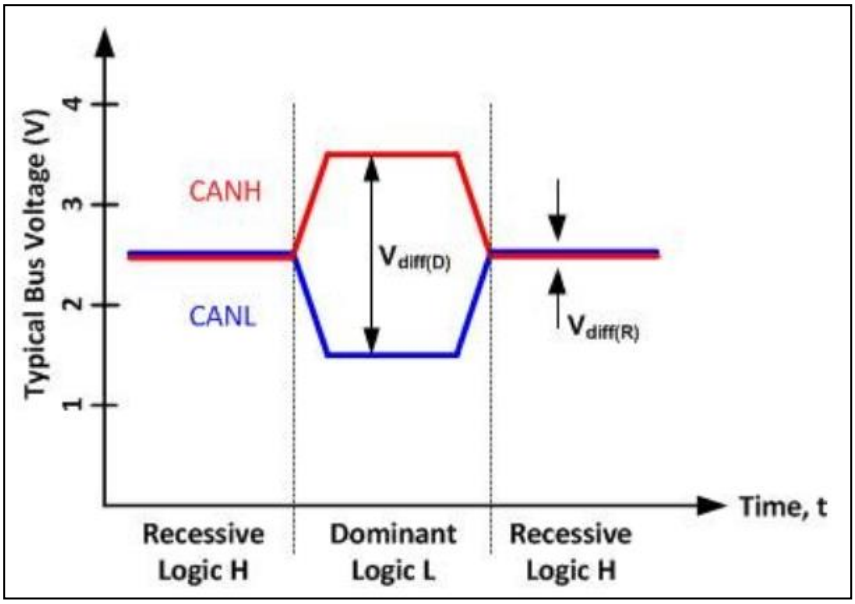


Рисунок 3.45 – Рівні сигналів у шині CAN

Як бачимо, у рецесивному стані контакти шини CANH і CANL зміщені на однаковий рівень: приблизно 2,5 В. Під час домінуючого стану висновок шини CANH зміщується до вищого потенціалу напруги (приблизно 3,5 В), а висновок шини CANL зміщується до нижчого потенціалу напруги (приблизно 1,5 В). Віднімаючи потенціал напруги двох контактів шини, ми зможемо визначити логічний стан шини за допомогою рівняння на рисунку 3.46.

$$V_{diff} = V_{CANH} - V_{CANL}$$

Рисунок 3.46

Якщо значення Vdiff на шині менше 0,5 В, шина вважається рецесивною. З іншого боку, значення Vdiff більше 0,9 В вказують на те, що шина перебуває в домінантному стані. Нарешті, для значень Vdiff між 0,5 В і 0,9 В стан шини не визначений.

Оскільки різниця між двома сигналами використовується для визначення стану шини, цей тип сигналізації відомий як диференціальна сигналізація. Крім того, сигнали CANH і CANL зазвичай називають комплементарними одинарними, оскільки нам потрібно знати потенціал напруги обох сигналів, щоб визначити логічний стан шини.

Рецесивний стан існуватиме на шині, лише якщо всі трансивери, підключені до шини, передають рецесивний стан, оскільки рецесивний стан є слабко зміщеним, тоді як домінантний стан сильно зміщеним. Це аналогічно дротовому логічному з’єднанню AND. Усі трансивери повинні передавати логічний високий сигнал, щоб вихід шини був логічним високим. Якщо хоча б один трансивер передає логічний 0, вся шина слідуватиме цьому стану та перебуватиме в домінантному стані.

Але постає питання, як практично застосовувати ці сигнали під час проведення діагностики?

Відповідно до закону, майже всі автомобілі, які виробляються сьогодні, повинні мати інтерфейс для підключення діагностичного тестового обладнання. Передача даних через ці інтерфейси відповідає кільком стандартам, але жоден із них не можна безпосередньо використовувати на персональному комп’ютері чи інших інтелектуальних пристроях. На щастя, є рішення – програмований контролер ELM327 [12] (див. рис. 3.47), призначений для роботи в якості моста між портами бортової діагностики (OBD) і стандартним інтерфейсом RS232. На додаток до можливості

автоматичного виявлення та інтерпретації дев'яти протоколів OBD, ELM327 також забезпечує підтримку високошвидкісного зв'язку, режиму сну з низьким енергоспоживанням і стандарт J1939 для вантажівок і автобусів.

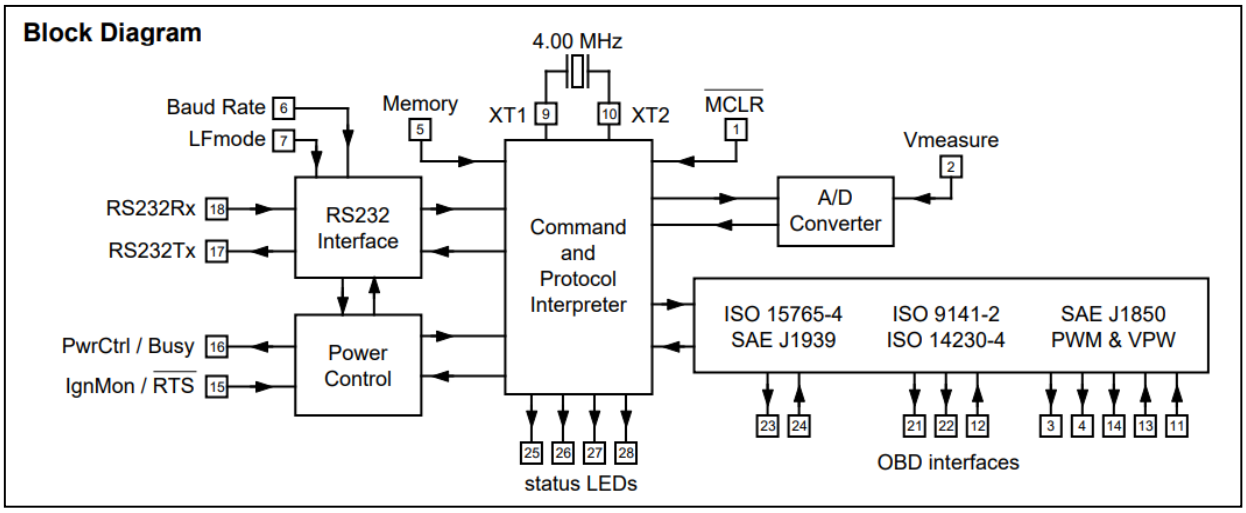


Рисунок 3.47 – Схематична діаграма мікроконтролера ELM327

OBD-II надає доступ до даних з блоку керування двигуном (ECU) і є цінним джерелом інформації під час усунення несправностей усередині автомобіля. Стандарт SAE J1979 визначає метод запиту різноманітних діагностичних даних і список стандартних параметрів, які можуть бути доступні з ECU. Різні доступні параметри адресуються «ідентифікаційними номерами параметрів» або PID, які визначені в J1979. Щоб отримати список основних PID, їх визначення та формулу для перетворення необроблених вихідних даних OBD-II у значущі діагностичні одиниці, див. OBD-II PID. Від виробників не вимагається впровадження всіх PID, перелічених у J1979, і їм дозволяється включати власні PID, яких немає в списку. Система запиту PID і отримання даних надає доступ до даних продуктивності в реальному часі, а також до позначених DTC. Перелік загальних кодів DTC (Diagnostic Trouble Codes) OBD-II, запропонованих SAE, можна знайти у таблиці кодів OBD-II зазначеного вище стандарту. Індивідуальні виробники часто доповнюють набір кодів OBD-II додатковими власними кодами DTC.

Нижче, на рисунку 3.48, наведено пояснення структури коду DTC [13].



Рисунок 3.48 – Структура коду несправності, згідно стандарту SAE J1979

Перший символ – «Система». Перша літера коду несправності ідентифікує систему (частину автомобіля), до якої належить помилка:

– P = Powertrain (двигун та\або АКПП);
– B=Body (подушки безпеки, центральний замок, електросклопідйомники);
– C = Chassis (ходова частина);
– U=Undefined (шина взаємодії між електроними блоками).

Другий символ – «Тип коду». Друга позначка визначає код несправності до загального типу помилок (однаковому для всіх OBD II двигунів) або це спецефічна для кожного виробника несправність:

– 0 = Generic (загальний код для OBD2 – це цифра, а не літера);
– 1=Enhanced (код виробника);
– 2=Enhanced (код виробника);
– 3=Reserve (резерв).

Третій символ – «Підсистема». Третя цифра позначає підсистему, до якої належить несправність:

– 1 = Emission Management (Fuel or Air) (керування вприском палива, паливна система чи подача повіття);

– 2 = Injector Circuit (Fuel or Air) (ланцюг інжектора, паливна система чи подача повіттря);
– 3 = Ignition or Misfire (система запалювання);
– 4 = Emission Control (допоміжний контроль);
– 5 = Vehicle Speed & Idle Control (швидкість та холостний ход);
– 6 = Computer & Output Circuit (ЕБК та їх зовнішні ланцюги);
– 7 = Transmission (трансмісія);
– 8 = Transmission (трансмісія);
– 9=SAE Reserved (резерв);
– 10=SAE Reserved (резерв).

Четвертий та п’ятий символи – порядковий номер помилки. Спільно з іншими символами відмічають особливість несправності.

3.3.2 Налаштування взаємодії з OBD-II адаптером

Насамперед встановимо необхідний пакет для роботи із Bluetooth пристроями (рис. 3.49) [14].

```
npm install noble  // Communication for OBD-II ELM327 devices.
```

Рисунок 3.49

Внесемо зміни у vehiclesController. Функція readErrors у контролері відповідає за зчитування даних із OBD-II адаптера:

– підключається до OBD-II адаптера через функцію connectToOBD;
– викликає функцію getDiagnosticCodes для отримання кодів помилок;
– зберігає отримані дані у базу даних;
– повертає список помилок клієнту.

Код функції наведено на рисунку 3.50.

```
      export const readErrors = async (req, res, next) => {
          try {
              const { vehicleId } = req.params;

              // Підключення до OBD-II адаптера
              await connectToOBD();

              // Отримання кодів помилок
              const errors = await getDiagnosticCodes();

              // Збереження кодів у базу даних
              const vehicle = await Vehicle.findByPk(vehicleId);
              if (!vehicle) {
                  return res.status(404).json({ message: 'Автомобіль не знайдено'
});
              }

              vehicle.diagnostics_history = [...(vehicle.diagnostics_history ||
[]), {
                  timestamp: new Date().toISOString(),
                  errors: errors,
              }];
              await vehicle.save();

              res.status(200).json({ message: 'Коди помилок зчитано успішно',
errors });
          } catch (error) {
              console.error('Помилка зчитування кодів:', error);
              next(error);
          }
      };
```

Рисунок 3.50 – Функція для роботи із адаптером ELM-327

Для з’єднання та комунікації із адаптером створимо у нашому проєкті файл obdService.js, у якому на основі завчасно встановленої бібліотеки noble реалізовано логіку підключення до адаптера через бібліотеку noble. Включено динамічне визначення UUID сервісів і характеристик, що робить код гнучким для роботи з різними адаптерами.

Ключові функції, наявні у файлі obdService.js:

– connectToOBD: виконує сканування пристроїв, підключається до адаптера та знаходить потрібні характеристики;
– sendCommand: відправляє команди на адаптер і обробляє відповіді;
– getDiagnosticCodes: використовує команду «03» для отримання кодів помилок (див. рис. 3.51).

```
      export const getDiagnosticCodes = async () => {
          try {
              const response = await sendCommand('03'); // Команда для
отримання кодів помилок
              console.log('Коди помилок:', response);
              return response;
          } catch (error) {
              console.error('Помилка отримання кодів помилок:', error);
              throw error;
          }
      };
```

Рисунок 3.51 – Функція для отримання кодів помилок

3.4 Тестування та відлагодження

Для забезпечення стабільності та ефективної роботи системи було впроваджено багатоетапний процес тестування та налагодження. Цей процес охоплював перевірку всіх основних компонентів системи: маршрути API, бізнес-логіку на сервері, механізми взаємодії з базою даних, а також клієнтську частину.

На клієнтській частині проводилося мануальне тестування інтерфейсу користувача. Було перевірено:

- функціонування таблиць клієнтів, транспортних засобів та платежів, зокрема, коректність відображення даних, їх фільтрація та сортування;
- роботу кнопок для додавання, редагування та видалення записів, включаючи підтвердження дій через модальні вікна;
- адаптивність дизайну на різних пристроях, що забезпечує зручність для користувачів на мобільних і настільних платформах.

Наприклад, для функціоналу додавання клієнтів і транспортних засобів тестувалися форми, валідація даних і відповідність введених даних вимогам. Для перевірки роботи кнопки «Зчитати помилки» у таблиці транспортних засобів було додано тестовий сценарій (див. рис. 3.52), що імітує з’єднання з OBD-II адаптером і отримання діагностичних кодів. Цей функціонал перевірявся через модуль obdService.js.

На серверній частині особлива увага приділялася перевірці коректності бізнес-логіки. У маршрутах API для клієнтів, транспортних засобів і платежів (файли clientsRoute.js, vehiclesRoute.js, paymentsRoute.js) було протестовано:

- створення, редагування та видалення записів;
- повернення даних із бази в форматі, який відповідає клієнтській частині;
- ролі доступу для різних типів користувачів (Admin, Employee, Client), що забезпечує безпеку системи.

```
if (response.ok) {
      const data = await response.json();
      console.log('Коди помилок:', data.errors);
} else {
      console.error('Помилка зчитування кодів');
      }
catch (error) {
console.error('Помилка під час запиту:', error);
}
```

Рисунок 3.52 – Логування під час виконання запитів

Значна частина логіки перевірки нагадувань реалізована у файлах email.js та emailReminder.js. Нагадування зберігаються в базі даних разом із даними клієнтів. Кожного дня о 08:00 функція checkAndSendReminders перевіряє базу даних на наявність нагадувань і відправляє відповідні повідомлення на електронну пошту клієнтів.

Також були протестовані ендпоінти для отримання, редагування та видалення записів. Наприклад, у vehiclesController.js перевірявся функціонал зчитування діагностичних кодів із OBD-II адаптера, а також збереження цих даних у базу.

Для кожного маршруту API було перевірено коректність роботи з базою даних. У разі видалення клієнта (метод deleteClient у clientsController.js) перевірялося автоматичне видалення всіх пов’язаних записів, таких як транспортні засоби та платежі.

Фрагмент коду для видалення клієнта наведено на рисунку 3.53.

Для тестування використовувалися такі інструменти, як Postman [15] для тестування API, pgAdmin4 для адміністрування бази даних та створення тестових SQL запитів, React Developer Tools для моніторингу стану компонентів, Chrome DevTools для Аналізу мережевої активності та перегляду змісту сховища.

```
      const vehicles = await Vehicle.findAll({ where: { client_id: clientId
} });
      for (const vehicle of vehicles) {
          await Payment.destroy({ where: { vehicle_id: vehicle.id } });
          await vehicle.destroy();
      }
      await client.destroy();
```

Рисунок 3.53 – Логування під час виконання запитів

Комплексний підхід до тестування та налагодження дозволив створити стабільну систему з чітко визначеною логікою. Було досягнуто високу надійність роботи всіх компонентів, а також забезпечено зручність користувацького досвіду.

3.5 Розгортання програмного продукту

Ми виконаємо розгортання нашого проєкту на платформі Vercel [16], яка є популярним хмарним сервісом для автоматизації розгортання та управління вебдодатками. Проєкт включає клієнтську частину, реалізовану як статичний сайт з використанням React, і серверну частину, створену на основі Node.js (Express) із підключенням до PostgreSQL для зберігання даних.

Vercel надає ряд переваг, які роблять це рішення зручним і надійним. Платформа забезпечує простоту використання завдяки автоматичному збиранню залежностей, інтеграції з GitHub і зручній системі змінних середовища. Процеси масштабування відбуваються автоматично, що дозволяє легко додавати ресурси у разі збільшення навантаження. Крім того, Vercel пропонує безкоштовний рівень PostgreSQL із автоматичним резервним копіюванням і можливістю оновлення, що забезпечує надійне зберігання даних.

Платформа також гарантує стабільність роботи завдяки автоматичному відновленню серверних процесів у разі збоїв. Підтримка SSL (HTTPS) забезпечує безпечне з’єднання між клієнтською та серверною частинами. Інструменти моніторингу та логування дозволяють швидко виявляти й усувати проблеми, забезпечуючи безперебійну роботу додатка. Використання сучасних стеків забезпечує актуальність і підтримку більшості сучасних бібліотек.

Vercel мінімізує ручні налаштування завдяки автоматизації деплою, що знижує ризик помилок. Змінні середовища надійно зберігаються на сервері,

виключаючи можливість втрати конфігурацій. Платформа підтримує інтеграцію клієнтської частини з бекендом і дозволяє масштабувати додаток залежно від навантаження. Надійна база даних PostgreSQL забезпечує стабільне збереження даних і їх швидкий доступ, а резервне копіювання запобігає втраті інформації.

Таким чином, розгортання на Vercel дає можливість отримати глобально доступний додаток із надійним сервером і інтегрованою клієнтською частиною. Це рішення забезпечує стабільну роботу, автоматизацію процесів і простоту управління додатком, роблячи його готовим до використання у будь-якій точці світу.

Поточна версія проєкту знаходиться за адресою https://diploma-gamma.vercel.app/. Зовнішній вигляд головної сторінки зображено на рисунку 3.54.

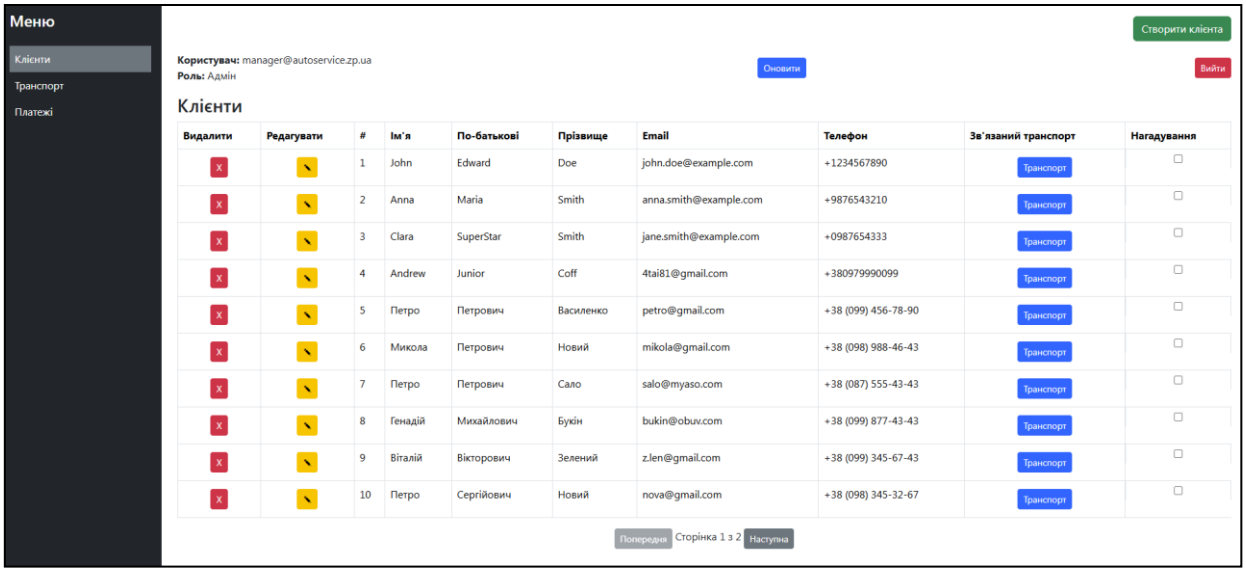


Рисунок 3.54 – Інтерфейс головної сторінки

ВИСНОВКИ

У результаті виконання кваліфікаційної роботи було проведено дослідження та практичну реалізацію вебзастосунку для автоматизації управління клієнтами, транспортними засобами та іншими ресурсами з використанням сучасного технологічного стеку на базі PostgreSQL. У процесі роботи було здійснено аналіз існуючих рішень на ринку та обґрунтовано доцільність розробки спеціалізованого програмного забезпечення для покращення ефективності бізнес-процесів.

Дослідження можливостей технологій, застосованих у проєкті, продемонструвало їх ефективність для створення високонавантажених вебзастосунків. Використання PostgreSQL як бази даних забезпечило високу продуктивність, надійність і гнучкість у роботі з великими обсягами структурованих даних. Комбінація Express.js та Node.js дозволила створити стабільний серверний компонент із широкими можливостями для інтеграції, а React забезпечив реалізацію зручного й сучасного користувацького інтерфейсу.

В рамках роботи було розроблено архітектуру системи, яка забезпечує ефективну організацію даних, їх обробку та оптимізований пошук за заданими критеріями. Реалізована серверна частина на базі Node.js та Express.js демонструє надійність у роботі з базою даних PostgreSQL та забезпечує високу швидкість обробки запитів. Клієнтська частина, створена з використанням React, надає користувачам інтуїтивно зрозумілий інтерфейс з інтерактивними компонентами для управління даними.

Практична реалізація проєкту підтвердила, що використання сучасних технологій веброзробки забезпечує створення масштабованих і продуктивних рішень. PostgreSQL як база даних забезпечила стабільність зберігання та швидкий доступ до даних, а JavaScript як єдина мова програмування для серверної та клієнтської частин спростила розробку. Доступність численних

бібліотек і активна підтримка спільноти сприяли швидкому та ефективному виконанню роботи.

Розроблений вебзастосунок демонструє можливість автоматизації ключових процесів управління ресурсами та клієнтами. Гнучка архітектура системи дозволяє легко інтегрувати нові функціональні модулі, що створює основу для подальшого розвитку та адаптації до потреб користувачів. Таким чином, реалізоване рішення відповідає сучасним стандартам і забезпечує можливості масштабування для майбутніх потреб.

ПЕРЕЛІК ПОСИЛАНЬ

1. RemOnline – зручна програма для бізнесу. URL: https://remonline.ua/blog/secrets-autoservice-staff/ (дата звернення: 16.09.2024).

2. FixIQ Pro – платформа для керування транспортним бізнесом. URL: https://fixiq.pro/uk/ (дата звернення: 16.09.2024).

3. Автоматизація бізнесу в Україні – питання та відповіді. URL: https://expresssoft.com.ua/uk/programa-dlja-avtoservisu-sto/ (дата звернення: 16.09.2024).

4. Sequelize is a promise-based Node.js ORM tool for Postgres, MySQL, MariaDB, SQLite, Microsoft SQL Server. URL: https://sequelize.org/docs/v6/ (дата звернення 17.09.2024).

5. React – The library for web and native user interfaces. URL: https://react.dev/ (дата звернення: 21.10.2024).

6. React Router Official Documentation. URL: https://reactrouter.com (дата звернення: 21.10.2024).

7. PostgreSQL is an advanced open-source database system with 35+ years of development. URL: https://www.postgresql.org/ (дата звернення: 24.10.2024).

8. Node.js – Run JavaScript Everywhere. URL: https://nodejs.org (дата звернення: 10.11.2024).

9. JWTs are a secure method for representing claims between parties. URL: https://codex.so/jwt (дата звернення: 12.11.2024).

10. Специфікація CAN 2.0 URL: https://web.archive.org/web/20100922201217/http://www.semiconductors.bosch.de/pdf/can2spec.pdf (дата звернення: 21.11.2024).

11. Texas Instruments. What Do CAN Bus Signals Look like? URL: https://www.ti.com/document-viewer/lit/html/SSZTCN3#:~:text=As%20stated%20in%20my%20first,low%20(CANL)%20bus%20wires (дата звернення: 21.11.2024).

12. OBD to RS232 Interpreter. URL: https://www.elmelectronics.com/DSheets/ELM327DSH.pdf (дата звернення: 23.11.2024).

13. Електронний довідник автосервісу. URL: https://avtoad.com.ua/ru/base/kak-rassifrovat-kod-obd2-obd-ii (дата звернення: 26.11.2024).

14. Noble. A node.js BLE (Bluetooth low energy) module. URL: https://www.npmjs.com/package/noble/v/0.2.1 (дата звернення: 24.11.2024).

15. Postman: The World’s Leading API Platform. URL: https://www.postman.com (дата звернення: 20.12.2024).

16. Vercel is a developer cloud to build and deploy web applications. URL: https://vercel.com/docs (дата звернення: 20.12.2024).