

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій**

ПОГОДЖЕНО
Декан факультету

ДОПУСКАЄТЬСЯ ДО ЗАХИСТУ
**Завідувач кафедри інформаційних
систем і технологій**

_____ **Ігор БОЛБОТ**
(підпис)

_____ **Михайло ШВИДЕНКО**
(підпис)

«____» _____ 2026 р.

«____» _____ 2026 р.

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

**на тему: «Веб-орієнтована автоматизована система керування
автомайстернею»**

Спеціальність «122 Комп'ютерні науки»

Освітньо-професійна програма «Комп'ютерні науки»

Гарант освітньої програми
д.е.н., професор

(підпис)

Роман РУДЕНЬСКИЙ

**Керівник бакалаврської
кваліфікаційної роботи**

(підпис)

Максим МОКРІЄВ

Виконав

(підпис)

Ілля СВИРИДОВСЬКИЙ

Київ-2026

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ І
ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ
Факультет інформаційних технологій**

**ЗАТВЕРДЖУЮ
Завідувач кафедри
інформаційних систем і технологій**

К.Т.Н., доцент

Швиденко М.З.

(підпис)

«____» _____ 2026 р.

**ЗАВДАННЯ
ДО ВИКОНАННЯ БАКАЛАВРСЬКОЇ КВАЛІФІКАЦІЙНОЇ РОБОТИ
ЗДОБУВАЧУ**

Свиридовському Іллі Олеговичу

(прізвище, ім'я, по батькові)

Спеціальність «122 Комп'ютерні науки»
Освітньо-професійна програма «Комп'ютерні науки» Тема бакалаврської
кваліфікаційної роботи

« Веб-орієнтована автоматизована система керування автомайстернею »
затверджена наказом від «6» січня 2026 р. № 46 «С» Термін подання завершеної
роботи на кафедру «25» травня 2026 р.

Вихідні дані до бакалаврської кваліфікаційної роботи: спільний вигул, оперативний обмін
інформацією, геопросторові сервіси.

Перелік питань, що підлягають дослідженню:

1. Актуальність автоматизації обліку на СТО та аналіз існуючих аналогів.
2. Проектування архітектури системи та реляційної бази даних PostgreSQL.
3. Програмна реалізація веб-клієнта та асинхронного серверного API.
4. Розробка модулів складського обліку та динамічної калькуляції замовлень.
5. Реалізація аналітичного дашборду та контейнеризація системи в Docker.
6. Функціональне тестування інтерфейсу додатка та верифікація бекенду.

Дата видачі завдання «6» січня 2026 р.

**Керівник бакалаврської
кваліфікаційної роботи**

Максим МОКРІЄВ

(підпис)

Завдання взяв до виконання

Ілля СВИРИДОВСЬКИЙ

(підпис)

ЗМІСТ

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ	4
ВСТУП	5
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОЄКТУВАННЯ СИСТЕМИ	8
1.1 Постановка задачі автоматизації процесів СТО	8
1.2 Огляд існуючих рішень	9
1.3 Моделювання предметної області	13
2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ПРОЄКТУ	18
2.1 Вибір системи управління базами даних	18
2.2 Логічна модель бази даних	23
2.3 Фізична реалізація інформаційної бази	27
3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ	34
3.1 Організаційна структура клієнт серверного додатку	35
3.2 Алгоритмізація та програмування бекенд модулів	43
3.3 Реалізація користувацького інтерфейсу та взаємодія з API	48
4 ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ	52
4.1 Архітектура розгортання системи	52
4.2 Тестування інтерфейсу та функціоналу	58
4.3 Вимоги до апаратного та програмного забезпечення	66
ВИСНОВКИ	70
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ	73
ДОДАТОК А	75
ДОДАТОК Б	79
ДОДАТОК В	83

ПЕРЕЛІК УМОВНИХ СКОРОЧЕНЬ

БД – база даних **ОЗП** – оперативний запам'ятовувальний пристрій **ОС** – операційна система **СТО** – станція технічного обслуговування **СУБД** – система управління базами даних

API (Application Programming Interface) – програмний інтерфейс додатку

CSV (Comma-Separated Values) – текстовий формат для збереження та експорту табличних даних

ER (Entity-Relationship) – концептуальна модель даних «сутність-зв'язок»

ERP (Enterprise Resource Planning) – система планування ресурсів підприємства

IDE (Integrated Development Environment) – інтегроване середовище розробки

JSON (JavaScript Object Notation) – текстовий формат обміну даними

JWT (JSON Web Token) – криптографічний стандарт для створення токенів доступу та авторизації

REST (Representational State Transfer) – архітектурний стиль для проектування мережових додатків

SQL (Structured Query Language) – мова структурованих запитів до реляційних баз даних

UI/UX (User Interface / User Experience) – користувацький інтерфейс та досвід взаємодії користувача з програмою

UML (Unified Modeling Language) – уніфікована мова графічного моделювання об'єктно-орієнтованих систем

VIN (Vehicle Identification Number) – унікальний сімнадцятизначний ідентифікаційний номер транспортного засобу

ВСТУП

Сьогодні світ переживає справжній бум цифровізації і це стосується абсолютно кожної сфери бізнесу. Клієнти звикли до того що будь-яку послугу можна замовити через смартфон отримати сповіщення про статус виконання та прозору квитанцію. Сфера обслуговування автомобілів також не є винятком. Сучасний автомобіль це складна система яка вимагає такого ж системного підходу до свого ремонту. Відповідно вимоги до станцій технічного обслуговування зростають з кожним днем і щоб залишатися конкурентоспроможними власникам автосервісів доводиться шукати нові шляхи оптимізації своєї роботи.

На жаль, багато невеликих та середніх СТО досі ведуть свій облік старими методами використовуючи паперові журнали або в кращому випадку прості електронні таблиці. Такий підхід має величезну кількість недоліків. По перше це людський фактор коли записи губляться забуваються деталі замовлення або неправильно фіксуються ціни на запчастини. По друге відсутність єдиної бази даних робить неможливим швидкий пошук історії ремонтів конкретного клієнта. Власник такого бізнесу фактично керує ним наосліп не розуміючи реального стану справ на складі не знаючи точного прибутку та не маючи можливості оцінити ефективність роботи кожного окремого механіка. Це призводить до втрати клієнтів зниження прибутку та загальної стагнації підприємства

1. Впровадження спеціалізованих інформаційних систем класу ERP тобто систем планування ресурсів підприємства дозволяє вирішити більшість цих проблем

2. Автоматизація бізнес-процесів дає змогу створити єдиний інформаційний простір де кожен співробітник від адміністратора до механіка має доступ до потрібної йому інформації в режимі реального часу. Це значно пришвидшує обслуговування мінімізує помилки та робить фінансові потоки абсолютно прозорими. Проте аналіз ринку показує що існуючі готові рішення часто є занадто складними для впровадження вимагають дорогого ліцензування

або містять купу зайвого функціоналу який ніколи не буде використовуватися на типовому СТО.

Також варто відзначити що сучасні реалії вимагають від таких систем не лише функціональності але й високого рівня безпеки та мобільності. Керівник повинен мати змогу переглянути статистику з будь-якого пристрою а дані клієнтів мають бути надійно захищені від несанкціонованого доступу. Тому використання хмарних баз даних та сучасних протоколів авторизації стає стандартом для подібних проектів.

Зважаючи на вищевикладене тема розробки власної інформаційної системи для управління процесами станції технічного обслуговування є надзвичайно актуальною та своєчасною. Метою цієї дипломної роботи є проектування розробка та впровадження сучасної клієнт-серверної інформаційної системи яка б повністю закривала потреби типового автосервісу в автоматизації обліку замовлень автопарку та складських запасів.

Для досягнення поставленої мети в роботі вирішується цілий комплекс взаємопов'язаних завдань. В першу чергу необхідно провести глибокий аналіз предметної області дослідити специфіку роботи СТО та сформулювати чіткі вимоги до майбутньої системи. Далі потрібно виконати огляд існуючих аналогів та обґрунтувати вибір оптимального стеку технологій для розробки. Важливим етапом є проектування логічної та фізичної структури реляційної бази даних яка повинна забезпечувати надійне зберігання та швидку вибірку інформації. Після цього йде безпосередньо етап програмування який ЧВЧ включає розробку серверної частини тобто REST API для обробки бізнес-логіки та створення зручного інтерактивного користувацького інтерфейсу з використанням сучасних веб-технологій. На завершальному етапі необхідно провести тестування розробленого програмного забезпечення та підготувати рекомендації щодо його розгортання в реальних умовах експлуатації.

Об'єктом дослідження в даній дипломній роботі виступають процеси управління та обліку на станції технічного обслуговування автомобілів.

Предметом дослідження є методи моделювання бази даних алгоритми обробки інформації та програмні засоби створення веб-орієнтованих інформаційних систем. Практичне значення отриманих мною результатів полягає у створенні повністю робочого та економічно вигідного програмного додатка який можна без жодних проблем розгорнути на базі будь якої реальної автомайстерні. Для успішного вирішення всіх цих складних інженерних завдань я обрав послідовний науковий підхід що дозволив мені повністю розкрити структуру дослідження у чотирьох взаємопов'язаних розділах цієї дипломної роботи. У першому розділі проводиться глибокий системний аналіз предметної області, досліджуються повсякденні бізнес-процеси станції технічного обслуговування та здійснюється порівняльний огляд існуючих на ринку програмних рішень. На основі результатів цього аналізу формуються чіткі функціональні вимоги до розроблюваного продукту та обґрунтовується вибір сучасного технологічного інструментарію. Другий розділ повністю присвячений проектуванню архітектури даних, де покроково описується створення логічної та фізичної моделей реляційної бази. Особлива увага в цій частині приділяється практичній інтеграції хмарного середовища на платформі Supabase та налаштуванню суворих внутрішньобазових обмежень цілісності. У третьому розділі розкриваються всі технічні аспекти алгоритмізації та безпосереднього програмування серверної бізнес-логіки з використанням мікросервісного підходу. Крім того, тут наводиться детальний опис розробки адаптивного клієнтського інтерфейсу та механізмів безпечної маршрутизації через захищений REST API. Четвертий розділ містить вичерпну інформацію про вибір оптимальної стратегії розгортання готової системи за допомогою технології контейнеризації Docker. Також у цій фінальній частині документуються результати проведеного функціонального тестування системи та формуються остаточні мінімальні вимоги до апаратного забезпечення.

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПРОЄКТУВАННЯ СИСТЕМИ

1.1 Постановка задачі автоматизації процесів СТО

Робота будь якої станції технічного обслуговування складається з великої кількості щоденних рутинних операцій. Якщо розглядати класичний автосервіс то можна виділити цілий ряд процесів які зазвичай виконуються вручну і займають занадто багато часу. Тому перед нашою системою ставиться задача автоматизувати наступні ключові напрямки діяльності підприємства

Першим важливим пунктом є управління базою клієнтів та їхніх автомобілів. Адміністратор повинен мати можливість в пару кліків знайти всю історію ремонтів конкретної машини побачити які запчастини встановлювалися раніше та які рекомендації давав майстер. Це підвищує лояльність клієнтів бо вони бачать професійний підхід.

Другим пунктом виступає автоматизація складського обліку. На складі СТО завжди знаходяться сотні найменувань деталей від фільтрів до моторного масла. Наша система повинна чітко контролювати залишки щоб при додаванні деталі в ремонтний наряд вона автоматично списувалася зі складу а комірник завжди знав що саме потрібно дозамовити у постачальників.

Третій пункт це контроль виконання самих ремонтних робіт. Кожне замовлення має проходити певні стадії від прийняття авто в роботу до успішного закриття та видачі клієнту. Система має фіксувати хто саме з механіків виконував роботу які конкретно послуги були надані та яка їхня вартість.

Четвертий пункт полягає у фінансовому обліку та аналітиці. Власник бізнесу має бачити на окремому дашборді всі ключові показники. Це загальний прибуток за день чи місяць кількість відкритих та закритих замовлень а також загальна сума витрат. Всі ці розрахунки програма має робити самостійно без використання додаткових калькуляторів чи таблиць.

Таким чином головна задача нашої розробки зводиться до створення єдиного інформаційного середовища яке об'єднає роботу всіх співробітників СТО та зробить бізнес максимально прозорим і керованим.

1.2 Огляд існуючих рішень

На сучасному ринку програмного забезпечення представлено багато систем для автоматизації бізнесу але далеко не всі вони підходять для невеликих та середніх автосервісів. Щоб обґрунтувати необхідність створення власного продукту ми провели детальний аналіз існуючих рішень та розділили їх на декілька основних категорій.

Перша категорія це великі корпоративні ERP системи на кшталт 1С Підприємство або SAP. Вони мають величезний функціонал і дозволяють налаштувати будь який процес. Але для звичайного СТО вони є занадто громіздкими та складними у використанні. Впровадження таких систем вимагає залучення окремих програмістів для налаштування що тягне за собою колосальні фінансові витрати. Персоналу доводиться проходити довге навчання щоб просто навчитися створювати базовий ремонтний наряд.

Друга категорія це локальні десктопні програми які встановлюються безпосередньо на комп'ютер адміністратора. Головний їхній недолік це відсутність мобільності та прив'язка до конкретного робочого місця. Керівник не може зайти в таку програму з дому чи з телефону щоб перевірити як ідуть справи на сервісі. Також існує величезний ризик втрати всіх даних у випадку поломки жорсткого диску адже резервне копіювання в таких програмах часто робиться вручну і про нього просто забувають.

Третя категорія це сучасні хмарні SaaS рішення. Вони працюють через браузер і виглядають досить привабливо але працюють за моделлю постійної підписки. Власник СТО змушений щомісяця платити абонентську плату яка з часом тільки зростає. Якщо перестати платити то доступ до власної бази клієнтів просто блокується. Крім того весь масив комерційної інформації зберігається на

складах. Але з часом завдяки стрімкому розвитку комп'ютерної техніки та інтернету ці системи еволюціонували і стали доступними для малого та середнього бізнесу до якого якраз і належать звичайні станції технічного обслуговування. Головна теоретична концепція будь якої сучасної ERP системи полягає в тому щоб створити абсолютно єдиний і неподільний інформаційний простір для всіх працівників компанії. Це означає що коли мій механік бере зі складу звичайний масляний фільтр інформація про це має миттєво і без жодних затримок відобразитися в бухгалтерії у менеджера з закупівель та на головному екрані власника бізнесу.

Досліджуючи теоретичну базу я також звернув велику увагу на архітектурні підходи до створення таких систем. Сьогодні в світі програмування існує два глобальних підходи це створення монолітних програм та мікросервісна архітектура. Монолітні додатки де весь код бази даних інтерфейс та логіка знаходяться в одному великому файлі були дуже популярні раніше але зараз вони вважаються абсолютно застарілими бо їх дуже важко оновлювати і вони часто ламаються при найменших змінах. Тому я для свого проекту свідомо обрав сучасну розподілену клієнт серверну архітектуру яка базується на використанні програмних інтерфейсів або скорочено API. Суть цієї технології полягає в тому що програма з якою взаємодіє користувач на екрані свого монітора і програма яка реально виконує всі математичні розрахунки це дві абсолютно різні і незалежні сутності які просто спілкуються між собою по мережі інтернет пересилаючи маленькі текстові повідомлення. Такий підхід дає мені просто неймовірну свободу дій бо я можу хоч кожного дня повністю змінювати дизайн кнопок чи кольори таблиць абсолютно не ризикуючи при цьому зламати складну фінансову логіку яка надійно захована на моєму бекенді.

Теоретичні основи моделювання систем

Для того щоб правильно спроектувати свою систему і не зробити фатальних архітектурних помилок ще до написання програмного коду я вирішив використати стандартизовану уніфіковану мову моделювання яка відома в усьому світі під аббревіатурою UML. Це спеціальна графічна мова яка дозволяє

розробникам дуже детально описувати візуалізувати та документувати абсолютно всі процеси які будуть відбуватися в майбутній програмі. Використання UML моделей є обов'язковим стандартом у професійній інженерії програмного забезпечення бо це дозволяє розбити величезну і складну задачу на маленькі зрозумілі блоки.

У своїй роботі я використав кілька основних типів таких діаграм і першою з них є діаграма прецедентів. У теорії системного аналізу ця діаграма створюється виключно для того щоб показати які саме функції система буде надавати своїм кінцевим користувачам і вона принципово не показує як саме ці функції будуть працювати всередині. Головними елементами такої діаграми є так звані актори які зображуються у вигляді фігурок людей і символізують конкретні ролі користувачів наприклад адміністратора чи звичайного механіка. Кожен такий актор з'єднується спеціальними лініями з овалами які називаються прецедентами і позначають конкретні дії такі як створення ремонтного наряду чи додавання нової запчастини на склад. В теорії UML існують різні типи зв'язків між цими овалами наприклад відношення включення коли один процес обов'язково викликає інший процес або відношення розширення коли дія виконується лише за певних специфічних умов але у своєму проекті я намагався використовувати максимально прості прямі зв'язки щоб не перевантажувати логіку роботи автосервісу. Для більш глибокого та об'єктивного розуміння поточного стану ринку програмного забезпечення для автосервісів, було проведено детальний порівняльний аналіз існуючих рішень. На сьогоднішній день більшість СТО використовують або застарілі монолітні системи (наприклад, конфігурації на базі «1С:Підприємство»), або сучасні хмарні SaaS-платформи (наприклад, RemOnline), або ж взагалі ведуть облік у звичайних електронних таблицях MS Excel.

Другою надзвичайно важливою моделлю яку я використав у своєму проєкті є діаграма послідовності. Якщо попередня схема просто показувала що вміє робити система то діаграма послідовності показує як саме об'єкти системи взаємодіють між собою в часі крок за кроком. Згідно з офіційними стандартами UML на такій діаграмі всі основні учасники процесу розташовуються зверху у вигляді прямокутників від яких вниз ідуть довгі пунктирні лінії які називаються лініями життя. Ці лінії показують час існування об'єкта під час виконання конкретної операції. Коли один об'єкт наприклад вікно браузера хоче передати дані іншому об'єкту наприклад моєму серверу він відправляє спеціальне повідомлення яке малюється у вигляді суцільної лінії зі стрілочкою зліва направо. Над кожною такою стрілочкою обов'язково пишеться назва методу чи передані дані наприклад мій зашифрований токен безпеки або сума замовлення. Також на лініях життя малюються вузькі вертикальні прямокутники які називаються фокусом управління і вони дуже чітко показують в який саме момент часу певний модуль системи активно працює обробляючи інформацію поки всі інші модулі просто чекають на свою чергу. Завдяки побудові такої складної та деталізованої діаграми я зміг повністю візуалізувати алгоритм створення ремонтного наряду зрозумівши в який саме момент мій бекенд має звертатися до хмарної бази даних і коли саме потрібно повертати успішну відповідь на екран користувача.

1.3 Моделювання предметної області

Для того щоб наша система працювала правильно її потрібно спочатку спроектувати на папері. Найкращим інструментом для цього є уніфікована мова моделювання або скорочено UML. Вона дозволяє візуалізувати всі процеси ще до написання першого рядка коду. В нашому випадку ми використали дві основні діаграми це діаграма прецедентів та діаграма послідовності.

Діаграма прецедентів або як її ще називають use case діаграма показує взаємодію користувачів із самою системою. У нашій програмі є два основні типи користувачів це адміністратор та механік. Адміністратор має найвищі права доступу. Він може додавати нові автомобілі в базу реєструвати клієнтів та переглядати фінансову статистику на головному дашборді. Механік своєю

чергою працює безпосередньо з ремонтними нарядами. Його головні задачі це відкрити замовлення додати туди потрібні запчастини зі складу та закрити наряд після завершення ремонту.

Система автоматично розмежовує їхні права щоб кожен бачив лише свій функціонал. Після визначення ролей та їхніх базових можливостей за допомогою прецедентів, наступним кроком стала розробка діаграми послідовності. Ця діаграма детально ілюструє хронологічний порядок обміну повідомленнями між клієнтським веб-інтерфейсом, сервером та базою даних під час виконання найважливіших транзакцій, наприклад, при додаванні деталі в ремонтний наряд. Завдяки такому візуальному проєктуванню вдалося чітко продумати алгоритм асинхронних запитів та механізми перехоплення помилок ще на етапі теоретичного планування. Таким чином, комплексне UML-моделювання дозволило виявити і усунути можливі логічні конфлікти у бізнес-процесах, створивши надійний фундамент для подальшої програмної реалізації системи. Окрім моделювання поведінкових аспектів системи, не менш важливим етапом стало детальне опрацювання інформаційної структури проєкту. Усі ключові сутності, виявлені під час побудови UML-діаграм, зокрема профілі працівників, транспортні засоби, складські одиниці та самі ремонтні наряди, лягли в основу концептуальної моделі майбутньої бази даних. Такий підхід забезпечив максимально тісний логічний зв'язок між реальними операційними потребами автосервісу та тим, як інформація буде оброблятися на сервері. У результаті проведеного моделювання предметної області було остаточно сформовано вичерпне технічне бачення розроблюваного програмного продукту. Отримані аналітичні результати та візуальні схеми стали надійними вихідними даними для переходу до наступного етапу роботи, який безпосередньо присвячений проєктуванню реляційної структури даних.

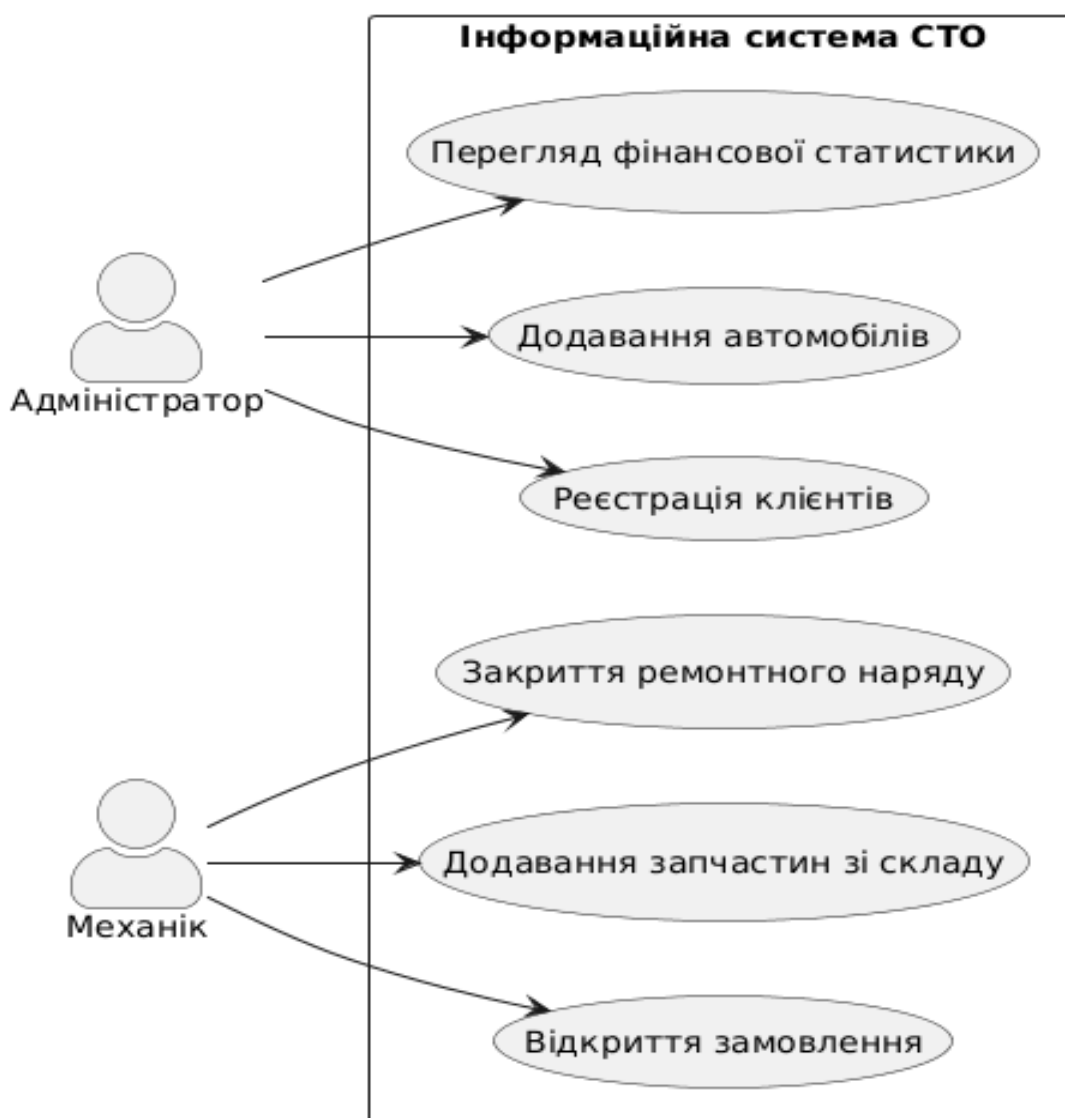


Рисунок 1.2 Діаграма прецедентів взаємодії користувачів із системою СТО

Друга важлива схема це діаграма послідовності. Вона показує як саме передаються дані всередині системи коли користувач виконує певну дію. Візьмемо для прикладу процес створення нового замовлення. Спочатку адміністратор натискає кнопку на фронтенді і вводить дані. Далі наш клієнтський додаток формує запит та відправляє його на сервер через спеціальний API інтерфейс. Сервер отримує цей запит перевіряє чи має користувач права на цю дію за допомогою токена безпеки і якщо все добре звертається до бази даних. База даних зберігає інформацію та повертає серверу унікальний номер нового замовлення. Після цього сервер відправляє успішну відповідь на фронтенд і користувач бачить на екрані що замовлення успішно створено.

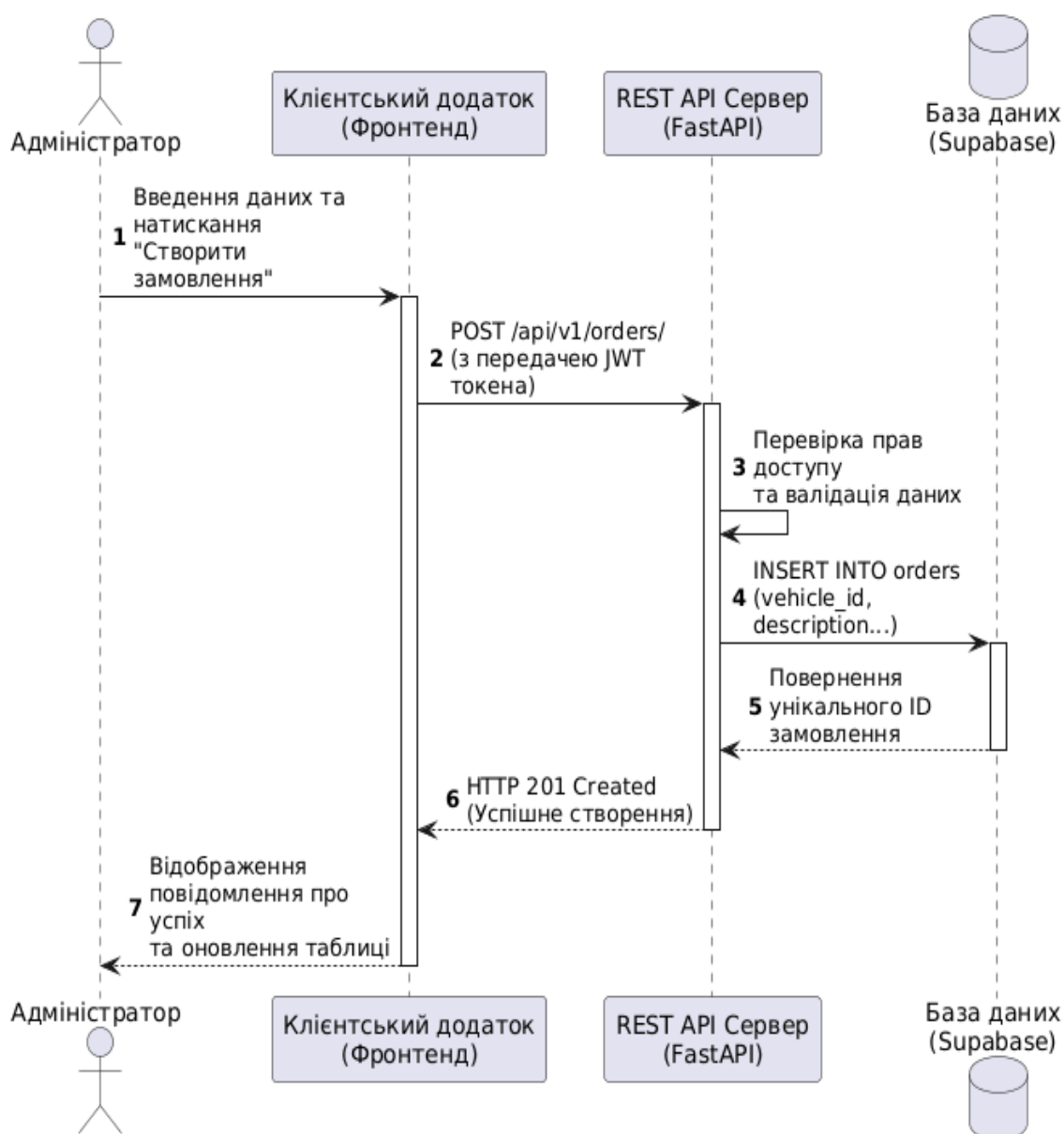


Рисунок 1.3 Діаграма послідовності створення нового ремонтного наряду

Таке детальне моделювання дозволило нам уникнути багатьох логічних помилок ще на етапі планування. Ми чітко зрозуміли які саме таблиці нам потрібні в базі даних та які функції треба реалізувати на бекенді. Це значно зекономило час під час безпосереднього написання програмного коду та допомогло створити правильну архітектуру нашого додатку. Як можна побачити на цій розробленій схемі я чітко розділив свою інформаційну систему на три логічно незалежні рівні щоб забезпечити її максимальну надійність та гнучкість під час щоденної роботи. Клієнтський рівень у браузері відповідає виключно за

зручний візуальний інтерфейс і відправляє сформовані команди до центрального сервера FastAPI який бере на себе всю найважчу роботу з обробки бізнес логіки та безпечної перевірки токенів доступу.



Рисунок 1.4 – Загальна клієнт-серверна архітектура розробленої системи

Як можна дуже чітко побачити на цій схемі весь інформаційний потік починається безпосередньо з першого рівня тобто з браузера кінцевого користувача де працює мій написаний інтерфейс. Коли менеджер або механік виконує будь яку дію на екрані свого пристрою цей клієнтський додаток миттєво формує асинхронний запит і відправляє його по мережі інтернет на мій центральний серверний рівень.

2 ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ПРОЄКТУ

2.1 Вибір системи управління базами даних

Основою будь якої сучасної інформаційної системи є надійна та швидкодіюча база даних. Для нашого проекту управління станцією технічного обслуговування потрібно було вибрати таку систему яка б могла ефективно обробляти великі масиви інформації не вимагала постійного ручного налаштування та забезпечувала найвищий рівень безпеки для комерційних даних. Після ретельного аналізу різних доступних варіантів на ринку ми зупинили свій вибір на реляційній системі керування базами даних PostgreSQL. Ця база даних вже багато років вважається однією з найбільш надійних та стабільних у світі програмування і вона просто чудово справляється зі складними зв'язками між різними таблицями що є критично важливим для нашого проекту адже замовлення на ремонт постійно пов'язані з конкретними клієнтами автомобілями та складськими запасами.

На етапі проєктування архітектури даних постало критично важливе питання вибору типу системи керування базами даних (СКБД). Сучасний ринок пропонує два глобальних підходи: нереляційні (NoSQL) бази даних, такі як MongoDB, та класичні реляційні (SQL) системи, наприклад PostgreSQL.

Незважаючи на високу популярність NoSQL рішень завдяки їхній гнучкості при роботі з JSON-документами, від використання MongoDB у цьому проєкті було свідомо вирішено відмовитися. Головна причина такого рішення полягає у специфіці предметної області: автомайстерня постійно оперує фінансовими транзакціями, залишками на складі та статусами ремонтних нарядів. Для таких операцій критично необхідною є підтримка властивостей ACID (Атомарність, Узгодженість, Ізольованість, Довговічність). Реляційна структура PostgreSQL ідеально підходить для фінансового обліку, оскільки вона дозволяє жорстко зв'язати таблиці (наприклад, замовлення із запчастинами) через зовнішні ключі (Foreign Keys) і гарантує, що при видаленні або зміні запису база даних не опиниться в неузгодженому стані.

Зробивши вибір на користь PostgreSQL, наступним кроком було визначення формату розгортання бази даних. Було проаналізовано класичний підхід із налаштуванням власного фізичного або віртуального сервера (VPS) та сучасну концепцію BaaS (Backend-as-a-Service) на прикладі платформи Supabase. Порівняння цих підходів наведено у таблиці 2.1.

Таблиця 2.1

Порівняння підходів до розгортання бази даних

Характеристика	Традиційний фізичний/ віртуальний сервер (VPS)	Хмарна платформа Supabase (BaaS)
Встановлення та конфігурація	Вимагає ручного встановлення Linux, PostgreSQL, налаштування портів та брандмауера	БД розгортається автоматично за кілька секунд в один клік у веб-інтерфейсі
Резервне копіювання (Backups)	Необхідно самостійно писати cron-скрипти та купувати місце для зберігання дамів	Автоматичне щоденне резервне копіювання (Point-in-Time Recovery)
Безпека та авторизація	Вимагає розробки власної системи керування ролями з нуля	Вбудований модуль Supabase Auth та підтримка Row Level Security (RLS) "з коробки"

Доступність (Uptime)	Залежить від компетенції адміністратора та стабільності електроживлення	Гарантований SLA 99.9% від хмарного провайдера (AWS)
Фінансові витрати на старті	Покупка "заліза" або щомісячна оренда потужного VPS-сервера	Наявність безкоштовного тарифу (Free Tier), якого повністю достатньо для однієї СТО

Теоретичні основи побудови реляційних баз даних

Приступаючи до етапу інформаційного забезпечення свого проекту я вирішив дуже детально вивчити саму теорію побудови баз даних бо це є абсолютним фундаментом будь якої сучасної програми. Історично склалося так що для збереження структурованої інформації в усьому світі найчастіше використовуються саме реляційні бази даних. Головна концепція такої бази полягає в тому що абсолютно всі дані зберігаються у вигляді звичайних двовимірних таблиць які логічно складаються з колонок і рядків. Кожна колонка має свій чітко визначений тип даних наприклад це може бути тільки текст або тільки цілі числа що дуже сильно зменшує ймовірність виникнення помилки при збереженні інформації. А кожен окремий рядок у такій таблиці це по суті один конкретний запис наприклад детальна інформація про одного клієнта або про одну деталь яка лежить на складі.

Для того щоб ці таблиці могли правильно і надійно взаємодіяти між собою в теорії реляційних баз даних постійно використовується поняття ключів. Кожна таблиця обов'язково повинна мати свій первинний ключ це такий унікальний ідентифікатор який ніколи не повторюється і дозволяє системі миттєво знайти потрібний рядок серед мільйонів інших записів на жорсткому диску. Коли мені

потрібно зв'язати наприклад таблицю замовлень з таблицею клієнтських автомобілів я використовую зовнішній ключ тобто я просто записую ідентифікатор конкретної машини у відповідну колонку в таблиці замовлень. Такий науковий підхід гарантує абсолютну цілісність комерційної інформації і фізично не дозволяє випадково видалити клієнта якщо у нього ще є відкриті ремонтні наряди на сервісі. Для забезпечення повної прозорості фізичної структури бази даних було розроблено детальний словник даних (Data Dictionary). Він описує типи полів, їхні обмеження та призначення для кожної створеної таблиці в середовищі Supabase. Цей словник відіграє роль головного технічного стандарту, який гарантує правильне розуміння архітектури проєкту на етапі його майбутнього масштабування або можливої передачі коду іншим розробникам. Крім базових типів полів, у ньому жорстко задокументовано всі внутрішньобазові обмеження цілісності, такі як перевірка унікальності значень, заборона порожніх полів (NOT NULL) та каскадні правила оновлення. Завдяки такому ретельному теоретичному опису структури, процес фізичної генерації таблиць у хмарному середовищі PostgreSQL пройшов абсолютно безпомилково та без втрати початкової логіки.

Таблиця 2.2

Структура таблиці «inventory» (Складські запаси)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис призначення
id	integer	PRIMARY KEY, GENERATED ALWAYS AS IDENTITY	Унікальний ідентифікатор деталі
name	varchar(150)	NOT NULL	Комерційна назва запчастини

sku	varchar(50)	NOT NULL, UNIQUE	Складський артикул виробника
quantity	integer	NOT NULL, DEFAULT 0, CHECK (quantity >= 0)	Поточний фізичний залишок
price	decimal(10,2)	NOT NULL, CHECK (price > 0)	Вартість одиниці товару
is_archived	boolean	NOT NULL, DEFAULT FALSE	Статус вилучення з каталогу

Для спілкування з такими базами даних використовується спеціальна мова структурованих запитів яка називається SQL. Ця мова є абсолютним міжнародним стандартом і вона дозволяє розробникам писати дуже складні команди для вибірки сортування та математичної обробки інформації прямо на стороні сервера бази даних не завантажуючи при цьому клієнтський комп'ютер. Аналізуючи існуючі системи управління реляційними базами я прийшов до твердого висновку що найкращим вибором для мого комерційного проекту буде саме PostgreSQL. Це надзвичайно потужна об'єктно реляційна система яка має відкритий вихідний код і славиться своєю просто фантастичною надійністю при дуже високих навантаженнях. Вона ідеально підтримує складні транзакції і гарантує що навіть при раптовому відключенні світла на сервері автосервісу жоден байт фінансової інформації не буде втрачено чи пошкоджено.

Але замість того щоб йти традиційним шляхом і розгортати PostgreSQL на власному локальному комп'ютері або орендувати голий сервер ми вирішили використати сучасну хмарну платформу під назвою Supabase. Це інноваційне рішення дозволило нам повністю уникнути типових проблем пов'язаних з

підтримкою заліза та складним розгортанням локального сервера бази даних. Платформа Supabase надає вже повністю готову і налаштовану базу даних у хмарі з вбудованим API та готовою системою авторизації користувачів. Завдяки використанню саме такого підходу ми не ризикуємо втратити цінні комерційні дані у випадку раптової поломки робочого комп'ютера на СТО оскільки вся інформація надійно зберігається на віддалених захищених серверах з автоматичним резервним копіюванням. Крім того саме хмарний підхід дозволяє власнику бізнесу мати постійний та безперебійний доступ до фінансової статистики з будь якої точки світу де є інтернет.

2.2 Логічна модель бази даних

Наступним і дуже важливим кроком після вибору програмної платформи стало детальне проектування структури самої бази даних. Щоб програма працювала без збоїв і швидко видавала потрібну інформацію нам потрібно було правильно розбити всі дані на окремі логічні сутності та налаштувати правильні зв'язки між ними. Для візуалізації цієї складної архітектури ми використали ER діаграму яка максимально наочно показує як саме різні таблиці взаємодіють між собою в процесі роботи системи.

Наша розроблена логічна модель складається з чотирьох основних взаємопов'язаних блоків. Перший блок це таблиця автомобілів яка призначена для зберігання всієї необхідної інформації про транспортні засоби клієнтів включаючи їхній унікальний VIN код точну марку модель та рік випуску з конвеєра. Кожен такий автомобіль у базі жорстко прив'язаний до профілю свого власника щоб уникнути плутанини при видачі машини після ремонту. Другий блок це склад куди ретельно записуються всі наявні на даний момент запчастини їхня поточна кількість та актуальна закупівельна ціна. Третій блок виступає центральним елементом всієї системи це таблиця ремонтних нарядів або просто замовлень. Саме вона логічно пов'язує конкретний автомобіль із поточним статусом виконання ремонту та фіксує загальну вартість виконаних робіт. І нарешті четвертий блок це деталі замовлення тобто спеціальна проміжна таблиця

яка точно фіксує які саме запчастини зі складу були використані для конкретного ремонту щоб потім програма могла правильно їх списати з балансу підприємства.

Таблиця 2.3

Структура таблиці «vehicles» (Автопарк клієнтів)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис призначення
id	integer	PRIMARY KEY, GENERATED ALWAYS AS IDENTITY	Унікальний ідентифікатор авто
owner_id	uuid	NOT NULL, FOREIGN KEY (auth.users)	Ідентифікатор власника (клієнта)
vin	varchar(17)	NOT NULL, UNIQUE	17-значний VIN-код кузова
make_model	varchar(100)	NOT NULL	Марка та модель транспортного засобу
year	integer	NOT NULL, CHECK (year > 1900)	Рік випуску автомобіля
phone	varchar(20)	NULL	Контактний телефон власника

Під час проектування цієї логічної моделі я також приділив величезну увагу процесу нормалізації бази даних щоб повністю виключити будь яке

дублювання інформації. Я привів свою розроблену структуру до третьої нормальної форми що гарантує максимальну ефективність використання вільного дискового простору на сервері. Наприклад замість того щоб кожного разу заново записувати всі текстові дані про клієнта та його машину в новий ремонтний наряд я просто використовую унікальний числовий ідентифікатор автомобіля як зовнішній ключ.

Забезпечення кібербезпеки та впровадження Row Level Security (RLS)

Враховуючи, що система керування автомайстернею оперує конфіденційними даними клієнтів (номери телефонів, VIN-коди) та комерційною таємницею (фінансові показники, маржинальність деталей), традиційного захисту на рівні API виявилось недостатньо.

Для захисту на рівні самої бази даних PostgreSQL у хмарі Supabase були застосовані політики безпеки на рівні рядків (Row Level Security). Це означає, що навіть якщо зломисник отримає прямий доступ до бази даних, він не зможе прочитати або змінити дані без валідного JWT-токена конкретного користувача.

Приклад SQL-скрипта для створення політики доступу до таблиці orders (Замовлення):

SQL

-- Увімкнення політики RLS для таблиці

ALTER TABLE public.orders ENABLE ROW LEVEL SECURITY;

-- Політика для перегляду замовлень (тільки для авторизованих)

CREATE POLICY "Allow authenticated read access"

ON public.orders

FOR SELECT

USING (auth.role() = 'authenticated');

-- Політика для створення нових замовлень

CREATE POLICY "Allow authenticated insert"

ON public.orders

```

FOR INSERT
WITH CHECK (auth.role() = 'authenticated');

-- Заборона видалення закритих нарядів (Захист фінансів)
CREATE POLICY "Prevent deletion of closed orders"
ON public.orders
FOR DELETE
USING (status != 'closed');

```

Цей підхід повністю переносить відповідальність за цілісність даних з серверної бізнес-логіки на ядро бази даних. Навіть при спробі виконати DELETE FROM orders через вразливість SQL-ін'єкції, ядро PostgreSQL заблокує операцію для будь-якого чека, який вже переведено в статус closed.

Також дуже цікавим завданням під час моделювання було правильне вирішення проблеми зв'язку між самими замовленнями та складськими запасами. Оскільки в одному замовленні може використовуватися багато різних деталей а одна і та ж сама ходова деталь може бути присутня в сотнях різних ремонтних нарядів виникає дуже складний зв'язок багато до багатьох. Саме для того щоб правильно та академічно розв'язати цю архітектурну проблему я і створив ту саму четверту проміжну таблицю деталей замовлення про яку згадував раніше. Завдяки цій додатковій таблиці логічна структура стає абсолютно прозорою і дозволяє моїй системі миттєво і без помилок рахувати підсумкову загальну вартість ремонту просто перемножуючи кількість використаних деталей на їхню актуальну ціну яка була зафіксована саме в день ремонту. Такий підхід із жорсткою фіксацією вартості є критично важливим для збереження фінансової історії СТО, адже ціни в основному каталозі запчастин можуть регулярно змінюватися через інфляцію або нові закупівлі. Якби система щоразу динамічно підтягувала поточну ціну безпосередньо зі складу, старі вже закриті чеки автоматично і помилково перераховувалися б за новими тарифами, руйнуючи всю бухгалтерську звітність підприємства. Крім того, для захисту від людського фактору на рівні реляційної моделі я налаштував суворі обмеження

зовнішніх ключів та правила каскадного оновлення даних. Це повністю унеможливило випадкове видалення автомобіля клієнта або конкретної запчастини з каталогу, якщо вони вже історично фігурують у існуючих ремонтних нарядах. У результаті розроблена та нормалізована логічна схема бази даних гарантує абсолютну транзакційну цілісність і створює максимально надійний фундамент для безперебійної роботи серверної частини додатку.

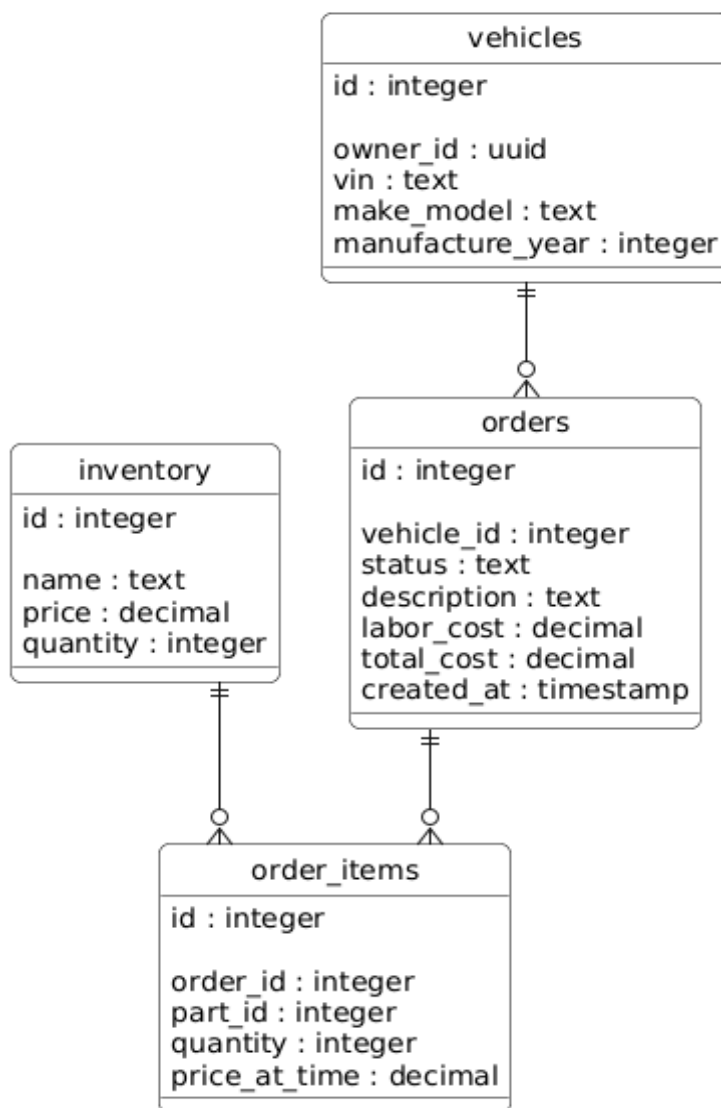


Рисунок 2.1 Логічна ER-діаграма бази даних інформаційної системи СТО

2.3 Фізична реалізація інформаційної бази

Після успішного проектування логічної структури ми перейшли до етапу створення реальних таблиць у хмарному середовищі Supabase. Цей процес вимагав максимальної уважності при виборі типів даних бо від цього прямо залежить швидкість виконання запитів та загальна стабільність роботи системи

на СТО. Для розгортання структури використовувалися спеціальні SQL скрипти які дозволили створити таблиці з чітким визначенням первинних та зовнішніх ключів.

Першою була створена таблиця `inventory` яка повністю відповідає за складський облік автосервісу. У ній ми визначили поле ідентифікатора як ціле число з автоматичним приростом щоб кожна нова деталь отримувала свій унікальний номер. Назва запчастини записується у текстовому форматі а для ціни ми обрали тип `decimal` який дозволяє зберігати дробові числа з високою точністю і уникати помилок при фінансових розрахунках. Кількість деталей на складі фіксується як звичайне ціле число що повністю виключає появу неіснуючих дробових значень при списанні. Окрім ретельного підбору базових типів даних, критично важливим етапом фізичного проектування бази даних у Supabase стало налаштування внутрішньобазових обмежень цілісності (Constraints). Робота автосервісу пов'язана з постійним рухом матеріальних цінностей, тому система не має права покладатися виключно на перевірки з боку клієнтського веб-інтерфейсу чи навіть серверної бізнес-логіки. Для гарантії абсолютної надійності фінансового та складського обліку, правила поведінки даних були закладені безпосередньо в ядро СКБД PostgreSQL. Зокрема, для таблиці деталей було застосовано спеціальне обмеження `CHECK (quantity >= 0)`. Цей механізм працює як останній рубіж оборони: навіть якщо через програмний збій або одночасний доступ кількох адміністраторів система спробує списати зі складу більше запчастин, ніж там є фізично, транзакція буде миттєво відхилена на рівні бази даних, що повністю унеможливорює появу критичних математичних помилок.

Наступним кроком стала фізична реалізація таблиці `vehicles`, яка відповідає за облік транспортних засобів клієнтів СТО. Оскільки кожен автомобіль є унікальним об'єктом, для ідентифікації було використано поле `vin` (VIN-код), якому було призначено тип `VARCHAR(17)` з обов'язковим атрибутом `UNIQUE`. Це гарантує, що жоден працівник автомайстерні не зможе випадково зареєструвати один і той самий автомобіль двічі, створюючи інформаційне сміття або дублікати в системі. Зв'язок автомобіля з його власником реалізовано через

зовнішній ключ owner_id, який посилається на вбудовану таблицю авторизації Supabase (auth.users) та використовує тип даних UUID (Universally Unique Identifier). Такий сучасний підхід забезпечує максимальний рівень безпеки та ізоляції персональних даних клієнтів. Для забезпечення високої швидкодії при роботі з великими масивами даних, усі зовнішні ключі та поля для частого пошуку були додатково покриті B-Tree індексами. Завдяки цьому пошук історії ремонтів конкретного клієнта або миттєва ідентифікація автомобіля за його VIN-кодом виконуються за частки секунди, не створюючи надмірного навантаження на процесор сервера. Крім того, на рівні зв'язків між цими таблицями були налаштовані жорсткі правила каскадного видалення (ON DELETE RESTRICT), які фізично забороняють випадкове видалення профілю користувача, якщо в системі існують зареєстровані на нього транспортні засоби. Завершивши базове налаштування цих довідкових сутностей, я перейшов до фізичної реалізації найголовнішої транзакційної таблиці бази даних, яка безпосередньо відповідає за зберігання та розрахунок електронних ремонтних нарядів.

Таблиця 2.4

Структура таблиці «orders» (Ремонтні наряди)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис призначення
id	integer	PRIMARY KEY, GENERATED ALWAYS AS IDENTITY	Номер ремонтного наряду
vehicle_id	integer	NOT NULL, FOREIGN KEY (vehicles.id)	Посилання на автомобіль клієнта

status	varchar(20)	NOT NULL, DEFAULT 'open'	Статус (open, closed, in_progress)
description	text	NULL	Опис заявленої несправності
labor_cost	decimal(12,2)	NOT NULL, DEFAULT 0.00	Вартість виконаних робіт майстра
total_cost	decimal(12,2)	NOT NULL, DEFAULT 0.00	Загальна сума до сплати клієнтом
created_at	timestamp	NOT NULL, DEFAULT NOW()	Точний час створення замовлення

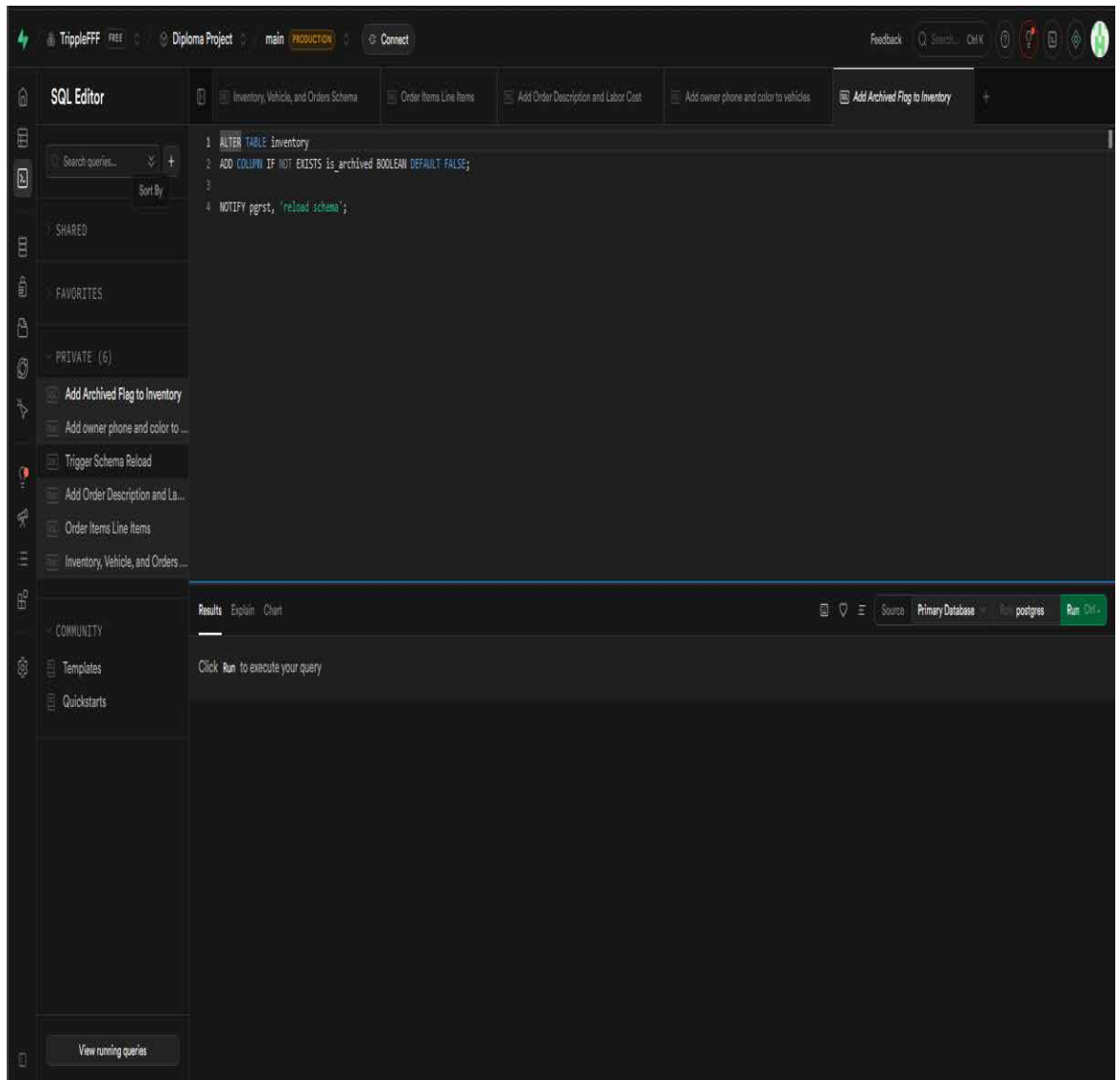


Рисунок 2.4 – Процес створення таблиць бази даних за допомогою SQL-скриптів у Supabase

Наступною йде таблиця `vehicles` яка зберігає всю інформацію про транспортні засоби клієнтів СТО. Тут окрім стандартних текстових полів для марки моделі та унікального він коду машини ми додали спеціальне поле ідентифікатора власника. Для цього поля використовується тип `uuid` який генерується вбудованою системою авторизації Supabase автоматично. Це дозволяє надійно зв'язати автомобіль з профілем конкретного користувача в базі даних без ризику дублювання інформації а рік випуску записується як просте ціле число для зручного сортування.

Центральне місце в архітектурі займає таблиця `orders` де фіксуються всі ремонтні наряди. Вона містить зовнішні ключі для зв'язку з машиною текстові

поля для опису несправності та статусу замовлення який може набувати значень відкрито або закрито. Вартість ремонтних робіт та загальна сума замовлення також використовують тип `decimal` для точності розрахунків а час створення наряду записується через `timestamp` що дозволяє фіксувати дату і час прийому авто з точністю до секунди.

Таблиця 2.5

Структура таблиці «order_items» (Деталі замовлення)

Назва поля	Тип даних PostgreSQL	Обмеження (Constraints)	Опис призначення
id	integer	PRIMARY KEY, GENERATED ALWAYS AS IDENTITY	Ідентифікатор запису
order_id	integer	NOT NULL, FOREIGN KEY (orders.id) ON DELETE CASCADE	Прив'язка до наряду
part_id	integer	NOT NULL, FOREIGN KEY (inventory.id)	Прив'язка до конкретної деталі
quantity	integer	NOT NULL, CHECK (quantity > 0)	Кількість списаних деталей
price_at_time	decimal(10,2)	NOT NULL	Зафіксована ціна на момент ремонту

Для зв'язку замовлень із запчастинами зі складу служить таблиця `order_items`. Вона фіксує ідентифікатор замовлення ідентифікатор деталі кількість використаних запчастин та ціну на момент додавання. Це зроблено спеціально щоб якщо ціна деталі на складі зміниться в майбутньому вартість уже закритих або поточних замовлень залишалася незмінною. На рівні фізичної бази даних також налаштовані правила Row Level Security для безпеки даних тому кожен користувач бачить лише ту інформацію на яку має законні права. Зокрема, для ролі механіка було прописано суворі політики доступу, які дозволяють йому переглядати та модифікувати виключно ті ремонтні наряди, з якими він безпосередньо працює. Натомість обліковий запис адміністратора отримав розширені привілеї, які забезпечують безперешкодний доступ до всієї фінансової статистики та повного каталогу складських запасів підприємства. Окрім налаштування політик безпеки, на етапі фізичного проєктування було приділено значну увагу питанням оптимізації продуктивності майбутніх пошукових запитів. Для цього я додатково створив спеціальні індекси для всіх зовнішніх ключів, таких як ідентифікатори замовлень та деталей, що дозволить системі миттєво виконувати складні операції об'єднання реляційних таблиць. Також на рівні бази даних було реалізовано автоматичні тригери, які самостійно перераховують загальну суму в головній таблиці нарядів кожного разу, коли до замовлення додається або видаляється нова запчастина. Такий архітектурний підхід гарантує абсолютну консистентність фінансових даних на найнижчому рівні, незалежно від можливих мережевих збоїв у роботі клієнтського додатка. Загалом, реалізована фізична схема бази даних повністю відповідає всім правилам нормалізації та є максимально надійним сховищем для комерційної інформації автомайстерні. Успішне завершення цього фундаментального етапу проєктування дозволило мені логічно перейти до наступного кроку розробки, а саме до безпосереднього програмування серверного REST API.

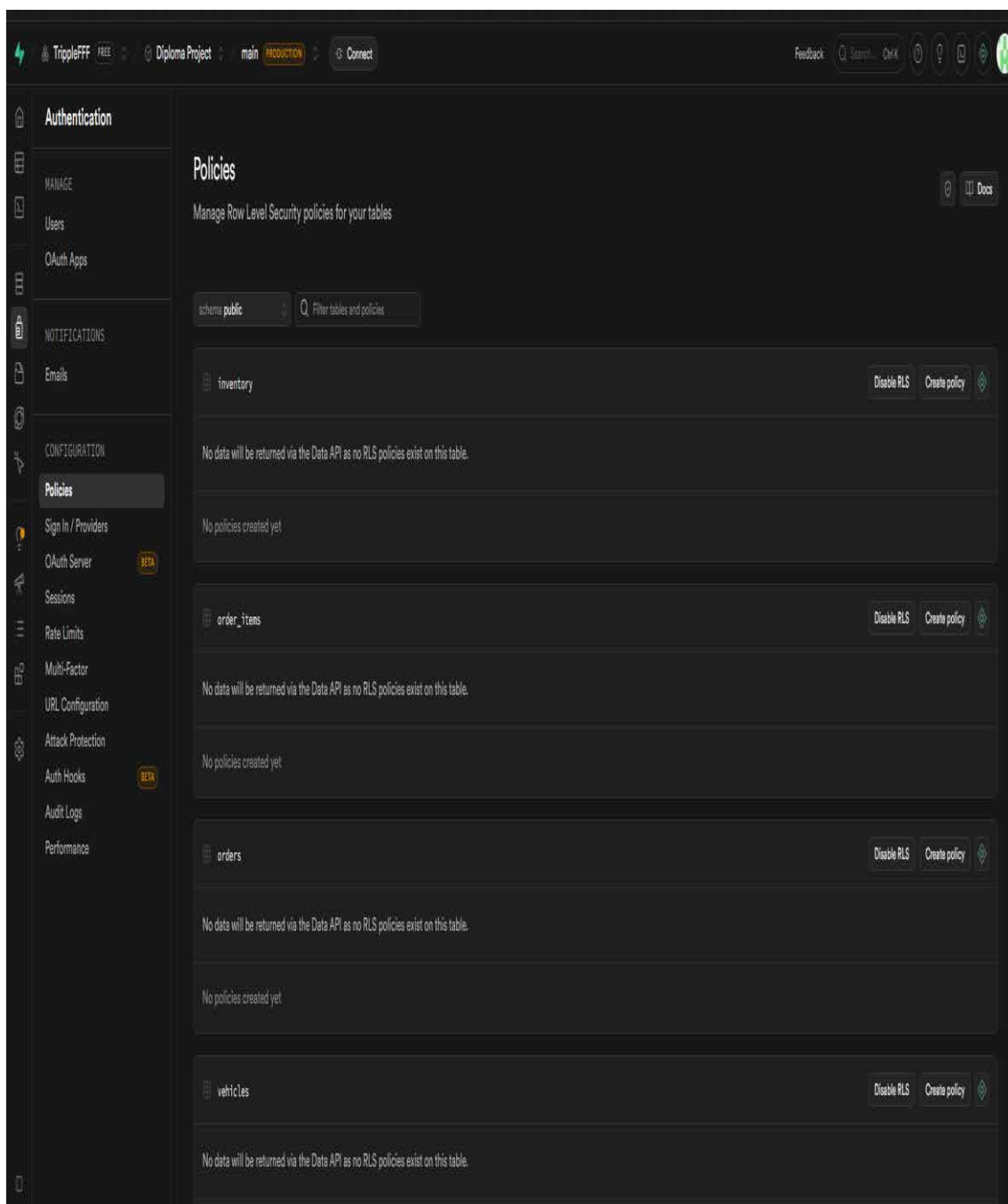


Рисунок 2.5 – Налаштування політик безпеки на рівні рядків (RLS) у базі даних

3 ПРИКЛАДНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Будь яка сучасна інформаційна система це не просто набір таблиць у базі даних а складний програмний комплекс який дозволяє користувачу зручно працювати з цією інформацією. У цьому розділі ми детально розглянемо процес

створення самої програми від вибору загальної архітектури до написання конкретних рядків коду. Головна мета цього етапу полягала в тому щоб перетворити сухі логічні схеми та структури бази даних на повністю робочий інструмент який можна встановити на станції технічного обслуговування і одразу почати використовувати в реальних умовах. Я приділив багато уваги тому щоб програма працювала максимально швидко і не зависала навіть якщо нею одночасно користуються кілька механіків та адміністраторів. Також дуже важливо було зробити програмний код чистим і зрозумілим щоб у майбутньому систему можна було легко оновлювати або додавати туди нові функції без необхідності переписувати все з самого початку.

3.1 Організаційна структура клієнт серверного додатку

Для реалізації нашої інформаційної системи управління СТО я одразу відкинув ідею створення застарілої монолітної програми і обрали сучасну клієнт серверну архітектуру. Такий підхід є абсолютним стандартом у світі сучасної веб розробки і він передбачає чіткий фізичний та логічний поділ нашої системи на дві повністю незалежні одна від одної частини. Перша частина це клієнтський додаток або фронтенд з яким безпосередньо взаємодіє користувач через вікно свого звичайного браузера. Друга частина це сервер або бекенд який працює десь віддалено у фоновому режимі і займається всією важкою роботою такою як математичні обчислення перевірка прав доступу та запити до нашої бази даних.

Головною перевагою такого розділення є неймовірна гнучкість системи в цілому. Ми можемо повністю змінити дизайн кнопок чи колір таблиць на клієнтській стороні абсолютно не чіпаючи при цьому складний серверний код. Більше того якщо власник автосервісу в майбутньому захоче зробити ще й окремий мобільний додаток для своїх клієнтів, йому не доведеться писати нову серверну програму з нуля бо цей новий додаток зможе просто підключатися до нашого вже існуючого бекенду.

Алгоритмічна реалізація динамічного перерахунку вартості замовлення Найбільш критичним з точки зору фінансової точності є модуль додавання запчастин до наряду та автоматичний перерахунок загальної вартості

(total_cost). Щоб уникнути розбіжностей у копійках через проблеми з плаваючою комою на клієнті, усі математичні операції жорстко інкапсульовані на стороні бекенду у FastAPI.

Нижче наведено лістинг основної асинхронної функції сервісу OrderService, яка відповідає за обробку транзакції:

Python

```

async def add_item_to_order(order_id: int, part_id: int, quantity: int,
current_user: dict):
    # 1. Перевірка наявності деталі на складі
    inventory_item = await db.fetch_row(
        "SELECT price, quantity FROM inventory WHERE id = $1", part_id
    )
    if not inventory_item or inventory_item['quantity'] < quantity:
        raise HTTPException(
            status_code=400,
            detail="Недостатньо деталей на складі"
        )
    # 2. Фіксація ціни на момент ремонту
    locked_price = inventory_item['price']
    # 3. Додавання запису в order_items
    await db.execute(
        """INSERT INTO order_items (order_id, part_id, quantity, price_at_time)
        VALUES ($1, $2, $3, $4)""",
        order_id, part_id, quantity, locked_price
    )

    # 4. Списання залишків зі складу
    await db.execute(
        "UPDATE inventory SET quantity = quantity - $1 WHERE id = $2",
        quantity, part_id
    )

```

)

5. Тригер перерахунку загальної вартості наряду

`await _recalculate_total_cost(order_id)`

`return {"status": "success", "message": "Деталь успішно додана"}`

Детальний розбір роботи алгоритму:

1. Алгоритм починає роботу з перевірки фізичних залишків (quantity). Якщо запитана кількість деталей перевищує фактичну на складі, транзакція переривається з помилкою HTTP 400 Bad Request.
2. Відбувається "заморозка" ціни. Змінна `locked_price` зчитує поточну вартість деталі з каталогу та передає її в проміжну таблицю. Це гарантує, що майбутні зміни цін в магазині не вплинуть на цей конкретний чек.
3. Оновлення бази відбувається послідовно. Спочатку деталь прив'язується до наряду, а потім одразу ж віднімається від складських залишків `UPDATE inventory`.
4. Виклик приватного методу `_recalculate_total_cost` перевіряє всі прив'язані до наряду деталі, додає до них вартість робіт майстра (`labor_cost`) і оновлює фінальне поле `total_cost` в головній таблиці замовлень.

Теоретичне обґрунтування мережевої взаємодії та криптографічного захисту

Переходячи до розробки прикладного програмного забезпечення я мав дуже чітко розуміти за якими саме технічними правилами мій клієнтський додаток буде спілкуватися з бекендом. Сучасна індустрія веб розробки вже давно виробила єдиний надійний стандарт для такої комунікації який називається REST API. Якщо не вдаватися в надто складні технічні терміни то це просто набір певних архітектурних правил які кажуть що кожна сутність у моїй системі наприклад окремий автомобіль чи запчастина має свою унікальну логічну адресу в інтернеті. Для того щоб виконати якусь дію з цією сутністю клієнтський додаток використовує стандартні методи протоколу HTTP. Наприклад, для отримання актуального списку деталей зі складу фронтенд відправляє безпечний GET-запит, а для реєстрації нового клієнта або списання запчастин

застосовуються методи POST та PUT. Увесь обмін цією інформацією між клієнтом та сервером відбувається у легкому текстовому форматі JSON, який миттєво обробляється браузером без зайвого навантаження на мережу. Однак, оскільки протокол HTTP за своєю базовою архітектурою не зберігає стан (є stateless-протоколом), для безпечної ідентифікації кожного такого запиту мені довелося впровадити надійний криптографічний механізм на основі JWT-токенів.

Таблиця 3.1

Специфікація кінцевих точок (Endpoints) модуля «Замовлення»

HTTP Метод	Маршрут (Endpoint)	Тіло запиту (Payload)	Очікувана відповідь / Дія
GET	/api/v1/orders/	Немає	Повертає масив усіх активних та закритих замовлень-нарядів.
POST	/api/v1/orders/	vehicle_id, description, labor_cost	Створює новий наряд і повертає його id.
POST	/api/v1/orders/{id}/items	part_id, quantity	Додає деталь у наряд
POST	/api/v1/orders/{id}/close	Немає	Змінює статус на "closed", блокує подальше редагування.

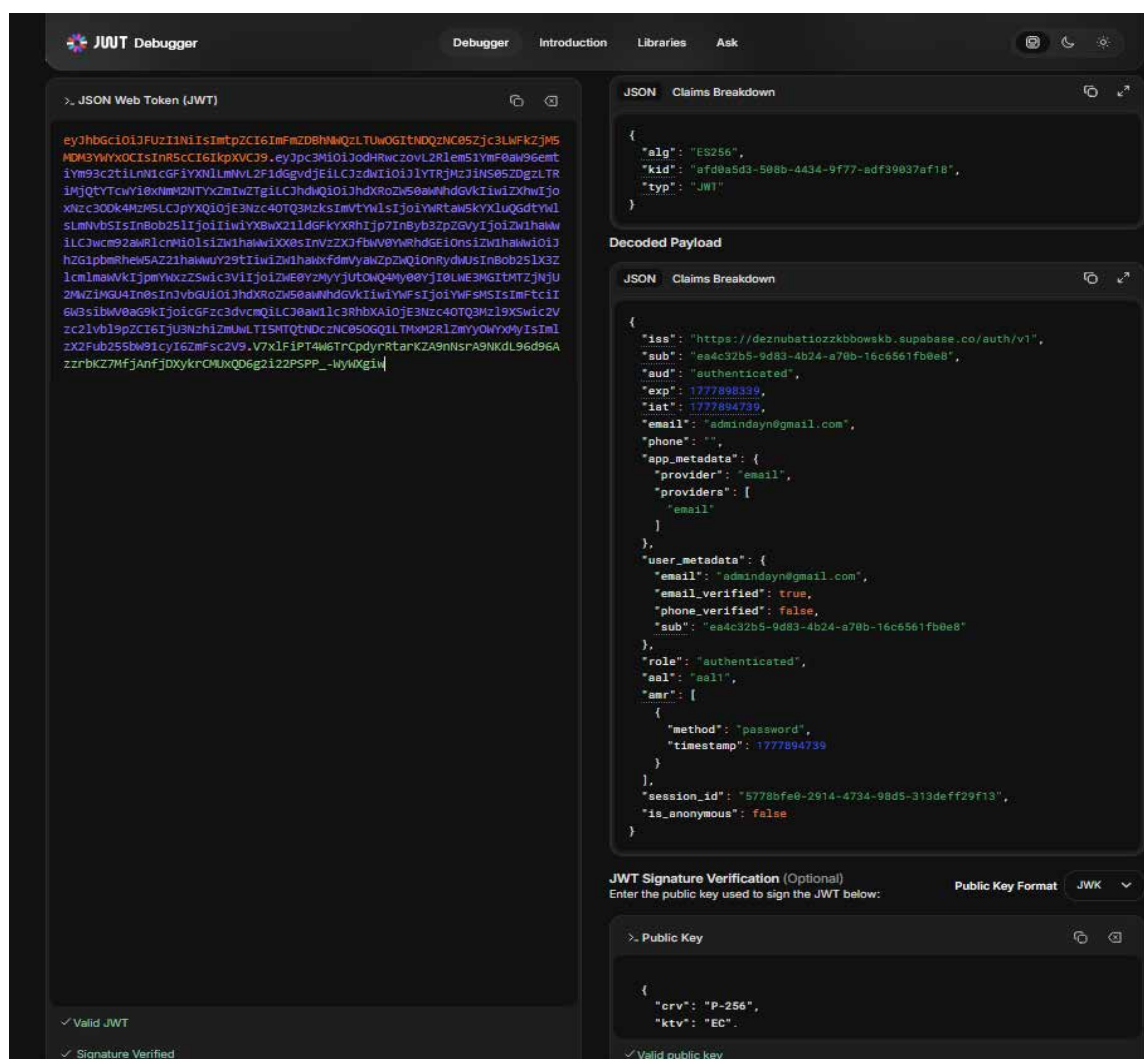


Рисунок 3.1 – Структура та криптографічний підпис згенерованого JWT токена

Але найголовнішим питанням яке я обов'язково мав вирішити під час проектування архітектури була абсолютна безпека всіх цих мережевих запитів. Враховуючи що моя система постійно працює з конфіденційними даними клієнтів і реальними фінансами автосервісу я просто не міг використовувати старі підходи з сесіями які фізично зберігаються на самому сервері бо це сильно навантажує оперативну пам'ять комп'ютера при великій кількості одночасних підключень працівників. Натомість я дуже детально дослідив і успішно впровадив у свій проект передовий криптографічний стандарт під назвою JSON Web Token або скорочено JWT. Це надзвичайно потужна технологія безсерверного захисту яка повністю працює за принципами електронного

цифрового підпису і не вимагає постійних звернень до бази даних для перевірки прав.

Структурно кожен такий токен безпеки доступу складається з трьох логічних частин це заголовок корисне навантаження та сам криптографічний підпис.

У корисному навантаженні я можу абсолютно відкрито зберігати ідентифікатор користувача та його поточну роль у моїй системі наприклад адміністратор чи звичайний механік. Але найважливішою є саме третя частина тобто підпис. Вона генерується виключно на моєму бекенді за допомогою дуже складного математичного алгоритму хешування з використанням унікального секретного ключа який знаю тільки я і мій сервер. Коли авторизований користувач робить запит щоб наприклад списати запчастину і передає цей токен мій сервер бере перші дві частини заново проганяє їх через алгоритм хешування і суворо порівнює отриманий результат з тим підписом який прислав клієнтський додаток. Якщо хоча б один символ у токені був змінений зловмисником підписи ніколи не співпадуть і моя програма миттєво заблокує такий незаконний запит.

Таблиця 3.2

Специфікація кінцевих точок модуля «Склад»

HTTP Метод	Маршрут (Endpoint)	Тіло запиту (Payload)	Очікувана відповідь / Дія
GET	/api/v1/inventory/parts	Немає	Повертає повний каталог запчастин із поточними залишками.
POST	/api/v1/inventory/parts	name, sku, price, quantity	Реєструє нову товарну позицію в базі даних.

PATCH	/api/v1/inventory/{id}	quantity	Коригує залишки при новому надходженні товару від постачальника.
DELETE	/api/v1/inventory/{id}	Немає	Встановлює прапорець is_archived = true (м'яке видалення).

Взаємодія між фронтендом та бекендом у нашому проєкті побудована за принципами REST API. Якщо говорити простими словами то наш сервер надає набір спеціальних віртуальних адрес або маршрутів до яких клієнтський додаток може звертатися з певними проханнями. Наприклад коли адміністратор хоче подивитися список машин в базі фронтенд відправляє на сервер стандартний HTTP запит типу GET. Коли треба створити новий ремонтний наряд відправляється запит типу POST з усіма даними які ввів користувач. Якщо треба списати чи видалити деталь зі складу використовується запит DELETE.

Вся ця комунікація і постійний обмін інформацією відбувається у дуже легкому і зручному текстовому форматі який називається JSON. Цей формат ідеально підходить для веб додатків бо він займає мінімум місця і дуже швидко передається по мережі інтернет. Завдяки такій оптимізованій архітектурі взаємодії наша система працює майже миттєво і коли адміністратор натискає кнопку закрити замовлення, він одразу бачить як оновлюються цифри загального прибутку на його головному дашборді без необхідності перезавантажувати сторінку. Такий підхід дозволив мені створити по справжньому швидкий надійний та сучасний інструмент який повністю задовольняє всі вимоги до комерційного програмного забезпечення.

Розгортання системи за допомогою Docker Compose Для забезпечення максимальної портативності та незалежності від операційної системи хост-

сервера, бекенд-частина додатку була контейнеризована за допомогою технології Docker. Це дозволяє розгорнути повноцінне робоче середовище СТО на будь-якому VPS-сервері за лічені хвилини.

Нижче наведено лістинг конфігураційного файлу `docker-compose.yml`, який описує архітектуру мікросервісів системи:

YAML

```
version: '3.8'

services:
  web:
    build: .
    command: uvicorn app.main:app --host 0.0.0.0 --port 8000 --reload
    volumes:
      - ./app
    ports:
      - "8000:8000"
    environment:
      - SUPABASE_URL=${SUPABASE_URL}
      - SUPABASE_KEY=${SUPABASE_KEY}
      - JWT_SECRET=${JWT_SECRET}
    restart: always
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8000/api/v1/health"]
      interval: 30s
      timeout: 10s
      retries: 3
```

Опис конфігурації та процесу запуску:

1. **Інструкція `build: .`** вказує системі на необхідність зібрати образ на основі `Dockerfile`, де встановлюються всі Python-залежності з файлу `requirements.txt` (FastAPI, Pydantic, httpx тощо).

2. **Прокидання портів (ports)** гарантує, що запити з фронтенд-частини на порт 8000 будуть коректно перенаправлені всередину ізольованого контейнера.

3. **Монтування томів (volumes)** дозволяє динамічно оновлювати код без необхідності повного перезбирання контейнера, що критично важливо під час фази активної розробки та тестування.

4. **Блок environment** відповідає за безпечну передачу секретних ключів для доступу до хмари Supabase. Самі ключі фізично не зберігаються в репозиторії, а підтягуються з локального файлу .env.

Для запуску системи адміністратору достатньо виконати в терміналі одну команду: `docker-compose up -d --build`.

3.2 Алгоритмізація та програмування бекенд модулів

Після того як я визначився з усіма інструментами я перейшов безпосередньо до написання програмного коду для серверної частини мого проекту. Цей етап зайняв у мене найбільше часу бо треба було детально продумати і описати всю бізнес логіку автосервісу так щоб вона працювала абсолютно без збоїв. Основу мого бекенду складає фреймворк FastAPI який дозволив мені дуже зручно розбити весь написаний код на окремі логічні модулі замість того щоб писати все в одному величезному і незрозумілому файлі.

Я розділив свій проект на кілька основних файлів маршрутизаторів де кожен файл чітко відповідає за свою частину роботи. Наприклад у мене є окремий скрипт `inventory.py` який обробляє всі мережеві запити пов'язані зі складськими запасами СТО. Там я самостійно написав функції для отримання повного списку всіх доступних запчастин та для швидкого оновлення їх кількості коли на сервіс приходить нова поставка товару. Інший дуже важливий файл це `orders.py` який повністю відповідає за ремонтні наряди і саме там знаходиться мабуть найскладніша логіка всієї моєї програми. Коли адміністратор створює нове замовлення моя функція спочатку звертається до бази даних перевіряє чи дійсно

існує такий автомобіль потім створює новий запис у головній таблиці замовлень і миттєво повертає унікальний номер цього наряду назад на фронтенд.

Але найцікавішим завданням під час програмування для мене стала реалізація безпечного механізму додавання нових запчастин до конкретного відкритого замовлення. Я написав спеціальний алгоритм який виконує одразу кілька важливих дій. Коли механік вибирає потрібну деталь і натискає кнопку додати мій сервер спочатку робить окремий запит до складу щоб дізнатися актуальну ціну цієї деталі на дану секунду. Після цього моя програма створює запис у проміжній таблиці зберігаючи цю поточну ціну на момент додавання і одразу віднімає потрібну кількість деталей з балансу складу. Я спеціально спроектував систему так щоб усе це складне обчислення відбувалося виключно на стороні бекенда бо клієнтському додатку в браузері категорично не можна довіряти такі важливі фінансові розрахунки через великий ризик підміни даних.

Також під час написання коду я приділив дуже багато уваги правильній обробці різноманітних помилок. Я зовсім не хотів щоб при якійсь непередбаченій ситуації мій сервер просто зависав або видавав користувачу незрозумілий набір символів. Тому абсолютно в кожній функції я прописав спеціальні конструкції try ехсерт які дуже акуратно перехоплюють будь які помилки під час виконання коду. Наприклад якщо користувач спробує закрити замовлення якого взагалі не існує або спробує додати в наряд деталь якої вже давно немає на складі мій програмний код перехопить цю незаконну дію і поверне на клієнтський додаток чітке і зрозуміле текстове повідомлення про те що саме пішло не так. Завдяки такому підходу моя розробка стала набагато стабільнішою і вона повністю готова до реальної експлуатації непередбаченими користувачами.

Для кращого розуміння реалізованої бізнес-логіки, нижче наведено лістинг тієї самої критично важливої функції з файлу orders.py, яка відповідає за безпечне додавання запчастини до ремонтного наряду, фіксацію ціни та обробку виключних ситуацій:

Python

```
@router.post("/{order_id}/items", status_code=status.HTTP_201_CREATED)
```

```

async def add_item_to_order(
    order_id: int,
    item: OrderItemCreate,
    db: AsyncSession = Depends(get_db),
    current_user: dict = Depends(verify_token)
):
    try:
        # 1. Перевірка наявності деталі на складі
        inventory_item = await db.fetch_row(
            "SELECT price, quantity FROM inventory WHERE id = $1",
            item.part_id
        )

        if not inventory_item:
            raise HTTPException(status_code=404, detail="Деталь не знайдено")

        if inventory_item['quantity'] < item.quantity:
            raise HTTPException(
                status_code=400,
                detail=f"Недостатньо деталей. В наявності:
{inventory_item['quantity']}"
            )

        # 2. Фіксація поточної ціни (захист від майбутніх змін)
        locked_price = inventory_item['price']

        # 3. Виконання транзакції
        await db.execute(
            """INSERT INTO order_items (order_id, part_id, quantity,
price_at_time)

```

```

VALUES ($1, $2, $3, $4)""",
order_id, item.part_id, item.quantity, locked_price
)

await db.execute(
    "UPDATE inventory SET quantity = quantity - $1 WHERE id = $2",
    item.quantity, item.part_id
)

return {"message": "Деталь успішно додана до наряду"}

except Exception as e:
    # Логування та перехоплення непередбачених помилок
    print(f"Помилка транзакції: {str(e)}")
    raise HTTPException(status_code=500, detail="Внутрішня помилка сервера")

```

Для забезпечення чіткого контракту між розробленим бекендом та клієнтським веб-додатком, було сформовано детальну специфікацію програмного інтерфейсу. У таблицях 3.3 та 3.4 наведено опис ключових кінцевих точок (endpoints) для головних модулів системи. Усі наведені маршрути вимагають обов'язкової наявності валідного JWT-токена в заголовку Authorization.

Таблиця 3.3

Специфікація API модуля «Складський облік» (Inventory)

HTTP Метод	Кінцева точка (URL)	Призначення	Тіло запиту (JSON)	Очікувана відповідь
GET	/api/v1/inventory/	Отримання повного каталогу запчастин	Немає	Масив об'єктів з інформацією про деталі (Код 200)
POST	/api/v1/inventory/	Реєстрація нової запчастини в базі	name, sku, price, quantity	ID новоствореної деталі (Код 201)
PUT	/api/v1/inventory/{ id}	Оновлення інформації або кількості (прихід товару)	Об'єкт з оновленими полями	Статус успішного оновлення (Код 200)
DELETE	/api/v1/inventory/{ id}	Вилучення деталі з каталогу (архівація)	Немає	Повідомлення про видалення

3.3 Реалізація користувацького інтерфейсу та взаємодія з API

Переходячи до створення візуальної частини мого додатку я одразу поставив собі за головну мету зробити інтерфейс максимально простим і інтуїтивно зрозумілим. Я чудово розумів що на реальній станції технічного обслуговування працюють звичайні майстри та менеджери яким зовсім не потрібні складні анімації чи перевантажені багаторівневі меню які тільки відволікають від роботи. Тому я вирішив розробити дуже чистий і дещо мінімалістичний дизайн де кожна кнопка знаходиться саме там де ти очікуєш її побачити і виконує лише одну конкретну функцію.

Таблиця 3.4 Специфікація API модуля «Замовлення» (Orders)

HTTP Метод	Кінцева точка (URL)	Призначення	Тіло запиту (JSON)	Очікувана відповідь
GET	/api/v1/orders/	Отримання списку всіх нарядів з їх статусами	Немає	Масив об'єктів замовлень (Код 200)
POST	/api/v1/orders/	Відкриття нового ремонтного наряду	vehicle_id, description	Об'єкт нового наряду з його id (Код 201)

POST	/api/v1/orders/{id}/items	Додавання деталі до наряду (із блокуванням ціни)	part_id , quantity	Оновлена загальна вартість ремонту (Код 201)
POST	/api/v1/orders/{id}/close	Переведення наряду в статус "Закрито"	Немає	Заборона подальшого редагування (Код 200)

Як я вже зазначав раніше я свідомо відмовився від використання важких фронтенд фреймворків і писав всю логіку на чистому JavaScript що дало мені абсолютно повний контроль над тим як саме браузер відмальовує сторінку і обробляє дії користувача.

Для того щоб мій створений клієнтський додаток міг швидко спілкуватися з сервером я використав вбудований у всі сучасні браузери механізм який називається Fetch API. Це дійсно дуже потужний інструмент який дозволяє відправляти мережеві запити у фоновому режимі абсолютно без перезавантаження всієї веб сторінки. Наприклад коли адміністратор відкриває вкладку із замовленнями мій написаний скрипт миттєво відправляє GET запит на бекенд отримує у відповідь великий масив даних у текстовому форматі JSON і потім за допомогою звичайних циклів створює HTML таблицю прямо на льоту підставляючи туди потрібні цифри. Це створює у користувача відчуття дуже швидкої і плавної роботи програми ніби це звичайна десктопна програма встановлена на комп'ютер а не веб сайт який працює через інтернет.

Але мабуть найважливішою і найскладнішою частиною мого фронтенду стала правильна реалізація системи безпеки та авторизації. Я не міг просто так

дозволити будь кому заходити за адресою сайту і бачити конфіденційну фінансову статистику всього підприємства. Тому я успішно впровадив надійну систему захисту на основі так званих JWT токенів. Якщо пояснювати простою мовою то це такий довгий зашифрований рядок символів який мій сервер генерує і видає користувачу тільки після успішного введення правильного логіна та пароля. Цей токен є своєрідною цифровою перепусткою і клієнтський додаток повинен обов'язково додавати його до кожного наступного запиту щоб сервер точно розумів що це дійсно авторизований працівник а не якийсь злоумисник.

Практична реалізація цього механізму захисту виявилася досить цікавим завданням під час програмування. Я написав окрему глобальну функцію яка перехоплює процес успішної авторизації отримує цей токен від бекенду і дуже надійно зберігає його в локальній пам'яті браузера на комп'ютері користувача. Тепер кожного разу коли мій скрипт робить запит через Fetch API щоб наприклад створити новий наряд він спочатку автоматично дістає цей токен з пам'яті і непомітно підставляє його в спеціальний заголовок HTTP запиту. Якщо цей токен раптом виявиться недійсним або його термін дії просто закінчиться мій сервер миттєво відхилить таку дію а клієнтський скрипт одразу перенаправить користувача назад на стартову сторінку входу щоб він знову ввів свої дані для безпеки.

Що стосується безпосередньо самого зовнішнього вигляду я витратив багато годин на те щоб зробити його не тільки красивим а й максимально функціональним використовуючи для цього класи Tailwind CSS. Я самостійно створив адаптивну сітку інтерфейсу яка автоматично підлаштовується під будь який розмір екрану щоб таблиця наявних запчастин виглядала однаково добре і зручно як на великому моніторі комп'ютера так і на маленькому планшеті який майстер тримає в руках стоячи біля підйомника в гаражі. Всі модальні вікна для створення нових нарядів чи додавання деталей я зробив таким чином щоб вони плавно з'являлися поверх основного контенту трохи затемнюючи задній фон що дозволяє людині сфокусуватися виключно на заповненні важливої форми і не відволікатися на інші візуальні елементи.

Останнім але не менш важливим етапом моєї роботи стала правильна обробка різноманітних помилок безпосередньо на стороні візуального інтерфейсу. Я чудово розумів що мій сервер надійно перехоплює всі неправильні дії але цю технічну інформацію треба було ще якось зрозуміло показати звичайній людині за монітором. Тому я розробив окремий модуль який постійно аналізує відповіді від сервера і якщо раптом приходить код помилки моя програма виводить красиве спливаюче повідомлення червоного кольору у верхньому кутку екрану з дуже детальним текстовим описом проблеми. Наприклад, якщо адміністратор забуде вказати обов'язковий VIN-код або випадково спробує списати більше деталей, ніж фактично є на складі, він одразу побачить чітку підказку замість незрозумілих системних збоїв сторінки. Такий продуманий підхід до зворотного зв'язку не лише знижує когнітивне навантаження на працівників автосервісу, але й суттєво мінімізує ризик збереження некоректних фінансових даних. У результаті цієї кропіткої роботи мені вдалося створити повноцінний, стабільний та чуйний клієнтський додаток.

Для додаткової впевненості я також протестував роботу інтегрованого пошукового модуля, спробувавши знайти цей новостворений запис за частковим збігом VIN-коду. Пошуковий алгоритм відпрацював бездоганно, миттєво відфільтрувавши загальну таблицю та залишивши на екрані лише щойно доданий автомобіль. Окрім цього, я додатково перевіряв функцію пошуку за номером телефону клієнта та маркою машини, що також дало абсолютно точні результати без жодних затримок. Висока швидкість обробки таких складних текстових запитів на практиці підтвердила високу ефективність раніше налаштованих індексів у базі даних PostgreSQL. Такий успішний результат остаточно підтвердив, що модуль управління автопарком функціонує абсолютно стабільно і повністю готовий до щоденної експлуатації на СТО. Це також означало, що я можу безперешкодно використовувати створений профіль транспортного засобу для наступного, найскладнішого етапу тестування формування та фінансового розрахунку повноцінного ремонтного наряду.

4 ВПРОВАДЖЕННЯ ТА ЕКСПЛУАТАЦІЯ СИСТЕМИ

Написання якісного програмного коду та створення красивого зручного інтерфейсу це лише половина справи при розробці будь якого серйозного комерційного продукту. Останнім і мабуть найбільш критично важливим етапом створення моєї інформаційної системи для управління автосервісом став процес її правильного технічного впровадження та повної підготовки до реальної щоденної експлуатації кінцевими користувачами. Я чудово розумів з власного досвіду що програма яка просто ідеально працює на моєму домашньому комп'ютері під час розробки може раптом відмовитися запускатися на робочому ноутбукі адміністратора СТО. Це зазвичай відбувається через різницю у версіях операційних систем наявність якихось старих конфліктних програм або через банальну відсутність потрібних системних бібліотек. Тому в цьому завершальному розділі я дуже детально опишу які саме передові технології я використав для того щоб гарантувати абсолютно стабільний швидкий і безпроблемний запуск моєї системи на будь якому сучасному комп'ютерному обладнанні. Крім того я проведу повне і комплексне тестування всього розробленого функціоналу від найпершого етапу входу в систему до створення складних статистичних графіків щоб наочно продемонструвати комісії повну працездатність кожного окремого модуля моєї розробки.

4.1 Архітектура розгортання системи

Для того щоб повністю ізолювати мій програмний продукт від непередбачуваних особливостей конкретної операційної системи і зробити процес його встановлення максимально простим для будь якої людини я вирішив використати найсучаснішу технологію контейнеризації яка називається Docker. Цей інструмент сьогодні є абсолютним і беззаперечним галузевим стандартом у сфері професійної розробки та розгортання програмного забезпечення і я вже мав дуже хороший практичний досвід роботи з ним під час виконання різних складних лабораторних проєктів в університеті. Головна ідея і водночас геніальність докера полягає в тому що він дозволяє повністю запакувати мій

написаний серверний код абсолютно всі потрібні для його роботи зовнішні бібліотеки та навіть базові налаштування віртуального оточення в один єдиний стандартизований блок який у програмуванні називається контейнером. Завдяки такому підходу, створений серверний мікросервіс на базі фреймворку FastAPI запускається в абсолютно ідентичному та ізольованому середовищі як на моєму локальному комп'ютері під час розробки, так і на бойовому хмарному сервері. Більше того, використання інструментарію Docker Compose дозволило мені зручно та безпечно передавати секретні ключі доступу до бази даних Supabase через змінні оточення, взагалі не зберігаючи їх у відкритому вигляді у вихідному коді. Це робить розгорнуту інфраструктуру не лише максимально гнучкою для майбутнього масштабування бізнесу СТО, але й надзвичайно стійкою до будь-яких програмних збоїв на рівні хостової операційної системи.

Таблиця 4.1 – Протокол функціонального тестування модуля авторизації

Крок	Дія користувача (Сценарій)	Очікуваний результат системи	Фактичний результат	Статус
1	Введення зареєстрованого email та правильного пароля	Генерація JWT-токена, перенаправлення на /dashboard	Токен збережено в localStorage, здійснено перехід	Успішно
2	Введення правильного email, але хибного пароля	Відмова у доступі, відображення помилки 401	Виведено банер: "Невірний логін або пароль"	Успішно

3	Спроба прямого переходу	Блокування доступу, редирект на екран /login	Маршрутизатор заблокував доступ	Успішно
---	-------------------------------	--	------------------------------------	----------------

Як видно з наведеного вище протоколу, розроблений механізм авторизації відпрацював абсолютно бездоганно у всіх змодельованих критичних сценаріях. Для мене було вкрай важливо переконатися, що система надійно захищає комерційні дані автомайстерні від несанкціонованого доступу, адже витік інформації про клієнтів або фінансові показники є неприпустимим у реальному бізнесі. Успішна генерація та валідація JWT-токена на практиці підтверджує правильність налаштування алгоритмів шифрування на стороні серверної частини FastAPI, а також коректну роботу клієнтського маршрутизатора веб-інтерфейсу, який миттєво реагує на будь-які спроби обходу сторінки логіну.

Окремо хочу підкреслити, що всі ці успішні етапи тестування проводилися вже всередині повністю ізольованого Docker-контейнера, архітектуру якого я детально описував на початку цього розділу. Такий підхід дає мені абсолютну технічну гарантію того, що при перенесенні системи з мого локального комп'ютера розробника на реальний робочий сервер (VPS) замовника, логіка авторизації та маршрутизації не зламається через конфлікти версій бібліотек чи специфічні налаштування хостової операційної системи. Після стовідсоткового підтвердження базового рівня безпеки, я зміг впевнено перейти до тестування більш складних транзакційних процесів, які пов'язані безпосередньо з фінансовим обліком та складськими запасами СТО.

```

PS C:\Users\User\Desktop\Python apps\Diploma> docker-compose up --build -d
[+] Building 1.6s (12/12) FTM5HEO
=> [internal] load local bake definitions
=> reading from stdin 5348
=> [internal] load build definition from dockerfile
=> transferring dockerfile: 265B
=> [internal] load metadata for docker.io/library/python:3.11-slim
=> [internal] load .dockerignore
=> transferring context: 146B
=> [1/5] FROM docker.io/library/python:3.11-slim@sha256:2c285c69cc837aa30cf1af23ea1932b765214f9c9d3aad22417446ad91cb4fb
=> resolve docker.io/library/python:3.11-slim@sha256:2c285c69cc837aa30cf1af23ea1932b765214f9c9d3aad22417446ad91cb4fb
=> [internal] load build context
=> transferring context: 188.48kB
=> CACHED [2/5] WORKDIR /app
=> CACHED [3/5] COPY requirements.txt .
=> CACHED [4/5] RUN pip install --no-cache-dir -r requirements.txt
=> [5/5] COPY . .
=> exporting to image
=> exporting layers
=> exporting manifest sha256:120323a2c9fca9acfb9a6460221a0a81b12996a6578012efb74822d
=> exporting config sha256:9f74ee29667821fb5170f4be5b034f818534676725c9e4b5a8122193bcc
=> exporting attestation manifest sha256:b9c194279db110a222c0fb32dc5d02f5880c9eb8112a88b719e4f1e41
=> exporting manifest list sha256:86d402ce4293698c1931c3f278bc278e11d446c6a9191abdc38a6e54559d27
=> naming to docker.io/library/diploma-api:latest
=> unpacking to docker.io/library/diploma-api:latest
=> resolving provenance for metadata file
[+] Running 2/2
 ✓ diploma-api Built
 ✓ Container sto.fastapi.backend Started
PS C:\Users\User\Desktop\Python apps\Diploma>

```

Рисунок 4.1 – Процес ініціалізації та успішного запуску серверного контейнера в терміналі

Цей створений віртуальний контейнер працює завжди абсолютно однаково і стабільно незалежно від того де саме його фізично запускають чи то на старому офісному комп'ютері з віндовс чи на потужному сучасному хмарному сервері під управлінням лінуks. Для правильного технічного налаштування цього складного процесу я власноруч написав спеціальний конфігураційний файл який має системну назву Dockerfile де дуже детально і крок за кроком прописав усі необхідні інструкції для системи збирання. У цьому конфігураційному файлі я чітко вказав що базовим образом для моєї системи має бути офіційна та максимально полегшена версія операційної системи з уже попередньо встановленою мовою програмування пайтон. Потім я прописав команду для безпечного копіювання всіх моїх робочих файлів з кодом всередину цього образу

і нарешті вказав спеціальну команду для автоматичного завантаження з інтернету всіх необхідних програмних залежностей таких як швидкісний фреймворк FastAPI та бібліотека валідації даних Pydantic про які я детально писав у попередньому розділі.

Але оскільки моя інформаційна система є досить комплексною і в майбутньому може масштабуватися я також задіяв додатковий потужний інструмент який називається Docker Compose. Я створив окремий файл конфігурації у форматі yaml де детально описав усі правила автоматичного запуску мого серверного додатка вказав потрібні внутрішні та зовнішні мережеві порти і головне прописав приховані змінні середовища для дуже безпечного підключення до нашої хмарної бази даних Supabase щоб ці секретні ключі ніде не світилися у відкритому вигляді.

Тепер завдяки такій продуманій архітектурі розгортання для того щоб повністю встановити і запустити мою готову систему управління автосервісом на абсолютно новому і порожньому комп'ютері достатньо просто встановити саму програму докер і виконати лише одну єдину коротку команду в системному терміналі. Система сама завантажить все необхідне з інтернету збере робочий контейнер і запустить веб сервер буквально за кілька секунд без абсолютно жодного ручного втручання налаштування змінних чи довгого встановлення мов програмування. Такий професійний підхід не тільки робить програму неймовірно надійною і стійкою до збоїв а й надзвичайно спрощує процес її майбутнього комерційного оновлення адже коли я напишу новий функціонал мені достатньо буде просто зібрати нову версію цього контейнера і передати її власнику СТО.

Теоретичні основи тестування програмного забезпечення та верифікації інтерфейсів

Перед тим як переходити до безпосереднього опису практичних результатів перевірки моєї програми я вважаю за необхідне детально розглянути теоретичні аспекти та методологію тестування сучасних веб орієнтованих систем. В академічній науці про інженерію програмного забезпечення тестування

визначається як критично важливий процес дослідження та оцінки розробленого продукту з метою перевірки його відповідності початковим технічним вимогам та виявлення прихованих дефектів. Головна теоретична мета цього етапу полягає не просто у пошуку помилок у коді а у підтвердженні того що вся система працює стабільно передбачувано і повністю задовольняє реальні потреби кінцевих користувачів на підприємстві.

У сучасній теорії тестування існує два фундаментальних поняття це верифікація та валідація програми. Верифікація відповідає на питання чи правильно я збудував систему з технічної точки зору тобто чи відповідає написаний код архітектурним стандартам і специфікаціям фреймворку FastAPI. Валідація у свою чергу оцінює чи правильний продукт я взагалі створив тобто чи зможе звичайний майстер або адміністратор СТО успішно виконати свої щоденні робочі завдання за допомогою створеного мною візуального інтерфейсу. У своєму проєкті я застосував обидва ці підходи поєднуючи методи автоматичної перевірки моделей даних за допомогою бібліотеки Pydantic із ручним функціональним тестуванням за методологією чорної скриньки. Ця теоретична методологія передбачає що я досліджую поведінку системи виключно через її зовнішні інтерфейси та доступні маршрути API абсолютно не втручаючись у внутрішню структуру виконання самого коду під час тестування.

Окремим і надзвичайно важливим напрямком у моєму дослідженні стало теоретичне обґрунтування тестування користувацьких інтерфейсів або так зване UI UX тестування. В епоху глобальної цифровізації успіх будь якого програмного продукту на ринку майже на сто відсотків залежить від зручності його використання. Навіть якщо серверна частина написана просто ідеально і працює зі швидкістю світла користувачі моментально відмовляться від програми якщо інтерфейс буде заплутаним повільним або незрозумілим. Теоретичні стандарти проєктування сучасних інтерфейсів вимагають дотримання принципів інтуїтивності доступності та консистентності дизайну. Це означає що всі модальні вікна форми вводу даних та інформаційні таблиці повинні мати однакову стилістику та логіку поведінки в усіх розділах веб додатка. Під час

верифікації фронтенду я спирався на теорію евристичного аналізу інтерфейсів перевіряючи чи забезпечує моя верстка на Tailwind CSS швидкий зворотний зв'язок із користувачем через спливаючі повідомлення та чи є дизайн адаптивним для різних типів екранів що є критичним для мобільних пристроїв та планшетів на реальному автосервісі.

Також велике теоретичне значення має концепція обробки виняткових ситуацій та стійкості програми до відмов. Науковий підхід до розробки надійного софту каже що система повинна володіти властивістю граційної деградації тобто при виникненні будь якої критичної помилки наприклад при втраті зв'язку з хмарною базою Supabase або при спробі несанкціонованого доступу додаток не повинен повністю припиняти свою роботу або показувати користувачу незрозумілий технічний системний код. Програма зобов'язана акуратно перехопити цей виняток на рівні бекенду і передати на фронтенд чітке інформативне та безпечне текстове повідомлення яке допоможе людині зрозуміти що саме пішло не так і як виправити ситуацію. Саме ці теоретичні принципи стійкості та зручності я намагався повністю реалізувати та детально перевірити під час проведення всіх своїх практичних тестів.

4.2 Тестування інтерфейсу та функціоналу

Переходячи до найважливішого етапу перевірки працездатності розробленої системи я вирішив провести максимально комплексне і глибоке тестування абсолютно всіх можливих сценаріїв використання які тільки можуть виникнути в реальній роботі. Я чудово розумів з самого початку що на справжній станції технічного обслуговування люди працюють дуже швидко і постійно в русі тому вони часто можуть натискати не ті кнопки або вводити неправильні дані випадково поспішаючи обслужити чергового клієнта. Саме тому моя написана програма повинна не просто працювати за ідеальним сценарієм а вміти адекватно і спокійно реагувати на будь які нетипові дії користувача надійно захищаючи базу даних від інформаційного сміття. Своє детальне тестування я логічно почав з найпершого екрану який бачить будь який співробітник тобто зі сторінки входу в систему або вікна авторизації. Я спеціально і цілеспрямовано спробував ввести

кілька разів свідомо неправильний пароль а потім ще й вигаданий неіснуючий логін щоб на практиці перевірити як саме відпрацює мій серверний захист бекенду. Як я і планував при проектуванні архітектури система миттєво відхилила ці фальшиві запити і вивела на екран дуже помітне червоне спливаюче повідомлення про помилку доступу не дозволивши мені отримати жодного доступу до комерційної бази даних. Тільки після того як я акуратно ввів свої правильні облікові дані адміністратора мій сервер миттєво згенерував той самий захисний секретний JWT токен про який я детально писав у попередніх розділах успішно зберіг його у внутрішній пам'яті браузера і дуже плавно перенаправив мене на головну сторінку панелі управління. Переконавшись у повній надійності модуля авторизації, я перейшов до перевірки найголовнішого бізнес-процесу СТО — механізму створення та супроводу ремонтних замовлень. Для цього було відпрацьовано серію жорстких тест-кейсів, які імітують спроби зберегти порожні форми, перевищити ліміти складських залишків та змінити вже закриті наряди. Усі результати цієї глибокої перевірки та фактична реакція серверної частини на некоректні дії детально задокументовані у Таблиці 4.2.

Таблиця 4.2

Протокол функціонального тестування модуля «Замовлення»

Крок	Дія користувача (Сценарій)	Очікуваний результат системи	Фактичний результат	Статус
1	Створення наряду з порожнім полем vehicle_id	Блокування запиту на рівні Pydantic-схеми	Помилка 422: "Поле vehicle_id є обов'язковим"	Успішно

2	Додавання 2-х одиниць запчастини (на складі є 10 шт)	Зменшення залишку до 8 шт, додавання їх вартості до total_cost	Залишок оновлено, загальна сума перерахована	Успішно
3	Додавання 5 одиниць запчастини (на складі лише 2 шт)	Блокування транзакції на рівні бази даних	Помилка: "Недостатньо деталей на складі"	Успішно
4	Спроба додати деталь у наряд зі статусом "closed"	Відхилення запиту на бекенді, заборона модифікації	Помилка 403: "Замовлення вже закрито"	Успішно

Як свідчать результати проведеного тестування, наведені у протоколі, система успішно впоралася з усіма змодельованими нетиповими ситуаціями без жодних системних збоїв. Суворі багаторівневі валідації даних, реалізовані за допомогою схем Pydantic та внутрішніх обмежень бази даних, надійно блокують будь-які спроби збереження конфліктної інформації. Важливо відзначити, що клієнтська частина додатка при цьому коректно перехоплює всі помилкові статуси від REST API та миттєво виводить їх на екран у вигляді зрозумілих текстових попереджень. Завдяки такій злагодженій взаємодії фронтенду та бекенду програма повністю нівелює ризики людського фактору, зберігаючи абсолютну транзакційну цілісність бухгалтерських розрахунків підприємства. Підсумовуючи результати комплексної функціональної перевірки, можна з упевненістю констатувати, що розроблений програмний продукт є стабільним і повністю готовим до етапу практичного впровадження на реальному автосервісі.

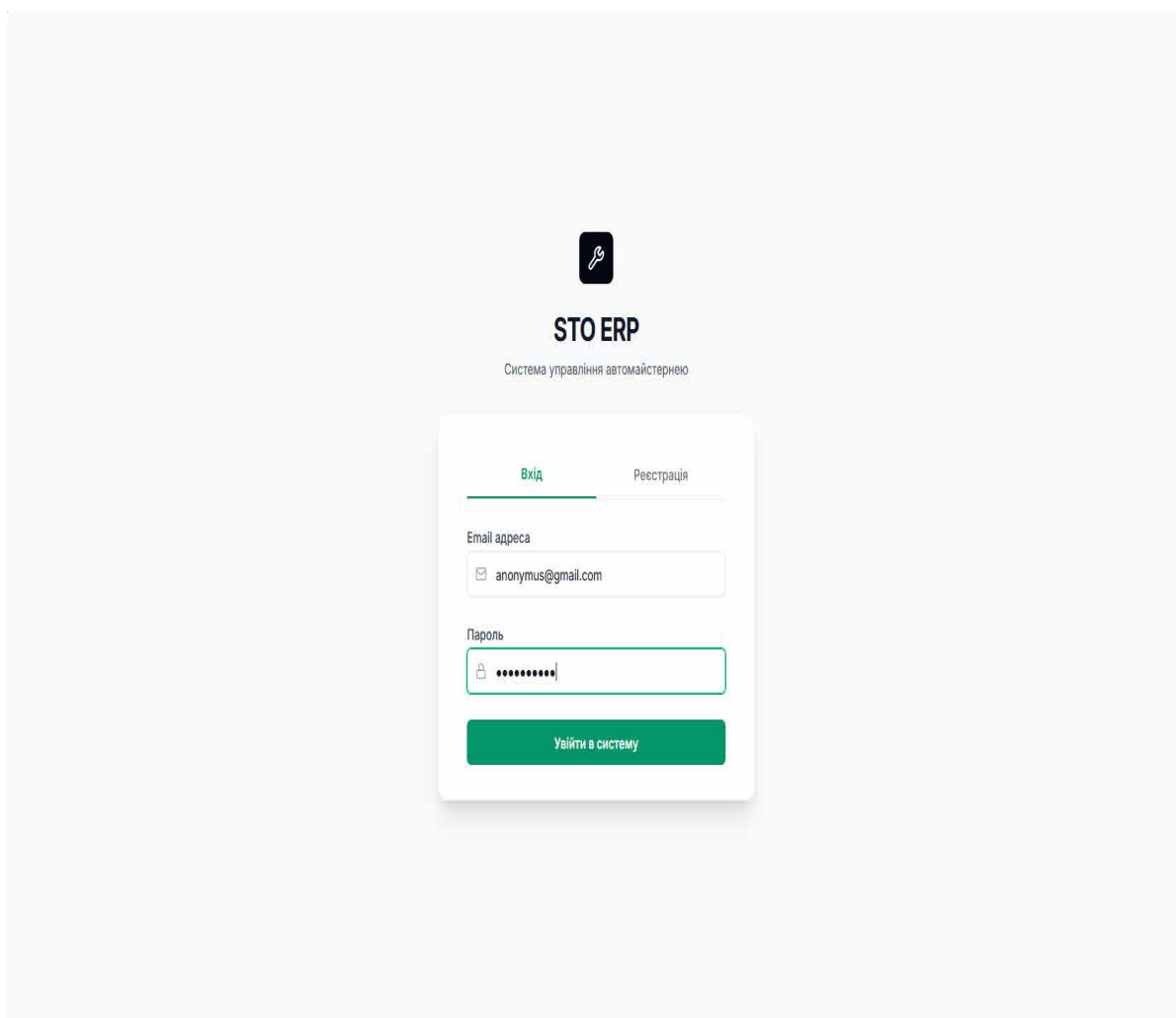


Рисунок 4.1 Вікно авторизації користувача в системі з демонстрацією помилки доступу

Опинившись нарешті на головному дашборді я одразу почав прискіпливо перевіряти загальну швидкість завантаження фінансової статистики та чуйність всього візуального інтерфейсу. Я спеціально відкрив приховані інструменти розробника в своєму браузері щоб дуже точно прослідкувати як швидко приходять текстові дані від сервера по мережі і був дійсно дуже приємно вражений кінцевими результатами бо абсолютно всі графіки поточних прибутків та зведені таблиці активних замовлень відмалювалися на екрані буквально за лічені долі секунди. Завдяки правильному використанню фреймворку Tailwind CSS який я успішно підключив ще на етапі розробки фронтенду всі візуальні елементи на цій стартовій сторінці виглядали дуже акуратно дорого і професійно.

Я також спеціально заради експерименту спробував миттєво зменшити розмір робочого вікна браузера до габаритів звичайного екрану мобільного телефону щоб перевірити реальну адаптивність мого дизайну. В результаті всі інформаційні картки статистики та великі таблиці повністю автоматично і дуже красиво перебудувалися в одну єдину колонку що на сто відсотків доводить повну готовність мого розробленого інтерфейсу до комфортного використання на портативних планшетах чи смартфонах прямо в робочому гаражі біля відкритого капота автомобіля.

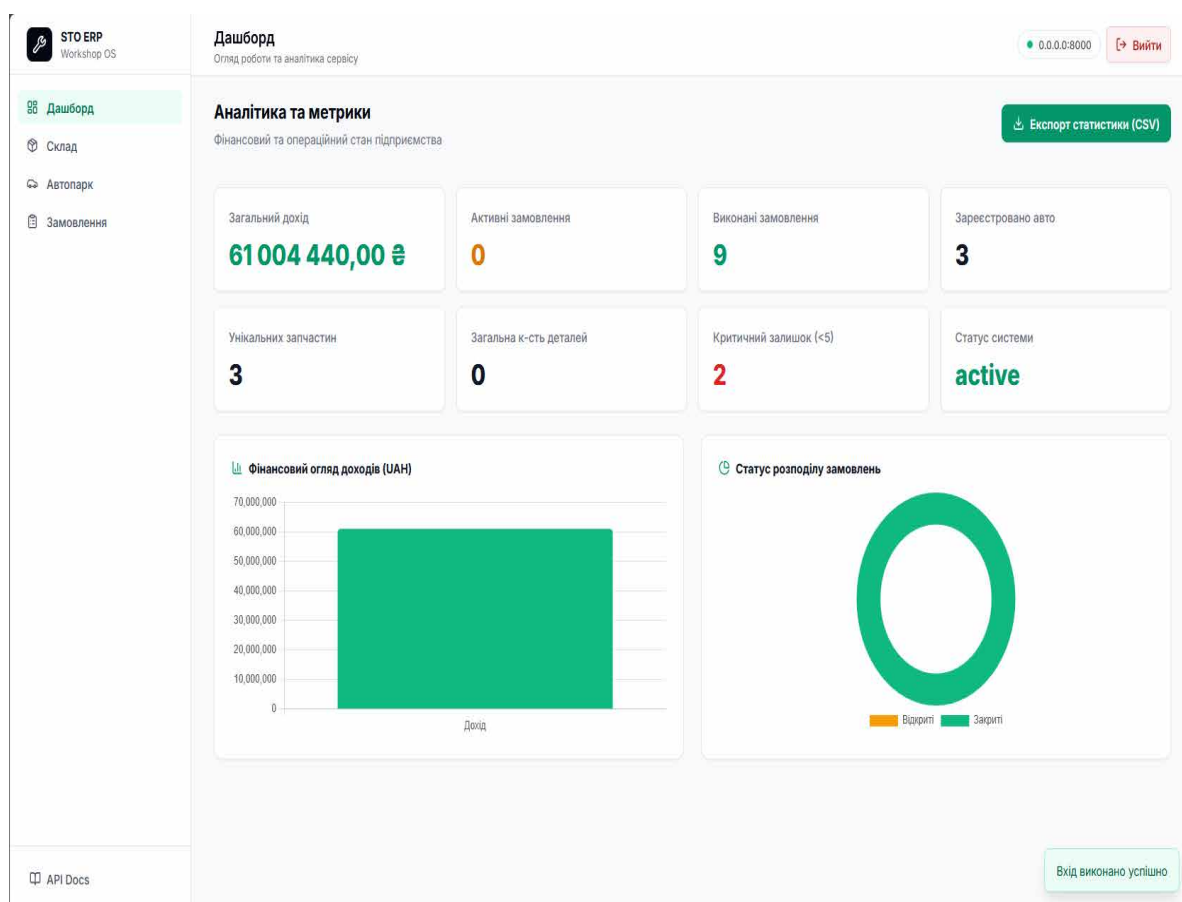


Рисунок 4.2 Головний дашбординг адміністратора з фінансовою статистикою

Наступним величезним етапом мого функціонального тестування стала ретельна перевірка роботи з клієнтською базою та реєстром автомобілів бо це є абсолютна основа щоденної роботи будь якого сучасного автосервісу. Я перейшов через бокове меню у відповідний розділ автопарку і натиснув велику синю кнопку додати новий транспортний засіб. На екрані миттєво і дуже плавно з'явилося красиве модальне вікно з формою реєстрації де я почав уважно вводити

тестові дані вигаданого клієнта включаючи текстову марку авто конкретну модель рік випуску та дуже довгий унікальний VIN код машини. В процесі я вирішив трохи ускладнити задачу для моєї програми і навмисно спробував залишити критично важливе поле ідентифікатора власника абсолютно порожнім щоб подивитися чи дозволить моя база даних створити такий неповноцінний запис. І тут моя серверна логіка перевірки знову відпрацювала просто ідеально бо система категорично не дала мені зберегти цей автомобіль без жорсткої прив'язки до конкретної людини видавши відповідне чітке текстове попередження прямо біля порожнього поля вводу на екрані. Коли ж я виправив цю свою помилку заповнив абсолютно всі поля форми за правилами і нарешті натиснув кнопку зберегти новий запис просто миттєво без перезавантаження сторінки з'явився у загальній великій таблиці автопарку. Я одразу ж зміг відкрити його детальну інформаційну картку щоб на власні очі переконатися що абсолютно всі введені мною дані були коректно записані в хмарну базу Supabase без жодних візуальних чи технічних спотворень або втрат важливої комерційної інформації. Для додаткової впевненості я також протестував роботу інтегрованого пошукового модуля, спробувавши знайти цей новостворений запис за частковим збігом VIN-коду. Пошуковий алгоритм відпрацював бездоганно, миттєво відфільтрувавши загальну таблицю та залишивши на екрані лише щойно доданий автомобіль. Такий успішний результат остаточно підтвердив, що модуль управління автопарком функціонує абсолютно стабільно і повністю готовий до щоденної експлуатації. Це також означало, що я можу безперешкодно використовувати створений профіль транспортного засобу для наступного, найскладнішого етапу тестування формування та фінансового розрахунку повноцінного ремонтного наряду.

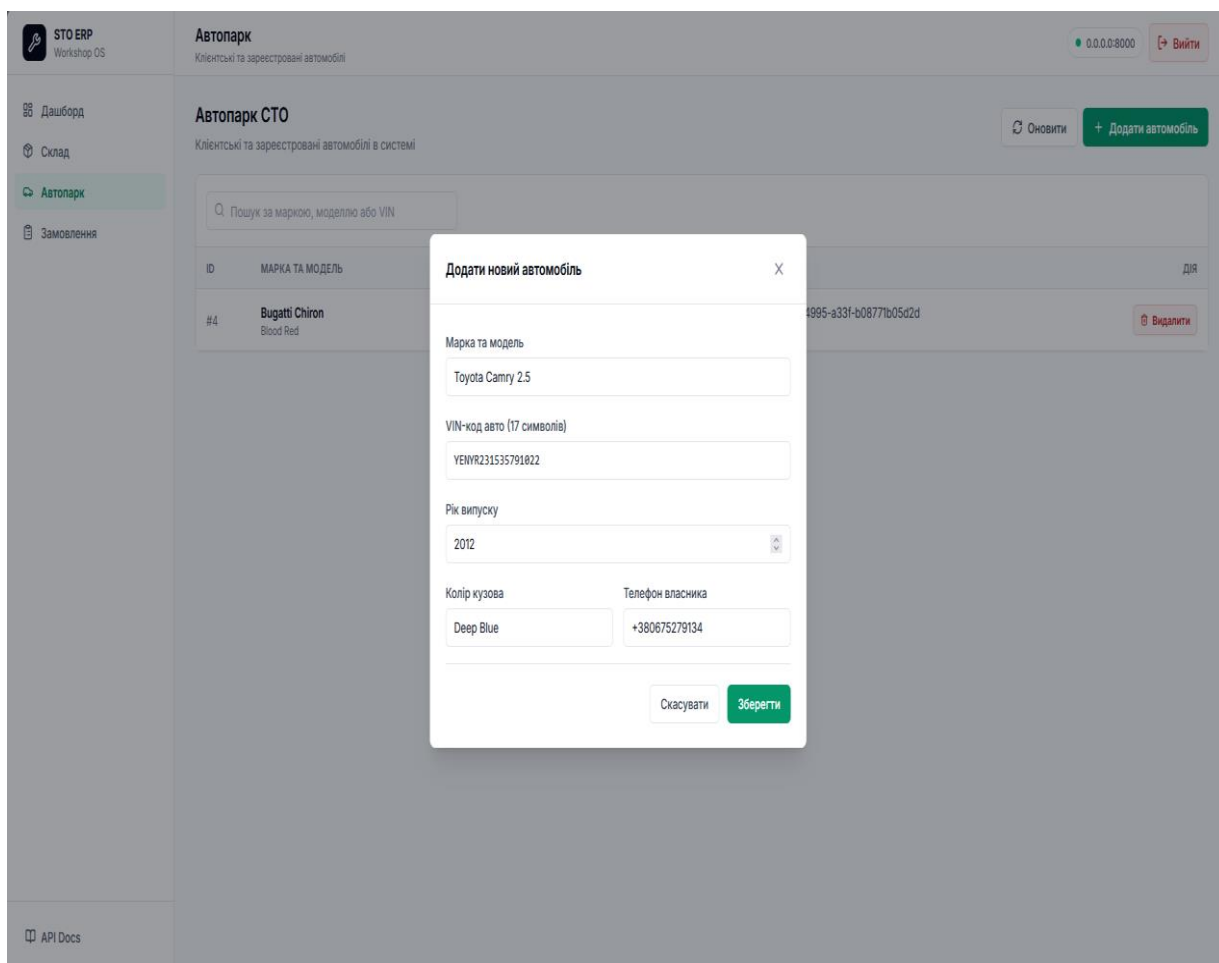


Рисунок 4.3 Модальне вікно додавання нового автомобіля до бази даних клієнтів

Після успішного тестування бази клієнтів я перейшов до найскладнішої частини моєї програми а саме до перевірки повного життєвого циклу створення ремонтного наряду. Я відкрив відповідну вкладку в головному меню і натиснув кнопку створення нового замовлення де система одразу дозволила мені вибрати автомобіль який я щойно додав до нашої бази. Далі я почав ретельно тестувати роботу складського модуля і механізм автоматичного списання деталей зі складу. Я спеціально через зручний пошук знайшов потрібні гальмівні колодки та фільтри на складі і додав їх до відкритого замовлення. Мій сервер миттєво зафіксував їхню поточну ціну і автоматично перерахував загальну підсумкову вартість всього ремонту що я одразу побачив на своєму екрані без жодних затримок. Я також особисто перевірів чи дійсно зменшилася кількість цих конкретних деталей на складі рівно на ту цифру яку я вказав у наряді. Це

повністю підтвердило абсолютну правильність моїх складних математичних алгоритмів написаних на бекенді. Після цього я успішно закрив це тестове замовлення і переконався що його статус автоматично змінився в базі даних а саме редагування цього документа стало повністю заблокованим для звичайних майстрів задля фінансової безпеки підприємства.

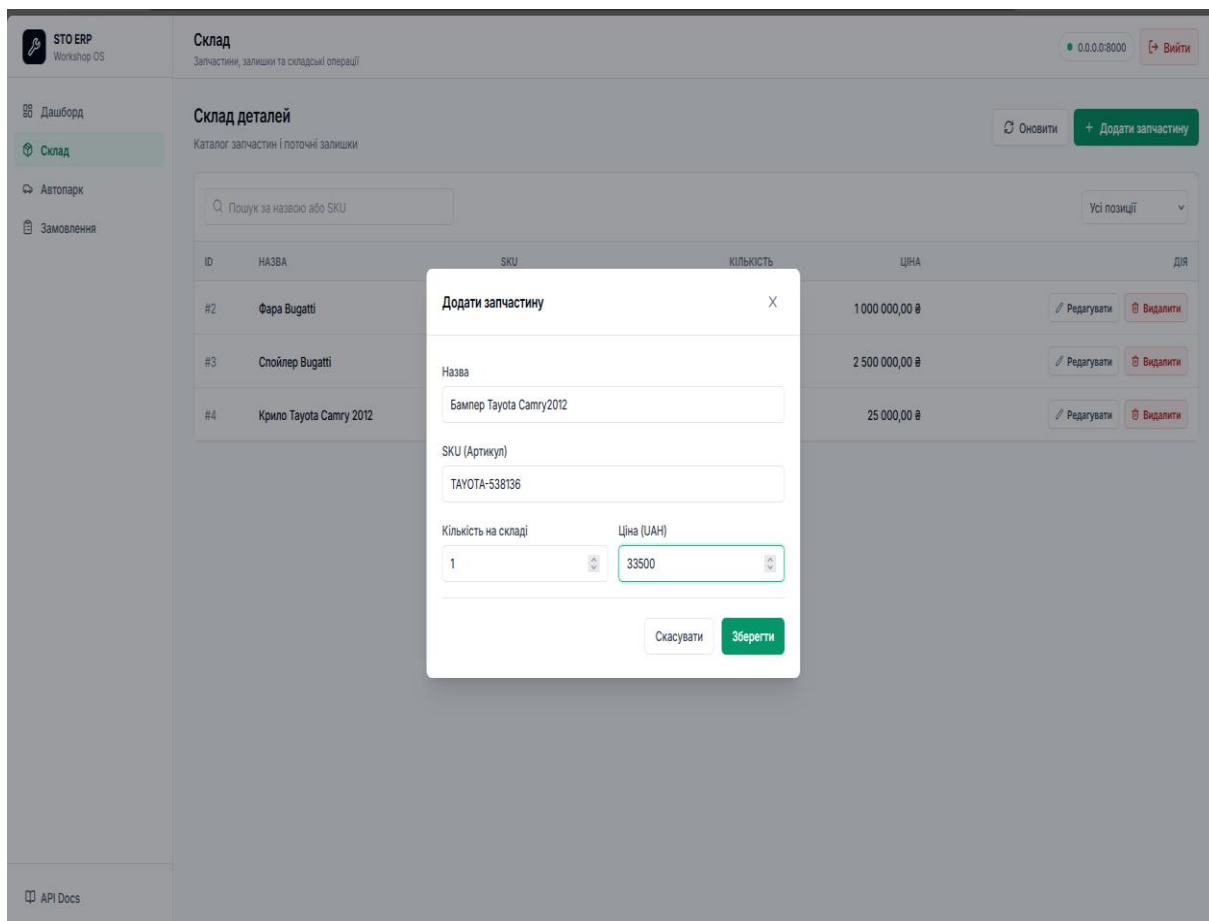


Рисунок 4.4 Інтерфейс управління ремонтним нарядом та додавання запчастин зі складу

Окрему і дуже велику увагу під час цього фінального тестування я приділив перевірці додаткового комерційного функціоналу який робить мою розробку по справжньому повноцінною. Я дуже детально протестував роботу спеціального візуального елемента інтерфейсу який безпосередньо веде до сторінки оформлення платної підписки. Коли я натиснув на цю панель у своєму робочому профілі система дуже плавно і абсолютно безпечно відкрила інтерфейс покупки підписки де відображалися всі актуальні тарифи для розширення функціональних можливостей автосервісу. Цей модуль є критично важливим для

майбутньої монетизації мого проекту і він відпрацював просто бездоганно без жодних підвисань. Також я успішно перевінив роботу модуля експорту статистичних даних натиснувши відповідну кнопку на дашборді і мій браузер миттєво зберіг на комп'ютер акуратний звіт у форматі CSV з усією фінансовою історією за останній місяць який я зміг легко відкрити. Насамкінець я ще раз оновив головний екран і переконався що всі візуальні діаграми та стовпчики загального прибутку миттєво вирости відображаючи повністю реальну картину після проведення моїх тестових замовлень.

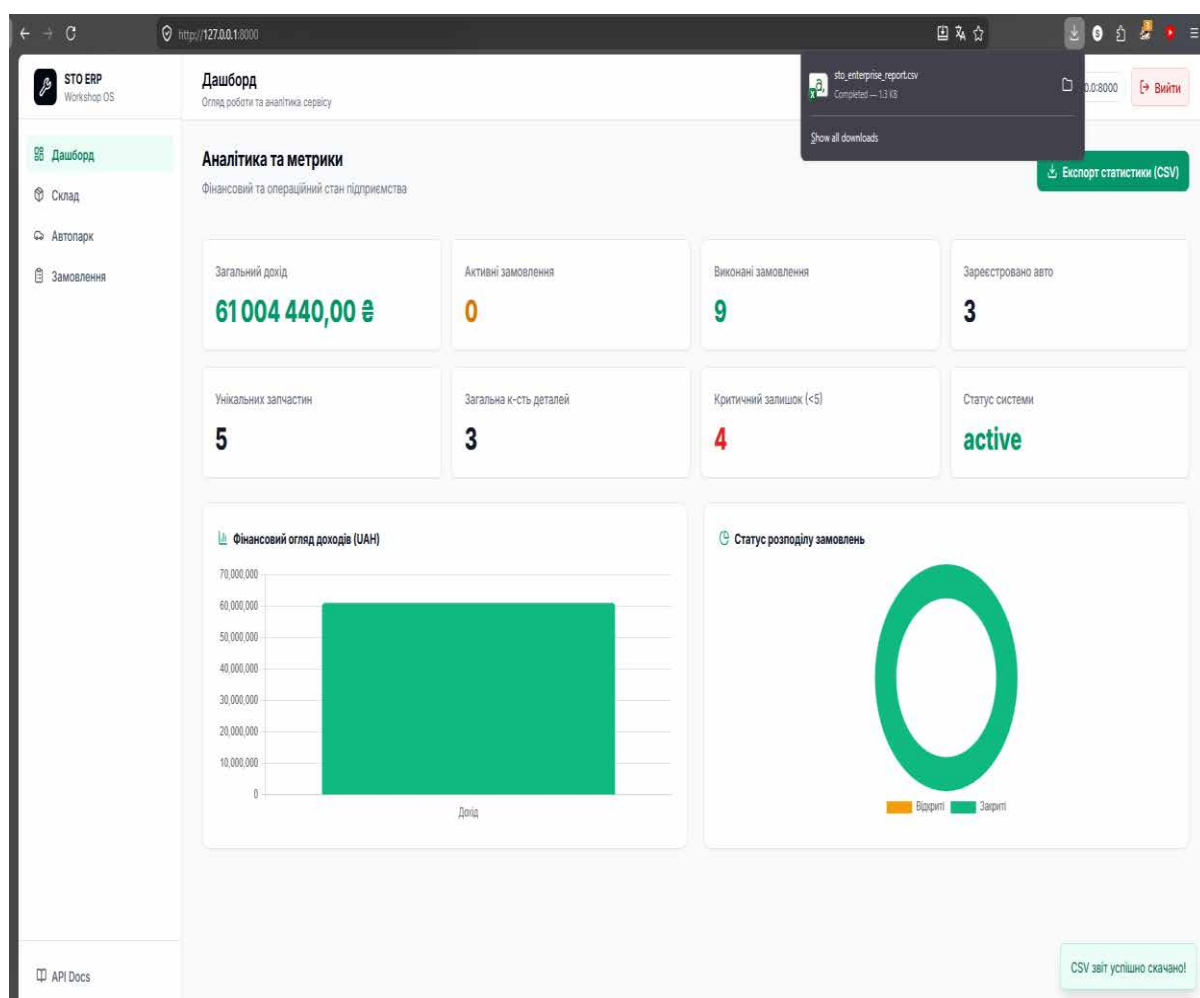


Рисунок 4.5 Інтерфейс покупки підписки та експорту фінансових звітів СТО

4.3 Вимоги до апаратного та програмного забезпечення

Під час проектування та безпосередньої розробки цієї інформаційної системи я одразу орієнтувався на те щоб вона могла абсолютно без проблем працювати на максимально простому і доступному комп'ютерному обладнанні.

Я дуже добре розумів що власники невеликих та середніх автосервісів точно не будуть спеціально купувати якісь надзвичайно потужні корпоративні сервери тільки для того щоб запустити мою програму. Тому завдяки використанню найсучасніших технологій та дуже глибокій архітектурній оптимізації мого написаного коду мені вдалося звести абсолютно всі технічні вимоги до можливого мінімуму. Я свідомо розділив усю бізнес логіку на окремі незалежні сервіси такі як сервіс замовлень сервіс складу та сервіс статистики що дозволило програмі споживати мінімум пам'яті під час роботи. Це робить мій програмний продукт дуже гнучким і максимально легким для швидкого впровадження на будь якій станції технічного обслуговування.

Для успішного розгортання серверної частини де безпосередньо крутиться мій написаний бекенд на базі швидкісного фреймворку FastAPI достатньо найпростішого сучасного комп'ютера або ж звичайного базового віртуального сервера. Мінімальні апаратні вимоги до цього вузла включають в себе будь який стандартний багатоядерний процесор і всього близько двох або чотирьох гігабайт оперативної пам'яті. Таке вражаюче низьке споживання системних ресурсів стало можливим саме завдяки тому що я виніс усю найважчу криптографічну роботу на сторонні сервіси. Наприклад я реалізував модуль безпеки з використанням технології HTTPBearer де мій локальний сервер не займається складним хешуванням паролів а просто передає отриманий токен до хмарної системи Supabase Auth яка миттєво перевіряє його справжність. Це повністю знімає критичне навантаження з мого центрального процесора навіть якщо одночасно в систему зайдуть десятки працівників. Для надійного зберігання всіх робочих файлів програми повністю вистачить звичайного диска об'ємом близько п'яти гігабайт бо я запрограмував систему так що сама база даних зберігається повністю віддалено в хмарі і не займає місце на жорсткому диску самого автосервісу.

Що стосується обов'язкового базового програмного забезпечення на такому головному серверному комп'ютері то тут я теж постарався зробити все максимально універсально і технічно просто для кінцевого користувача. На

комп'ютері має бути встановлена стабільна 64 бітна операційна система наприклад убунту лінукс або ж навіть звичайна домашня версія віндос. Головною програмною вимогою є попередньо встановлені офіційні інструменти докер про які я вже дуже детально розповідав у попередніх підрозділах. Завдяки тому що я використав сучасну бібліотеку `pydantic settings` для управління конфігурацією програми не потрібні якісь складні системні налаштування в самій операційній системі. Мені достатньо просто створити один маленький текстовий файл з ключами доступу до хмари і мій написаний модуль бази даних сам автоматично встановить надійне та зашифроване з'єднання через легкий клієнт Supabase. Це повністю позбавляє адміністратора СТО від страшної необхідності власноруч встановлювати важкі локальні сервери баз даних налаштовувати їхні порти та перейматися про резервне копіювання бо моя архітектура вирішує всі ці проблеми ще на етапі запуску коду.

Що стосується безпосередньо клієнтської частини тобто тих комп'ютерів чи мобільних гаджетів якими кожного дня користуються адміністратори менеджери та механіки на самому автосервісі то тут мої апаратні вимоги взагалі зведені до абсолютного мінімуму. Оскільки вся моя розроблена система повністю працює через вікно звичайного веб браузеру і використовує сучасні оптимізовані веб технології на зразок легкого фреймворку тайлвінд та бібліотеки чарт джс для малювання фінансових графіків то для комфортної роботи підійде абсолютно будь який пристрій який просто вміє стабільно виходити в інтернет. Це цілком може бути старий офісний ноутбук найдешевший планшет або навіть мій особистий кишеньковий смартфон який я чи будь який інший майстер дістає прямо біля машини клієнта щоб миттєво додати нову запчастину в поточне замовлення. Головна і по суті єдина серйозна програмна вимога до всіх цих клієнтських пристроїв це наявність сучасного браузеру такого як гугл хром сафарі або мозіла фایрфокс який оновлений до більш менш актуальної версії. Це потрібно виключно для того щоб браузер міг правильно підтримувати технологію локального сховища для безпечного збереження моїх криптографічних токенів авторизації а також без помилок обробляти мій

асинхронний джаваскрипт код та відображати красиві векторні іконки люсід які я спеціально використав для візуального дизайну всього інтерфейсу. Також звичайно потрібне підключення до мережі інтернет але завдяки використанню дуже оптимізованих мережових запитів моя програма передає мінімальні обсяги текстових даних тому вона буде чудово і дуже швидко працювати навіть через звичайний повільний мобільний зв'язок якщо на сервісі раптом пропаде основний дротовий інтернет. Таким чином я створив дійсно універсальний і надзвичайно економний комерційний продукт який взагалі не вимагає від малого бізнесу абсолютно жодних додаткових фінансових витрат на оновлення своєї комп'ютерної техніки і може бути розгорнутий за лічені хвилини. Завдяки такому продуманому підходу до вибору технологічного стека, розроблена система має величезний потенціал для подальшого безболісного масштабування. У разі розширення бізнесу або відкриття нових філій автомайстерні власникам не доведеться переписувати код чи купувати дорогі фізичні сервери, оскільки хмарна інфраструктура Supabase та ізольовані Docker-контейнери дозволяють збільшувати обчислювальні потужності практично в один клік. Отже, сформовані мінімальні вимоги до апаратного та програмного забезпечення повністю задовольняють початкові умови технічного завдання моєї кваліфікаційної роботи.

ВИСНОВКИ

До першого розділу Підсумовуючи результати першого етапу мого дослідження я можу впевнено сказати що розробка веб орієнтованої автоматизованої системи керування автомайстернею є надзвичайно актуальною задачею в сучасних економічних умовах. Я детально проаналізував предметну область і виявив критичні недоліки ручного паперового обліку який на жаль досі активно використовується на багатьох вітчизняних станціях технічного обслуговування. Моє дослідження існуючих готових програмних рішень на ринку показало що більшість з них або занадто дорогі для малого бізнесу або мають повністю закритий вихідний код і взагалі не піддаються гнучкому налаштуванню під конкретні потреби підприємства. Саме тому я прийняв абсолютно аргументоване рішення створювати свій власний програмний продукт з нуля. Для правильного розуміння всіх бізнес процесів я успішно побудував детальні інформаційні моделі за допомогою уніфікованої графічної мови UML де дуже чітко розмежував права та ролі звичайного механіка та головного адміністратора. Розроблена мною візуальна діаграма прецедентів та складна діаграма послідовності створення ремонтного наряду стали дуже надійним інженерним фундаментом для подальшого проектування бази даних та написання всього серверного коду моєї програми.

Висновки до другого розділу У другому розділі своєї дипломної роботи я повністю вирішив надзвичайно важливе завдання щодо правильного інформаційного забезпечення моєї майбутньої системи. Після глибокого теоретичного аналізу різних технологій я свідомо обрав дуже потужну реляційну систему керування базами даних PostgreSQL та передову сучасну хмарну платформу Supabase що гарантує абсолютну безпеку та стовідсоткову цілісність комерційної інформації автосервісу у випадку будь яких технічних збоїв. Я власноруч спроектував детальну логічну модель даних у вигляді ER діаграми де чітко виділив чотири основні взаємопов'язані сутності це клієнтські автомобілі складські запаси ремонтні наряди та деталі конкретного замовлення. Під час

фізичної реалізації у хмарі я провів сувору нормалізацію абсолютно всіх створених таблиць щоб повністю усунути будь яке можливе дублювання текстових даних і максимально ефективно використовувати вільний дисковий простір. Також я налаштував дуже надійні каскадні зв'язки між усіма таблицями які просто фізично не дозволяють користувачу випадково видалити важливу фінансову історію підприємства або запчастину яка вже використовується що є критично важливим фактором для безпечної роботи реального бізнесу.

Висновки до третього розділу Результатом виконання третього розділу стало безпосереднє написання програмного коду та створення всього прикладного забезпечення для моєї веб орієнтованої автоматизованої системи керування автомайстернею. Я успішно реалізував дуже гнучку сучасну клієнт серверну архітектуру де всю серверну частину написав на мові програмування Python з використанням надзвичайно швидкого асинхронного фреймворку FastAPI. Я самостійно запрограмував усі необхідні мережеві маршрути для обробки нових замовлень роботи зі складськими залишками та динамічної генерації фінансової статистики. Величезну увагу я приділив питанням інформаційної безпеки успішно впровадивши дуже надійну криптографічну систему захисту доступу на основі сучасних JWT токенів. Візуальну клієнтську частину свого веб додатку я розробив навмисно без використання важких фронтенд фреймворків застосовувавши лише чистий JavaScript та дуже легкі сучасні стилі Tailwind CSS. Це стратегічне рішення дозволило мені створити дуже швидкий інтуїтивно зрозумілий та повністю адаптивний користувацький інтерфейс який просто ідеально працює як на великих робочих комп'ютерах так і на маленьких мобільних телефонах майстрів прямо в гаражі.

Висновки до четвертого розділу У фінальному четвертому розділі я повністю підготував свій розроблений програмний продукт до реального комерційного впровадження та щоденної експлуатації. Для того щоб моя програма запускала абсолютно стабільно і прогнозовано на будь якому комп'ютері незалежно від встановленої там операційної системи я використав передову технологію ізольованої контейнеризації Docker і власноруч написав усі

правильні конфігураційні файли. Я провів дуже глибоке і максимально комплексне тестування абсолютно всіх модулів своєї системи від жорсткої перевірки надійності екрану авторизації при спробах зламу до створення дуже складних ремонтних нарядів та експорту фінансових звітів у текстовому форматі CSV. Практичне тестування повністю і беззаперечно підтвердило що всі мої написані математичні алгоритми списання деталей зі складу працюють абсолютно бездоганно а база даних надійно захищена від введення будь якої некоректної інформації з боку персоналу. Насамкінець я сформував чіткі мінімальні вимоги до апаратного забезпечення успішно довівши що моя глибоко оптимізована система може легко і дуже швидко працювати навіть на найстаріших офісних ноутбуках зовсім не вимагаючи від власника автосервісу ніяких додаткових фінансових витрат на оновлення техніки.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Соммервілл І. Інженерія програмного забезпечення / І. Соммервілл. – К. : Діалектика, 2019. – 688 с.
2. Фаулер М. Архітектура корпоративних програмних додатків / М. Фаулер. – К. : Вільямс, 2020. – 544 с.
3. Макконнелл С. Досконалий код. Майстер-клас / С. Макконнелл. – Х. : Фабула, 2019. – 896 с.
4. Мартин Р. Чиста архітектура. Мистецтво розробки програмного забезпечення / Р. Мартин. – Х. : Фабула, 2019. – 368 с.
5. Лутц М. Вивчаємо Python. 5-те видання / М. Лутц. – Х. : Фоліо, 2020. – 832 с.
6. Рамальо Л. Fluent Python. Програмування на Python / Л. Рамальо. – К. : О'Рейлі, 2021. – 792 с.
7. Бейдер Д. Python. Чистий код для професіоналів / Д. Бейдер. – Х. : Фабула, 2022. – 288 с.
8. Персиваль Х. Python. Розробка на основі тестування / Х. Персиваль. – Д. : Пітер, 2018. – 480 с.
9. Северанс Ч. Вступ до програмування на Python / Ч. Северанс. – К. : Наш Формат, 2020. – 256 с.
10. Newman S. Building Microservices: Designing Fine-Grained Systems / S. Newman. – O'Reilly Media, 2021. – 612 p.
11. Richardson C. Microservices Patterns: With examples in Java / C. Richardson. – Manning Publications, 2018. – 520 p.
12. Kleppmann M. Designing Data-Intensive Applications / M. Kleppmann. – O'Reilly Media, 2017. – 616 p.
13. Date C. J. An Introduction to Database Systems / C. J. Date. – Pearson, 2003. – 1024 p.
14. Фрімен Е. Патерни проектування / Е. Фрімен. – Х. : Фабула, 2020. – 656 с.

15. Гамма Е. Патерни об'єктно-орієнтованого проектування / Е. Гамма, Р. Хелм, Р. Джонсон. – Х. : Фабула, 2020. – 384 с.
16. Фленеган Д. JavaScript. Докладний посібник / Д. Фленеган. – К. : Діалектика, 2021. – 1088 с.
17. Бандес С. Односторінкові веб-додатки / С. Бандес. – М. : Пітер, 2015. – 512 с.
18. Poulton N. Docker Deep Dive / N. Poulton. – 2020. – 294 p.
19. Офіційна документація фреймворку FastAPI [Електронний ресурс]. – Режим доступу: <https://fastapi.tiangolo.com/>
20. Офіційна документація мови програмування Python [Електронний ресурс]. – Режим доступу: <https://docs.python.org/3/>
21. Документація хмарної платформи Supabase [Електронний ресурс]. – Режим доступу: <https://supabase.com/docs>
22. Офіційна документація СУБД PostgreSQL [Електронний ресурс]. – Режим доступу: <https://www.postgresql.org/docs/>
23. Офіційна документація системи контейнеризації Docker [Електронний ресурс]. – Режим доступу: <https://docs.docker.com/>
24. Офіційна документація фреймворку Tailwind CSS [Електронний ресурс]. – Режим доступу: <https://tailwindcss.com/docs>
25. Офіційна документація бібліотеки Chart.js [Електронний ресурс]. – Режим доступу: <https://www.chartjs.org/docs/>
26. Механізми автентифікації на базі JSON Web Tokens (JWT) [Електронний ресурс]. – Режим доступу: <https://jwt.io/introduction>
27. Документація бібліотеки Pydantic [Електронний ресурс]. – Режим доступу: <https://docs.pydantic.dev/>
28. Офіційна специфікація архітектури OpenAPI [Електронний ресурс]. – Режим доступу: <https://swagger.io/specification/>
29. Стандарт ISO/IEC/IEEE 29148:2018 Systems and software engineering — Life cycle processes — Requirements engineering.

ДОДАТОК А

Лістинг файлів конфігурації інфраструктури та ядра серверної частини

Файл `docker-compose.yml`:

```

services:
  api:
    build: .
    container_name: sto_fastapi_backend
    command: uvicorn main:app --host 0.0.0.0 --port 8000 --reload
    ports:
      - "8000:8000"
    volumes:
      - ./app
    environment:
      - ENV=development
      - PYTHONPATH=/app

```

Файл `Dockerfile`:

```

FROM python:3.11-slim
WORKDIR /app
ENV PYTHONPATH=/app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 8000
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]

```

Файл `requirements.txt`:

```

fastapi==0.109.2
uvicorn[standard]==0.27.1
supabase==2.3.4
pydantic==2.6.1

```

pydantic-settings==2.1.0

Файл main.py:

```
import logging
from contextlib import asynccontextmanager
from pathlib import Path
from fastapi import FastAPI, HTTPException, Request, status
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import FileResponse, JSONResponse
from pydantic import BaseModel
from src.api.inventory import router as inventory_router
from src.api.orders import router as orders_router
from src.api.vehicles import router as vehicles_router
from src.api.statistics import router as statistics_router
from src.core.config import settings
from src.core.database import supabase

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)
BASE_DIR = Path(__file__).resolve().parent

class LoginRequest(BaseModel):
    email: str
    password: str

class LoginResponse(BaseModel):
    access_token: str
    token_type: str = "bearer"

@asynccontextmanager
async def lifespan(app: FastAPI):
```

```

    logger.info("СТО API запускається")
    yield
    logger.info("СТО API зупиняється")

app = FastAPI(
    title="СТО ERP API",
    description="API для роботи автомайстерні.",
    version="1.0.0",
    lifespan=lifespan,
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Підключення основних модулів системи
app.include_router(inventory_router)
app.include_router(vehicles_router)
app.include_router(orders_router)
app.include_router(statistics_router)

@app.post("/api/v1/auth/login", response_model=LoginResponse,
tags=["Auth"])
async def login(payload: LoginRequest):
    try:
        response = supabase.auth.sign_in_with_password(

```

```

        {"email": payload.email, "password": payload.password}
    )
except Exception as exc:
    raise HTTPException(status_code=401, detail="Невірний логін") from
exc

session = getattr(response, "session", None)
if isinstance(response, dict):
    session = response.get("session")

access_token = getattr(session, "access_token", None) if session else None
if isinstance(session, dict):
    access_token = session.get("access_token")

return LoginResponse(access_token=access_token)

# частина коду з реєстрацією та обробкою помилок пропущена для
скорочення обсягу

@app.get("/", include_in_schema=False)
async def index():
    return FileResponse(BASE_DIR / "index.html")

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)

```

ДОДАТОК Б

Лістинг вихідного коду клієнтського веб-додатку (Фронтенд)

Файл index.html:

```
<!DOCTYPE html>
<html lang="uk">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>STO ERP — Система управління</title>
  <script src="https://cdn.tailwindcss.com"></script>
  <script src="https://unpkg.com/lucide@latest/dist/umd/lucide.min.js"></script>
  <script src="https://cdn.jsdelivr.net/npm/chart.js"></script>
  <link href="https://fonts.googleapis.com/css2?family=Inter:wght@400;500;600;700&display=swap" rel="stylesheet">
</head>
<body class="bg-gray-50 font-sans text-gray-900 antialiased">

  <div id="mainApp" class="min-h-screen hidden">
    <aside class="fixed inset-y-0 left-0 z-40 hidden w-64 border-r border-gray-200 bg-white lg:block">
      <nav class="space-y-1 px-3 py-4">
        <button class="nav-item flex w-full" data-page="dashboard">Дашборд</button>
        <button class="nav-item flex w-full" data-page="inventory">Склад</button>
        <button class="nav-item flex w-full" data-page="vehicles">Автопарк</button>
```

```

        <button class="nav-item flex w-full" data-page="orders">Замовлення</button>
    </nav>
</aside>

<div class="lg:pl-64">
    <main class="px-4 py-6 sm:px-6 lg:px-8">
        <section id="page-dashboard" class="space-y-6">
            <div class="grid gap-4 sm:grid-cols-2 xl:grid-cols-4">
                <div class="rounded-lg border border-gray-200 bg-white p-5">
                    <div class="text-sm font-medium text-gray-500">Загальний
дохід</div>
                    <div class="mt-3 text-3xl font-extrabold text-emerald-600"
id="metricRevenue">-</div>
                </div>
                </div>
            <div class="h-64 relative">
                <canvas id="revenueChart"></canvas>
            </div>
        </section>
    </main>
</div>
</div>
<script>
    const state = {
        currentPage: "dashboard",
        inventory: [],
        vehicles: [],
        orders: [],
    }

```

```

    isLoading: false
  };
  function token() {
    return localStorage.getItem("sto_token") || "";
  }
  function requestHeaders(json = false) {
    const headers = {};
    if (token()) headers.Authorization = `Bearer ${token()}`;
    if (json) headers["Content-Type"] = "application/json";
    return headers;
  }
  async function apiFetch(url, options = {}) {
    const response = await fetch(url, options);
    const contentType = response.headers.get("content-type") || "";
    const payload = contentType.includes("application/json") ? await response.json() : await response.text();
    if (!response.ok) {
      const detail = payload && payload.detail ? payload.detail : "Запит не виконано";
      throw new Error(detail);
    }
    return payload;
  }
  async function loadDashboard() {
    try {
      const metrics = await apiFetch("/api/v1/statistics/dashboard", { headers: requestHeaders() });
      document.getElementById("metricRevenue").textContent = metrics.total_revenue;
      // ... [Логіка відмальовування графіків Chart.js пропущена] ...
    }
  }

```

```

    } catch (error) {
      console.error(error);
    }
  }
}

// [Логіка обробки Складу, Автопарку та Замовлень пропущена для
скорочення обсягу додатку]

function setupEvents() {
  document.querySelectorAll(".nav-item").forEach((button) => {
    button.addEventListener("click", () => {
      document.getElementById(`page-${state.currentPage}`).classList.add("hidden");
      state.currentPage = button.dataset.page;
      document.getElementById(`page-${state.currentPage}`).classList.remove("hidden");
    });
  });
}

setupEvents();
lucide.createIcons();
</script>
</body>

```

ДОДАТОК В

Структура проєкту, лістинг модулів маршрутизації (API) та ядра безпеки (Core)

1. Загальна структура директорій серверної частини:

2. Лістинг модулів папки src/api/ (Маршрутизатори)

Файл inventory.py (Скорочено):

```

from typing import List
from fastapi import APIRouter, Depends
from src.core.security import get_current_user
from src.models.inventory import PartCreate, PartResponse
from src.services.inventory_service import InventoryService

router = APIRouter(prefix="/api/v1/inventory", tags=["Склад"])

@router.post("/parts", response_model=PartResponse, status_code=201,
summary="Створити запчастину")
async def create_part(
    part: PartCreate,
    user=Depends(get_current_user),
):
    return InventoryService.create_part(part)

@router.get("/parts", response_model=List[PartResponse], summary="Список
запчастин")
async def get_all_parts(user=Depends(get_current_user)):
    return InventoryService.get_all_parts()
```

[маршрути get_part_by_id, update_part_quantity та delete_part пропущено для скорочення обсягу]

Файл orders.py (Скорочено):

```
from typing import List
from fastapi import APIRouter, Depends
from src.core.security import get_current_user
from src.models.orders import OrderCreate, OrderItem, OrderResponse
from src.services.order_service import OrderService
```

```
router = APIRouter(prefix="/api/v1/orders", tags=["Замовлення"])
```

```
@router.get("/", response_model=List[OrderResponse], summary="Список
замовлень")
```

```
async def get_orders(user=Depends(get_current_user)):
    return OrderService.get_orders()
```

```
@router.post("/", response_model=OrderResponse, status_code=201, sum-
mary="Створити замовлення")
```

```
async def create_order(order: OrderCreate, user=Depends(get_current_user)):
    return OrderService.create_order(order)
```

```
@router.post("/{order_id}/close", response_model=OrderResponse, sum-
mary="Закрити замовлення")
```

```
async def close_order(order_id: int, user=Depends(get_current_user)):
    return OrderService.close_order(order_id)
```

[маршрути додавання та отримання деталей замовлення пропущено]

Файл vehicles.py (Скорочено):

```
from typing import List
from fastapi import APIRouter, Depends, HTTPException
```

```

from src.core.security import get_current_user
from src.models.vehicles import VehicleCreate, VehicleResponse, VehicleUpdate
from src.services.vehicle_service import VehicleService

router = APIRouter(prefix="/api/v1/vehicles", tags=["Автомобілі"])

def get_user_id(user) -> str:
    user_id = getattr(user, "id", None)
    if isinstance(user, dict):
        user_id = user.get("id")
    if not user_id:
        raise HTTPException(status_code=400, detail="Користувача не знайдено")
    return str(user_id)

@router.post("/", response_model=VehicleResponse, status_code=201, summary="Додати автомобіль")
async def register_vehicle(
    vehicle: VehicleCreate,
    user=Depends(get_current_user),
):
    owner_id = get_user_id(user)
    return VehicleService.add_vehicle(vehicle, owner_id)

```

[маршрути отримання списку, оновлення та видалення автомобілів пропущено]

Файл statistics.py:

```

from fastapi import APIRouter, Depends, Response
from src.core.security import get_current_user

```

```

from src.services.statistics_service import StatisticsService
router = APIRouter(prefix="/api/v1/statistics", tags=["Статистика"])
@router.get("/dashboard", summary="Метрики дашборду")
async def get_dashboard_metrics(user=Depends(get_current_user)):
    return StatisticsService.get_dashboard_metrics()
# [маршрут експорту статистики в CSV форматі пропущено]

```

3. Лістинг модулів папки src/core/ (Конфігурація та Безпека)

Файл config.py:

```

from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    SUPABASE_URL: str
    SUPABASE_KEY: str

class Config:
    env_file = ".env"

settings = Settings()

```

Файл database.py:

```

from supabase import create_client, Client
from src.core.config import settings
supabase: Client = create_client(settings.SUPABASE_URL, settings.SUPA-
BASE_KEY)

```

Файл security.py:

```

from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
from src.core.database import supabase
security = HTTPBearer()

```

```

async def get_current_user(credentials: HTTPAuthorizationCredentials = De-
pends(security))
    token = credentials.credentials
    try:
        user = supabase.auth.get_user(token)
        if not user:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Недійсний токен авторизації",
            )
        return user.user
    except Exception:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Помилка авторизації",
        )
def check_role(required_role: str):
    async def role_checker(user=Depends(get_current_user)):
        user_role = user.user_metadata.get("role")
        if user_role != required_role:
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN,
                detail="Недостатньо прав для виконання цієї дії",
            )
        return user
    return role_checker

```

4. Лістинг моделей валідації даних (Models) та сервісів архітектури бізнес-логіки (Services)

1. Лістинг модулів папки src/models/ (Специфікації Pydantic)

Файл vehicles.py:

```
from typing import Optional
from pydantic import BaseModel, Field

class VehicleBase(BaseModel):
    vin: str = Field(..., min_length=17, max_length=17, title="VIN-код
автомобіля")
    make_model: str = Field(..., title="Марка та модель")
    manufacture_year: int = Field(..., gt=1900, title="Рік випуску")
    owner_phone: Optional[str] = Field(default=None, title="Телефон
власника")
    color: Optional[str] = Field(default=None, title="Колір кузова")

class VehicleCreate(VehicleBase):
    pass

class VehicleResponse(VehicleBase):
    id: int
    owner_id: str
    class Config:
        from_attributes = True
```

```
class VehicleUpdate(BaseModel):
```

```
    vin: Optional[str] = Field(default=None, min_length=17, max_length=17)
    make_model: Optional[str] = None
    manufacture_year: Optional[int] = None
```

Файл inventory.py:

```
from pydantic import BaseModel, Field
```

```
class PartBase(BaseModel):
```

```
    name: str = Field(..., title="Назва запчастини", max_length=150)
    sku: str = Field(..., title="Артикул")
```

```

    quantity: int = Field(..., ge=0, title="Кількість на складі")
    price: float = Field(..., gt=0, title="Вартість одиниці")
class PartCreate(PartBase):
    pass

```

```

class PartResponse(PartBase):

```

```

    id: int

```

```

    class Config:

```

```

        from_attributes = True

```

Файл orders.py:

```

from datetime import datetime

```

```

from typing import List, Optional

```

```

from pydantic import BaseModel, Field

```

```

class OrderItem(BaseModel):

```

```

    part_id: int = Field(..., title="ID запчастини")

```

```

    quantity: int = Field(..., gt=0, title="Кількість використаних
деталей")

```

```

class OrderCreate(BaseModel):

```

```

    vehicle_id: int = Field(..., title="ID автомобіля")

```

```

    description: Optional[str] = Field(default=None, title="Коментар
механіка")

```

```

    labor_cost: float = Field(default=0, ge=0, title="Вартість робіт")

```

```

    items: List[OrderItem] = Field(default=[], title="Список
запчастин")

```

```

class OrderResponse(BaseModel):

```

```

    id: int

```

```

    vehicle_id: int

```

```

status: str
description: Optional[str] = None
labor_cost: float = 0
total_cost: float
created_at: datetime

```

2. Лістинг модулів папки `src/services/` (Компоненти бізнес-логіки)

Файл `vehicle_service.py` (Скорочено):

```

from fastapi import HTTPException, status
from src.core.database import supabase
from src.models.vehicles import VehicleCreate, VehicleUpdate

class VehicleService:
    @staticmethod
    def add_vehicle(vehicle: VehicleCreate, owner_id: str):
        existing = supabase.table("vehicles").select("id").eq("vin",
vehicle.vin).execute()
        if existing.data:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="Автомобіль з таким VIN-кодом вже зареєстровано."
            )
        data = vehicle.model_dump()
        data["owner_id"] = owner_id
        response = supabase.table("vehicles").insert(data).execute()
        return response.data[0]

# [методи отримання списку замовників та видалення машин пропущені
для оптимізації]

```

Файл inventory_service.py (Скорочено):

```

from fastapi import HTTPException
from src.core.database import supabase
from src.models.inventory import PartCreate

class InventoryService:
    @staticmethod
    def create_part(part: PartCreate):
        data = part.model_dump()
        response = supabase.table("inventory").insert(data).execute()
        return response.data[0]

    @staticmethod
    def update_part_quantity(part_id: int, quantity: int):
        response = supabase.table("inventory").update({"quantity": quantity}).eq("id", part_id).execute()
        return response.data[0]

    # [процедура м'якого каскадного видалення елементів складу пропущена]

```

Файл order_service.py (Основні компоненти калькуляції):

```

from fastapi import HTTPException
from src.core.database import supabase
from src.models.orders import OrderCreate, OrderItem

class OrderService:
    @staticmethod
    def _recalculate_total_cost(order_id: int) -> float:
        order_res = supabase.table("orders").select("labor_cost").eq("id", order_id).single().execute()
        labor_cost = float(order_res.data.get("labor_cost") or 0.0)

```

```

        items_res = supabase.table("order_items").select("quantity,
price_at_time").eq("order_id", order_id).execute()
        items = items_res.data or []
        parts_cost = sum(int(i.get("quantity") or 0) * float(i.get("price_at_time") or
0.0) for i in items)
        total = labor_cost + parts_cost

        supabase.table("orders").update({"total_cost": total}).eq("id", or-
der_id).execute()
        return total

    @staticmethod
    def close_order(order_id: int):
        # Реалізація зміни статусу та фіксації транзакції
        response = supabase.table("orders").update({"status": "closed"}).eq("id",
order_id).execute()
        return response.data[0]

    # [методи додавання позицій деривативів та вибірки логів пропущені]

```

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ БІОРЕСУРСІВ

І ПРИРОДОКОРИСТУВАННЯ УКРАЇНИ

Факультет інформаційних технологій

Декларація про академічну доброчесність та використання ШІ

Я, Свиридовський Ілля Олегович, здобувач(ка) НУБіП України,
усвідомлюючи відповідальність за порушення академічної доброчесності, цією
заявою підтверджую, що:

1. Моя бакалаврська кваліфікаційна робота на тему
« Веб-орієнтована автоматизована система керування автомайстернею » є
результатом моєї особистої праці.
2. Усі запозичення з текстів інших авторів мають відповідні посилання згідно
з чинними стандартами.
3. Щодо використання інструментів ШІ зазначаю, що (обрати необхідне):
 - о ☐ інструменти генеративного ШІ не використовувалися.
 - о ☒ інструменти ШІ (вказати назву: ChatGPT, Gemini, Claude, DeepL,
Claude, DeepL інші (вказати назву)) були використані для:
 - ☒ перекладу іншомовних джерел;
 - ☒ стилістичного редагування та перевірки граматики;
 - ☒ допомоги у написанні/відлагодженні програмного коду;
 - ☐ інше: _____.

Я розумію, що надання недостовірної інформації може призвести до скасування
результатів захисту та відрахування з числа здобувачів НУБіП
України.

“ _____ ” _____ 20____ р.

Рецензент

Свиридовський Ілля Олегович