Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет комп'ютерних наук
(повна назва)

Кафедра програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти перший (бакалаврський)

Програмна система моніторингу зарядних станцій електромобілів. Розробник серверної частини.
(тема)

Виконала:
студентка 4 курсу, групи ПЗПІ_20-10

Рубель Д.А..
(прізвище, ініціали)

Спеціальність 121 – Інженерія програмного забезпечення
(код і повна назва спеціальності)
Тип програми освітньо-професійна
Освітня програма Програмна інженерія
(повна назва освітньої програми)

Керівник доц. каф. ПІ Русакова Н.Є.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри _____ З.В.Дудар
(підпис) (прізвище, ініціали)

2024 р.

Харківський національний університет радіоелектроніки

Факультет комп’ютерних наук
Кафедра програмної інженерії
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 – Інженерія програмного забезпечення
Тип програми Освітньо-професійна
Освітня програма Програмна Інженерія
(шифр і назва)

ЗАТВЕРДЖУЮ:
Зав. кафедри _____
(підпис)
«___» _____ 2024 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

студентові Рублю Денису Андрійовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Програмна система моніторингу зарядних станцій електромобілів. Розробник серверної частини

Затверджена наказом по університету від 20.05. 2024р. No 471 Ст

2. Термін подання студентом роботи до екзаменаційної комісії 17.06.2024

3. Вихідні дані до роботи В програмному модулі передбачити: виконання серверної частини програмного забезпечення, забезпечення взаємодії клієнтських частин з серверною частиною, Роль back-end розробник.

4. Перелік питань, що потрібно опрацювати в роботі

Вступ, аналіз предметної галузі, формування вимог, архітектура проекту та проектування, опис прийнятих рішень, масштабованість проекту, безпека даних, опис прийнятих програмних рішень, впровадження програмного забезпечення висновки, додатки, додатки.

КАЛЕНДАРНИЙ ПЛАН

№	Назва етапів роботи	Термін виконання етапів роботи	Примітка
1	Аналіз предметної галузі	22.04.2024	*виконано*
2	Створення специфікації ПЗ	22.05.2024	*виконано*
3	Проектування ПЗ	24.05.2024	*виконано*
4	Розробка ПЗ	28.05.2024	*виконано*
5	Тестування ПЗ	30.05.2024	*виконано*
6	Оформлення пояснювальної записки	05.06.2024	*виконано*
7	Підготовка презентації та доповіді	06.06.2024	*виконано*
8	Попередній захист	12.06.2024	*виконано*
9	Нормоконтроль, рецензування	12.06.2024	*виконано*
10	Здача роботи у електронний архів	14.06.2024	*виконано*
11	Допуск до захисту у зав. кафедри	16.06.2024	*виконано*

Дата видачі завдання 8 квітня 2024р.

Студент (ка)_____ Рубель Д.А.
(підпис)

Керівник роботи _____ доц. кафедри ПІ Русакова Н.Є.

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи бакалавра, 117 сторінки, 26 рисунків, 4 додатки, 13 джерел, .

ЗАРЯДНІ СТАНЦІЇ ЕЛЕКТРОМОБІЛІВ, МОВА ПРОГРАМУВАННЯ C#, МОНІТОРИНГ, ПРОГРАМНА СИСТЕМА, СЕРВЕРНА ЧАСТИНА, ASP.NET, DOCKER, MS SQL SERVER, RABBITMQ, OCPP.

Об’єкт розробки - програмна система моніторингу зарядних станцій електромобілів, розробка серверної частини продукту.

Мета розробки - розробка серверної частини веб-додатку для забезпечення моніторингу та заощадження електроенергії на зарядних станціях електромобілей.

Методи розробки, що використовуються - це мікросервісна архітектури з використанням ASP.NET, RabbitMQ як брокера повідомлень, API Gateway на базі Ocelot, gRPC для комунікації між сервісами та протоколу OCPP на основі WebSocket для взаємодії з зарядними станціями. Для налаштування процесів CI/CD використовуються Docker, Azure Kubernetes, Azure DevOps Pipelines та Azure Container Registry.

Результатом роботи є готова серверна частина, яка забезпечує надійну та безпечну роботу програмної системи моніторингу зарядних станцій для електромобілів. Створення ефективної системи з компонентами для обробки даних, інтерфейсами для взаємодії та модулями безпеки.

Готова серверна частина програмної системи моніторингу зарядних станцій для електромобілів представляє собою потужний інструмент для власників та операторів депо зарядних станцій. Завдяки цій системі користувачі отримують можливість гнучко налаштовувати профілі споживання електроенергії, що дозволяє оптимізувати витрати та ефективно використовувати ресурси.

ASP.NET, BACK-END, CHARGING STATIONS OF ELECTRIC VEHICLES, DOCKER, MONITORING, MS SQL SERVER, OCPP, PROGRAMMING LANGUAGE C#, RABBITMQ, SOFTWARE SYSTEM.

The object of development is a software system for monitoring electric vehicle charging stations, developing the server part of the product.

The purpose of the development is to develop the server part of a web application to provide monitoring and saving electricity at electric vehicle charging stations.

The development methods used are microservice architecture using ASP.NET, RabbitMQ as a message broker, Ocelot-based Gateway API, gRPC for inter-service communication, and WebSocket-based OCPP protocol for interaction with charging stations. Docker, Azure Kubernetes, Azure DevOps Pipelines and Azure Container Registry are used to configure CI/CD processes.

The result is a ready-made server part that provides reliable and safe operation of the software monitoring system of charging stations for electric vehicles. Create an efficient system with data components, interfaces for interaction and security modules.

The finished server part of the charging station monitoring software system for electric vehicles is a powerful tool for owners and operators of charging station depots. Thanks to this system, users can flexibly configure power consumption profiles, which allows them to optimize costs and use resources efficiently.

Я, Рубель Денис Андрійович , студент гр. ПЗПІ-20-10, здобувач вищої освіти на першому (бакалаврському) рівні кафедри «Програмна інженерія», заявляю: моя кваліфікаційна робота на тему «Програмна система моніторингу зарядних станцій електромобілів. Розробник серверної частини», що буде представлена в екзаменаційну комісію для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу ElAr KhNURE. Усі запозичення з друкованих та електронних джерел мають відповідні посилання. Я ознайомлений з діючим положенням «Про

протидію академічному плагіату в ХНУРЕ», згідно з яким виявлення плагіату є підставою для відмови до допуску кваліфікаційної роботи до захисту та застосування дисциплінарних заходів

ЗМІСТ

ВСТУП

Розвиток технологій у сучасному світі неухильно веде до змін у різних сферах нашого життя, зокрема, у транспортній індустрії. Одним із ключових напрямків цього розвитку є впровадження електромобілів, які стають все більш популярними серед власників автотранспорту. За останні кілька років спостерігається збільшення кількості зарядних станцій для електромобілів, які мають велике значення для забезпечення мобільності та екологічної чистоти у містах.

У зв'язку з цим, у рамках даної роботи була проведена розробка серверної частини програмної системи моніторингу зарядних станцій електромобілів. Ця система призначена для надання можливостей власникам та операторам депо зарядних станцій для гнучкого налаштування профілів споживання електроенергії, моніторингу стану станцій та процесів заряджання зарядок, а також аналізу статистичних даних. Такий комплексний підхід до управління зарядними станціями сприятиме їх ефективній роботі, підвищенню рівня обслуговування та зниженню навантаження на електромережі.

У цьому контексті, робота має на меті детальний аналіз вимог до системи моніторингу, проектування та розробку серверної частини, валідацію та тестування розробленого продукту з метою підтвердження його ефективності та можливостей для використання у практичних умовах. Дана робота є актуальною та перспективною у контексті сучасних тенденцій розвитку інфраструктури для електромобілів та внесе вагомий внесок у підвищення ефективності їх використання.

Ця робота ставить перед собою ряд важливих завдань та мету, яка полягає у розробці серверної частини програмної системи моніторингу зарядних станцій для електромобілів. Головна мета дослідження — створення ефективного інструменту управління зарядними станціями, який відповідатиме сучасним вимогам та враховуватиме потреби власників та операторів таких станцій.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної галузі

Аналіз предметної області зарядних станцій для електромобілів включає в себе докладне вивчення існуючих технологій, стандартів, тенденцій ринку та вимог користувачів. Предметна область зарядних станцій є динамічною та постійно розвивається, особливо в контексті поширення електромобілів та зростання популярності зелених технологій.

Один із основних аспектів аналізу — це технічні рішення зарядних станцій.

Існують різні типи зарядних станцій, такі як звичайні зарядки для дому, швидкі зарядки для комерційних майданчиків та супершвидкі зарядки для автомагістралей. Кожен тип має свої технічні характеристики, такі як потужність, тип підключення (AC або DC), можливість комунікації з сервером для моніторингу тощо.

Також важливим аспектом є стандарти зарядних станцій. Наприклад, стандарти з'єднання, такі як CCS, CHAdeMO та Type 2, визначають типи роз'ємів та протоколи комунікації, що використовуються для заряджання електромобілів. Розуміння цих стандартів є важливим для правильного вибору зарядної станції та забезпечення сумісності з різними моделями автомобілів [1].

Ринок зарядних станцій для електромобілів переживає швидкий ріст, особливо з урахуванням зростання популярності електромобілів та зелених технологій. Цей ринок відзначається високою конкурентністю, оскільки на ньому працюють багато постачальників та виробників, що стимулює постійне вдосконалення технологій та зниження цін.

Технологічні інновації в ринку зарядних станцій включають розробку швидких та супершвидких зарядних станцій, впровадження "розумних" функцій та покращення системи підтримки різних стандартів зарядки. Державні заходи також впливають на ринок, зокрема, стимулюючи розвиток інфраструктури через субсидії, зниження податків та встановлення норм [2].

Попит на інтегровані рішення зарядних станцій зростає, особливо серед власників бізнесу та організацій, які цінують зручність та ефективність таких рішень. Усі ці фактори вказують на те, що ринок зарядних станцій для електромобілів є перспективним та динамічно розвивається, пропонуючи широкі можливості для виробників, постачальників та користувачів. Нарешті, в аналізі предметної області не можна оминути важливість інформаційних технологій. Розвиток ІТ-рішень у сфері зарядних станцій дозволяє впроваджувати системи моніторингу, керування та аналізу даних, що покращує ефективність та надійність роботи зарядних станцій.

1.2 Виявлення проблем та актуалізація рішень

Виявлення проблем є ключовим етапом у процесі розробки програмної системи моніторингу зарядних станцій для електромобілів. Основним завданням цього підрозділу є ідентифікація потенційних проблем та визначення напрямків подальшої роботи для їх вирішення.

- різні типи зарядних станцій використовують різні стандарти зарядки, що може призводити до проблем з сумісністю між станціями та електромобілями. Це може потребувати розробки універсальних рішень для забезпечення сумісності з різними стандартами [3].
- важливо забезпечити стійкий та надійний зв'язок між зарядними станціями та серверною частиною системи моніторингу. Проблеми зі зв'язком можуть призвести до некоректного моніторингу стану станцій та процесів заряджання.
- збір та обробка великого обсягу даних, таких як інформація про стан станцій та енергоспоживання, вимагає високого рівня захисту даних. Проблеми з безпекою можуть стати причиною витоку конфіденційної інформації або недозволених втручань у роботу системи.
- важливо забезпечити ефективне використання електроенергії під час заряджання електромобілів. Проблеми з енергоефективністю можуть

призвести до зайвого споживання ресурсів та збільшення витрат для власників зарядних станцій.

– необхідно забезпечити ефективну систему підтримки та обслуговування для вирішення можливих проблем користувачів та операторів зарядних станцій. Недостатня підтримка може призвести до невдоволення користувачів та зниження рівня сервісу.

Ретельне виявлення цих проблем дозволить розробити ефективні стратегії для їх вирішення під час подальшої роботи над системою моніторингу зарядних станцій для електромобілів.

На ринку зарядних станцій для електромобілів існує значна кількість конкурентів. Це включає виробників зарядних станцій, розробників програмного забезпечення для моніторингу та керування, а також постачальників послуг інтеграції та обслуговування систем зарядних станцій. Конкурентна боротьба на цьому ринку сприяє постійному вдосконаленню технологій та покращенню обслуговування для кінцевих користувачів.

Конкурентами для нашого продукту є:

– Ampecp [4]: це компанія, яка спеціалізується на моніторингу електротранспорту. Вони розробляють та постачають рішення для відстеження руху електричних транспортних засобів, забезпечуючи точний аналіз даних щодо шляхів, часу руху та стану енергоспоживання. Ampecp допомагає організаціям ефективно використовувати та управляти своєю електротранспортною інфраструктурою.

– ChargeLab [5] - це компанія, яка спеціалізується на розробці та постачанні рішень для зарядки електромобілів. Вони надають комплексні програмні рішення для управління зарядною інфраструктурою, включаючи управління зарядними станціями, платіжні системи, моніторинг енергоспоживання та аналітику. ChargeLab сприяє розвитку електромобільної індустрії та забезпечує зручний доступ до зарядної інфраструктури для користувачів електромобілів.

– GreenFlux [6] - це компанія, яка спеціалізується на розробці інноваційних рішень для управління зарядною інфраструктурою для електромобілів. Вони пропонують інтегровані платформи для керування зарядними станціями, віддаленого моніторингу, оптимізації енергоспоживання та управління мережею зарядних станцій. GreenFlux сприяє розвитку зеленої мобільності та забезпечує ефективне використання зарядних інфраструктурних ресурсів.

Недоліки кожної з компаній:

– хоча Ampecp спеціалізується на моніторингу електротранспорту, можуть виникати проблеми з точністю даних або інтеграцією їх систем з різними типами електротранспорту та мережами.
– недоліками ChargeLab можуть бути висока вартість їхніх рішень, складність інтеграції з існуючими системами управління та обмежені можливості адаптації під конкретні потреби.
– хоча GreenFlux відома своїми інноваційними рішеннями, серед недоліків можна відзначити високу вартість, складність інтеграції з різними технологіями та обмежену службу підтримки.

Ці недоліки можуть бути вирішені через додаткові інвестиції в дослідження та розвиток, покращення технічної підтримки та співпрацю з клієнтами для розробки більш гнучких та ефективних рішень.

Програмна система для моніторингу зарядних станцій електромобілів має свої переваги, порівняно з продуктами компаній Ampecp, ChargeLab та GreenFlux:

– інтегрована функціональність: програмна система може поєднувати в собі функції моніторингу (як у Ampecp), управління зарядною інфраструктурою (як у ChargeLab) та оптимізації енергоспоживання (як у GreenFlux), що робить її більш комплексним та універсальним рішенням.
– гнучкість та адаптивність: програмна система може бути налаштована під конкретні потреби та умови клієнта, що робить її більш гнучкою у використанні порівняно з фіксованими продуктами від окремих компаній.

– ефективність витрат: інтегрована програмна система може допомогти уникнути додаткових витрат на інтеграцію та обслуговування різних продуктів від різних вендорів.
– оновлення та підтримка: відповідальність за оновлення та підтримку системи лежить на розробнику програмної системи, що зменшує навантаження на клієнта і забезпечує актуальність та надійність рішення.

Ці переваги можуть бути важливими для вибору програмної системи управління електротранспортом у рамках роботи, оскільки вони враховують інтегрований та комплексний підхід до управління, що відповідає сучасним вимогам та тенденціям у сфері зеленої мобільності.

1.3 Постановка задачі

Задача цієї роботи полягає у розробці серверної частини програмної системи моніторингу зарядних станцій для електромобілів. Перед постановкою завдань слід провести ретельний аналіз вимог, що включає в себе як функціональні, так і нефункціональні аспекти. Функціональні вимоги описують конкретні можливості системи, наприклад, моніторинг стану зарядних станцій, управління процесами заряджання, аналіз та візуалізація статистичних даних, підтримка різних типів зарядних станцій тощо. Нефункціональні вимоги описують якісні аспекти, такі як продуктивність, надійність, безпека даних, ергономіка інтерфейсу та інші параметри, що впливають на зручність та ефективність використання системи.

Далі, поставляються завдання з визначення ресурсів та технологій. Це означає визначення необхідного обладнання та програмного забезпечення для реалізації системи, також визначення технологій, які будуть використовуватися, включаючи мови програмування, бази даних, протоколи комунікації тощо. При цьому важливо врахувати питання сумісності та інтеграції з існуючими системами, які можуть використовуватися в контексті зарядних станцій та електромобілів.

Окремим етапом є планування робіт та графік виконання завдань. Тут визначається послідовність дій, розподіл завдань між учасниками команди, визначення термінів виконання, а також планування резервного часу на випадок

можливих затримок чи проблем у розробці. Також важливо розробити план тестування та валідації, щоб перевірити правильність реалізації функціональності та відповідність системи вимогам.

Завершальним етапом постановки задачі є визначення критеріїв успіху та очікуваних результатів. Це включає в себе не лише технічні показники, а й оцінку впливу системи на підвищення ефективності використання зарядних станцій та покращення досвіду користувачів. Також важливо забезпечити план подальшого супроводу та підтримки розробленої системи після завершення дипломної роботи.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

2.1 Функціональні вимоги

Функціональні вимоги до системи визначають конкретні функції, які система повинна виконувати для забезпечення заданих функціональностей та відповідності потребам користувачів і бізнесу. Основна мета функціональних вимог - це описати, що система повинна робити, яким чином і які результати вона має виробляти.

- реєстрація та аутентифікація користувачів. Система повинна забезпечувати можливість реєстрації та входу в систему для користувачів з різними ролями, та надавати відповідний доступ до функціоналу відповідно до їхніх прав доступу.
- керування депо. Система має підтримувати створення, редагування та видалення депо, а також надавати можливість управління його параметрами.
- керування зарядними станціями та роз’ємами для зарядки в межах депо. Вимога передбачає можливість створення, редагування та видалення зарядних станцій та роз’ємів, а також керування їхніми параметрами.
- налаштування параметрів споживання енергії. Система повинна дозволяти налаштовувати параметри споживання енергії як для депо в цілому, так і для окремих зарядних станцій.
- реагування на перевищення лімітів енергії. Система мусить виявляти та реагувати на перевищення встановлених лімітів споживання енергії, сповіщаючи користувача через інтерфейс та електронну пошту.
- отримання статистики щодо споживання енергії. Вимога передбачає збір та відображення статистичних даних про споживання електроенергії користувачами.
- налаштування доступу користувачів до депо. Система має надавати можливість налаштовувати рівні доступу користувачів до функціоналу депо.

- підтримка протоколу OCPP. Система повинна використовувати протокол OCPP для взаємодії з зарядними станціями.
- бронювання роз’ємів зарядних станцій. Система повинна надавати можливість користувачам бронювати роз’єми зарядних станцій для зарядки електромобілів.
- вибір мови інтерфейсу. Система має дозволяти користувачам обирати мову інтерфейсу між українською та англійською.

Ці функціональні вимоги визначають основні можливості та функціонал, які має підтримувати система моніторингу зарядних станцій електромобілів для задоволення потреб користувачів та забезпечення ефективності використання.

2.2 Нефункціональні вимоги

Нефункціональні вимоги до системи - це вимоги, які не стосуються конкретної функціональності програми, але визначають її якості, характеристики та параметри роботи. Ці вимоги описують "якість" системи, зокрема її продуктивність, безпеку, надійність, швидкодію, масштабованість, доступність та інші аспекти, які впливають на ефективність та задоволення користувачів.

Швидкодія: система повинна працювати швидко та ефективно, забезпечуючи мінімальну затримку при обробці та відповіді на запити користувачів. Для досягнення цієї мети, важливо оптимізувати процеси обробки даних та використовувати потужні алгоритми для швидкого виконання завдань.

Масштабованість: система повинна бути здатною масштабуватися для обробки збільшеної кількості запитів та користувачів без втрати продуктивності. Вона повинна підтримувати горизонтальне масштабування, що дозволить додавати нові сервери для розподіленої обробки завдань. Це забезпечить надійну та ефективну роботу системи навіть при зростанні обсягів даних та навантаження.

Надійність: система повинна бути надійною та стійкою до відмов. Вона повинна мати механізми для виявлення та відновлення випадків відмов, а також забезпечувати резервне копіювання даних для запобігання їх втраті. Це дозволить

уникнути втрати даних та забезпечити безперервну доступність системи для користувачів.

Безпека: система повинна мати механізми захисту від несанкціонованого доступу до даних та користувацької інформації. Вона повинна використовувати автентифікацію та авторизацію користувачів, а також захищені канали комунікації для передачі даних. Це забезпечить конфіденційність та цілісність інформації в системі.

Сумісність: система повинна бути сумісною з існуючими системами та протоколами обміну даними. Вона повинна забезпечувати інтеграцію з бухгалтерськими та фінансовими системами, а також з іншими програмними рішеннями, що використовуються в організації. Це дозволить зберігати синхронізовану та узгоджену інформацію між різними системами.

Легкість використання: система повинна мати зрозумілий та інтуїтивний інтерфейс користувача, який дозволить співробітникам легко та швидко збирати та обробляти чеки. Вона повинна також підтримувати механізми імпорту та експорту даних для зручності користувачів. Це сприятиме швидкому навчанню та ефективному використанню системи користувачами.

3 АРХІТЕКТУРА ПРОЕКТУ ТА ПРОЕКТУВАННЯ

3.1 Основний сервіс з бізнес логікою

З розвитком сучасних технологій та зростанням складності програмного забезпечення стає все важче та складніше зберігати його ефективність, масштабованість та гнучкість. У цьому контексті виникає потреба у нових підходах до розробки, які дозволяють забезпечити високу якість та швидкість впровадження програмних продуктів.

Проект, що присвячений розробці серверної частини, не є винятком у цьому контексті. Щоб забезпечити оптимальну продуктивність та гнучкість цієї частини системи, досить важливим стає використання мікросервісної архітектури. У наступному тексті ми розглянемо переваги та особливості мікросервісів у контексті розробки серверних додатків, щоб краще зрозуміти, як цей підхід може зробити наш проект більш ефективним та зручним у плані управління, масштабованості та підтримки.

Мікросервіси — це архітектурний підхід до розробки програмного забезпечення, в якому програма розбивається на невеликі, незалежні компоненти, що називаються мікросервісами. Кожен мікросервіс виконує конкретну функцію або набір функцій і має власний інтерфейс програмування додатків (API). Основна ідея мікросервісної архітектури полягає в тому, щоб розбити складну систему на менші, керовані окремо компоненти, що спрощує розробку, впровадження та підтримку програмного забезпечення.

Концепція мікросервісної архітектури полягає у дизайні архітектури програмної системи таким чином, що функціональність розподіляється між сервісами. Кожен із сервісів виконується незалежно і у власному потоці. Тож мікросервіси не повинні мати ніяких залежностей від платформ, інструментів або мов програмування і повинні бути розроблені за допомогою інструментів, які найкраще підходять для конкретної цілі. Найбільш імовірно, що кожен з сервісів використовується декількома різними клієнтами, серед яких інші мікросервіси системи, що розроблені і працюють на різних платформах. Тому важливо використовувати комунікаційні протоколи та засоби, що можуть ефективно

використовуватися як мобільним додатком чи web застосунком, так і компонентами системи. [7]

Мікросервіси відкривають широкі можливості для розробників, спрощуючи взаємодію з кодом та забезпечуючи більшу гнучкість та швидкість у впровадженні нового функціоналу. Їх головна перевага полягає в тому, що вони дозволяють розбити складну систему на невеликі, самодостатні компоненти, кожен з яких може функціонувати індивідуально. Це не лише полегшує розробку, а й сприяє покращенню масштабованості системи. Крім того, мікросервіси спрощують розподілену роботу між командами розробників, дозволяючи їм працювати над окремими компонентами паралельно та ефективно координувати свою діяльність.

Такий підхід також забезпечує більшу гнучкість в управлінні версіями та швидку адаптацію до змін у бізнес-потребах [8].

- розбиття на невеликі, самодостатні компоненти. Підходячи до розробки як до набору окремих мікросервісів, розробники отримують можливість розбивати систему на менші, керовані окремо компоненти. Це дозволяє зосередитися на конкретному функціоналі кожного сервісу без великих зусиль зберігати та розуміти увесь код системи як єдину цілісність.
- гнучкість у розробці та впровадженні нового функціоналу. Будучи незалежними один від одного, мікросервіси дозволяють розробникам працювати над різними компонентами системи паралельно. Це значно прискорює час розробки та впровадження нового функціоналу, оскільки зміни в одному сервісі не впливають на решту системи.
- масштабованість. Оскільки кожен мікросервіс може бути масштабований окремо, це дозволяє оптимізувати використання ресурсів та забезпечувати високу продуктивність системи навіть при зростанні навантаження. Крім того, масштабованість мікросервісів робить їх ідеальними для великих проектів, які потребують постійного розширення та оновлення.
- легка заміна та оновлення. Оскільки кожен мікросервіс виконує обмежену кількість функцій і має власний API, заміна або оновлення окремих компонентів не впливає на решту системи. Це дозволяє швидко реагувати

на зміни та вимоги, а також полегшує підтримку та управління версіями програмного забезпечення.

– розподілена робота команд. Мікросервісна архітектура дозволяє розподілити розробку між декількома невеликими командами. Кожна команда може відповідати за розробку та підтримку конкретного сервісу, що сприяє зниженню комунікаційних накладних та підвищенню продуктивності.

Недоліки мікросервісної архітектури включають складність управління багатьма сервісами, необхідність ефективного моніторингу та керування конфігураціями, а також проблеми з цілісністю та консистентністю даних при великій кількості мікросервісів. Крім того, мікросервіси можуть вимагати більше ресурсів для управління та розгортання порівняно з монолітною архітектурою, а також вони можуть бути складнішими у відлагодженні та виправленні помилок через їх розподілену природу.

Щоб запобігти складностям у роботі з мікросервісами, слід дотримуватися кількох важливих принципів:

– чітка архітектура. Докладне планування архітектури системи. Визначення меж між мікросервісами та їх взаємодію. Гарно спроектована архітектура допомагає уникнути змішування відповідальностей та зберегти чіткість у системі.

– ефективне моніторинг та логування. Забезпечення високого рівня моніторингу кожного мікросервісу. Використання спеціальних інструментів для збору метрик, аналізу логів та виявлення проблем.

– керування конфігураціями. Використання системи управління конфігураціями для ефективного керування налаштуваннями кожного сервісу. Це дозволяє уникнути конфліктів та забезпечити стабільну роботу системи.

– автоматизація інфраструктури. Використання інструментів для автоматизації розгортання та управління інфраструктурою, такі як Docker.

Це спрощує управління сервісами та зменшує ризик помилок під час розгортання.

- тестування та контейнеризація. Ретельне тестування кожного мікросервісу та використання контейнеризації для ізоляції сервісів. Це допомагає уникнути непередбачених проблем при взаємодії між компонентами.
- документація та комунікація. Розробка та підтримка докладної документацію для кожного мікросервісу. Забезпечення ефективної комунікації між командами розробників для уникнення недорозумінь та покращення спільної роботи.

Дотримання цих практик допомогає зменшити складність у роботі з мікросервісами та забезпечує більш ефективне управління та розвиток системи.

Мікросервісна архітектура на мові C# з використанням платформи .NET 8 та фреймворку ASP.NET Core надає гнучкість, продуктивність та масштабованість для розробки серверних додатків. ASP.NET Core є одним з найпопулярніших інструментів для розробки веб-додатків на C#, а .NET 8 пропонує широкі можливості для створення мікросервісних систем.

Основні переваги використання мікросервісів на .NET 8 та ASP.NET Core включають:

Крос-платформеність: .NET 8 та ASP.NET Core підтримують роботу на різних операційних системах, таких як Windows, Linux та macOS, що дозволяє вам розгортати ваші мікросервіси на різних середовищах. Висока продуктивність: .NET 8 має оптимізовану швидкість виконання коду, що дозволяє розробляти швидкі та ефективні мікросервіси. ASP.NET Core надає можливості для розробки високопродуктивних веб-додатків з використанням асинхронного програмування та підтримки різних протоколів (gRPC, HTTP і т.д.).

Вбудована безпека: ASP.NET Core має вбудовані механізми безпеки, такі як підтримка HTTPS, валідація вхідних даних, автентифікація та авторизація користувачів, що дозволяє забезпечити захист вашого додатку та даних користувачів.

Легка масштабованість: Мікросервісна архітектура дозволяє легко масштабувати окремі компоненти системи, що використовують .NET 8 та ASP.NET Core. Ви можете використовувати контейнеризацію з Docker та оркестрування контейнерів з Kubernetes для ефективного керування масштабуванням.

Легка розширюваність та модульність. Це дозволяє додавати та змінювати функціонал незалежно, зменшує час розгортання та сприяє гнучкості в розробці, оновленні та масштабуванні додатків.

Обираючи технології .NET 8 та ASP.NET Core для реалізації мікросервісної архітектури, можна користуватися широким спектром бібліотек та інструментів, що сприяють зручності розробки та інтеграції з іншими технологіями. Ось як обрані технології підтримують роботу з хмарними технологіями, gRPC та RabbitMQ:

- хмарні технології: NET 8 та ASP.NET Core мають вбудовану підтримку для роботи з різними хмарними сервісами, такими як Azure. Це означає, що розробники можуть легко інтегрувати свої мікросервіси з різними хмарними сервісами та використовувати їх функціонал для забезпечення високої доступності, масштабованості та безпеки. Перевагами хмарних обчислень є те, що користувач має можливість не купувати потужні комп’ютери. Зокрема, і організації можуть відмовлятися від придбання потужних серверів і йти “в хмари”. Для розробника – контрольованість усього процесу. У разі виникнення проблеми їм істотно простіше буде змоделювати ситуацію, що викликала помилку, – адже усі дані і так зберігаються в них. [9]
- gRPC: ASP.NET Core підтримує gRPC, що є високопродуктивним механізмом для взаємодії між розподіленими системами. gRPC дозволяє ефективно використовувати HTTP/2 для передачі даних та генерації коду на основі Protocol Buffers (protobuf). Це робить комунікацію між мікросервісами швидкою та ефективною.
- RabbitMQ: .NET має багато бібліотек для роботи з RabbitMQ, такі як `RabbitMQ.Client`, яка надає зручний API для створення та керування

повідомленнями через RabbitMQ. Це дозволяє реалізувати асинхронну та надійну комунікацію між мікросервісами через повідомлення.

Зарядна станція взаємодіє через протокол OCPP. Протокол OCPP (Open Charge Point Protocol) є стандартом відкритого зв'язку для зарядних станцій електромобілів. Цей протокол визначає спосіб комунікації між зарядними станціями та системою управління зарядними точками (ЦЗТ), що дозволяє взаємодіяти зі змінними параметрами зарядних станцій, такими як потужність заряду, статус зарядження, розподіл заряду між різними точками та багато іншого. OCPP є стандартом з відкритим кодом, що дозволяє виробникам зарядних станцій та розробникам систем ЦЗТ використовувати єдиний протокол для взаємодії між різними системами. Підтримує багатопотокову комунікацію, що дозволяє одній системі управління керувати декількома зарядними станціями одночасно.

Протокол включає функції для керування зарядкою, такі як запуск/зупинка зарядження, контроль потужності заряду, моніторинг статусу зарядження та інші. OCPP дозволяє планувати та керувати розкладом зарядки, що дозволяє оптимізувати використання зарядних станцій у різні періоди доби.

Протокол має можливості для діагностики зарядних станцій, збору даних про статус, енергоспоживання та звітності про зарядження.

Протокол OCPP став важливим стандартом у сфері електромобілів, оскільки він дозволяє різним виробникам та розробникам інтегрувати свої системи та зарядні станції для забезпечення ефективного та зручного користування електромобілями.

Також зарядна станція взаємодіє через веб-сокети. Веб-сокети у контексті протоколу OCPP (Open Charge Point Protocol) використовуються для забезпечення зв'язку між зарядними станціями та іншими сервісами, такими як системи управління зарядними точками (ЦЗТ). Основна мета веб-сокетів у цьому контексті - забезпечити надійний та ефективний обмін повідомленнями між зарядними станціями та центральною системою управління.

Chargepoint WebSocket Service (CWS) - це відповідний сервіс обробки підключень, який відповідає за управління та обмін повідомленнями через веб-

сокети між зарядними станціями та іншими сервісами. Деякі особливості та функції цього сервісу:

- підтримка веб-сокетів. Chargepoint WebSocket Service використовує веб-сокети для забезпечення двонаправленого зв'язку між зарядними станціями та центральною системою управління.
- обмін повідомленнями. Цей сервіс дозволяє обмінюватися різними типами повідомлень між зарядними станціями та центральною системою, такими як статус зарядження, запити на старт/стоп зарядження, інформація про доступність зарядних станцій тощо.
- керування підключеннями. Chargepoint WebSocket Service відповідає за керування підключеннями до веб-сокетів, включаючи управління сесіями, аутентифікацію, авторизацію та забезпечення безпеки зв'язку.
- моніторинг та діагностика. Сервіс також надає можливості для моніторингу стану підключень, діагностики помилок та збору даних про використання зарядних станцій.

Загалом, Chargepoint WebSocket Service є ключовим компонентом для ефективної роботи з протоколом OCPP, забезпечуючи надійну та швидку комунікацію між зарядними станціями та центральною системою управління, що сприяє ефективному керуванню зарядними точками та покращенню досвіду користувачів електромобілів.

Взаємодія між сервісом обробки підключень зарядних станцій та іншою частиною системи (наприклад, центральною системою управління або іншими сервісами) відбувається асинхронно за допомогою брокера повідомлень RabbitMQ. RabbitMQ є популярним брокером повідомлень, який забезпечує надійну та ефективну асинхронну комунікацію між різними компонентами системи. Він дозволяє надсилати, отримувати та обробляти повідомлення між різними частинами системи, що дозволяє реалізувати архітектуру з подіями та асинхронною обробкою даних.

Сервіс обробки підключень використовує RabbitMQ для асинхронної взаємодії з іншими частинами системи, надсилаючи повідомлення про статус

зарядження до центральної системи управління. Це забезпечує швидку та надійну обробку даних між компонентами, дозволяючи реалізувати ефективну та масштабовану асинхронну обробку подій та даних, що полегшує роботу всієї системи [10].

Використання асинхронної взаємодії з допомогою RabbitMQ дозволяє збільшити ефективність та швидкість обробки даних, зменшити блокування та забезпечити надійну доставку повідомлень між компонентами системи.

Отже, асинхронна взаємодія за допомогою брокера повідомлень RabbitMQ дозволяє забезпечити ефективну та надійну комунікацію між сервісом обробки підключень зарядних станцій та іншими частинами системи, що сприяє покращенню функціональності та продуктивності системи управління зарядними точками.

Детальніше про мікросервіси з діаграми А.1.

SignalR Hub є частиною ASP.NET Core та дозволяє реалізувати спілкування між клієнтом та сервером у реальному часі за допомогою веб-сокетів або інших технологій, які автоматично вибираються. Цей сервіс особливо корисний для розробки додатків, які потребують миттєвого оновлення даних, наприклад, в месенджерах або панелях моніторингу.

Aggregator є сервісом, який виконує функцію збору, агрегації та відправлення даних з різних мікросервісів на клієнтську сторону, забезпечуючи цілісність та ефективну роботу системи.

Depots Service відповідає за управління депо.

Charge Points Service відповідає за управління зарядними точками для електромобілів. Цей сервіс може включати в себе функції керування статусом зарядження, планування зарядження, моніторингу потужності та інші.

OCPP Tags Service (Open Charge Point Protocol) сервіс OCPP tags виконує повний цикл управління авторизаційними тегами у контексті OCPP - від їх реєстрації та змін до обробки запитів на авторизацію, забезпечуючи надійну та ефективну роботу системи заряджання електромобілів.

Connectors Service забезпечує надійне та ефективне управління роз'ємами зарядних станцій, дозволяючи системі реагувати на зміни стану роз'ємів та забезпечуючи безперебійну роботу усієї інфраструктури заряджання електромобілів.

Transaction Service забезпечує точне та ефективне ведення обліку транзакцій заряджання, дозволяючи системі ефективно керувати ресурсами та забезпечувати звітність про спожиту енергію електромобіля.

Reservations Service дозволяє ефективно керувати доступом до конекторів, забезпечуючи користувачам можливість зарезервувати конектори для зарядки своїх електромобілів та скасувати бронювання в разі потреби.

Charging Profile Service допомагає управляти процесами заряджання електромобілів, забезпечуючи належні налаштування та оптимальні умови зарядки для кожного транспортного засобу, а також ефективне використання зарядних станцій у депо.

Heartbeats Service відповідає за надсилання регулярних "пульсаційних" сигналів для підтвердження життєздатності та доступності компонентів системи.

Кожен з цих сервісів виконує важливі функції в екосистемі управління зарядними станціями для електромобілів, забезпечуючи надійну та ефективну роботу системи в цілому.

Поглиблюючи розгляд сервісу heartbeats, важливо зазначити його стратегічне значення в контексті стабільності та надійності системи, особливо у великомасштабних проектах з управління багатьма станціями. Саме серцебиття відіграє роль віртуального "пульсу" кожної станції, надсилаючи періодичні сигнали для підтвердження працездатності та готовності до роботи.

Основною проблемою, з якою стикаються при обробці серцебиття, є необхідність ефективного зберігання цих даних та оптимізація навантаження на основну базу даних. У зв'язку з цим, було прийнято стратегічне рішення використовувати Azure Table Storage - компонент платформи Azure, що дозволяє зберігати структуровані дані у вигляді таблиць, з врахуванням високого рівня доступності та надійності.

Azure Table Storage, хоча і працює трохи повільніше порівняно з Azure SQL Database, відіграє ключову роль у забезпеченні економічної ефективності та оптимізації витрат, а також у розподіленні навантаження, оскільки може обробляти великі обсяги структурованих даних без значного впливу на швидкодію системи. Особливо варто відзначити, що затримки в обробці серцебиття не мають такого великого впливу, як, наприклад, в обробці транзакцій чи інших важливих для швидкодії процесах. Таким чином, використання Azure Table Storage для даних про серцебиття дозволяє забезпечити необхідний рівень продуктивності та функціональності, зменшуючи витрати на обробку цих даних та забезпечуючи стабільну роботу системи у масштабах великого проекту з багатьма станціями.

Вибором системи керування базами даних (СКБД) є Microsoft SQL Server для проекту. SQL Server від Microsoft є однією з найпопулярніших та найбільш широко використовуваних СКБД у світі, особливо у корпоративному середовищі. Це відбувається не без причин, оскільки SQL Server має безліч переваг, які роблять його відмінним вибором для проектів будь-якої масштабності та складності.

Перш за все, SQL Server відомий своєю надійністю та стабільністю. Він має високий рівень захисту даних і може оптимально працювати навіть при великому обсязі даних та високих навантаженнях. Це дозволяє забезпечити безперервну доступність та інтеграцію даних в вашій системі, що є критично важливим для бізнес-застосунків.

Далі, SQL Server має широкий функціонал та набір інструментів для оптимізації продуктивності бази даних. Це включає в себе індексацію, розпаралелені запити, кешування та інші оптимізаційні методи, які дозволяють забезпечити швидкодіючий доступ до даних та оптимальну роботу з базою даних в цілому.

Окрім того, SQL Server підтримує масштабування як вертикально, що означає збільшення ресурсів на одному сервері, так і горизонтально, що дозволяє розподілити навантаження на декілька серверів або хмарних обчислень. Це дозволяє пристосувати SQL Server до зростаючих потреб вашого проекту та забезпечити ефективну роботу навіть у високонавантажених середовищах.

Наступним важливим аспектом є широка підтримка та інтеграція з іншими продуктами Microsoft. SQL Server добре інтегрується з іншими продуктами Microsoft, такими як Microsoft Azure, SharePoint, Excel, Power BI тощо. Це спрощує розробку та інтеграцію рішень, а також забезпечує сумісність з іншими сервісами та продуктами, які можуть бути вам потрібні для повноцінного функціонування вашої системи.

Розроблена логічна схема БД в додатку Б.

Зважаючи на зростаючу важливість та розширення областей застосування сучасних технологій у сфері інформаційних систем та програмного забезпечення, стає очевидним, що використання новітніх інструментів та сервісів має вирішальне значення для успішного впровадження та функціонування проектів. У цьому контексті особливо важливим є використання хмарних технологій та платформ для розгортання, моніторингу, масштабування та управління різними аспектами проекту.

Використання Microsoft Azure - однієї з найпопулярніших та найбільш розвинених хмарних платформ у сучасному IT-середовищі. Microsoft Azure володіє широким набором інструментів, служб та сервісів, що дозволяє розробникам та адміністраторам створювати, впроваджувати та управляти різноманітними додатками та сервісами в хмарному середовищі.

Використання Microsoft Azure для реалізації сучасних програмних рішень, а також аналіз практичних прикладів застосування платформи в різних сферах IT-індустрії. У цьому контексті розглянемо інтеграцію та взаємодію з іншими технологіями, сервісами та продуктами Microsoft, а також можливість використання широкого спектру інструментів для автоматизації, моніторингу та аналізу даних.

Використання Microsoft Azure у проекті може мати численні переваги, особливо для розгортання та керування великими та складними системами.

Azure надає гнучкість та масштабованість для розгортання ваших додатків та сервісів у хмарному середовищі. Створення віртуальних машин, контейнерів,

функцій, баз даних та інших ресурси, що дозволяє гнучко пристосовувати ваші рішення до змінних потреб проекту.

Azure забезпечує широкий спектр інструментів та сервісів для захисту інформації та даних. Це включає в себе управління доступом, шифрування, моніторинг безпеки та інші функції, які допомагають забезпечити високий рівень безпеки проекту.

Azure має ряд служб для інтеграції додатку з іншими сервісами та ресурсами. Azure пропонує інструменти та сервіси для аналізу даних, машинного навчання, обробки медіа та інших аналітичних задач.

Azure має різноманітні інструменти для розробки, тестування та розгортання додатків у хмарі.

Завдяки гнучкості та масштабованості Azure, можна легко збільшувати або зменшувати обсяги ресурсів відповідно до потреб проекту. Це дозволяє забезпечити високу продуктивність та ефективність роботи системи.

Microsoft Azure надає інтегровану підтримку для Microsoft SQL Server, що охоплює різноманітні функції, такі як автоматичне масштабування, реплікація даних і автоматичне створення резервних копій. Це забезпечує високий рівень надійності та доступності бази даних в середовищі Azure.

Автоматичне масштабування у Azure дозволяє динамічно змінювати розмір ресурсів, виділених для SQL Server, у відповідності до поточного обсягу роботи та потреб системи. Це допомагає оптимізувати використання ресурсів та забезпечує ефективне функціонування бази даних навіть у змінних умовах навантаження.

Реплікація даних в Azure забезпечує наявність копій даних на різних серверах або в різних датацентрах. Це дозволяє уникнути втрати даних у випадку відмови обладнання або збою системи та швидко відновити роботу бази даних.

Автоматичне створення резервних копій в Azure дозволяє регулярно створювати копії даних для забезпечення можливості відновлення в разі втрати даних або несправності системи.

Загалом, бізнес-логіка проекту базується на забезпеченні ефективного та безпечного користування зарядними станціями для електромобілів, що включає в

себе реєстрацію та авторизацію користувачів, управління транзакціями та резерваціями, налаштування профілів зарядки та моніторинг стану системи.

3.2 Взаємодія з клієнтською частиною

Зв'язок між клієнтською та серверною частинами важливий для функціонування будь-якого веб-додатка. Основний спосіб цієї взаємодії в багатьох сучасних додатках - це використання HTTP протоколу та RESTful архітектури.

Клієнтська частина, яка є веб-браузером, надсилає HTTP запити на серверну частину, щоб отримати або зберегти дані. Сервер обробляє ці запити та повертає відповіді, які можуть бути у форматі JSON, XML або HTML.

RESTful контролери в системі взаємодіють з клієнтською частиною та серверною для забезпечення обміну даними через HTTP протокол. Основна ідея REST полягає в тому, щоб кожний ресурс системи був доступний за унікальним URL і здатний відповідати на запити HTTP методами, такими як GET, POST, PUT та DELETE.

Наприклад, у RESTful архітектурі може бути контролер для управління зарядними станціями (Charge Points Controller). Цей контролер містить методи, які дозволяють клієнтській частині взаємодіяти зі станціями зарядки через HTTP запити.

Наприклад, метод GET `/chargepoints` повертає список доступних зарядних станцій, а метод GET `/chargepoints/{id}` повертає інформацію про конкретну станцію за її ідентифікатором. Метод POST `/chargepoints` дозволяє створювати нові зарядні станції, а метод PUT `/chargepoints/{id}` дозволяє змінювати існуючі дані про станцію.

Контролер для управління транзакціями (Transactions Controller) містить методи для роботи з фінансовими операціями. Наприклад, метод POST `/transactions` приймає дані про нову транзакцію та додає її до бази даних. Метод GET `/transactions/{id}` дозволяє отримати інформацію про конкретну транзакцію за її ідентифікатором.

Ці контролери допомагають взаємодіяти клієнтській частині з сервером через зрозумілі HTTP запити та відповіді, що сприяє ефективності та зручності розробки, розширення та підтримки системи.

Зв'язок між клієнтською та серверною частинами також включають додаткові механізми для оптимізації взаємодії та підвищення ефективності додатка.

Асинхронність. У випадках, коли обробка запитів на сервері може займати тривалий час або потребує великих обчислювальних ресурсів, використання асинхронних запитів може покращити продуктивність. Клієнт може надсилати асинхронні запити, які не блокують потік виконання, а отримувати відповіді пізніше, коли вони будуть готові.

Кешування. Використання кешування на клієнтській та серверній сторонах може допомогти зменшити час відповіді на запити. Наприклад, сервер може кешувати попередньо обчислені результати запитів, а клієнт може кешувати дані для подальшого використання без необхідності повторних запитів до сервера.

Обробка помилок. Передача інформації про помилки від сервера до клієнта важлива для забезпечення коректної реакції на непередбачені ситуації. Використання стандартних кодів помилок та повідомлень допомагає зрозуміти та обробити проблеми, які можуть виникнути під час взаємодії.

Автентифікація та авторизація. Важливим аспектом є забезпечення безпеки взаємодії між клієнтом та сервером. Використання механізмів автентифікації (підтвердження ідентичності) та авторизації (призначення прав доступу) дозволяє контролювати доступ до ресурсів та захищати дані від несанкціонованого доступу.

Ці додаткові аспекти взаємодії клієнтської та серверної частин системи допомагають забезпечити надійну, швидку та безпечну роботу додатка для кінцевих користувачів.

3.3 Використання протоколу gRPC (HTTP/2)

З кожним днем розвиток інформаційних технологій набирає все більшого обертів, ведучи до ефективнішого та швидкого обміну даними між різними системами. У цьому контексті, gRPC відіграє ключову роль у сучасному

програмуванні, забезпечуючи високопродуктивну, масштабовану та надійну взаємодію між розподіленими компонентами додатків.

Розглядаючи ці тенденції, новий підрозділ у звіті розглядає впровадження gRPC як одну з ключових технологій для забезпечення ефективного обміну даними між мікросервісами. gRPC, оснований на HTTP/2 і здатний працювати з різними мовами програмування, дозволяє створювати швидкі та ефективні мережеві додатки з розподіленою архітектурою.

Взаємодія між сервісами за допомогою gRPC є ефективним та швидким способом обміну даними у мікросервісних архітектурах. gRPC використовує Protocol Buffers для серіалізації даних та HTTP/2 для передачі даних, що забезпечує високу продуктивність та надійність взаємодії.

Один із сценаріїв використання gRPC - це взаємодія між сервісами, яка відбувається синхронно та безпосередньо через мережу. Наприклад, сервіс управління зарядними станціями (Charge Tags Service) може використовувати gRPC для взаємодії з сервісом управління транзакціями (Transaction Service).

У той же час, коли комунікація відбувається асинхронно та потребує обробки повідомлень через брокера повідомлень, такого як RabbitMQ, використання gRPC може бути менш практичним. Наприклад, для обробки повідомлень про статус зарядних станцій або інших подій, які можуть виникати асинхронно, RabbitMQ може бути кращим варіантом. Якщо розглядати процес обміну повідомленнями через RabbitMQ, то можна взяти до уваги Chargepoint Websocket Service, що відповідає за цей обмін, приймаючи повідомлення від зарядних станцій, а потім відправляє ці повідомлення через RabbitMQ до наших центральних сервісів для подальшої обробки та аналізу.

gRPC використовується для синхронної та надійної взаємодії між сервісами, коли потрібно отримати швидкий та ефективний обмін даними. У той же час, асинхронна комунікація за допомогою RabbitMQ відповідає за обробку подій та повідомлень, що виникають асинхронно та потребують обробки через брокера повідомлень. Такий підхід дозволяє оптимально розподілити завдання між сервісами та забезпечити ефективну взаємодію в мікросервісній архітектурі.

gRPC базується на протоколі HTTP/2, який є оновленою версією HTTP і відрізняється від попередньої версії, HTTP/1.1, у багатьох аспектах. Основні риси HTTP/2, які становлять основу для ефективної роботи gRPC, включають декілька пунктів.

Мультиплексування. Однією з ключових особливостей HTTP/2 є здатність мультиплексувати (одночасно обробляти) більше одного потоку даних через одне з'єднання. Це дозволяє знизити затримки в передачі даних та підвищити продуктивність, оскільки можна використовувати одне з'єднання для взаємодії з різними сервісами чи ресурсами.

Стиснення заголовків. HTTP/2 підтримує стиснення заголовків, що дозволяє зменшити обсяг переданих даних та покращити швидкість обробки запитів та відповідей. Це особливо важливо для великих обсягів даних, таких як серіалізовані повідомлення у gRPC.

Паралельність. HTTP/2 підтримує паралельність запитів, що дозволяє виконувати кілька запитів одночасно через одне з'єднання. Це сприяє підвищенню ефективності та швидкості обміну даними, особливо в умовах великої кількості одночасних запитів до сервера.

Підтримка заголовків (headers) і куки (cookies). HTTP/2 дозволяє відправляти заголовки та куки з попередніх запитів без повторної передачі, що допомагає знизити обсяг переданих даних та зберегти ресурси мережі та сервера.

Приоритети потоків. HTTP/2 має механізм для встановлення пріоритетів потоків, що дозволяє задати важливість запитів та відповідей. Це корисно для оптимізації обробки різних видів запитів в залежності від їхньої важливості.

3.4 Використання протоколу HTTP

Протокол HTTP (Hypertext Transfer Protocol) є одним із найбільш поширених протоколів в мережі Інтернет. Використання HTTP в контексті веб-додатків має кілька ключових аспектів.

HTTP дозволяє клієнтській частині (наприклад, веб-браузеру) надсилати запити на сервер та отримувати відповіді. Це основний механізм взаємодії між клієнтом та сервером в веб-додатках.

HTTP визначає різні методи запитів, такі як GET (отримання даних), POST (створення даних), PUT (оновлення даних) та DELETE (видалення даних). Це дозволяє виконувати різні дії з ресурсами на сервері.

Кожен ресурс на сервері має унікальний URI, за яким його можна ідентифікувати. Наприклад, веб-сторінки мають URI у вигляді URL (Uniform Resource Locator), який вказує на конкретний шлях до ресурсу.

HTTP використовує статусні коди у відповідях сервера, які вказують на стан виконання запиту. Наприклад, код 200 означає успішну відповідь, а коди 4xx та 5xx вказують на помилки чи проблеми на сервері.

HTTP підтримує механізм кешування, що дозволяє зберігати копії ресурсів на клієнтській стороні. Це допомагає знизити навантаження на сервер та покращує швидкість завантаження сторінок.

HTTP може використовувати механізм сесій та куків для збереження стану взаємодії між клієнтом та сервером. Це дозволяє зберігати інформацію про користувача або стан сесії під час переходів між сторінками.

Загалом, використання протоколу HTTP є ключовим у веб-розробці, оскільки він надає стандартні механізми для взаємодії між клієнтом та сервером, що дозволяє створювати ефективні та функціональні веб-додатки.

3.5 Використання протоколу AMQP

AMQP (Advanced Message Queuing Protocol) є стандартним протоколом для асинхронної обміну повідомленнями у розподілених системах. RabbitMQ, у свою чергу, є популярним брокером повідомлень, який підтримує AMQP та інші протоколи, такі як MQTT, STOMP та HTTP. Основне використання AMQP у RabbitMQ полягає в створенні надійних та масштабованих систем обміну повідомленнями в архітектурі мікросервісів.

Асинхронна взаємодія: AMQP дозволяє реалізувати асинхронну взаємодію між сервісами. Коли сервіс A потребує виконання певної операції сервісом B, він може відправити повідомлення через RabbitMQ, а сервіс B обробити це повідомлення, коли буде готовий.

Маршрутизація повідомлень: RabbitMQ дозволяє налаштовувати правила маршрутизації повідомлень, що дозволяє ефективно керувати потоком даних у системі. Наприклад, повідомлення можуть бути автоматично розподілені між різними чергами в залежності від їхнього типу чи властивостей.

Гарантія доставки: AMQP забезпечує гарантію доставки повідомлень до призначеного отримувача. Це означає, що навіть у випадку збою або перебоїв у мережі, повідомлення будуть збережені у черзі та доставлені при наступній можливості.

Масштабованість: RabbitMQ дозволяє масштабувати систему обміну повідомленнями шляхом додавання нових вузлів (nodes) та кластерів. Це дозволяє підтримувати велику кількість повідомлень та обробляти їх ефективно.

Робота з різними протоколами: Окрім AMQP, RabbitMQ підтримує інші протоколи, що розширює можливості взаємодії з різними типами систем.

Загалом, використання AMQP та RabbitMQ дозволяє створювати надійні та масштабовані системи обміну повідомленнями в розподілених додатках, що є важливим аспектом при побудові мікросервісної архітектури.

3.6 Використання OCPP (WebSocket)

OCPP (Open Charge Point Protocol) - це стандартний протокол для взаємодії зарядних станцій електромобілів з управляючою системою заряду [11]. Його використання за допомогою WebSocket дозволяє створювати ефективні та надійні системи управління зарядженням, забезпечуючи багатофункціональність та зручність для користувачів.

Використання WebSocket у протоколі OCPP [12] дозволяє встановлювати постійне з'єднання між зарядною станцією та управляючою системою. Це дозволяє

отримувати та надсилати дані у реальному часі, що є критичним для ефективного управління зарядкою.

WebSocket забезпечує двосторонній обмін даними, тобто як зарядна станція, так і управляюча система можуть надсилати та отримувати повідомлення у будь-який момент часу. Це дозволяє налагоджувати та контролювати процес зарядки електромобіля в реальному часі.

WebSocket дозволяє зменшити споживання ресурсів мережі та сервера, оскільки з'єднання залишаються активними лише тоді, коли вони дійсно потрібні для передачі даних. Це дозволяє підвищити ефективність використання мережевих ресурсів.

WebSocket забезпечує стабільний та надійний зв'язок між зарядною станцією та управляючою системою, навіть у випадку зміни умов мережі або тимчасових перебоїв у зв'язку.

WebSocket дозволяє застосовувати різноманітні заходи безпеки, такі як шифрування даних та автентифікація, що забезпечує безпеку обміну даними між зарядними станціями та управляючою системою.

Загалом, використання OCPP через WebSocket дозволяє створювати потужні та ефективні системи управління зарядженням електромобілів, забезпечуючи швидкий, надійний та безпечний обмін даними у реальному часі.

3.7 Використання Docker та Kubernetes

Docker – це програмне забезпечення з відкритим кодом, найпопулярніша платформа для управління контейнерами. [13]

З введенням Docker у світ програмування та розробки виникло безліч нових можливостей та підходів до створення, тестування та розгортання програмних продуктів. Docker перетворився на один із найважливіших інструментів для розробників і системних адміністраторів, який дозволяє створювати ізольовані середовища, що включають усі необхідні залежності та компоненти програмного забезпечення. Важливість та переваги використання Docker у контексті управління системою зарядження електромобілів за стандартом OCPP через WebSocket.

Docker допомагає створювати ефективні, масштабовані та надійні системи, забезпечуючи швидкість розгортання, масштабованість та зручність у керуванні окремими компонентами системи зарядження.

Загалом, використання Docker сприяє ефективному та гнучкому управлінню системою зарядження електромобілів за стандартом OCPP через WebSocket, забезпечуючи швидкість, масштабованість та надійність у роботі системи.

Kubernetes - це платформа з відкритим вихідним кодом для управління контейнеризованими робочими навантаженнями та супутніми службами. [14] Він надає набір інструментів для оркестрації та керування контейнерами, що дозволяє легко керувати складними додатками та інфраструктурою, які використовують Docker контейнери. Основні переваги використання Kubernetes у контексті системи управління зарядженням електромобілів через OCPP та WebSocket включають наступне. Kubernetes дозволяє оркеструвати та керувати контейнерами, автоматизуючи процеси їх розгортання, масштабування та управління життєвим циклом. Це дозволяє легко керувати великими кількостями контейнерів, що використовуються для різних частин системи.

Kubernetes дозволяє створювати, управляти та масштабувати складні додатки у контейнеризованому середовищі, забезпечуючи високу доступність, ефективність та надійність системи управління зарядженням електромобілів.

4 МАСШТАБОВАНІСТЬ ПРОЕКТУ

Масштабованість проекту відноситься до здатності системи ефективно збільшувати свої ресурси та пропускну здатність у відповідь на зростаючі потреби або навантаження. Це означає, що система може адаптуватися до змін у розмірі даних, кількості користувачів або обсягу операцій, не втрачаючи при цьому продуктивності або доступності.

В масштабованих системах можна легко збільшувати обчислювальні потужності, включаючи сервери та бази даних, розподіляти навантаження між різними компонентами, автоматизувати процеси моніторингу та управління ресурсами. Це дозволяє забезпечувати стабільну роботу системи навіть при збільшенні обсягів даних, зростанні кількості користувачів або змінах у вимогах до функціональності.

Проект системи управління зарядженням електромобілів відзначається високою масштабованістю та гнучкістю завдяки використанню сучасних технологій та підходів. Для оркестрації та керування різними компонентами системи використовується Kubernetes, що дозволяє автоматизувати процеси розгортання, масштабування та управління контейнеризованими додатками. Kubernetes забезпечує механізми автоматичного масштабування, балансування навантаження та високу доступність, що дозволяє оптимально використовувати ресурси та забезпечувати стабільну роботу системи при змінних обсягах роботи.

Крім того, використання мікросервісної архітектури дозволяє незалежно масштабувати окремі частини системи, що сприяє гнучкості та зручності у вдосконаленні окремих компонентів без впливу на інші. Docker використовується для контейнеризації компонентів системи, що спрощує розгортання та управління окремими сервісами.

Проект системи управління зарядженням електромобілів знаходиться на високому рівні масштабованості завдяки використанню Microsoft Azure та MS SQL Server.

Microsoft Azure надає рішення для масштабування та керування інфраструктурою, включаючи можливість легко розширювати ресурси в

залежності від навантаження та автоматизувати процеси управління. Також Azure забезпечує надійність за рахунок резервного копіювання та відновлення даних.

У поєднанні з MS SQL Server, який підтримує великі обсяги даних та має механізми автоматичного масштабування, Azure дозволяє створювати розподілені та надійні системи. Реплікація даних та резервне копіювання забезпечують високу доступність та стабільність роботи системи навіть при збільшенні обсягів роботи.

Ця комбінація інструментів дозволяє ефективно реагувати на зміни в навантаженні та забезпечує високу продуктивність та доступність системи управління зарядженням електромобілів.

5 БЕЗПЕКА ДАНИХ

У даному проекті безпека даних є однією з найважливіших складових, оскільки він працює з чутливою інформацією про користувачів, транзакції та стан системи.

Автентифікація та авторизація. Використання .NET 8 та ASP.NET Core дозволяє легко налаштовувати систему автентифікації і авторизації, включаючи механізми аутентифікації з використанням токенів, ролей та дозволів для керування доступом до ресурсів.

Шифрування даних. Дані, які передаються через мережу або зберігаються в базі даних, шифруються за допомогою сучасних шифрувальних алгоритмів для захисту від несанкціонованого доступу.

Захист мережевого трафіку. Використання протоколів HTTPS, AMQP та gRPC забезпечує шифрування мережевого трафіку, що знижує ризик перехоплення та зламу інформації.

Резервне копіювання та відновлення. Microsoft Azure надає можливості для резервного копіювання та відновлення даних, що дозволяє відновлювати інформацію в разі втрати чи пошкодження.

Автоматизація безпеки в контейнерах. Використання Docker та Kubernetes дозволяє налаштовувати політики безпеки контейнерів, включаючи обмеження прав доступу, моніторинг вразливостей та автоматичне виявлення та виправлення проблем безпеки.

Загалом, використання цих технологій та підходів дозволяє побудувати систему з надійним захистом даних та забезпечити високий рівень безпеки усіх аспектів проекту.

6 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

6.1 Структура проекту

Сучасні технології та складне програмне забезпечення вимагають нових підходів до розробки для забезпечення ефективності, масштабованості та гнучкості. Мікросервісна архітектура є одним із таких підходів, де програма розбивається на невеликі, незалежні компоненти - мікросервіси, кожен з яких виконує певну функцію і має свій API.

Основні переваги мікросервісів:

- розбиття на компоненти: дозволяє спростити розробку та підтримку програмного забезпечення;
- гнучкість у розробці: зміни в одному сервісі не впливають на інші, що прискорює впровадження нових функцій;
- масштабованість: кожен мікросервіс можна масштабувати окремо;
- легка заміна та оновлення: незалежні компоненти легко замінювати та оновлювати;
- розподілена робота команд: команди можуть працювати над різними компонентами паралельно.

Недоліки включають складність управління великою кількістю сервісів, необхідність ефективного моніторингу, конфігурації, та проблеми з консистентністю даних. Для вирішення цих проблем слід дотримуватись принципів чіткої архітектури, ефективного моніторингу, керування конфігураціями, автоматизації інфраструктури, тестування, контейнеризації, документації та комунікації.

Мікросервісна архітектура на базі C# з використанням .NET 8 та ASP.NET Core забезпечує продуктивність і масштабованість серверних додатків. ASP.NET Core підтримує крос-платформеність, високу продуктивність, вбудовану безпеку, легку масштабованість та модульність.

Для інтеграції з хмарними технологіями, gRPC та RabbitMQ ці інструменти надають широкий спектр можливостей. Хмарні сервіси забезпечують високу

доступність та масштабованість, gRPC - ефективну взаємодію між розподіленими системами, а RabbitMQ - надійну асинхронну комунікацію між компонентами.

Використання протоколу OCPP (Open Charge Point Protocol) для зарядних станцій електромобілів дозволяє стандартизувати комунікацію між зарядними станціями та системами управління, забезпечуючи надійний та ефективний обмін повідомленнями. RabbitMQ використовуються для забезпечення надійної асинхронної взаємодії між компонентами системи після отримання даних з самої зарядної станції. Протокол OCPP передбачає використання веб-сокетів (WebSockets) для забезпечення двостороннього зв'язку між зарядною станцією і сервером в реальному часі. Вони дозволяють обмінюватися даними без необхідності встановлення нового з'єднання для кожної передачі, що значно знижує затримки і покращує ефективність взаємодії.

Проєкт розроблено з урахуванням сучасних принципів мікросервісної архітектури, що забезпечує гнучкість, масштабованість та легкість підтримки системи. Він складається з кількох ключових компонентів, кожен з яких виконує свою специфічну роль у загальній структурі проєкту. Така організація дозволяє чітко розділити відповідальності, спростити процес розробки та тестування, а також забезпечити ефективну інтеграцію різних частин системи. Нижче наведено опис основних розділів проєкту, що допоможе краще зрозуміти його внутрішню структуру та взаємодію між компонентами.

Проєкт складається з таких основних частин (див. рис. 6.1):

- Domain: це область проєкту, яка включає всі доменні моделі та загальні утиліти, необхідні для роботи бізнес-логіки.
- Emulator: це компонент проєкту, який призначений для емуляції зарядних станцій.
- Infrastructure: це компонент проєкту містить інфраструктурні елементи, такі як доступ до баз даних, налаштування зовнішніх сервісів, репозиторії та інші інфраструктурні компоненти, що забезпечують підтримку роботи доменних моделей та основної бізнес-логіки проекту.

– Services: це компонент проєкту, де зберігаються всі сервіси, які забезпечують бізнес-логіку і функціональність додатка.
– docker-compose: цей файл, що містить налаштування для контейнеризації додатку за допомогою Docker. Він описує, як різні сервіси взаємодіють між собою, їх налаштування, мережі, томи та інші параметри, необхідні для розгортання додатку в контейнерах.

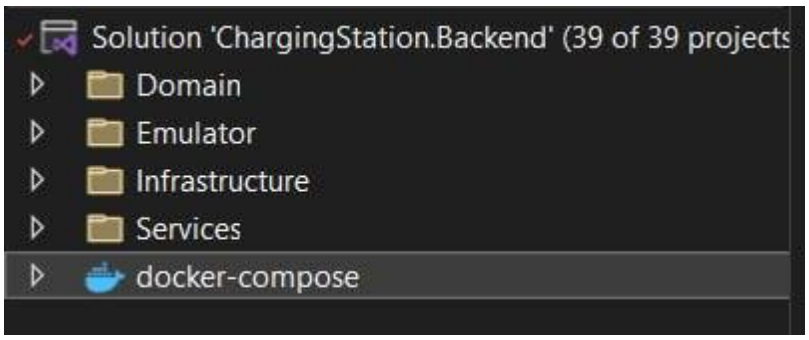


Рисунок 6.1 - Структура проекту (рисунок виконано самостійно)

Ця структура дозволяє ефективно організувати код і розділити різні аспекти додатку на окремі модулі, що спрощує розробку, тестування та підтримку.

Якщо більш детальніше розглядати структуру, то можна вилучити таку інформацію (див. рис. 6.2 та 6.3):

Domain:

– ChargingStation.Common: загальні утиліти та спільні компоненти, що використовуються в різних частинах проекту.
– ChargingStation.Domain: зберігаються доменні моделі проекту.

Emulator:

– ChargePointEmulator.Application: логіка застосування для емуляції зарядних станцій.
– ChargePointEmulator.Persistence: сховище даних для емуляції.
– ChargePointEmulator.UI: користувацький інтерфейс для взаємодії з емулятором.

Infrastructure:

- ChargingStation.CacheManager: управління кешем для швидкого доступу до даних.
- ChargingStation.Infrastructure: це частина проекту, яка забезпечує основні сервіси та функціональні можливості для підтримки та інтеграції різних компонентів системи
- ChargingStation.InternalCommunication: внутрішня комунікація між сервісами.
- ChargingStation.Mailing: управління відправкою електронної пошти.
- ChargingStation.TableStorage: управління даними у Azure Table Storage, забезпечуючи ефективне збереження та доступ до структурованих даних, що дозволяє зберігати великі обсяги інформації з високою продуктивністю.

Services:

а) Aggregator:

1) Aggregator: агрегує дані з різних джерел (мікросервісів) для спрощеного доступу.

б) ChargePoints:

1) ChargePoints.Api: API для взаємодії з зарядними точками.
2) ChargePoints.Application: відповідає за роботу бізнес-логіки з зарядними точками.
3) ChargePoints.Grpc: gRPC сервіс для комунікації між мікросервісами, з метою отримання даних о зарядних точках.

в) ChargingProfiles:

1) ChargingStation.ChargingProfiles: управління профілями зарядки.

г) Connectors:

1) Connectors.Api: API для взаємодії з конекторами зарядних станцій.
2) Connectors.Application: відповідає за роботу бізнес-логіки з конекторами.
3) Connectors.Grpc: gRPC сервіс для комунікації між мікросервісами з метою отримання даних о конекторах.

д) Depots:

1) Depots.Api: API для взаємодії з депо.
2) Depots.Application: відповідає за роботу бізнес-логіки з роботи з депо.
3) Depots.Grpc: gRPC сервіс для комунікації між мікросервісами з метою отримання даних о депо.

ж) EnergyConsumption:

1) EnergyConsumption.Api: API для відстеження споживання енергії.
2) EnergyConsumption.Application: відповідає за роботу бізнес-логіки зі споживанням енергії.
3) EnergyConsumption.Grpc: gRPC сервіс для комунікації між мікросервісами з метою отримання даних щодо споживання енергії.

к) Gateway:

1) ChargingStation.Gateway: Шлюз для управління трафіком між клієнтами та сервісами.

л) Heartbeats:

1) ChargingStation.Heartbeats: моніторинг стану та доступності зарядних станцій.

м) OcppTags:

1) OcppTags.Api: API для роботи з OCPP тегами.
2) OcppTags.Application: відповідає за роботу бізнес-логіки управління OCPP тегами.
3) OcppTags.Grpc: gRPC сервіс для комунікації між мікросервісами з використанням OCPP тегів.

н) Reservations:

1) Reservations.Api: API для управління бронюваннями.
2) Reservations.Application: відповідає за роботу бізнес-логіки для роботи з бронюваннями.
3) Reservations.Grpc: gRPC сервіс для комунікації між мікросервісами з метою отримання даних щодо бронювань.

п) SignalR:

1) ChargingStation.SignalR: реалізує функціонал для реального часу комунікації з використання технлогії SignalR.

р) Transactions:

1) Transactions.Api: API для управління транзакціями.
2) Transactions.Application: відповідає за роботу бізнес-логіки для роботи з транзакціями.
3) Transactions.Grpc: gRPC сервіс для комунікації між мікросервісами з метою отримання даних щодо транзакцій.

с) UserManagement:

1) UserManagement.API: API для управління користувачами.

т) WebSockets:

1)ChargingStation.WebSockets: Реалізація WebSockets для забезпечення двостороннього зв'язку між зарядною станцією і сервером в реальному часі.

Docker-compose:

– docker-compose: файл для конфігурації та розгортання всіх сервісів у контейнерах Docker, забезпечуючи їх взаємодію та масштабування.

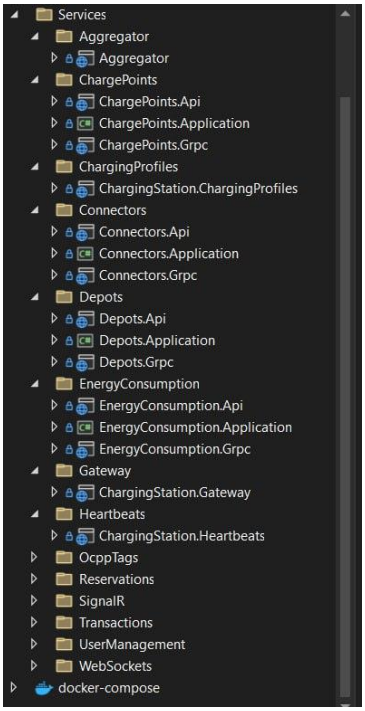


Рисунок 6.2 - Сервіси (рисунок виконано самостійно)

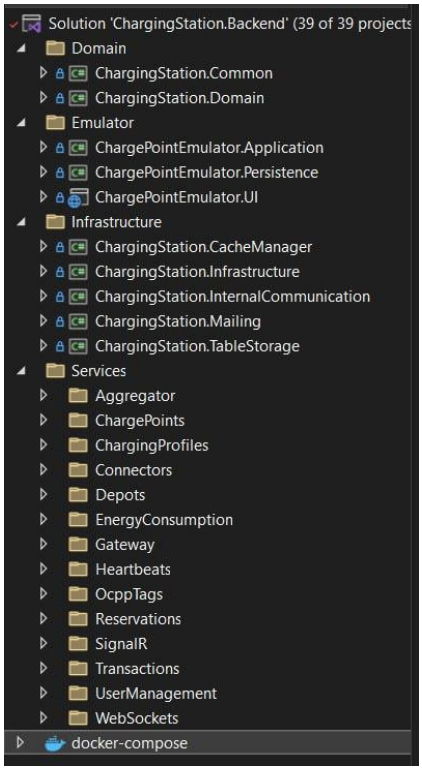


Рисунок 6.3 - Розгорнута структура проекту (рисунок виконано самостійно)

Ці модулі разом забезпечують повний функціонал системи управління зарядними станціями, включаючи емуляцію, внутрішню комунікацію, управління користувачами та інші необхідні функції.

6.2 Використання Ocelot

В проекті було використано ряд технологій. Для розробки та реалізації функціоналу в серверній частині проекту були задіяні:

- Ocelot (див. рис. 6.4) - це легкий API шлюз, розроблений спеціально для платформ .NET. Він дозволяє об'єднувати, маршрутизувати та захищати HTTP-запити між клієнтами і мікросервісами або іншими кінцевими точками.
- ASP.NET Core — це кросплатформенний, високопродуктивний, відкритий фреймворк для побудови сучасних, хмарних, інтернет-зв'язаних додатків.
- gRPC (Google Remote Procedure Call) — це високоефективний, кросплатформенний фреймворк для віддалених викликів процедур (RPC), розроблений компанією Google. Він дозволяє клієнтським і серверним

додаткам взаємодіяти між собою прозорим чином, викликаючи методи на віддалених серверах так, ніби це локальні виклики.

– OCPP (Open Charge Point Protocol) — це відкритий стандарт протоколу комунікації між зарядними станціями для електромобілів (EV) та центральними системами управління. Він дозволяє інтероперабельність та спільну роботу зарядних станцій різних виробників із центральними системами, забезпечуючи стандартизацію і взаємодію у сфері зарядної інфраструктури для електромобілів.
– SignalR — це бібліотека для .NET, яка спрощує процес додавання реального часу веб-функціональності до ваших додатків.]

Для обміну повідомленнями та сховищем проекту були задіяні:

– Redis (REmote DIctionary Server) — це високопродуктивна база даних, що зберігає дані в пам'яті (in-memory), з відкритим вихідним кодом. Вона використовується як кеш, брокер повідомлень та іноді як основна база даних.
– Microsoft SQL Server (MS SQL Server) — це система управління реляційними базами даних (RDBMS). Більш детально про реалізацію цієї технології можна дізнатися у співкомандника по проєкту - Чубарова Є.Е..
– RabbitMQ — це система брокера повідомлень з відкритим вихідним кодом, яка реалізує протокол Advanced Message Queuing Protocol (AMQP) . Вона дозволяє різним компонентам системи обмінюватися повідомленнями і забезпечує асинхронну комунікацію між ними.
– Azure Table Storage — це сервіс збереження NoSQL даних від Microsoft Azure, призначений для зберігання структурованих даних у вигляді таблиць. Він забезпечує масштабованість, високу доступність і низькі витрати на зберігання великих обсягів даних .

```
{
  "DownstreamPathTemplate": "/api/depot/{id}",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "depots.api",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/api/depot/{id}",
  "UpstreamHttpMethod": [ "GET" ]
},
{
  "DownstreamPathTemplate": "/api/depot",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "depots.api",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/api/depot",
  "UpstreamHttpMethod": [ "POST" ]
},
```

Рисунок 6.4 - Використання Ocelot (рисунок виконано самостійно)

Рисунок 6.4 показує конфігураційний файл маршрутизації для API шлюзу, з використанням Ocelot - популярного .NET API Gateway. Конфігурація визначає маршрути для перенаправлення HTTP-запитів від клієнтів до сервісів мікросервісної архітектури. Бачимо 2 маршрути: перший - для запиту GET, другий - для запиту POST.

Перший маршрут обробляє GET запити до "/api/depot/{id}". Запити перенаправляються до сервісу "depots.api" на порт 8080.

Другий маршрут обробляє POST запити до "/api/depot". Запити перенаправляються до сервісу "depots.api" на порт 8080.

Детальніше про реалізацію:

- DownstreamPathTemplate: шаблон шляху для внутрішнього (downstream) сервісу.
- DownstreamScheme: схема протоколу для downstream сервісу (HTTP).
- DownstreamHostAndPorts: хост і порт для downstream сервісу.
- UpstreamPathTemplate: шаблон шляху для зовнішнього (upstream) запиту.

– UpstreamHttpMethod: HTTP метод, який обробляється (GET або POST).

Коли клієнт надсилає запит до "/api/depot/{id}" з методом GET, цей запит перенаправляється до внутрішнього сервісу "http://depots.api:8080/api/depot/{id}".

Коли клієнт надсилає запит до "/api/depot" з методом POST, цей запит перенаправляється до внутрішнього сервісу "http://depots.api:8080/api/depot".

Це дозволяє абстрагувати внутрішню структуру сервісів від клієнтів і забезпечує централізовану точку доступу до них через API шлюз.

6.3 Використання ASP.NET Core

Веб-додаток реалізован за допомогою ASP.NET Core. Далі акцентуємо увагу на деякі реалізовані частини функціоналу нашого проекту.

Для передбачування часу зарядки (див. рис. 6.5) за допомогою екстраполяції, ми скористалися бібліотекою Math.NET, яка надає потужні інструменти для обробки даних і математичного моделювання. Для передбачення часу зарядки на основі наявних даних про стан заряду (SoC) за допомогою екстраполяції та Math.NET, ми можемо скористатися методом лінійної регресії. Процес включає кілька кроків.

Збір даних про стан заряду (SoC) і час. Дані про стан заряду та відповідний час їх вимірювання збираються з бази даних або з запитів клієнтів. Ці дані зазвичай представлені як пари значень: час вимірювання і відповідний стан заряду.

Побудова моделі на основі лінійної регресії. Використовуючи бібліотеку Math.NET, ми можемо побудувати лінійну модель, яка описує залежність між часом зарядки і станом заряду. Для цього застосовується метод найменших квадратів, який знаходить лінію, що найкраще підходить до наявних даних.

Екстраполяція для передбачення часу завершення зарядки. Використовуючи отриману лінійну модель, можна передбачити час, необхідний для досягнення повного заряду (SoC = 100%). Для цього ми вирішуємо рівняння щодо часу x.

```
1 reference
private async Task<DateTime> CalculateChargingEndTime(Guid transactionId, DateTime transactionStartTime, CancellationToken cancellationToken = default)
{
    var meterValues = new List<SoCDateTime>
    {
        new()
        {
            MeterValueTimestamp = transactionStartTime,
            SoCValue = 0
        }
    };

    var meterValuesFromDb :List<SoCDateTime> = await _connectorMeterValueRepository.GetSoCForTransactionAsync(transactionId, cancellationToken);
    meterValues.AddRange(meterValuesFromDb);

    var times :double[] = meterValues.Select(x :SoCDateTime => (x.MeterValueTimestamp - transactionStartTime).TotalSeconds).ToArray();
    var charges :double[] = meterValues.Select(x :SoCDateTime => x.SoCValue).ToArray();

    var (intercept :double, slope :double) = Fit.Line( x: times, y: charges);

    var remainingCharge :double = 100 - charges[^1];
    var timeToFullCharge :double = (remainingCharge - intercept) / slope;

    var endTime = transactionStartTime.AddSeconds(times[^1] + timeToFullCharge);
    return endTime;
}
```

Рисунок 6.5 - Передбачування часу зарядки (рисунок виконано самостійно)

Збір даних про стан заряду. Код збирає дані про стан заряду з запитів клієнтів (MeterValuesRequest). Кожне значення стану заряду зберігається в базі даних, якщо воно має значення.

Використання Math.NET для побудови лінійної регресійної моделі. У коді є метод CalculateChargingEndTime, який викликає функцію Fit.Line з Math.NET для побудови лінійної моделі на основі зібраних даних. Функція Fit.Line повертає нахил і зсув лінії, які використовуються для передбачення часу завершення зарядки (див. рис. 6.6).

```
var (intercept :double, slope :double) = Fit.Line( x: times, y: charges);
```

Рисунок 6.6 - Функція Fit.Line (рисунок виконано самостійно)

Екстраполяція часу до повного заряду. Метод CalculateChargingEndTime обчислює залишок заряду, необхідний до досягнення 100%, і використовує нахил та зсув лінійної моделі для визначення часу, необхідного для досягнення цього рівня заряду. Результат екстраполяції використовується для повідомлення клієнта про очікуваний час завершення зарядки.

Загалом, застосування лінійної регресії з Math.NET для передбачення часу зарядки є ефективним підходом, що добре підходить для задач із досить простими

залежностями. Однак, для більш складних випадків можуть знадобитися інші методи машинного навчання або розширення існуючої моделі.

Директорія Specifications в проекті ChargingStation.Infrastructure (див. рис. 6.7) містить файли, що забезпечують специфікації для системи зарядних станцій. Ці файли допомагають створювати та керувати логічними умовами і запитами.

– ExpressionBuilder.cs. Допоміжний клас для створення динамічних LINQ-виразів.
– Specification.cs. Абстрактний клас, що визначає специфікації для різних запитів і умов (див. рис.6.7).

Ці файли дозволяють формулювати гнучкі та динамічні запити до бази даних, що полегшує фільтрацію та відбір даних відповідно до заданих критеріїв.

```
protected void AddFilter(Expression<Func<TEntity, bool>> filterExpression)
{
    if(Filter is not null)
        Filter = ExpressionBuilder<TEntity>.AndAlso(Filter, filterExpression);
    else
        Filter = filterExpression;
}

protected void AddInclude(string includeString) => Includes.Add(includeString);

private void AddOrderBy(Expression<Func<TEntity, object>> orderByExpression, bool isDescending = false)
{
    OrderBy = orderByExpression;
    IsDescendingOrderBy = isDescending;
}

private void AddThenBy(Expression<Func<TEntity, object>> thenByExpression, bool isDescending = false)
{
    ThenBy = thenByExpression;
    IsDescendingThenBy = isDescending;
}

private readonly ConcurrentDictionary<string, PropertyInfo> DiscoveredProperties = new();

protected void AddSorting(IEnumerable<OrderPredicate> orderPredicates)
{
    var significantPredicates = orderPredicates
        .Where(x => !string.IsNullOrEmpty(x.PropertyName))
        .Distinct();

    foreach (var predicate in significantPredicates)
    {
        var property = DiscoverProperty(predicate.PropertyName);

        if (!property.PropertyType.IsValueType && property.PropertyType != typeof(string))
            throw new NotSupportedException("Property type is not supported.");

        if (OrderBy is null)
            AddOrderBy(property.Name, predicate.OrderDirection);
        else
            AddThenBy(property.Name, predicate.OrderDirection);
    }
}

private PropertyInfo DiscoverProperty(string propertyName)
{
    if (DiscoveredProperties.TryGetValue(propertyName, out var property))
        return property;

    property = typeof(TEntity).GetProperty(
        propertyName, BindingFlags.IgnoreCase | BindingFlags.Instance | BindingFlags.Public);

    if (property == null)
        throw new InvalidOperationException("Invalid property name.");

    DiscoveredProperties.TryAdd(propertyName, property);

    return property;
}

private void AddOrderBy(string propertyName, OrderDirection option)
{
    var expression = ExpressionBuilder<TEntity>.OrderByExpression(propertyName);

    switch (option)
    {
        case OrderDirection.Descending:
            AddOrderBy(expression, true);
            break;
        case OrderDirection.Ascending:
            AddOrderBy(expression);
            break;
        default:
            throw new NotSupportedException("Order direction is not supported.");
    }
```

Рисунок 6.7 - Specification.cs (рисунок виконано самостійно)

Specification<TEntity> є абстрактним класом, що визначає умови для фільтрації, сортування та включення пов'язаних даних при роботі з сутностями в системі зарядних станцій.

Фільтрація:

– Filter: лямбда-вираз для фільтрації даних.

– AddFilter: додає нову умову фільтрації.

Включення:

– Includes: список пов'язаних сутностей, які потрібно включити.

– AddInclude: додає нову сутність до списку включень.

Сортування:

– OrderBy, ThenBy: лямбда-вирази для сортування.

– AddOrderBy, AddThenBy: додають вирази для сортування.

– AddSorting: додає сортування на основі наданих умов.

Відкриття властивостей:

– DiscoverProperty: метод для отримання властивостей сутностей.

Клас Specification<TEntity> дозволяє формулювати динамічні запити до бази даних з різними умовами фільтрації, включення та сортування, що полегшує роботу з даними.

Також було використано JWT-токен для реалізації автентифікації. Для реєстрації було використано бібліоткеку AspNetCore.Identity, а також JWT-токен для безпечної передачі інформації між серверною та клієнтською частинами.

AuthService, (див. додаток В) який відповідає за автентифікацію і реєстрацію користувачів в системі та підтвердження реєстрації. Давайте розглянемо структуру.

– Інтерфейси. IAuthService.cs: інтерфейс для сервісу автентифікації, який визначає основні методи, такі як Login, Register та ConfirmRegistration

– Реалізації. AuthService.cs: клас, який реалізує інтерфейс IAuthService і містить основну логіку для автентифікації, логування та реєстрації користувачів, використовуюч різні залежності, такі як UserManager, SignInManager та JwtHandler. UserManager надається бібліотекою AspNetCore.Identity, JwtHandler використовується для генерації токену

(див. рис. 6.8), EmailService для відправки повідомлення на пошту для завершення етапу реєстрації.

– Моделі. LoginRequest.cs (модель запиту для логіну), RegisterRequest.cs (модель запиту для реєстрації), ConfirmRegistrationRequest (модель запиту на завершення реєстрації).

```
public string GenerateAuthToken(ApplicationUser user, List<string> roles, DateTime expires)
{
    var claims = new List<Claim>
    {
        new( type: ClaimTypes.NameIdentifier,  value: user.Id.ToString()),
        new( type: ClaimTypes.Name,  value: $"{user.FirstName} {user.LastName}"),
        new( type: ClaimTypes.Email,  value: user.Email)
    };

    foreach (var role :string  in roles)
    {
        claims.Add( item: new Claim( type: ClaimTypes.Role,  value: role));
    }

    return GenerateToken(claims.ToArray(), expires);
}
```

Рисунок 6.8 - Генерація токена (рисунок виконано самостійно)

ChargingStation.Mailing відповідає за надсилання електронних листів в системі зарядних станцій. Для написання ми використовували бібліотеку MailJet.

Дозволяє автоматизувати процес надсилання повідомлень користувачам, включаючи підтвердження реєстрації, нотифікації про стан зарядки та інші повідомлення.

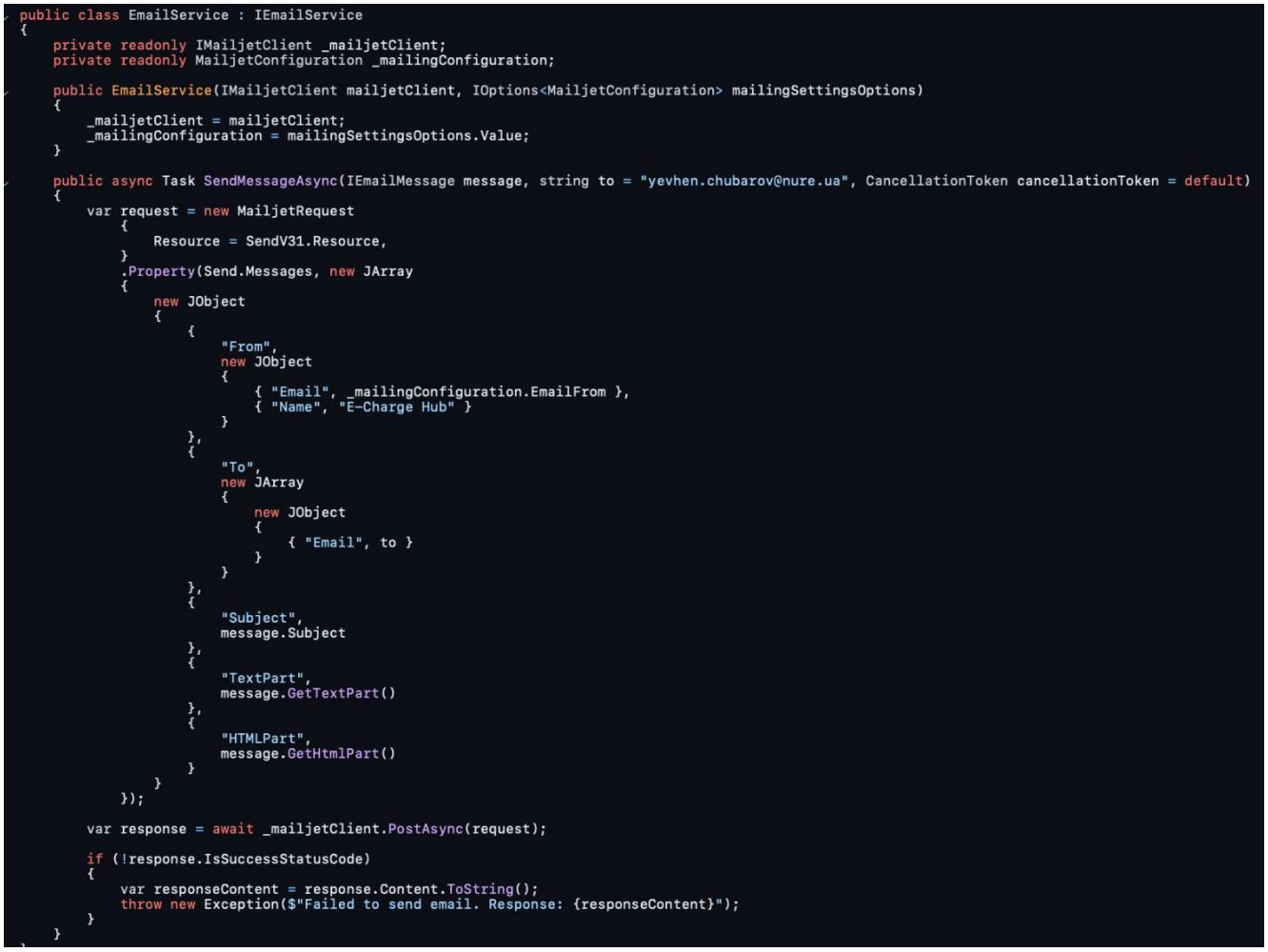
Services: клас реалізує логіку для відправки електронних листів (див. рис. 6.9).

Ін'єкція залежностей: Клієнт Mailjet і конфігурація надходять через конструктор.

Створення запиту: Формується запит для Mailjet API з усіма необхідними даними (відправник, отримувач, тема, текст і HTML частини).

Відправка запиту: Запит відправляється асинхронно.

Перевірка відповіді: Якщо відповідь неуспішна, генерується виключення з деталями помилки.



```
public class EmailService : IEmailService
{
    private readonly IMailjetClient _mailjetClient;
    private readonly MailjetConfiguration _mailingConfiguration;

    public EmailService(IMailjetClient mailjetClient, IOptions<MailjetConfiguration> mailingSettingsOptions)
    {
        _mailjetClient = mailjetClient;
        _mailingConfiguration = mailingSettingsOptions.Value;
    }

    public async Task SendMessageAsync(IEmailMessage message, string to = "yevhen.chubarov@nure.ua", CancellationToken cancellationToken = default)
    {
        var request = new MailjetRequest
            {
                Resource = SendV31.Resource,
            }
            .Property(Send.Messages, new JArray
            {
                new JObject
                {
                    {
                        "From",
                        new JObject
                        {
                            { "Email", _mailingConfiguration.EmailFrom },
                            { "Name", "E-Charge Hub" }
                        }
                    },
                    {
                        "To",
                        new JArray
                        {
                            new JObject
                            {
                                { "Email", to }
                            }
                        }
                    },
                    {
                        "Subject",
                        message.Subject
                    },
                    {
                        "TextPart",
                        message.GetTextPart()
                    },
                    {
                        "HTMLPart",
                        message.GetHtmlPart()
                    }
                }
            });

        var response = await _mailjetClient.PostAsync(request);

        if (!response.IsSuccessStatusCode)
        {
            var responseContent = response.Content.ToString();
            throw new Exception($"Failed to send email. Response: {responseContent}");
        }
    }
```

Рисунок 6.9 - EmailService.cs (рисунок виконано самостійно)

Наступним кроком опишемо реалізацію gRPC в нашому проекті.

6.4 Використання gRPC

GrpcClients є частиною інфраструктури для внутрішньої комунікації, зокрема через gRPC. gRPC (Google Remote Procedure Call) - це фреймворк для віддалених викликів процедур (RPC), який дозволяє клієнтам і серверам взаємодіяти між собою.

Каталог GrpcClients може містити різні клієнти для роботи з gRPC сервісами. До прикладу розглянемо TransactionGrpcClientService (див. рис. 6.10).

```
public class TransactionGrpcClientService
{
    private readonly TransactionsGrpc.TransactionsGrpcClient _transactionsGrpcClient;

    0 references
    public TransactionGrpcClientService(TransactionsGrpc.TransactionsGrpcClient transactionsGrpcClient)
    {
        _transactionsGrpcClient = transactionsGrpcClient;
    }

    1 reference
    public async Task<TransactionResponse> GetByIdAsync(Guid transactionId, CancellationToken cancellationToken = default)
    {
        var grpcRequest = new GetTransactionByIdGrpcRequest
        {
            Id = transactionId.ToString()
        };

        var grpcResponse = await _transactionsGrpcClient.GetByIdAsync(grpcRequest, cancellationToken: cancellationToken);

        var response = grpcResponse.ToResponse();
        return response;
    }
}
```

Рисунок 6.10 - TransactionGrpcClientService (рисунок виконано самостійно)

Цей сервіс реалізує (див. рис. 6.11) клієнт для взаємодії з gRPC сервісом транзакцій. Він надає метод для отримання інформації про транзакцію за її ідентифікатором. Він використовує згенерований клієнт gRPC для відправки запитів і обробки відповідей. Сервіс асинхронний, що дозволяє не блокувати потік виконання під час очікування відповіді від сервера.

```
import "google/protobuf/timestamp.proto";
import "google/protobuf/wrappers.proto";

option csharp_namespace = "Transactions.Grpc.Protos";

package transactions;

service TransactionsGrpc {
  rpc GetById (GetTransactionByIdGrpcRequest) returns (TransactionGrpcResponse);
}

message GetTransactionByIdGrpcRequest {
  string id = 1;
}

message TransactionGrpcResponse {
  string id = 1;
  int32 transaction_id = 2;
  string start_tag_id = 3;
  google.protobuf.StringValue stop_tag_id = 4;
  google.protobuf.Timestamp start_time = 5;
  google.protobuf.Timestamp stop_time = 6;
  google.protobuf.Timestamp created_at = 7;
  google.protobuf.Timestamp updated_at = 8;
  google.protobuf.StringValue stop_reason = 9;
  string connector_id = 10;
  google.protobuf.StringValue reservation_id = 11;
}
```

Рисунок 6.11 - Протокол транзакцій (рисунок виконано самостійно)

Цей прото-файл визначає gRPC сервіс для отримання інформації про транзакцію за її ідентифікатором. Він містить одне RPC метод GetById, який приймає запит з ідентифікатором транзакції і повертає розширену відповідь з детальною інформацією про транзакцію, включаючи час початку і завершення, ідентифікатори тегів та інші поля. Використання google.protobuf.StringValue та google.protobuf.Timestamp забезпечує більш гнучку і точну обробку даних.

```
var connector = await _connectorGrpcClientService.GetByIdAsync(transaction.ConnectorId, cancellationToken);
```

Рисунок 6.12 - Рядок визову GrpcClients (рисунок виконано самостійно)

Цей протокол описує сервіс, який було згадано раніше. Визов методу з ChargingProfileService (див. рис. 6.12). Ми застосовуємо виклик цього методу з метою не порушення принципів мікросервісної архітектури, коли один мікросервіс має одну зону відповідальності і не повинен займатися роботою іншого мікросервісу.

6.5 Використання OCPP

Open Charge Point Protocol (OCPP) — це відкритий стандарт для комунікації між зарядними станціями для електромобілів та центральними системами керування. Розроблений Open Charge Alliance (OCA), OCPP забезпечує взаємодію між різним обладнанням та програмним забезпеченням.

OcppWebSocketConnectionHandler (див. додаток В) відповідає за обробку підключень і повідомлень від зарядних станцій через протокол OCPP (Open Charge Point Protocol) за допомогою WebSocket.

WebSocketMiddleware (див. рис. 6.13) - проміжне ПЗ для обробки WebSocket з’єднань.

Процес прийняття повідомлень та методи відправки відповідей.

встановлення з'єднання. Коли клієнт (зарядна станція) ініціює WebSocket з'єднання, WebSocketMiddleware приймає це з'єднання та передає його до OcppConnectionHandler.

– прийняття повідомлень. OcppConnectionHandler приймає повідомлення від зарядної станції в асинхронному режимі, розшифровує їх та обробляє.
– закриття з'єднання. Коли з'єднання закривається, OcppConnectionHandler обробляє закриття з'єднання і виконує необхідні дії.
– відправки повідомлень OCPP на зарядні станції. Це можуть бути відповіді на запити або повідомлення про помилки.
– відправки запиту на скидання зарядної станції.

```
public class OcppWebSocketMiddleware
{
    private readonly RequestDelegate _next;

    0 references
    public OcppWebSocketMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    0 references
    public async Task InvokeAsync(HttpContext context, IOcppWebSocketConnectionHandler ocppWebSocketConnectionHandler,
        ILogger<OcppWebSocketMiddleware> logger)
    {
        if (context.WebSockets.IsWebSocketRequest)
        {
            await ocppWebSocketConnectionHandler.HandleConnectionAsync(_next, context);
            return;
        }

        // passed on to next middleware
        await _next(context);
    }
}
```

Рисунок 6.13 - WebSocketMiddleware (рисунок виконано самостійно)

OcppConnectionHandlers відповідає за обробку підключень та повідомлень від зарядних станцій через WebSocket з використанням протоколу OCPP. Основні компоненти включають інтерфейси та їх реалізації для обробки підключень і повідомлень, проміжне ПЗ для обробки WebSocket з'єднань. Такий підхід дозволяє забезпечити ефективну комунікацію між зарядними станціями та сервером, що є критично важливим для коректної роботи інфраструктури зарядних станцій.

6.6 Використання SignalR

SignalR — це бібліотека для .NET, яка спрощує процес додавання реального часу веб-функціональності до ваших додатків. Реальний час означає, що сервер

може миттєво надсилати оновлення до підключених клієнтів. SignalR надає API для створення віддалених викликів процедур (RPC) між сервером і клієнтами, дозволяючи двосторонню комунікацію.

HubFacade (рис. 6.14): інкапсулює логіку надсилання повідомлень через SignalR хаб HubFacade: Метод SendCentralSystemMessageAsync: Надсилає повідомлення всім підключеним клієнтам, використовуючи вказаний тип повідомлення.

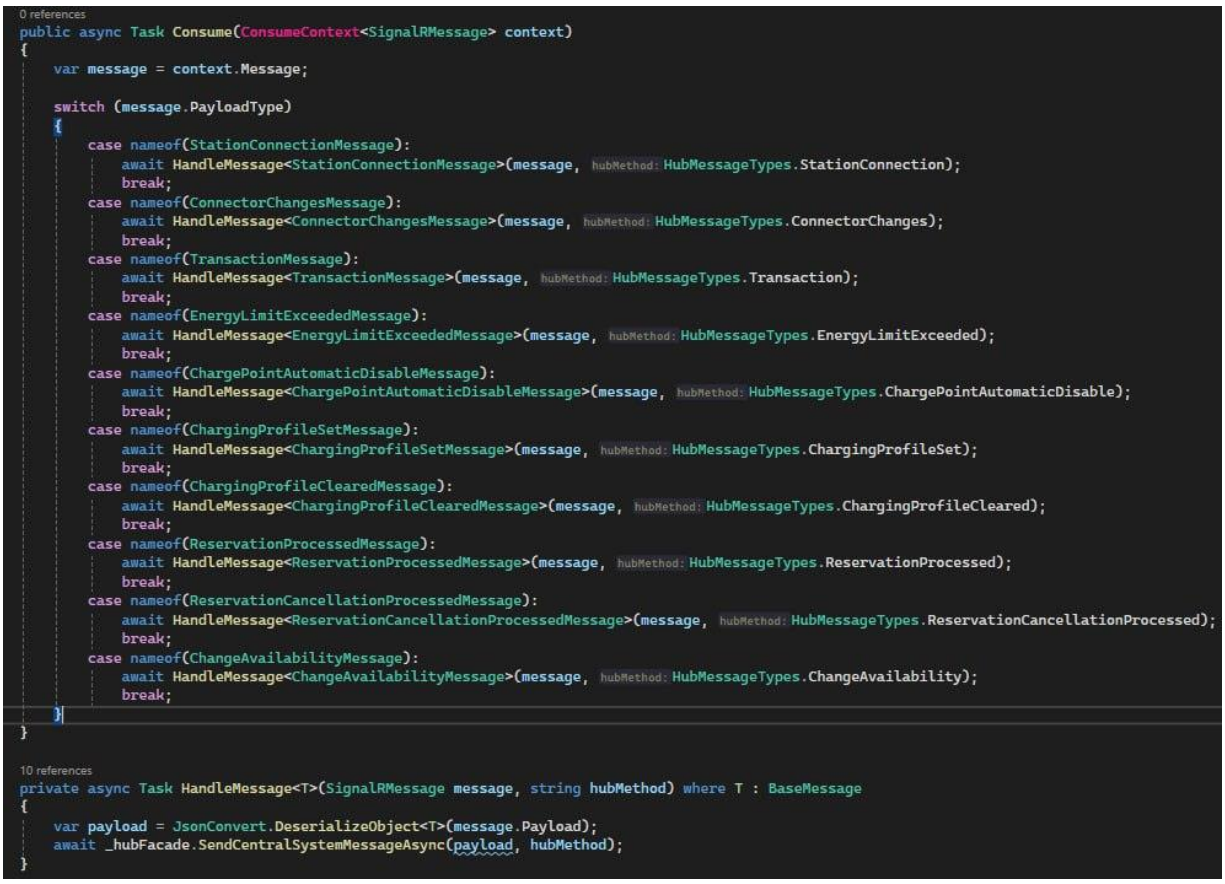
SignalRResponseConsumer: цей клас споживає повідомлення типу SignalRMessage через RabbitMQ

Метод Consume (див. рис. 7.15): цей метод отримує повідомлення, перевіряє його тип (PayloadType) і викликає відповідний метод для обробки.

Метод HandleMessage: десеріалізує повідомлення і передає його через HubFacade у SignalR.

```
public class HubFacade
{
    private readonly IHubContext<ChargingStationHub> _hubContext;

    0 references
    public HubFacade(IHubContext<ChargingStationHub> hubContext)
    {
        _hubContext = hubContext;
    }

    1 reference
    public async Task SendCentralSystemMessageAsync(BaseMessage message, string messageType)
    {
        await _hubContext.Clients.All.SendAsync(messageType, message);
    }
}
```

Рисунок 6.14 - Фасад для надсилання повідомлень через SignalR хаб (рисунок виконано самостійно)



```
0 references
public async Task Consume(ConsumeContext<SignalRMessage> context)
{
    var message = context.Message;

    switch (message.PayloadType)
    {
        case nameof(StationConnectionMessage):
            await HandleMessage<StationConnectionMessage>(message, hubMethod: HubMessageTypes.StationConnection);
            break;
        case nameof(ConnectorChangesMessage):
            await HandleMessage<ConnectorChangesMessage>(message, hubMethod: HubMessageTypes.ConnectorChanges);
            break;
        case nameof(TransactionMessage):
            await HandleMessage<TransactionMessage>(message, hubMethod: HubMessageTypes.Transaction);
            break;
        case nameof(EnergyLimitExceededMessage):
            await HandleMessage<EnergyLimitExceededMessage>(message, hubMethod: HubMessageTypes.EnergyLimitExceeded);
            break;
        case nameof(ChargePointAutomaticDisableMessage):
            await HandleMessage<ChargePointAutomaticDisableMessage>(message, hubMethod: HubMessageTypes.ChargePointAutomaticDisable);
            break;
        case nameof(ChargingProfileSetMessage):
            await HandleMessage<ChargingProfileSetMessage>(message, hubMethod: HubMessageTypes.ChargingProfileSet);
            break;
        case nameof(ChargingProfileClearedMessage):
            await HandleMessage<ChargingProfileClearedMessage>(message, hubMethod: HubMessageTypes.ChargingProfileCleared);
            break;
        case nameof(ReservationProcessedMessage):
            await HandleMessage<ReservationProcessedMessage>(message, hubMethod: HubMessageTypes.ReservationProcessed);
            break;
        case nameof(ReservationCancellationProcessedMessage):
            await HandleMessage<ReservationCancellationProcessedMessage>(message, hubMethod: HubMessageTypes.ReservationCancellationProcessed);
            break;
        case nameof(ChangeAvailabilityMessage):
            await HandleMessage<ChangeAvailabilityMessage>(message, hubMethod: HubMessageTypes.ChangeAvailability);
            break;
    }
}

10 references
private async Task HandleMessage<T>(SignalRMessage message, string hubMethod) where T : BaseMessage
{
    var payload = JsonConvert.DeserializeObject<T>(message.Payload);
    await _hubFacade.SendCentralSystemMessageAsync(payload, hubMethod);
}
```

Рисунок 6.15 - Споживає повідомлення від RabbitMQ і передає їх через SignalR хаб до клієнтів. (рисунок виконано самостійно)

ChargingStationHub (див. рис. 6.16): порожній клас хабу, який успадковує від Hub. Використовується як точка підключення клієнтів до SignalR.

```
3 references
public class ChargingStationHub : Hub
{ }
```

Рисунок 6.16 - клас ChargingStationHub (рисунок виконано самостійно)

Взаємодія компонентів:

—отримання повідомлення: коли SignalRResponseConsumer отримує повідомлення через MassTransit, він визначає тип повідомлення (PayloadType).

—обробка повідомлення: SignalRResponseConsumer викликає метод HandleMessage, який десеріалізує повідомлення і викликає відповідний метод у HubFacade.

—надсилання повідомлення: HubFacade використовує SignalR для надсилання повідомлення всім підключеним клієнтам через метод SendCentralSystemMessageAsync.

6.7 Використання Redis

Redis (див. рис. 6.17) — це високопродуктивна база даних, що зберігає дані в пам'яті (in-memory). Вона використовується як кеш, брокер повідомлень та іноді як основна база даних. Redis підтримує різноманітні структури даних, такі як рядки, списки, множини, хеші, бітові карти, гіперлоглоги і просторові індекси. На рисунку 6.18 можна побачити використання Redis.

```
public class CacheManager : ICacheManager
{
    private readonly IDatabase _database;

    0 references
    public CacheManager(IConnectionMultiplexer connectionMultiplexer)
    {
        _database = connectionMultiplexer.GetDatabase();
    }

    4 references
    public async Task<T?> GetAsync<T>(string key)
    {
        var value = await _database.StringGetAsync( key);

        if (value.IsNullOrEmpty)
        {
            return default;
        }

        return value.HasValue ? JsonConvert.DeserializeObject<T>( value) : default;
    }

    4 references
    public Task RemoveAsync(string key)
    {
        return _database.KeyDeleteAsync( key);
    }

    5 references
    public async Task SetAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        await _database.StringSetAsync( key,  value:  JsonConvert.SerializeObject(value), expiry);
    }
}
```

Рисунок 6.17 - Реалізація Redis в проєкті (рисунок виконано самостійно)

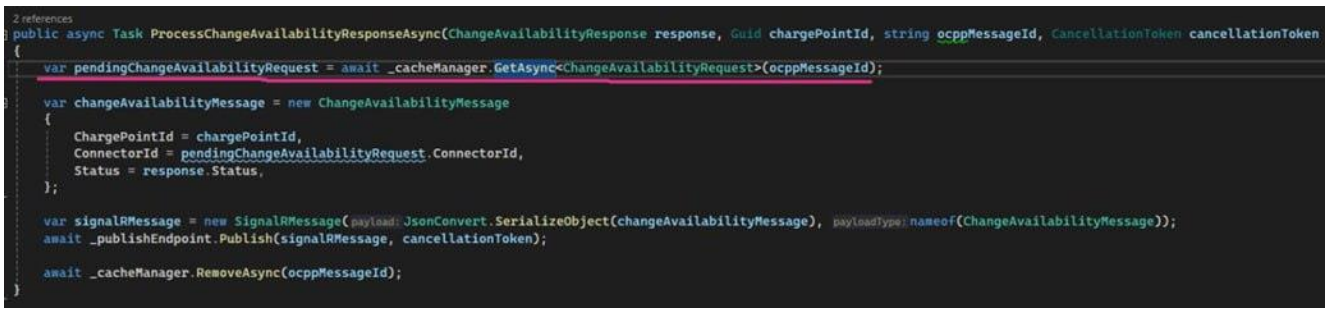


Рисунок 6.18 - Використання Redis (рисунок виконано самостійно)

Клас CacheManager є реалізацією інтерфейсу ICacheManager, який використовується для управління кешем за допомогою бібліотеки StackExchange.Redis. Цей клас забезпечує основні операції з кешування, такі як зчитування, збереження та видалення даних з Redis.

6.8 Використання RabbitMQ

RabbitMQ — це система брокера повідомлень з відкритим вихідним кодом, яка реалізує протокол Advanced Message Queuing Protocol (AMQP). Вона дозволяє різним компонентам системи обмінюватися повідомленнями і забезпечує асинхронну комунікацію між ними.

Реалізація обробки повідомлень Open Charge Point Protocol (OCPP) для старту транзакцій у проекті. Використовується ієрархія класів для обробки різних типів повідомлень OCPP, де кожен клас відповідає за конкретну операцію.

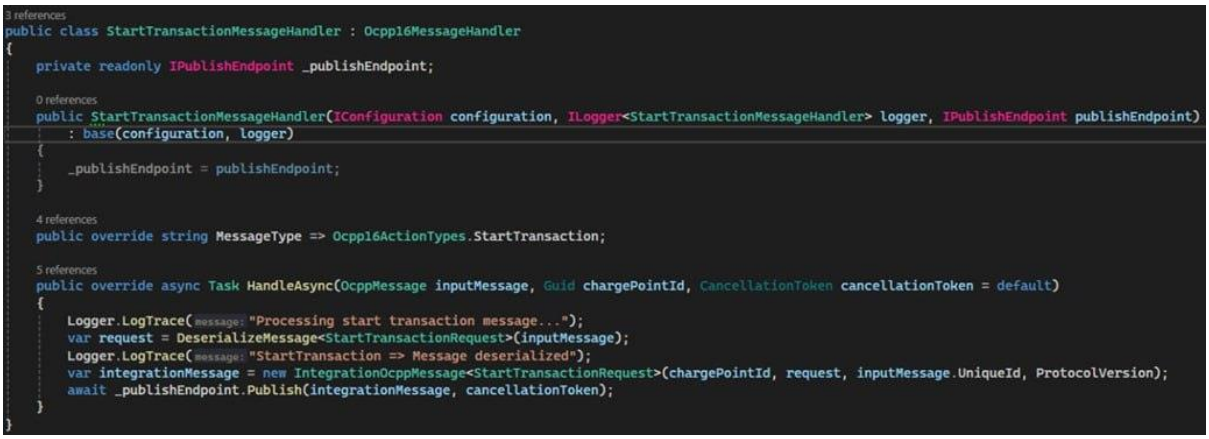


Рисунок 6.19 - Обробка повідомлень типу StartTransaction (рисунок виконано самостійно)

Цей клас (див. рис. 6.19) обробляє повідомлення типу StartTransaction. Він успадковує від Ocpp16MessageHandler і реалізує специфічну логіку для обробки повідомлень про старт транзакції.

Цей абстрактний клас (див. рис. 6.20) спеціалізований для обробки повідомлень протоколу OCPP 1.6 і успадковує від OcppMessageHandler.

Абстрактний клас (див. додаток В) є базовим для всіх обробників повідомлень OCPP. Він визначає основні методи і властивості, які успадковують специфічні обробники.

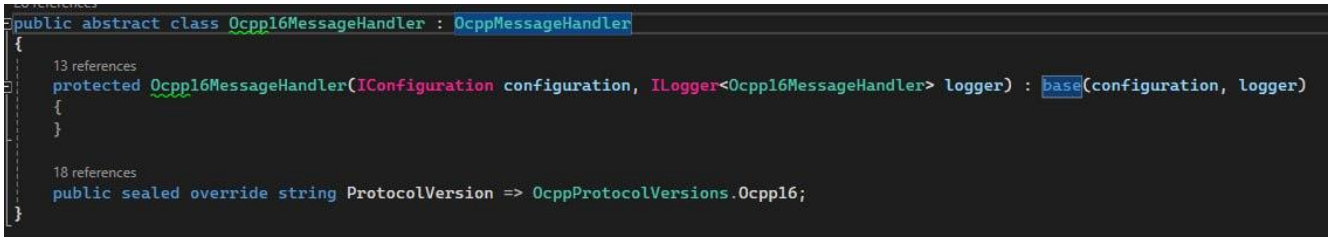


Рисунок 6.20 - Обробка повідомлень OCPP 1.6 (рисунок виконано самостійно)

Взаємодія компонентів:

а) StartTransactionMessageHandler:

1)Специфічний обробник для повідомлень типу StartTransaction.

2)Викликає метод DeserializeMessage, щоб десеріалізувати вхідне повідомлення.

3)Створює об'єкт IntegrationOcppMessage і публікує його через RabbitMQ.

б)Ocpp16MessageHandler:

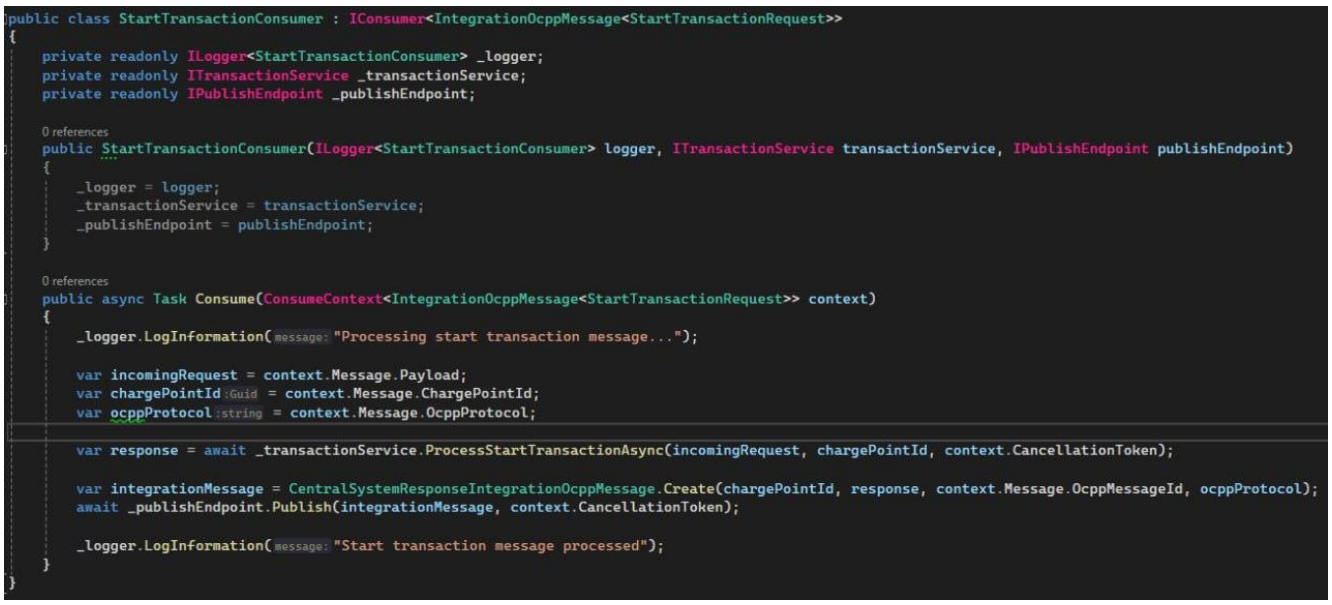
1)Спеціалізований абстрактний клас для обробки повідомлень протоколу OCPP 1.6.

2)Встановлює версію протоколу як Ocpp16.

в) OcppMessageHandler:

1) Базовий абстрактний клас для всіх обробників повідомлень OCPP.

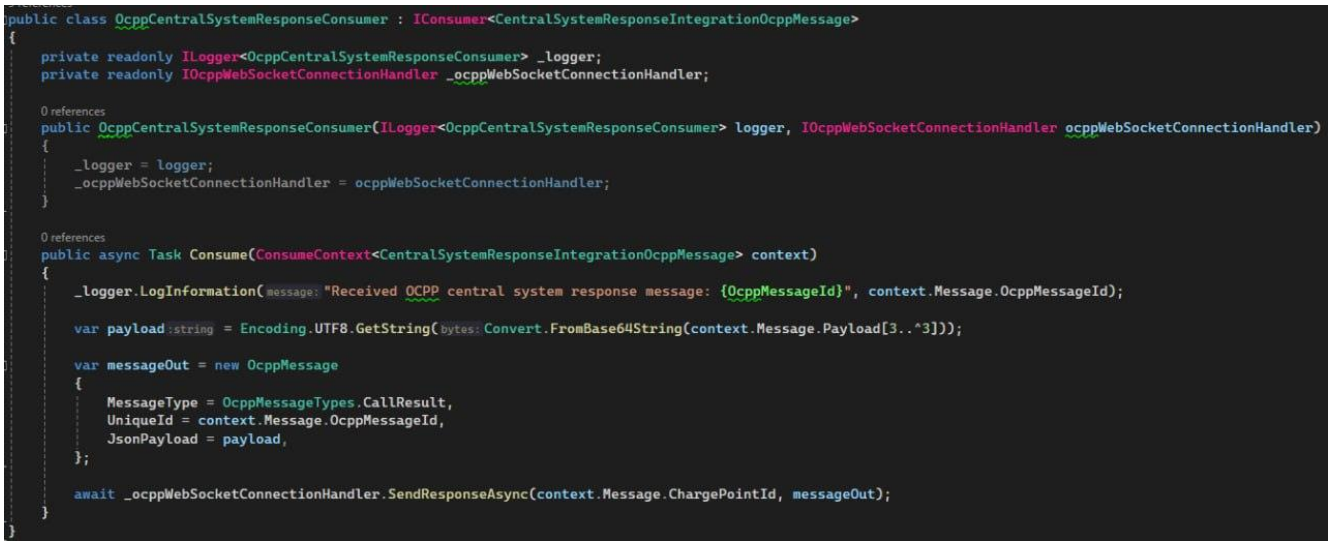
2)Визначає основні методи і властивості, включаючи DeserializeMessage, який десеріалізує повідомлення з можливістю валідації схеми JSON.



```
public class StartTransactionConsumer : IConsumer<IntegrationOcppMessage<StartTransactionRequest>>
{
    private readonly ILogger<StartTransactionConsumer> _logger;
    private readonly ITransactionService _transactionService;
    private readonly IPublishEndpoint _publishEndpoint;

    0 references
    public StartTransactionConsumer(ILogger<StartTransactionConsumer> logger, ITransactionService transactionService, IPublishEndpoint publishEndpoint)
    {
        _logger = logger;
        _transactionService = transactionService;
        _publishEndpoint = publishEndpoint;
    }

    0 references
    public async Task Consume(ConsumeContext<IntegrationOcppMessage<StartTransactionRequest>> context)
    {
        _logger.LogInformation(message: "Processing start transaction message...");

        var incomingRequest = context.Message.Payload;
        var chargePointId :Guid = context.Message.ChargePointId;
        var ocppProtocol :string = context.Message.OcppProtocol;

        var response = await _transactionService.ProcessStartTransactionAsync(incomingRequest, chargePointId, context.CancellationToken);

        var integrationMessage = CentralSystemResponseIntegrationOcppMessage.Create(chargePointId, response, context.Message.OcppMessageId, ocppProtocol);
        await _publishEndpoint.Publish(integrationMessage, context.CancellationToken);

        _logger.LogInformation(message: "Start transaction message processed");
    }
}
```

Рисунок 6.21 - Обробка повідомлень про початок транзакції (рисунок виконано самостійно)

Клас StartTransactionConsumer (див. рис. 6.21) відповідає за обробку повідомлень про початок транзакції. Він отримує повідомлення, обробляє їх за допомогою сервісу транзакцій і публікує відповідь у чергу для подальшої обробки.



```
public class OcppCentralSystemResponseConsumer : IConsumer<CentralSystemResponseIntegrationOcppMessage>
{
    private readonly ILogger<OcppCentralSystemResponseConsumer> _logger;
    private readonly IOcppWebSocketConnectionHandler _ocppWebSocketConnectionHandler;

    0 references
    public OcppCentralSystemResponseConsumer(ILogger<OcppCentralSystemResponseConsumer> logger, IOcppWebSocketConnectionHandler ocppWebSocketConnectionHandler)
    {
        _logger = logger;
        _ocppWebSocketConnectionHandler = ocppWebSocketConnectionHandler;
    }

    0 references
    public async Task Consume(ConsumeContext<CentralSystemResponseIntegrationOcppMessage> context)
    {
        _logger.LogInformation(message: "Received OCPP central system response message: {OcppMessageId}", context.Message.OcppMessageId);

        var payload :string = Encoding.UTF8.GetString(bytes: Convert.FromBase64String(context.Message.Payload[3..^3]));

        var messageOut = new OcppMessage
        {
            MessageType = OcppMessageTypes.CallResult,
            UniqueId = context.Message.OcppMessageId,
            JsonPayload = payload,
        };

        await _ocppWebSocketConnectionHandler.SendResponseAsync(context.Message.ChargePointId, messageOut);
    }
}
```

Рисунок 6.22 - Посилання обробленого повідомлення до зарядних станцій (рисунок виконано самостійно)

Клас OcppCentralSystemResponseConsumer (рис. 6.22) відповідає за обробку повідомлень від центральної системи. Він отримує повідомлення через RabbitMQ, декодує їх, і передає до зарядних станцій через WebSocket-з'єднання. Такий підхід забезпечує надійну і ефективну комунікацію між центральною системою і зарядними станціями, дозволяючи оперативно передавати інформацію і управляти процесом заряджання.

6.9 Використання Azure Table Storage

Azure Table Storage — це сервіс збереження NoSQL даних від Microsoft Azure, призначений для зберігання структурованих даних у вигляді таблиць.

Файл AzureTableStorageManager (див. рис. 6.23) містить клас AzureTableStorageManager, який забезпечує функціональність для роботи з таблицями в Azure Table Storage. Основні методи включають операції створення, зчитування, оновлення та видалення (CRUD). Клас використовує Azure SDK для взаємодії з Table Storage.

AzureTableStorageManager є важливим компонентом, який забезпечує роботу з Azure Table Storage, дозволяючи зберігати та отримувати дані ефективно і надійно. Він абстрагує специфічні деталі роботи з Azure Table Storage, забезпечуючи простий інтерфейс для інших частин проекту. Це робить код чистішим і зручнішим для підтримки.

```
using Azure.Data.Tables;
using ChargingStation.TableStorage.Models;

namespace ChargingStation.TableStorage.Managers;

public class AzureTableStorageManager<T> : ITableManager<T> where T : BaseTableStorageEntity
{
    private readonly TableServiceClient _tableServiceClient;

    public AzureTableStorageManager(TableServiceClient tableServiceClient)
    {
        _tableServiceClient = tableServiceClient;
    }

    public async Task AddEntityAsync(string tableName, T entity)
    {
        var tableClient = await GetTableClientAsync(tableName);
        await tableClient.AddEntityAsync(entity);
    }

    public async Task<T> GetEntityAsync(string tableName, string partitionKey, string rowKey)
    {
        var tableClient = await GetTableClientAsync(tableName);
        return await tableClient.GetEntityAsync<T>(partitionKey, rowKey);
    }

    public async Task<List<T>> GetEntitiesByPartitionKeyAsync(string tableName, string partitionKey)
    {
        var tableClient = await GetTableClientAsync(tableName);
        return await tableClient.QueryAsync<T>(filter: $"PartitionKey eq '{partitionKey}'").ToListAsync();
    }

    public async Task<List<T>> GetAllEntitiesAsync(string tableName)
    {
        var tableClient = await GetTableClientAsync(tableName);
        return await tableClient.QueryAsync<T>().ToListAsync();
    }

    public async Task UpsertEntityAsync(string tableName, T entity)
    {
        var tableClient = await GetTableClientAsync(tableName);
        await tableClient.UpsertEntityAsync(entity);
    }

    public async Task DeleteEntityAsync(string tableName, T entity)
    {
        var tableClient = await GetTableClientAsync(tableName);
        await tableClient.DeleteEntityAsync(entity.PartitionKey, entity.RowKey);
    }

    private async Task<TableClient> GetTableClientAsync(string tableName)
    {
        await _tableServiceClient.CreateTableIfNotExistsAsync(tableName);
        var tableClient = _tableServiceClient.GetTableClient(tableName);
        return tableClient;
    }
}
```

Рисунок 6.23 - AzureTableStorageManager (рисунок виконано самостійно)

Перейдемо до демонстрації Azure Table Storage в мікросервісі ChargingStation.Heartbeats.

ChargingStation.Heartbeats (див.рис. 6.24) відповідає за обробку та моніторинг стану зарядних станцій. Цей компонент забезпечує можливість відстежувати "серцебиття" (heartbeat) зарядних станцій, тобто регулярні сигнали, які підтверджують, що станція працює належним чином.

```
public class HeartbeatService : IHeartbeatService
{
    private readonly ChargePointGrpcClientService _chargePointGrpcClientService;
    private readonly ITableManager<HeartbeatEntity> _tableManager;
    private readonly string _tableName;

    public HeartbeatService(ChargePointGrpcClientService chargePointGrpcClientService, ITableManager<HeartbeatEntity> tableManager,
        IConfiguration configuration)
    {
        _chargePointGrpcClientService = chargePointGrpcClientService;
        _tableManager = tableManager;
        _tableName = configuration.GetValue<string>("TablesConfiguration:HeartbeatTable")!;
    }

    public async Task AddHeartbeatAsync(HeartbeatEntity request, CancellationToken cancellationToken = default)
    {
        var chargePoint = await _chargePointGrpcClientService.GetByIdAsync(Guid.Parse(request.PartitionKey), cancellationToken);

        if (chargePoint is null)
        {
            throw new NotFoundException($"ChargePointId {request.PartitionKey} not found");
        }

        await _tableManager.AddEntityAsync(_tableName, request);
    }

    public async Task<HeartbeatEntity> GetByIdAsync(GetHeartbeatRequest request,
        CancellationToken cancellationToken = default)
    {
        var heartbeat = await _tableManager.GetEntityAsync(_tableName, request.PartitionKey, request.RowKey);

        if (heartbeat is null)
            throw new NotFoundException(nameof(HeartbeatEntity), request);

        return heartbeat;
    }

    public async Task<List<HeartbeatEntity>> GetAsync(CancellationToken cancellationToken = default)
    {
        var heartbeats = await _tableManager.GetAllEntitiesAsync(_tableName);

        if (!heartbeats.Any())
            return Enumerable.Empty<HeartbeatEntity>().ToList();

        return heartbeats;
    }

    public async Task<HeartbeatResponse> ProcessHeartbeatAsync(HeartbeatRequest request, Guid chargePointId,
        CancellationToken cancellationToken = default)
    {
        var currentTime = DateTimeOffset.UtcNow;

        var response = new HeartbeatResponse(currentTime);
        var createHeartbeatRequest = new HeartbeatEntity(chargePointId.ToString(), currentTime);

        await AddHeartbeatAsync(createHeartbeatRequest, cancellationToken);

        return response;
    }
}
```

Рисунок 6.24 - HeartbeatService (рисунок виконано самостійно)

```
public class HeartbeatEntity : BaseTableStorageEntity
{
    2 references
    public DateTimeOffset CurrentTime { get; set; }

    0 references
    public HeartbeatEntity(string partitionKey, ETag eTag, DateTimeOffset currentTime) : base(partitionKey, eTag)
    {
        CurrentTime = currentTime;
    }

    1 reference
    public HeartbeatEntity(string partitionKey, DateTimeOffset currentTime) : base(partitionKey)
    {
        CurrentTime = currentTime;
    }
}
```

Рисунок 6.25 - HeartbeatEntity (рисунок виконано самостійно)

Всі операції з даними серцебиття виконуються через ITableManager<HeartbeatEntity> (див.рис. 7.25), який інкапсулює логіку взаємодії з Azure Table Storage:

—AddEntityAsync: Додає новий запис у таблицю.

—GetEntityAsync: Отримує запис з таблиці за заданими ключами.

—GetAllEntitiesAsync: Отримує всі записи з таблиці.

Клас використовує назву таблиці, що зберігається в конфігураційному файлі, для виконання операцій з даними серцебиття.

Цей сервіс забезпечує ефективне управління серцебиттями зарядних станцій, гарантуючи зберігання та доступ до даних у хмарному сховищі Azure Table Storage.

7 ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Шляхом аналізу та розробки програмної системи була підготовлена публікація «Проєктування серверної частини системи керування та моніторингу зарядними станціями» для виступу на конференції. «Інформаційні інтелектуальні системи» XXVIII Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті (див. додаток Г).

У роботі було досліджено проектування та створення серверної складової системи управління та моніторингу зарядних станцій для електромобілів, а також налаштування процесів CI/CD для автоматизації постачання рішень користувачам.

Також було опубліковано працю Горкуна Дмитра, який займається розробкою клієнтської частини нашого проекту, та Налєскіної Тетяни, проектного менеджера та тестувальника проекту, на тему «Проектування Angular додатку для програмної системи моніторингу зарядних станцій електромобілів з використанням Onion архітектури» на V Міжнародній студентській науковій конференції «Розвиток суспільства та науки в умовах цифрової трансформації».

У роботі було висвітлено тему проектування Angular-додатку для програмної системи моніторингу зарядних станцій електромобілів з використанням архітектури Onion. У цих тезах розглядаються ключові аспекти проектування клієнтської частини системи, зокрема використання Onion архітектури для забезпечення модульності та зручності у підтримці коду

ВИСНОВКИ

В даній кваліфікаційній роботі була розглянута та реалізована програмна система моніторингу зарядних станцій для електромобілів з фокусом на розробку серверної частини. Досліджено сучасний стан ринку електромобілів та зарядної інфраструктури, виявлено основні проблеми та потреби користувачів.

В ході дослідження було доведено, що використання мікросервісної архітектури, контейнеризації з використанням Docker та оркестрації з Kubernetes дозволяє побудувати високопродуктивну та масштабовану систему, яка забезпечує ефективне управління процесом зарядження електромобілів.

Технологічні рішення, такі як використання Microsoft Azure, Microsoft SQL Server, RabbitMQ, а також протоколи HTTP, AMQP, gRPC (HTTP/2) та OCPP через WebSocket, вдало поєднуються для створення безпечної, надійної та ефективної системи.

Особлива увага приділялася питанням безпеки даних, аутентифікації та авторизації, шифрування та захисту мережевого трафіку, а також моніторингу та аудиту безпеки. Всі ці аспекти сприяють створенню високоякісного та сучасного рішення для управління зарядженням електромобілів.

Результатом роботи є стабільна та функціональна серверна частина системи, яка забезпечує надійний моніторинг станцій зарядки, збір та аналіз даних, можливість віддаленого управління та інтеграцію з іншими системами. Розроблена програма має потенціал для подальшого розширення та удосконалення шляхом впровадження нових функцій та технологій.

Висновки роботи підтверджують актуальність теми та важливість подальших досліджень у галузі розвитку інфраструктури для електромобілів. Розроблена система може бути використана в комерційних проектах та сприяти подальшому зростанню ринку транспорту з електричним приводом.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Яких типів зарядних станцій потребує наявний парк електромобілів в Україні. Куди ринок буде рухатись далі? Частина 1: Авто новини від AUTO-Consulting - електро. Все про автобізнес: ринок автомобілів, автобусів, вантажівок. Статистика продажів. Продаж авто. AUTO-Consulting. URL: http://autoconsulting.ua/article.php?sid=53861.

2. Main Site - SMART CITY: ТЕХНОЛОГІЇ «розумного міста» ТА ЇХ ЦІЛЬОВЕ ПРИЗНАЧЕННЯ. Головна | Е-Україна E-Ukraine - INNOVATION PLATFORM FOR REFORMS. URL: https://eukraine.org.ua/ua/news/smart-city-tehnologiyi-rozumnogo-mista-ta-yih-cilove-priznachennya .

3. Типи і види зарядок для електромобілів - новини автомобільного світу від kitaec.ua. Автозапчасти на китайские авто - купить в интернет-магазин Китаец | kitaec.ua. URL: https://kitaec.ua/ua/articles/tipy-i-vidy-zaryadok-dlya-elektromobiley/.

4. All-in-one EV Charging Software - AMPECO. AMPECO. URL: https://www.ampeco.com.

5. ChargeLab: Electric vehicle charging software. ChargeLab: Electric vehicle charging software. URL: https://chargelab.co.

6. greenflux. URL: https://www.greenflux.com

7. Синкевич М. Е., Лєсна Н. С. ДОСЛІДЖЕННЯ ЗАСОБІВ І ТЕХНОЛОГІЙ МІЖСЕРВІСНОГО ЗВ'ЯЗКУ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ. *International Electronic Scientific Journal "Science Online" http://nauka-online.com/*. 2019. С. 1–2. URL: https://nauka-online.com/publications/informatsionnye-tehnologii/2019/5/doslidzhennya-zasobiv-i-tehnologij-mizhservisnogo-zv-yazku-v-mikroservisnij-arhitekturi/

8. Кульмінський А.К., Лановий О. Ф. ВИКОРИСТАННЯ ДАНИХ ЯК СЕРВіСУ ЗА ДОПОМОГОЮ ХМАРНИХ ТЕХНОЛОГіЙ. *БИОНИКА ИНТЕЛЛЕКТА*.2017.С. 181.URL: https://openarchive.nure.ua/server/api/core/bitstreams/e5237c28-b92a-4e6a-8c22-bd9aee8c9d53/content (дата звернення: 09.05.2024).

9. Zmerzlyi I. Мікросервісна архітектура. Medium. URL: https://medium.com/@IvanZmerzlyi/microservices-architecture-461687045b3d.

10. Пояснення OCPP: вичерпна інформація про бізнес | BENY Нова енергія. Beny New Energy | BENY Electric. URL: https://www.beny.com/uk/understanding-ocpp-and-how-it-benefits-you/ .

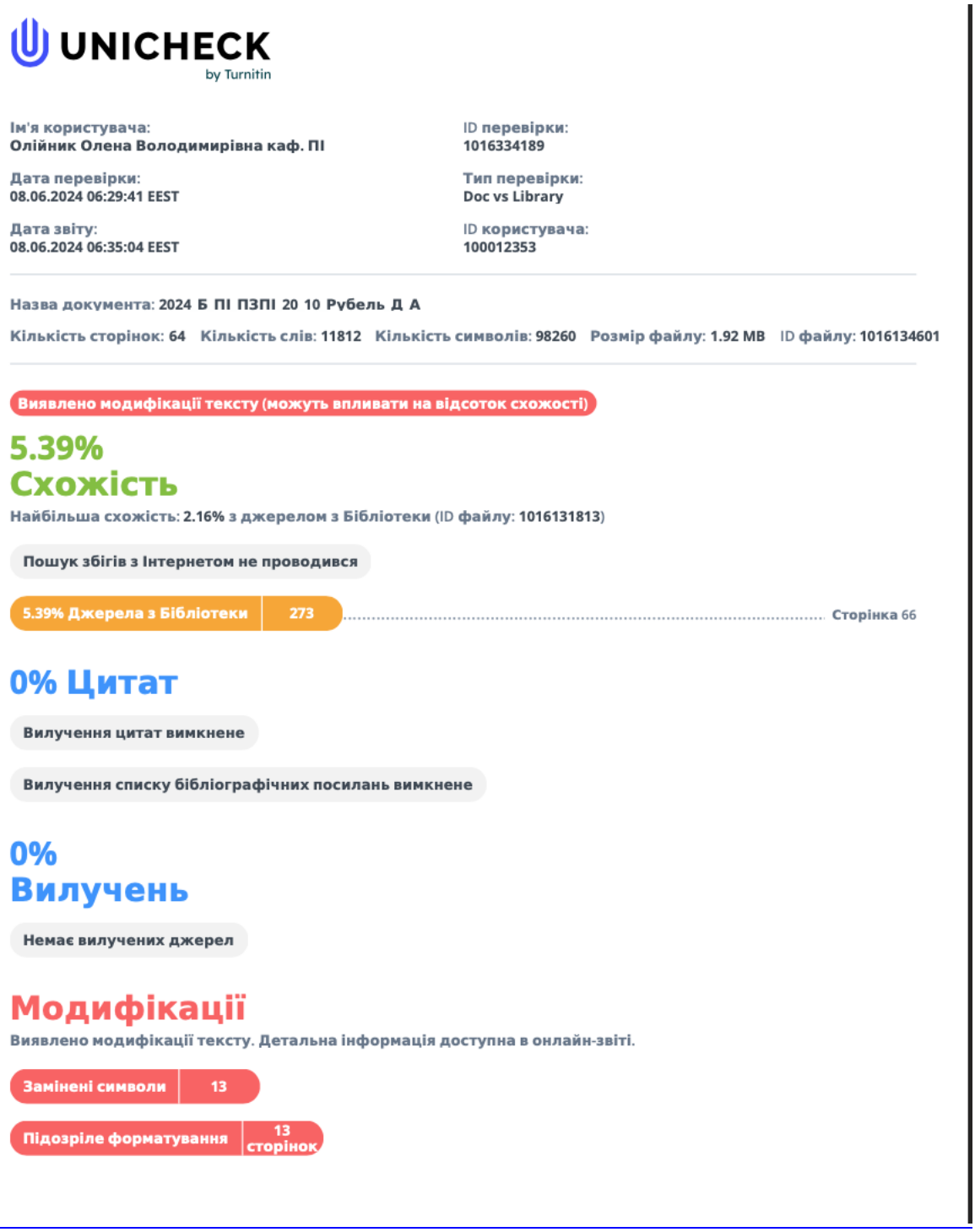
10. . RabbitMQ and WebSockets, Part 2: Web-MQTT, Web-STOMP and Web-AMQP - CloudAMQP. CloudAMQP. URL: https://www.cloudamqp.com/blog/rabbitmq-and-websockets-part-2-web-mqtt-web-stomp-and-web-amqp.html

12. Що таке Docker і навіщо він?. QAGroup. URL: https://qagroup.com.ua/publications/shcho-take-docker-i-navishcho-vin/

13. Що таке Kubernetes?. Kubernetes. URL: https://kubernetes.io/uk/docs/concepts/overview/what-is-kubernetes/

ДОДАТОК А

Звіт результатів перевірки на унікальність тесту в базі ХНУРЕ



UNICHECK
by Turnitin

Ім'я користувача:
Олійник Олена Володимирівна каф. ПІ

ID перевірки:
1016334189

Дата перевірки:
08.06.2024 06:29:41 EEST

Тип перевірки:
Doc vs Library

Дата звіту:
08.06.2024 06:35:04 EEST

ID користувача:
100012353

Назва документа: 2024 Б ПІ ПЗПІ 20 10 Рубель Д А

Кількість сторінок: 64 Кількість слів: 11812 Кількість символів: 98260 Розмір файлу: 1.92 MB ID файлу: 1016134601

Виявлено модифікації тексту (можуть впливати на відсоток схожості)

5.39%
Схожість

Найбільша схожість: 2.16% з джерелом з Бібліотеки (ID файлу: 1016131813)

Пошук збігів з Інтернетом не проводився

5.39% Джерела з Бібліотеки 273 ..... Сторінка 66

0% Цитат

Вилучення цитат вимкнене

Вилучення списку бібліографічних посилань вимкнене

0%
Вилучень

Немає вилучених джерел

Модифікації

Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті.

Замінені символи 13

Підозріле форматування 13 сторінок

ДОДАТОК Б

Слайди презентації

Харківський національний університет радіоелектроніки
Кафедра ПІ
Кваліфікаційна робота бакалавра

ПРОГРАМНА СИСТЕМА МОНІТОРИНГУ ЗАРЯДНИХ СТАНЦІЙ ЕЛЕКТРОМОБІЛІВ.

Розробник серверної частини

Виконав ст.гр. ПЗПІ-20-10 Рубель Д.А.
Керівник: доц. каф. ПІ Русакова Н.Є.

Постановка задачі

- Провести аналіз предметної області: розглянути дослідження існуючих технологій, стандартів, ринкових тенденцій та вимог користувачів.
- Визначити функціональні та нефункціональні вимоги до програмної системи .
- Деталізувати програмні рішення, прийняті в рамках проекту.
- Розглянути аспекти масштабування проекту, його здатність до розширення та обробку збільшеного навантаження без втрати продуктивності та надійності.
- Описати заходи безпеки, впроваджені для захисту даних в програмній системі.

2

Реалізація технологій в проекті

Ocelot

API шлюз, розроблений спеціально для платформ .NET.

Значення в проекті:
- маршрутизація запитів;
- агрегація запитів;
- безпека;
- простота налаштування роботи з Kubernetes

```
{
  "DownstreamPathTemplate": "/api/depot/{id}",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "depots.api",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/api/depot/{id}",
  "UpstreamHttpMethod": [ "GET" ]
},
{
  "DownstreamPathTemplate": "/api/depot",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "depots.api",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/api/depot",
  "UpstreamHttpMethod": [ "POST" ]
},
```


Реалізація технологій в проекті

Сервіси gRPC

Високоефективний, кросплатформенний фреймворк для віддалених викликів процедур

Значення в проекті:
- висока продуктивність;
- строга типізація;
- двосторонній стрімінг;
- зменшує накладні витрати на мережевий трафік;
- сумісність;
- безпека.

```
import "google/protobuf/timestamp.proto";
import "google/protobuf/wrappers.proto";

option csharp_namespace = "Transactions.Grpc.Protos";

package transactions;

service TransactionsGrpc {
  rpc GetById (GetTransactionByIdGrpcRequest) returns (TransactionGrpcResponse);
}

message GetTransactionByIdGrpcRequest {
  string id = 1;
}

message TransactionGrpcResponse {
  string id = 1;
  int32 transaction_id = 2;
  string start_tag_id = 3;
  google.protobuf.StringValue stop_tag_id = 4;
  google.protobuf.Timestamp start_time = 5;
  google.protobuf.Timestamp stop_time = 6;
  google.protobuf.Timestamp created_at = 7;
  google.protobuf.Timestamp updated_at = 8;
  google.protobuf.StringValue stop_reason = 9;
  string connector_id = 10;
  google.protobuf.StringValue reservation_id = 11;
}
```

Реалізація технологій в проекті

SignalR

Бібліотека для .NET, яка спрощує процес додавання реального часу веб-функціональності до ваших додатків.

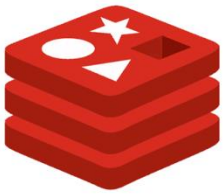
Значення в проекті:
- реальний час оновлень;
- простота впровадження;
- підтримка різних транспортних протоколів;
- масштабованість;
- двостороння комунікація;
- підтримка відключень і повторних з'єднань.

```
public class HubFacade
{
    private readonly IHubContext<ChargingStationHub> _hubContext;

    0 references
    public HubFacade(IHubContext<ChargingStationHub> hubContext)
    {
        _hubContext = hubContext;
    }

    1 reference
    public async Task SendCentralSystemMessageAsync(BaseMessage message, string messageType)
    {
        await _hubContext.Clients.All.SendAsync(messageType, message);
    }
}
```

5

Реалізація технологій в проекті

Redis



Високопродуктивна база даних, що зберігає дані в пам'яті

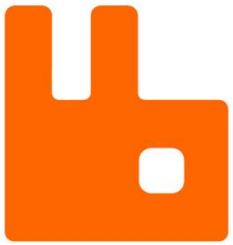
Значення в проекті:
- швидкість;
- сессійне кешування;
- масштабованість;
- висока доступність;
- низькі затримки;
- простота використання.

```
public class CacheManager : ICacheManager
{
    private readonly IDatabase _database;

    0 references
    public CacheManager(IConnectionMultiplexer connectionMultiplexer)
    {
        _database = connectionMultiplexer.GetDatabase();
    }

    4 references
    public async Task<T?> GetAsync<T>(string key)
    {
        var value = await _database.StringGetAsync(key);

        if (value.IsNullOrEmpty)
        {
            return default;
        }

        return value.HasValue ? JsonConvert.DeserializeObject<T>(value) : default;
    }

    4 references
    public Task RemoveAsync(string key)
    {
        return _database.KeyDeleteAsync(key);
    }

    5 references
    public async Task SetAsync<T>(string key, T value, TimeSpan? expiry = null)
    {
        await _database.StringSetAsync(key, value: JsonConvert.SerializeObject(value), expiry);
    }
}
```

6

Реалізація технологій в проекті

RabbitMQ



Система брокера повідомлень, яка реалізує протокол AMQP

Значення в проекті:

- асинхронна обробка повідомлень;
- надійність;
- підтримка різних протоколів;
- гнучка маршрутизація;
- проста інтеграція;
- моніторинг і керування;
- висока доступність.

```
public class StartTransactionMessageHandler : Ocpp16MessageHandler
{
    private readonly IPublishEndpoint _publishEndpoint;

    public StartTransactionMessageHandler(IConfiguration configuration, ILogger<StartTransactionMessageHandler> logger, IPublishEndpoint publishEndpoint)
        : base(configuration, logger)
    {
        _publishEndpoint = publishEndpoint;
    }

    public override string MessageType => Ocpp16ActionTypes.StartTransaction;

    public override async Task HandleAsync(OcppMessage inputMessage, Guid chargePointId, CancellationToken cancellationToken = default)
    {
        Logger.LogTrace(message: "Processing start transaction message...");
        var request = DeserializeMessage<StartTransactionRequest>(inputMessage);
        Logger.LogTrace(message: "StartTransaction => Message deserialized");
        var integrationMessage = new IntegrationOcppMessage<StartTransactionRequest>(chargePointId, request, inputMessage.UniqueId, ProtocolVersion);
        await _publishEndpoint.Publish(integrationMessage, cancellationToken);
    }
}
```

```
public class StartTransactionConsumer : IConsumer<IntegrationOcppMessage<StartTransactionRequest>>
{
    private readonly ILogger<StartTransactionConsumer> _logger;
    private readonly ITransactionService _transactionService;
    private readonly IPublishEndpoint _publishEndpoint;

    public StartTransactionConsumer(ILogger<StartTransactionConsumer> logger, ITransactionService transactionService, IPublishEndpoint publishEndpoint)
    {
        _logger = logger;
        _transactionService = transactionService;
        _publishEndpoint = publishEndpoint;
    }

    public async Task Consume(ConsumeContext<IntegrationOcppMessage<StartTransactionRequest>> context)
    {
        _logger.LogInformation(message: "Processing start transaction message...");

        var incomingRequest = context.Message.Payload;
        var chargePointId:Guid = context.Message.ChargePointId;
        var ocppProtocol:string = context.Message.OcppProtocol;

        var response = await _transactionService.ProcessStartTransactionAsync(incomingRequest, chargePointId, context.CancellationToken);

        var integrationMessage = CentralSystemResponseIntegrationOcppMessage.Create(chargePointId, response, context.Message.OcppMessageId, ocppProtocol);
        await _publishEndpoint.Publish(integrationMessage, context.CancellationToken);

        _logger.LogInformation(message: "Start transaction message processed");
    }
}
```

7

Реалізація технологій в проекті

Azure Table Storage

Сервіс збереження NoSQL даних, призначений для зберігання структурованих даних у вигляді таблиць.

Значення в проекті:

- масштабованість;
- висока швидкодія;
- простота використання;
- гнучка схема даних;
- висока доступність і надійність;
- ідеально підходить для зберігання неструктурованих і напівструктурованих даних;

```
public class AzureTableStorageManager<T> : ITableManager<T> where T : BaseTableStorageEntity
{
    private readonly TableServiceClient _tableServiceClient;

    public AzureTableStorageManager(TableServiceClient tableServiceClient)
    {
        _tableServiceClient = tableServiceClient;
    }

    public async Task AddEntityAsync(string tableName, T entity)
    {
        var tableClient = await GetTableClientAsync(tableName);
        await tableClient.AddEntityAsync(entity);
    }

    public async Task<T> GetEntityAsync(string tableName, string partitionKey, string rowKey)
    {
        var tableClient = await GetTableClientAsync(tableName);
        return await tableClient.GetEntityAsync<T>(partitionKey, rowKey);
    }

    public async Task<List<T>> GetEntitiesByPartitionKeyAsync(string tableName, string partitionKey)
    {
        var tableClient = await GetTableClientAsync(tableName);
        return await tableClient.QueryAsync<T>(filter: $"PartitionKey eq '{partitionKey}'").ToListAsync();
    }

    public async Task<List<T>> GetAllEntitiesAsync(string tableName)
    {
        var tableClient = await GetTableClientAsync(tableName);
        return await tableClient.QueryAsync<T>().ToListAsync();
    }
```

8

Реалізація технологій в проекті

Mailing

Це потужний постачальник послуг електронної пошти, розроблений для маркетингових і розробницьких команд.

Значення в проекті:
- сповіщення;
самостійне встановлення пароля;
- запрошення;
- персоналізація;
- автоматизація.

```
public class EnergyConsumptionWarningEmailMessage : IEmailMessage
{
    private readonly Guid _depotId;
    private readonly Guid _chargePointId;
    private readonly DateTime _warningTimestamp;
    private readonly double _energyConsumption;
    private readonly double _energyConsumptionLimit;

    3 references
    public EnergyConsumptionWarningEmailMessage(Guid depotId, Guid chargePointId, DateTime warningTimestamp, double energyConsumption, double energyConsumptionLimit)
    {
        _depotId = depotId;
        _chargePointId = chargePointId;
        _warningTimestamp = warningTimestamp;
        _energyConsumption = energyConsumption;
        _energyConsumptionLimit = energyConsumptionLimit;
    }

    2 references
    public string Subject => "Energy consumption warning";

    2 references
    public string GetTextPart()
    {
        return $"Your energy consumption is too high!\nDepot ID: {_depotId}\nCharge point ID: {_chargePointId}" +
               $"\nWarning timestamp: {_warningTimestamp}\nEnergy consumption: {_energyConsumption}\nEnergy consumption limit: {_energyConsumptionLimit}";
    }

    2 references
    public string GetHtmlPart()
    {
        return $"<h3>Your energy consumption is too high!</h3><p>Depot ID: {_depotId}</p><p>Charge point ID: {_chargePointId}" +
               $"</p><p>Warning timestamp: {_warningTimestamp}</p><p>Energy consumption: {_energyConsumption}</p><p>Energy consumption limit: {_energyConsumptionLimit}</p>";
    }
}
```

9

Реалізація технологій в проекті

Math.NET

Math.NET Numerics має на меті надати методи та алгоритми для чисельних обчислень у науці, техніці та щоденному використанні.

Значення в проекті:
- передбачення;
- лінійна алгебра
- статистика

```
1 reference
private async Task<DateTime> CalculateChargingEndTime(Guid transactionId, DateTime transactionStartTime, CancellationToken cancellationToken = default)
{
    var meterValues = new List<SoCDateTime>
    {
        new()
        {
            MeterValueTimestamp = transactionStartTime,
            SoCValue = 0
        }
    };

    var meterValuesFromDb :List<SoCDateTime> = await _connectorMeterValueRepository.GetSoCForTransactionAsync(transactionId, cancellationToken);
    meterValues.AddRange(meterValuesFromDb);

    var times :double[] = meterValues.Select(x :SoCDateTime => (x.MeterValueTimestamp - transactionStartTime).TotalSeconds).ToArray();
    var charges :double[] = meterValues.Select(x :SoCDateTime => x.SoCValue).ToArray();

    var (intercept :double, slope :double) = Fit.Line(x: times, y: charges);

    var remainingCharge :double = 100 - charges[^1];
    var timeToFullCharge :double = (remainingCharge - intercept) / slope;

    var endTime = transactionStartTime.AddSeconds(times[^1] + timeToFullCharge);
    return endTime;
}
```

10

Реалізація технологій в проекті

EF Core

Entity Framework Core

Відкритий кросплатформний фреймворк, що виконує зіставлення таблиць у реляційній базі даних з об'єктами.

Значення в проекті:

- міграції;
- запити LINQ;
- відстеження змін;
- гнучка конфігурація;
- асинхронні операції;
- конфігурація моделей;
- лезі завантаження.

```
public class ApplicationDbContext : IdentityDbContext<InfrastructureUser, InfrastructureRole, string, IdentityUserClaim<string>, InfrastructureUserRole,
    IdentityUserLogin<string>, IdentityRoleClaim<string>, IdentityUserToken<string>> {

    public required DbSet<Depot> Depots { get; set; }
    public required DbSet<ChargePoint> ChargePoints { get; set; }
    public required DbSet<OcppTag> OcppTags { get; set; }
    public required DbSet<OcppTransaction> Transactions { get; set; }
    public required DbSet<Connector> Connectors { get; set; }
    public required DbSet<Reservation> Reservations { get; set; }
    public required DbSet<TimeZone> TimeZones { get; set; }
    public required DbSet<ApplicationUser> Users { get; set; }

    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options) : base(options)
    {
        if (!Database.IsInMemory())
        {
            Database.Migrate();
        }
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.ConfigureWarnings(warnings => warnings.Ignore(CoreEventId.NavigationBaseIncludeIgnored, CoreEventId.NavigationBaseIncluded));
        base.OnConfiguring(optionsBuilder);
    }

    public override int SaveChanges()
    {
        PreSaveChanges();
        return base.SaveChanges();
    }
```


Тестування

Test	Duration	Traits	Error Message
Depots.UnitTests (6)	117 ms		
Depots.UnitTests (6)	117 ms		
DepotServiceUnitTests (6)	117 ms		
CreateAsync_ShouldReturnDepot...	90 ms		
DeleteAsync_ShouldCallRemove_...	6 ms		
GetAsync_ShouldReturnPagedColl...	6 ms		
GetByIdAsync_ShouldReturnDepo...	3 ms		
GetTimeZonesAsync_ShouldRetur...	4 ms		
UpdateAsync_ShouldReturnDepot...	8 ms		

xUnit

Тестування депо

- стандартизованість;
- легкість використання;
- автоматизація тестування;
- структурованість тестів.

```
[Fact]
0 references
public async Task GetAsync_ShouldReturnPagedCollection_WhenDepotsExist()
{
    // Arrange
    var request = new GetDepotsRequest { PagePredicate = new PagePredicate { Page = 1, PageSize = 10 } };
    var depots = new PagedCollection<Depot>(new List<Depot> { CreateDepot() }, elementsTotalCount: 1, pageSize: 10, currentPage: 1);
    var depotsResponse = new PagedCollection<DepotResponse>(new List<DepotResponse>
        { CreateDepotResponse() }, elementsTotalCount: 1, pageSize: 10, currentPage: 1);

    _depotRepositoryMock.Setup(expression: repo :IRepository<Depot> => repo.GetPagedCollectionAsync
            (specification: It.IsAny<GetDepotsSpecification>(), pageNumber: It.IsAny<int?>(), pageSize: It.IsAny<int?>(),
                applyTracking: false, cancellationToken: It.IsAny<CancellationToken>())) // ISetup<IRepository<_>,Task<_>>
        .ReturnsAsync(depots);

    _mapperMock.Setup(expression: mapper => mapper.Map<IPagedCollection<DepotResponse>>(depots)) // ISetup<IMapper,IPagedCollection<_>>
        .Returns(depotsResponse);

    // Act
    var result :IPagedCollection<DepotResponse> = await _depotService.GetAsync(request);

    // Assert
    Assert.NotNull(result);
    Assert.Equal(expected: depotsResponse, actual: result);
}
```

Масштабованість системи

Kubernetes забезпечує механізми:
- автоматичного масштабування
- балансування навантаження
- високу доступність

Мікросервісна архітектура дозволяє незалежно масштабувати окремі частини системи, що сприяє гнучкості та зручності у вдосконаленні окремих компонентів без впливу на інші.

Ця комбінація інструментів дозволяє ефективно реагувати на зміни в навантаженні та забезпечує високу продуктивність та доступність системи управління зарядженням електромобілів.

13

ВИСНОВКИ

- Розроблено сервіси для роботи серверної частини проекта.
- Успішно впроваджено ряд технологій у проект.
- Налагоджений зв'язок між клієнтом та сервером.
- Протестовано готовий проект схемою.

Результатом роботи є стабільна та функціональна серверна частина системи

- **Апробація результатів.**
  «Проєктування серверної частини системи керування та моніторингу зарядними станціями»
  «Інформаційні інтелектуальні системи» XXVIII Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті.

14

ДОДАТОК В

Програмний код

AuthService

```
public class AuthService : IAuthService
{
    private readonly UserManager<InfrastructureUser> _userManager;
    private readonly IRepository<ApplicationUser>
_applicationUserRepository;
    private readonly JwtHandler _jwtHandler;
    private readonly IMapper _mapper;
    private readonly IEmailService _emailService;
    private readonly IConfiguration _configuration;

    public AuthService(UserManager<InfrastructureUser> userManager,
IRepository<ApplicationUser> applicationUserRepository,
        JwtHandler jwtHandler, IMapper mapper, IEmailService
emailService, IConfiguration configuration)
    {
        _userManager = userManager;
        _jwtHandler = jwtHandler;
        _applicationUserRepository = applicationUserRepository;
        _mapper = mapper;
        _emailService = emailService;
        _configuration = configuration;
    }

    public async Task<TokenResponse> LoginAsync(LoginRequest
loginRequest)
    {
        var user = await
_userManager.FindByEmailAsync(loginRequest.Email);

        if (user != null && await _userManager.CheckPasswordAsync(user,
loginRequest.Password))
        {
            var userRoles = await _userManager.GetRolesAsync(user);
            var applicationUser = await
_applicationUserRepository.GetByIdAsync(user.ApplicationUserId);

            var token =
_jwtHandler.GenerateAuthToken(applicationUser!, userRoles.ToList(),
DateTime.UtcNow.AddHours(1));

            return new TokenResponse() { Token = token };
        }

        throw new UnauthorizedException("Invalid credentials");
    }

    public async Task RegisterAsync(RegisterRequest registerRequest)
    {
        ValidateRole(registerRequest.Role);
```

```
            var applicationUser = _mapper.Map<ApplicationUser>(registerRequest);
            await _applicationUserRepository.AddAsync(applicationUser);

            var user = new InfrastructureUser
            {
                UserName = registerRequest.FirstName + applicationUser.LastName,
                Email = registerRequest.Email,
                ApplicationUserId = applicationUser.Id,
                PhoneNumber = registerRequest.Phone
            };
            var result = await _userManager.CreateAsync(user);

            if (result.Succeeded)
            {
                await _userManager.AddToRoleAsync(user, registerRequest.Role);

                var roles = new List<string> { registerRequest.Role };
                await AssignAdditionalRolesAsync(user, registerRequest.Role, roles);

                var token = _jwtHandler.GenerateAuthToken(applicationUser, roles, DateTime.UtcNow.AddHours(1));

                var clientApplicationHost = _configuration["ClientApplicationHost"]!;
                var registrationLink = $"https://{clientApplicationHost}/auth/password-confirmation/?token={token}";
                var emailMessage = new RegistrationEmailMessage(registrationLink, registerRequest.FirstName);

                await _emailService.SendMessageAsync(emailMessage, registerRequest.Email);
            }
            else
            {
                throw new BadRequestException("Registration failed");
            }
        }

        public async Task ConfirmRegistration(ConfirmRegistrationRequest confirmRegistrationRequest)
        {
            var infrastructureUser = await _userManager.FindByEmailAsync(confirmRegistrationRequest.Email);

            if (infrastructureUser is null)
                throw new NotFoundException(nameof(InfrastructureUser), confirmRegistrationRequest.Email);

            await _userManager.AddPasswordAsync(infrastructureUser, confirmRegistrationRequest.Password);
        }

        private void ValidateRole(string role)
```

```
        {
            var validRoles = new[]
            {
                CustomRoles.SuperAdministrator,
                CustomRoles.Administrator,
                CustomRoles.Employee,
                CustomRoles.Driver
            };

            if (!validRoles.Contains(role))
            {
                throw new BadRequestException($"Invalid role: {role}");
            }
        }
        private async Task AssignAdditionalRolesAsync(InfrastructureUser
user, string role, List<string> roles)
        {
            if (role == CustomRoles.SuperAdministrator)
            {
                await                    _userManager.AddToRoleAsync(user,
CustomRoles.Administrator);
                roles.Add(CustomRoles.Administrator);
            }
            else if (role == CustomRoles.Administrator)
            {
                await                    _userManager.AddToRoleAsync(user,
CustomRoles.Employee);
                roles.Add(CustomRoles.Employee);
            }
        }
    }
```

OcppWebSocketConnectionHandler

```
    public         class         OcppWebSocketConnectionHandler         :
IOcppWebSocketConnectionHandler
    {
        private static readonly string[] SupportedProtocols = [
            OcppProtocolVersions.Ocpp20,
            OcppProtocolVersions.Ocpp16
            /*, "ocpp1.5" */
        ];

        private        const        string        MessageRegExp        =
"^\\[\\s*(\\d)\\s*,\\s*\"([^\"]*)\"\\s*,(?:\\s*\"(\\w*)\"\\s*,)?\\s*(.*)\\s
*\\]$";

        private readonly ILogger<OcppWebSocketConnectionHandler> _logger;
        private         readonly         IChargePointCommunicationService
_chargePointCommunicationService;
        private      static      readonly      ConcurrentDictionary<Guid,
ChargePointInfo> ActiveChargePointsDictionary = new ();
        public   static   List<ChargePointInfo>   ActiveChargePoints   =>
ActiveChargePointsDictionary.Values.ToList();

        private            readonly            IOcppMessageHandlerProvider
_ocppMessageHandlerProvider;
```

```
        private string _subProtocol;

        public
OcppWebSocketConnectionHandler(ILogger<OcppWebSocketConnectionHandler>
logger, IChargePointCommunicationService chargePointCommunicationService,
IOcppMessageHandlerProvider ocppMessageHandlerProvider)
        {
            _logger = logger;
            _chargePointCommunicationService                                =
chargePointCommunicationService;
            _ocppMessageHandlerProvider = ocppMessageHandlerProvider;
        }

        public async Task HandleConnectionAsync(RequestDelegate next,
HttpContext context)
        {
            var requestPath = context.Request.Path.Value;
            var requestParts = requestPath.Split('/');
            var chargePointId = Guid.Parse(requestParts[^1]);

            await
_chargePointCommunicationService.CheckChargePointPresenceAsync(chargePointI
d);
            var protocolValidationResult = CheckProtocol(context);

            if (protocolValidationResult != null)
            {
                _logger.LogWarning("Protocol validation failed for charge
point {ChargePointId}: {ProtocolValidationResult}", chargePointId,
protocolValidationResult);
                await                    CloseInvalidSocketAsync(context,
protocolValidationResult);
                context.Response.StatusCode                                =
StatusCodes.Status400BadRequest;
                return;
            }

            using          var          socket          =          await
context.WebSockets.AcceptWebSocketAsync();

            if (socket is null || socket.State != WebSocketState.Open)
            {
                await next(context);
                return;
            }

            if (!ActiveChargePointsDictionary.ContainsKey(chargePointId))
            {
                ActiveChargePointsDictionary.TryAdd(chargePointId,     new
ChargePointInfo(chargePointId, socket));
                _logger.LogInformation("No.    of    active    chargers:
{ChargersCount}", ActiveChargePointsDictionary.Count);
            }
            else
            {
                try
                {
```

```
                    var                    oldSocket                    =
ActiveChargePointsDictionary[chargePointId].WebSocket;
                    ActiveChargePointsDictionary[chargePointId].WebSocket
= socket;

                    if (oldSocket != null)
                    {
                        _logger.LogInformation("New   websocket   request
received for {ChargePointId}", chargePointId);

                        if  (oldSocket  !=  socket  &&  oldSocket.State  !=
WebSocketState.Closed)
                        {
                            _logger.LogInformation("Closing old websocket
for {ChargePointId}", chargePointId);

                            await
oldSocket.CloseAsync(WebSocketCloseStatus.NormalClosure,  "Client  sent  new
websocket request", CancellationToken.None);
                        }
                    }
                    _logger.LogInformation("Websocket            replaced
successfully for {ChargePointId}", chargePointId);
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, ex.Message);
                }

                if (socket.State == WebSocketState.Open)
                {
                    await HandleActiveConnection(socket, chargePointId);
                }
            }
        }
        public async Task SendResponseAsync(Guid chargePointId, OcppMessage
messageOut)
        {
            if    (ActiveChargePointsDictionary.TryGetValue(chargePointId,
out var chargePointInfo))
            {
                var socket = chargePointInfo.WebSocket;

                if (socket.State == WebSocketState.Open)
                {
                    string ocppTextResponse;

                    if (string.IsNullOrEmpty(messageOut.ErrorCode))
                    {

                        if            (messageOut.MessageType           ==
OcppMessageTypes.Call)
                        {
                            // OCPP-Request
                            ocppTextResponse                             =
string.Format("[{0},\"{1}\",\"{2}\",{3}]", messageOut.MessageType,
                                messageOut.UniqueId,    messageOut.Action,
messageOut.JsonPayload);
```

```
                        }
                        else
                        {
                            // OCPP-Response
                            ocppTextResponse                                    =
string.Format("[{0},\"{1}\",{2}]", messageOut.MessageType,
                                messageOut.UniqueId,
messageOut.JsonPayload);
                        }
                    }
                    else
                    {
                        // OCPP-Error
                        ocppTextResponse                                    =
string.Format("[{0},\"{1}\",\"{2}\",\"{3}\",{4}]",  messageOut.MessageType,
messageOut.UniqueId,   messageOut.ErrorCode,   messageOut.ErrorDescription,
"{}");
                    }


                    var                    responseBytes                    =
Encoding.UTF8.GetBytes(ocppTextResponse);
                    var          responseSegment          =          new
ArraySegment<byte>(responseBytes, 0, responseBytes.Length);
                    await                socket.SendAsync(responseSegment,
WebSocketMessageType.Text, true, CancellationToken.None);
                }
            }
        }

        public async Task SendResetAsync(Guid chargePointId, ResetRequest
incomingRequest)
        {
            if    (ActiveChargePointsDictionary.TryGetValue(chargePointId,
out var chargePointInfo))
            {
                var socket = chargePointInfo.WebSocket;

                if (socket.State == WebSocketState.Open)
                {
                    var            resetRequest            =            new
OcppMessage(OcppMessageTypes.Call,               Guid.NewGuid().ToString(),
Ocpp16ActionTypes.Reset, JsonConvert.SerializeObject(incomingRequest));
                    await SendResponseAsync(chargePointId, resetRequest);


chargePointInfo.RequestDictionary.TryAdd(resetRequest.UniqueId,
resetRequest);
                }
            }
        }

        public      async      Task      SendCentralSystemRequestAsync(Guid
chargePointId,    OcppMessage    centralSystemRequest,    CancellationToken
cancellationToken = default)
        {
            if    (ActiveChargePointsDictionary.TryGetValue(chargePointId,
out var chargePointInfo))
```

```
            {
                var socket = chargePointInfo.WebSocket;

                if (socket.State == WebSocketState.Open)
                {
                    await                  SendResponseAsync(chargePointId,
centralSystemRequest);


chargePointInfo.RequestDictionary.TryAdd(centralSystemRequest.UniqueId,
centralSystemRequest);
                }
            }
        }

        private async Task HandleActiveConnection(WebSocket socket, Guid
chargePointId)
        {
            if (socket.State == WebSocketState.Open)
                await HandlePayloadAsync(chargePointId, socket);

            if (socket.State != WebSocketState.Open
                && ActiveChargePointsDictionary.TryGetValue(chargePointId,
out var chargePointInfo)
                && chargePointInfo.WebSocket == socket)
                await RemoveConnectionsAsync(chargePointId, socket, null);
        }
        private async Task HandlePayloadAsync(Guid chargePointId, WebSocket
webSocket)
        {
            while (webSocket.State == WebSocketState.Open)
            {
                try
                {
                    var           payloadString           =           await
ReceiveDataFromChargerAsync(webSocket, chargePointId);

                    if (payloadString == null)
                        return;

                    await                ProcessPayloadAsync(payloadString,
chargePointId);
                }
                catch (Exception ex)
                {
                    _logger.LogError(ex, ex.Message);
                }
            }
        }

        private async Task ProcessPayloadAsync(string payloadString, Guid
chargePointId)
        {
            var match = Regex.Match(payloadString, MessageRegExp);

            if (match?.Groups is { Count: >= 3 })
            {
                var messageTypeId = match.Groups[1].Value;
```

```
                var uniqueId = match.Groups[2].Value;
                var action = match.Groups[3].Value;
                var jsonPayload = match.Groups[4].Value;
                _logger.LogInformation("OCPPMiddleware.Receive16 => OCPP-
Message:  Type={MessageTypeId}  /  ID={UniqueId}  /  Action={Action})",
messageTypeId, uniqueId, action);

                var  incomingMessage  =  new  OcppMessage(messageTypeId,
uniqueId, action, jsonPayload);

                //           var           validationResponse           =
ValidateSchema(JObject.Parse(jsonPayload), action);
                //
                // if (!validationResponse.Valid)
                // {
                //     throw new NotImplementedException();
                // }

                switch (incomingMessage.MessageType)
                {
                    case OcppMessageTypes.Call:
                        // Process Request
                        var                messageHandler                =
_ocppMessageHandlerProvider.GetRequestHandler(action, _subProtocol);
                        await  messageHandler.HandleAsync(incomingMessage,
chargePointId);
                        break;
                    case OcppMessageTypes.CallResult:

                        if
(ActiveChargePointsDictionary.TryGetValue(chargePointId,      out      var
chargePointInfo))
                        {
                            if
(chargePointInfo.RequestDictionary.Remove(uniqueId, out var request))
                            {
                                var           responseHandler           =
_ocppMessageHandlerProvider.GetResponseHandler(request.Action,
_subProtocol);
                                await
responseHandler.HandleAsync(incomingMessage, chargePointId);
                            }
                            else
                            {
                                _logger.LogWarning("No active request with
id  {RequestId}  found  for  charge  point  {ChargePointId}",  uniqueId,
chargePointId);
                            }
                        }
                        else
                        {
                            _logger.LogWarning("No  active  charge  point
found for {ChargePointId}", chargePointId);
                        }

                        break;
                    case OcppMessageTypes.CallError:
                        // Process Error
```

```
                    break;
                default:
                    throw new InvalidOperationException("Invalid
message type");
            }
        }
    }

    private JsonValidationResponse ValidateSchema(JObject payload,
string action)
    {
        var response = new JsonValidationResponse { Valid = false };

        try
        {
            //Getting Schema FilePath
            var currentDirectory = Directory.GetCurrentDirectory();
            var filePath = Path.Combine(currentDirectory, "Schemas",
$"{action}.json");
            //Parsing schema
            var content = JObject.Parse(File.ReadAllText(filePath));
            var schema = JSchema.Parse(content.ToString());
            var json = JToken.Parse(payload.ToString()); // Parsing
input payload
```

Абстрактний клас OcppMessageHandler

```
public abstract class OcppMessageHandler : IOcppMessageHandler
{
    protected OcppMessageHandler(IConfiguration configuration,
ILogger<OcppMessageHandler> logger)
    {
        _configuration = configuration;
        Logger = logger;
    }

    public abstract string ProtocolVersion { get; }
    public abstract string MessageType { get; }
    public virtual bool IsResponseHandler => false;

    private readonly IConfiguration _configuration;
    protected ILogger<OcppMessageHandler> Logger { get; }

    public abstract Task HandleAsync(OcppMessage inputMessage, Guid
chargePointId, CancellationToken cancellationToken = default);

    protected T DeserializeMessage<T>(OcppMessage msg)
    {
        var path = Assembly.GetExecutingAssembly().Location;
        var codeBase = Path.GetDirectoryName(path);

        var validateMessages =
_configuration.GetValue<bool>("ValidateMessages", false);

        string schemaJson = null;
```

```
            if (validateMessages &&
                !string.IsNullOrEmpty(codeBase) &&
                Directory.Exists(codeBase))
            {
                var msgTypeName = typeof(T).Name;
                var filename = Path.Combine(codeBase,
$"Schema{ProtocolVersion}", $"{msgTypeName}.json");
                if (File.Exists(filename))
                {
                    Logger.LogTrace("DeserializeMessage => Using schema
file: {0}", filename);
                    schemaJson = File.ReadAllText(filename);
                }
            }

            var reader = new JsonTextReader(new
StringReader(msg.JsonPayload));
            var serializer = new JsonSerializer();

            if (!string.IsNullOrEmpty(schemaJson))
            {
                var validatingReader = new
JSchemaValidatingReader(reader);
                validatingReader.Schema = JSchema.Parse(schemaJson);

                var messages = new List<string>();
                validatingReader.ValidationEventHandler += (o, a) =>
messages.Add(a.Message);
                var obj = serializer.Deserialize<T>(validatingReader);

                if (messages.Count > 0)
                {
                    foreach (var err in messages)
                    {
                        Logger.LogWarning("DeserializeMessage {0} =>
Validation error: {1}", msg.Action, err);
                    }

                    throw new FormatException("Message validation
failed");
                }

                return obj;
            }

            // Deserialization WITHOUT schema validation
            Logger.LogTrace("DeserializeMessage => Deserialization without
schema validation");
            return serializer.Deserialize<T>(reader);
        }
    }
```

ДОДАТОК Г

Тези XXVIII МІЖНАРОДНОГО МОЛОДІЖНОГО ФОРУМУ «РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ У XXI СТОЛІТТІ»

УДК 004.415 DOI: https://doi.org/10.30837/IYF.IIS.2024.306

ПРОЄКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ КЕРУВАННЯ ТА МОНІТОРІНГУ ЗАРЯДНИМИ СТАНЦІЯМИ

Чубаров Є. Е., Рубель Д. А.
Науковий керівник – к.т.н., доц. Русакова Н. Є.
Харківський національний університет радіоелектроніки, каф. ПІ,
м. Харків, Україна
e-mail: yevhen.chubarov@nure.ua, denys.rubel@nure.ua

This work is devoted to the design of the backend of the control and monitoring system for charging stations for electric vehicles. The use of the Open Charge Point Protocol (OCPP) protocol to ensure compatibility between different charging stations was considered. Also, application of microservice architecture to ensure modularity, scalability, and reliability with Docker and Kubernetes to ease deployment and increase system availability were considered.

У сучасному світі використання електромобілів стрімко набирає обертів, підсилюючи потребу в ефективних системах керування зарядними станціями. Збільшення кількості електромобілів вимагає не лише розширення інфраструктури зарядних станцій, але й вдосконалення програмного забезпечення, що стоїть за їх керуванням та моніторингом. Важливість цього питання полягає не тільки в забезпеченні доступності та ефективності зарядки, але й в інтеграції зарядних станцій у ширшу енергетичну систему, що включає відновлювані джерела енергії та розумні електромережі.

Розробка системи керування та моніторингу зарядними станціями є відповіддю на ці виклики. Мета даної роботи полягає в проєктуванні надійної, ефективної та легко масштабованої серверної частини системи керування зарядними станціями на основі протоколу OCPP (Open Charge Point Protocol).

Open Charge Point Protocol (OCPP) [1] є відкритим стандартом для забезпечення комунікації між зарядними станціями для електромобілів та центральними системами управління. Розроблений для підвищення взаємодії та сумісності між різними типами зарядних станцій та систем управління, OCPP сприяє створенню відкритого та гнучкого ринку для зарядних технологій.

Використання OCPP дозволяє системі легко інтегруватися з різними зарядними станціями, незалежно від їх виробника, та підтримувати широкий спектр функцій зарядки, включаючи, але не обмежуючись, аутентифікацією користувачів, управлінням сеансами зарядки, моніторингом статусу, віддаленим управлінням і т.д..

Завдяки OCPP можна забезпечити високий рівень адаптивності та майбутню сумісність системи зі змінами в технологіях та стандартах

Рисунок В.1 - Скріншот тез зі збірника (рисунок виконано самостійно)

зарядних станцій. Це важливо для забезпечення сталого розвитку та мінімізації потреби в постійних оновленнях інфраструктури, що в свою чергу знижує загальні витрати власників зарядних станцій та споживачів.

Враховуючи глобальний перехід до електромобілів та важливість відновлювальних джерел енергії, OCPP відіграє ключову роль у сприянні цієї трансформації, забезпечуючи інтеграцію рішень для зарядки у ширшу екосистему електротранспорту.

В ході розробки буде використовуватись мікросервісна архітектура [2], застосування якої надає такі переваги, як модульність (мікросервіси можуть розроблятися, оновлюватися, розгортатися та масштабуватися незалежно один від одного та з використанням різних технологій, що забезпечує велику гнучкість та швидкість внесення змін), масштабованість (в залежності від навантаження, окремі мікросервіси можна масштабувати горизонтально, додаючи більше екземплярів сервісів) та надійність (помилки в одному сервісі менш вірогідно вплинуть на роботу всієї системи, оскільки кожен сервіс є ізольованим).

Архітектуру програмної системи можна побачити на рисунку 1.

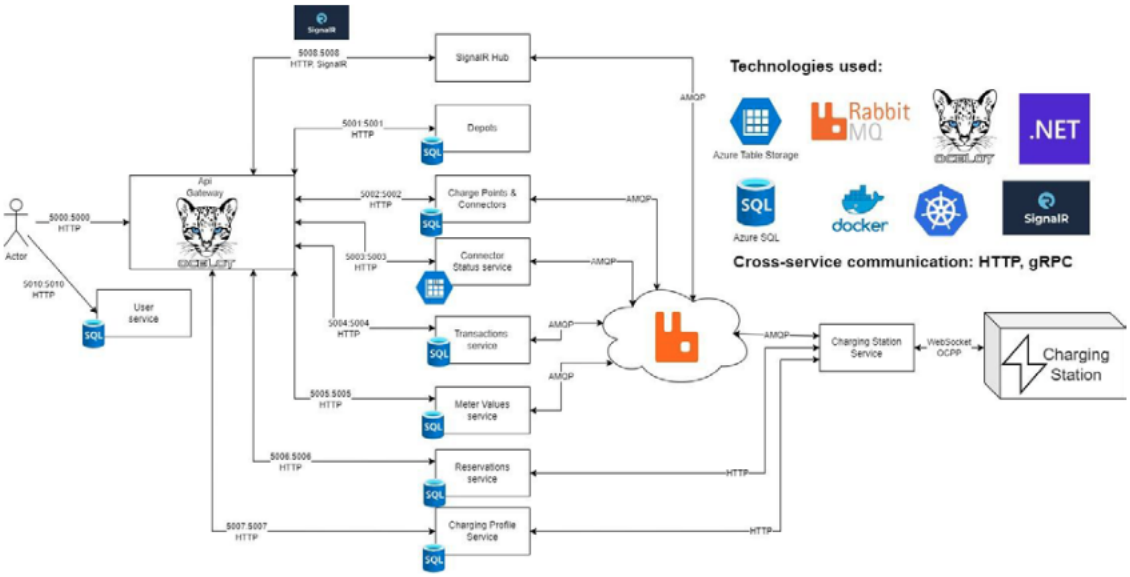


Рисунок 1 – Архітектура програмної системи

З рисунку можна побачити, що система складається з декількох сервісів, кожен з яких відповідає за окремий аспект управління зарядними станціями, такий як авторизація користувачів, управління транзакціями чи моніторинг стану станцій.

Розглядаючи взаємодію зарядних станцій з системою, слід зазначити, що комунікація відбуватиметься за допомогою веб-сокетів та протоколу OCPP через сервіс Charging Station, а обробка запитів та відповідей відбуватиметься асинхронно за допомогою брокера RabbitMQ сервісами, написаними з використанням платформи ASP.NET.

Рисунок В.2 - Другий скріншот тез зі збірника (рисунок виконано самостійно)

З боку клієнтського застосунку [3] взаємодія відбуватиметься за протоколами HTPP та WebSocket за допомогою SignalR для забезпечення взаємодії у реальному часі. Взаємодія відбуватиметься через API Gateway на основі Ocelot. Перед безпосередньою взаємодією відбувається авторизація через окремий User service.

В якості основного сховища даних буде використана база даних Azure SQL. Також додатково буде використане сховище Azure Table Storage для зберігання статусів зарядних станцій, які постійно оновлюються. Таке рішення було прийнято для того, щоб зекономити місце в основній БД та зменшити навантаження на неї.

Окремо варто звернути увагу на використання Docker для контейнеризації та Kubernetes для оркестрації, що дозволить легко розгортати та масштабувати систему. Docker забезпечує консистентність незалежно від середовища, в якому працює система (наприклад, dev-середовище або production), а Kubernetes автоматизує розподіл та масштабування контейнерів, забезпечуючи високу доступність та відмовостійкість.

Підбиваючи підсумки, можна зазначити, що проєктування серверної частини системи керування та моніторингу зарядними станціями є складним завданням, що вимагає глибокого розуміння як бізнес-вимог, так і сучасних технологічних рішень. Вибір мікросервісної архітектури та комбінація вищезгаданих технологій надає ряд стратегічних переваг, таких як гнучкість та швидкість розробки; надійність та масштабованість; оптимізація взаємодії завдяки брокеру повідомлень; безпека. Використання цих технологій дозволяє створити систему, яка може адаптуватись до швидких змін у сфері зарядних станцій, забезпечуючи сталість та довгостроковість інвестицій у розвиток інфраструктури.

Список використаних джерел:

1. Open charge point protocol – Open Charge Alliance. Open Charge Alliance. URL: https://openchargealliance.org/protocols/open-charge-point-protocol/ (дата звернення: 10.02.2024).

2. What are microservices?. microservices.io. URL: https://microservices.io/ (дата звернення: 10.02.2024).

3. Горкун Д. О, Налєскіна Т. С. Проектування angular додатку для програмної системи моніторингу зарядних станцій електромобілей використовуючи onion архітектуру. Розвиток суспільства та науки в умовах цифрової трансформації : V Міжнар. студент. наук. конф., м. Умань, 2 лют. 2024 р. / Наук. керівник Русакова Н. Є.

Рисунок В.3 - Третій скріншот тез зі збірника (рисунок виконано самостійно)

ДОДАТОК Е

Специфікація проекту

E-Charge Hub

Специфікація вимог до програмного забезпечення

3.0

01.06.2024

Налєскіна Тетяна

Горкун Дмитро

Чубаров Євген

Рубель Денис

Історія версій

Дата	Опис	Автор	Коментар
01.05.24	Версія 1.0	Налєскіна Тетяна	Створення документу
20.05.24	Версія 2.0	Горкун Дмитро	Оновлення використаних технологій
01.06.24	Версія 3.0	Чубаров Євген	Оновлення використаних технологій з боку бекенду

Зміст

1. Вступ

1.1 Огляд продукту

E-Charge Hub – програмна система моніторингу зарядних станцій електромобілів забезпечує контроль, управління та оптимізацію процесу зарядки електромобілів. Основні функції включають збір даних, аналіз ефективності станцій, управління процесом зарядки, відстеження споживання енергії та забезпечення безпеки даних. Система спрямована на ефективне управління зарядними станціями з урахуванням зростання попиту на електромобілі та необхідності розвитку відповідної інфраструктури. Монетизація проекту досягається шляхом придбання програмного забезпечення нашого продукту.

1.2 Мета

Метою програмної системи моніторингу зарядних станцій електромобілів є забезпечення зручності та ефективності процесу зарядки для користувачів електромобілів. Наша програмна система спрямована на те, щоб забезпечити адміністраторам зарядних станцій інноваційний та інтегрований підхід до управління, моніторингу та оптимізації їх інфраструктури. Ми розглянемо, як система надає централізоване управління, реальний час, аналітику, а також забезпечує високий рівень безпеки та енергоефективності. Розкриючи ці аспекти, ми створимо повністю осяжний образ та зрозуміння проекту, який сприятиме не тільки подальшому розвитку інфраструктури зарядних станцій, але й покращенню користувацького досвіду та сприянню сталому розвитку.

1.3 Межі

Програмна система E-Charge Hub має надавати можливості що дозволить зручно керувати депо та зарядними станціями в них, завчасно бронювати зарядні станції для подальшого використання та моніторингу енергоспоживання. Продукт має складатися з веб-додатку та мобільної версії, яка буде повністю відтворювати функціонал продукту. Система повинна мати можливість реєстрації та аутентифікації користувачів з різними ролями та надавати доступ до функцій згідно з ролями користувачів системи.

Серверна частина має бути реалізована з використанням мікросервісної архітектури, клієнтська з використанням Onion архітектури. Здійснення контролю над процесом розробки має проводитися за допомогою програмного забезпечення Jira.

1.4 Посилання

Текст документу містить наступні посилання:

1. Горкун Д.О., Налєскіна Т.С. Проектування Angular додатку для програмної системи моніторингу зарядних станцій електромобілів з використанням Onion архітектури. Тези V Міжнародна студентська наукова конференція «РОЗВИТОК СУСПІЛЬСТВА ТА НАУКИ В УМОВАХ ЦИФРОВОЇ ТРАНСФОРМАЦІЇ», м. Умань, 2024. С. 60-62. URL: https://doi.org/10.36074/liga-inter-02.02.2024
2. Чубаров Є.Е., Рубель Д.А. Проєктування серверної частини системи керування та моніторингу зарядними станціями. Тези XXVIII Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті» конференції «Інформаційні інтелектуальні системи», м. Харків, 2024. С. 306-308. URL: https://doi.org/10.30837/IYF.IIS.2024.306
3. Посилання на репозиторій: https://github.com/Nexus20/ChargingStation.Backend , https://github.com/Vspominay/ev-charging-station

1.5 Означення та абревіатури

В документі використовуються наступні означення та абревіатури:

OCPP – Open Charge Point Protocol

HTTP – HyperText Transfer Protocol

gRPC – Google Remote Procedure Calls

REST – REpresentational State Transfer

SQL – Structured Query Language

K8S – Kubernetes

ACR – Azure Container Registry

AKS – Azure Kubernetes Service

.2. Загальний опис

.2.1 Перспективи продукту

У зв'язку зі стрімким розвитком та впровадженням електромобільної технології та зростанням інтересу до сталих енергетичних рішень, створення програмної системи моніторингу зарядних станцій електромобілів стає необхідністю. За даними Міжнародного агентства з енергетики (IEA), кількість електромобілів на світі стрімко зростає, досягнувши позначки у 10 мільйонів одиниць у 2022 році, що свідчить про поступову трансформацію транспортної системи. Цей іноваційний підхід вимагає ретельного врахування численних передумов для успішного розгортання та оптимального функціонування системи.

Програмна система орієнтована на такі сегменти ринку:

1. Ринок послуг з моніторингу та аналізу: розробка системи, яка забезпечує детальний моніторинг ефективності зарядних станцій, створює можливість для підприємців надавати послуги з аналізу та оптимізації зарядних станцій для власників та операторів електротранспорту.

2. Системи управління та сервісні контракти: реалізація систем управління дозволяє підприємцям пропонувати сервісні контракти для підтримки та оптимізації роботи зарядних станцій, що включають в себе віддалене моніторинг, технічну підтримку та регулярне обслуговування.

3. Розвиток інфраструктури зарядних станцій: сприяння розвитку інфраструктури зарядних станцій для електромобілів за допомогою вдосконаленої системи моніторингу може стати ключовим фактором для бізнесу в області будівництва та управління інфраструктурою.

4. Платформи для користувачів електромобілів: розробка мобільних додатків та платформ для користувачів електромобілів, які надають інформацію про доступність та стан зарядних станцій, може створити нові можливості для реклами, партнерських програм та користувачських планів.

5. Аналітика для електромобільних виробників: надання аналітичних звітів та статистики виробникам електромобілів щодо використання їх продукції може стати основою для покращення технічних характеристик, а також розробки нових моделей, що задовольнять потреби ринку.

6. Енергетичні консалтингові послуги: врахування споживання електроенергії та оптимізація часу зарядки може створити можливості для компаній, які спеціалізуються на консалтингових послугах у галузі енергетики.

7. Екологічно орієнтовані ініціативи:розвиток системи моніторингу може бути використаний для підтримки екологічних ініціатив та отримання фінансової підтримки від урядових та неприбуткових організацій.

2.2 Функції продукту

Програмна система для моніторингу зарядних станцій електромобілів є комплексним інформаційним рішенням, яке об'єднує в собі різноманітні компоненти і функції для ефективного керування та контролю зарядними станціями. Основні складові програмної системи включають:

Моніторинг і збір даних. Система здатна отримувати дані з різних зарядних станцій, включаючи інформацію про стан заряду, використання енергії, доступність станцій та інші параметри.

Аналітика та звітність. Можливість аналізувати накопичені дані для отримання важливої інформації щодо ефективності зарядних станцій, споживання енергії, роботи системи тощо. Генерація звітів та статистики для подальшого аналізу та прийняття управлінських рішень.

14 Управління зарядками. Функціонал для керування процесом зарядки,

включаючи планування заряду, встановлення пріоритетів, управління через віддалений доступ тощо.

Система оповіщень та аварійного управління. Можливість автоматичного виявлення проблем та аварій, надсилання сповіщень та управління ситуаціями навіть у відсутності користувача.

Інтерфейс для користувачів. Зручний та інтуїтивно зрозумілий інтерфейс для користувачів, що дозволяє легко взаємодіяти з програмною системою, переглядати дані, встановлювати параметри та отримувати звіти.

Захист та безпека. Забезпечення захисту даних, використання шифрування, аутентифікації користувачів та інших заходів для забезпечення безпеки і конфіденційності інформації.

2.3 Характеристики користувачів

Функції, які планується реалізувати у першому релізі, було розділено на групи користувачів для полегшення їх представлення. Відомо, що система буде взаємодіяти з п'ятьма типами користувачів: власники, адміністратори від компаній, моніторингова система, водії та працівники.

Власник (Owner) може займатись менеджментом адміністраторів та депо, моніторингом статусу зарядних станцій депо, що включає в собі моніторинг конкретного роз'єму, авторизуватись у системі.

Адміністратор (Administrator) може займатись менеджментом користувачів, дивитись статистику, що отримується зі звітів, зробити резервацію, авторизуватись. Займатись менеджментом графіку зарядки, що включає налаштування інтервалів споживання електроенергії та налаштування обмежень потужності.

Моніторингова система (Monitoring system) відправляє пуш-повідомлення.

Водій (Driver) може моніторити конкретний роз’єм.

Працівник (Employee) може моніторити статус зарядних станцій депо, займатись менеджментом графіку зарядки, що включає налаштування інтервалів споживання електроенергії та налаштування обмежень потужності, робити резервації та авторизуватись у системі.

2.4 Загальні обмеження

Програмна система для моніторингу зарядних станцій електромобілів склається з серверної та клієнтської частини. Серверна частина буде реалізована за

допомогою мікросервісної архітектури. Концепція мікросервісної архітектури полягає у дизайні архітектури програмної системи таким чином, що функціональність розподіляється між сервісами. Розробка мікросервісів для системи буде проводитися на мові програмування C# з використанням платформи .NET 8 та фреймворку ASP.NET Core. Обрання цих конкретних технологій пояснюється їхньою здатністю підтримувати широкий спектр бібліотек, зокрема ті, які використовуються для роботи з хмарними технологіями, gRPC і RabbitMQ. Технологічний стек для розгортання програмної системи моніторингу зарядних станцій електромобілів використовує сучасні інструменти та підходи для ефективного управління та функціонування системи. У цьому контексті, використання Docker є важливим елементом для забезпечення ізольованості сервісів, а K8S є ключовим елементом оркестрації ресурсів. Для якнайшвидшої доставки оновлень клієнту, налаштовано CI/CD процеси за допомогою Azure DevOps Pipelines, що дозволяє автоматизувати розгортання рішення в хмару з використанням ACR та AKS відповідно.

Клієнтська частина розробляється за допомогою декількох технологій: Angular, Ionic, RxJs, Ng-bootstrap, FullCalendar, Ngx-translate, dayjs, Jest. Для управління базами даних у проекті обрано систему Microsoft SQL Server (MS SQL Server).

2.5 Припущення та залежності

Припущення 1. Продукт буде затребуваний на ринку;

Припущення 2. Вихід обладнання з експлуатації може призвести до відмови сервісу;

Припущення 3. Зворотна сумісність;

Залежність 1. Використання протоколу OCPP для стандартизації підходів комунікації між станціями та центральною системою;

Залежність 2. Серверна частина буде реалізована на платформі ASP.NET;

Залежність 3. Використання СУБД Azure SQL Server та Azure Table Storage зберігання даних;

Залежність 4. Використання Docker для контейнеризації та K8S для оркестрації ресурсів;

Залежність 5. Використання брокеру повідомлень RabbitMQ для асинхронної комунікації;

Залежність 6. Використання Redis Cache для кешування;

Залежність 7. Застосування Azure DevOps Pipelines для налаштування процесів CI/CD;

Залежність 8. Використання ACR та AKS для розгортання серверної частини в хмарі Azure;

Залежність 9. Для реалізації клієнтської частини використовується Angular, Ionic, RxJs, Ng-bootstrap, FullCalendar, Ngx-translate, dayjs, Jest.

Залежність 10. Застосування Render для розгортання клієнтської частини.

Залежність 11. Швидкість відгуку та оновлення залежать від потужності та стійкості сервера, який приймає та обробляє дані, а також від швидкості інтернет-з'єднання;

Залежність 12. Залежність часу розробки цього проекту від часу, який розробники витрачають вивчення та розробку паралельних завдань.

3. Конкретні вимоги

3.1 Вимоги до зовнішніх інтерфейсів

3.1.1 Інтерфейс користувача

Неавторизований користувач має потрапляти на сторінку входу в систему (рисунок 1).

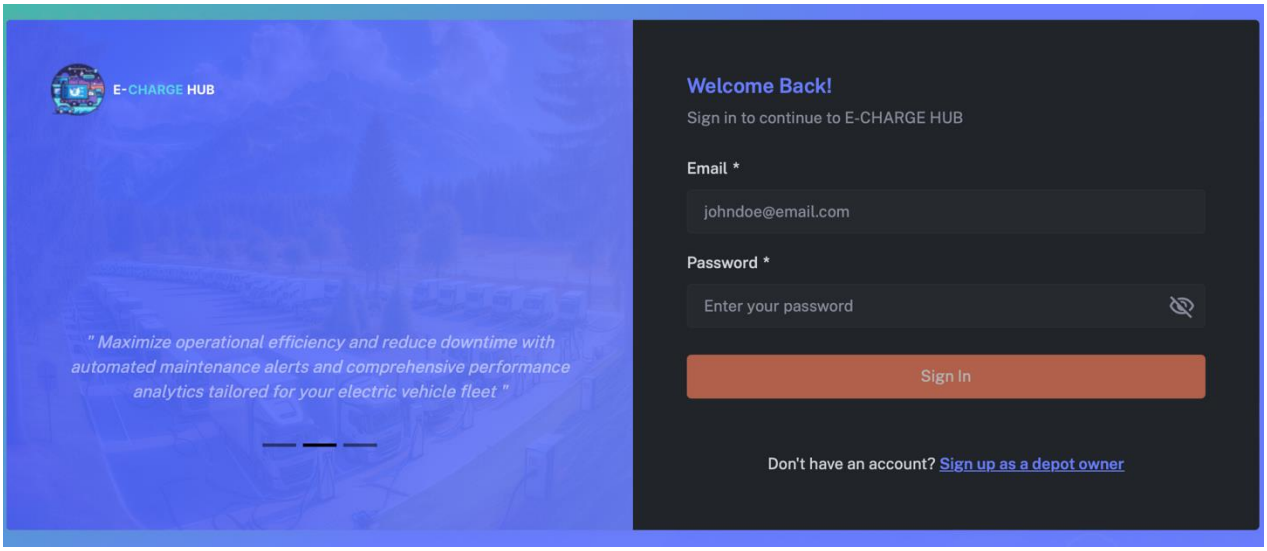


Рисунок 1 – Лендінг сторінка

На сторінці входу мусить бути логотип та слайдер з описом переваг системи. Також має бути форма входу в систему за поштою та паролем. Тут же мусить бути посилання на сторінку реєстрації.

На формі реєстрації (рисунок 2) повинні бути наступні елементи. Поле для введення електронної пошти та номеру телефону, де користувач вводить свою адресу електронної пошти для реєстрації в системі. Поле для введення пароля, яке дозволяє користувачеві створити пароль для захисту свого облікового запису. Також необхідно мати поле для введення імені, де користувач вводить своє ім'я, і поле для введення прізвища, де вводиться прізвище. Після заповнення всіх полів користувач натискає кнопку "Реєстрація", щоб завершити процес реєстрації. Для тих, хто вже зареєстрований, повинна бути кнопка "Увійти" для переходу на сторінку авторизації. Крім того, на формі повинно бути посилання на політику конфіденційності, яку користувач повинен прийняти під час реєстрації.

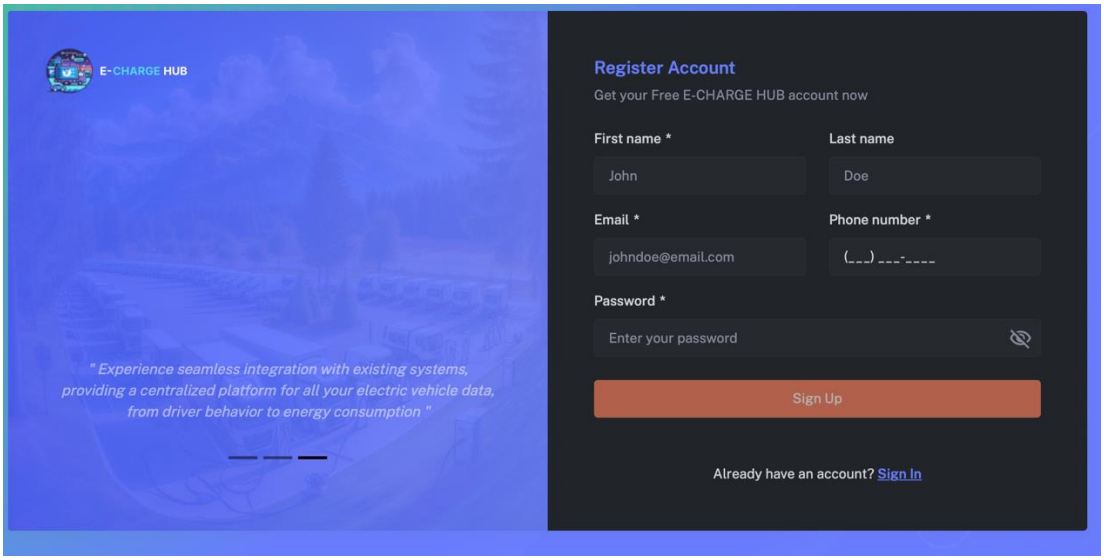


Рисунок 2 – Форма для залишення заявки

На сторінці списку користувачів системи моніторингу зарядних станцій депо (рисунок 3) необхідно відобразити заголовок сторінки, який ясно вказує, що це список користувачів системи. Далі має бути пошуковий рядок, що дозволяє швидко знайти конкретного користувача за ім'ям або іншими критеріями. Сам список користувачів повинен містити таблицю з такими даними: ім'я користувача, прізвище, електронна пошта, роль у системі (наприклад, адміністратор) та дату останнього входу в систему. Кожний рядок таблиці повинен мати кнопки для редагування або видалення користувача, а також можливість перегляду детальної інформації про користувача. Для зручності можна додати можливість сортування даних у таблиці за різними критеріями, такими як ім'я або дата останнього входу. Також бажано мати кнопку для додавання нового користувача, яка перенаправляє на форму реєстрації нового користувача.

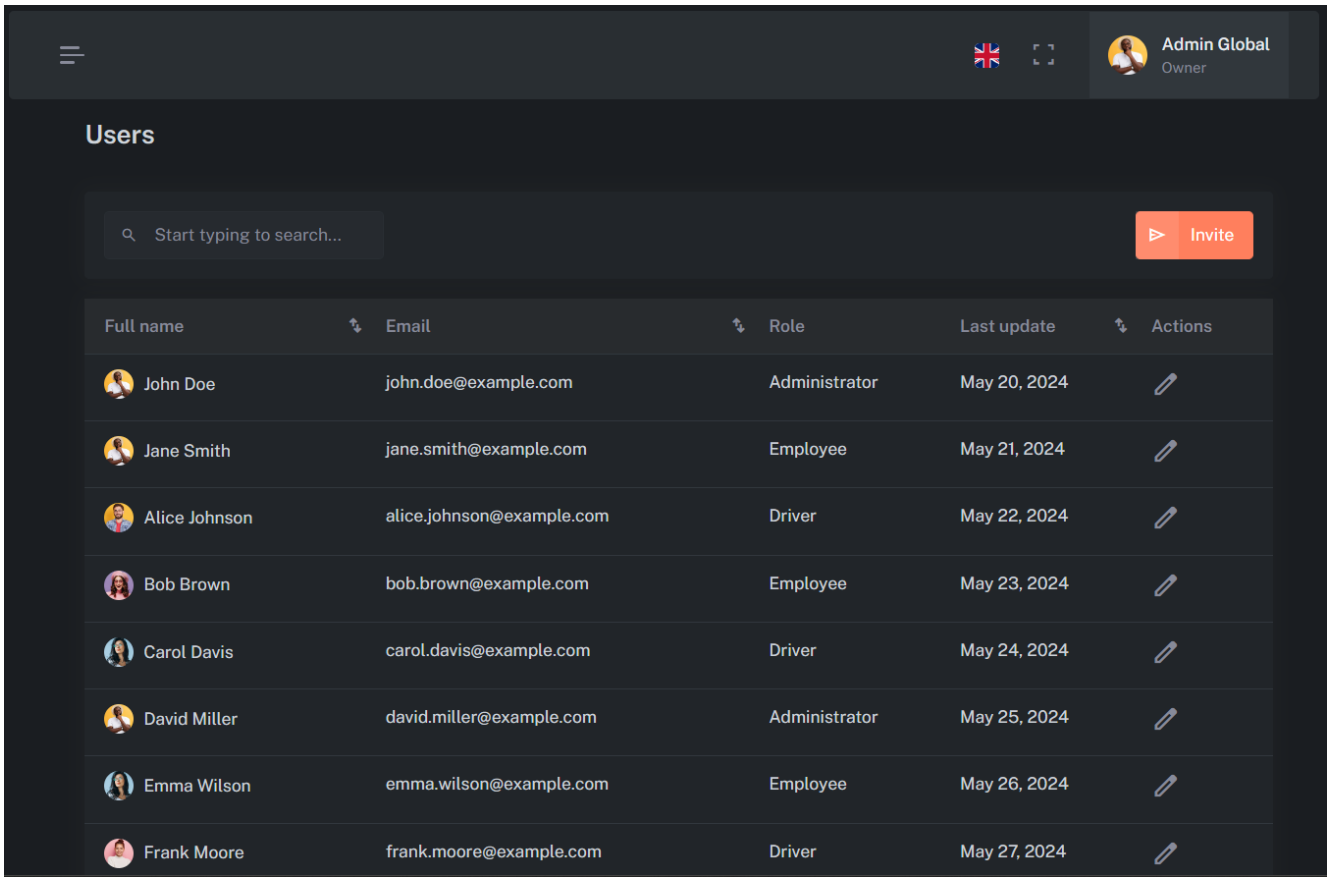


Рисунок 3 - Сторінка списку користувачів системи моніторингу зарядних станцій депо

На сторінці створення резервації зарядної станції (рисунок 4) повинні бути представлені наступні елементи. Спочатку має бути заголовок сторінки, який чітко вказує, що це форма для створення резервації зарядної станції. Повинно бути поле для вибору зарядної станції зі списку доступних станцій, де користувач може вибрати потрібну станцію для резервації та Ocpp тег. Додати поле для коментарів, де користувач може залишити додаткові примітки щодо резервації, початок зарядки та кінець. Кнопка підтвердження резервації повинна бути добре помітною, щоб користувач міг легко завершити процес. Після натискання на кнопку підтвердження повинно з'явитися повідомлення про успішне створення резервації або повідомлення про помилку у разі некоректного введення даних. На сторінці також має бути кнопка для скасування, яка дозволяє користувачеві повернутися до попередньої сторінки без збереження змін.

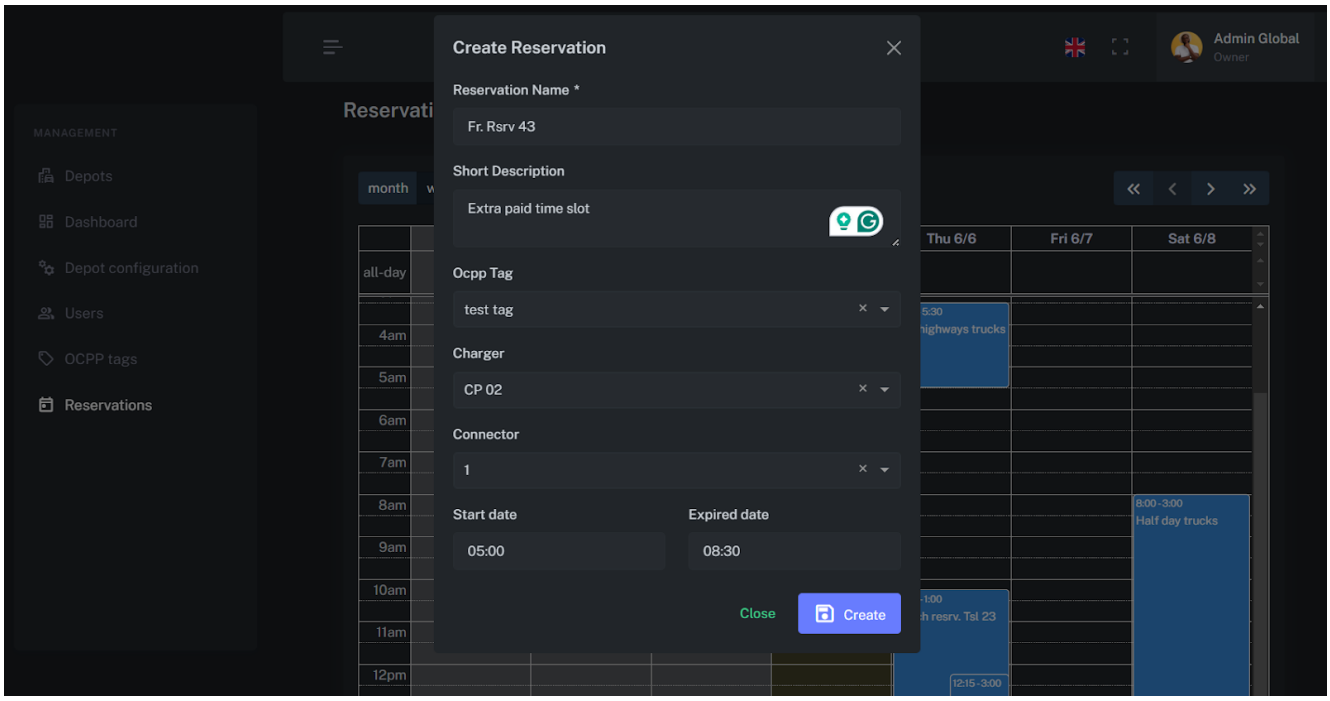


Рисунок 4 - Сторінка створення резервації зарядної станції

На сторінці календаря (рисунок 5) повинні бути представлені наступні елементи. Заголовок сторінки, який чітко вказує, що це календар, де відображаються заплановані резервації зарядних станцій. Сам календар має бути інтерактивним, дозволяючи користувачам переглядати дні, тижні та місяці. Користувачі повинні мати можливість легко переходити між різними періодами, використовуючи кнопки для переміщення вперед і назад. У кожній клітинці календаря має бути відображена інформація про наявні резервації, такі як час і станція, яка зарезервована. Для зручності можна додати можливість перегляду деталей резервації при наведенні курсору або натисканні на конкретну резервацію. Повідомлення про помилки або відсутність резервацій також мають бути чітко відображені.

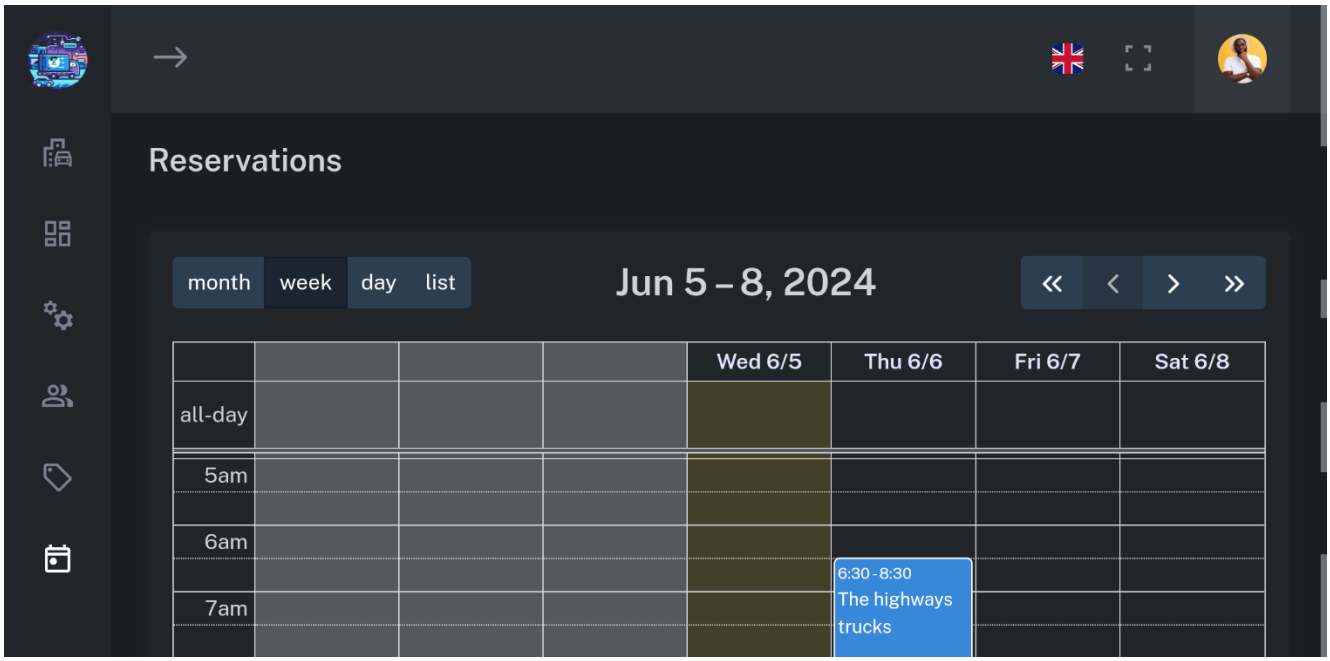


Рисунок 5 - Календар резервацій

.

3.1.2 Апаратний інтерфейс

Оскільки веб-сайт не має призначеного обладнання, прямих апаратних інтерфейсів немає.

3.1.3 Програмний інтерфейс

Веб-додаток повинен бути розроблений з урахуванням сумісності з усіма сучасними браузерами, такими як Google Chrome, Mozilla Firefox, Safari, Microsoft Edge тощо. Використання технологій HTML5, CSS3 та JavaScript ES6 допоможе забезпечити сумісність та підтримку різних браузерів. Для роботи з мобільною версією потрібно буде її завантажити через Play Market та App Store

3.1.4 Комунікаційний протокол

Зарядна станція взаємодіє через протокол OCPP та веб-сокети з відповідним сервісом обробки підключень. RabbitMQ підтримує протокол AMQP. Протокол OCPP є стандартним протоколом для обміну повідомленнями з зарядними станціями. Взаємодія між сервісами – за допомогою gRPC (окрім випадків, де комунікація відбувається асинхронно за допомогою RabbitMQ). В

3.1.5 Обмеження пам’яті

Веб-сайт не може безпосередньо контролювати кількість доступної пам'яті на комп'ютері користувача. Прямого обмеження пам'яті комп'ютера для користування веб-сайтом немає.

3.1.6 Операції

Для розробки веб-додатку використовується архітектурний стиль REST, що базується на принципах HTTP-протоколу. Основні операції, які можна виконувати з використанням REST, включають наступні:

- GET - для отримання ресурсу або його представлення;
- POST - для створення нового ресурсу;
- PUT - для оновлення існуючого ресурсу або створення нового, якщо він не існує;
- DELETE - для видалення ресурсу.

REST також використовує URL-шаблони для ідентифікації ресурсів та параметризації запитів.

3.1.7 Функції продукту

ГФ-1: Управління інформацією; створення депо. Надання власникам мереж депо можливості керувати інформацією; створювати свої депо.

ГФ-2: Створення співробітника. Надання адміністрації депо можливості створювати користувачів (співробітників/адміністраторів) для управління мережами зарядних станцій.

ГФ-3: Управління стану заряду станцій. Забезпечення інструментів для управління роботи станцій.

ГФ-4: Моніторинг стану депо в реальному часі. Надати можливість слідкувати за станом депо/зарядними станціями в реальному часі.

ГФ-5: Налаштування споживання електроенергії в рамках депо. Можливість глобального налаштування обмежень споживання енергії в заданих часових інтервалах депо.

ГФ-6: Статистика та звітність. Збір та надання статистичних даних для аналізу ефективності роботи зарядних станцій.

ГФ-7: Створення профілів зарядки. Конфігурація індивідуальних та групових профілів для контролю потужності зарядних станцій.

ГФ-8: Управління резервацією зарядних станцій. Організація системи резервації для ефективного використання станцій.

ГФ-9: Зміна мови додатку. Надані інструменту для зміни мови додатку

ГФ-10: Запрошення існуючого співробітника в депо. Реалізувати можливість надсилати запрошення наявним користувачам в системі приєднатися до депо.

3.2 Властивості програмного продукту

Користувачі розроблюваної програмної системи будуть широко розпорошені. Для доступу до системи необхідне стабільне інтернет-з’єднання. Оскільки дані будуть зберігатися віддалено на сервері, максимальний час відгуку встановлено до хвилини. Доступ до різноманітних функцій буде залежати від типу користувача. Цілісність та безпека зберігання даних буде забезпечуватися базою даних.

Програмна система має мобільну версію.

3.3 Атрибути програмного продукту

3.5.1 Надійність

3.5.1.1 Система повинна мати високий рівень надійності, користувач повинен отримувати тільки перевірені дані.

3.5.1.2 В разі виникнення помилки, система не повинна припиняти свою роботу, користувач повинен побачити зрозуміле йому повідомлення з описом

проблеми, що виникла (без коду помилки і інформації, яку зрозуміти зможе тільки розробник).

3.5.2 Доступність

3.5.2.1 Користуватися веб-додатком зможуть всі люди, що володіють українською або англійською мовою.

3.5.2.2 Для отримання доступу до мобільної версії потрібно її завантажити.

3.5.2.3 Тільки авторизовані користувачі можуть отримати доступ до функцій продукту.

3.5.2.4 Тільки адміністратору депо доступні функції видалення / додавання / зміни інформації про працівників.

3.5.3 Безпека

3.5.3.1 Кожен обліковий запис в системі буде захищено паролем, який встановлює користувач. Надійний пароль має складатися з більш ніж з 6 символів і обов'язково містити в собі принаймні одну цифру.

3.5.3.2 Увійти в обліковий запис можна тільки після введення правильних пароля та логіна.

3.5.3.3 Усі дані компанії мають зберігатися в окремій захищеній базі.

3.5.3.4 Вкладки мають бути доступні в залежності від дозволів ролі співробітника.

3.5.4 Супровід

3.5.4.1. Система повинна бути розроблена таким чином, щоб була можливість додавання нових функцій в майбутньому.

3.5.4.2. Система повинна бути розділена на частини (серверна, клієнтська, мобільно версії), щоб виправлення помилок займало якомога менше часу.

3.5.5 Переносимість

3.5.5.1 Веб-додаток повинен запускатися в усіх сучасних браузерах незалежно від пристрою, який використовує користувач.

3.4.5.2 Мобільна версія повинен працювати на Android та iOS пристроях.

3.5.6 Продуктивність

3.5.6.1 Час відгуку на перехід між вкладками веб-додатку повинен бути менше 1 секунди.

3.5.6.2 Після натискання кнопки "Додати співробітника" співробітник має бути одразу доданий у список працівників, йому має прийти лист для встановлення паролю на пошту протягом 5 хвилин.

3.4 Вимоги бази даних

В якості сховища даних було обрано СКБД Microsoft SQL Server, яка має наступні переваги:

масштабованість та продуктивність: MS SQL Server демонструє високу продуктивність та масштабованість. Він відмінно працює на великих корпоративних серверах, так і на десктопах або дрібних серверах, дозволяючи ефективно керувати великими обсягами даних;

інтеграція з іншими продуктами Microsoft: Оскільки MS SQL Server — продукт Microsoft, він бездоганно інтегрується з іншими продуктами Microsoft, такими як Windows Server, .NET, SharePoint, а також з Microsoft Office;

безпека даних: MS SQL Server має потужні механізми безпеки. Він пропонує різноманітні функції для контролю доступу, шифрування даних та аудиту, що дозволяє компаніям відповідати вимогам регуляторів.

3.5 Інші вимоги

Система не повинна дозволяти незареєстрованим користувачам отримувати доступ до функцій сервісу. Лендінг сторінка має бути окремою частиною, яка буде

загальнодоступною, а веб-сайт з функціональністю доступний лише при переході на авторизацію. Використання зображень, фото та інших матеріалів на сайті не повинно порушувати авторських прав. Інтерфейс веб додатку повинен змінюватися (підганяти розміри) під пристрої з різними розмірами екрану. Інтерфейс повинен бути інтуїтивно зрозумілий і приємний у використанні для людей будь-яких вікових категорій.

4. Додаткові матеріали

4.1 Діаграми баз даних

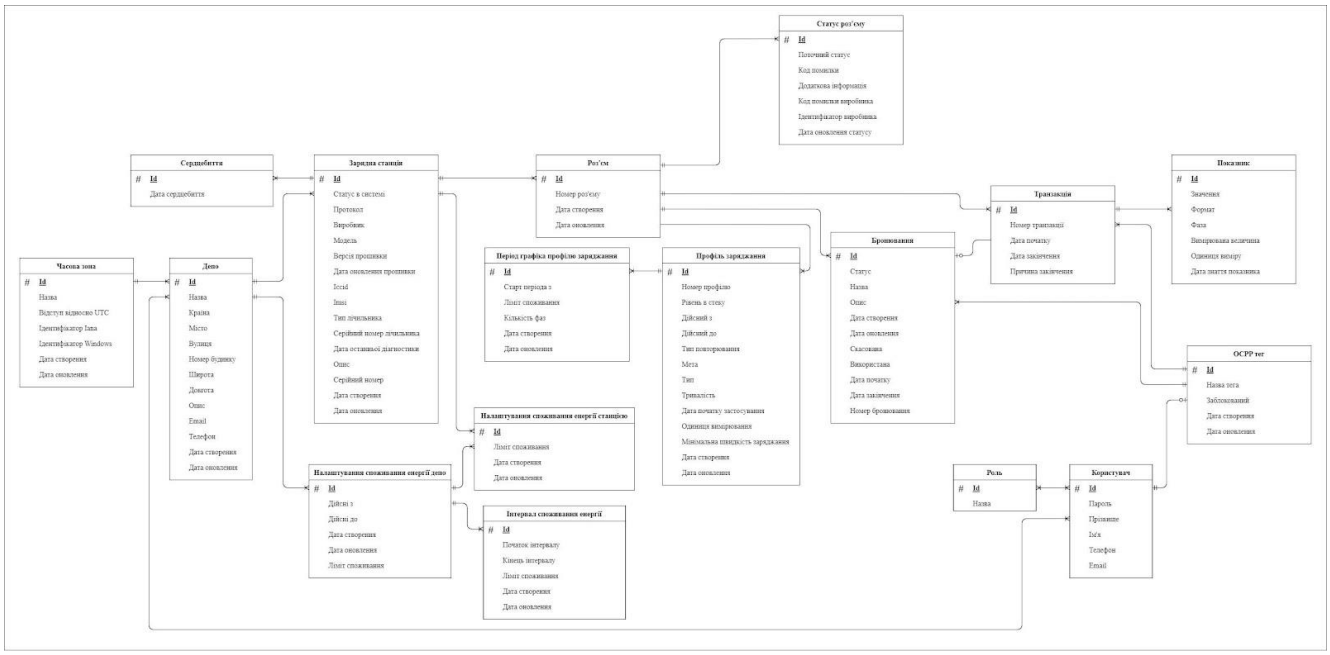


Рисунок 4 – ER-діаграма бази даних

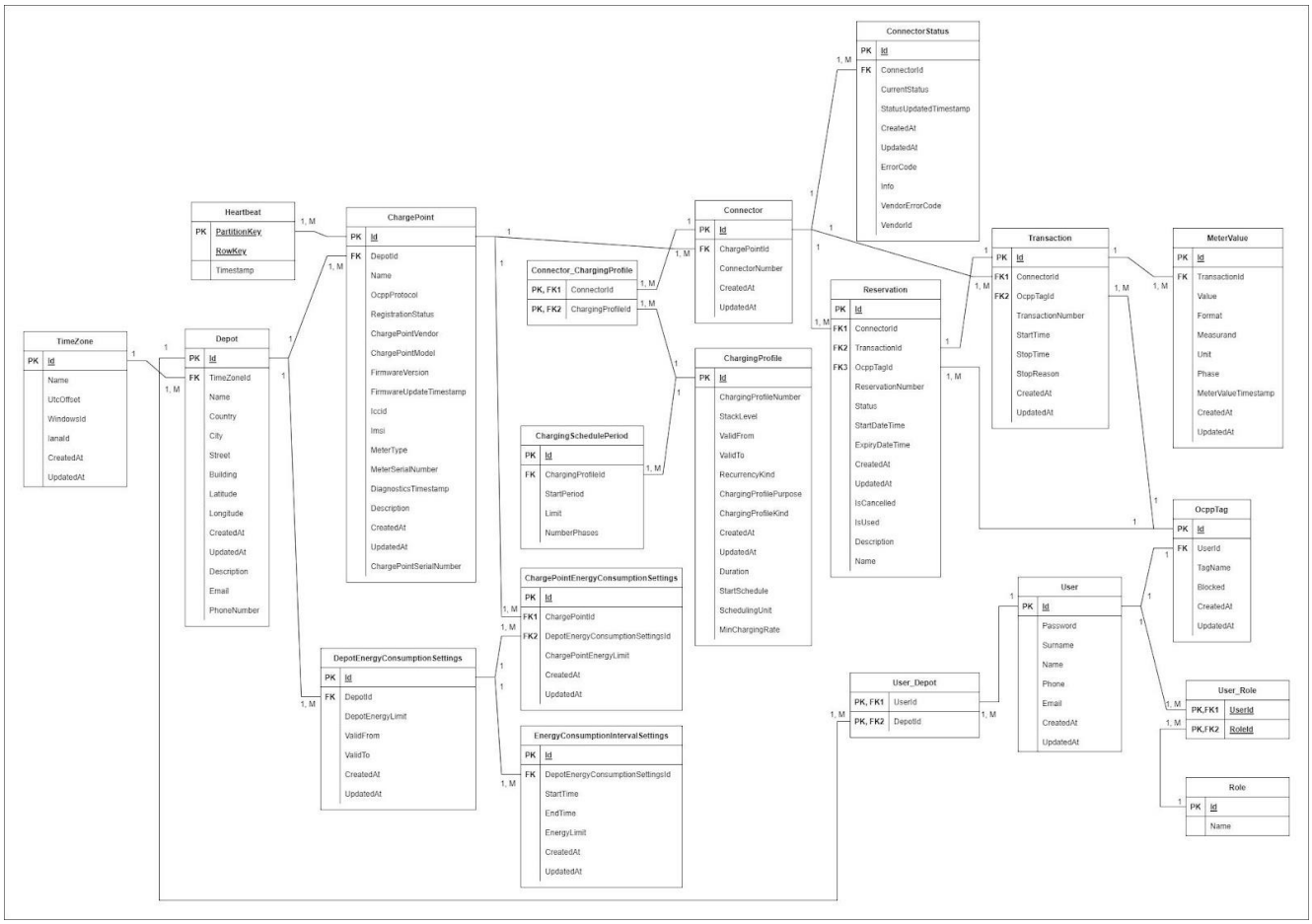


Рисунок 5 – Логічна модель бази даних

4.2 Діаграма прецедентів

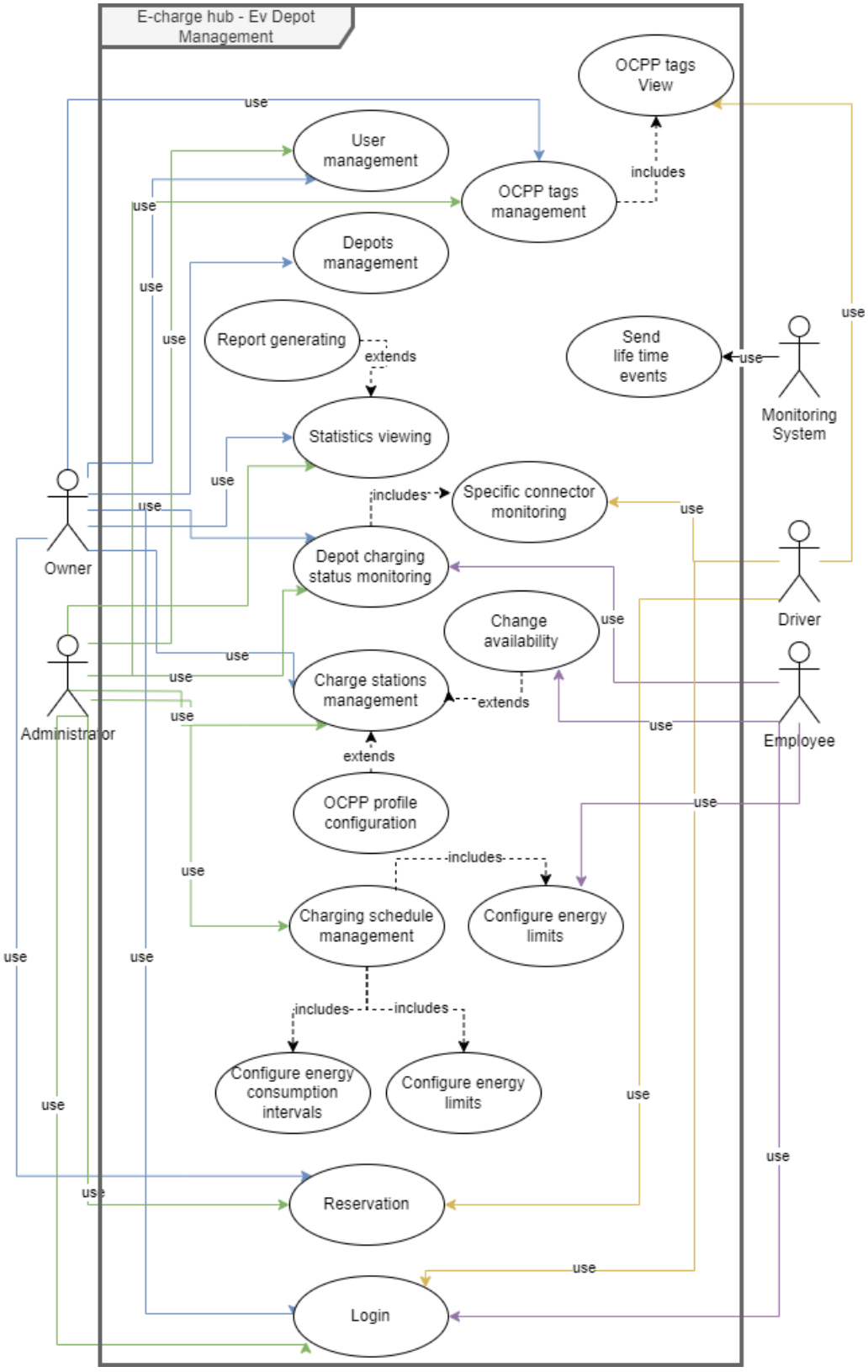


Рисунок 6 – Діаграма прецедентів

4.3 Діаграма діяльності

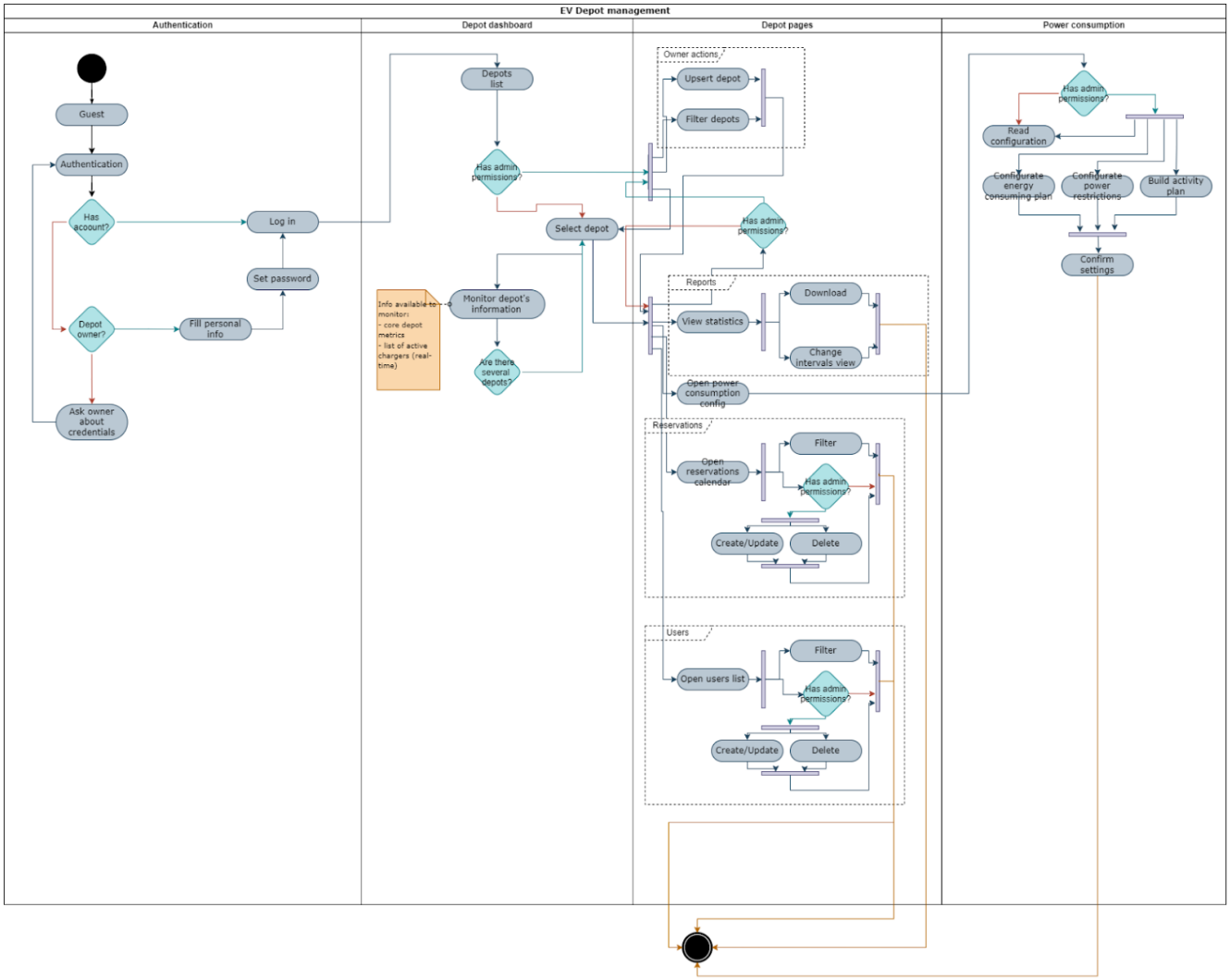


Рисунок 7 – Діаграма діяльності системи для управлінням споживання електроенергії депо електротранспорту