

Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Факультет _____ комп'ютерних наук (або центр післядипломної освіти, або навчально-науковий центр заочної форми навчання)
(повна назва)

Кафедра _____ програмної інженерії
(повна назва)

КВАЛІФІКАЦІЙНА РОБОТА
Пояснювальна записка

рівень вищої освіти _____ перший (бакалаврський)

Програмна система моніторингу зарядних станцій електромобілів.
Архітектор бази даних та проєкту, DevOps інженер
(тема)

Виконав:
студент 4 курсу, групи ПЗПІ-20-10

_____ Чубаров Є.Е.
(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. кафедри ПІ Русакова Н.Є.
(посада, прізвище, ініціали)

Допускається до захисту
Зав. кафедри

(підпис)

_____ З.В.Дудар
(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 _____ Чубарову Євгену Едуардовичу _____

(прізвище, ім'я, по батькові)

1. Тема роботи _____ Програмна система моніторингу зарядних станцій електромобілів. Архітектор бази даних та проєкту, DevOps інженер
Затверджена наказом по університету від 20.05. 2024р. № 471 Ст
2. Термін подання студентом роботи до екзаменаційної комісії 17.06.2024
3. Вихідні дані до роботи Розробити базу даних та архітектуру програмної системи моніторингу зарядних станцій електромобілів. Налаштувати CI/CD процеси для розгортання та поставки рішення.
4. Перелік питань, що потрібно опрацювати в роботі
Вступ, аналіз предметної галузі, формування вимог до програмної системи, архітектура та проєктування програмного забезпечення, опис прийнятих програмних рішень, впровадження програмного забезпечення, висновки, 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 _____
(підпис)

Чубаров Є.Е.

Керівник роботи _____
(підпис)

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

РЕФЕРАТ / ABSTRACT

Пояснювальна записка до кваліфікаційної роботи бакалавра, 127 стор., 21 рис., 2 табл., 12 додатків, 18 джерел.

ЕЛЕКТРОМОБІЛІ, ЗАРЯДНІ СТАНЦІЇ, МІКРОСЕРВІСНА АРХІТЕКТУРА, МОВА ПРОГРАМУВАННЯ C#, МОНІТОРИНГ, СПОЖИВАННЯ ЕЛЕКТРОЕНЕРГІЇ, ASP.NET, AZURE, DOCKER, ENTITY FRAMEWORK, GRPC, KUBERNETES, MS SQL, OCPP.

Об'єктом розробки є програмна система для централізованого керування та моніторингу зарядних станцій для електромобілів.

Метою роботи є проєктування та розробка бази даних, проєктування та розробка серверної частини програмної системи, що буде відмовостійкою та масштабованою, а також налагодження процесів CI/CD.

До методів проєктування та розробки БД можна віднести використання UML для побудови діаграм та SQL. Методи розробки серверної частини базуються на використанні мікросервісної архітектури із використанням платформи ASP.NET, брокера повідомлень RabbitMQ, API Gateway на базі Ocelot, протоколу gRPC для міжсервісної комунікації, а також протоколу OCPP на основі WebSocket для стандартизації взаємодії з зарядними станціями. Для налагодження процесів CI/CD використовуються Docker, Azure Kubernetes, Azure DevOps Pipelines, Azure Container Registry.

В результаті було розроблено серверну частину програмної системи, що надає можливості власникам та операторам депо зарядних станцій для гнучкого налаштування профілів споживання електроенергії, моніторингу стану станцій та процесів заряджання зарядок, а також аналізу статистичних даних.

Завдяки використанню хмарних сервісів та інструментів контейнеризації та оркестрації, система здатна обробляти велику кількість даних, що надходять від

зарядних стацій, а застосовані архітектурні рішення дозволяють далі підтримувати і покращувати систему.

ELECTRIC VEHICLES, CHARGING STATIONS, MICROSERVICE ARCHITECTURE, C# PROGRAMMING LANGUAGE, MONITORING, ENERGY CONSUMPTION, ASP.NET, AZURE, DOCKER, ENTITY FRAMEWORK, GRPC, KUBERNETES, MS SQL, OCPP.

The object of development is a software system for centralized control and monitoring of charging stations for electric vehicles.

The purpose of the work is the design and development of the database, the design and development of the server part of the software system, which will be fault-tolerant and scalable, and establishing of CI/CD processes.

Database design and development methods include the use of UML for diagramming and SQL. The server-side development methods are based on the use of the microservice architecture using the ASP.NET platform, the RabbitMQ message broker, Ocelot-based API Gateway, the gRPC protocol for inter-service communication, as well as the WebSocket-based OCPP protocol for standardizing interaction with charging stations. Docker, Azure Kubernetes, Azure DevOps Pipelines, Azure Container Registry are used to establish CI/CD processes.

As a result, the server part of the software system was developed, which provides opportunities for owners and operators of charging stations depots to flexibly configure energy consumption profiles, monitor the status of stations and charging processes, as well as analyze statistical data.

Thanks to the use of cloud services and containerization and orchestration tools, the system can process a large amount of data coming from the charging station, and the applied architectural solutions allow further support and improvement of the system.

Я, Чубаров Євген Едуардович, 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. Архітектор бази даних та проєкту, DevOps інженер», що буде представлена до екзаменаційної комісії для публічного захисту, виконана самостійно, в ній не містяться елементи плагіату і вона може бути опублікована в електронному архіві відкритого доступу EIAr 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

Перелік скорочень	9
Вступ.....	10
1 Аналіз предметної галузі	11
1.1 Аналіз предметної області	11
1.2 Виявлення та вирішення проблем	12
1.2.1 Аналіз конкурентів	15
1.3 Постановка задачі	18
2 Формування вимог до програмної системи.....	20
2.1 Функціональні вимоги.....	20
2.2 Нефункціональні вимоги.....	21
3 Архітектура і проєктування програмного забезпечення.....	22
3.1 Проєктування архітектури ПЗ	22
3.2 Проєктування БД.....	27
3.3 Розгортання рішення, DevOps та CI/CD	40
4 Опис прийнятих програмних рішень	42
4.1 Опис програмного рішення.....	42
4.1.1 Опис частини Services	42
4.1.2 Використання Aggregator	46
4.1.3 Використання Gateway	47
4.1.4 Опис частини Domain	48
4.1.5 Опис частини Infrastructure	49
4.1.6 Робота з Entity Framework.....	50
4.1.7 Робота з RabbitMQ та MassTransit.....	51
4.1.8 Використання gRPC.....	55
4.1.9 Використання Redis	57
4.2 Використання Kubernetes	59

4.2.1 Загальний підхід.....	61
4.2.1 Розгортання stateless мікросервісів	62
4.2.2 Розгортання RabbitMQ та використання StatefulSet.....	63
4.2.3 Розгортання Redis	66
4.3 Налаштування CI/CD процесів	69
4.3.1 Застосування Azure DevOps.....	70
5 Впровадження програмного забезпечення	76
Висновки	77
Перелік джерел посилання	78
Додаток А Архітектура серверної частини програмної системи	80
Додаток Б Діаграми бази даних	81
Додаток В Тези XXVIII Міжнародного молодіжного форуму «Радіoeлектроніка та молодь у XXI столітті» конференції «Інформаційні інтелектуальні системи»	83
Додаток Г Код Kubernetes маніфестів для розгортання stateless сервісів	86
Додаток Д Код Kubernetes маніфестів для розгортання RabbitMQ кластера	87
Додаток Е Код Kubernetes маніфестів для розгортання кластера Redis Sentinel....	89
Додаток Ж Код Kubernetes маніфестів для розгортання кластера Redis.....	91
Додаток И Код Azure DevOps CI/CD пайплайну	93
Додаток К Скріншоти інтерфейсу клієнтської частини.....	97
Додаток Л Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ	99
Додаток М Слайди презентації	100
Додаток Н Специфікація програмного забезпечення.....	106

ПЕРЕЛІК СКОРОЧЕНЬ

ARC – Azure Container Registry
AKS – Azure Kubernetes Service
UML – Unified Modeling Language
SQL – Structured Query Language
CI/CD – Continuous Integration/Continuous Delivery
OCPP – Open Charge Point Protocol
HTTP – HyperText Transfer Protocol
gRPC – Google Remote Procedure Calls
REST – REpresentational State Transfer
EF – Entity Framework
ORM – Object-Relational Mapping
FQDN – Fully Qualified Domain Name

ВСТУП

У сучасному світі електромобілі стають все більш популярними завдяки їхній екологічності та ефективності. Проте, для підтримки швидкого розвитку цього виду транспорту, необхідна належна інфраструктура зарядних станцій. Ефективні моніторинг та управління цією інфраструктурою є ключовими аспектами для забезпечення доступності та надійності сервісів зарядки. Програмна система, яка дозволяє ефективно контролювати, керувати та оптимізувати процеси зарядки, є ключовим елементом інфраструктури зарядних станцій.

У даній дипломній роботі висвітлюється процес створення програмної системи моніторингу зарядних станцій електромобілів, починаючи від аналізу вимог вимоги до такої системи та конкурентів і закінчуючи розробкою, тестуванням та створенням механізмів CI/CD для поставки рішення.

У зв'язку з актуальністю проблеми та необхідністю подальшого розвитку інфраструктури зарядних станцій для електромобілів, ця робота є важливою для розвитку та впровадження екологічно чистих видів транспорту.

Мета кваліфікаційної роботи полягає в проектуванні архітектури системи та бази даних, а також налаштуванні механізмів CI/CD. Основні завдання включають аналіз вимог до системи, вибір технологій, проектування бази даних, створення надійної та масштабованої архітектури програмної системи та розробка механізмів автоматизації поставки та розгортання рішення.

Результатом цієї роботи буде успішна розробка та впровадження програмної системи моніторингу зарядних станцій електромобілів, яка відповідає вимогам замовника та забезпечує ефективне керування процесом зарядки електромобілів.

1 АНАЛІЗ ПРЕДМЕТНОЇ ГАЛУЗІ

1.1 Аналіз предметної області

Зарядні станції для електромобілів – це спеціальні пристрої, призначені для зарядки акумуляторів електромобілів. Вони надають можливість користувачам електромобілів підключатися до електричної мережі для того, щоб зарядити свої транспортні засоби. Зарядні станції можуть бути встановлені в різних місцях, наприклад, на громадських майданчиках, парковках, заправних станціях тощо. Такі пристрої можуть мати різні типи роз'ємів, працювати з різними стандартами зарядки (CHAdeMO, CCS, Type 2 (Mennekes)) та підтримувати різні потужності. Зарядні станції є ключовою складовою інфраструктури для розвитку та популяризації електромобілів.

Загальнодоступні зарядні станції в усьому світі наблизилися до 1,8 млн. точок зарядки у 2021 році, з яких третину становили швидкі зарядні пристрої. У 2021 році було встановлено майже 500 тис. зарядних пристроїв, що більше, ніж загальна кількість громадських зарядних пристроїв, доступних у 2017 році. Кількість загальнодоступних зарядних пристроїв зросла на 37% у 2021 році, що нижче темпу зростання у 2020 році (45%). Середній річний темп зростання становив майже 50% між 2015 і 2019 роками. У 2021 році кількість громадських станцій швидкої зарядки зросла трохи більше, аніж у 2020 році (48% порівняно з 43%), а повільної зарядки – на 33% порівняно з 46% попереднього року.

Як і в попередні роки, світовим лідером за кількістю загальнодоступних зарядних станцій залишається Китай. На нього припадає близько 85% зарядних станцій швидкого типу у світі та 55% – повільного типу [1].

Станом на 2023 рік в Україні налічувалось близько 5000 зарядних станцій, які були розподілені по всій країні. Більшість зарядних станцій знаходилися в мегаполісах, таких як Київ, Львів, Харків, Одеса та Дніпро, але також станції можна знайти в менших містах та на автодорогах між ними. Однак слід зазначити, що швидкісних зарядних станцій потужністю від 80 кВт, які є більш популярними серед користувачів, станом на 1 вересня 2023 року, в Україні всього 100 штук [2].

Наразі в Україні немає великих депо електричних зарядних станцій, але, незважаючи на те, що інфраструктура зарядних станцій для електромобілів знаходиться в постійному розвитку, побудова депо для обслуговування та керування зарядними станціями у великих масштабах ще не стало загальнопоширеною практикою.

Не дивлячись на те, що на даний момент в Україні відсутні депо для зарядки електромобілів, з розвитком та зростанням популярності цього виду транспорту можлива поява таких структур у майбутньому, особливо у великих містах або на транспортних вузлах, де буде великий попит на електричний транспорт та відповідно – інфраструктуру для його обслуговування.

В якості прикладу можна навести розвинені країни Європи, Північної Америки та Азії, де спостерігається значний попит на великі електричні зарядні депо, що обумовлюється швидким зростанням попиту на електромобілі та стрімким розвитком інфраструктури для їх зарядки. Депо зарядних станцій дозволяють ефективно обслуговувати велику кількість електромобілів. Додатково такі депо можуть слугувати центрами обслуговування, місцями зберігання та ремонту зарядних станцій, а також виконувати функції моніторингу та керування зарядними станціями.

1.2 Виявлення та вирішення проблем

Аналізуючи предметну галузь, можна відзначити перелік потенційних проблем, що стосуються розробки та впровадженні програмної системи моніторингу зарядних станцій електромобілів в бізнес:

- розробка та впровадження програмної системи може вимагати значних фінансових витрат на дослідження, розробку, тестування та інтеграцію програмного забезпечення;
- в галузі електромобілів та інфраструктури зарядних станцій конкуренція може бути досить інтенсивною. Таким чином, треба надати

конкурентоспроможні переваги, що змусять потенційних клієнтів звернути увагу на програмну систему;

- бізнесу слід враховувати можливі зміни в законодавстві щодо електромобілів та інфраструктури для них і бути готовим до змін та адаптації згідно з новими вимогами;
- до технологічних ризиків можна віднести швидкий розвиток технологій, виникнення та оновлення нових стандартів, що може призвести до застаріння програмного забезпечення. Тобто продукт, що розробляється, мусить мати можливості до легкої адаптації до нових технологій з мінімальними змінами;
- окремо варто виділити ризики щодо безпеки та конфіденційності персональних даних користувачів. Важливо забезпечити достатній рівень захисту та дотримуватися вимог законодавства щодо захисту персональних даних (наприклад, вимог GDPR).

Також окремо варто відзначити потенційні технічні проблеми з програмною системою, з якими можна стикнутись при розробці ПЗ:

- сумісність з різними типами зарядних станцій від різних виробників;
- безпека та конфіденційність даних. Необхідно забезпечити високий рівень безпеки даних як з боку інфраструктури програмної системи, так і з боку програмної реалізації;
- система мусить бути готовою до можливих технічних збоїв або перерв у роботі зарядних станцій та адекватно це обробляти;
- масштабованість: з ростом кількості електромобілів збільшується навантаження на програмну систему. Система повинна бути здатна масштабуватися для обробки зростаючої кількості даних та запитів від користувачів;
- система мусить бути спроможною до взаємодії з іншими системами, такими як системи платежів, моніторингу трафіку електромережі тощо. Важливо забезпечити сумісність та надійну взаємодію з цими системами.

Для вирішення проблем, що стосуються умов введення бізнесу можна використовувати ряд системних рішень, такі як:

- аналіз та ретельне фінансове планування витрат на розробку, впровадження та підтримку програмної системи. Розробка бюджету, що передбачає можливі ризики та непередбачені витрати;
- аналіз ринку та конкурентів допоможе розробити стратегію, що базується на унікальних перевагах компанії. Це може включати розвиток нових функцій, покращення якості обслуговування або створення інноваційних пропозицій для клієнтів;
- постійне відслідковування оновлень змін законодавства щодо регулювання електромобілів та відповідної інфраструктури допоможе уникнути проблем та штрафів через невідповідність законодавству;
- інвестування в дослідження та розвиток: активне інвестування в дослідження та розвиток для постійного вдосконалення програмної системи та її адаптації до нових технологій та вимог ринку;
- захист даних та безпека інформації: розробка та впровадження ефективних стратегій та заходів захисту даних, включаючи шифрування, механізми аутентифікації та авторизації, а також регулярні аудити та оновлення систем безпеки.

Для вирішення технічних проблем можна звернути увагу на такі системні рішення, як:

- використання стандартних протоколів зв'язку, таких як OCPP для забезпечення комунікації зі станціями різних виробників;
- використання хмарної інфраструктури від надійних провайдерів, таких як Microsoft Azure, та впровадження сучасних методів шифрування та захисту даних може забезпечити високий рівень захищеності інформації та запобігти несанкціонованому доступу до неї;

- реалізація систем аварійного відновлення та резервного збереження даних може забезпечити надійність та стійкість роботи програмної системи навіть у випадку відмов чи перерв у роботі;
- використання розподіленої архітектури, механізмів асинхронної комунікації та хмарних технологій дозволить підвищити масштабованість системи;
- використання стандартних механізмів взаємодії з іншими системами дозволить забезпечити сумісність та ефективну взаємодію з системами платежів, моніторингу тощо.

1.2.1 Аналіз конкурентів

Окремо варто зупинитись на ризиках, що пов'язані з конкуренцією, з кількох причин.

По-перше, наявність конкурентів може призвести до зменшення частки продукту на ринку, тому що при наявності кращих альтернативних рішень, користувачі можуть надати перевагу саме їм, що призведе до втрати клієнтів.

По-друге, наявність конкурентів може спонукає до постійного вдосконалення та розвитку продукту, щоб відповідати вимогам ринку та зберігати свою конкурентоспроможність.

Таким чином, конкуренція є однією з головних проблем для продукту, оскільки вона створює серйозний тиск на всі аспекти бізнесу, включаючи ринкову позицію, цінову стратегію, рівень інновацій, унікальність продукту і т.д..

Для продукту, що розробляється, головними конкурентами є такі продукти, як Omega app, JetApp та Ev.energy. Перейдемо до детальнішого аналізу цих продуктів, для виявлення переваг та недоліків.

Щодо Omega app (рис. 2.1) [3] варто відзначити широкий вибір позицій для придбання ПЗ компанії, цілодобова доступність, а також наявність мобільного застосунку для Android та iOS.

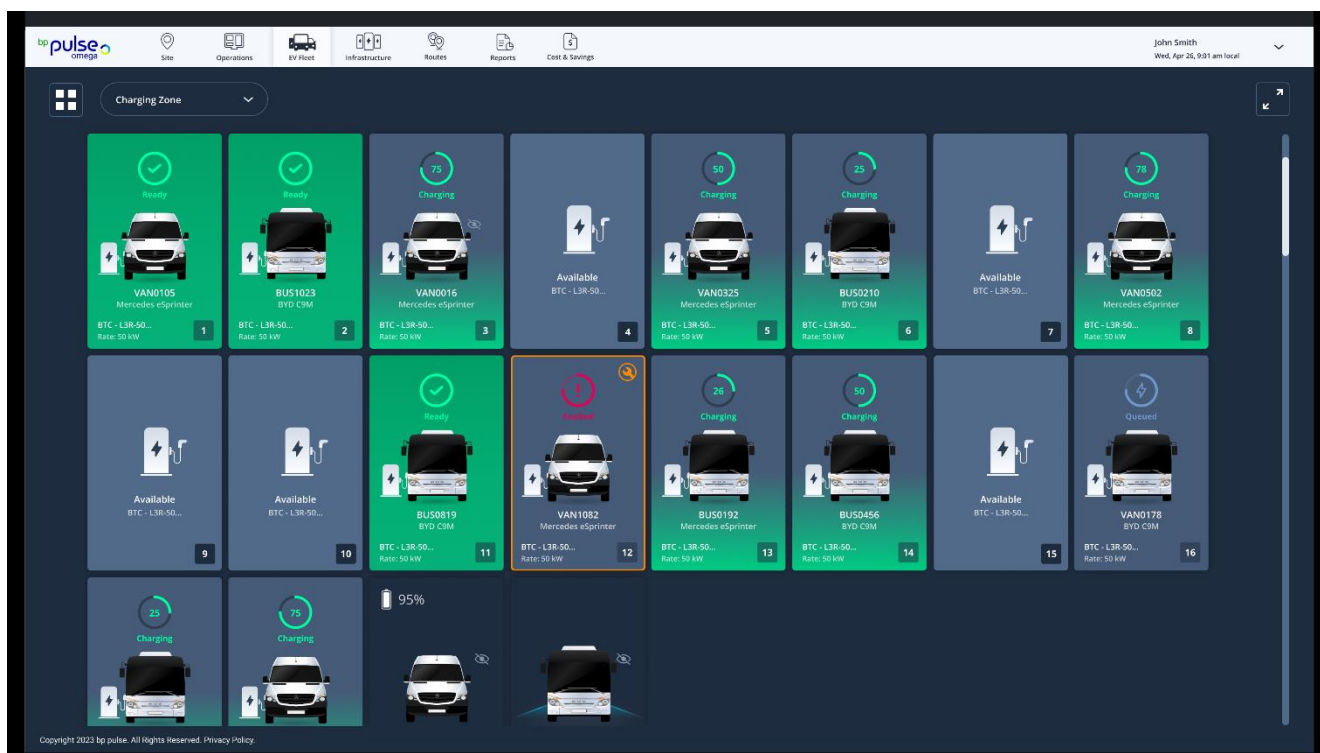


Рисунок 1.1 – Інтерфейс Omega App (за даними [3])

Говорячи про географію роботи системи, слід зазначити, що вона успішно працює на ринку США, пропонує програми лояльності для бізнесу та має зручний, сучасний інтерфейс. Якщо казати про недоліки, то перш за все слід відзначити велику кількість негативних відгуків від користувачів щодо автоматичного планування часу заряджання та авторизації. Негативні відгуки призвели до зниження репутації, що призвело до неоднозначного позиціонування на ринку Європи. Наявність проблем з оплатою через застосунок або ApplePay для клієнтів банків Великобританії та Німеччини заважають програмі працювати безперервно та ефективно.

Серед переваг системи JetApp (рис. 2.2) [4] можна виділити можливості віддаленого контролю для менеджерів та адміністраторів, наявність мобільного застосунку та зручний, сучасний інтерфейс, але ця система є доволі локальною, так як доступна лише для користувачів з Швейцарії.

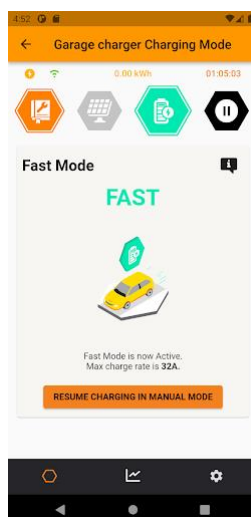


Рисунок 1.2 – Інтерфейс JetApp (за даними [4])

Аналізуючи Ev.energy (рис. 2.3) [5], можна відзначити перелік як переваг, так і недоліків. До переваг можна віднести можливість встановлення програмного продукту для домашнього використання, наявність мобільного застосунку та багатий функціонал продукту. До недоліків можна віднести проблеми з використанням мобільного застосунку користувачами Tesla та проблеми у користувачів, що користуються альтернативними джерелами енергії. Крім того, варто відзначити наявність проблем зі службою підтримки.

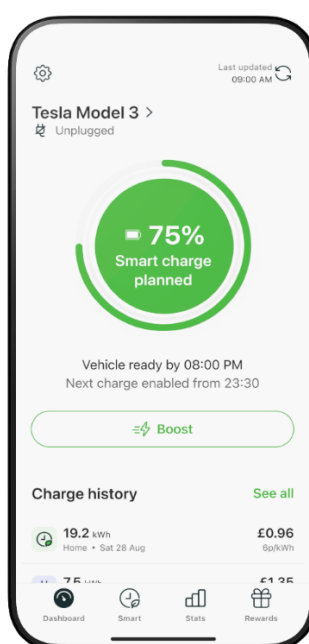


Рисунок 1.3 – Інтерфейс Ev.energy (за даними [5])

Отже, провівши аналіз програм Omega app, JetApp та Ev.energy, можна дійти висновку, що кожна з систем має свої переваги та недоліки. Таким чином, для того щоб продукт, що розробляється в ході цієї роботи, міг бути конкурентоспроможним, необхідно забезпечити такі переваги, як достатній перелік функцій, таких як робота з депо, налаштування споживання електроенергії та бронювання зарядних станцій; можливість використання продукту з різними моделями автомобілів та системами електроживлення; наявність зручного інтерфейсу, що відповідатиме кращим UX практикам; надійну та стабільну роботу системи та наявність якісної служби підтримки; можливості інтеграції продукту з іншими системами та сервісами, такими як програми моніторингу енергоспоживання та мобільні застосунки, робить його більш адаптивним та зручним у використанні.

Таким чином, у вирішенні проблеми конкуренції важливо зосередитися на створенні унікального продукту для клієнтів; розбудовувати сильний бренд та ефективну маркетингову стратегію для привертання уваги клієнтів та їх утримання; активно відслідковувати тенденції на ринку та реагувати на зміни задля збереження конкурентних переваг.

1.3 Постановка задачі

Метою проекту є створення системи, що дозволить зручно керувати депо та зарядними станціями в них, завчасно бронювати зарядні станції для подальшого використання та моніторингу енергоспоживання.

Першочерговим завданням є визначення чітких вимог до системи, які відповідають потребам та очікуванням користувачів, щоб краще контролювати роботу проектної команди та впевнитись в дотриманні термінів. Це включає визначення цілей, очікуваних результатів та вимоги до функціональності, надійності, відмовостійкості та масштабованості. Також важливо враховувати вимоги до користувацького інтерфейсу для того, щоб системою було зручно користуватись. Окремо слід звернути увагу на безпекові вимоги: оскільки система

зберігає особисті дані користувачів, вона мусить бути надійною та безпечною для користувачів, включаючи перевірку на помилки та можливість відновлення. Сформулювати відповідну продуктивність, особливо щодо масштабування та швидкості, також важливо.

Після того, як вимоги будуть сформовані, їх необхідно провалідувати на предмет того, чи відповідають вони потребам користувачів. Також слід звернути увагу на те, що вимоги можуть доповнюватися та змінюватись в ході розробки системи.

Окрему увагу слід приділити плануванню, так як якісне планування допоможе забезпечити вчасне виконання робіт згідно з графіком. В план необхідно додати всі етапи процесу розробки проєкту в коректній послідовності, з визначеними термінами та залежностями.

Після того, як було визначено вимоги до системи, можна розпочати її реалізацію, де спочатку треба визначитись з архітектурним підходом, який найкраще застосувати для розробки системи. Це може бути моноліт, Onion-архітектура або мікросервісна архітектура. В разі вибору мікросервісного підходу, слід одразу проаналізувати та обрати кращий варіант для міжсервісної комунікації.

Одним з надважливих етапів у проєктуванні системи є розробка структури баз даних. Правильно спроектована БД з використанням правильно підібраної технології забезпечить швидкість роботи системи та цілісність даних. Також під час проєктування БД слід звернути увагу на те, який архітектурний підхід буде використано при розробці бекенду системи.

Після того, як БД та серверна частина спроектовані, можна розпочинати реалізацію БД та кодування серверної частини, після чого необхідно підготувати код для налаштування CI/CD, а далі провести розгортання та тестування: мануальне та навантаженням

На цьому етапі задачі сформовані і можна переходити до формування вимог до програмної системи.

2 ФОРМУВАННЯ ВИМОГ ДО ПРОГРАМНОЇ СИСТЕМИ

Важливим кроком у створенні системи моніторингу зарядних станцій електромобілів є встановлення чітких вимог задля забезпечення ефективності та надійності у роботі системи. Нижче можна побачити перелік функціональних та нефункціональних вимог до системи, що розробляється.

2.1 Функціональні вимоги

З урахуванням даних, отриманих під час аналізу предметної галузі, можна встановити наступні вимоги, що стосуються властивостей, якими мусить володіти система, та функціоналу, який система мусить підтримувати:

- реєстрація користувачів: система повинна мати можливість реєстрації та аутентифікації користувачів з різними ролями та надавати доступ до функцій згідно з ролями користувачів системи;
- керування депо: створення, редагування, видалення;
- керування зарядними станціями та роз'ємами для зарядки в рамках депо: створення, редагування, видалення;
- налаштування параметрів споживання енергії як для депо, так і для станцій в рамках депо;
- система повинна реагувати на перевищення споживання встановлених лімітів енергії, якщо такі є, та сповіщати користувача через інтерфейс та електронну пошту;
- отримання статистики щодо споживання енергії;
- налаштування доступу користувачів до депо;
- система мусить підтримувати протокол ОСРР в якості стандартного протоколу для комунікації із зарядними станціями;
- бронювання роз'ємів зарядних станцій для зарядки;
- система повинна надавати змогу користувачу обрати мову: українську або англійську.

2.2 Нефункціональні вимоги

Виходячи з результатів аналізу предметної галузі та постановки завдання, можна визначити нефункціональні вимоги до системи:

- масштабованість: враховуючи зростання популярності електромобілів та розвиток інфраструктури зарядних станцій, система повинна бути здатна масштабуватись, щоб справлятися зі збільшенням обсягу даних та кількістю користувачів;
- відмовостійкість: система мусить бути готовою до можливих збоїв у інфраструктурі та адекватно реагувати на такі випадки.
- стійкість до навантажень: система має бути достатньо стійкою, щоб справлятися з високими навантаженнями без збоїв або втрати продуктивності. Це означає, що система має бути здатна обробляти великі обсяги запитів, забезпечуючи при цьому швидку та ефективну обробку;
- тестованість: система повинна бути легко тестованою, щоб забезпечувати високу якість та надійність її роботи. Всі компоненти системи повинні бути розроблені таким чином, щоб їх можна було легко перевіряти на наявність помилок чи збоїв, а також забезпечувати можливість проведення тестування для забезпечення високої якості роботи системи.

На даному етапі функціональні та нефункціональні вимоги визначені і можна розпочинати проектування архітектури системи.

3 АРХІТЕКТУРА І ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

3.1 Проєктування архітектури ПЗ

Враховуючи висунуті вимоги до програмної системи, а також дані, отримані в ході аналізу предметної області, застосувати мікросервісний [7] підхід. Такий підхід має наступні переваги:

- гнучкість розгортання: мікросервіси можна розгортати незалежно один від одного, що дозволяє оновлювати, тестувати та впроваджувати нові функції в окремих сервісах без впливу на роботу всієї системи.
- масштабованість: кожен мікросервіс може масштабуватися незалежно, забезпечуючи більшу ефективність у використанні ресурсів, особливо в пікові періоди використання.
- відмовостійкість: у разі виникнення помилки в одному мікросервісі, вона не розповсюджується на інші сервіси, що допомагає забезпечити стабільність всієї системи.
- краще розподіл відповідальності: мікросервісна архітектура дозволяє чітко розділити відповідальність між командами, які працюють над різними частинами системи, сприяючи більш ефективній співпраці та спеціалізації.
- легкість обслуговування та оновлення: Оскільки кожен мікросервіс є відносно невеликим та самостійним, його легше розуміти, тестувати та оновлювати. Це може сприяти швидшому виявленню та вирішенню проблем.

На рисунку А.1 додатку А можна побачити схему архітектури системи. Розробка мікросервісів буде вестись на мові C# з використанням платформи .NET 8 та фреймворку ASP.NET Core. Вибір саме цих технологій обумовлений підтримкою широкого спектру бібліотек, включно з бібліотеками для роботи з хмарними технологіями, gRPC та RabbitMQ. Зі схеми можна побачити, що зарядна станція взаємодіє через протокол OCPP та веб-сокети з відповідним сервісом обробки підключень (chargepoint websocket service), що відповідає за обмін повідомленнями між станціями та іншими сервісами. Взаємодія між сервісом

обробки підключень станцій та іншою частиною системи відбувається асинхронно за допомогою брокера повідомлень RabbitMQ [8]. Застосування такого підходу надає наступні переваги:

- відокремлення продюсерів (виробників повідомлень) та споживачів повідомлень: RabbitMQ дозволяє продюсерам відправляти повідомлення без необхідності знати про конкретних споживачів, тим самим забезпечуючи слабке зв'язування між сервісами;
- надійність: повідомлення зберігаються у чергах на RabbitMQ сервері до тих пір, поки споживач не підтвердить їх обробку. Це гарантує, що жодне повідомлення не буде втрачено в разі відмови споживача або виробника;
- масштабованість: споживачі можуть гнучко масштабуватися відповідно до навантаження;
- гнучкість маршрутизації: RabbitMQ пропонує різні моделі обміну повідомленнями, які дозволяють реалізувати складні сценарії маршрутизації повідомлень. В даному випадку використовуються черги (queues);
- відновлення після збоїв: RabbitMQ може автоматично відновлювати з'єднання і стан черг після відмови, що знижує вимоги до відновлення системи вручну;
- підтримка різних протоколів: RabbitMQ підтримує різноманітні протоколи обміну повідомленнями, включаючи AMQP, MQTT, STOMP тощо, що робить його гнучким вибором для інтеграції різних систем. В даному випадку використовується протокол AMQP.
- моніторинг та управління: RabbitMQ надає широкі можливості для моніторингу (наприклад, за допомогою окремої панелі керування) і керування механізмами обміну повідомлень, що дозволяє легко стежити за станом системи і вчасно виявляти проблеми.

Протокол OSCPP [9] є стандартним протоколом для обміну повідомленнями з зарядними станціями, що дозволяє нашій системі уникати залежності від коректних

виробників станцій. Даний протокол налічує перелік повідомлень для взаємодії зі станціями. Таким чином, в залежності від типу повідомлення, що було відправлено зарядною станцією, воно буде скеровано до одного з сервісів:

- charge points service – сервіс для роботи з зарядними станціями. Обробляє повідомлення про підключення зарядних станцій до системи (BootNotificationRequest);
- OCPP tags service – сервіс для роботи з авторизаційними тегами, який також обробляє запити на авторизацію (AuthorizeRequest);
- connectors service – сервіс для роботи роз'ємами (конекторами) зарядних станцій. В контексті обробки повідомлень від зарядних станцій цей сервіс відповідальний за обробку зміни статусів роз'ємів (StatusNotificationRequest);
- transactions service – сервіс обробки транзакцій. Сюди надсилаються запити на початок (StartTransactionRequest) та завершення (StopTransactionRequest) транзакції, а також показники спожитої енергії в ході транзакції (MeterValuesRequest);
- reservations service – сервіс обробки резервацій, що відповідає за бронювання та скасування бронювання конекторів (ReserveNowRequest та CancelReservationRequest);
- charging profiles service – сервіс налаштувань профілів зарядки. В контексті обміну повідомленнями цей сервіс відповідає за встановлення або зміну профілю налаштування процесу заряджання. Також цей сервіс відповідальний за налаштування процесів заряджання в рамках депо;
- heartbeats service – сервіс обробки «сердцебиття» станції. Кожна станція під час своєї роботи посилає «сердцебиття», сигналізуючи про те, що вона онлайн.

Після обробки повідомлень згідно з бізнес-логікою сервіси надсилають повідомлення-відповіді на сервіс обробки підключень зарядних станцій, який передає повідомлення самій станції.

Також окрім сервісів, що безпосередньо взаємодіють з зарядними станціями, є ще наступні:

- depots service – сервіс для роботи з депо зарядних станцій;
- user service – сервіс керування користувачами;
- aggregator – імплементація патерну Aggregator. Цей мікросервіс потрібен для того, щоб збирати комплекси дані з різних мікросервісів і далі відправляти ці дані до зовнішніх споживачів. Наприклад, з різних сервісів можна зібрати дані про депо, зарядні станції цього депо та налаштування заряджання. Таким чином, між фронтендом та бекендом знижується кількість запитів;
- gateway – даний мікросервіс реалізує єдину точку входу в API, що побудоване з використанням мікросервісної архітектури. Як можна побачити з діаграми, він буде побудований на основі бібліотеки Ocelot, яка дозволяє швидко створювати API Gateway та зручно його конфігурувати;
- signalR Hub – сервіс для роботи з SignalR для забезпечення доставки повідомлень в реальному часі за допомогою веб-сокетів, так як під час роботи системи користувача необхідно нотифікувати про такі події, як підключення нової станції, перевищення лімітів споживання енергії та ін.

Взаємодія між клієнтською та серверною частинами відбуватиметься за допомогою RESTful контролерів, а взаємодія між сервісами – за допомогою gRPC (окрім випадків, де комунікація відбувається асинхронно за допомогою RabbitMQ). Використання gRPC працює на основі протоколу HTTP/2 [10], який підтримує мультиплексування та стиснення заголовків, що дозволяє зменшити затримку та збільшити швидкість передачі даних між сервісами.

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.

Слід зазначити, що всі сервіси матимуть одне спільне джерело даних, але кожен сервіс буде взаємодіяти лише з тією частиною БД, яка відноситься до зони відповідальності сервісу, тобто таким чином буде реалізовано схему «Shared Instance»[11]. Єдине джерело даних забезпечує можливість використання ACID транзакцій для забезпечення цілісності даних, більш легку розробку та менші фінансові витрати. Головним мінусом такого підходу є те, що зв'яжується слабе місце у роботі системи – якщо база даних виходить з ладу, то виходить з ладу вся система. Для того, щоб нівелювати цей недолік, було прийнято рішення розгорнути базу даних в хмарі Microsoft Azure. Azure забезпечує чудову підтримку Microsoft SQL Server, надаючи механізми автоматичного масштабування, реплікації даних, створення автоматичних бекапів. Також слід відзначити такі механізми, як Locally Redundant Availability та Zone Redundant Availability. Відповідно до цих механізмів БД має декілька копій – три копії в рамках одного датацентру для Locally Redundant Availability або по копії в кожній з трьох зон доступності в рамках регіону Azure. Таким чином, навіть якщо один з серверів або навіть датацентрів виходить з ладу, система все одно матиме доступ до БД.

Також окремо хотілося б звернути увагу на сервіс heartbeats. Так як за замовчуванням серцебиття відправляється кожні декілька хвилин і враховуючи те, що система потенційно працюватиме з великою кількістю станцій, то для зменшення навантаження на основну БД та економії коштів, було прийнято

рішення зберігати дані про сердцебиття в Azure Table Storage в рамках Azure Storage Account. В даному випадку Azure Table Storage працює повільніше ніж Azure SQL Database, але затримки в контексті обробки «сердцебиття» станції не такі критичні у порівнянні з, наприклад, обробкою транзакцій.

3.2 Проєктування БД

На основі даних, отриманих в ході аналізу предметної області, висунутих вимог, а також особливостей протоколу OCPP, було побудовано наступну ER-діаграму (рисунок Б.1 додатка Б).

Розглянемо кожну сутність окремо. Виділимо атрибути сутності «Часова зона»:

- #Id – ідентифікатор зони, первинний ключ;
- назва – назва часової зони;
- відступ відносно UTC – зсув у годинах відносно UTC – Coordinated Universal Time;
- ідентифікатор IANA – ідентифікатор часової зони відповідно до Internet Assigned Numbers Authority;
- ідентифікатор Windows – ідентифікатор часової зони в системі Windows;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Зберігання ідентифікаторів IANA та Windows необхідно для того, щоб часові зони можна було коректно відобразити на клієнтській частині.

Сутність «Депо» включає інформацію про депо, що знаходиться в одній з часових зон:

- #Id – ідентифікатор депо, первинний ключ;
- назва – назва депо;
- країна, місто, вулиця, номер будинку, широта та довгота – географічні дані про депо;
- опис – додатковий опис депо за необхідності;

- email та телефон – контактні дані депо;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

До сутності «Зарядна станція» належать такі атрибути:

- #Id – ідентифікатор автомобіля, первинний ключ;
- назва – назва станції;
- опис – додатковий опис за необхідності;
- статус в системі – станція може мати різні статуси під час роботи з системою, наприклад, до фізичного підключення станції, запис в БД про цю станцію міститиме статус «Створено»;
- виробник – виробник станції;
- модель – модель станції;
- серійний номер станції;
- версія прошивки;
- дата оновлення прошивки;
- ICCID (Integrated Circuit Card Identifier) – унікальний ідентифікатор, що присвоюється кожній сім-карті. У контексті зарядної станції для електромобілів ICCID може використовуватися, якщо зарядна станція підключається до центральної системи управління через мобільну мережу;
- IMSI (International Mobile Subscriber Identity) – це унікальний номер, який використовується для ідентифікації абонентів у мобільних телефонних мережах. У контексті зарядних станцій для електромобілів IMSI використовується в тому випадку, якщо станція оснащена SIM-картою для забезпечення зв'язку з центральною системою керування через мобільний інтернет;
- тип лічильника електроенергії;
- серійний номер лічильника;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

До сутності «Роз'єм» належать такі атрибути:

- #Id – ідентифікатор роз'єму, первинний ключ;
- номер роз'єму – числовий номер роз'єму в рамках зарядної станції;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Сутності «Статус роз'єму» належать такі атрибути:

- #Id – ідентифікатор запису про статус;
- поточний статус – поточний статус роз'єму відповідно до протоколу OCPP;
- код помилки – в разі помилки станція може надіслати її код разом зі статусом;
- додаткова інформація – за необхідністю станція також може надіслати додаткову інформацію;
- дата оновлення статусу;
- код помилки виробника та ідентифікатор виробника – в залежності від виробника станції, в разі помилки, станція може надсилати додаткову інформацію про помилку, що є специфічною саме для виробника;

Сутність «Показник» містить інформацію про показники, що отримуються під час заряджання:

- #Id – ідентифікатор запису про показники, первинний ключ;
- значення – значення показника;
- формат – в якому форматі дані передано;
- фаза – деякі станції під час передачі деяких показників передають також інформацію про фазу вказуючи, як слід інтерпретувати виміряне значення. (Наприклад, між L1 і нейтраллю (L1-N));
- вимірювана величина – потужність, сила току, напруга і т.д.;
- одиниця виміру – вольти для напруги, амperi для сили току, вати або кіловати для потужності і т.д.;
- дата зняття показника.

Сутності «Транзакція» належать такі атрибути:

- #Id – ідентифікатор транзакції, первинний ключ;
- номер транзакції – в контексті роботи протоколу ОСРР, кожна транзакція мусить мати унікальний числовий номер;
- дата початку транзакції;
- дата закінчення транзакції;
- причина закінчення – додаткова інформація про причину закінчення, наприклад, успішне закінчення чи закінчення через технічний збій.

Сутність «Бронювання» включає інформацію про бронювання роз'ємів:

- #Id – ідентифікатор бронювання;
- назва та опис – додаткові необов'язкові параметри;
- дата початку;
- дата закінчення;
- статус – статус в системі. Коли запис про бронь додається, то бронь має статус «створена», після відправки запиту на станцію – «запит відправлено», після підтвердження станцією – «підтверджена»;
- скасована – прапорець для індикації того, чи було бронь скасовано;
- використана – прапорець для індикації того, чи було бронь використано;
- номер бронювання – як і з транзакціями, в контексті роботи протоколу ОСРР, кожне бронювання мусить мати унікальний числовий номер;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.
- До сутності «Сердцебиття», що показує, чи є станція онлайн, належать такі атрибути:
 - #Id – ідентифікатор запису, первинний ключ;
 - Дата «сердцебиття».

Сутності «ОСРР тег», що містить в собі дані про теги для ідентифікації користувача станцією, належать такі атрибути:

- #Id – ідентифікатор тега, первинний ключ;

- назва тегу;
- заблокований – індикатор того, чи було заблоковано тег системою;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Сутність «Профіль заряджання» містить в собі налаштування процесу заряджання для роз'єму або всієї зарядної станції або так званий «профіль». До цієї сутності належать такі атрибути:

- #Id – ідентифікатор профілю, первинний ключ;
- номер профілю – протокол OCPP вимагає, щоб кожний профіль мав унікальний числовий номер;
- дійсний з – дата, з якої налаштування є дійсними;
- дійсний до – дата, до якої налаштування є дійсними;
- тривалість – в який проміжок часу буде працювати цей профіль, наприклад, профіль може бути дійсний місяць, але використовуватись кожен день по годині;
- рівень в стеку – значення, що визначає рівень в ієрархічному стеку профілів. Вищі значення мають перевагу над нижчими. Найнижчий рівень 0;
- мета – визначає призначення розкладу, переданого цим повідомленням. Може мати наступні значення: ChargePointMaxProfile – конфігурація для максимальної потужності або струму, доступного для всієї зарядної станції; TxDefaultProfile – профіль за замовчуванням. Коли розпочинається нова транзакція, цей профіль мусить використовуватися крім тих випадків, коли транзакція була розпочата віддалено запитом RemoteStartTransaction.req, в якому було передано інший профіль; TxProfile – профіль для поточної транзакції. Є валідним до закінчення поточної транзакції;
- тип – тип профілю. Може бути абсолютним (періоди розкладу відносяться до фіксованого моменту часу, визначеного в розкладі.), відносним (періоди

розкладу відносяться до конкретної ситуації початкової точки (наприклад, початку транзакції), яку визначає зарядна станція.) та повторюваним (розклад періодично перезапущається в перший період розкладу).

- дата початку застосування – початкова точка абсолютного розкладу. Якщо його немає, графік буде відносним до початку заряджання.
- тип повторюваності – вказує на початкову точку повторення. Може бути щоденним і щотижневим. Щоденний вказує на те, що розклад перезапущається кожні 24 години в той самий час, що вказаний в полі «дата початку застосування». Щотижневий вказує на те, що розклад перезапущається кожні 7 днів у той самий час і день тижня, що вказані в полі «дата початку застосування»;
- одиниця вимірювання – визначає в яких одиницях вимірювання визначається ліміт споживання енергії. Це можуть бути вати або амperi;
- мінімальна швидкість заряджання – опціональний параметр. Мінімальна швидкість зарядки, яку підтримує електромобіль. Одиниця вимірювання визначається значенням з поля «одиниця вимірювання». Цей параметр призначений для використання алгоритмом заряджання зарядної станції для оптимізації розподілу потужності у випадку, якщо процес заряджання є неефективним при нижчих рівнях заряджання;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Сутність «Період графіка профілю заряджання» включає інформацію про періоди, які входять до профілю заряджання, і містить такі атрибути:

- #Id – ідентифікатор періоду, первинний ключ;
- початок періоду – кількість секунд, що визначають початок періоду відносно початку розкладу;
- ліміт – максимальна потужність або сила струму, що може бути використана;

- кількість фаз – додатковий параметр – кількість фаз, які можна використовувати для зарядки (три за замовчуванням);
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Використання профілів зарядки дозволяє гнучко налаштовувати параметри заряджання для кожного роз'єму зарядної станції, а також задавати ліміти сили току та потужності.

Сутність «Налаштування споживання енергії депо» призначена для того, щоб налаштувати кількість Ват-годин, яка може бути спожита депо, і містить такі поля:

- #Id – ідентифікатор налаштувань, первинний ключ;
- «дійсні з» та «дійсні до» – з якої дати та по яку налаштування є дійсними;
- ліміт споживання енергії – скільки енергії може спожити депо;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Якщо за допомогою профілів зарядки можна обмежувати потужність та силу току, то сутність «Налаштування споживання енергії станцією» дозволяє налаштовувати кількість спожитих Ват-годин, тобто енергії, в рамках лімітів споживання депо. Ця сутність містить такі поля:

- #Id – ідентифікатор налаштувань, первинний ключ;
- «дійсні з» та «дійсні до» – з якої дати та по яку налаштування є дійсними;
- ліміт споживання енергії – скільки енергії може спожити станція;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Сутність «Інтервал споживання енергії» схожа на «Період графіка профілю заряджання», але використовується для налаштування інтервалів споживання енергії (Ват-годин) в рамках ліміту споживання для всього депо. Ця сутність містить такі поля:

- #Id – ідентифікатор інтервалу, первинний ключ;

- «початок інтервалу» та «кінець інтервалу» – початок та кінець інтервалу відповідно;
- ліміт споживання енергії – скільки енергії може бути спожито протягом цього інтервалу;
- дата створення та дата оновлення – технічні дані про те, коли було додано або оновлено запис.

Сутність «Користувач» містить інформацію про користувачів системи:

- #Id – ідентифікатор користувача, первинний ключ;
- пароль – пароль користувача;
- прізвище – прізвище користувача;
- ім'я – ім'я користувача;
- телефон – телефон користувача;
- email – електронна пошта користувача.

До сутності «Роль» належать такі атрибути:

- #Id – ідентифікатор ролі, первинний ключ;
- назва – назва ролі, до якої належить користувач.

Розглянувши детально сутності, можна переходити до опису зв'язків між ними. Між сутностями «Часова зона» і «Депо» зв'язок типу «один-до-багатьох», тому що в одній часовій зоні може знаходитись декілька депо.

Аналогічно моделюємо зв'язок «один-до-багатьох» між сутностями «Депо» і «Зарядна станція», тому що до в одному депо може бути багато зарядних станцій.

Кожна станція може мати декілька роз'ємів, тож між сутностями «Зарядна станція» та «Роз'єм» зв'язок типу «один-до-багатьох».

Протягом часу статус роз'ємів може змінюватись, тобто один роз'єм в ході використання може мати багато статусів, тож зв'язок між сутностями «Роз'єм» та «Статус роз'єму» «один-до-багатьох».

З використанням одного роз'єма може бути виконано багато транзакцій, тож між сутностями «Роз'єм» та «Транзакція» «один-до-багатьох». Також в ході однієї

транзакції передається багато показників, тому між сутностями «Транзакція» та «Показник» також зв'язок «один-до-багатьох».

На один роз'єм може бути багато бронювань на різний час, таким чином маємо зв'язок «один-до-багатьох» між сутностями «Роз'єм» та «Бронювання». Одне бронювання відповідає лише одній транзакції, тому сутностями «Транзакція» та «Бронювання» є зв'язок «один-до-одного».

Впродовж своєї роботи станція відправляє багато сигналів про «сердцебиття», тобто між сутностями «Зарядна станція» та «Сердцебиття» зв'язок «один-до-багатьох».

Стосовно профілів заряджання слід зазначити, що один і той самий профіль заряджання може бути застосовано на різних роз'ємах, а один роз'єм може комбінувати декілька профілів заряджання, тож між сутностями «Профіль заряджання» та «Роз'єм» є зв'язок «багато-до-багатьох». Також в рамках одного профілю заряджання є декілька періодів, що означає наявність зв'язку «один-до-багатьох» між сутностями «Профіль заряджання» та «Період графіка профілю заряджання».

Фактично депо в конкретний момент часу може мати лише одні налаштування споживання енергії, але з метою збереження історичних даних для перегляду того, які налаштування були застосовані до депо, зв'язок між сутностями «Депо» та «Налаштування споживання енергії депо» зв'язок «один-до-багатьох». Так само з налаштуваннями споживання енергії для станції, тобто сутності «Зарядна станція» та «Налаштування споживання енергії станцією» мають зв'язок «один-до-багатьох». Налаштування споживання енергії депо можуть містити декілька інтервалів, що означає наявність зв'язку «один-до-багатьох» у сутностей «Налаштування споживання енергії депо» та «Інтервал споживання енергії».

Говорячи про ОСРР-теги, слід зазначити, що за допомогою одного тега можна зробити багато транзакцій та багато бронювань, тобто між сутностями «ОСРР тег» та «Транзакція», а також «ОСРР тег» та «Бронювання» буде зв'язок «один-до-багатьох».

В свою чергу, користувачі можуть мати лише один тег, тож зв'язок між сутністю «Користувач» та сутністю «ОСРР тег» є «один-до-одного». Також користувач одночасно може мати багато ролей, а одна й та сама роль може бути застосована до багатьох користувачів, тобто між користувачами та ролями є зв'язок «багато-до-багатьох». Так само зв'язок «багато-до-багатьох» є між депо та користувачем, бо користувач одночасно може мати доступ до декількох депо, а до одного депо можуть мати доступ декілька користувачів.

Після того, як була сформована ER-діаграма закінчено, було створено логічну схему бази даних (рисунк Б.2 додатка Б), в якій сутності та атрибути було перекладено на англійську мову, а також було додано проміжні таблиці для зв'язків «багато-до-багатьох».

Сутності та їх атрибути з ER-діаграми було перекладено на англійську. Переклад сутностей наведено в таблиці 4.1. Переклад атрибутів наведено в таблиці 4.2.

Таблиця 4.1 – Переклад сутностей з української на англійську (таблиця виконана самостійно)

Сутність у ER-діаграмі	Сутність у логічній моделі
1	2
Часова зона	TimeZone
Депо	Depot
Зарядна станція	ChargePoint
Роз'єм	Connector
Статус роз'єму	ConnectorStatus
Транзакція	Transaction
Показник	MeterValue
Бронювання	Reservation
ОСРР тег	OcppTag
Користувач	User

Кінець таблиці 4.1

1	2
Роль	Role
Профіль заряджання	ChargingProfile
Період графіка профілю заряджання	ChargingSchedulePeriod
Налаштування споживання енергії депо	DepotEnergyConsumptionSettings
Налаштування споживання енергії станцією	ChargePointEnergyConsumptionSettings
Інтервал споживання енергії	EnergyConsumptionIntervalSettings
Сердцебиття	Heartbeat

Таблиця 4.2 – Переклад атрибутів з української на англійську (таблиця виконана самостійно)

Атрибут у ER-діаграмі	Атрибут у логічній моделі
1	2
Назва	Name
Відступ відносно UTC	UtcOffset
Ідентифікатор Iana	IanaId
Ідентифікатор Windows	WindowsId
Дата створення	CreatedAt
Дата оновлення	UpdatedAt
Країна	Country
Місто	City
Вулиця	Street
Номер будинку	Building
Широта	Latitude

Продовження таблиці 4.2

1	2
Довгота	Longitude
Опис	Description
Email	Email
Телефон	Phone
Статус в системі	RegistrationStatus
Протокол	OcppProtocol
Виробник	ChargePointVendor
Модель	ChargePointModel
Версія прошивки	FirmwareVersion
Дата оновлення прошивки	FirmwareUpdateTimestamp
Iccid	Iccid
Imsi	Imsi
Тип лічильника	MeterType
Серійний номер лічильника	MeterSerialNumber
Дата останньої діагностики	DiagnosticsTimestamp
Серійний номер	ChargePointSerialNumber
Номер роз'єму	ConnectorNumber
Поточний статус	CurrentStatus
Код помилки	ErrorCode
Додаткова інформація	Info
Код помилки виробника	VendorErrorCode
Ідентифікатор виробника	VendorId
Дата оновлення статусу	StatusUpdatedTimestamp
Номер транзакції	TransactionNumber
Дата початку	StartTime
Дата закінчення	StopTime

Продовження таблиці 4.2

1	2
Причина закінчення	StopReason
Значення	Value
Формат	Format
Вимірювана величина	Measurand
Одиниця виміру	Unit
Фаза	Phase
Дата зняття показника	MeterValueTimestamp
Назва тега	TagName
Заблокований	Blocked
Ім'я	Name
Прізвище	Surname
Пароль	Password
Статус	Status
Скасована	IsCancelled
Використана	IsUsed
Номер бронювання	ReservationNumber
Номер профілю	ChargingProfileNumber
Рівень в стеку	StackLevel
Дійсний з	ValidFrom
Дійсний до	ValidTo
Тип повторювання	RecurrencyKind
Мета	ChargingProfilePurpose
Тип	ChargingProfileKind
Тривалість	Duration
Дата початку застосування	StartSchedule
Одиниця вимірювання	SchedulingUnit

Кінець таблиці 4.2

1	2
Мінімальна швидкість заряджання	MinChargingRate
Старт періоду з	StartPeriod
Ліміт споживання	Limit
Кількість фаз	NumberPhases
Дійсні з	ValidFrom
Дійсні до	ValidTo
Початок інтервалу	StartTime
Кінець інтервалу	EndTime

Окремо варто звернути увагу на таблицю для сутності «Сердцебиття», де можна побачити PartitionKey та RowKey. Як було зазначено в розділі 3.1, для зберігання записів про «сердцебиття» зарядних станцій буде використано Azure Table Storage Service. В таблицях цього сервісу дані зберігаються по розділам, тобто кожний запис має ключ розділу (PartitionKey) та ключ рядка в межах розділу (RowKey). В даному випадку ключ розділу являтиме собою ідентифікатор станції, а ключ рядка генеруватиметься за допомогою реалізації патерну Log tail, що забезпечить автоматичне сортування рядків у зворотному порядку для отримання актуальної інформації про статус зарядної станції.

3.3 Розгортання рішення, DevOps та CI/CD

Після того, як систему буде побудовано, її треба розгорнути в хмарі. Враховуючи, що для імплементації системи використовується мікросервісний підхід, то кращим вибором, як для локального запуску, так і для розгортання, буде використання інструментів контейнеризації, а саме – Docker [12].

Для оркестрації та масштабування можна буде використовувати Kubernetes [13]. Таким чином, Docker забезпечуватиме 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. Також слід зазначити, що хмара Azure має чудову підтримку обох цих сервісів. Для того, щоб забезпечити надійне місце зберігання та приватність для Docker images, буде використано Azure Container Registry Service [14] – Azure сервіс, що надає можливість мати свої приватні репозиторії для Docker images.

Для імплементації процесів CI/CD буде використано інструмент Azure Devops, а саме – Azure DevOps Pipelines [15]. Тобто коли відбуваються зміни в коді, на це реагуватиме Azure DevOps Pipeline, який буде отримувати останню версію коду з репозиторію GitHub, будувати з нього Docker images, доставляти ці images в Azure Container Registry, а далі оновлені версії images будуть запущені в якості контейнерів за допомогою Azure Kubernetes – імплементації Kubernetes в якості Platform as a Service у хмарі Azure.

На даному етапі, базу даних, архітектуру серверної частини, а також процеси CI/CD можна вважати спроектованими.

4 ОПИС ПРИЙНЯТИХ ПРОГРАМНИХ РІШЕНЬ

4.1 Опис програмного рішення

Бекенд частина рішення є комплексною та складається з декількох частин. На рисунку 4.1 можна побачити його загальну структуру.

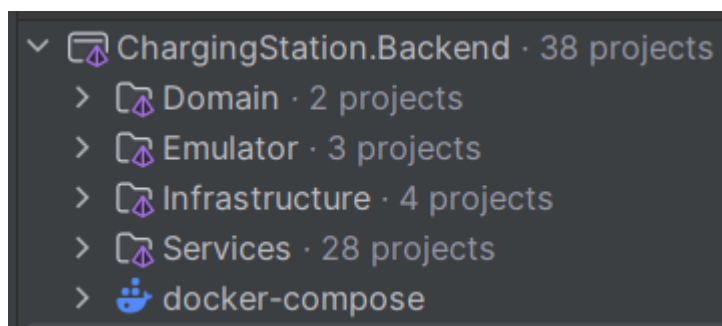


Рисунок 4.1 – Загальна структура проєкту (рисунок виконано самостійно)

Отже, з рисунку можна побачити, що рішення складається з наступних частин:

- Domain – містить загальні базові компоненти та доменні моделі системи;
- Emulator – містить логіку емулятора зарядних станцій;
- Infrastructure – містить інфраструктурні компоненти, такі як робота з поштою, базою даних, кешем і т.д.
- Services – містить мікросервіси.

4.1.1 Опис частини Services

Як було зазначено вище, дана частина містить мікросервіси для роботи з різними аспектами системи. Більш детальну структуру цієї частини можна побачити на рисунках 4.2 та 4.3.

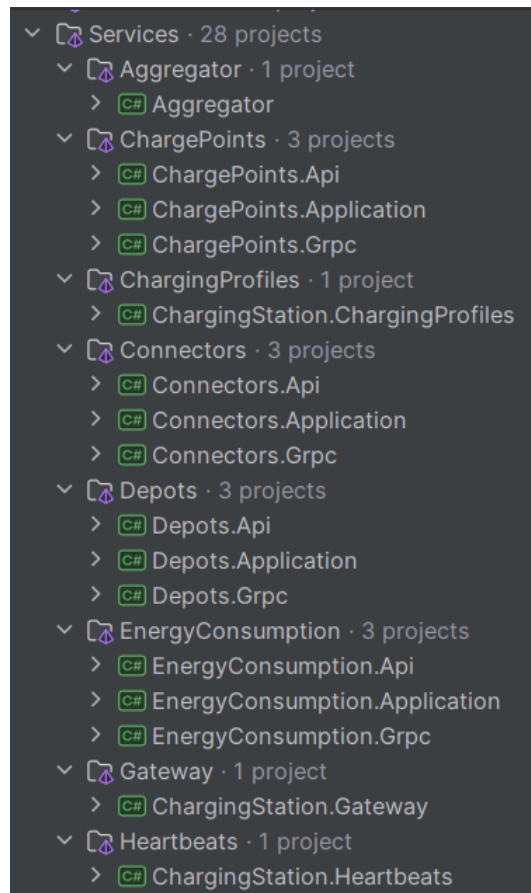


Рисунок 4.2 – Структура частини Services (рисунок виконано самостійно)

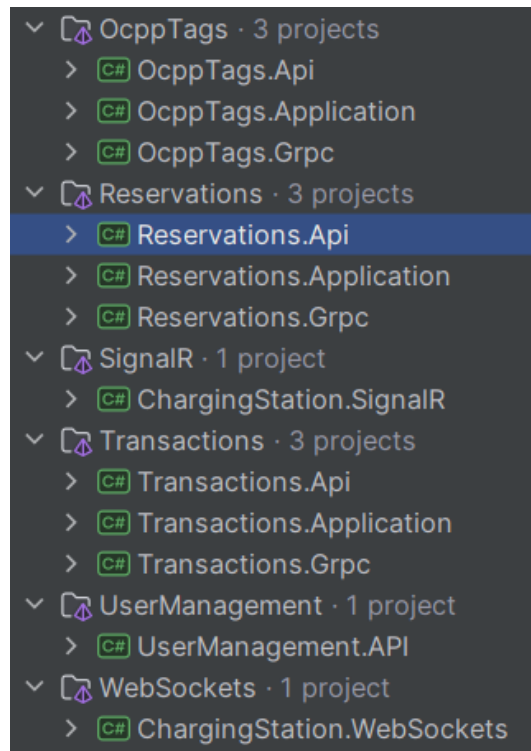


Рисунок 4.3 – Структура частини Services (рисунок виконано самостійно)

Отже, з рисунків можна побачити, що система налічує такі мікросервіси:

- Depots.Api та Depots.Grpc – мікросервіси для роботи з депо (створення, оновлення, отримання, видалення). Api мікросервіс використовується для зовнішньої комунікації за допомогою RESTful контролерів та протоколу HTTP, gRPC мікросервіс використовується для внутрішньої комунікації за допомогою gRPC сервісів та протоколу RPC. Спільні компоненти, що використовуються обома сервісами, розташовані в проєкті Depots.Application.
- ChargePoints.Api та ChargePoints.Grpc – мікросервіси для роботи з зарядними станціями (створення, оновлення, обробка подій підключення станції і т.д.). За аналогією до попередніх сервісів, Api призначений для комунікації з зовнішнім світом, Grpc – для синхронної міжсервісної комунікації, а Application проєкт містить їх спільні компоненти.
- Connectors.Api та Connectors.Grpc – мікросервіси для керування роз'ємами зарядних станцій. Відповідають за керування станом роз'ємів зарядних станцій.
- OcppTags.Api та OcppTags.Grpc – мікросервіси для керування OCPP тегами – ключовими елементами в контексті ідентифікації та авторизації користувачів.
- Heartbeats – мікросервіс для обробки так званого «серцебиття» зарядної станції, що є аналогом ping-pong в контексті мережевої комунікації наприклад в SignalR. За допомогою обробки «серцебиття» зарядної станції, можна зрозуміти, чи підключена вона до центральної системи.
- Transactions.Api та Transactions.Grpc – мікросервіси обробки транзакцій, тобто сесій заряджання. Також за допомогою API можна подивитись інформацію щодо спожитої енергії зарядними станціями в рамках транзакцій або за певний проміжок часу.

- Reservations.Api та Reservations.Grpc – мікросервіси для керування бронюванням роз'ємів зарядних станцій. Ці мікросервіси відповідальні за те, щоб користувачі могли завчасно забронювати роз'єм для зарядки.
- ChargingProfiles – мікросервіс, що відповідальний за так звані «профілі зарядки» – спеціальний перелік налаштувань, до яких входять періоди споживання енергії, мінімальні межі та ліміти потужності або сили струму. Таким чином завдяки цьому мікросервісу імплементовано механізм «smart charging» протоколу OCPP.
- EnergyConsumption.Api та EnergyConsumption.Grpc – мікросервіси для керування кількістю спожитої енергії в рамках депо. Вони дозволяють налаштовувати періоди та ліміти споживання енергії як для кожної станції в рамках депо, так і для всього депо разом. В той час як профілі заряджання, які є частиною протоколу OCPP, дозволяють обмежувати силу струму в амперах та потужність у ватах для окремих зарядних станцій, даний механізм є власною надбудовою, що використовує дані, отримані від станцій за допомогою протоколу OCPP, що дозволяє задати ліміти спожитої енергії у Ват-годинах в рамках цілого депо протягом деякого періоду часу та більш тонко доналаштувати ці ліміти для окремих станцій. Підсумовуючи, можна зазначити, що профілі заряджання дозволяють обмежувати споживання енергії за одиницю часу, тобто потужність та силу струму, а налаштування споживання енергії депо – задають певний ліміт у Ват-годинах для всього депо протягом якогось періоду часу;
- UserManagement – мікросервіс керування користувачами;
- SignalR – мікросервіс з SignalR хабом, для імплементції надсилання сповіщень на клієнтську сторону в реальному часі без необхідності перезавантаження сторінки;
- WebSockets – мікросервіс, що забезпечує обробку підключень від зарядних станцій та комунікацію між станціями та іншими мікросервісами. Комунікація зі станціями відбувається за допомогою протоколу OCPP 1.6j,

що базується на протоколі WebSockets, а комунікація з іншими мікросервісами відбувається асинхронно за допомогою брокера повідомлень RabbitMQ.

Також в системі мікросервіси Aggregator та Gateway, використання яких більш детально буде описано далі.

4.1.2 Використання Aggregator

В переліку мікросервісів можна побачити сервіс, що називається Aggregator, що є імплементацією однойменного патерну в мікросервісній архітектурі.

У архітектурі мікросервісів велику складна програма розділена на невеликі незалежні автономні служби. Тому необхідно подумати про те, як об'єднати дані, які повертає кожна служба. В ІТ-індустрії агрегатор відноситься до веб-сайту або програми, яка збирає пов'язані елементи даних і відображає їх. Отже, у мікросервісах Aggregator Design Pattern – це сервіс, який отримує запит, потім робить запити до кількох сервісів, об'єднує результати та відповідає на початковий запит. За необхідності у великих системах може бути декілька агрегаторів в залежності від тієї частини предметної галузі, яку вони обробляють.

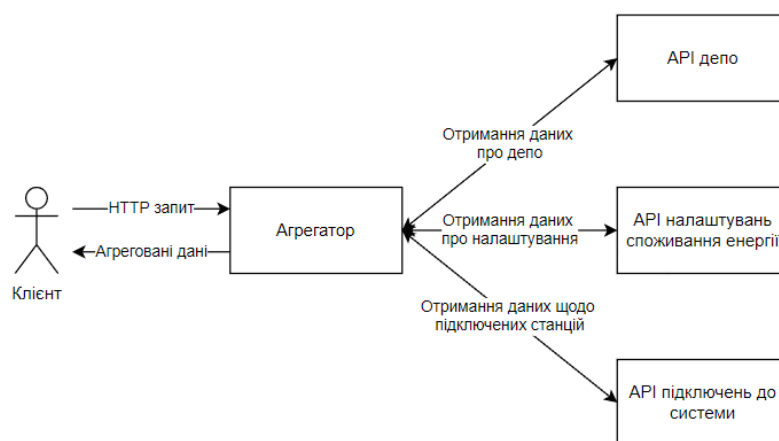


Рисунок 4.4 – Схематичне зображення використання агрегатора (рисунок виконано самостійно)

В контексті поточної системи (рисунок 4.4), агрегатор відповідає за те, щоб збирати комплексні дані про депо: базові дані про депо, що повертає сервіс роботи з депо, налаштування споживання енергії, за які відповідає сервіс налаштувань споживання енергії, а також статистика по підключеним зарядним станціям, за що відповідає сервіс обробки підключень.

Застосування такого підходу дозволяє знизити навантаження на клієнтський застосунок шляхом зменшення кількості запитів до різних сервісів та переносу агрегації даних на бекенд, що значно полегшує відображення комплексного UI.

4.1.3 Використання Gateway

Хоча агрегатор дозволяє зменшити кількість запитів від клієнтських застосунків до бекенду, все одно залишається проблема того, що фронтенду необхідно звертатися по різним адресам різних мікросервісів, щоб отримати різні дані.

Для забезпечення єдиної точки входу в систему використовується API Gateway. Також це дозволяє зробити нашу систему більш захищеною, тому що при використанні API Gateway адреси та порти мікросервісів можна не відкривати для загального доступу з мережі, обмежуючись тільки сервісом Gateway. Окрім забезпечення єдиної точки входу в систему, API Gateway також дозволяє реалізувати аутентифікацію та авторизацію, кешування, балансування навантаження та ін.

Для імплементації API Gateway було обрано фреймворк Ocelot, що дозволяє швидко та легко за допомогою простої конфігурації створити власний API Gateway. Приклади конфігурації можна побачити на рисунку 4.5.

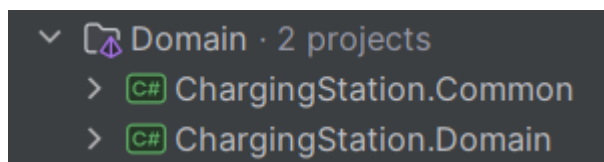
```
{
  "DownstreamPathTemplate": "/api/user/{id}",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "usermanagement.api",
      "Port": 8080
    }
  ],
  "UpstreamPathTemplate": "/api/user/{id}",
  "UpstreamHttpMethod": ["Get"]
},
```

Рисунок 4.5 – Частина конфігурації API Gateway на основі Ocelot (рисунок виконано самостійно)

З рисунку можна побачити частину конфігурації маршрутів. Наприклад, якщо від клієнта надходить запит GET за шаблоном `/api/user/{id}`, то необхідно перенаправити цей запит до API `usermanagement` на порт 8080 за таким самим шаблоном.

4.1.4 Опис частини Domain

З рисунку 4.1 можна також побачити, що в проєкті міститься частина під загальною назвою `Domain`. Структуру цієї частини можна побачити на рисунку 4.6.



```

v [Domain] · 2 projects
  > [C#] ChargingStation.Common
  > [C#] ChargingStation.Domain
  
```

Рисунок 4.6 – Структура частини Domain (рисунок виконано самостійно)

З рисунку видно, що частина `Domain` включає в себе два проєкти:

- Common – проєкт, що містить загальні компоненти, що використовуються всіма сервісами, наприклад виключення, конфігурації, повідомлення протоколу OCSP;
- Domain – проєкт, що містить доменні моделі бази даних. Рішення щодо інкапсуляції доменних моделей в один проєкт пов'язане з тим, що відповідно до пункту 3.1 в якості джерела даних буде використовуватись спільна хмарна БД у форматі Shared Instance.

Об'єднання спільних компонентів для різних мікросервісів в рамках одного проєкту має свої недоліки та переваги. Головним недоліком є те, що при зміні коду проєкту із загальними компонентами у випадку, наприклад, виявлення помилки, може знадобитись перерозгортання всіх мікросервісів, що може призвести того, що система буде деякий час недоступна. З іншого боку, якщо робити для кожного мікросервісу окрему незалежну кодову базу з базовими компонентами, що будуть дублюватись в коді кожного мікросервісу, то при виявленні критичної помилки в коді базових компонентів все одно доведеться перерозгортати всі мікросервіси, що так само призведе до недоступності системи. Таким чином, перевагу було віддано підходу, який забезпечить необхідність виправлень лише в одному місці, тобто використанню спільного проєкту з базовими компонентами, що значно полегшуватиме підтримку рішення в майбутньому.

4.1.5 Опис частини Infrastructure

Частина Infrastructure, як можна зрозуміти за назвою, містить інфраструктурні частини системи, які згруповані в наступних проєктах:

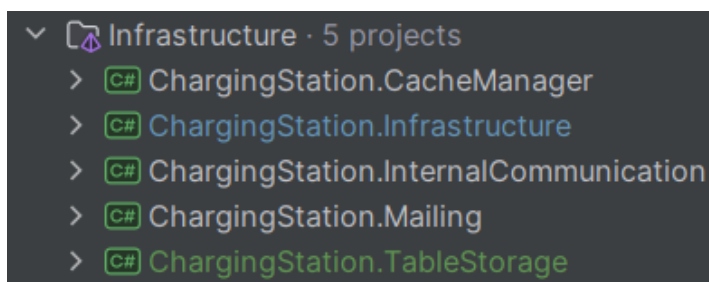


Рисунок 4.7 – Структура частини Infrastructure (рисунок виконано самостійно)

З рисунку 4.7 можна побачити, що частина Infrastructure налічує наступні проєкти:

- CacheManager – проєкт, що містить сервіси для роботи з Redis Cache. Застосування кешу буде описано більш детально далі;
- Infrastructure – проєкт, що містить головну інфраструктурну частину – роботу з БД через Entity Framework;
- InternalCommunication – як можна зрозуміти з назви, цей проєкт містить в собі спільні компоненти, що забезпечують міжсервісну комунікацію за допомогою протоколів HTTP та gRPC;
- Mailing – проєкт, що інкапсулює необхідні сервіси для надсилання листів електронною поштою за допомогою MailJet;
- TableStorage – відповідно до пункту 3.1, де було прийнято рішення про використання Azure Table Storage для зберігання інформації про «сердцебиття» станцій, цей проєкт містить інфраструктурні частини для роботи саме з Azure Table Storage.

Слід зазначити, що інфраструктурні проєкти побудовані по такому ж принципу, як і проєкти частини Domain, тобто можуть бути перевикористані в різних проєктах з метою зменшення дублювання коду.

4.1.6 Робота з Entity Framework

Взаємодія серверної частини з БД здійснюватиметься за допомогою ORM Entity Framework Core. Entity Framework (EF) – це високопродуктивна ORM, що надає значні переваги при розробці серверної частини програмного забезпечення:

- усунення необхідності написання SQL: EF автоматично генерує SQL для виконання операцій вставки, видалення, оновлення та вибірки на основі об'єктів, що використовуються в системі. Такий підхід також дозволяє уникати SQL-ін'єкцій, підвищуючи таким чином загальний рівень безпеки системи;

- підхід «Code first»: EF дозволяє працювати з даними на більш високому рівні абстракції, ніж SQL, тобто на рівні C# об'єктів, спрощуючи розробку та підтримку коду;
- контроль версій БД: Entity Framework дозволяє вести контроль версій бази даних з використанням міграцій, що є важливою особливістю при ітеративній розробці;
- підтримка LINQ, що дозволяє виконувати потужні, типобезпечні (type-safe) запити прямо в C#, без використання рядків SQL;
- спрощення тестування: EF дозволяє використовувати «підроблені» або in-memory бази даних при модульному тестуванні, що значно спрощує тестування коду, що взаємодіє з базою даних;
- EF підтримує роботу не тільки з MS SQL Server, але й з іншими СУБД, такими як PostgreSQL, MySQL, SQLite тощо. Таким чином, у випадку зміни СУБД, модифікації коду будуть мінімальні;
- EF добре інтегрується з іншими частинами .NET екосистеми, такими як ASP.NET, що спрощує створення повнофункціональних веб-додатків.

4.1.7 Робота з RabbitMQ та MassTransit

Відповідно до пункту 3.1 під час проєктування було визначено, що для асинхронної комунікації в системі буде використовуватись брокер повідомлень RabbitMQ. Для програмної реалізації роботи з RabbitMQ було обрано бібліотеку MassTransit. Використання такого інструменту надає наступні переваги:

- абстракція від низькорівневих деталей та простота використання. MassTransit дозволяє розробникам абстрагуватися від низькорівневих деталей конкретного брокера повідомлень (в даному випадку RabbitMQ), таких як управління чергами, обмінами і з'єднаннями, забезпечуючи простий та зрозумілий API для роботи з брокером повідомлень. Це дозволяє розробникам зосередитися саме на бізнес-логіці та спрощує розробку та підтримку системи;

- підтримка різних засобів обміну повідомленнями: окрім RabbitMQ, MassTransit підтримує інші брокери повідомлень, такі як Azure Service Bus, Amazon SQS, і ActiveMQ. Це забезпечує гнучкість у виборі інфраструктури для вашого проекту, а також мінімізує втручання в код у разі переходу на іншу інфраструктуру для обміну повідомленнями;
- розширені можливості маршрутизації: MassTransit підтримує складні схеми маршрутизації повідомлень, що дозволяє легко реалізовувати патерни, такі як publish/subscribe, send/receive та запит/відповідь;
- підтримка автоматичної ретрансляції: MassTransit надає вбудовану підтримку для обробки помилок і повторних спроб доставки повідомлень, що підвищує надійність системи;
- детальне логування та моніторинг: MassTransit має детальне логування, що допомагає швидко виявляти і вирішувати проблеми.
- гнучка конфігурація: MassTransit дозволяє гнучко налаштовувати з'єднання, маршрутизацію і обробку повідомлень, що забезпечує можливість точного налаштування системи відповідно до потреб системи.

```

services.AddMassTransit(busConfigurator =>
{
    busConfigurator.SetKebabCaseEndpointNameFormatter();

    busConfigurator.AddConsumer<StatusNotificationConsumer>();
    busConfigurator.AddConsumer<ChangeAvailabilityResponseConsumer>();

    busConfigurator.UsingRabbitMq( configure: (ctx :IBusRegistrationContext , cfg :IRabbitMqBusFactoryConfigurator ) =>
    {
        var connectionString = configuration.GetSection( key: MessageBrokerConfiguration.SectionName ) // IConfigurationSection
        |.Get<MessageBrokerConfiguration>().GetConnectionString();
        cfg.Host(connectionString);

        cfg.ReceiveEndpoint( queueName: "status-notification-queue-1_6", configureEndpoint: c :IRabbitMqReceiveEndpointConfigurator => {
            c.ConfigureConsumer<StatusNotificationConsumer>(ctx);
        });

        cfg.ReceiveEndpoint( queueName: "change-availability-response-queue-1_6", configureEndpoint: c :IRabbitMqReceiveEndpointConfigura
            c.ConfigureConsumer<ChangeAvailabilityResponseConsumer>(ctx);
        });
    });
});

```

Рисунок 4.8 – Приклад конфігурації MassTransit для RabbitMQ

На рисунку 4.8 можна побачити налаштування MassTransit для RabbitMQ. З рисунку можна побачити, що налаштування цілком прості та інтуїтивно зрозумілі. Все що потрібно для роботи, додати споживачів повідомлень (метод `AddConsumer`), зазначити використання RabbitMQ (метод `UsingRabbitMq`) та сконфігурувати використання RabbitMQ, вказавши рядок підключення та зв'язавши споживачів з конкретними RabbitMQ чергами повідомлень (методи `cfg.Host` та `cfg.ReceiveEndpoint`). В даному випадку на рисунку можна побачити налаштування поживачів таких повідомлень, як `StatusNotification` (запит від станції щодо зміни статусу роз'єма) та `ChangeAvailabilityReponse` (відповідь від станції щодо повідомлення про зміну доступності роз'єма). Клас споживача повідомлень на прикладі `StatusNotificationConsumer` можна побачити на рисунку 4.9.

```

4 usages  Yevhen Chubarov +1
public class StatusNotificationConsumer : IConsumer<IntegrationOcppMessage<StatusNotificationRequest>>
{
    private readonly ILogger<StatusNotificationConsumer> _logger;
    private readonly IConnectorService _connectorService;
    private readonly IPublishEndpoint _publishEndpoint;

    Yevhen Chubarov
    public StatusNotificationConsumer(ILogger<StatusNotificationConsumer> logger, IConnectorService connectorService, IPublishEndpoint publishEndpoint)
    {
        _logger = logger;
        _connectorService = connectorService;
        _publishEndpoint = publishEndpoint;
    }

    Yevhen Chubarov +1
    public async Task Consume(ConsumeContext<IntegrationOcppMessage<StatusNotificationRequest>> context)
    {
        _logger.LogInformation(message: "Processing status notification message...");

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

        var response = await _connectorService.ProcessStatusNotificationAsync(incomingRequest, chargePointId, context.CancellationToken);

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

        await _publishEndpoint.Publish(integrationMessage, context.CancellationToken);

        _logger.LogInformation(message: "Status notification message processed");
    }
}

```

Рисунок 4.9 – Структура частини Infrastructure (рисунок виконано самостійно)

Для створення споживача необхідно лише імплементувати інтерфейс `IConsumer` з необхідним типом повідомлення та метод інтерфейсу `Consume`,

записавши туди необхідну бізнес логіку. В даному випадку бізнес-логіка інкапсульована в сервісі IConnectorService, а в якості моделі використовується generic-тип `IntegrationOcppMessage<StatusNotificationRequest>`. Generic-модель `IntegrationOcppMessage<T>` відповідає схемі стандартного повідомлення OCPP [16] та складається з ідентифікатора типу повідомлення, ідентифікатора повідомлення для ідентифікації пари запита та відповіді, назви дії, що треба виконати, та корисного навантаження. У випадку, зображеному на рисунку 4.9, ідентифікатором типу повідомлення буде 2, тобто це є повідомлення від станції до системи, дією буде `StatusNotification`, а корисним навантаженням – дані, що містяться в `StatusNotificationRequest`. Також для розуміння того, з якою станцією відбувається комунікація та за яким протоколом, в нашому рішенні схема повідомлення була доповнена ідентифікатором станції та ідентифікатором протоколу (рис. 4.10).



```

44 usages 2 inheritors Yevhen Chubarov * 0+2 exposing APIs
public class IntegrationOcppMessage<TPayload>
{
    16 usages Yevhen Chubarov
    public IntegrationOcppMessage(Guid chargePointId, TPayload payload, string ocppMessageId, string ocppProtocol)
    {
        ChargePointId = chargePointId;
        Payload = payload;
        OcppMessageId = ocppMessageId;
        OcppProtocol = ocppProtocol;
    }

    19 usages
    public Guid ChargePointId { get; private set; }
    17 usages
    public TPayload Payload { get; set; }
    22 usages
    public string OcppMessageId { get; set; }
    9 usages
    public string OcppProtocol { get; set; }
}

```

Рисунок 4.10 – Структура системного повідомлення (рисунок виконано самостійно)

Після того, як повідомлення було оброблено, формується OCPP-відповідь та надсилається до черги RabbitMQ за допомогою IPublishEndpoint. Слід зазначити, що при роботі зі споживачами повідомлень, всі необхідні інструменти та сервіси для

обробки повідомлення можна зручно ін'єктувати за допомогою механізмів ін'єкції залежностей.

Таким чином, використання RabbitMQ забезпечує потужний і гнучкий механізм для обробки повідомлень, в той час як використання MassTransit спрощує розробку та підтримку кодової бази системи.

4.1.8 Використання gRPC

Як вже було зазначено в пунктах 3.1 та 4.1.1, для внутрішньої міжсервісної комунікації використовується протокол gRPC. Застосування такого підходу має наступні суттєві переваги:

- висока продуктивність: gRPC використовує HTTP/2 для транспортування даних, що забезпечує нижчі затримки та кращу продуктивність у порівнянні з HTTP/1.1. HTTP/2 дозволяє мультиплексувати кілька запитів через одне з'єднання, зменшуючи затримки і підвищуючи швидкість передачі;
- ефективна серіалізація: gRPC використовує протокол Protobuf для серіалізації даних, що є більш компактним і швидким у порівнянні з іншими форматами серіалізації, такими як JSON або XML. Це знижує об'єм даних, що передаються мережею, і підвищує продуктивність;
- сильна типізація: використання Protobuf забезпечує сильну типізацію даних, що дозволяє виявляти помилки на етапі компіляції, підвищуючи надійність і безпеку додатків;
- gRPC підтримує шифрування за допомогою TLS, що забезпечує безпечну передачу даних між клієнтом і сервером. Це критично важливо для систем, що обробляють конфіденційну інформацію;

Окремо варто відзначити високий рівень інтеграції з dotnet. В сучасних IDE слід лише описати protobuf-файли, після чого автоматично буде скомпільовано базовий код сервісів обміну даними та код моделей запитів та відповідей.

На рисунку 4.11 можна побачити приклад опису файлу protobuf.

```

syntax = "proto3";

import "google/protobuf/timestamp.proto";

option csharp_namespace = "EnergyConsumption.Grpc.Protos";

package energyconsumption;

service EnergyConsumptionSettingsGrpc {
  rpc GetEnergyConsumptionSettingsByDepotId (GetByDepotIdGrpcRequest) returns (DepotEnergyConsumptionSettingsGrpcResponse);
  rpc GetEnergyConsumptionSettingsByChargePointId (GetByChargePointIdGrpcRequest) returns (DepotEnergyConsumptionSettingsGrpcResponse);
}

message GetByDepotIdGrpcRequest {
  string depot_id = 1;
}

message GetByChargePointIdGrpcRequest {
  string charge_point_id = 1;
}

message DepotEnergyConsumptionSettingsGrpcResponse {
  string id = 1;
  string depot_id = 2;
  double depot_energy_limit = 3;
  google.protobuf.Timestamp valid_from = 4;
  google.protobuf.Timestamp valid_to = 5;
  repeated ChargePointEnergyConsumptionSettingsGrpcMessage charge_points_limits = 6;
  repeated EnergyConsumptionIntervalSettingsGrpcMessage intervals = 7;
  google.protobuf.Timestamp created_at = 8;
  google.protobuf.Timestamp updated_at = 9;
}

```

Рисунок 4.11 – Структура protobuf файлу (рисунок виконано самостійно)

В даному випадку на рисунку можна побачити опис сервісу для обміну інформацією про налаштування споживання енергії депо. З рисунку видно опис сервісу та його методів, а також опис моделей. Як було сказано вище, після опису protobuf файлу, автоматично створюється базові класи C#, що відповідають інформації з protobuf файлу (рис. 4.12).

```

public class EnergyConsumptionSettingsGrpcService : EnergyConsumptionSettingsGrpc.EnergyConsumptionSettingsGrpcBase
{
    private readonly IEnergyConsumptionSettingsService _energyConsumptionSettingsService;

    [Yevhen Chubarov]
    public EnergyConsumptionSettingsGrpcService(IEnergyConsumptionSettingsService energyConsumptionSettingsService)
    {
        _energyConsumptionSettingsService = energyConsumptionSettingsService;
    }

    [0+2 usages] [Yevhen Chubarov]
    public override async Task<DepotEnergyConsumptionSettingsGrpcResponse> GetEnergyConsumptionSettingsByDepotId(GetByDepotIdGrpcRequest request, ServerCallContext context)
    {
        var depotId : Guid = Guid.Parse(request.DpotId);

        var response = await _energyConsumptionSettingsService.GetByDepotIdAsync(depotId);

        return response?.ToGrpcResponse() ?? new DepotEnergyConsumptionSettingsGrpcResponse();
    }

    [0+2 usages] [Yevhen Chubarov]
    public override async Task<DepotEnergyConsumptionSettingsGrpcResponse> GetEnergyConsumptionSettingsByChargePointId(GetByChargePointIdGrpcRequest request, ServerCallContext context)
    {
        var chargePointId : Guid = Guid.Parse(request.ChargePointId);

        var response = await _energyConsumptionSettingsService.GetByChargingStationIdAsync(chargePointId);

        return response?.ToGrpcResponse() ?? new DepotEnergyConsumptionSettingsGrpcResponse();
    }
}

```

Рисунок 4.12 – Серверний gRPC сервіс (рисунок виконано самостійно)

Все що залишається розробникам, це перевантажити методи, заповнивши їх необхідної логікою. На рисунку 4.12 можна побачити gRPC-сервіс, що виступає в ролі сервера. При генерації коду з protobuf файлу також створюється gRPC-клієнт, використання якого можна побачити на рисунку 4.13.

```
public class EnergyConsumptionSettingsGrpcClientService
{
    private readonly EnergyConsumptionSettingsGrpc.EnergyConsumptionSettingsGrpcClient _energyConsumptionSettingsGrpcClient;

    Yevhen Chubarov
    public EnergyConsumptionSettingsGrpcClientService(EnergyConsumptionSettingsGrpc.EnergyConsumptionSettingsGrpcClient energyConsumptionSettin
    {
        _energyConsumptionSettingsGrpcClient = energyConsumptionSettingsGrpcClient;
    }

    Yevhen Chubarov
    public async Task<DepotEnergyConsumptionSettingsResponse?> GetEnergyConsumptionSettingsByDepotAsync(Guid depotId, CancellationToken cancell
    {
        var grpcRequest = new GetByDepotIdGrpcRequest
        {
            DepotId = depotId.ToString()
        };

        var grpcResponse = await _energyConsumptionSettingsGrpcClient.GetEnergyConsumptionSettingsByDepotIdAsync(grpcRequest, cancellationTokem

        if(CheckIfResponseIsEmpty(grpcResponse))
            return null;

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

Рисунок 4.13 – Застосування gRPC клієнта (рисунок виконано самостійно)

EnergyConsumptionSettingsGrpc.EnergyConsumptionSettingsGrpcClient в даному випадку і є тим самим автоматично згенерованим клієнтом.

Отже, використання gRPC в системі забезпечує високу продуктивність, надійність та безпеку, що робить його ідеальним вибором для розробки сучасних розподілених систем і мікросервісів.

4.1.9 Використання Redis

Протокол OCPP передбачає як відправку запитів від станції до центральної системи, так і навпаки, тобто від системи до станції. Наприклад, однією з функціональних можливостей протоколу OCPP є керування профілями заряджання з боку центральної системи, забезпечуючи механізм Smart Charging, або зміна

доступності роз'єма або всієї станції з боку центральної системи. Проблемою є те, що не завжди за відповіддю станції на запити від центральної системи можна ідентифікувати, наприклад, для якого роз'єма було встановлено або видалено профіль заряджання або для якого роз'єма було змінено доступність [16]. Єдиною інформацією, за якою можна ідентифікувати, для якого запиту було отримано відповідь, це ідентифікатор самого повідомлення, що передбачений протоколом ОСРР. Такий підхід змушує зберігати відправлені запити для того, щоб коректно обробити відповіді, що потім надходять. Для зберігання запитів можна використовувати різні підходи:

- зберігання в оперативній пам'яті. Хоча зберігання в оперативній пам'яті забезпечує високу швидкість доступу до збережених даних, але в той же час, якщо через якісь обставини мікросервіс виходить з ладу до того, як відповідь від станції буде оброблено, всі дані про запит пропадуть, що унеможлиблює подальшу обробку. Також проблемою є те, що при горизонтальному масштабуванні, тобто при збільшенні контейнерів або подів, якщо говорити в контексті використання Kubernetes (пункт 3.3), необхідно якимось чином забезпечити або реплікацію даних, або механізми, що будуть гарантувати, що відповідь буде оброблена тим самим подом (контейнером), який відправляв запит;
- зберігання в файловій системі. Дані можна зберігати в так званих *persistent volumes* контейнера, що дозволить зберігати дані навіть в разі перезапуску контейнера, але в той же час це знижує швидкість доступу до даних та не вирішує проблему реплікації при масштабуванні;
- зберігання в БД. Дані можна зберігати в БД, що вирішує проблему з реплікацією та цілісністю даних, а також отримати дані з БД можна доволі швидко, особливо якщо правильно налаштувати індекси. В такому випадку є декілька потенційних мінусів: по-перше, так як система працює з реляційною БД, то під кожен новий тип запиту потрібно створювати нову таблицю. Потенційно це може призвести до появи великої кількості

допоміжних таблиць, що буде ускладнювати розуміння схеми БД та її підтримку. Це частково можна вирішити тим, щоб зберігати дані рядком, попередньо серіалізувавши їх. По-друге, таке використання БД призведе до додаткових навантажень на неї;

- оптимальним рішенням стало використання кешу Redis. Redis зберігає дані у форматі «ключ – значення» та забезпечує високу швидкість доступу до даних, а також має вбудовані механізми реплікації для забезпечення цілісності даних в разі горизонтального масштабування або відмови одного з екземплярів Redis. Таким чином, це позбавляє від необхідності створення додаткових таблиць в БД та від додаткового навантаження на БД, а також позбавляє від необхідності розробки механізмів реплікації даних для контейнерів в разі горизонтального масштабування. Окремо варто відзначити гарну інтеграцію з .NET та можливість легко налаштовувати час життя кешу.

Таким чином, перед відправкою запиту від центральної системи до станції, в разі необхідності дані зберігаються в кеш, де ключем є ідентифікатор повідомлення. Так як відповідь має такий самий ідентифікатор, як і запит, то при отриманні відповіді з кешу зчитується інформація про запит і далі обробляється відповідно до правил бізнес-логіки.

4.2 Використання Kubernetes

Як було зазначено в пункті 3.3, для розгортання та оркестрації мікросервісів буде застосовано Kubernetes. Застосування такого підходу забезпечує такі переваги, як:

- автоматизована оркестрація: Kubernetes автоматизує розгортання, управління, масштабування та обслуговування контейнеризованих додатків, що значно спрощує керування складними системами та їх підтримку;

- масштабованість: Kubernetes дозволяє автоматично масштабувати додатки на основі навантаження. Використовуючи ресурси кластеру, Kubernetes може динамічно додавати або видаляти поди (контейнери), щоб забезпечити оптимальну продуктивність;
- висока доступність: Завдяки механізмам відновлення після збоїв, Kubernetes забезпечує високу доступність застосунків. Якщо один з вузлів виходить з ладу, Kubernetes автоматично перезапускає контейнери на інших вузлах кластеру;
- самовідновлення: Kubernetes постійно контролює стан контейнерів і, якщо виявляє проблему, автоматично перезапускає або переміщує контейнер, щоб забезпечити стабільну роботу системи.
- ролінгові оновлення: Kubernetes підтримує ролінгові оновлення, що дозволяє оновлювати додатки без простоїв. Таким чином, при оновленні образу контейнера, Kubernetes поступово замінюватиме старі версії новими, забезпечуючи безперервну роботу сервісів;
- управління конфігурацією і секретами: Kubernetes дозволяє безпечно зберігати та управляти конфігураціями та секретами, такими як паролі, ключі API і конфігураційні файли. Це забезпечує безпечний доступ до конфіденційної інформації;

Слід зазначити, що для більш зручної роботи з Kubernetes та легшої підтримки кодової бази, а також версіонування завдяки системам контролю версій (наприклад, git), для розгортання ресурсів буде використано декларативний підхід на основі Kubernetes маніфестів.

Для роботи з мікросервісами будуть використовуватись такі види ресурсів Kubernetes, як services, deployments та statefulsets, про що буде більш детально описано далі. Для безпечного зберігання секретів будуть використані Kubernetes Secrets, а для конфігурацій (наприклад, адреси хоста RabbitMQ або Redis) – Config Maps.

4.2.1 Загальний підхід

Кожен мікросервіс в Kubernetes розгортається на одному або декількох подах. Под (Pod) – це мінімальна одиниця розгортання в Kubernetes. В більшості випадків, под являє собою один контейнер, але для більш складних випадків, в поді можуть бути декілька контейнерів. Контейнери в рамках одного пода мають спільні ресурси: CPU, пам'ять та файлове сховище.

Описати те, як будуть розгортатись поди, можна за допомогою deployments або stateful sets маніфестів. Вибір підходу залежить від того, яким є застосунок, що необхідно розгорнути: stateful (зі збереженням стану) чи stateless (без збереження стану). Головна відмінність в тому, що stateful застосункам необхідно зберігати свій стан. Stateful програми дозволяють користувачам зберігати, записувати та повертатися до вже встановленої інформації та процесів через Інтернет. У додатках із збереженням стану сервер відстежує стан кожного сеансу користувача та зберігає інформацію про взаємодії користувача та попередні запити. До них можна повертатися знову і знову, наприклад до онлайн-банкінгу чи електронної пошти. Вони виконуються в контексті попередніх транзакцій, і на поточну транзакцію може вплинути те, що сталося під час попередніх транзакцій. З цих причин stateful програми використовують ті самі сервери щоразу, коли обробляють запит від користувача. Тобто stateful сервіси вимагають наявності постійного сховища, доменного імені, ip та ін. Такі сервіси важче масштабувати, бо необхідно забезпечити механізми реплікації. Прикладами stateful застосунків є, наприклад, бази даних. На відміну від stateful, stateless сервіси не вимагають зберігання стану і є більш масштабованими, тобто будь який екземпляр такого застосунку може обробити запит користувача. Прикладом stateless сервісів є веб-сервери. В контексті системи, що розробляється, stateful сервісами є Redis та RabbitMQ, а stateless – інші мікросервіси, що являють собою веб-застосунки.

4.2.1 Розгортання stateless мікросервісів

Як було зазначено в попередньому розділі, stateless мікросервіси можна розгорнути за допомогою deployment маніфесту, в якому описано, який образ використовувати, скільки реплік необхідно розгорнути, які команди використовувати під час розгортання, скільки ресурсів можна виділити і т.д. При розгортанні подів через deployment, кожен под має унікальне ім'я. Якщо под виходить з ладу, то перестворюється вже з новим ім'ям. Працювати з ір-адресами таких подів напрямку важко, тож для того, щоб звернутись до застосунку, який працює на одному або декількох подах та був розгорнутий через deployments, існує такий інструмент Kubernetes, як сервіс – це метод для надання доступу до мережевої програми, яка працює як один або кілька подів у кластері. В додатку Г можна побачити приклад deployment та service маніфестів для розгортання мікросервісу для роботи з депо.

В маніфесті deployment можна побачити описах таких аспектів, як: ім'я deployment, мітки для ідентифікації, які контейнери будуть розгорнуті, як вони будуть налаштовані та скільки ресурсів використовувати. В даному випадку буде розгорнуто один под з одним контейнером, що буде використовувати образ yevhen0825/depotsapi. Оскільки зазначено imagePullPolicy: IfNotPresent, то образ буде завантажено тільки в тому випадку, якщо його ще не було завантажено локально. Контейнер працюватиме на порті 8080 із заданими змінними середовища ASPNETCORE_ENVIRONMENT, ASPNETCORE_HTTP_PORTS та ConnectionStrings__DefaultConnection. Значення останньої змінної дістається з секрета mssql-db-secret за ключем mssql-main-db-connection-string.

В маніфесті service можна побачити інформацію про селектор, що визначає поди, на які буде направлений трафік, на основі міток, що були зазначені в deployment, порт самого service в мережі Kubernetes та з яким портом контейнера цей порт пов'язаний. В даному випадку трафік, що прямуватиме за адресою http://depotsapi-service:8080 буде перенаправлений на контейнери подів з мітками app: depotsapi на порт 8080.

Також важливо зазначити, що ці контейнери не налаштовані на доступ до них ззовні. Відповідно до архітектури системи, ззовні можна підключатись лише до через API Gateway до API, через SignalR сервіс до SignalR хабу, а також станції будуть підключатись до сервісу вебсокетів.

4.2.2 Розгортання RabbitMQ та використання StatefulSet

RabbitMQ є основним механізмом асинхронної комунікації між мікросервісом обробки підключень зарядних станцій та іншими мікросервісами. Таким чином, надзвичайно важливо забезпечити відмовостійкість та надійність роботи цього сервісу. Для досягнення цієї мети, необхідно розгорнути не один под для RabbitMQ, а цілий кластер, а також налаштувати реплікацію даних.

RabbitMQ вимагає використання Stateful Set для розгортання кластера RabbitMQ у Kubernetes. Stateful Set гарантує, що вузли RabbitMQ розгортаються в порядку, по одному. Це дозволяє уникнути потенційної гонки peer discovery [17] під час розгортання багатовузлового кластера RabbitMQ.

Існують інші, не менш важливі причини для використання Stateful Set замість Deployment: фіксована ідентифікація в мережі, прості мережеві ідентифікатори, постійне сховище та можливість виконувати впорядковані поточні оновлення (rolling updates). Саме завдяки тому, що в stateful set кожен под має власне FQDN в рамках мережі Kubernetes, поди можуть ідентифікувати один одного та сформувати кластер.

Маніфест Stateful Set може посилатися на Kubernetes сервіс, який надає подам в Stateful Set їх мережеву ідентифікацію. Це потрібно RabbitMQ для кластеризації, і, як зазначено в документації Kubernetes, мусить бути створено перед створенням stateful set.

В додатку Д можна побачити stateful set та service маніфести, що забезпечують розгортання кластера RabbitMQ.

З маніфесту stateful set можна побачити, що для пода RabbitMQ використовуються три контейнери.

Контейнер `config` є ініціалізаційним контейнером (`initContainer`), який виконується перед запуском основного контейнера `RabbitMQ`. Його основна мета – підготувати конфігураційні файли для `RabbitMQ`. Він використовує образ `busybox`, який є легким базовим образом для виконання простих команд.

Після запуску контейнер виконує наступні дії:

- копіює файл `rabbitmq.conf` з тимчасової директорії `/tmp/config/` до `/config/`, яка буде використовуватись основним контейнером `RabbitMQ`. В цьому файлі описані налаштування хоста та адреси подів, а також те, який плагін необхідно використати подам для ідентифікації один одного, щоб сформувати кластер;
- виконує команду `ls -l /config/`, яка виводить список файлів у директорії `/config/`. Це допомагає у відлагодженні, показуючи, що файл успішно скопійовано;
- копіює файл `enabled_plugins` з тимчасової директорії `/tmp/config/` до `/etc/rabbitmq/`, яка використовується для зберігання налаштувань плагінів `RabbitMQ`. Цей файл з налаштуванням плагінів для `RabbitMQ`, в якому описано, що необхідно використовувати плагін для реплікації, плагін для панелі моніторингу та плагін для `Kubernetes Service Discovery`, який допомагає подам ідентифікувати один одного для формування кластера.

Контейнер монтує три томи:

- `config`, який містить конфігураційні файли і монтований у `/tmp/config/`;
- `config-file`, який використовується для зберігання кінцевих конфігураційних файлів і монтований у `/config/`;
- `plugins-file`, який містить файли плагінів і монтований у `/etc/rabbitmq/`.

Контейнер `set-erlang-cookie` також є ініціалізаційним контейнером. Його завдання – налаштувати `Erlang cookie`, необхідний для аутентифікації подів кластера `RabbitMQ`. Він використовує образ `busybox` для виконання команд оболонки.

Після запуску контейнер виконує наступні дії:

- виконує команду, яка записує значення змінної середовища `RABBITMQ_ERLANG_COOKIE` до файлу `/var/lib/rabbitmq/.erlang.cookie`. Ця змінна отримується з секрету Kubernetes, що забезпечує безпечне зберігання конфіденційної інформації;
- змінює права доступу до файлу `.erlang.cookie`, встановлюючи їх на 400, що дозволяє лише читання файлу власником. Це забезпечує безпеку файлу `cookie`.

Контейнер монтує один том `data`, який використовується для зберігання даних RabbitMQ і монтований у `/var/lib/rabbitmq`.

Після завершення роботи ініціалізаційних контейнерів запускається основний контейнер `rabbitmq`, який використовує попередньо підготовлені конфігураційні файли та налаштування. Це забезпечує належну конфігурацію і кластеризацію RabbitMQ у Kubernetes.

Разом зі `stateful set` створюється `service`, де важливо звернути увагу на наступні речі:

- **Headless Service.** Використання `clusterIP: None` створює `headless service`. Це означає, що сервіс не отримує власної IP-адреси, але забезпечує DNS-імена для кожного поду. Це важливо для `stateful` застосунків, таких як RabbitMQ, оскільки кожен под у `StatefulSet` має стабільний DNS-ідентифікатор. Наприклад, кожен под може бути доступний за іменем `rabbitmq-0.rabbitmq`, `rabbitmq-1.rabbitmq`, і так далі;
- сервіс відкриває два порти: 4369 для служби виявлення (`discovery`) і 5672 для AMQP. Це дозволяє клієнтам підключатися до RabbitMQ через ці порти;
- `Discovery` порт використовується RabbitMQ для знаходження і взаємодії з іншими вузлами RabbitMQ у кластері.
- AMQP порт використовується для обміну повідомленнями між клієнтами і RabbitMQ;

– сервіс використовує селектор app: rabbitmq для визначення, які поди він обслуговує. Це означає, що сервіс буде автоматично знаходити і маршрутизувати трафік до всіх подів, які мають цю мітку. У даному випадку, це поди, створені StatefulSet для RabbitMQ.

На рисунку можна 4.14 можна побачити панель керування RabbitMQ з кластером з трьох екземплярів.

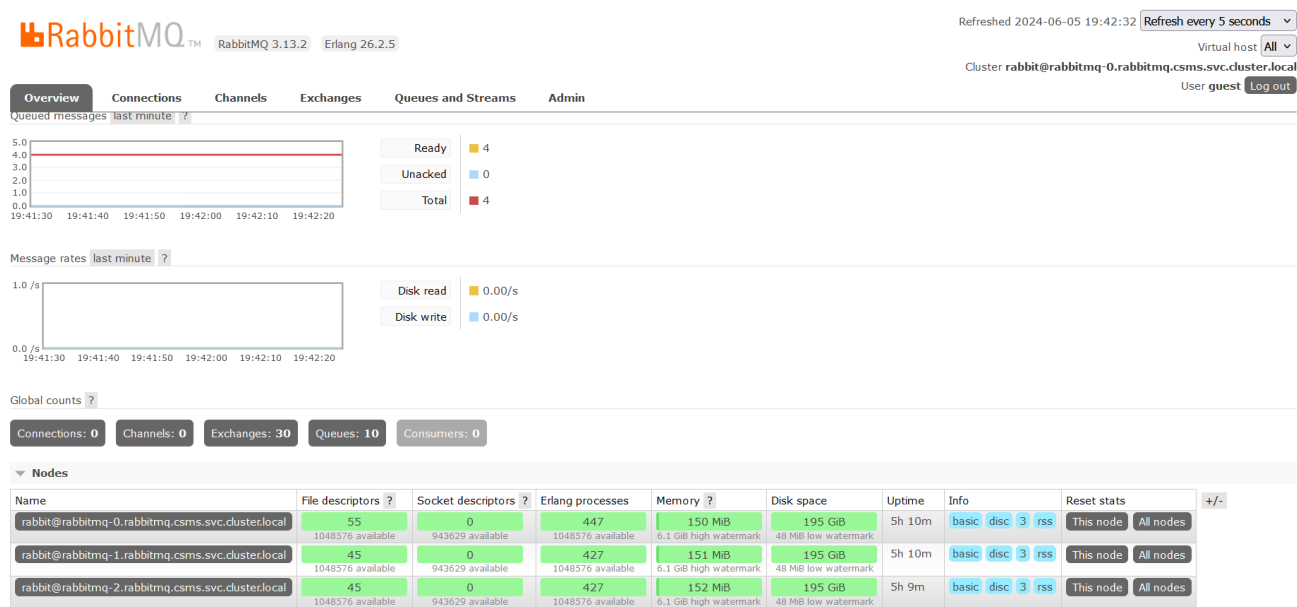


Рисунок 4.14 – Розгорнутий кластер RabbitMQ (рисунок виконано самостійно)

Отже, завдяки використанню stateful set, контейнерів ініціалізації та headless service, вдається автоматично створювати, налаштовувати та розгортати кластер RabbitMQ, що складається з трьох екземплярів. За необхідності, Kubernetes дозволяє створити окремі маніфести для налаштування горизонтального масштабування.

4.2.3 Розгортання Redis

Як і у випадку з RabbitMQ, Redis також є stateful сервісом, який необхідно розгорнути таким самим підходом – за допомогою stateful set та headless service. Однак на відміну від RabbitMQ, де поди самі формують кластер, для формування

кластера Redis, необхідно запускати додатковий sentinel-сервіс також у форматі stateful set. В зону відповідальності sentinel входить наступне:

- моніторинг. sentinel постійно перевіряє, чи працюють екземпляри належним чином;
- сповіщення: sentinel може повідомити системного адміністратора або інші комп'ютерні програми через API про те, що щось не так з одним із контрольованих екземплярів Redis;
- автоматичне перемикавання після відмови. Якщо головний сервер не працює належним чином, Sentinel може розпочати процес відновлення після відмови, у якому репліка стає головною, інші додаткові репліки переналаштовуються для використання нового головного сервера, а програми, які використовують сервер Redis, отримують інформацію про нову адресу для використання при підключенні;
- постачальник конфігурації: sentinel діє як джерело повноважень для виявлення служб клієнта: клієнти підключаються до sentinel, щоб запитати адресу поточного майстер-екземпляра Redis, відповідального за дану послугу. Якщо станеться відмова, sentinel повідомить про нову адресу.

В додатку Е можна побачити маніфести розгортання кластеру sentinel з трьох реплік. Розгортання кластеру було обрано для підвищення надійності роботи sentinel.

Як і у випадку з RabbitMQ, спочатку запускається контейнер ініціалізації для налаштування конфігураційного файлу Redis Sentinel перед запуском основного контейнера. Цей контейнер виконує наступні задачі:

Контейнер config виконує наступні дії:

- встановлює значення змінної REDIS_PASSWORD як пароль для доступу до Redis;
- визначає список Redis вузлів у кластері (redis-0, redis-1, redis-2);
- в циклі перебирає кожен вузол для знаходження master-вузла, використовуючи команду redis-cli info replication;

- після знаходження master-вузла, записує відповідну конфігурацію у файл `/tmp/master`;
- створює конфігураційний файл `sentinel.conf` у директорії `/etc/redis/`, додаючи параметри для налаштування Sentinel, включаючи інформацію про майстер-вузол і аутентифікацію;
- виводить вміст створеного конфігураційного файлу для перевірки.

Також контейнер монтує том `redis-config`, щоб зберігати конфігураційні файли у директорії `/etc/redis/`.

Після завершення роботи ініціалізаційного контейнера запускається основний контейнер `sentinel`, який виконує Redis Sentinel на основі підготовленого конфігураційного файлу. Таким чином, основний контейнер:

- запускає команду `redis-sentinel`, яка запускає Redis Sentinel з конфігураційним файлом `/etc/redis/sentinel.conf`;
- відкриває порт 5000 для доступу до Sentinel;
- монтує томи `redis-config` для доступу до конфігураційних файлів і `data` для зберігання даних.

Також, так само як і з RabbitMQ, разом зі `stateful set` також створюється `headless service` для Redis Sentinel, що забезпечує доступ до нього у кластері.

Після того, як `sentinel` для `redis` розгорнуто, можна розгортати власне кластер `redis`. В додатку Ж можна побачити код маніфестів для розгортання кластеру. Як і в попередніх розгортаннях, в цьому випадку використовується спочатку контейнер ініціалізації, а потім вже основний.

Контейнер `config` виконує наступні дії:

- копіює конфігураційний файл `redis.conf` з тимчасової директорії `/tmp/redis/` до `/etc/redis/`;
- визначає повне доменне ім'я (FDQN) master-вузла за допомогою команди `hostname -f`;
- якщо Sentinel не доступний (`redis-cli -h sentinel -p 5000 ping` не повертає «PONG»), контейнер перевіряє, чи є він сам master-вузлом (`redis-0`). Якщо

так, він не змінює конфігурацію. В іншому випадку, він додає налаштування `replicaof` у конфігураційний файл `redis.conf`, вказуючи на `master`-вузол;

- якщо Sentinel доступний, контейнер використовує команду `redis-cli` для визначення адреси `master`-вузла за допомогою Sentinel. Потім він додає налаштування `replicaof` у конфігураційний файл `redis.conf`, вказуючи на знайдений `master`-вузол.

Контейнер монтує два томи:

- `redis-config`, який містить конфігураційні файли, монтований у `/etc/redis/`;
- `config`, який містить початкові конфігураційні файли, монтований у `/tmp/redis/`.

Після завершення роботи ініціалізаційного контейнера запускається основний контейнер `redis`, який виконує Redis з використанням підготовленого конфігураційного файлу. Основний контейнер:

- запускає команду `redis-server`, яка запускає Redis з конфігураційним файлом `/etc/redis/redis.conf`;
- відкриває порт 6379 для доступу до Redis;
- монтує два томи: `data`, який використовується для зберігання даних Redis і монтований у `/data`, а також `redis-config`, який містить конфігураційні файли і монтований у `/etc/redis/`.

Таким чином, завдяки `stateful set` та `headless service`, було сформовано кластери Redis та Sentinel для Redis для забезпечення високої доступності та відмовостійкості системи.

4.3 Налаштування CI/CD процесів

CI/CD (Continuous Integration/Continuous Delivery або Continuous Deployment)

- це підходи до розробки програмного забезпечення, які допомагають автоматизувати та прискорити процес розробки, тестування та розгортання. Ось основні аспекти кожного компонента:

Continuous Integration (CI) (або безперервна інтеграція) – це процес частого злиття коду розробників у основний репозиторій, з метою виявлення та виправлення помилок на ранніх етапах розробки ПЗ. Для забезпечення CI автоматизовані зборки та тести запускаються щоразу, коли новий код додається до репозиторію. Це дозволяє швидко виявляти помилки та запобігати їх накопиченню.

Continuous Delivery (CD) (або безперервна доставка) – це процес, що автоматично готує код для випуску в production. Основною метою є зменшення часу між написанням коду та його випуском у продакшн. Цей процес полягає в автоматизації тестування, зборки та пакування коду для подальшого випуску.

Continuous Deployment (CD) (або безперервне розгортання) – це повністю автоматизований процес розгортання коду, що забезпечує швидку доставку нового функціоналу та виправлень клієнтам. Цей процес імплементується за допомогою пайплайнів, що автоматично розгортають код без ручного втручання.

Застосування CI/CD в системі, що розробляється, може надати такі переваги, як:

- швидке виявлення та виправлення помилок завдяки автоматизації тестування;
- збільшення швидкості випуску завдяки автоматизації процесів збірки, пакування та публікації.
- підвищення якості коду завдяки постійному автоматичному тестуванню;
- зменшення людських помилок: автоматизація рутинних завдань зменшує ймовірність помилок, обумовлених людським фактором.

4.3.1 Застосування Azure DevOps

Azure DevOps – це набір хмарних сервісів від Microsoft, які підтримують весь життєвий цикл розробки програмного забезпечення, включаючи планування, розробку, тестування та розгортання.

В контексті поточної роботи, для налаштування CI/CD процесів, будуть застосовані Azure DevOps пайплайни. Пайплайн – це послідовність

автоматизованих кроків, які виконуються для побудови, тестування та розгортання додатків. Пайплайни є невід’ємною частиною процесів CI/CD і допомагають автоматизувати рутинні завдання, забезпечуючи більш швидке і якісне розгортання. В Azure DevOps пайплайн описується за допомогою мови розмітки YAML.

Для поточної системи є дві основні речі, які можна автоматизувати за допомогою CI/CD пайплайну:

- побудова, тестування, створення Docker-образів та завантаження їх в репозиторії образів реєстру контейнерів;
- автоматичне розгортання образів в Kubernetes.

Перед тим, як створювати пайплайни, було розгорнуто необхідну інфраструктуру – Azure Container Registry (ACR) – приватний реєстр для Docker образів, а також Azure Kubernetes Service (AKS) – кластер Kubernetes в хмарі Azure.

В додатку II можна побачити код CI/CD пайплайну.

Перш за все, слід зазначити, що тригер пайплайну налаштований на гілку main, тобто коли в гілку main додаються нові зміни, пайплайн буде автоматично запущено.

Далі можна побачити перелік вхідних параметрів пайплайну:

- складний параметр, що являє собою перелік об’єктів з інформацією про те, за яким шляхом треба скопіювати код, в який образ (image) та за яким шляхом знаходиться відповідний маніфест;
- force build параметр, що сигналізує про те, чи треба робити побудову образу та публікацію, якщо код мікросервісу не було змінено.

Хоча перший параметр здається складним, він дозволяє в разі необхідності налаштувати побудову та публікацію лише окремих мікросервісів.

Після параметрів можна побачити перелік необхідних змінних, таких як назва ACR, тип образу агента (агент – віртуальна машина в хмарі, на якій запускається пайплайн. В даному випадку використовуються віртуальні машини з

ОС ubuntu), контекст виконання Docker та ін. Також поряд знаходиться визначення пулу агентів.

Далі після визначення змінних та пулу агентів, можна побачити головні компоненти пайплайну: stages (етапи), jobs (роботи) та steps (кроки). Крок – це найменша одиниця виконання пайплайну, що виконує якусь одну дію, наприклад, виконання скрипту, компіляція коду і т.д. Робота – це набір кроків. Роботи виконуються в рамках одного етапу на одному агенті. Етап – це набір робіт. І роботи, і етапи, якщо вони не взаємопов’язані, можуть виконуватись паралельно, але роботи можуть виконуватись паралельно в рамках одного агента, а етапи можуть виконуватись паралельно на різних агентах.

Отже, повертаючись до CI/CD пайплайну системи, що розробляється, можна побачити, що в циклі запускається пайплайн під кожен об’єкт з першого вхідного параметру. Таким чином, вдалось досягти того, що для кожного мікросервісу запускається окремий етап і при додаванні нових мікросервісів слід лише зазначити шляхи до нього в першому параметрі.

Етап для кожного мікросервісу складається з наступних робіт:

- визначення змін – в ході цієї роботи виконуються скрипти, що аналізують останні зміни в git-репозиторії та перевіряють, чи було змінено код мікросервісу або код загальних компонентів;
- побудова образу та відправка в ACR. Слід зазначити, що за замовчуванням, якщо попередня робота не виявила змін в коді мікросервісу або коді загальних компонентів, то цю роботу буде пропущено;
- розгортання маніфесту в AKS. Ця робота займається тим, що застосовує Kubernetes маніфест для відповідного мікросервісу. Також ця робота залежить від попередньої, тобто якщо новий образ не було сформовано, то перепублікацію проведено не буде. Отже, завдяки механізму виявлення змін в коді, що виконується в першій роботі, можна значно час виконання пайплайну та публікувати тільки ті мікросервіси, які дійсно зазнали змін. Якщо трапляється випадок, коли незалежно від змін необхідно, щоб

пайплайн відпрацював (наприклад, тестування оновлення пайплайну), то за допомогою параметру `force build` примусово буде виконано побудову та публікацію образів в Container Registry, а також розгортання в Azure Kubernetes, але при цьому в кожному етапі роботу по визначенню змін виконано не буде, бо в разі примусового виконання немає сенсу проводити аналіз на зміни в коді.

На рисунку 4.15 можна побачити роботи, які було виконані та пропущені в ході виконання пайплайну. Можна помітити, що завдяки механізму виявлення змін в коді, роботи по побудові образів та розгортанню мікросервісів для роботи з депо були пропущені, тому змін в їх коді та в коді загальних компонентів не відбулося.

The screenshot shows the 'Summary' tab of a pipeline run. It indicates the run was manually triggered by Yevhen Chubarov. The repository is 'Nexus20/ChargingStation.Backend' and the commit is '0ed67b3e'. The run started at 17:31 and took 8 minutes. There are 0 work items and 0 artifacts. The 'Jobs' tab is active, showing a list of jobs with their status and duration.

Name	Status	Stage	Duration
✓ Detecting changes for depotsapi	Success	Build and push ...	4s
⌚ Build and push image depotsapi into ACR	Skipped	Build and push ...	
⌚ Deploy depotsapi	Skipped	Build and push ...	
✓ Detecting changes for depotsgrpc	Success	Build and push ...	4s
⌚ Build and push image depotsgrpc into ACR	Skipped	Build and push ...	
⌚ Deploy depotsgrpc	Skipped	Build and push ...	
✓ Detecting changes for aggregator	Success	Build and push ...	4s

Рисунок 4.15 – Виконання пайплайну з урахуванням механізму аналізу змін
(рисунок виконано самостійно)

На рисунку 4.16 можна побачити виконання пайплайну з включеним `force build`.

Manually run by  Yevhen Chubarov

Repository and version	Time started and elapsed	Related
 Nexus20/ChargingStation.Backend	 Today at 18:28	 0 work items
 feature/kubernetes-continue  f1bbe9e4	 5m 46s	 0 artifacts

Warnings 3

-  No data was written into the file /home/vsts/work/_temp/task_outputs/build_1717687803090.txt
Build and push depotsapi • Build and push image depotsapi into ACR • Build and push an image to container registry
-  No data was written into the file /home/vsts/work/_temp/task_outputs/build_1717687922038.txt
Build and push depotsgrpc • Build and push image depotsgrpc into ACR • Build and push an image to container registry
-  No data was written into the file /home/vsts/work/_temp/task_outputs/build_1717688068049.txt
Build and push aggregator • Build and push image aggregator into ACR • Build and push an image to container registry

Stages Jobs

Name	Status	Stage	Duration
 Build and push image depotsapi into ACR	Success	Build and push ...	 1m 17s
 Deploy depotsapi	Success	Build and push ...	 9s
 Build and push image depotsgrpc into ACR	Success	Build and push ...	 1m 16s
 Deploy depotsgrpc	Success	Build and push ...	 12s
 Build and push image aggregator into ACR	Running	Build and push ...	 1m 44s

Рисунок 4.16 – Виконання пайплайну з увімкненим force build (рисунок виконано самостійно)

Як видно з рисунку, йде виконання робіт по побудові образів та розгортанню їх в Azure Kubernetes.

Результати виконання пайплайну можна побачити на рисунках 4.17 та 4.18. На рисунку 4.17 показано розгорнутий ACR з переліком опублікованих Docker-образів, а на рисунку 4.18 – розгорнутий AKS з розгорнутими подами мікросервісів.

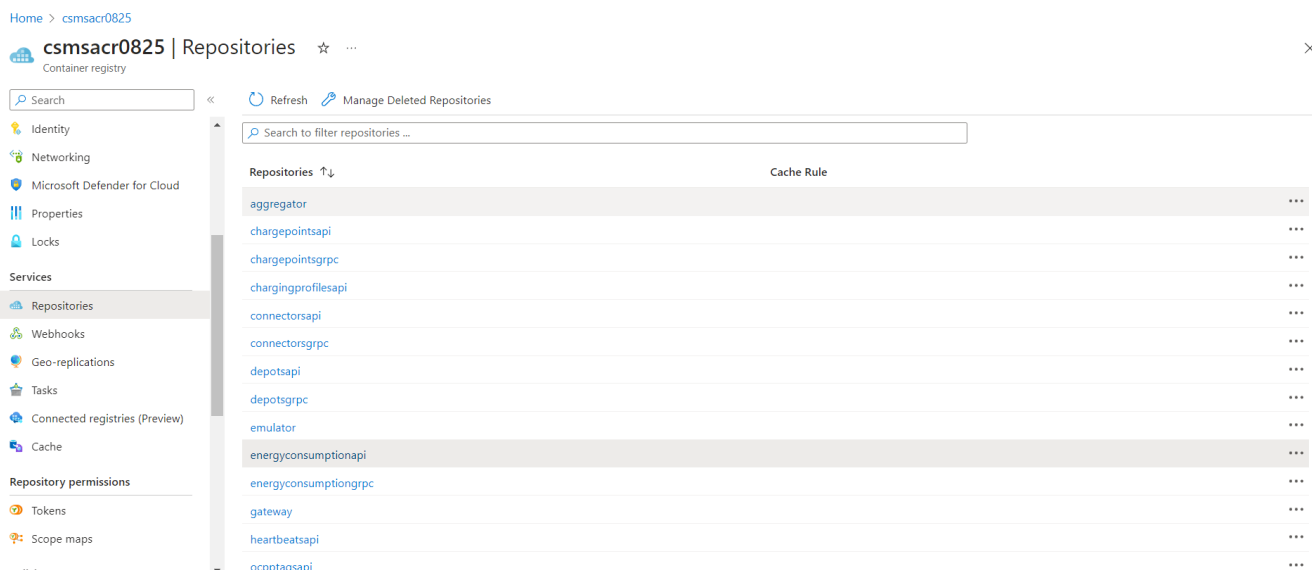


Рисунок 4.17 – Azure Container Registry з опублікованими Docker-образами
(рисунок виконано самостійно)

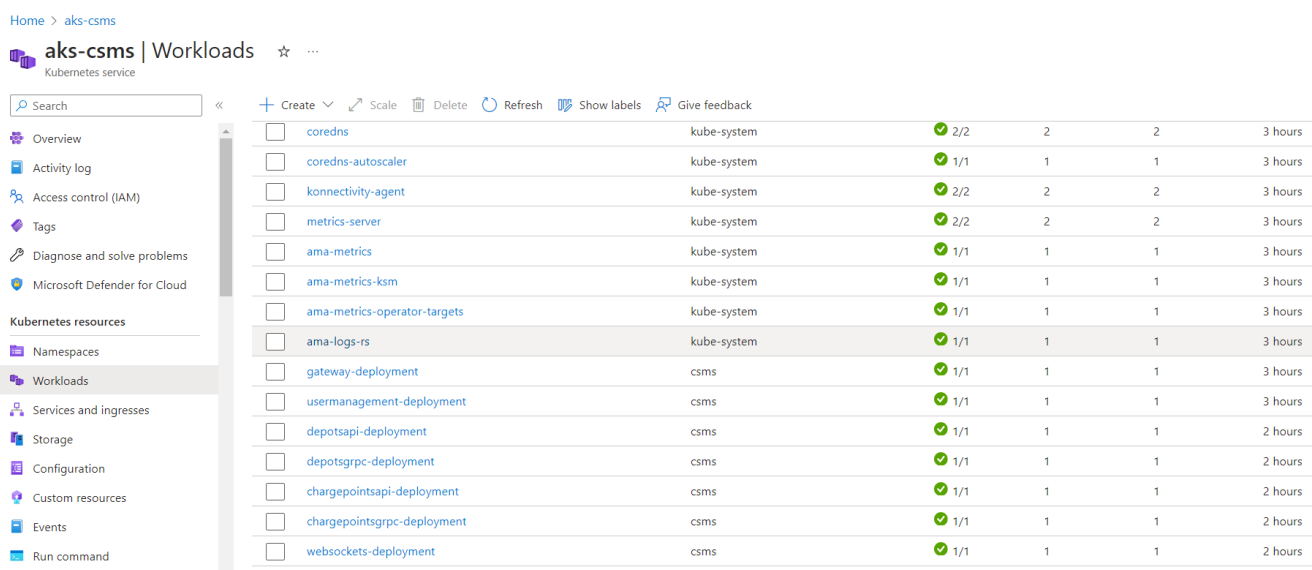


Рисунок 4.18 – Azure Kubernetes Service з розгорнутими подами мікросервісів
(рисунок виконано самостійно)

Отже, завдяки використанню Azure DevOps Pipelines вдалося автоматизувати процес побудови та публікації Docker-образів в Azure Container Registry та розгортання образів в Azure Kubernetes Service. Працюючу клієнтську частину, що комунікує з мікросервісами, розгорнутими в AKS, можна побачити в додатку К.

5 ВПРОВАДЖЕННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

На основі результатів аналізу та розробки програмної системи було створено публікацію для конференції «Інформаційні інтелектуальні системи» XXVIII Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті», яка була проведена 16 – 18 квітня 2024 р. у місті Харків. Назва публікації «Проектування серверної частини системи керування та моніторингу зарядними станціями» (додаток В).

Робота була присвячена як проектуванню та розробці серверної частини системи керування та моніторингу зарядних станцій для електромобілів, так і налаштуванню CI/CD процесів для автоматизації доставки рішення користувачам. Було розглянуто такі аспекти, як використання протоколу Open Charge Point Protocol (OCPP) для забезпечення сумісності між різними зарядними станціями; застосування архітектури мікросервісів для забезпечення модульності, масштабованості та надійності; використання RabbitMQ в якості брокера повідомлень для асинхронної комунікації; застосування Docker і Kubernetes для полегшення розгортання і оркестрації ресурсів, а також підвищення загальної доступності системи; застосування Azure DevOps пайплайнів для налаштування CI/CD процесів. Також було описано загальні перспективи розвитку системи.

Крім цього, розробник клієнтської частини системи Горкун Д.О. та тестувальник Налескіна Т.С. опублікували інші тези 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». В цих тезах було розглянуто ключові аспекти проектування клієнтської частини системи, зокрема використання Onion архітектури для забезпечення модульності та зручності у підтримці коду, а також було детально описано структуру додатку, підходи до реалізації основних функцій та забезпечення продуктивності системи [18].

ВИСНОВКИ

В результаті виконаної роботи було проведено ретельний аналіз предметної області, включно зі статистичними даними, учасниками ринку та їх потребами, а також поточними рішеннями та технологіями. Також в ході аналізу програмної системи було визначено основні проблеми, що необхідно вирішити, та розроблено чіткі функціональні та нефункціональні вимоги, до яких відносяться: керування депо та станціями, резервування станцій, аналітика та звітність, моніторинг стану депо в реальному часі, керування користувачами та авторизаційними тегами, безпека даних і інтеграція зі сторонніми системами.

Було проведено аналіз технологій та можливість їх застосування в контексті обмеженого бюджету, після чого було спроектовано базу даних, архітектуру серверної частини програмної системи моніторингу зарядних станцій електромобілів, а також CI/CD процеси.

Було визначено, що в основі серверної частини буде знаходитись мікросервісна архітектура, де всі сервіси будуть звертатись до однієї БД (окрім сервісу, що обробляє «сердцебиття» зарядних станцій), але мати доступ лише до своєї частини даних, таким реалізуючи підхід «Shared Instance». Сервіси будуть реалізовані на платформі ASP.NET, міжсервісна комунікація буде реалізована за допомогою брокера RabbitMQ та протоколу gRPC. Для комунікації зі станціями будуть застосовані стандартні протоколи OCPP та WebSocket. Щодо CI/CD процесів, то вони будуть налагоджені за допомогою Docker, Azure Kubernetes, Azure Container Registry та Azure DevOps Pipelines.

Наступним кроком було розроблено архітектуру, імplementовано бізнес-логіку та налаштовано CI/CD процесі відповідно до підходів, що було зазначено на етапі проєктування.

Таким чином, завдання кваліфікаційної роботи було виконано в повному обсязі – було розроблено серверну частину системи моніторингу зарядних станцій для електромобілів та налаштовано процеси CI/CD. Завдяки використаним підходам, розроблена система є масштабованою та відмовостійкою.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1. Тенденції на ринку зарядних станцій для електромобілів. Faraday. URL: <https://faraday.in.ua/bez-rubriki/yaki-zaraz-tendentsiyi-na-rynku-zaryadnyh-stantsij-dlya-elektromobiliv-u-yevropi/> (дата звернення: 07.06.2024).
2. Стан інфраструктури зарядних станцій електромобілів в Україні. Офіційний сайт YASNO. URL: https://yasno.com.ua/news/all_news/How_to_make_money_at_charging_stations_for_electric_cars (дата звернення: 07.06.2024).
3. Charge management software – bp pulse. Bp pulse fleet. URL: <https://bppulsefleet.com/fleet/products/charge-management-software/> (дата звернення: 14.05.2024).
4. JET Charge Connect by JET Charge Pty Ltd. AppAdvice. URL: <https://appadvice.com/app/jet-charge-connect/1568030784> (дата звернення: 14.05.2024).
5. Smart EV Charging App. Ev.energy. URL: <https://www.ev.energy/> (дата звернення: 14.05.2024).
6. Чубаров Є. Е., Рубель Д. А. Проєктування серверної частини системи керування та моніторингу зарядними станціями. 28-й Міжнародний молодіжний форум «Радіoeлектроніка та молодь у ХХІ столітті» : Зб. матеріалів форуму, м. Харків, 8–9 трав. 2024 р. / Наук. керівник Н. Є. Русакова. Харків, 2026. С. 306–308.
7. What are microservices?. microservices.io. URL: <https://microservices.io/> (дата звернення: 10.05.2024).
8. RabbitMQ Documentation. RabbitMQ. URL: <https://www.rabbitmq.com/docs> (дата звернення: 10.05.2024).
9. Open charge point protocol – Open Charge Alliance. Open Charge Alliance. URL: <https://openchargealliance.org/protocols/open-charge-pointprotocol/> (дата звернення: 10.05.2024).
10. gRPC Documentation. gRPC. URL: <https://grpc.io/docs/> (дата звернення: 10.05.2024).

11. Motisi F. Database per service, shared instance or shared database?. Medium.com. URL: <https://mts88.medium.com/database-per-service-or-shared-database-e73cfb756aa1> (дата звернення: 10.05.2024).

12. Docker overview. Docker Documentation. URL: <https://docs.docker.com/get-started/overview/> (дата звернення: 10.05.2024).

13. Kubernetes Documentation. Kubernetes. URL: <https://kubernetes.io/docs/home/> (дата звернення: 10.05.2024).

14. Azure Container Registry documentation. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/azure/container-registry/> (дата звернення: 10.05.2024).

15. Azure Pipelines documentation - Azure DevOps. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/azure/devops/pipelines/?view=azure-devops> (дата звернення: 10.05.2024).

16. Open Charge Point Protocol JSON 1.6, OCPP-J 1.6 Specification. Open Charge Alliance. URL: <https://openchargealliance.org/my-oca/ocpp/> (дата звернення: 06.06.2024).

17. Klishin M. Deploying RabbitMQ to Kubernetes: What's Involved? | RabbitMQ. rabbitmq.com. URL: <https://www.rabbitmq.com/blog/2020/08/10/deploying-rabbitmq-to-kubernetes-whats-involved> (дата звернення: 06.06.2024).

18. Горкун Д., Налєскіна Т. Проектування angular додатку для програмної системи моніторингу зарядних станцій електромобілей використовуючи onion архітектур. Розвиток суспільства та науки в умовах цифрової трансформації : Міжнар. студент. наук. конф., м. Умань, 2 трав. 2024 р. Умань, 2024. С. 60–62. URL: [https://archive.liga.science/plugins/generic/pdfJsViewer/pdf.js/web/viewer.html?file=https://archive.liga.science/index.php/conference-proceedings/issue/download/inter-02.02.2024/62#\[%7B"num":137,"gen":0%7D,%7B"name":"FitR"%7D,-193,362,789,842\]](https://archive.liga.science/plugins/generic/pdfJsViewer/pdf.js/web/viewer.html?file=https://archive.liga.science/index.php/conference-proceedings/issue/download/inter-02.02.2024/62#[%7B%22num%22%3A137%2C%22gen%22%3A0%2C%22name%22%3A%22FitR%22%2C%22page%22%3A193%2C%22total%22%3A362%2C%22start%22%3A789%2C%22end%22%3A842%22%7D%5B%22num%22%3A137%2C%22gen%22%3A0%2C%22name%22%3A%22FitR%22%2C%22page%22%3A193%2C%22total%22%3A362%2C%22start%22%3A789%2C%22end%22%3A842%22%7D%5D%5D) (дата звернення: 05.05.2024).

ДОДАТОК А

Архітектура серверної частини програмної системи

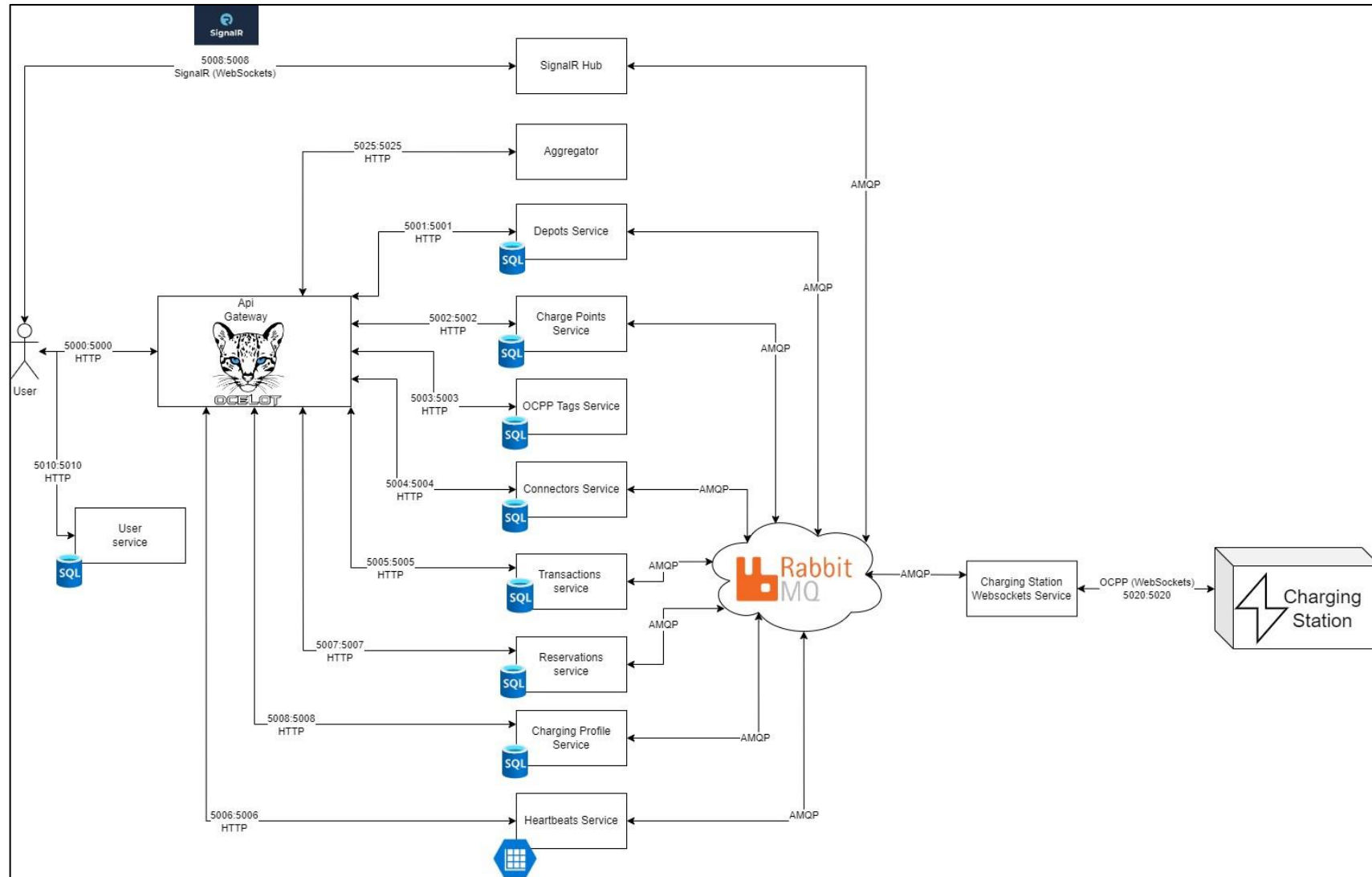


Рисунок А.1 – Архітектура серверної частини програмної системи (Рисунок виконано самостійно)

ДОДАТОК Б

Діаграми бази даних

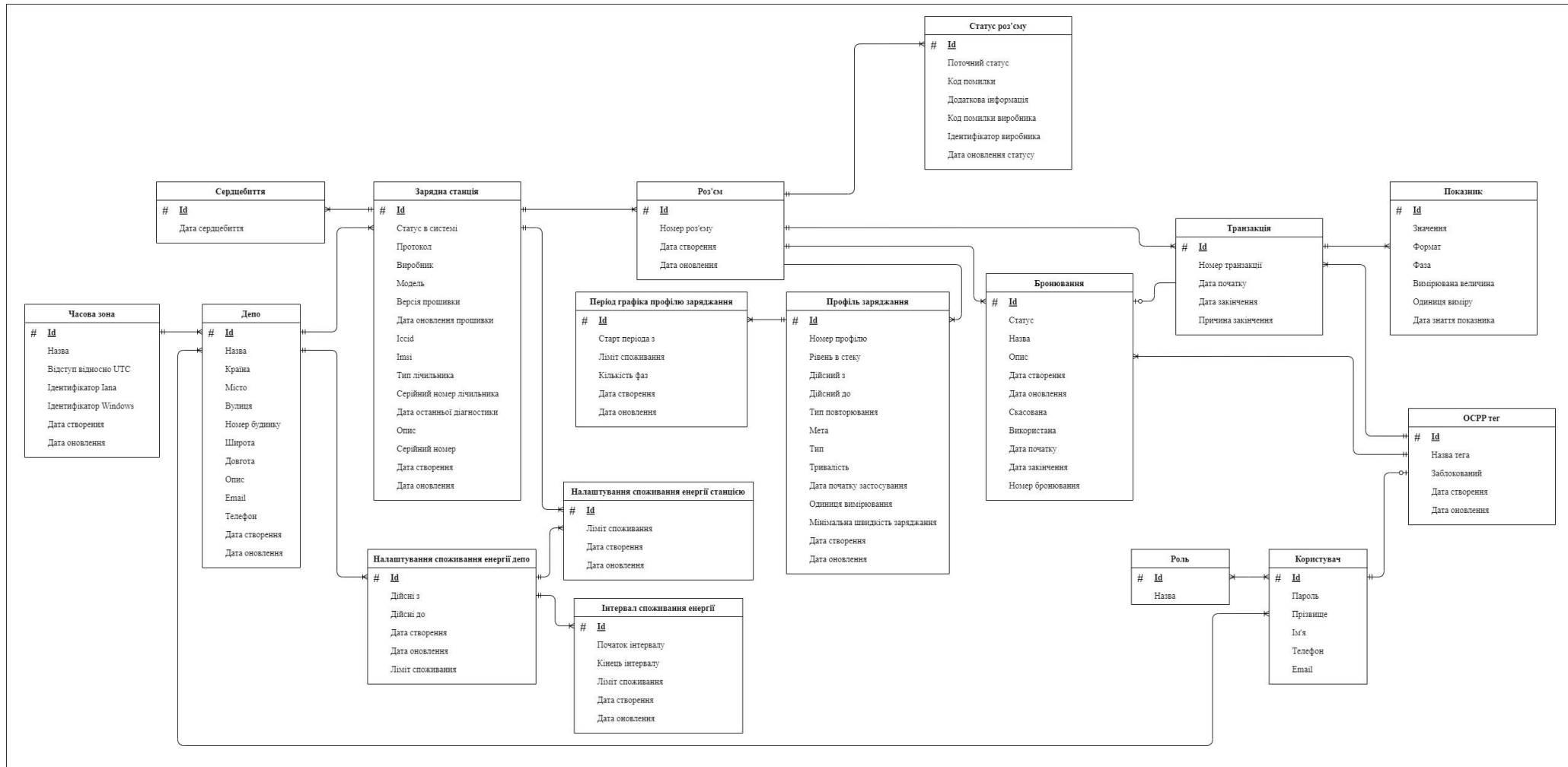


Рисунок Б.1 – ER-діаграма бази даних (Рисунок виконано самостійно)

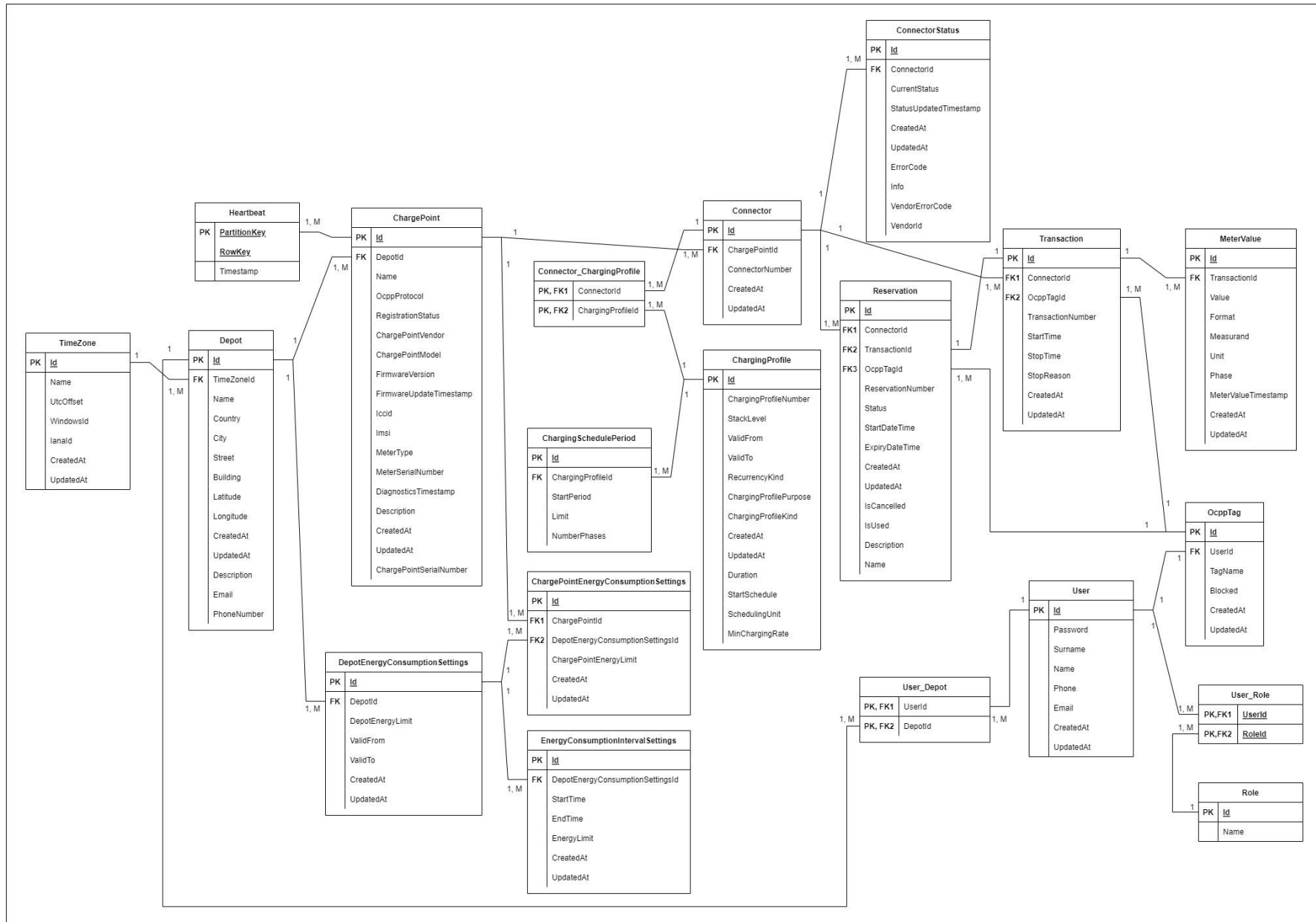


Рисунок Б.2 – Логічна модель бази даних (Рисунок виконано самостійно)

ДОДАТОК В

Тези XXVIII Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті» конференції «Інформаційні інтелектуальні системи»

УДК 004.415

DOI: <https://doi.org/10.30837/TFP.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, ефективною 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 – Тези доповіді (за даними [6])

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, OSCP 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.

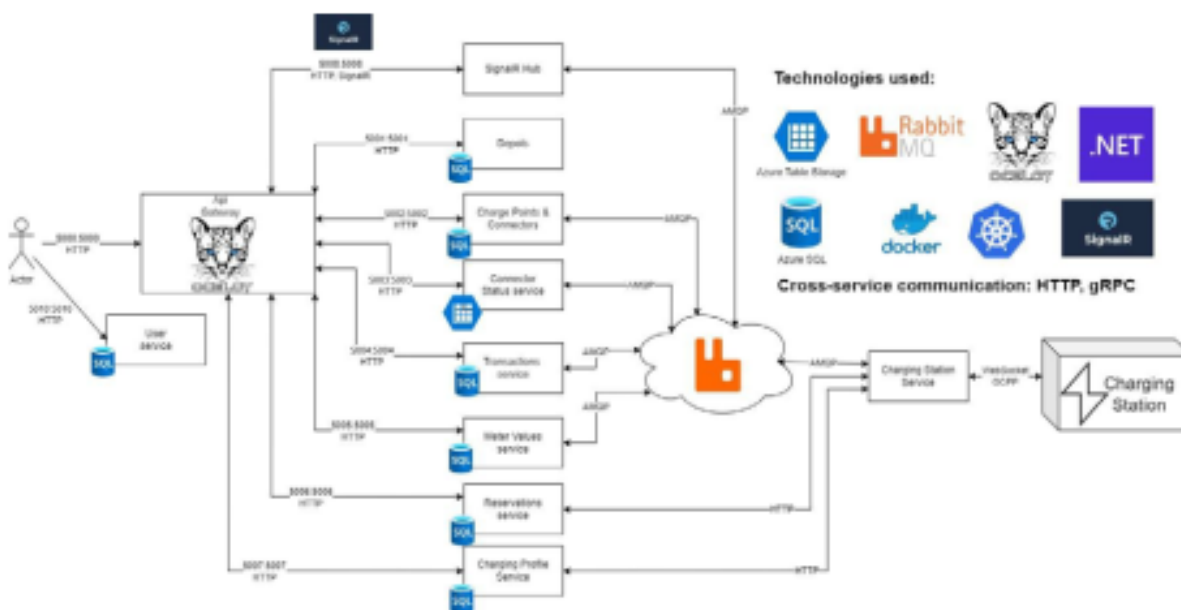


Рисунок 1 – Архітектура програмної системи

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 OSCP через сервіс Charging Station, а обробка запитів та відповідей відбуватиметься асинхронно за допомогою брокера RabbitMQ сервісами, написаними з використанням платформи ASP.NET.

З боку клієнтського застосунку [3] взаємодія відбуватиметься за протоколами HTTP 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, Налескіна Т. С. Проєктування 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 – Тези доповіді (за даними [6])

ДОДАТОК Г

Код Kubernetes маніфестів для розгортання stateless сервісів

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: depotsapi-deployment
  labels:
    app: depotsapi
spec:
  replicas: 1
  selector:
    matchLabels:
      app: depotsapi
  template:
    metadata:
      labels:
        app: depotsapi
    spec:
      containers:
        - name: depotsapi
          image: yevhen0825/depotsapi:latest
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 8080
          env:
            - name: ASPNETCORE_ENVIRONMENT
              value: Development
            - name: ASPNETCORE_HTTP_PORTS
              value: "8080"
            - name: ConnectionStrings__DefaultConnection
              valueFrom:
                secretKeyRef:
                  name: mssql-db-secret
                  key: mssql-main-db-connection-string
      resources:
        requests:
          memory: "256Mi"
          cpu: "250m"
        limits:
          memory: "512Mi"
          cpu: "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: depotsapi-service
  labels:
    app: depotsapi
spec:
  selector: app: depotsapi
  ports:
    - protocol: TCP
      name: http
      port: 8080
      targetPort: 8080

```

ДОДАТОК Д

Код Kubernetes маніфестів для розгортання RabbitMQ кластера

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: rabbitmq
spec:
  serviceName: rabbitmq
  replicas: 3
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata: labels: app: rabbitmq
    spec:
      serviceAccountName: rabbitmq
      initContainers:
        - name: config
          image: busybox
          command: ['/bin/sh', '-c', 'cp /tmp/config/rabbitmq.conf
/config/rabbitmq.conf && ls -l /config/ && cp /tmp/config/enabled_plugins
/etc/rabbitmq/enabled_plugins']
          volumeMounts:
            - name: config
              mountPath: /tmp/config/
              readOnly: false
            - name: config-file
              mountPath: /config/
            - name: plugins-file
              mountPath: /etc/rabbitmq/
        - name: set-erlang-cookie
          image: busybox
          command: ['/bin/sh', '-c', 'echo -n $RABBITMQ_ERLANG_COOKIE >
/var/lib/rabbitmq/.erlang.cookie && chmod 400
/var/lib/rabbitmq/.erlang.cookie']
          env:
            - name: RABBITMQ_ERLANG_COOKIE
              valueFrom:
                secretKeyRef:
                  name: rabbit-secret
                  key: RABBITMQ_ERLANG_COOKIE
          volumeMounts:
            - name: data
              mountPath: /var/lib/rabbitmq
      containers:
        - name: rabbitmq
          image: rabbitmq:3.13.2-management
          ports:
            - containerPort: 4369
              name: discovery
            - containerPort: 5672
              name: amqp
          env:
            - name: RABBIT_POD_NAME
              valueFrom:

```

```

        fieldRef:
          apiVersion: v1
          fieldPath: metadata.name
      - name: RABBIT_POD_NAMESPACE
        valueFrom:
          fieldRef:
            apiVersion: v1
            fieldPath: metadata.namespace
      - name: RABBITMQ_NODENAME
        value:
rabbit@$(RABBIT_POD_NAME).rabbitmq.$(RABBIT_POD_NAMESPACE).svc.cluster.local
1
      - name: RABBITMQ_USE_LONGNAME
        value: "true"
      - name: RABBITMQ_CONFIG_FILE
        value: "/config/rabbitmq"
      - name: K8S_HOSTNAME_SUFFIX
        value: .rabbitmq.$(RABBIT_POD_NAMESPACE).svc.cluster.local
    volumeMounts:
      - name: data
        mountPath: /var/lib/rabbitmq
        readOnly: false
      - name: config-file
        mountPath: /config/
      - name: plugins-file
        mountPath: /etc/rabbitmq/
    volumes:
      - name: config-file
        emptyDir: {}
      - name: plugins-file
        emptyDir: {}
      - name: config
        configMap:
          name: rabbitmq-config
          defaultMode: 0755
    volumeClaimTemplates:
      - metadata: name: data
        spec:
          accessModes: [ "ReadWriteOnce" ]
          storageClassName: "hostpath"
          resources:
            requests:
              storage: 50Mi
---
apiVersion: v1
kind: Service
metadata: name: rabbitmq
spec:
  clusterIP: None
  ports:
    - port: 4369
      targetPort: 4369
      name: discovery
    - port: 5672
      targetPort: 5672
      name: amqp
  selector: app: rabbitmq

```

ДОДАТОК Е

Код Kubernetes маніфестів для розгортання кластера Redis Sentinel

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: sentinel
spec:
  serviceName: sentinel
  replicas: 3
  selector:
    matchLabels:
      app: sentinel
  template:
    metadata:
      labels:
        app: sentinel
    spec:
      initContainers:
        - name: config
          image: redis:7.2.5-alpine
          command: [ "sh", "-c" ]
          args:
            - |
              REDIS_PASSWORD=redis-complex-password
              nodes=redis-0.redis.csms.svc.cluster.local,redis-
1.redis.csms.svc.cluster.local,redis-2.redis.csms.svc.cluster.local
              loop=$(echo $nodes | sed -e "s/,/\n/g")

              for i in $loop
              do
                echo "finding master at $i"
                MASTER=$(redis-cli --no-auth-warning --raw -h $i -a
$REDIS_PASSWORD info replication | awk '{print $1}' | grep master_host: |
cut -d ":" -f2)
                if [ "$MASTER" = "" ]; then
                  echo "no master found"
                  MASTER=
                else
                  echo "found $MASTER"
                  break
                fi
              done
              echo "sentinel monitor mymaster $MASTER 6379 2" >> /tmp/master
              echo "port 5000"
              sentinel resolve-hostnames yes
              sentinel announce-hostnames yes
              $(cat /tmp/master)
              sentinel down-after-milliseconds mymaster 5000
              sentinel failover-timeout mymaster 60000
              sentinel parallel-syncs mymaster 1
              sentinel auth-pass mymaster $REDIS_PASSWORD
              " > /etc/redis/sentinel.conf
              cat /etc/redis/sentinel.conf
      volumeMounts:
        - name: redis-config

```

```

        mountPath: /etc/redis/
containers:
- name: sentinel
  image: redis:7.2.5-alpine
  command: ["redis-sentinel"]
  args: ["/etc/redis/sentinel.conf"]
  ports:
  - containerPort: 5000
    name: sentinel
  volumeMounts:
  - name: redis-config
    mountPath: /etc/redis/
  - name: data
    mountPath: /data
volumes:
- name: redis-config
  emptyDir: {}
volumeClaimTemplates:
- metadata:
  name: data
  spec:
    accessModes: [ "ReadWriteOnce" ]
    storageClassName: "default"
    resources:
      requests:
        storage: 64Mi
---
apiVersion: v1
kind: Service
metadata:
  name: sentinel
spec:
  clusterIP: None
  ports:
  - port: 5000
    targetPort: 5000
    name: sentinel
  selector:
    app: sentinel

```

ДОДАТОК Ж

Код Kubernetes маніфестів для розгортання кластера Redis

```

apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: redis
spec:
  serviceName: redis
  replicas: 3
  selector:
    matchLabels:
      app: redis
  template:
    metadata:
      labels:
        app: redis
    spec:
      initContainers:
        - name: config
          image: redis:7.2.5-alpine
          command: [ "sh", "-c" ]
          args:
            - |
              cp /tmp/redis/redis.conf /etc/redis/redis.conf

              echo "finding master..."
              MASTER_FDQN=`hostname -f | sed -e 's/redis-[0-9]\./redis-0./'`
              if [ "$(redis-cli -h sentinel -p 5000 ping)" != "PONG" ]; then
                echo "master not found, defaulting to redis-0"

                if [ "$(hostname)" = "redis-0" ]; then
                  echo "this is redis-0, not updating config..."
                else
                  echo "updating redis.conf..."
                  echo "replicaof $MASTER_FDQN 6379" >> /etc/redis/redis.conf
                fi
              else
                echo "sentinel found, finding master"
                MASTER="$(redis-cli -h sentinel -p 5000 sentinel get-master-
addr-by-name mymaster | grep -E '(^redis-\d{1,})|([0-9]{1,3}\.[0-
9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})')")
                echo "master found : $MASTER, updating redis.conf"
                echo "replicaof $MASTER 6379" >> /etc/redis/redis.conf
              fi
            volumeMounts:
              - name: redis-config
                mountPath: /etc/redis/
              - name: config
                mountPath: /tmp/redis/
      containers:
        - name: redis
          image: redis:7.2.5-alpine
          command: ["redis-server"]
          args: ["/etc/redis/redis.conf"]
          ports:

```

```

    - containerPort: 6379
      name: redis
    volumeMounts:
    - name: data
      mountPath: /data
    - name: redis-config
      mountPath: /etc/redis/
  volumes:
  - name: redis-config
    emptyDir: {}
  - name: config
    configMap:
      name: redis-config
  volumeClaimTemplates:
  - metadata:
      name: data
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "default"
      resources:
        requests:
          storage: 64Mi
---
apiVersion: v1
kind: Service
metadata:
  name: redis
spec:
  clusterIP: None
  ports:
  - port: 6379
    targetPort: 6379
    name: redis
  selector:
    app: redis

```

ДОДАТОК И

Код Azure DevOps CI/CD пайплайну

```

trigger:
- main

resources:
- repo: self

parameters:
- name: 'forceBuild'
  type: boolean
  default: false
- name: 'projectsToBuild'
  type: object
  default:
    - path: "Services/Depots/Depots.Api"
      imageName: "depotsapi"
      k8sManifestPath: "aks/depots/api/deployment.yaml"
    - path: "Services/Depots/Depots.Grpc"
      imageName: "depotsgrpc"
      k8sManifestPath: "aks/depots/grpc/deployment.yaml"
    - path: "Services/Aggregator/Aggregator"
      imageName: "aggregator"
      k8sManifestPath: "aks/aggregator/deployment.yaml"
    - path: "Services/Gateway/ChargingStation.Gateway"
      imageName: "gateway"
      k8sManifestPath: "aks/gateway/deployment.yaml"
    - path: "Services/ChargePoints/ChargePoints.Api"
      imageName: "chargepointsapi"
      k8sManifestPath: "aks/chargepoints/api/deployment.yaml"
    - path: "Services/ChargePoints/ChargePoints.Grpc"
      imageName: "chargepointsgrpc"
      k8sManifestPath: "aks/chargepoints/grpc/deployment.yaml"
    - path: "Services/Connectors/Connectors.Api"
      imageName: "connectorsapi"
      k8sManifestPath: "aks/connectors/api/deployment.yaml"
    - path: "Services/Connectors/Connectors.Grpc"
      imageName: "connectorsgrpc"
      k8sManifestPath: "aks/connectors/grpc/deployment.yaml"
    - path: "Services/EnergyConsumption/EnergyConsumption.Api"
      imageName: "energyconsumptionapi"
      k8sManifestPath: "aks/energyconsumption/api/deployment.yaml"
    - path: "Services/EnergyConsumption/EnergyConsumption.Grpc"
      imageName: "energyconsumptiongrpc"
      k8sManifestPath: "aks/energyconsumption/grpc/deployment.yaml"
    - path: "Services/Transactions/Transactions.Api"
      imageName: "transactionsapi"
      k8sManifestPath: "aks/transactions/api/deployment.yaml"
    - path: "Services/Transactions/Transactions.Grpc"
      imageName: "transactionsgrpc"
      k8sManifestPath: "aks/transactions/grpc/deployment.yaml"
    - path: "Services/Reservations/Reservations.Api"
      imageName: "reservationsapi"
      k8sManifestPath: "aks/reservations/api/deployment.yaml"
    - path: "Services/Reservations/Reservations.Grpc"

```

```

    imageName: "reservationsgrpc"
    k8sManifestPath: "aks/reservations/grpc/deployment.yaml"
  - path: "Services/OcppTags/OcppTags.Api"
    imageName: "ocpptagsapi"
    k8sManifestPath: "aks/ocpptags/api/deployment.yaml"
  - path: "Services/OcppTags/OcppTags.Grpc"
    imageName: "ocpptagsgrpc"
    k8sManifestPath: "aks/ocpptags/grpc/deployment.yaml"
  - path: "Services/SignalR/ChargingStation.SignalR"
    imageName: "signalr"
    k8sManifestPath: "aks/signalr/deployment.yaml"
  - path: "Services/UserManagement/UserManagement.API"
    imageName: "usermanagement"
    k8sManifestPath: "aks/usermanagement/api/deployment.yaml"
  - path: "Services/WebSockets/ChargingStation.WebSockets"
    imageName: "websockets"
    k8sManifestPath: "aks/websockets/deployment.yaml"
  - path: "Services/ChargingProfiles/ChargingStation.ChargingProfiles"
    imageName: "chargingprofilesapi"
    k8sManifestPath: "aks/chargingprofiles/api/deployment.yaml"
  - path: "Services/Heartbeats/ChargingStation.Heartbeats"
    imageName: "heartbeatsapi"
    k8sManifestPath: "aks/heartbeats/api/deployment.yaml"
  - path: "Emulator/ChargePointEmulator.UI"
    imageName: "emulator"
    k8sManifestPath: "aks/emulator/deployment.yaml"

variables:
  dockerRegistryServiceConnection: '9b0814d5-f32c-4e81-966e-e60f137ff9b2'
  imageRepository: 'depotsapi'
  containerRegistry: 'csmsacr0825.azurecr.io'
  dockerBuildContext: '$(Build.SourcesDirectory)/ChargingStation.Backend'
  tag: '$(Build.BuildId)'
  vmImageName: 'ubuntu-latest'
  infrastructureProjectsPath:
  '$(Build.SourcesDirectory)/ChargingStation.Backend/Infrastructure'
  domainProjectsPath:
  '$(Build.SourcesDirectory)/ChargingStation.Backend/Domain'
  imagePullSecret: 'csmsacr08252120d50d-auth'

pool:
  vmImage: $(vmImageName)

stages:
- ${{ each project in parameters.projectsToBuild }}:
  - stage: 'Build_${{project.imageName}}'
    displayName: 'Build and push ${project.imageName}'
    jobs:

    - ${{ if not(parameters.forceBuild) }}:
      - job: 'DetectChanges'
        displayName: 'Detecting changes for ${project.imageName}'

    steps:

    - checkout: self
      fetchDepth: 2

```

```

- bash: |
    # Get microservice project changes
    MicroserviceProjectChanged=$(git diff HEAD~ HEAD --name-only |
grep -e "^${{ project.path }}")
    if [ ! -z "$MicroserviceProjectChanged" ]; then
        echo "##vso[task.setvariable
variable=MicroserviceProjectChanged;isOutput=true]true"
        echo "Changes detected in ${{ project.imageName }}, image
will be built..."
    else
        echo "##vso[task.setvariable
variable=MicroserviceProjectChanged;isOutput=true]false"
        echo "No changes detected in ${{ project.imageName }}, image
build will be skipped..."
    fi
    name: 'MicroserviceChanges'
    displayName: 'Detecting changes in ${{ project.imageName }}'

- bash: |
    # Get common projects changes
    InfrastructurePorjectsChanged=$(git diff HEAD~ HEAD --name-only
| grep -e "^${{ variables.infrastructureProjectsPath }}")
    DomainProjectsChanged=$(git diff HEAD~ HEAD --name-only | grep
-e "^${{ variables.domainProjectsPath }}")

    if [ ! -z "$InfrastructurePorjectsChanged" ] || [ ! -z
"$DomainProjectsChanged" ]; then
        echo "##vso[task.setvariable
variable=CommonProjectsChanged;isOutput=true]true"
        echo "Changes detected in common projects, ${{
project.imageName }} image will be built..."
    else
        echo "##vso[task.setvariable
variable=CommonProjectsChanged;isOutput=true]false"
        echo "No changes detected in common projects, ${{
project.imageName }} image build will be skipped..."
    fi
    name: 'CommonProjectsChanges'
    displayName: 'Detecting changes in common projects'

- job: 'BuildImageAndPushToAcr'
  displayName: 'Build and push image ${{ project.imageName }} into ACR'

  ${{ if not(parameters.forceBuild) }}:
    dependsOn: 'DetectChanges'
    condition:
or(eq(dependencies.DetectChanges.outputs['MicroserviceChanges.MicroserviceP
rojectChanged'], 'true'),
eq(dependencies.DetectChanges.outputs['CommonProjectsChanges.CommonProjects
Changed'], 'true'))

  steps:

- checkout: self
  fetchDepth: 2

```

```

- task: Docker@2
  displayName: Build and push an image to container registry
  inputs:
    command: buildAndPush
    repository: ${ project.imageName }
    dockerfile: $(Build.SourcesDirectory)/ChargingStation.Backend/${{
project.path }}/Dockerfile
    containerRegistry: $(dockerRegistryServiceConnection)
    buildContext: $(dockerBuildContext)
    tags: |
      $(tag)

- deployment: 'DeployToAKS'
  displayName: 'Deploy ${project.imageName}'

  dependsOn: 'BuildImageAndPushToAcr'
  condition: succeeded()

  environment: 'Nexus20ChargingStationBackend-1790.csms'
  strategy:
    runOnce:
      deploy:
        steps:
          - checkout: self
            fetchDepth: 2

          - task: KubernetesManifest@0
            displayName: Create imagePullSecret
            inputs:
              action: createSecret
              secretName: $(imagePullSecret)
              dockerRegistryEndpoint: $(dockerRegistryServiceConnection)

          - task: KubernetesManifest@0
            displayName: 'Deploy ${project.imageName} to Kubernetes
cluster'
            inputs:
              action: deploy
              manifests: |
                $(Build.SourcesDirectory)/${ project.k8sManifestPath }
              imagePullSecrets: |
                $(imagePullSecret)
              containers: |
                $(containerRegistry)/${ project.imageName }:$(tag)

```

ДОДАТОК К

Скріншоти інтерфейсу клієнтської частини

Depot configuration

General Chargers Time Slots **Finish**

 Name: **Senger - Willms**
Email: --

ADDRESS :
Brunei Darussalam, Anaiston, Bill Plaza, 00718

DEPOT ENERGY LIMIT (KWH)	START DATE	END DATE
2 kWh	Jun 2, 2024	Jun 8, 2024

#	Charger Name	Limit (Wh)
1	CP 02	300 W
2	CP 1	400 W
3	charger from FE	200 W

Total chargers	3
Configured chargers	3
Available chargers	0
Total used power	900 W
Left power	1.1 kW

Start time	End time	Limit (Wh)
Jun 2, 2024	Jun 5, 2024	1 kWh
Jun 6, 2024	Jun 6, 2024	0 Wh
Jun 7, 2024	Jun 8, 2024	1 kWh

NOTE : Please note that if any issues arise, they will be displayed on the depot dashboard. You can adjust the settings at any time.

[← Back to Time Intervals](#) [Save Configuration](#)

Рисунок И.1 – Сторінка налаштування депо (Рисунок виконано самостійно)

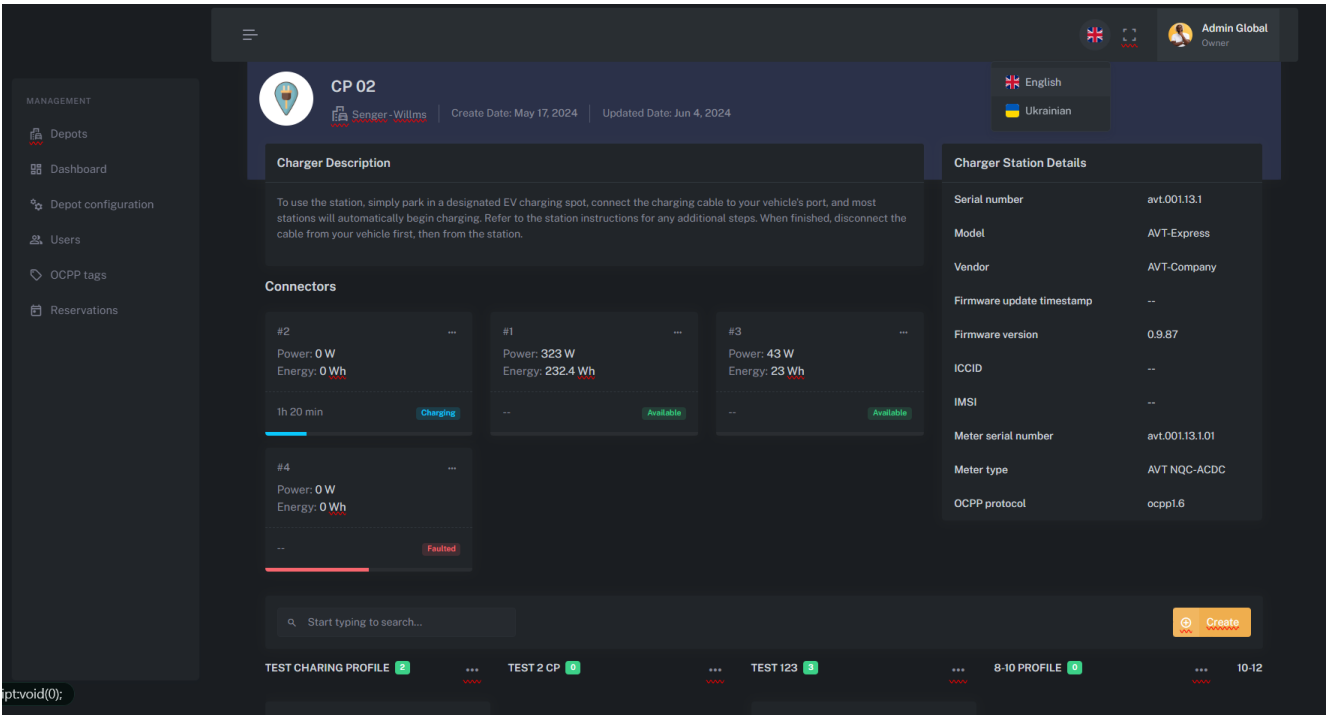


Рисунок И.2 – Сторінка інформації про зарядну станцію (Рисунок виконано самостійно)

ДОДАТОК Л

Звіт результатів перевірки на унікальність тексту в базі ХНУРЕ



Ім'я користувача:
Олійник Олена Володимирівна каф. ПІ

ID перевірки:
1016330931

Дата перевірки:
07.06.2024 09:07:05 EEST

Тип перевірки:
Doc vs Library

Дата звіту:
07.06.2024 09:08:07 EEST

ID користувача:
100012353

Назва документа: 2024_Б_ПІ_ПЗПІ-20-10_Чубаров_Є_Е

Кількість сторінок: 75 Кількість слів: 13847 Кількість символів: 110895 Розмір файлу: 1.81 MB ID файлу: 1016130506

6.54%
Схожість

Найбільша схожість: 2.52% з джерелом з Бібліотеки (ID файлу: 1015242317)

Пошук збігів з Інтернетом не проводився

6.54% Джерела з Бібліотеки

362

Сторінка 77

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

5

ДОДАТОК М

Слайди презентації

Харківський національний університет радіоелектроніки

Кафедра ПІ

Кваліфікаційна робота бакалавра

ПРОГРАМНА СИСТЕМА МОНІТОРИНГУ ЗАРЯДНИХ СТАНЦІЙ ЕЛЕКТРОМОБІЛІВ

АРХІТЕКТОР БАЗИ ДАНИХ ТА ПРОЄКТУ, DEVOPS ІНЖЕНЕР



Виконав: ст. гр. ПЗПІ-20-10 Чубаров Євген

Керівник: доц. каф. ПІ Русакова Н.Є.

Етапи розробки продукту



Аналіз предметної
області



Формування вимог



Проектування



Розробка



Тестування



Розгортання



Основні вимоги до продукту



Стабільність



Безпека



Швидкодія



Масштабованість

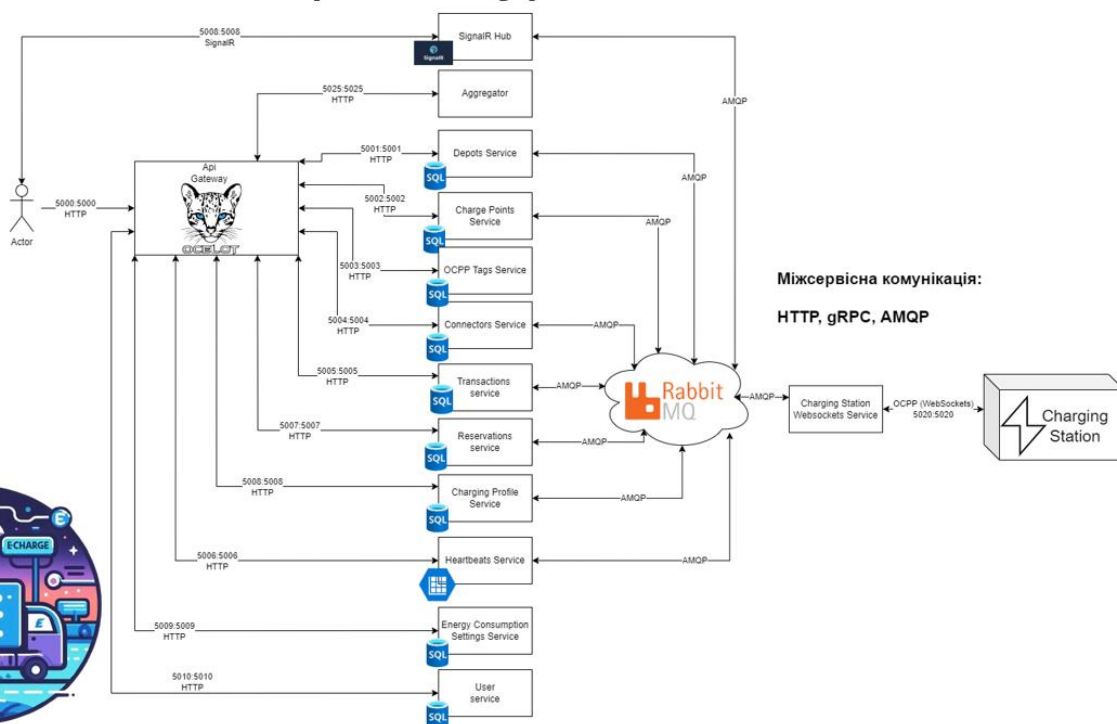


Функціональність



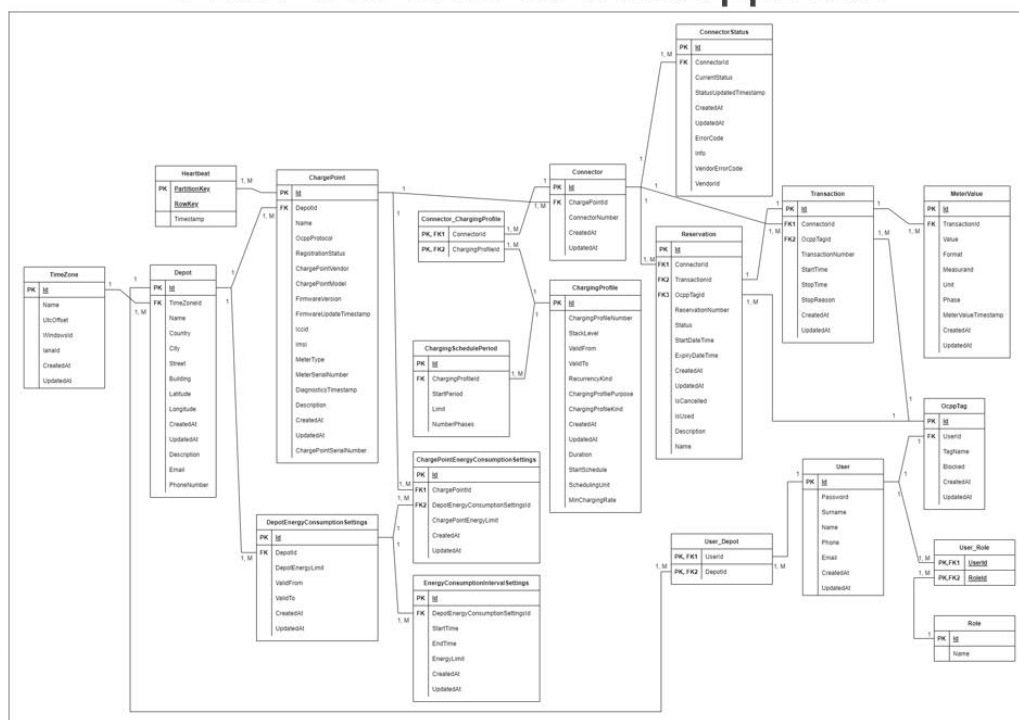
3

Архітектура системи



4

Логічна схема бази даних



5



Переваги мікросервісів



Гнучкість

Витрати ресурсів можна налаштовувати для кожного мікросервісу окремо



Масштабованість

Для кожного мікросервісу за необхідності можна розгорнути нові екземпляри



Надійність

При відмові одного з мікросервісів порушується робота тільки тих функцій, за які він відповідає



Швидкість розгортання

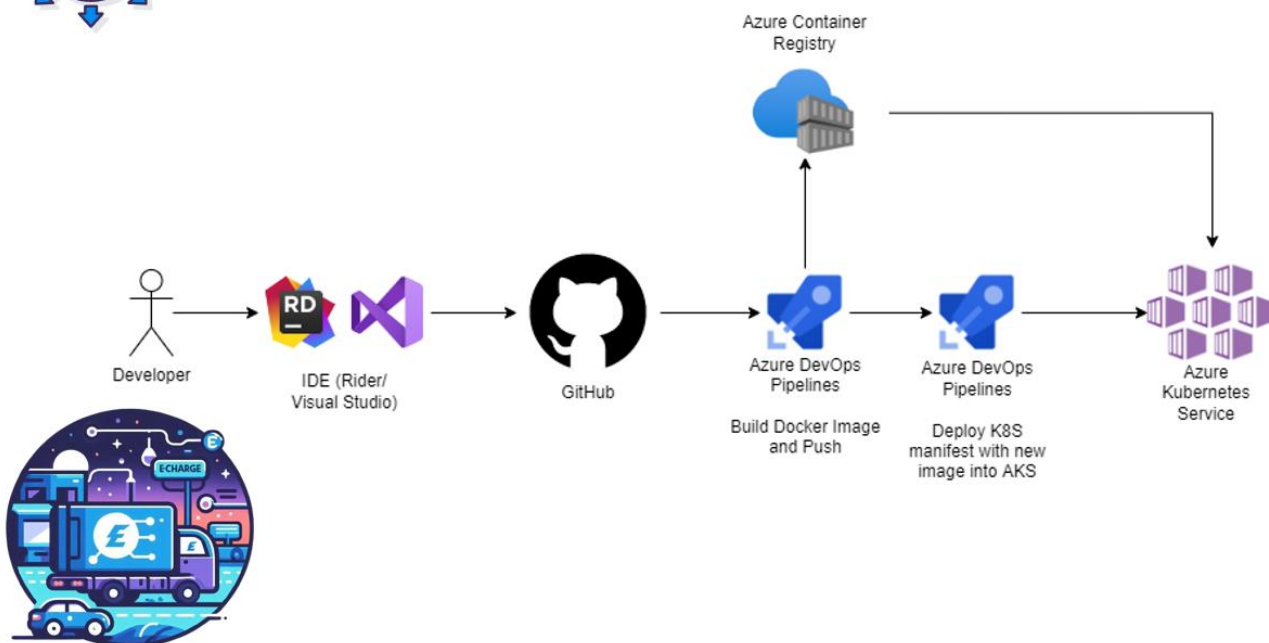
Кожен мікросервіс можна розгорнути окремо



6

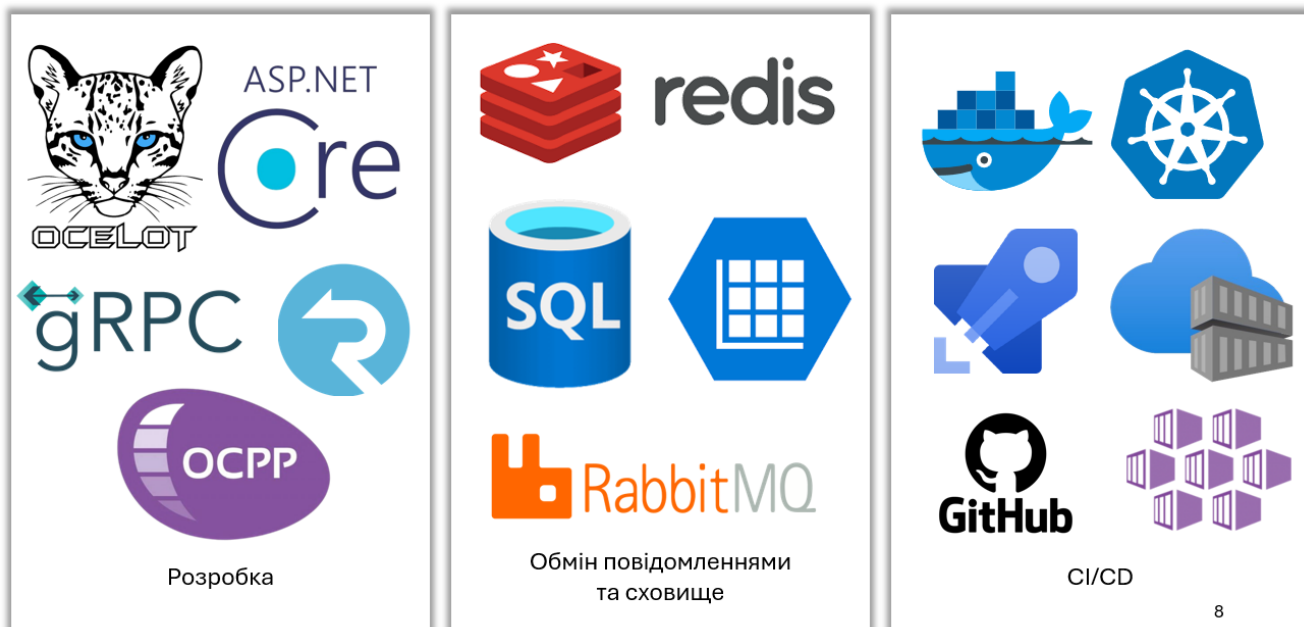


Розгортання мікросервісів



7

Використані технології



8



Результати розробки



The screenshot displays three main components of the development results:

- Code Coverage:** A table showing the status of various code coverage items. The table has columns for Name, Status, Stage, and Duration. The status is 'Success' for all items.
- Repositories:** A list of repositories managed by the system, including 'aggregator', 'chargepointapi', 'chargepointgpc', 'chargingprofileapi', 'connectorsapi', 'connectorgpc', 'depotapi', and 'depotgpc'.
- Charging Station Interface:** A detailed view of a charging station (CP 02) showing its status, connectors, and station details. The interface includes a 'Charger Description' section, a 'Connectors' section with a table of connector status, and a 'Charger Station Details' section with various technical specifications.



Впровадження програмного забезпечення



Конференція Інформаційні інтелектуальні системи» XXVIII Міжнародного молодіжного форуму «Радіоелектроніка та молодь у XXI столітті» 16-18 квітня 2024р. м. Харків

V Міжнародна студентська наукова конференції «Розвиток суспільства та науки в умовах цифрової трансформації»

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
РАДІОЕЛЕКТРОНІКИ

МАТЕРІАЛИ XXVIII МІЖНАРОДНОГО МОЛОДІЖНОГО
ФОРУМУ

«РАДІОЕЛЕКТРОНІКА ТА МОЛОДЬ
У XXI СТОЛІТТІ»

16 – 18 квітня 2024 р.

Том 6

КОНФЕРЕНЦІЯ
«ІНФОРМАЦІЙНІ ІНТЕЛЕКТУАЛЬНІ СИСТЕМИ»
INFORMATION INTELLIGENT SYSTEMS

УДК 004.415

DOI: <https://doi.org/10.30837/IYF.IIS.2024.306>

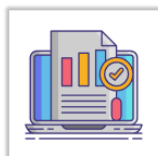
ПРОЄКТУВАННЯ СЕРВЕРНОЇ ЧАСТИНИ СИСТЕМИ КЕРУВАННЯ
ТА МОНІТОРІНГУ ЗАРЯДНИМИ СТАНЦІЯМИ

Чубаров Є. Е., Рубель Д. А.

Науковий керівник – к.т.н., доц. Русакова Н. С.
Харківський національний університет радіоелектроніки, каф. ПІ,
м. Харків, Україна



Висновки



Виконано аналіз предметної області та зокрема аналіз конкурентів



Обрано технології, спроектовано БД, архітектуру



Виконано розробку



Налаштовано CI/CD та виконано розгортання в Azure



11



Дякую за увагу



12

ДОДАТОК Н
Специфікація програмного забезпечення

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
- 1.2 Мета
- 1.3 Межі
- 1.4 Посилання
- 1.5 Означення та аббревіатури

2. Загальний опис

- 2.1 Перспективи продукту
- 2.2 Функції продукту
- 2.3 Характеристики користувачів
- 2.4 Загальні обмеження
- 2.5 Припущення та залежності

3. Конкретні вимоги

3.1 Вимоги до зовнішніх інтерфейсів

- 3.1.1 Інтерфейс користувача*
- 3.1.2 Апаратний інтерфейс*
- 3.1.3 Програмний інтерфейс*
- 3.1.4 Комунікаційний протокол*
- 3.1.5 Обмеження пам'яті*
- 3.1.6 Операції*
- 3.1.7 Функції продукту*

3.2 Властивості програмного продукту

3.3 Атрибути програмного продукту

- 3.5.1 Надійність*
- 3.5.2 Доступність*
- 3.5.3 Безпека*
- 3.5.4 Супроводжуваність*
- 3.5.5 Переносимість*
- 3.5.6 Продуктивність*

3.4 Вимоги бази даних

3.5 Інші вимоги

4. Додаткові матеріали

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., Налескіна 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. Проектування 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 аббревіатури

В документі використовуються наступні означення та аббревіатури:

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

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 задовольняють 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 складається 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 на вивчення та розробку паралельних завдань.

3. Конкретні вимоги

3.1 Вимоги до зовнішніх інтерфейсів

3.1.1 Інтерфейс користувача

Незареєстрований користувач має потрапляти на сторінку з лендінгом, де описані переваги роботи з продуктом (дивись рисунок 1).

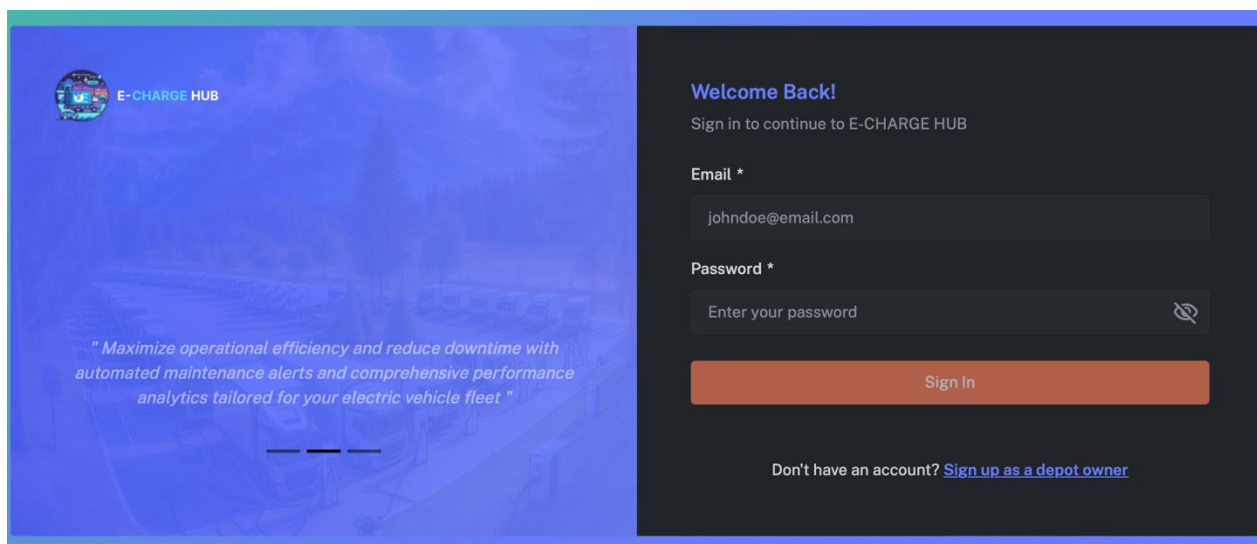


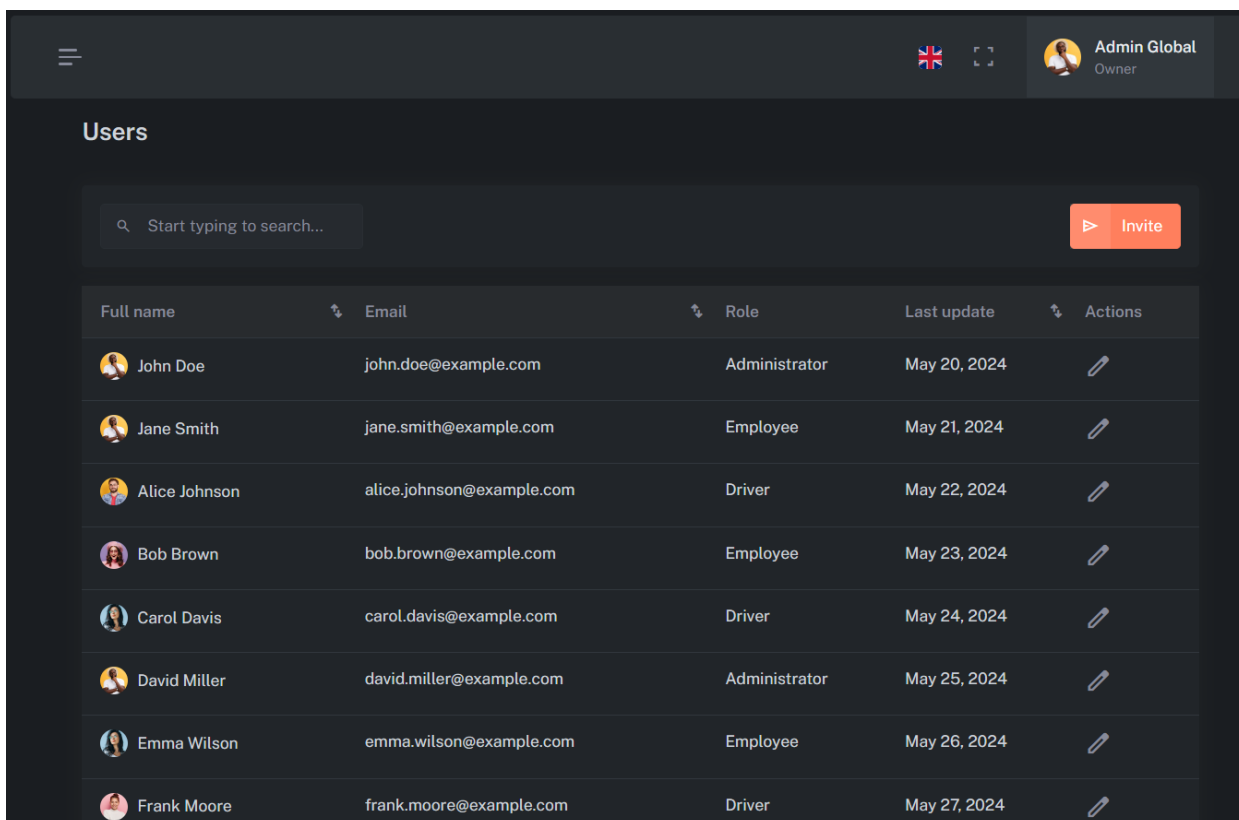
Рисунок 1 – Лендінг сторінка

На лендінг сторінці програмного продукту повинні бути логотип та навігація з верхньою панеллю, що містить логотип додатку, кнопки для авторизації та реєстрації користувачів. Також має бути форма реєстрації, яка дозволяє користувачам заповнити контактну форму для реєстрації, з полями для введення адреси електронної пошти та тіла повідомлення, яке буде надіслано адміністратору сервісу. У футері повинен бути логотип сервісу. Основний функціонал для авторизованих користувачів повинен включати меню з вкладками функцій системи.

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.



Full name	Email	Role	Last update	Actions
John Doe	john.doe@example.com	Administrator	May 20, 2024	
Jane Smith	jane.smith@example.com	Employee	May 21, 2024	
Alice Johnson	alice.johnson@example.com	Driver	May 22, 2024	
Bob Brown	bob.brown@example.com	Employee	May 23, 2024	
Carol Davis	carol.davis@example.com	Driver	May 24, 2024	
David Miller	david.miller@example.com	Administrator	May 25, 2024	
Emma Wilson	emma.wilson@example.com	Employee	May 26, 2024	
Frank Moore	frank.moore@example.com	Driver	May 27, 2024	

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 Осрр 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.

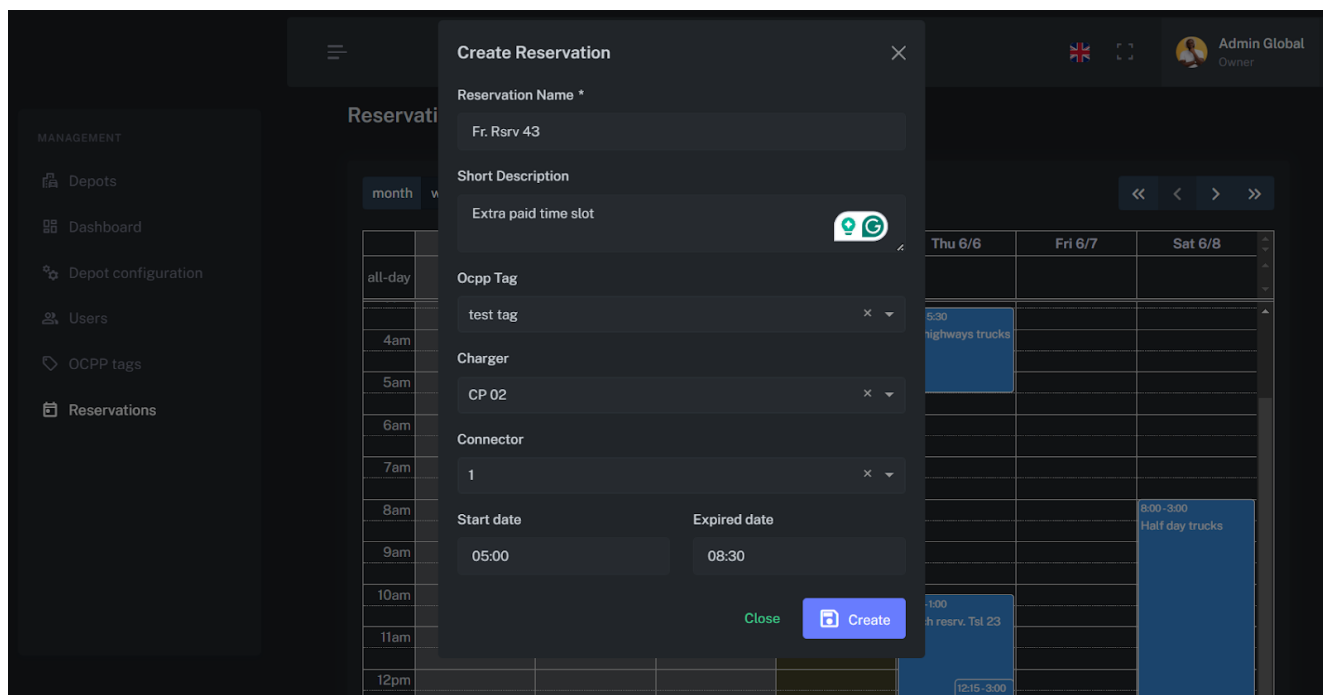


Рисунок 4 - Сторінка створення резервації зарядної станції

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.

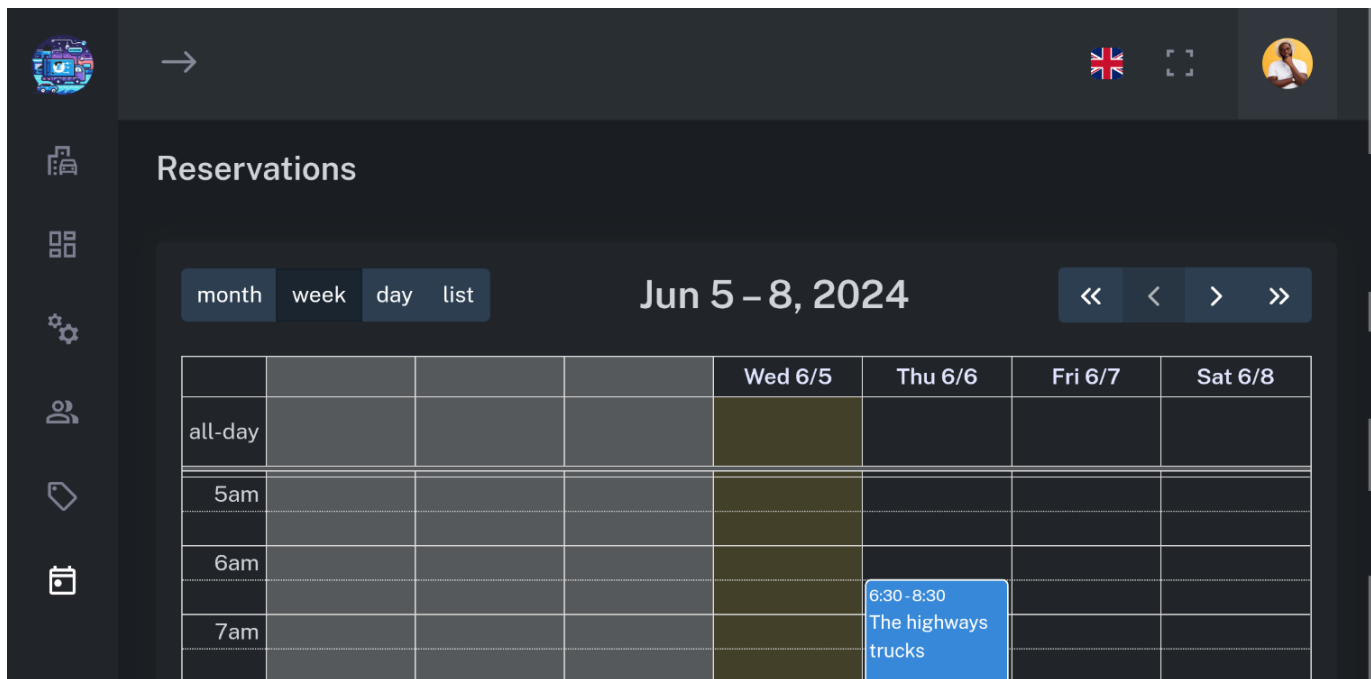


Рисунок 5 - Календар резервацій

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 OSCPP та веб-сокети з відповідним сервісом обробки підключень. RabbitMQ підтримує протокол AMQP. Протокол OSCPP 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 браузерях 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 Azure SQL Server, яка має наступні переваги:

масштабованість та продуктивність: Azure 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 Azure 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

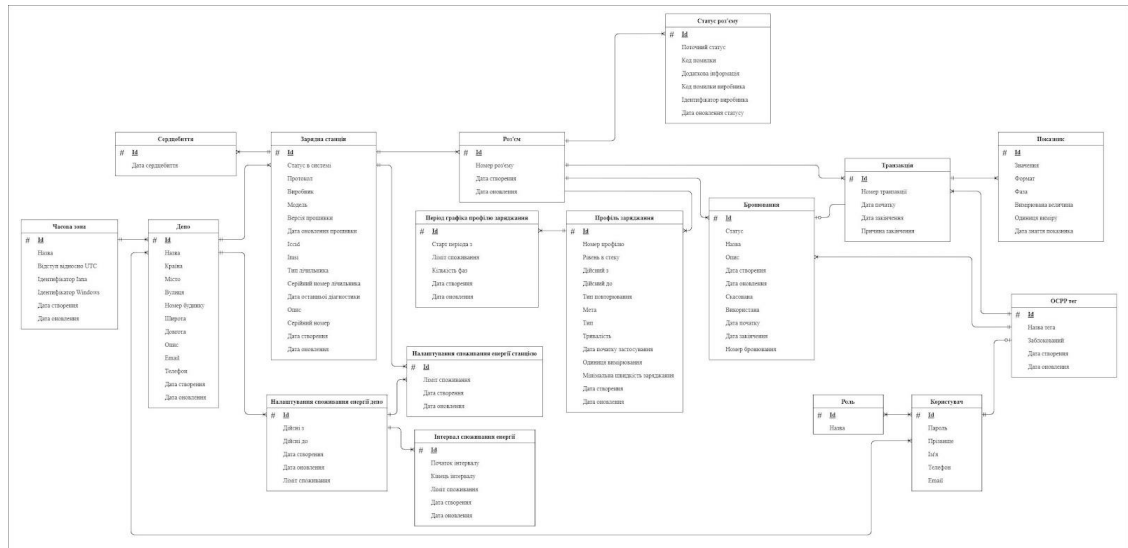


Рисунок 4 – ER-діаграма бази даних

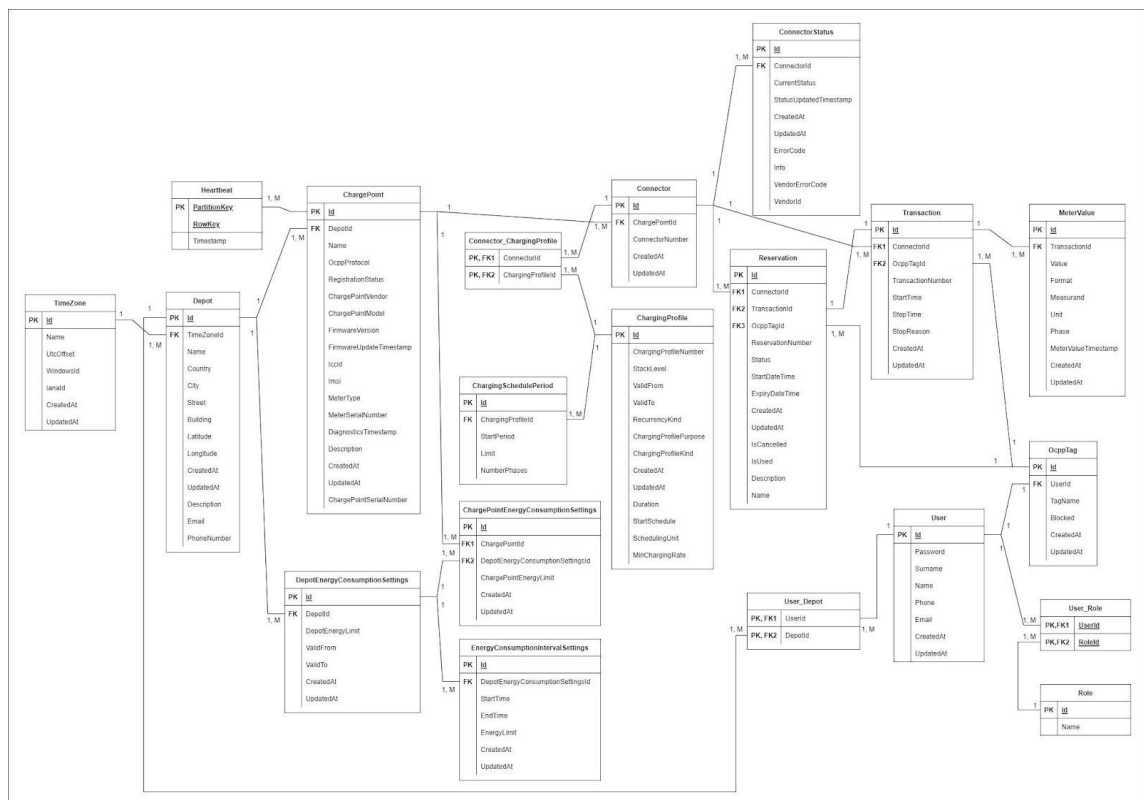


Рисунок 5 – Логічна модель бази даних

4.2 Діаграма прецедентів

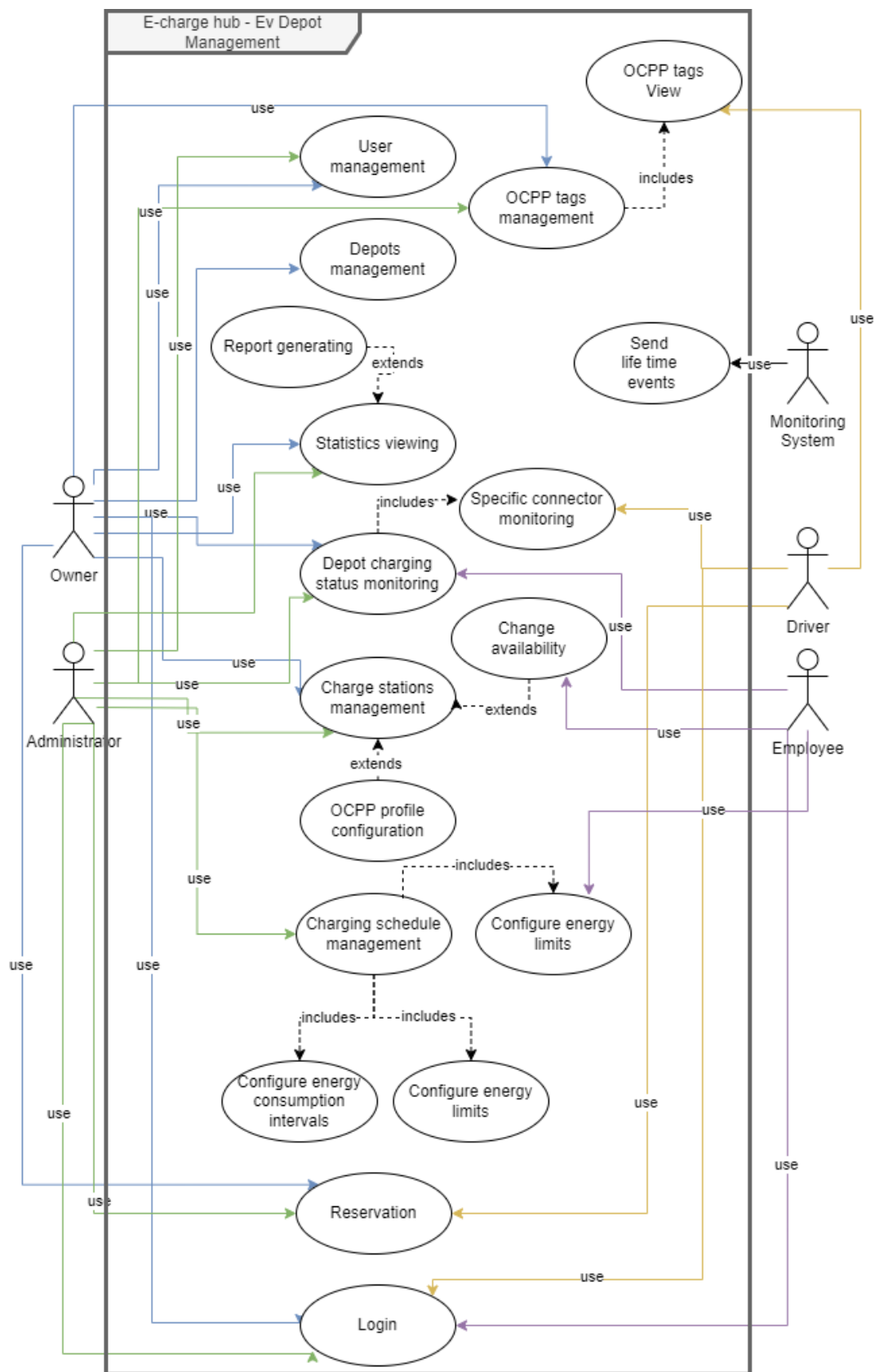


Рисунок 6 – Діаграма прецедентів

4.3 Діаграма діяльності

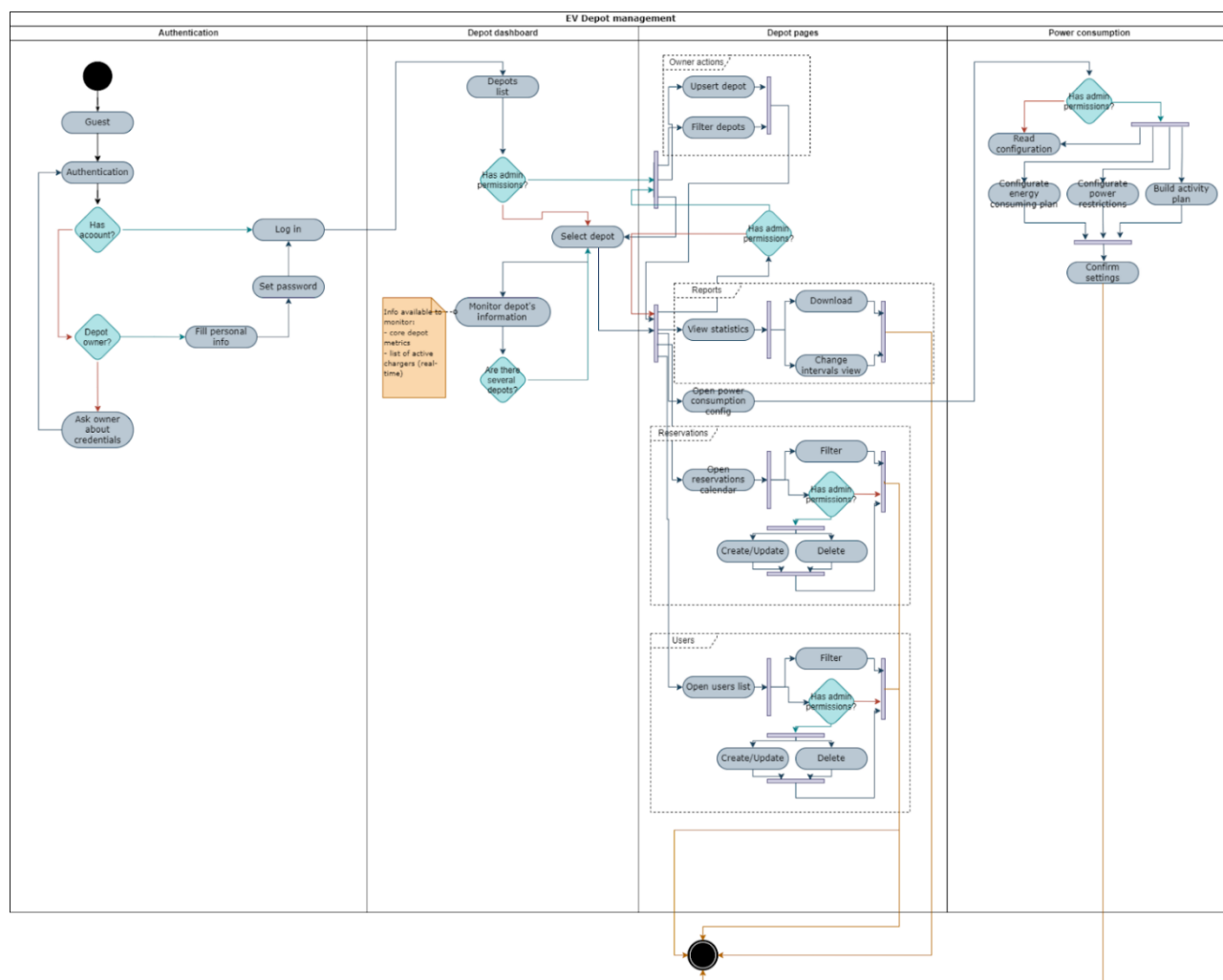


Рисунок 7 – Діаграма діяльності системи для управління споживання електроенергії депо електротранспорту