МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Чорноморський національний університет імені Петра Могили

Факультет комп'ютерних наук

Кафедра інженерії програмного забезпечення

ДОПУЩЕНО ДО ЗАХИСТУ

Завідувач кафедри_____ *Є. О. Давиденко*

«___» _____20____ р.

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

УНІВЕРСАЛЬНИЙ ПРОГРАМНИЙ КОМПЛЕКС СИСТЕМИ КОНТРОЛЮ ВАГИ

Спеціальність «Інженерія програмного забезпечення»

121 – КРМ.1 – 608. 21710924

Студент _____ Є. Д. Стоєв

підпис

«__» _____ 20__ р.

Керівник канд. пед. наук, доцент _____ К. О. Кірей

підпис

«__» _____ 20__ р.

Консультант д-р біологічних наук, професор _____ Л. І. Григор'єва

підпис

«__» _____ 20__ р.

Миколаїв – 2023

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Чорноморський національний університет імені Петра Могили
Факультет комп'ютерних наук
Кафедра інженерії програмного забезпечення

ЗАТВЕРДЖУЮ

Зав. кафедри Є.О. Давиденко

_____
(підпис)

«03» листопада 2022 р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи магістра

Видано студенту групи 608м факультету комп'ютерних наук
Стоєв Євгеній Дмитрович
(прізвище, ім'я, по батькові студента)

1. Тема кваліфікаційної роботи
Універсальний програмний комплекс системи контролю ваги

Затверджена наказом по ЧНУ від «03» листопада 2022р. № 200

2. Строк представлення кваліфікаційної роботи «17» лютого 2023р.

3. Очікуваний результат роботи та початкові дані, якщо такі потрібні
Очікуваним результатом є прототип універсального програмного комплексу системи контролю ваги

4. Перелік питань, що підлягають розробці: для досягнення мети необхідно вирішити такі задачі:
   - провести аналіз предметної галузі та аналогічних програмних систем зі схожим функціоналом;
   - обрати необхідний стек технологій для розробки універсального програмного комплексу системи контролю ваги;

- розробити архітектуру програмного комплексу з можливістю подальшого розширення функціональних можливостей системи;
- розробити архітектуру бази даних;
- розробити прототип універсального програмного комплексу системи контролю ваги;
- дослідити характеристики системи в різних експлуатаційних умовах.

5. Перелік графічних матеріалів
Слайди презентації

6. Завдання до спеціальної частини
Охорона праці та безпека у надзвичайних ситуаціях.

7. Консультанти:

Консультант	кафедра (організація)	Частина роботи
Григор'єва Л. І.	Кафедра екології	Спеціальна частина з охорони праці

Керівник роботи: канд. пед. наук, доцент Кірей К. О.
(посада, прізвище, ім'я, по батькові)

_____
(підпис)

Завдання прийнято до виконання

Стоєв Є. Д.
(прізвище, ім'я, по батькові студента)

_____
(підпис)

Дата видачі завдання «22» грудня 2022р.

КАЛЕНДАРНИЙ ПЛАН
виконання кваліфікаційної роботи

Тема: Універсальний програмний комплекс системи контролю ваги

№	Найменування роботи	Початок	Закінчення	Примітки
1.	Розробка та затвердження завдання на виконання КРМ	**10.11.2022**	**15.11.2022**	**Виконано**
2.	Огляд літератури за темою роботи	**20.11.2022**	**30.11.2022**	**Виконано**
3.	Аналіз предметної області	**05.12.2022**	**14.12.2022**	**Виконано**
4.	Розробка проєктних рішень	**15.12.2022**	**31.12.2022**	**Виконано**
5.	Моделювання та конструювання ПЗ	**02.01.2023**	**21.01.2023**	**Виконано**
6.	Кодування, тестування та апробація розробленого ПЗ, аналіз результатів тестування, розробка керівництва користувача	**10.01.2023**	**12.02.2023**	**Виконано**
7.	Розробка спеціальної частини з охорони праці	**01.02.2023**	**11.02.2023**	**Виконано**
8.	Відгук керівника КРМ	**16.02.2023**	**16.02.2023**	**Виконано**
8.	Оформлення КРМ та презентації	**05.02.2023**	**13.02.2023**	**Виконано**
10.	Попередній захист	**13.02.2023**	**13.02.2023**	**Виконано**
11.	Рецензування	**14.02.2023**	**14.02.2023**	**Виконано**
12.	Завершення оформлення КРМ та презентації	**17.02.2023**	**17.02.2023**	**Виконано**
13.	Захист кваліфікаційної роботи	**24.02.2023**	**24.02.2023**	**Виконано**

Розробив студент Стоєв Є. Д._____ _____
(*прізвище, ім'я, по батькові)* *(підпис)*

«__» _____ 20__ р.

Керівник роботи _канд. пед. наук, доцент Кірей К. О._____ _____
(*посада, прізвище, ім'я, по батькові)* *(підпис)*

«__» _____ 20__ р.

АНОТАЦІЯ

до кваліфікаційної роботи магістра
«Універсальний програмний комплекс системи контролю ваги»
Студент 608 гр.: Стоєв Євгеній Дмитрович
Керівник: канд. пед. наук, доцент Кірей К. О.

Тема: «Універсальний програмний комплекс системи контролю ваги».

Об'єктом кваліфікаційної роботи є процес зважування транспорту для перевезення агрокультур

Предметом дослідження кваліфікаційної роботи є набір сервісів програмного комплексу для автоматизації та уніфікації програмних комплексів дистанційного зважування.

Метою кваліфікаційної роботи є удосконалення та покращення набору програмних рішень для автоматизації та уніфікації комплексів дистанційного зважування шляхом розробки універсального програмного комплексу системи контролю ваги.

Методами дослідження є теоретичні: аналіз джерел інформації за проблемою дослідження, студіювання підходів отримання ваги від вагопроцесорів. Емпіричні: налагоджування та тестування створеного програмного комплексу; аналіз стабільності системи в умовах реального використання та вивчення одержаної системної інформації.

Інновація полягає у розробленні вендернонезалежного алгоритму отримання ваги від вагопроцесорів, стандартизації програмних інтерфейсів для отримання значення ваги, розробці гнучкої, багатоцільової архітектури програмного комплексу.

Кваліфікаційна робота складається з фахової частини і спеціальної – охорони праці. Фахова частина складається з наступних розділів: вступ, розділи з моделюванням, тестуванням та дослідження системи, всього п'ять; висновків та додатків. У першому розділі описано архітектуру та програмні рішення для вирішення поставленої задачі, приділено увагу розгляду аналогів. Наведено обґрунтування обраних підходів та графічне представлення системи. У другому розділі описано створення архітектури проєкту, моделювання та проєктування інформаційної та функціональної моделі системи контролю ваги. Для опису використовуються такі діаграми, як: контекстна діаграма, діаграма використання, розгортання, послідовності та діаграма класів. У третьому розділі описано основний функціонал системи, його кодова частина та рекомендації щодо використання. У четвертому розділі розглядається налагодження та тестування розробленого програмного комплексу. У спеціальній частині з охорони праці розкрито питання безпечного робочого місця для розробки та тестування сервісу отримання ваги на підприємстві.

КРМ викладена на 68 сторінках, вона містить 4 розділи, 48 ілюстрацій,6 таблиць, 27 джерел в переліку посилань.

Ключові слова: *знання, вебсервіси, контроль ваги, універсальний програмний комплекс*, *WebSocket, docker, оператор вагового комплексу.*

ASTRACT

Of the Master`s Thesis

«Universal software complex of the weight control system»

Student of group 608: Stoiev Yevhenii Dmytrovych

Supervisor: Ph.D. Sc., Associate Professor Kirei K. O.

Topic: "Universal software complex of the weight control system."

The object of the qualification work is the process of weighing transport for the transportation of agricultural crops.

The subject of the research of the qualification work is a set of services of the software complex for the automation and unification of software complexes of remote weighing.

The purpose of the qualification work is to improve and improve the set of software solutions for the automation and unification of remote weighing complexes by developing a universal software complex of the weight control system.

Research methods are theoretical: analysis of sources of information on the research problem, study of approaches to obtaining weight from weight processors. Empirical: debugging and testing of the created software complex; analysis of system stability under conditions of real use and study of the obtained system information.

The innovation consists in the development of a vendor-independent algorithm for obtaining weight from weighing processors, standardization of software interfaces for obtaining weight values, development of a flexible, multi-purpose architecture of the software complex.

The qualification work consists of a professional part and a special part - labor protection. The professional part consists of the following sections: introduction, sections with modeling, testing and system research, five in total; conclusions and applications. The first chapter describes the architecture and software solutions for solving the given task, attention is paid to consideration of analogs. The justification of the chosen approaches and a graphic representation of the system are given. The second chapter describes the creation of the project architecture, modeling and designing of the information and functional model of the weight control system. Diagrams such as context diagram, usage diagram, deployment diagram, sequence diagram, and class diagram are used for description. The third section describes the main functionality of the system, its code part, and recommendations for use. The fourth chapter deals with debugging and testing of the developed software complex. In the special part on labor protection, the issue of a safe workplace for the development and testing of the weighing service at the enterprise is disclosed.

MQW is laid out on 68 pages, it contains 4 chapters, 48 illustrations, 6 tables, 27 sources in the list of references.

Keywords: *knowledge, web services, weight control, universal software complex, WebSocket, docker, weight complex operator.*

ЗМІСТ

ВСТУП

Промислові системи – один із видів програмного продукту, який потребує ретельного планування, прототипування та універсальності в архітектурі програмного комплексу. Промислові програмні комплекси в першу чергу орієнтовані на комерційні заощадження для компанії – замовника, шляхом автоматизації типових процесів. Одним із етапів комерційного заощадження є уніфікація програмних комплексів систем.

Україна є одним із найважливіших світових виробників зерна. Країна вирощує та експортує в основному пшеницю, кукурудзу та ячмінь. За даними Європейської комісії, на Україну припадає 10% світового ринку пшениці, 15% – кукурудзи, 13% – ячменю. Жодна компанія, яка займається культурами, не може працювати без вагових комплексів.

Об’єктом кваліфікаційної роботи є процес зважування транспортних засобів для перевезення агрокультур.

Предметом дослідження кваліфікаційної роботи є набір сервісів програмного комплексу для автоматизації та уніфікації програмних комплексів дистанційного зважування.

Актуальність теми кваліфікаційної роботи полягає в тому, що ринок вагових комплексів переповнений програмними системами різної якості, зазвичай, вони орієнтовні на конкретну модель вагопроцесору, що унеможливлює використання даних програмних рішень для уніфікації програмних систем з єдиним інтерфейсом доступу.

Метою кваліфікаційної роботи є удосконалення та покращення набору програмних рішень для автоматизації та уніфікації комплексів дистанційного зважування за рахунок розробки універсального програмного комплексу системи контролю ваги.

Для досягнення мети необхідно вирішити такі задачі:

- провести аналіз предметної галузі та аналогічних програмних систем зі схожим функціоналом;

- обрати необхідний стек технологій для розробки універсального програмного комплексу системи контролю ваги;
- розробити архітектуру програмного комплексу з можливістю подальшого розширення функціональних можливостей системи;
- розробити архітектуру бази даних;
- розробити прототип універсального програмного комплексу системи контролю ваги;
- дослідити характеристики системи в різних експлуатаційних умовах.

Інновація полягає:

- у розробленні вендернонезалежного алгоритму отримання ваги від вагопроцесорів;
- у розробленні стандартизованих програмних інтерфейсів для отримання значення ваги;
- у розробленні гнучкої, багатоцільової архітектури програмного комплексу;
- відмовостійкості програмного комплексу під час експлуатації в умовах використання на підприємстві.

Апробація результатів МКР. Результати роботи було представлено на Всеукраїнської науково-практичної конференції молодих вчених, аспірантів і студентів «Інформаційні технології та інженерія» (7-10 лютого 2023 р., ЧНУ ім. Петра Могили).

Публікації. Результати роботи представлено тезами доповіді: Стоєв Є. Д., Кірей К. О., Універсальний програмний комплекс системи контролю ваги. Інформаційні технології та інженерія : тези доп. Всеукр. наук.-практ. конф. молодих вчених, аспірантів і студентів. Миколаїв, 7–10 лютого 2023 р. Миколаїв : Чорном. нац. ун-т ім. Петра Могили, 2023. С. 109–111.

1 АНАЛІЗ УНІВЕРСАЛЬНОГО ПРОГРАМНОГО КОМПЛЕКСУ СИСТЕМИ КОНТРОЛЮ ВАГИ

1.1 Аналіз предметної області

Системи зважування, це досить прості за своєю логікою програмні та апаратні комплекси. Апаратна частина складається з певною кількості датчиків, які вимірюють силу, яка на них впливає та передають показники до вагопроцесору, який в свою чергу, програмним шляхом, конвертує отримані дані у цілочисельні значення. Якщо таких датчиків багато – розраховується середній показник.

Зазвичай, програмні рішення, що отримають показники від вагопроцесорів, слугують у ролі другого екрану і не мають можливості для відправки показників до систем бухгалтерського обліку або надати програмний інтерфейс для отримання цих показників, чи можливості підключити декілька вагопроцесорів різних виробників.

Проведемо аналіз представлених на ринку програмних рішень – аналогів системи, що розробляться.

1.2 Огляд аналогів

На ринку програмних комплексів систем контролю ваги є пропозиції щодо готових систем. Найбільш цікавими з них є: «*DINA TEL 3 APP*» та «Автоматизація вагових. Системи оперативного обліку сировини». У табл. 1.1 – 1.2 наведено основні характеристик: назва, виробник, мова реалізації, архітектура, огляд функцій, переваги та недоліки.

Таблиця 1.1 – опис програмного комплексу-аналогу *Dina Tel* 3

Назва	***DINA TEL 3 APP***
Виробник	Компанія «*Dinamica Generale S.p.A.*»
Мова реалізації	*Java, Swift*
Архітектура	*Client – Server*

Продовження таблиці 1.1 – опис програмного комплексу-аналогу *Dina Tel* 3

Функції	1. Можливість виконання простого зважування та зважування з подачею звукового сигналу; 2. можливість блокування показів вагів, роздруківки або збереження результатів зважування; 3. реалізує всі функції вагових мікрокомп'ютерів, що випускаються компанією *Dinamica Generale*; 4. можливість контролю кількох мікрокомп'ютерів, використовуючи один і той самий застосунок.
Переваги	1. Зручна у використанні; 2. доступна для *iOS* та *Android*; 3. широкий спектр функціоналу.
Недоліки	1. Працює лише з пристроями *Dinamica Generale*; 2. відсутній клієнт для комп’ютерів; 3. відсутня можливість підключення стороннього програмного забезпечення; 4. для роботи необхідно купувати додатковий модем; 5. система не підтримує рольову систему; 6. відсутня можливість підключення сторонніх датчиків та інших периферійних пристроїв.
Веб-сайт	https://www.dinamicagenerale.com/ru-ru/dina-tel-3-app.aspx

Таблиця 2.2 – опис програмного комплексу-аналогу «Автоматизація вагових. Системи оперативного обліку сировини»

Назва	**Автоматизація вагових. Системи оперативного обліку сировини**
Виробник	Компанія «ВКФ Промсервіс»
Мова реалізації	*C++, WinForms*
Архітектура	*Monolith*

Продовження таблиці 1.2 – опис програмного комплексу-аналогу «Автоматизація вагових. Системи оперативного обліку сировини»

Функції	1. моніторинг проходження транспортом строго встановлених контрольних точок (*RFID* мітки *RFID* карти, розпізнавання номерів) 2. фото фіксація зважувань і подій на вагах з прив'язкою до електронного документу; 3. ідентифікація тривожних подій в режимі реального часу за фактом порушення проходження маршруту транспортним засобом; 4. введення і обробка результатів аналізів проб, інтеграція з експрес-аналізаторами; 5. ведення розрахунків з перевізниками; 6. віддалене управління системою через *WEB* інтерфейс; 7. автоматичний обмін даними з типовими *1С* конфігураціями, інтеграція ваг в існуючу систему обліку підприємства;
Переваги	1. Широкий спектр функціональних можливостей; 2. можливість працювати з різними периферійними пристроями; 3. можливість формування звітності різного призначення; 4. гнучка система конфігурації програмного комплексу; 5. можливість приєднати до бухгалтерського програмного забезпечення; 6. масштабування;

Продовження таблиці 1.2 – опис програмного комплексу-аналогу «Автоматизація вагових. Системи оперативного обліку сировини»

Недоліки	1. Висока ціна впровадження; 2. необхідність купувати дороге та спеціалізоване обладнання; 3. відсутня гнучка система взаємодії програмного комплексу та бухгалтерського програмного забезпечення; 4. можливість працювати лише з «*1С* Бухгалтерія»; 5. застарілі технології реалізації; 6. відсутність кросплатформного рішення для серверної та клієнтської частини; 7. перевантажений інтерфейс користувача; 8. необхідність попередньої підготовки місцевості; 9. відсутня можливість використовувати віддалену базу даних; 10. відсутня можливість резервного копіювання даних та конфігурацій; 11. відсутня можливість оновлення через мережу Інтернет.
Веб-сайт	https://vkf.com.ua/product/avtomatyzatsiya-vagovyh-systemy-operatyvnogo-obliku-syrovyny/

Отже, було розглянуто основні програмні рішення – аналоги програмного комплексу системи контролю ваги. Кожне рішення має свої переваги і свої недоліки. До загальних переваг можна віднести можливість працювати з декількома пристроями, можливість конфігурації вагопроцесорів та можливість формувати звітність. До основних недоліків можна віднести відсутність підтримки різних операційних систем, відсутність можливості працювати з вагопроцесорами різних виробників.

1.3.Аналіз програмного комплексу

1.3.1. Основні функції

1. Підтримка авторизації та аутентифікації;
2. розділення ролей в системі;
3. розділення функціональних можливостей в залежності від ролі;
4. підключення зовнішніх датчиків руху та постановки;
5. підключення до *RTSP* потоку відео нагляду під час зважування;
6. модульне підключення драйверів для роботи з вагопроцесором;
7. додавання драйверу до програмного комплексу у ручному та автоматичному режимі через мережу Інтернет;
8. конфігурація мережі пристрою;
9. формування звітності за обраний відрізок часу;
10. підтримка зовнішніх запитів на отримання показників;
11. фіксація ваги у автоматичному та ручному режимі;
12. конфігурація автоматичного режиму зважування;
13. підключення до хабу пристроїв;
14. відновлення пароля користувача за допомогою листа на пошту;
15. контроль «плаваючого» нуля;
16. оновлення програмного комплексу через мережу Інтернет;
17. контроль актуальності програмного забезпечення;
18. інформування адміністратора про доступні оновлення програмного комплексу шляхом відправки електронного листа;
19. ведення історії оновлення комплексу.

1.3.2. Актори

- Адміністратор. Має повний доступ до системи з можливістю налаштування програмного забезпечення;
- Оператор. Фіксація ваги;
- Користувач. Отримання показників з вагопроцесору та датчиків.

1.3.3. Сценарії використання

Так, як програмний комплекс, що розробляється є досить комплексним, розглянемо опис основних прецедентів. У табл. 1.3 наведено опис прецеденту «Зважування у ручному режимі».

Таблиця 1.3 – Опис прецеденту «Зважування у ручному режимі»

Use case section	***Comment***
Use case Name	Зважування у ручному режимі.
Scope	Універсальний програмний комплекс системи віддаленого зважування.
Level	Успішно пройти процес авторизації, отримати доступ до зважування.
Primary Actor	Оператор.
Stakeholders and interests	1. Оператор – ручна фіксація ваги.
Preconditions	1. Оператор пройшов процес авторизації.
Success guarantee	1. Оператор використовує броузер з підтримкою *HTML 5, CSS 3*; 2. оператор має використовувати стабільне підключення до мережі Інтернет.
Main Success Scenario	1. Оператор має можливість зафіксувати поточну вагу.
Extensions	1. Оператор намагається зафіксувати вагу при змінному значенні: – оператор проходить процес авторизації та аутентифікації; – оператор потрапляє на сторінку фіксації та спостереження за процесом зважування, натискає кнопку фіксації ваги при змінному значенні ваги; – система відображає повідомлення про неможливість фіксації ваги у випадку змінного показника.
Special Requirements	1. Швидкість доступу до мережі Інтернет має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	< 10%.
Miscellaneous	1. Чи активний обліковий запис оператора.

У табл. 1.4 наведено опис прецеденту «Налаштування обраного драйверу».

Таблиця 1.4 – Опис прецеденту «Налаштування обраного драйверу»

Use case section	***Comment***
Use case Name	Налаштування обраного драйверу.
Scope	Універсальний програмний комплекс системи віддаленого зважування.
Level	Успішно пройти процес авторизації як адміністратор, отримати доступ до сторінки налаштувань системи.
Primary Actor	Адміністратор.
Stakeholders and interests	1. Адміністратор – конфігурація необхідного драйверу для підключення до вагопроцесору.
Preconditions	1. Адміністратор пройшов процес авторизації і відкрив сторінку конфігурації драйверу системи.
Success guarantee	1. Адміністратор використовує броузер з підтримкою *HTML 5, CSS 3;* 2. адміністратор має використовувати стабільне підключення до мережі Інтернет.
Main Success Scenario	1. Адміністратор має можливість змінити та зберегти внесені зміни.
Extensions	1. Адміністратор зберігає налаштування з порожніми полями: – адміністратор обирає необхідний *COM* порт для встановлення нової конфігурації; – адміністратор зберігає налаштування без заповнення полей; – система зберігає значення і відображає повідомлення про порожність полей конфігурації; 2. Адміністратор зберігає налаштування без внесення змін: – адміністратор переходить на сторінку конфігурації системи; – адміністратор обирає необхідний *COM* порт

Продовження таблиці 1.4 – Опис прецеденту «Налаштування обраного драйверу»

Extensions	– адміністратор зберігає налаштування без внесення змін; – система не вносить змін до системи і повідомляє користувача про однакові дані.
Special Requirements	1. Швидкість доступу до мережі Інтернет має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	> 50%
Miscellaneous	1. Чи активний обліковий запис адміністратора. 2. Чи є доступ до бази даних.

У табл. 1.5 наведено опис прецеденту «Оновлення програмного комплексу через мережу Інтернет».

Таблиця 1.5 – Опис прецеденту «Оновлення програмного комплексу через мережу Інтернет»

Use case section	***Comment***
Use case Name	Оновлення програмного комплексу через мережу Інтернет.
Scope	Універсальний програмний комплекс системи віддаленого зважування.
Level	Успішно пройти процес авторизації як адміністратор, отримати доступ до сторінки перевірки та завантаження оновлень.
Primary Actor	Адміністратор.
Stakeholders and interests	1. Адміністратор – оновлення програмного комплексу до актуальної версії, виправлення програмних помилок та поява нового функціоналу для конфігурації програмного комплексу.
Preconditions	1. Адміністратор пройшов процес авторизації.
Success guarantee	1. Адміністратор використовує броузер з підтримкою *HTML 5, CSS 3;* 2. Адміністратор має використовувати стабільне підключення до мережі Інтернет.
Main Success Scenario	1. Адміністратор має доступ до оновлення системи.

Продовження таблиці 1.5 – Опис прецеденту «Оновлення програмного комплексу через мережу Інтернет»

Extensions	1. Адміністратор відкриває сторінку оновлення системи: – Адміністратор відкриває сторінку оновлення програмного комплексу; – система автоматично перевіряє доступність нової версії програмного комплексу; – система відображає повідомлення з номером нової версії, у разі наявності нової версії програмного комплексу, – адміністратор має можливість пропустити оновлення у разі, якщо нова версія є необов'язковою; – адміністратор має можливість розпочати процес оновлення, натиснувши кнопку для оновлення. 2. Адміністратор переглядає перелік змін у новій версії: – Адміністратор відкриває сторінку оновлення програмного комплексу; – адміністратор відкриває розділ з історією оновлення; – система отримує від серверу історію усіх змін новій версії; – система формує звіт з переліком усіх нововведень та виправлень.
Special Requirements	1. Швидкість доступу до мережі Інтернет має бути більшою ніж 60 кб/сек.
Frequency of Occurrence	< 10%.
Miscellaneous	1. Чи активний обліковий запис адміністратора.

Розглядаючи основні прецеденти системи, можна зрозуміти, які функціональні можливості є основними і першочерговими. У випадку універсального програмного комплексу системи контролю ваги це – підтримка актуального стану програмного забезпечення, наявність гнучкого налаштування та можливість контролю основного процесу – зважування.

1.3.4. Засоби реалізації

Правильно обрані технології – це, найважливіша частина будь-якого проекту. Обраний стек технологій для програмного комплексу було обрано з розрахунком на стабільність, гнучкість та продуктивність програмного середовища. У табл. 1.6 наведено опис засобів реалізації.

Таблиця 1.6 – Опис засобів реалізації

Позиція	Технологія
Мова програмування	*C# 11*
Технологія	*.NET 7*
База даних	*MariaDB*
ORM	*Entity Framework;*

1.3.5. Обґрунтування обраних технологій

Програми *ASP.NET* [1] є значною частиною сучасної веброзробки. Зрештою, платформа *.NET* є однією з найпопулярніших платформ розробників у всьому світі для створення вебдодатків і вебсервісів.

.NET – це платформа веброзробки, що складається з кількох різних інструментів, мов програмування та бібліотек. Розробники повного пакету *.NET* можуть очікувати, що мови програмування *C#, F#* і *Visual Basic* включені в базовий *.NET*.

ASP.NET – це платформа *.NET*, яка розширює можливості базової *.NET*. Розробники використовують фреймворки *.NET*, такі як *ASP.NET*, щоб отримати доступ до додаткових компонентів і інструментів, які роблять веброзробку простішою та ефективнішою. У випадку *ASP.NET* цей фреймворк найкраще використовувати для створення вебсайтів, вебдодатків і вебсервісів, орієнтованих на динамічний вміст.

C# – це об’єктно-орієнтована та статично типізована мова комп’ютерного програмування, створена Microsoft для використання на її платформі *.NET*. Його назва походить від мови *C*, з якої він успадковує подібний синтаксис [2].

C# був створений Microsoft і стандартизований *ISO* і *ECMA*. Її було офіційно випущено в 2002 році. З того часу мова зазнала численних удосконалень, останньою версією якої є *C#* 7.0.

Розробників *C#* часто називають розробниками *.NET*, оскільки мова *C#* майже виключно використовується з *.NET Framework*. Це досить популярна мова, яка зазвичай входить до п'ятірки найпопулярніших у багатьох чартах популярності. Він найчастіше використовується в розробці корпоративного програмного забезпечення, але також має процвітаючу екосистему з відкритим кодом.

MariaDB – це система керування реляційною базою даних із відкритим вихідним кодом, яка підтримується спільнотою та також є розширеною версією *MySQL*. *MariaDB* є швидкою, масштабованою та підтримує більше систем зберігання, ніж *MySQL*. Подібно до *MySQL*, *MariaDB* підтримує зовнішні плагіни, що означає, що ви можете розширити базу даних і застосувати її в інших випадках використання, таких як електронна комерція, сховище даних і програми журналювання.

MariaDB одночасно надійна та масштабована. Це додаткова заміна популярної бази даних *MySQL*, що означає, що ви можете замінити свій сервер *MySQL* на сервер *MariaDB*, не змінюючи код програми. [3]

Entity Framework – це *Object Relational Mapper*, який був розроблений Microsoft, щоб дозволити розробникам працювати з базою даних у програмах *.NET* за допомогою об'єктів *.NET*. Він піклується про встановлення з'єднання з базою даних, а також допомагає маніпулювати базами даних, дозволяючи створювати об'єкти та надає нам методи роботи з даними. Він розташований між бізнес-додатком і базою даних. *API Entity Framework* використовується для виконання всіх операцій, пов'язаних із базою даних, які відображають усі операції в базі даних [4].

Висновки до розділу 1

Будь-який проект розпочинається з постановки задачі та визначення пріоритетного вектору розвитку програмного продукту.

У першому розділі описано архітектуру та програмні рішення для вирішення поставленої задачі, приділено увагу розгляду аналогів. Наведено обґрунтування обраних підходів та графічне представлення системи.

Результатом розділу аналізу програмного комплексу системи контролю ваги є виділення основного функціоналу та функціональних вимог до програмного комплексу, що розробляється.

Результати були отримані шляхом визначення та вивчення головних існуючих аналогів проекту, а саме: розгляд головних переваг та недоліків кожного з них.

2 АРХІТЕКТУРА, МОДЕЛЮВАННЯ ТА ПРОЄКТУВАННЯ УНІВЕРСАЛЬНОГО ПРОГРАМНОГО КОМПЛЕКСУ СИСТЕМИ КОНТРОЛЮ ВАГИ

Розробка програмного продукту завжди розпочинається з моделювання об’єкту та предмету дослідження. Моделювання – довгий і складний процес, який потребує підготовки та аналізу області, для якої створюється програмне забезпечення. В свою чергу, моделювання складається з декількох етапів, а саме:

- Розробка функціональних моделей;
- розробка інформаційних моделей.

Для того, щоб краще зрозуміти сутність кожного етапу, розглянемо їх.

2.1.Функціональна модель

Функціональне моделювання надає схему того, що має робити система. Модель описує функції внутрішніх процесів за допомогою DFD (Data Flow Diagram).

Діаграми потоку даних – моделювання функцій представлено за допомогою DFD. DFD – це графічне представлення даних. Воно показує введення, вихід і обробку системи. Коли ми намагаємося створити власний бізнес, інформаційну систему, систему, проект, тоді потрібно з’ясувати, як інформація переходить від одного процесу до іншого. У DFD є кілька рівнів, але DFD до третього рівня достатньо для розуміння будь-якої системи [5].

Основними компонентами DFD є:

1. Зовнішня сутність – це сутність, яка отримує інформацію та передає її системі. Зображується прямокутником.
2. Потік даних – перехід даних з одного місця в інше показано потоком даних.
3. Процес – також називають символом функції. Використовується для обробки всієї інформації.

4. Сховище даних – використовується для зберігання інформації та отримання збереженої інформації.

2.1.1. Контекстна діаграма

Контекстна діаграма системи (також відома як *DFD* рівня 0) є найвищим рівнем у діаграмі потоку даних і містить лише один процес, що представляє всю систему, що встановлює контекст і межі системи, що моделюється. Він визначає потоки інформації між системою та зовнішніми об’єктами (тобто акторами). Контекстна діаграма зазвичай включається в документ вимог. Він повинен бути прочитаний усіма зацікавленими сторонами проекту, тому він повинен бути написаний простою мовою, щоб зацікавлені сторони могли зрозуміти елементи

Мета діаграми контексту системи – зосередити увагу на зовнішніх факторах і подіях, які слід враховувати при розробці повного набору системних вимог і обмежень. Контекстна діаграма системи часто використовується на початку проекту для визначення досліджуваного обсягу. Таким чином, у межах документа.

Контекстна діаграма системи представляє всі зовнішні сутності, які можуть взаємодіяти з системою. Вся система програмного забезпечення показана як єдиний процес. Така діаграма відображає систему в центрі без деталей її внутрішньої структури, оточену всіма її зовнішніми об’єктами, взаємодіючими системами та середовищами.

У системі, що розробляється є дві головні функції, а саме:

1. Отримання ваги через драйвер;
2. зміна конфігурації усієї платформи;

Розглянемо кожну функцію універсального програмного комплексу системи контролю ваги за допомогою діаграми *IDEF0*, починаючи з нульового рівня.

Нульовий рівень діаграми містить в собі лише назву проекту і є початковою точкою для декомпозиції на більш детальні процеси. Зробимо декомпозицію нульового рівня і виділимо основні функції проекту [6].

На рис. 2.1 наведено перший рівень діаграми IDEF0.

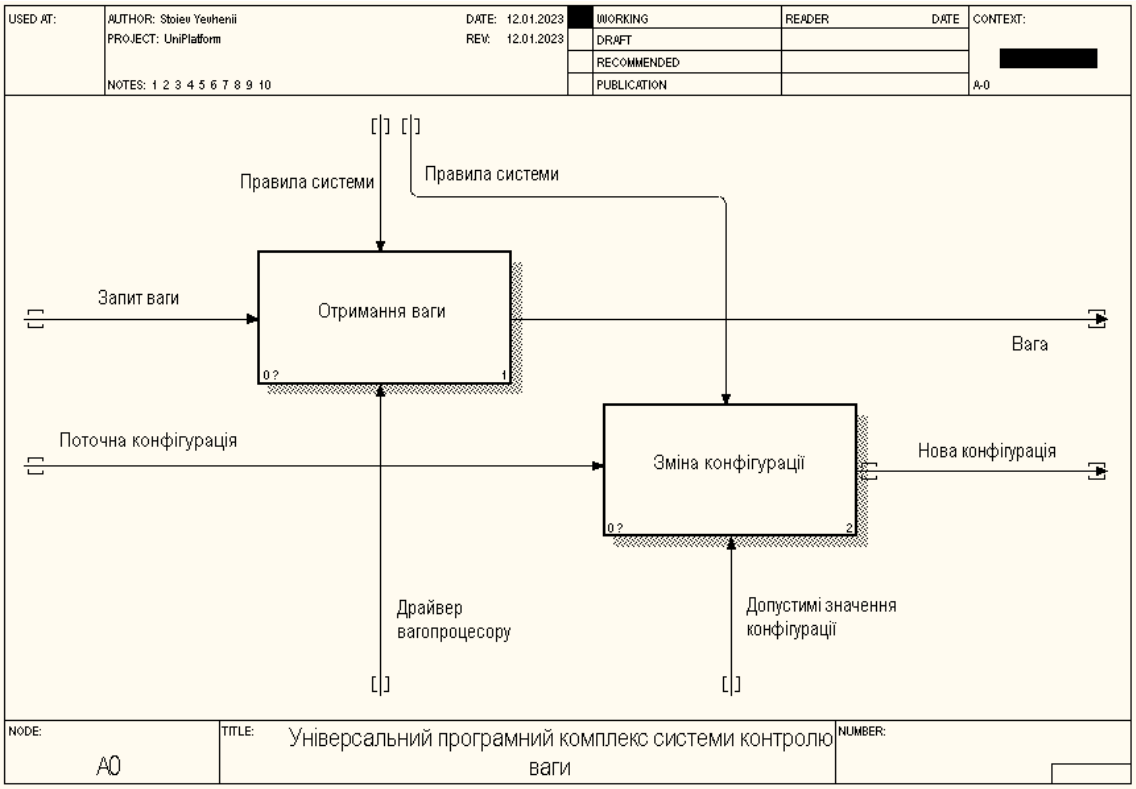

Рисунок 2.1 – Перший рівень діаграми IDEF0

Як бачимо з діаграми, в системі виділяється дві основні функції: отримання ваги та зміна конфігурації. Розглянемо кожну функцію окремо.

Сервіс отримання ваги є приймає запит від клієнтів, які діляться на дві категорії:

1. бухгалтерське програмне забезпечення;
2. вебінтерфейс.

Кожен з цих запитів фіксує вагу та додає позначку джерела виклику. На сам сервіс впливають певні правила системи, які координують правила отримання ваги, умов для отримання ваги, умов, коли вага рахується стабільною та інше.

Допоміжним елементом є безпосередньо драйвер системи.

IDEF0 діаграма передбачає детальний і простий опис кожного процесу, системи, що передбачає декомпозицію процесів, які потребують більш детального розгляду.

На рис. 2.2 наведено декомпозицію процесу отримання ваги.

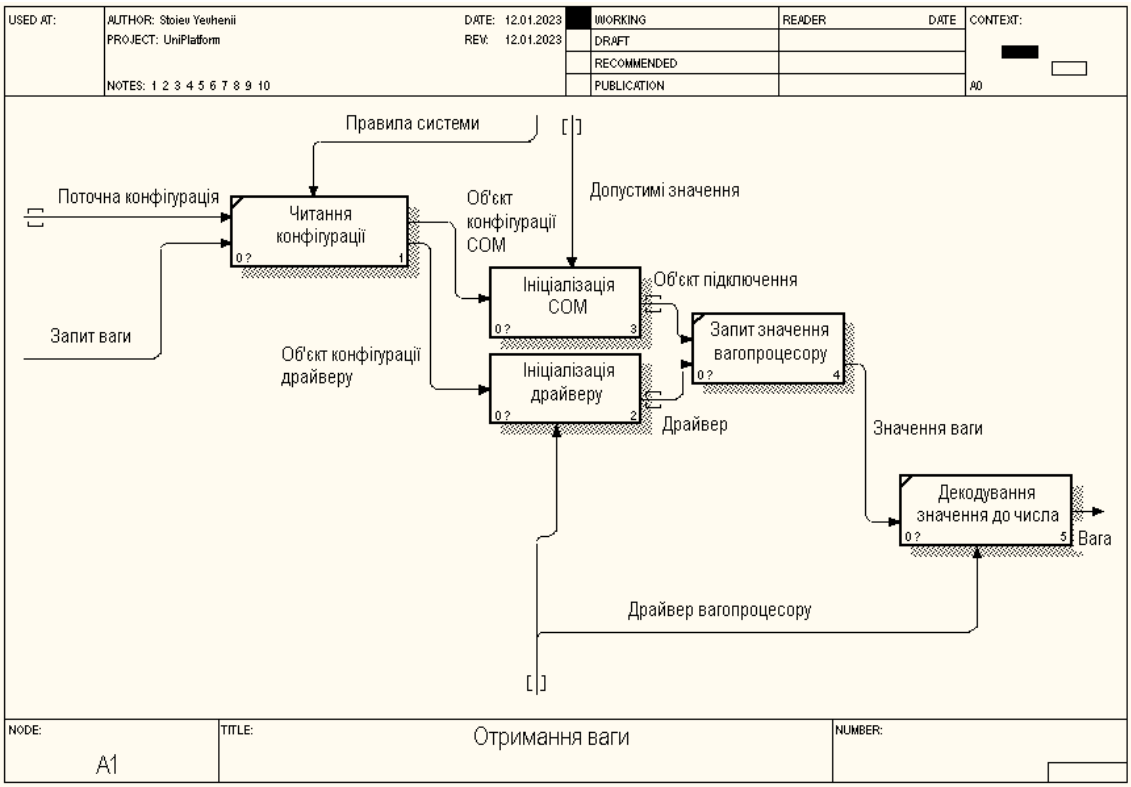

Рисунок 2.2 – Декомпозиція процесу отримання ваги

Як бачимо з декомпозиції, процес отримання ваги складається з 5 етапів:

1. отримання конфігурації для драйверу та *COM* порту;
2. ініціалізація *COM*;
3. ініціалізація драйверу;
4. запит значення через драйвер;
5. декодування отриманого значення у число за допомогою драйверу.

Усі конфігурації зберігаються у базі даних. У свою чергу, конфігурація драйверу містить його назву. При зчитувані цього значення система, за допомогою процесу рефлексії, проводить пошук необхідної збірки у директорії з драйверами. У разі, відсутності необхідного драйверу – система повідомить користувача про відсутність необхідної збірки.

Детально процес пошуку та повідомлення у разі помилки зображено на рис. 2.3.

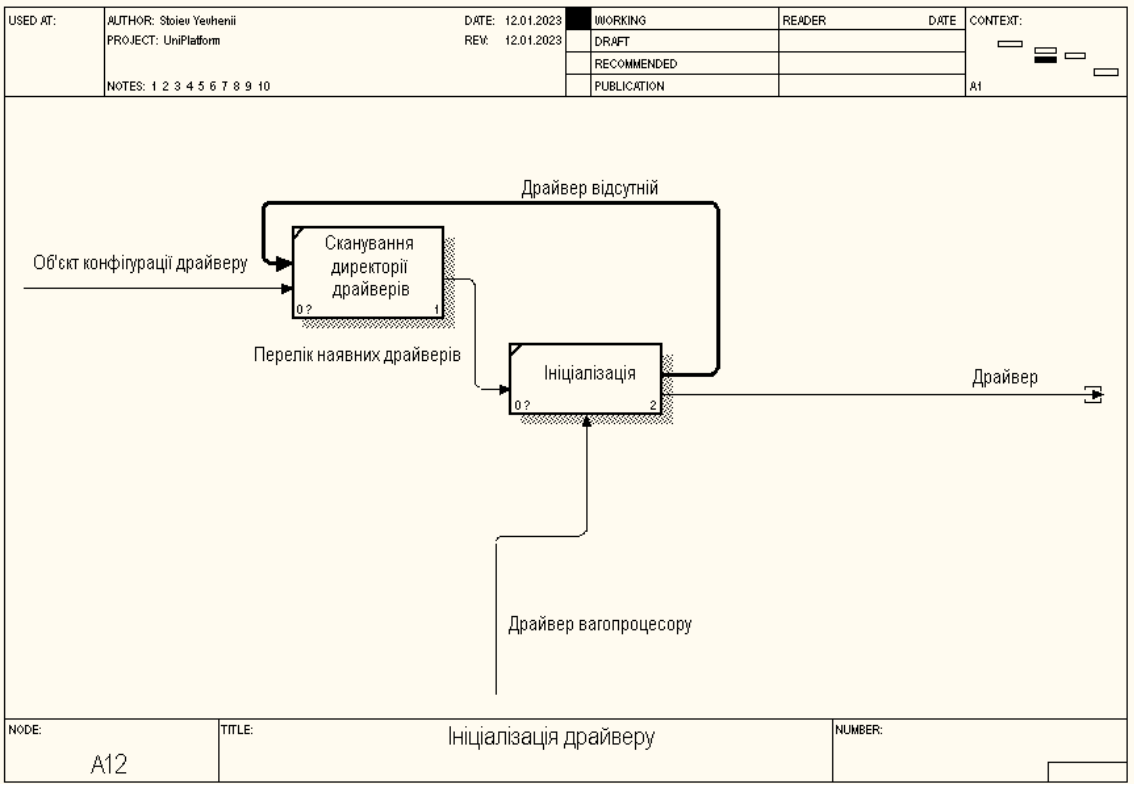

Рисунок 2.3 – Ініціалізація драйверу

Якщо драйвер знайдено – за допомогою рефлексії зберігається значення запиту у локальну змінну. Наступний крок – підключення до вагопроцесору для отримання значення за допомогою збереженого значення запиту.

Після отримання повідомлення від вагопроцесору, відбувається запит до драйверу для декодування отриманого значення у цілочисельне значення. Даний процес відбувається постійно до моменту зупинки усього програмного комплексу або зміни конфігурації. Зміна конфігурації запускає процес перезавантаження сервісу.

За подібною логікою працює сервіс пошуку та ініціалізації *COM* портів.

Сервіс сканує доступні у системі порти та повертає перелік знайдених об’єктів. Система отримує дані про конфігурацію обраного об’єкту, та перевіряє коректність введених даних. У разі помилки – повертає повідомлення про помилку.

Якщо помилка відсутня – встановлює зв’язок з обраним портом, та передає об’єкт до сервісу отримання ваги.

Опис ініціалізації *COM* порту відображено на рисунку 2.4.

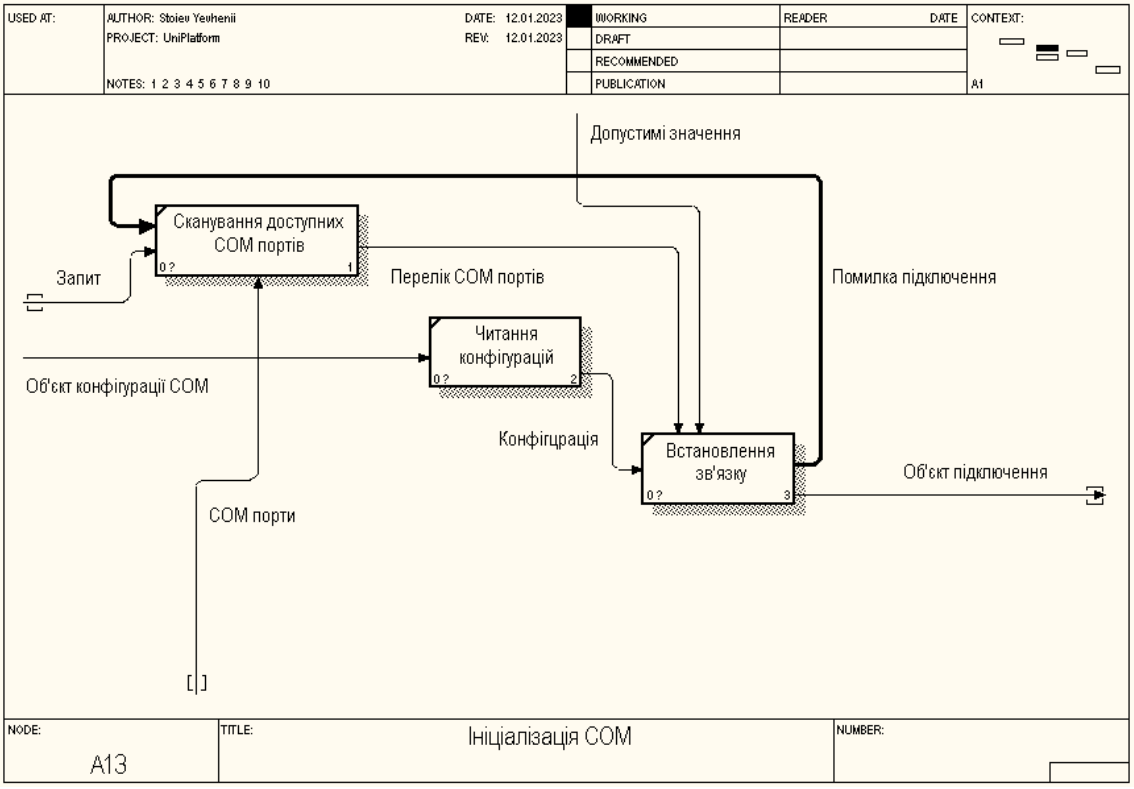

Рисунок 2.4 – Ініціалізація COM порту

Як бачимо, процес ініціалізації компонентів досить схожий за своєю логікою.

Розглянемо сервіс зміни конфігурації програмного комплексу. Сам сервіс досить простий. Функціональне рішення, так само, як і сервіси ініціалізації, зчитує існуючу конфігурацію та повертає об’єкт конфігурації. Під час редагування та збереження конфігурації, система перевіряє нові значення, у разі помилки в одному із змінених полів – повертається повідомлення про помилку конфігурації. Усі значення, які є специфічними, наприклад: параметри налаштування деяких значень *COM* порту, у базі даних не зберігаються. Значення отримуються або за допомогою вбудованих бібліотек *.NET 7,* або безпосередньо з пристрою, на якому працює проект.

Процес редагування конфігурації відображено на рисунку 2.5.

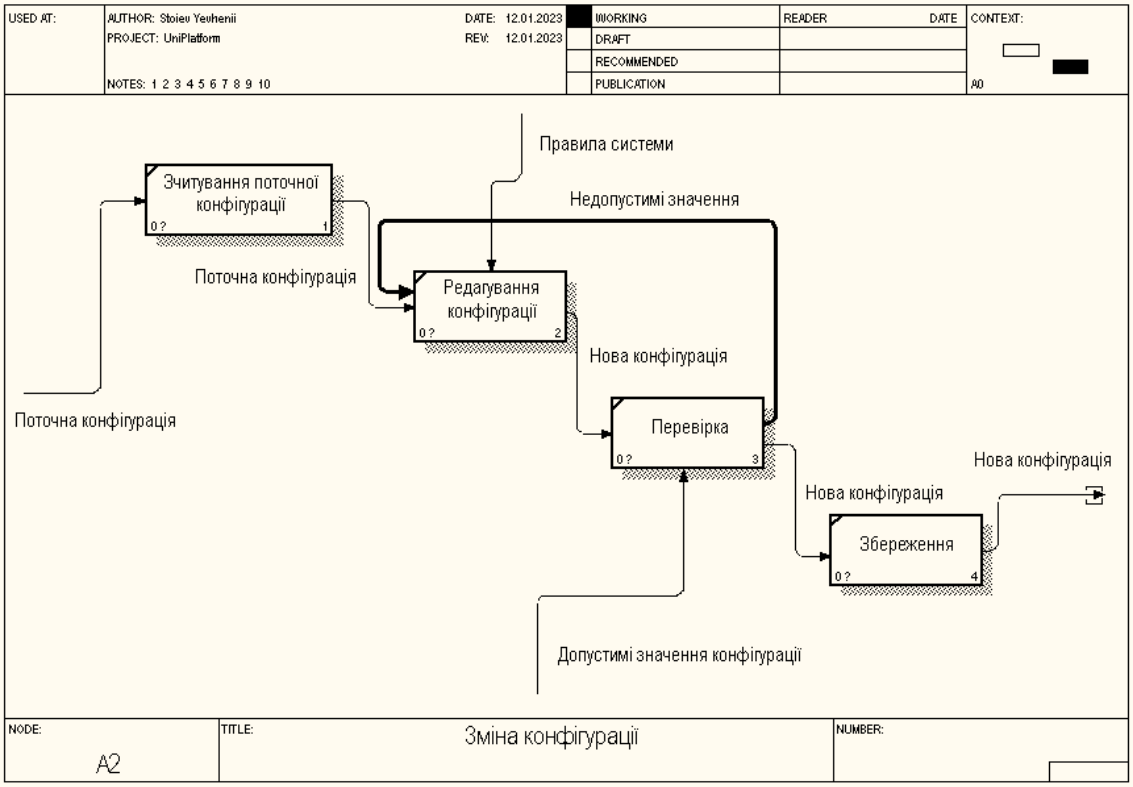

Рисунок 2.5 – Зміна конфігурації програмного комплексу

2.2.Інформаційна модель

Інформаційна модель використовується розробниками програмного забезпечення та дизайнерами вебсайтів для створення ефективної платформи, легкої у використанні та навігації. Якщо інженеру чи дизайнеру не вдається побудувати інформаційну модель, багато користувачів побачать, що вебсайту чи програмі бракує інтуїтивно зрозумілих функцій, а навігація може бути неакуратною, що викликає у користувачів розчарування. Більшість цих моделей побудовано в ієрархії, з головним доменом угорі та більш глибокими доменами внизу. Інженери повинні спланувати, що користувач хоче від програми чи вебсайту, щоб зробити їх ефективними.

Інженери-програмісти та дизайнери вебсайтів можуть почати з нуля та створити програму чи вебсайт без жодного плану чи моделі. Однак такий підхід, швидше за все, призведе до помилок як під час розробки, так і під час використання кінцевого продукту. Як правило, якщо перед створенням продукту не

використовується жодна інформаційна модель, веб-сайтом або програмою буде важко користуватися.

На діаграмі «Бізнес-сервіс/інформація» показано інформацію, необхідну для підтримки однієї чи кількох бізнес-служб. Діаграма «Бізнес-сервіс/інформація» показує, які дані споживає або створює бізнес-сервіс, а також може вказувати джерело інформації.

Діаграма «Бізнес-сервіс/інформація» показує початкове представлення інформації, наявної в архітектурі, і, отже, формує основу для розробки й уточнення на етапі С (архітектура даних) [7].

2.2.1. Діаграма використання

Діаграма варіантів використання в уніфікованій мові моделювання (*UML*) – це тип статичної структурної діаграми, яка представляє взаємодію користувача з системою. Ця діаграма зображує різні типи користувачів і різні способи їх взаємодії з системою. В *UML* користувач називається актором, і він може бути людиною або зовнішньою системою. Зв'язки між актором і варіантом використання називаються асоціаціями. Варіанти використання можуть бути узагальненими, розширеними та включеними. Розглянемо основні варіанти використання універсального програмного комплексу системи контролю ваги. На рис. 2.6 наведено діаграму основних варіантів використання.

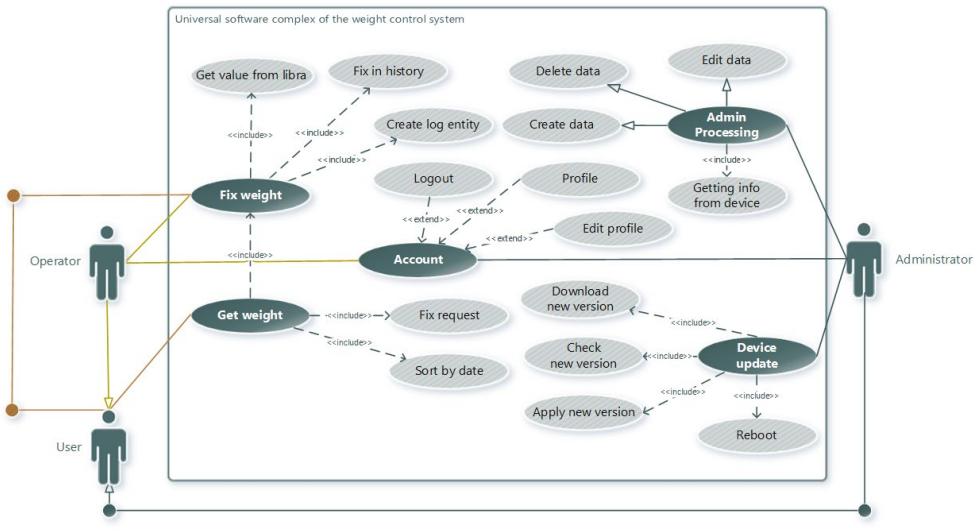

Рисунок 2.6 – Діаграма основних варіантів використання

Як бачимо з діаграми використання у системі представлено три актори:

– Користувач;
– оператор;
– адміністратор;

З найменшими права доступу – користувач, з найважчими – адміністратор. Усі актори в системі, наслідують права доступу користувача, який у свою чергу має можливість лише отримати останню ваги з автоматичною фіксацією. Доступ до налаштувань має лише адміністратор системи [8].

2.2.2. Діаграма розгортання

Діаграма розгортання – це спосіб ілюстрації апаратного та програмного забезпечення системи. Це допомагає візуалізувати процесори, вузли та підключені пристрої. У UML моделюванні ці діаграми служать чудовим способом опису часу роботи вузлів обробки та вказують їхні деталі для цілей побудови. Розглянемо детально процес розгортання проекту. На рис. 2.7 наведено діаграму розгортання програмного комплексу.

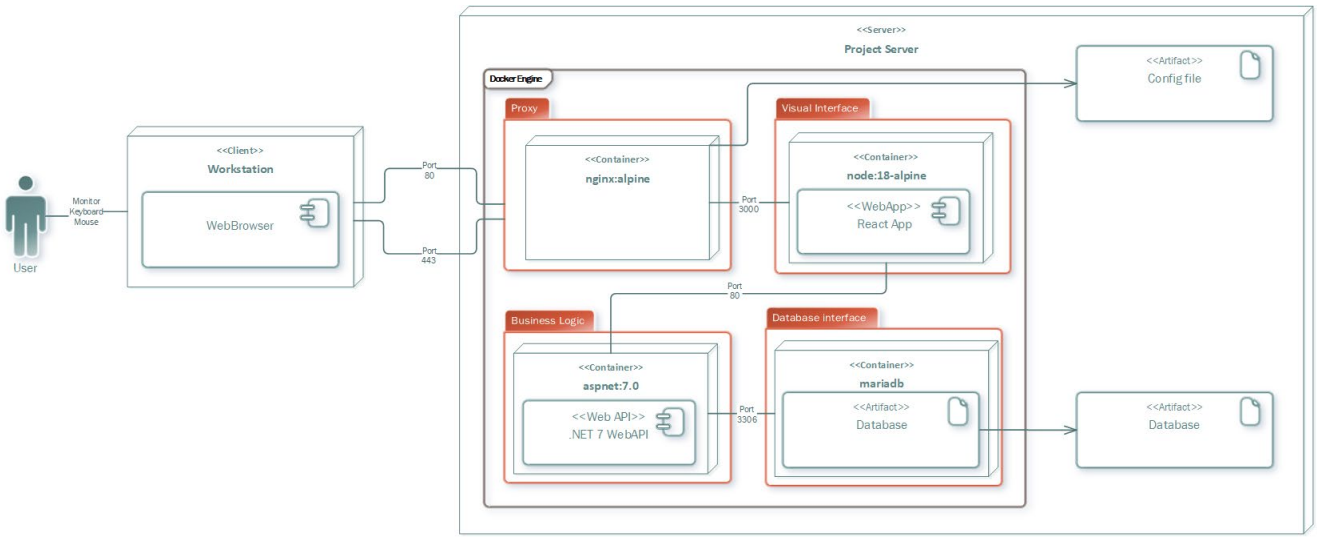

Рисунок 2.7 – Діаграма розгортання програмного комплексу

Для оптимізації роботи програмного комплексу і підвищення безпеки усіх компонентів, проект розгортається за допомогою *Docker*. Усі компоненти, такі як: *proxy*, вебінтерфейс, *WebAPI* та база даних розгортаються, як окремі контейнери з підключенням лише необхідних модулів та відкриттям необхідних, для роботи

проекту, портів. Усі важливі файли, наприклад: база даних та файл конфігурації *proxy* серверу, зберігаються безпосередньо на самому сервері. Для комфортної та продуктивної роботи з усіма модулями використовується додатковий компонент – *docker-compose*.

Дана модель розгортання допомагає вирішити питання супроводжуваності, так як ми не залежимо від компонентів на самому сервері і конфігурація проекту відбувається за допомогою одного конфігураційного файлу. Вирішується питання безпеки – усі контейнери незалежні та ізольовані один від одного. Усі контейнери об'єднані однією мережею [9].

2.2.3. Діаграма послідовності

Діаграма послідовності – це найпоширеніший вид діаграми взаємодії, який зосереджується на обміні повідомленнями між кількома лініями життя.

Діаграма послідовності описує взаємодію, зосереджуючись на послідовності повідомлень, якими обмінюються, разом із відповідними специфікаціями появи на лініях життя.

Розглянемо діаграму послідовності для найважливішого функціонального модулю системи – отримання ваги. На рис. 2.9 наведено діаграму послідовності для отримання ваги.

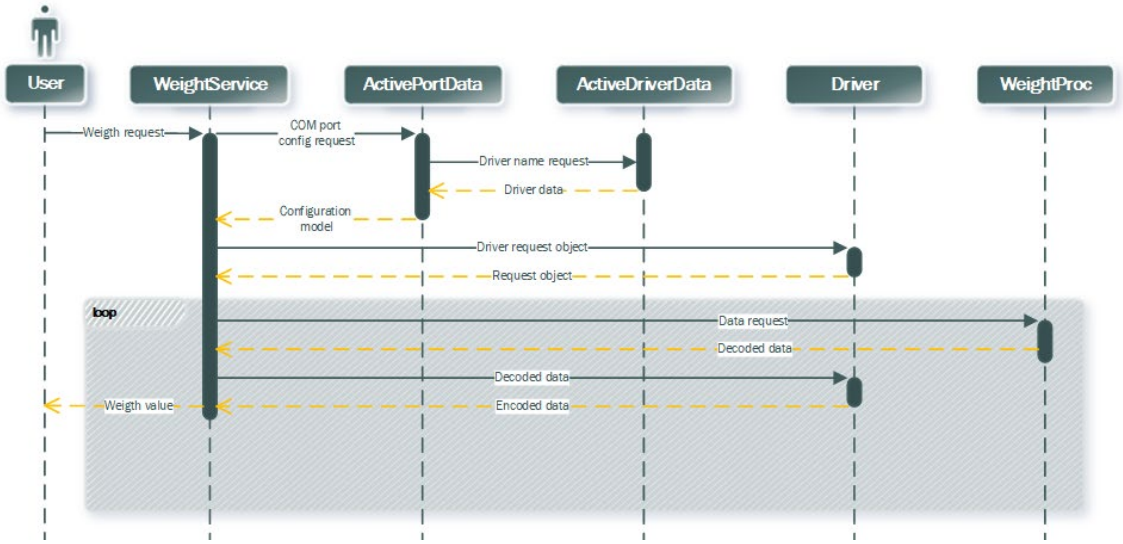

Рисунок 2.8 – Діаграма послідовності для отримання ваги

При запуску системи відбувається отримання збережених конфігурацій для налаштувань драйверу та *COM*-порту. Після отримання необхідних конфігурацій, відбувається процес підключення безпосередньо до вагопроцесору для отримання ваги. Після отримання даних, відбувається декодування даних до десяткового числа і відправка клієнту. Слід зауважити, що підключення до вагів відбувається у режимі прослуховування з періодичною відправкою запиту, якщо необхідно. Клієнтська частина отримує значення, використовуючи технологію *WebSocket*. Дане рішення надає можливість отримувати значення без затримки і навантажень на сервер, що оптимізує роботу всього універсального програмного комплексу [10].

2.2.4. Діаграма класів

Діаграма класів в уніфікованій мові моделювання – це статична структурна діаграма, яка описує структуру системи, показуючи її класи, їхні атрибути, операції (або методи) і зв'язки між об'єктами. Діаграма класів – це план системи або підсистеми. Ви можете використовувати діаграми класів для моделювання об'єктів, які складають систему, показувати зв'язки між об'єктами та описувати ролі цих об'єктів і послуги, які вони надають.

Діаграму класів можна використовувати для відображення класів, зв'язків, інтерфейсу, асоціації та співпраці. Оскільки класи є будівельним блоком програми, яка базується на ООП, діаграма класів має відповідну структуру для представлення класів, успадкування, зв'язків і всього, що ООП має у своєму контексті. Він описує різні типи об'єктів і статичний зв'язок між ними.

Основна мета використання діаграм класів:

- Це єдиний UML, який може належним чином відобразити різні аспекти концепції ООП.
- Правильний дизайн і аналіз додатків можуть бути швидшими та ефективнішими.
- Це основа для розгортання та діаграма компонентів.

На етапі аналізу діаграми класів можуть допомогти зрозуміти вимоги проблемної області та визначити її компоненти. У проектах об’єктно-орієнтованого програмного забезпечення діаграма класів, створена на ранніх стадіях проекту, містить класи, які перетворюються на реальні класи та об’єкти програмного забезпечення під час написання коду.

Розглянемо діаграму класів безпосередньо для проекту, що розробляється і проаналізуємо ключові моменти [11]. На рис. 2.9 наведено загальну діаграму класів.

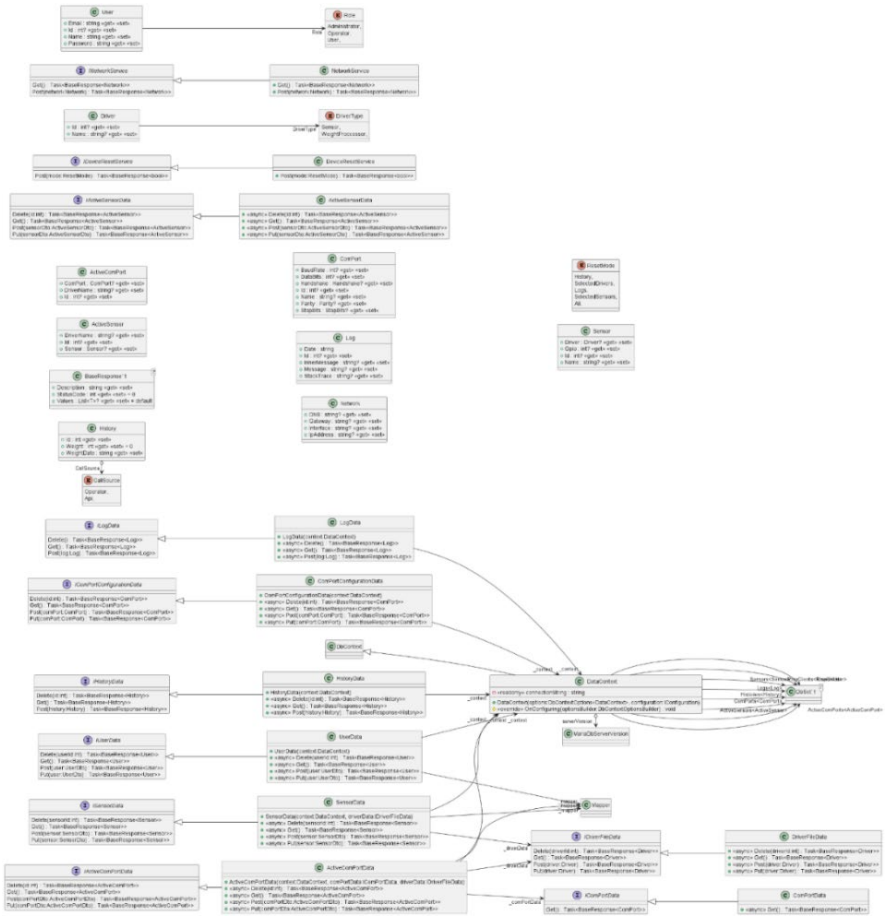

Рисунок 2.9 – Загальна діаграма класів

Як бачимо з діаграми, система побудована за архітектурою сервісів. Тобто, система розділена на сервіси, які відповідають за конкретний функціонал. Кожен сервіс має свій інтерфейс для реалізації, що в свою чергу надає можливість додати декілька сервісів з єдиним інтерфейсом. Усі сервіси, які працюють з базою даних – використовують «ін’єкцію» підключення до БД, що в свою чергу додає гнучкості системі.

Висновки до розділу 2

Побудова архітектури та моделювання будь-якого програмного продукту є одним із пріоритетних етапів створення програмного забезпечення. На цьому етапі конструюється основа всього проекту і невдалі рішення на цьому етапі можуть призвести до появи функціональних обмежень у подальшому, так як внесення змін може бути причиною появи помилок та виключень при експлуатації в реальних умовах. Архітектура програмного забезпечення визначає фундаментальну організацію системи і простіше визначає структуроване рішення. Він визначає, як збираються компоненти програмної системи, їх взаємозв'язок. Він служить основою розробки для команди розробників. Створення архітектури і моделювання програмного комплексу неможливе без діаграм, які описують основні аспекти програмної архітектури.

У другому розділі було розглянуто створення архітектури проєкту, моделювання та проєктування інформаційної та функціональної моделі універсального програмного комплексу системи контролю ваги. Для опису було використано діаграми наступного виду:

- контекстна діаграма;
- діаграма використання;
- діаграма розгортання;
- діаграма послідовності;
- діаграма класів.

Обрані діаграми описують різні аспекти програмного комплексу, що надає можливість детально розглянути архітектурні рішення з декількох сторін.

3 КОДУВАННЯ УНІВЕРСАЛЬНОГО ПРОГРАМНОГО КОМПЛЕКСУ СИСТЕМИ КОНТРОЛЮ ВАГИ

Універсальний програмний комплекс складається з трьох основних частин, а саме: сервіс отримання ваги від вагопроцесору, вебінтерфейс для конфігурації системи та серверна частина з бізнес логікою. Як зазначалось в розділі 1, однією з переваг програмного комплексу є масштабованість, що передбачає можливість швидкого додавання нового функціоналу, тому для реалізації проекту було обрано мікросервісну архітектуру. Мікросервіс – це процес реалізації сервіс-орієнтованої архітектури (*SOA*) шляхом поділу всього додатка на сукупність взаємопов'язаних сервісів, де кожна послуга буде обслуговувати тільки одну бізнес-потребу. У сервіс-орієнтованій архітектурі цілі програмні пакети будуть розділені на невеликі, взаємопов'язані бізнес-одиниці [9].

3.1. Кодування вебінтерфейсу програмного комплексу

3.1.1. Структура проекту

Розпочнемо огляд системи зі структури проекту. На рис. 3.1 зображено структуру проекту вебінтерфейсу.

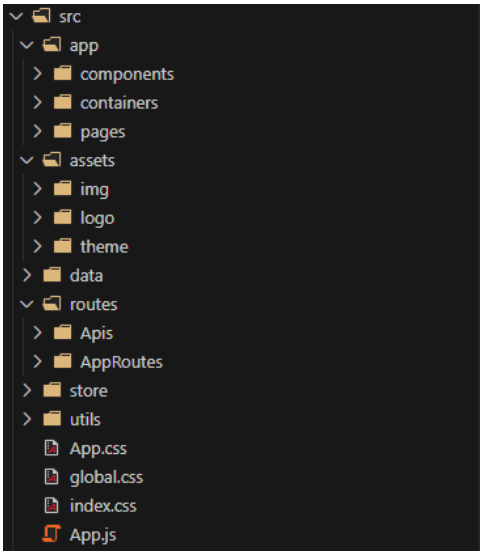

Рисунок 3.1 – Структура проекту вебінтерфейсу

Директорія *app* містить основні компоненти системи, а саме: основні сторінки, багаторазові компоненти, контейнери. Останні використовуються для сторінок, які розділяються на декілька підкатегорій, наприклад, сторінка конфігурації.

Директорія *assets* використовує для зберігання зображень, логотипів та опису теми вебінтерфейсу.

Директорі *data* призначена для зберігання статичних даних ,які використовуються в компонентах.

Архітектура вебінтерфейсу передбачає використання універсальних компонентів для полегшення супроводу програмного коду. Даний підхід дозволяє змінювати налаштування в одному компонентів і ці зміни будуть виконані по всій системі, де використовується цей компонент.

Директорія *routes* призначена для зберігання строкових значень посилань програмних інтерфейсів серверної частини та посилань внутрішніх сторінок.

Директорія *utils* використовується для розміщення допоміжних сервісів, наприклад, сервіс формування елементів для «випадаючих» списків.

Кореневим елементом для відображення проекту слугує функціональний елемент *App.js*. Використовуються стилі верхнього рівня, або *root* стилі. Псевдоклас root CSS відповідає кореневому елементу дерева, що представляє документ. У *HTML* :*root* представляє *html* елемент і ідентичний селектору *html*, за винятком того, що його специфіка вища [10]. Ключова відмінність від звичайних стилів полягає в тому, шо *root* стилі доступні для використання в будь-якому іншому файлі стилів проекту у вигляді змінних, що надає можливість гнучкого налаштування візуальних елементів усієї системи.

Розглянемо основні компоненти вебінтерфейсу універсального програмного комплексу системи контролю ваги.

3.1.2. Компоненти вебінтерфейсу

Як зазначалось в попередньому розділі, кореневим елементом є функціональний елемент *App.js*. Розглянемо кодову базу компонента, яка зображена на рис. 3.2.

```
function mainContent() {
  return (
    <ThemeProvider theme={BrandThemeDark}>
      <Pivot aria-label="Pivot" overflowBehavior={"menu"}>
        {AppRoutes && AppRoutes.map(({ name, element, visibleFor }) => {
          return (
            <PivotItem headerText={name} style={{ position: "absolute", height: "calc(100% - 116px)", width: "100%" }}>
              <ThemeProvider theme={BrandTheme} style={{ height: "100%", backgroundColor: "#f0f0f0" }}>
                {element}
              </ThemeProvider>
            </PivotItem>
          )
        })}
      </Pivot>
    </ThemeProvider>
  )
}

export const App = () => {
  return (
    <>
      <DynamicLayout
        isMobileView={
          <>
            {mainContent()}
            <Header isMobile={true} />
          </>
        }
        defaultView={
          <>
            <Header />
            {mainContent()}
          </>
        }
      />
    </>
  );
};
```

Рисунок 3.2 – Кодова база кореневого компонента

Так, як вебінтерфейс повинен підтримувати роботу на пристроях різного формфактору, то було створено компонент, який аналізує ширину екрану пристрою і в залежності від значення повертає необхідний компонент. Незалежно від формфактору, викликається функція з розміткою, яка в свою чергу отримує дані з файлу, які містить визначення усіх необхідних сторінок проекту і за допомогою циклу створює елементи для переходу і відображення сторінки.

Передбачена рольова система, яка визначає хто з користувачів може бачити сторінку.

Розглянемо структуру об’єкту для визначення основних сторінок проекту.

На рис. 3.3 зображено структуру об’єкту для формування моделі основних сторінок проекту.

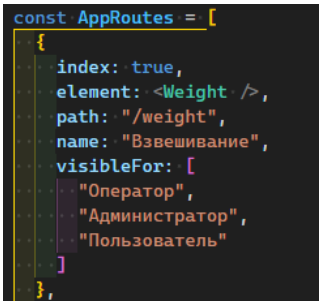

Рисунок 3.3 – Модель основних сторінок

Перелік сторінок визначається за допомогою масива. Кожен елемент містить поле з визначенням самої сторінку, яку необхідно повертатися під час звертання, текстове значення посилання і перелік ролей в системі, які мають можливість перейти на сторінку. На рис. 3.4 зображено зовнішній вигляд сторінки зважування.

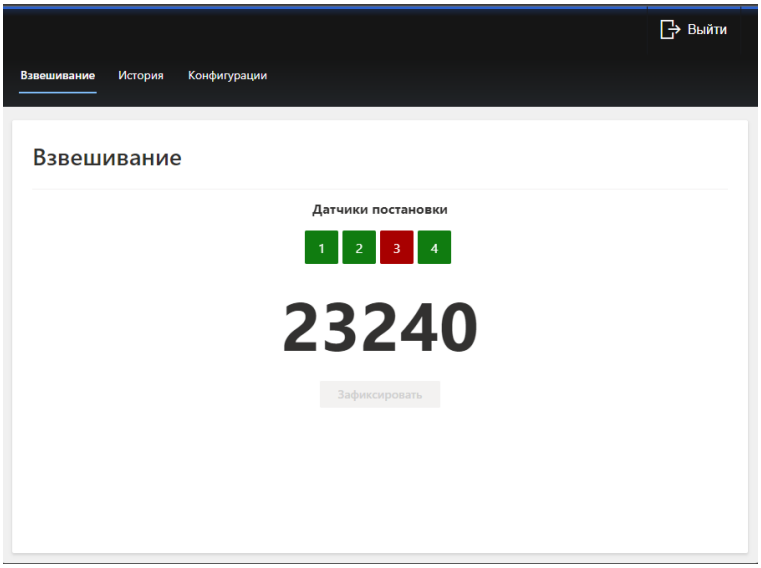

Рисунок 3.5 – Сторінка зважування

При додаванні нового елементу до масиву сторінок, новий елемент буде додано автоматично на головну сторінку.

За структурою, кожна сторінка має основне відображення і допоміжну панель. За бажанням, допоміжну панель можна приховати, вказавши відповідний параметр при створенні компоненти. На рис. 3.5 наведено приклад сторінки без допоміжної панелі. Розглянемо приклад з використанням цього допоміжного елементу.

На рис. 3.6 зображено сторінку з історією зважування. Розглянемо її більш детально.

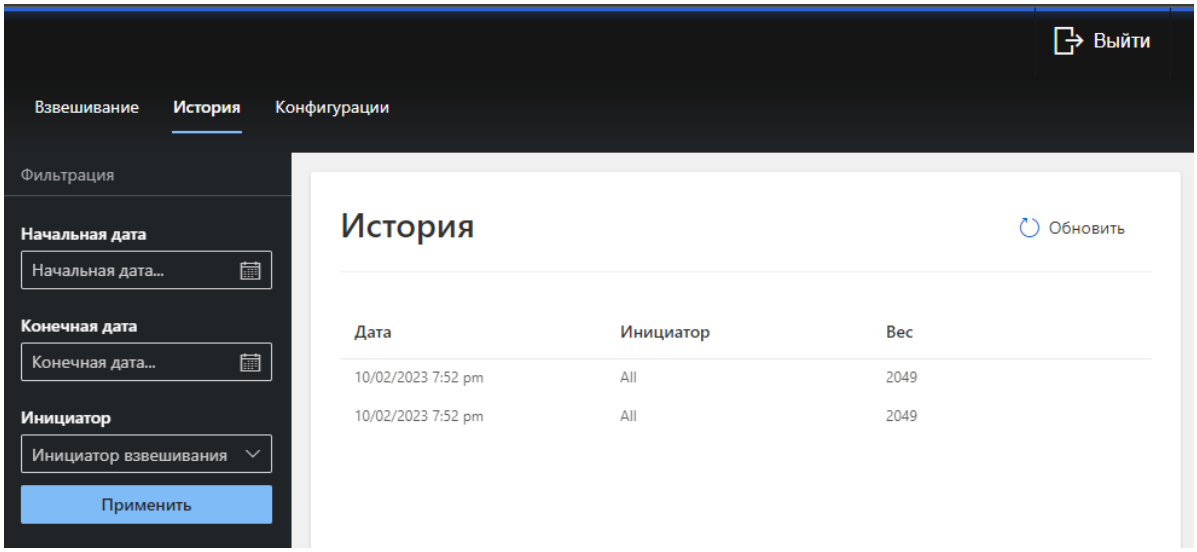

Рисунок 3.6 – Сторінка історії зважування

Сторінка історії містить допоміжну панель, яка використовується для розміщення інструментів, наприклад фільтрів. Усі дані в проекті відображаються у вигляді таблиці, а на мобільних пристроях у вигляді карток. Приклад наведено на рис. 3.7.

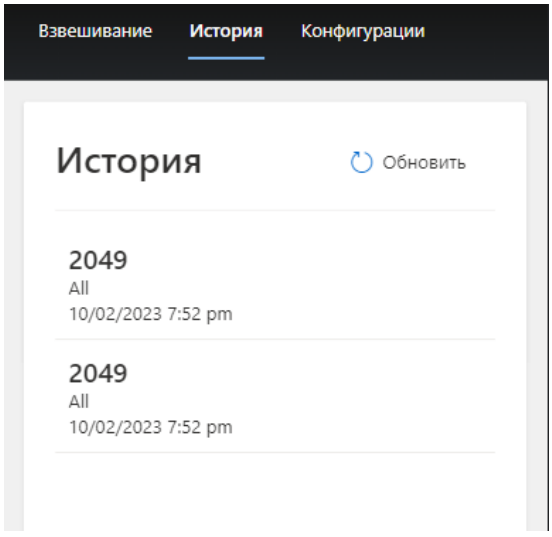

Рисунок 3.7 – Мобільне представлення даних

Кожна сторінка має однакове розміщення елементів для уніфікації користувацького інтерфейсу. Кожен заголовок сторінки містить основні доступні функції, на сторінці історії представлена можливість оновлення даних.

Для динамічного відображення необхідного компоненту було створено власний компонент, на рис 3.8 наведено кодову базу елементу.

```
function renderData() {
    if (props.items) {
        return (
            <>
                {
                    props.items.length < 1
                        ?
                        <EmptyState />
                        :
                        <DynamicLayout
                            isMobileView={
                                <List items={props.items} onRenderCell={onRenderCell} />
                            }
                            defaultView={
                                <DetailsList
                                    checkboxVisibility={props.checkboxVisibility}
                                    columns={props.columns}
                                    compact
                                    enableUpdateAnimations
                                    items={props.items}
                                    layoutMode={DetailsListLayoutMode.fixedColumns}
                                    selection={props.selection}
                                    selectionMode={SelectionMode.multiple}
                                />
                            }
                        />
                }
            </>
        )
    } else {
        return (<EmptyState />)
    }
}
```

Рисунок 3.8 – Компонент динамічного представлення даних

Компонент має декілька перевірок даних. Перша перевірка – перевірка на наявність даних для відображення. У разі відсутності записів – повертається компонент для відображення порожнього стану. У коді представлено дві перевірки на порожні дані, через те, що у разі виключення повертається об’єкт, зі значення *NULL* у полі даних. Якщо, під час другої перевірки, кількість даних більше нуля, то формується елемент для відображення записів.

Для розгляду усіх можливостей основних компонентів, розглянемо сторінку конфігурації системи.

Як зазначалось раніше, сторінка конфігурації є контейнером і містить декілька додаткових сторінок. Тому, допоміжна панель використовується, для навігації між сторінками конфігурації.

Розглянемо сторінки з найширшими функціональним можливостями. Розпочнемо з сторінки додавання нових датчиків та сенсорів.

На рис. 3.9 наведено зображення сторінки додавання нових датчиків та сенсорів.

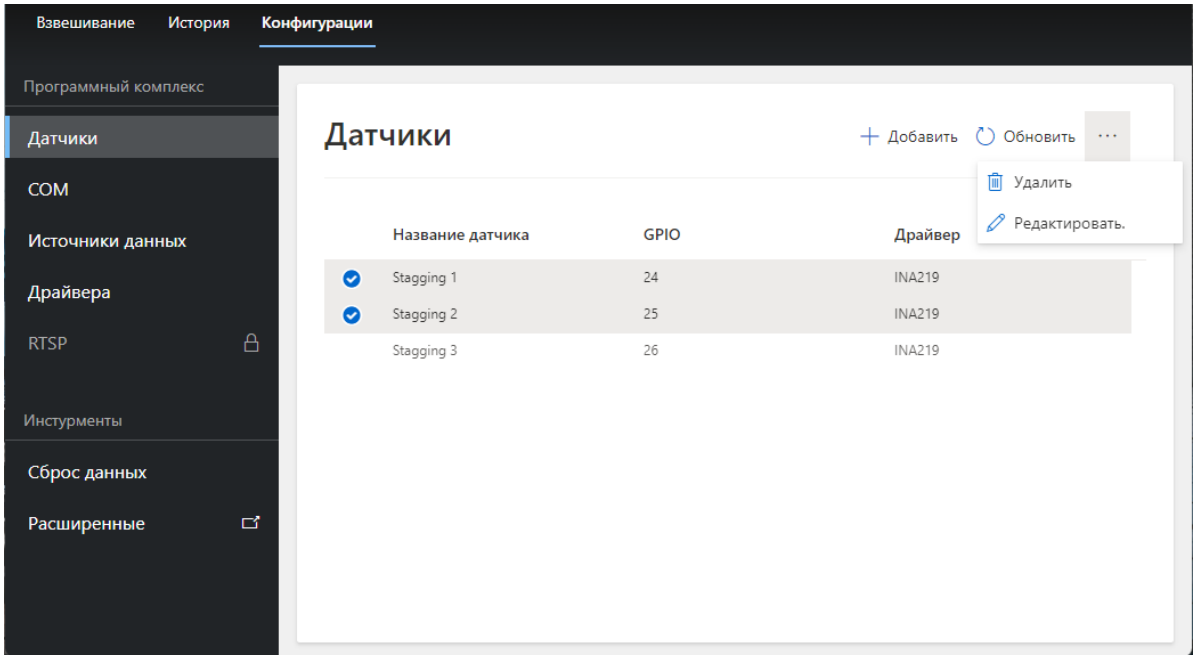

Рисунок 3.9 – Сторінка додавання датчиків

Сторінка додавання містить основні та допоміжні команди для роботи з даними. Основні команди доступні відразу, допоміжні заховані. Як і у попередньому прикладі, дані представлено у представлені таблиці. Для видалення достатньо обрати необхідні записи, підтримується виділення декількох записів, перейти до допоміжних інструментів та обрати функцію видалення. Під час видалення необхідне підтвердження користувача. Приклад підтвердження відображено на рис. 3.10.

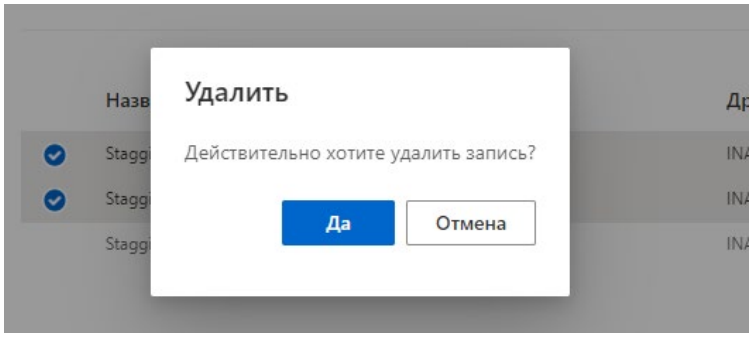

Рисунок 3.10 – Підтвердження видалення

Інтерфейс видалення записів має єдиний вигляд по всій системі.

Додавання нових записів відбувається з використанням модального вікна з відповідними полями. Усі дані, які потребують лише вибору користувачем необхідного варіанту, завантажено шляхом виклику необхідного програмного інтерфейсу на стороні серверу. Розглянемо приклад додавання нового датчика. Вікно додавання нового запису зображено на рис. 3.11.

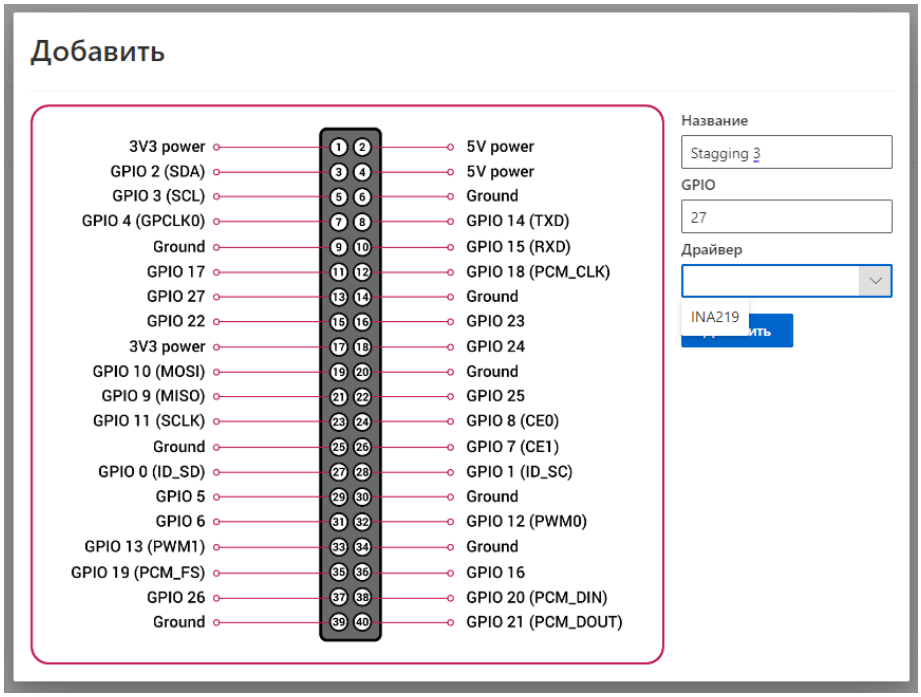

Рисунок 3.11 – Додавання нового запису

Під час додавання нового датчику, необхідно вказати його назву, номер інтерфейсу вводу/виводу, для зручності додано зображення схеми контактів і драйвер для взаємодії.

Усі драйвера розділено на групи. Якщо додається новий датчик, то в переліку буд відображено лише драйвера для датчиків, та ж сама логіка для додавання підключення до вагопроцесору. Кодову базу розділення розглянемо більш детально під час аналізу серверної частини.

Якщо для конфігурації використовується мобільний пристрій, то допоміжна панель не відображена за замовченням.

Приклад мобільної сторінки конфігурації наведено на рис. 3.12

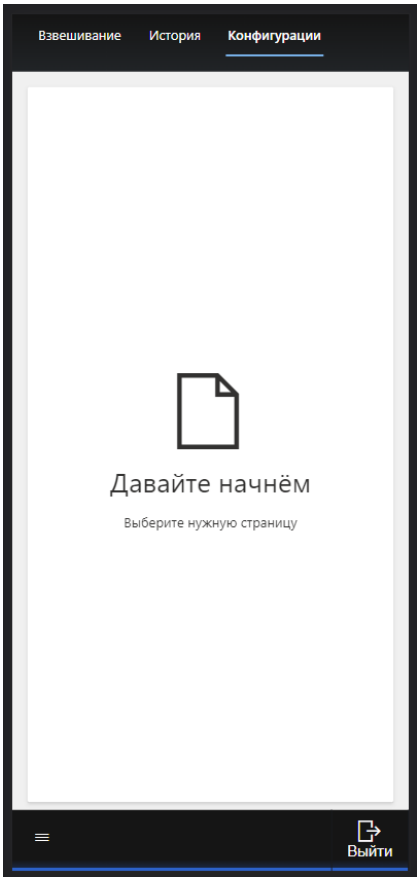

Рисунок 3.12 –Мобільна сторінка конфігурації

У мобільній версії, допоміжна панель, за замовчення, схована. Для виклику необхідно натиснути кнопку виклику панелі навігації. При першому відкриті конфігурацій жодна із сторінок не обрана. При виклику навігаційної панелі та виборі необхідної сторінки – панель автоматично ховається. Вигляд навігаційної панелі наведено на рис. 3.13.

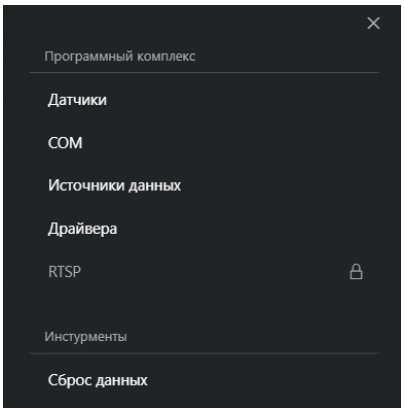

Рисунок 3.13 – Мобільна навігаційна панель

Мобільна версія меню відкривається у повноекранному режимі.

3.2. Кодування серверної частини програмного комплексу

Використання мікросервісів дозволяє створювати архітектуру таким чином, що додавання нового функціоналу не впливає на роботу вже існуючих можливостей. Такий підхід скорочує загальну кількість місць в які необхідно вносити зміни.

3.2.1. Структура проекту

Розпочнемо огляд серверної частини програмного комплексу зі структури проекту, яка наведена на рис. 3.14.

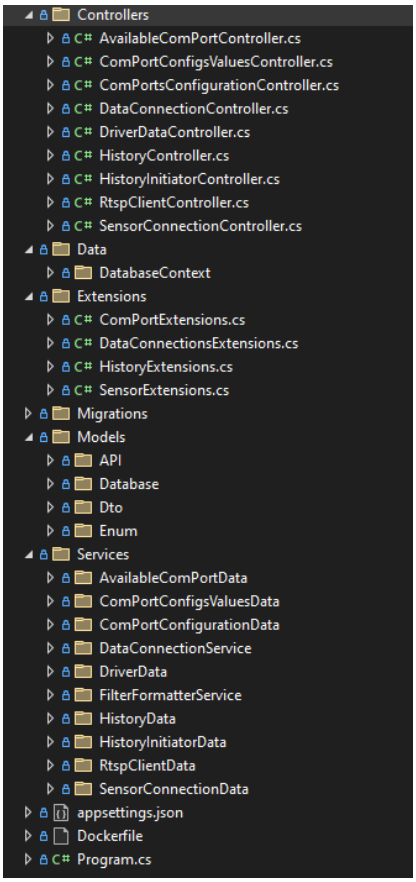

Рисунок 3.14 – Структура проекту серверу

Проект містить наступні директорії:

- *Controllers*: використовується для розміщення програмних інтерфейсів;
- *Data*: містить класи, які використовуються для отримання даних. В нашому прикладі містить директорію для підключення до бази даних;

– *Extensions*: містить класи розширення для моделей програмного комплексу;
– *Migrations*: системна директорія де зберігаються міграції для бази даних;
– *Models*: зберігаються моделі системи. Усі моделі розподілені на додаткові директорії, які групують класи за напрямом. Наприклад, моделі, які використовуються при роботі з *Application Programming Interface*, базою даних, перечислення та *Data transfer object* моделі.
– *Services*: основна директорія для зберігання сервісів.

Розглянемо приклади з кожної із директорій.

3.2.2. Контролери програмних інтерфейсів

Розпочнемо з директорії контролерів. На рис. 3.15 зображено кодову частину контролеру для зберігання конфігурацій *COM* портів.

```
[Route(template: "api/[controller]")]
[ApiController]
1 reference
public class ComPortsConfigurationController : ControllerBase
{
    private readonly IComPortConfigurationData _comPortConfigurationData;

    0 references
    public ComPortsConfigurationController(IComPortConfigurationData comPortConfigurationData)
    {
        _comPortConfigurationData = comPortConfigurationData;
    }

    [HttpDelete]
    0 references
    public async Task<BaseResponse<ComPort>> Delete(List<int> ids) => await _comPortConfigurationData.Delete(ids);

    [HttpDelete(template: "delete-all")]
    0 references
    public async Task<BaseResponse<ComPort>> Delete() => await _comPortConfigurationData.Delete();

    [HttpGet]
    0 references
    public async Task<BaseResponse<ComPortDto>> Get() => await _comPortConfigurationData.Get();

    [HttpPost]
    0 references
    public async Task<BaseResponse<ComPort>> Post(ComPort comPort) => await _comPortConfigurationData.Post(comPort);

    [HttpPut]
    0 references
    public async Task<BaseResponse<ComPort>> Put(ComPort comPort) => await _comPortConfigurationData.Put(comPort);
}
```

Рисунок 3.15 – Контролер конфігурацій COM портів

Кожен контролер має визначення посилання для доступу. Використовується значення за замовченням – «*api*/[ім'я контролеру]». Для полегшення роботи з сервісами використовується *Dependency injection*. *.NET* підтримує шаблон проектування програмного забезпечення для ін'єкцій залежностей, який є технікою для досягнення інверсії контролю між класами та їх залежностями. Введення залежностей у *.NET* є вбудованою частиною фреймворку, а також конфігурацією,

журналюванням та шаблоном параметрів [11]. Такий підхід дозволяє нам позбутися залежностей від конкретних класів, що надає можливість більш гнучко конфігурувати бізнес-логіку.

До основних вимог REST API відносяться:

- Єдиний вигляд кінцевих точок;
- принцип єдиної залежності;
- єдина модель відповіді.

Говорячи простою мовою, кінцева точка *API* – це точка входу в канал зв'язку, коли взаємодіють дві системи. Це стосується точок дотику зв'язку між *API* та сервером. Кінцеву точку можна розглядати як засіб, за допомогою якого *API* може отримати доступ до ресурсів, необхідних їм із сервера для виконання свого завдання [12]. Виходячи з основних вимог, було додано кінцеві точки для кожної з дій з відповідним *HTTP* запитом. *HTTP* – це протокол передачі гіпертексту. Він регулює структуру і мову запитів і відповідей, які відбуваються між клієнтами і серверами [13].

Архітектура контролерів побудована таким чином, що відсутня залежність від конкретних реалізацій, що надає можливість змінювати кодову базу реалізації, не змінюючи при цьому кодову базу контролеру.

Усі методи є асинхронними, з єдиною моделлю відповіді. Задача асинхронної моделі програмування забезпечує абстракцію над асинхронним кодом. Ви пишете код як послідовність висловлювань, як і завжди. Ви можете прочитати цей код так, ніби кожне твердження завершується до початку наступного. Компілятор виконує багато перетворень, оскільки деякі з цих тверджень можуть почати роботу та повернути завдання, яке представляє поточну роботу.

Метою є включити код, який читається як послідовність операторів, але виконується в набагато більш складному порядку на основі розподілу зовнішніх ресурсів і при виконанні завдань [14].

Єдина модель відповіді надає можливість простої взаємодії між клієнтською частиною та сервером.

Структура моделі відповіді наведена на рис. 3.16

Рисунок 3.16 – Модель відповіді програмних інтерфейсів

Модель містить три поля для зберігання значень:

– *Description*: слугує для зберігання текстового результату;
– *StatusCode*: використовується для зберігання *HTTP* коду операції;
– *Values*: зберігає основні дані.

Values визначено, як *Generic*. Загальні класи інкапсулюють операції, які не є специфічними для певного типу даних. Найбільш поширеним використанням для загальних класів є такі колекції, як зв'язані списки, хеш-таблиці, стеки, черги, дерева тощо. Такі операції, як додавання та видалення елементів із збору, виконуються в основному однаково незалежно від типу даних, що зберігаються [15]. Обраний підхід надає можливість зберігати будь-який тип даних у полі *Values*. Описана архітектура контролера використовується в реалізаціях усіх інших програмних інтерфейсах.

3.2.3. Модель підключення до бази даних

Розглянемо структуру файлу підключення до БД, яка зображена на рис. 3.17.

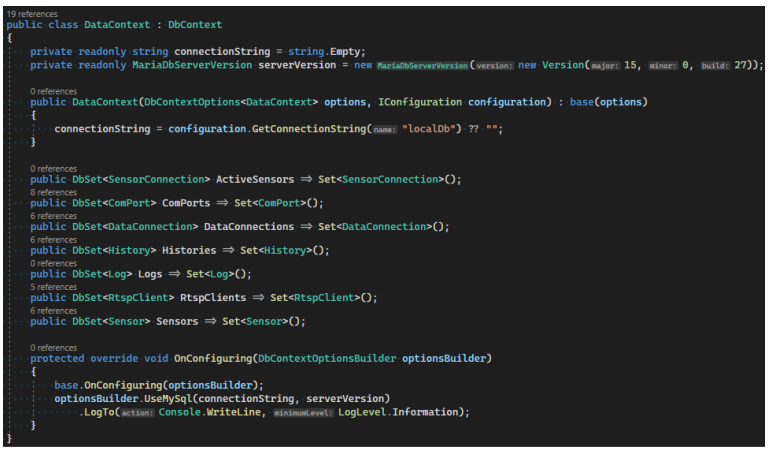

Рисунок 3.17 – Модель підключення до бази даних

Для роботи з базою даних використовується *EntityFramework*, який надає єдине *API* для взаємодії з різними СКБД. Модель наслідує клас *DbContext* і

перевизначає метод *OnConfiguring*. Строкові значення підключення зберігаються в файлі *appsettings.json.* Кожна модель таблиці бази даних реалізована у вигляді поля, яка має лише метод для встановлення відповідної моделі програмного комплексу.

Метод *OnConfiguring* відповідає за встановлення підключення до бази даних та конфігурування журналювання рівня повідомлень при взаємодії з базою даних.

Для того, щоб модель доступу до бази даних була доступна будь-якому сервісу програмного комплексу, вона є частиною *DI*.

3.2.4. Методи розширення

Методи розширення дозволяють "додавати" методи до існуючих типів без створення нового похідного коду, перекомпіляції або іншої зміни вихідного коду. Методи розширення є статичними методами, але вони називаються так, ніби вони є прикладними методами розширеного типу. Для клієнтського коду, написаного на *C#, F#* та *Visual Basic*, немає очевидної різниці між викликом методу розширення та методами, визначеними в типі.

Найбільш поширеними методами розширень є оператори запитів стандарту LINQ, які додають функціональність запиту до існуючих типів *System.Collections.IEnumerable* і *System.Collections.Generic.IEnumerable<T>*. Щоб використовувати стандартні оператори запитів, спочатку приведіть їх в область дії з директивою. Тоді будь-який тип, який реалізує *IEnumerable<T>*, схоже, має такі методи екземплярів, як *GroupBy, OrderBy, Average* тощо [16].

В реалізації програмного комплексу, методи розширення використовуються для перетворення моделі до *DTO*. Об'єкт передачі даних *(DTO)* – це об'єкт, який переносить дані між процесами. Використовується для полегшення зв'язку між двома системами без потенційного розкриття конфіденційної інформації.

На рис. 3.18 зображено реалізацію методу розширення універсального програмного комплексу контролю ваги.

```
0 references
public static class ComPortExtensions
{
    1 reference
    public static ComPortDto ToDto(this ComPort comPort) => new ComPortDto()
    {
        BaudRate = comPort.BaudRate,
        DataBits = comPort.DataBits,
        Handshake = comPort.Handshake == null ? "N/A" : Enum.GetName(enumType: typeof(Handshake), value: comPort.Handshake),
        Id = comPort.Id,
        Name = comPort.Name,
        Parity = comPort.Parity == null ? "N/A" : Enum.GetName(enumType: typeof(Parity), value: comPort.Parity),
        StopBits = comPort.StopBits == null ? "N/A" : Enum.GetName(enumType: typeof(StopBits), value: comPort.StopBits),
    };
}
```

Рисунок 3.18 – Метод розширення для перетворення у *DTO*.

Будь-який метод розширення є статичним і першим параметром приймає модель, методи якої доповнює. На прикладі наведено кодову базу методу для перетворення моделі конфігурації *COM* порту до об’єкту передачі даних. Методи розширення доступні з будь-якої частини системи.

3.2.5. Міграції бази даних

У реальних проектах моделі даних змінюються в міру реалізації функцій: додаються і видаляються нові сутності або властивості, а схеми баз даних потрібно відповідно змінювати, щоб вони синхронізувалися з додатком. Функція міграції в *EF Core* надає спосіб поступового оновлення схеми бази даних, щоб синхронізувати її з моделлю даних програми, зберігаючи наявні дані в базі даних.

Коли вводиться зміна моделі даних, розробник використовує інструменти *EF Core* для додавання відповідної міграції з описом оновлень, необхідних для синхронізації схеми бази даних. *EF Core* порівнює поточну модель зі знімком старої моделі, щоб визначити відмінності, і генерує вихідні файли міграції; Файли можна відстежувати у вихідному контролі проекту, як і будь-який інший вихідний файл.

Після створення нової міграції її можна застосувати до бази даних різними способами. *EF Core* записує всі застосовані міграції в спеціальну таблицю історії, дозволяючи їй знати, які міграції були застосовані, а які ні [18].

За структурою, міграції місять два методи, а саме:

– *Down*: відміна внесених змін;

– *Up*: оновлення бази даних.

Виклик необхідного методу відбувається за допомогою відповідної консольної команди. Приклад файлу міграції наведено на рис. 3.19.

Рисунок 3.19 – Файл міграції

3.2.6. Сервіси програмного комплексу

Вся бізнес-логіка реалізована у вигляді окремих сервісів, які є незалежними один від одного. Зміни в одному не впливають на логіку роботи іншого, хоча сервіси можуть взаємодіяти один з одним через інтерфейси. Розглянемо декілька прикладів реалізації.

Розпочнемо з сервісу отримання доступних *COM* портів, який містить лише один метод *GET*.

На рис. 3.20 наведено приклад сервісу для отримання переліку доступних у системі *COM* портів.

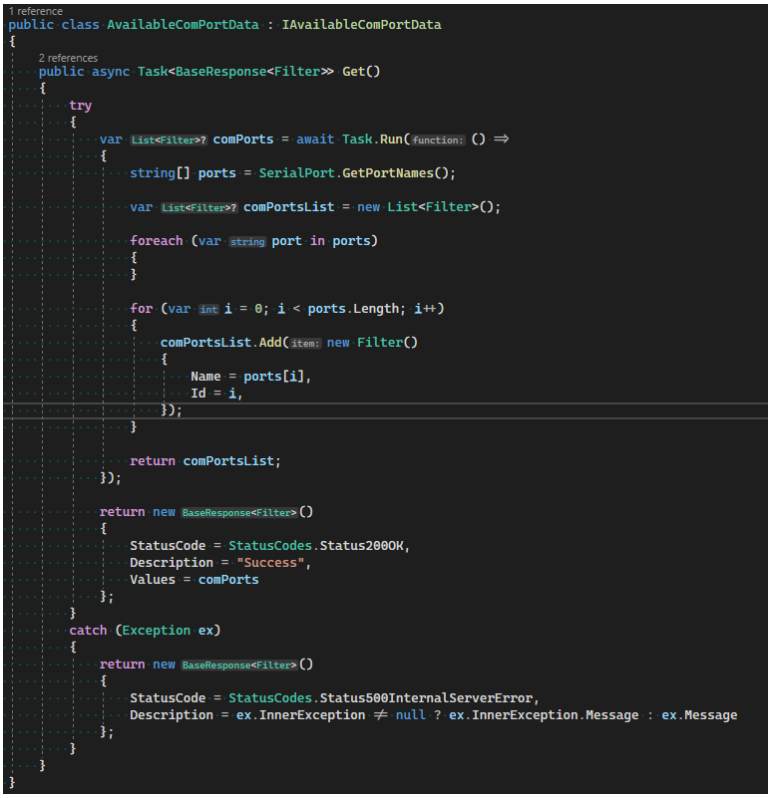

Рисунок 3.20 – Сервіс отримання переліку COM портів

Сервіс реалізує інтерфейс *IAvailableComPortData* і повертає базову модель з типом значень *Filter*. Бізнес-логіка знаходиться в блоці *try…catch*, у разі виключення буде повернено модель з відповідним повідомленням, та з кодом операції 500, що відповідає внутрішній помилці на стороні сервера. Для оптимізації роботи, операція виконується як окрема задача в окремому потоці. Для отримання імені порту, використовується програмні інтерфейси простору імен *System.IO*.

Простір імен *System.IO* містить типи, які дозволяють читати та записувати файли та потоки даних, а також типи, які забезпечують базову підтримку файлів і каталогів [19]. Даний простір імен є частиною *.NET Standard*. *.NET Standard* – це формальна специфікація *API .NET*, доступних для кількох реалізацій *.NET*. Ідея *.NET Standard* полягала у встановленні більшої однорідності в екосистемі *.NET*. *.NET* 5 і пізніші версії застосовують інший підхід до встановлення однорідності, що виключає необхідність стандарту *.NET* в більшості сценаріїв. Однак, якщо ви

хочете поділитися кодом між *.NET Framework* та будь-якою іншою реалізацією *.NET*, наприклад *.NET Core*, ваша бібліотека має бути націлена на *.NET Standard* 2.0. Нові версії *.NET Standard* не будуть випущені, але *.NET* 5, *.NET* 6 і всі майбутні версії продовжать підтримувати *.NET Standard* 2.1 і більш ранніх версій.

Завдяки такому підходу, сервіс отримання доступних *COM* портів буде коректно працювати на всіх існуючих операційних системах для серверів, персональних комп'ютерів та інших.

Розглянемо більш складну реалізацію сервісу з прикладом взаємодії між різними сервісами. Для прикладу було обрано сервіс додавання нової конфігурації джерела інформації. На рис. 3.21 наведено кодову базу сервісу.

```
2 references
public class DataConnectionsService : IDataConnectionsService
{
    private readonly IAvailableComPortData _comPortData;
    private readonly DataContext _context;
    private readonly IDriverData _driverData;

    0 references
    public DataConnectionsService(DataContext context, IAvailableComPortData comPortData, IDriverData driverData)...

    2 references
    public async Task<BaseResponse<DataConnection>> Delete(List<int> ids)...

    2 references
    public async Task<BaseResponse<DataConnection>> Delete()...

    2 references
    public async Task<BaseResponse<DataConectionDto>> Get()...

    2 references
    public async Task<BaseResponse<DataConnection>> Post(DataConnectionRequest request)...

    2 references
    public async Task<BaseResponse<DataConnection>> Put(DataConnectionRequest request)...
}
```

Рисунок 3.21 – Сервіс додавання джерела інформації

Сервіс взаємодіє одразу з трьома сервісами, а саме: сервіс взаємодії з базою даних, сервіс отримання конфігурації *COM* портів, сервіс отримання драйверів для підключення до джерела інформації.

Як зазначалось раніше, взаємодія відбувається не шляхом створення об'єкту конкретного класу, а створенням об'єкту типу інтерфейсу, завдяки чому відсутня прив'язка до конкретної реалізації.

Ініціалізація моделі кожного сервісу відбувається в конструкторі, моделі мають приватний модифікатор доступу з можливістю запису значення лише в тілі конструктору.

Розглянемо метод для отримання записів з бази даних. На рис. 3.22 наведено кодову базу методу.

```
public async Task<BaseResponse<DataConectionDto>> Get()
{
    try
    {
        var List<DataConnection>? activeDataConnections = await _context.DataConnections
        .Include(navigationPropertyPath: DataConnection comPort => comPort.ComPort)
        .ToListAsync();

        var BaseResponse<Driver>? driversList = await _driverData.Post(driverRequest: new Driver() { DriverType = DriverType.DataSource });

        if (driversList.Values == null)
        {
            return new BaseResponse<DataConectionDto>()
            {
                StatusCode = StatusCodes.Status200OK,
                Description = "Drivers list is empty"
            };
        }

        var List<DataConectionDto>? responseValues = new List<DataConectionDto>();

        foreach (var DataConnection record in activeDataConnections)
        {
            var DataConectionDto? dto = record.ToDto();
            dto.Driver = driversList.Values.Find(match: Driver d => d.Id == record.DriverId);

            responseValues.Add(item: dto);
        }

        return new BaseResponse<DataConectionDto>()
        {
            StatusCode = StatusCodes.Status200OK,
            Description = "Success",
            Values = responseValues
        };
    }
    catch (Exception ex)
    {
        return new BaseResponse<DataConectionDto>()
        {
            StatusCode = StatusCodes.Status500InternalServerError,
            Description = ex.InnerException != null ? ex.InnerException.Message : ex.Message
        };
    }
}
```

Рисунок 3.22 – Метод отримання записів з бази даних.

Усі методи сервісів знаходяться в блоці *try…catch*, для стабільності роботи програмного комплексу. Першим кроком є отримання усіх записів з бази даних. Тут слід зауважити, що записи джерел основної інформації містять зовнішній ключ на таблицю з конфігураціями *COM* портів. За замовченням, *Entity Framework* не отримує дані із зовнішніх таблиць з ціллю запобігання рекурсій у записах, тому необхідно власноруч додати завантаження даних шляхом виклику методу *Include*() з вказанням необхідної моделі даних.

Так, як драйвера не зберігаються у базі даних, то запис джерела основної інформації містить лише ідентифікатор драйверу. Після отримання списку джерел інформації, необхідно отримати специфікацію обраного драйверу. Для цього використовується сервісу роботи з драйверами з вказанням необхідного типу драйверу. Після отримання необхідних даних, перевіряється наявність зазначеного драйверу, у разі відсутності необхідного запису – повертається модель відповіді з відповідною інформацією.

3.2.7. Взаємодія з GPIO

General Purpose Input Output (GPIO) – це набір контактів в мікроконтролері, який функціонує шляхом передачі даних всередину і з плати. Вони служать двонаправленим штифтом, або вхідним, або вихідним штифтом, або альтернативним функціональним штифтом.

При виконанні функції входу він приносить інформацію в плату з пристрою введення в процесор на платі мікроконтролера.

При подачі в якості вихідного штифта дані з плати передаються на вихідний пристрій від процесора.

До популярних прикладів можна віднести миготіння світлодіодів і використання *PUSH*-кнопок. Він також використовується для видачі переривань, зчитування цифрових сигналів, пробудження процесора тощо [21].

Взаємодія з *GPIO* необхідна для отримання інформації з датчиків постановки автомобіля. На рис. 3.22 зображено приклад коду для взаємодії з датчиками.

```
if (sensorActivated)
{
    List<Sensors> sensorsNumberList = Sensors.GetStaggingSensors();
    var sensorsStateResults = new List<bool>();

    foreach (var sensor in sensorsNumberList)
    {
        try
        {
            GpioController controller = new GpioController(PinNumberingScheme.Logical);
            controller.OpenPin(sensor.GPIO, PinMode.InputPullUp);
            var pinValue = controller.Read(sensor.GPIO) == PinValue.High;
            sensorsStateResults.Add(pinValue);
        }
        catch (Exception ex)
        {
            var _logger = new Logger();
            _logger.Log(ex.Message);
            Debug.WriteLine($"GPIO ERROR - {ex.Message}");
            sensorsStateResults.Add(false);
        }
    }

    return sensorsStateResults;
}
```

Рисунок 3.23 – Взаємодія з GPIO

Перший крок – перевірка параметру на активність, тобто використовувати датчики чи ні. У разі позитивного результату необхідно отримати перелік конфігурацій для датчиків. Після формування переліку пристроїв – відбувається підключення до необхідних *GPIO* і зчитування значень.

Висновки до розділу 3

Створення програмного коду є не менш важливим етапом створення програмного комплексу. Від прийнятих програмних рішень залежить масштабованість проекту, його здатність коректно реагувати на зміни та виключення під час виконання в умовах реального використання.

У третьому розділі описано основний функціонал системи, його кодова частина та рекомендації щодо використання. Було розглянуто кодування вебінтерфейсу і серверної частини, проведено огляд структури проектів і основних підходів при створенні кодової бази.

Вебінтерфейс створювався за принципом багаторазових компонентних частин, що дозволило досягти єдиного стилю інтерфейсу на пристроях різного формфактору з можливістю гнучкого налаштування.

Серверна частина створювалась с використанням мікро-сервісної архітектури і дотриманням основних принципів підходу *SOLID*. *SOLID* – це абревіатура від перших п'яти принципів об'єктно-орієнтованого дизайну Роберта С. Мартіна. Ці принципи встановлюють практику, яка дозволяє розробляти програмне забезпечення з урахуванням підтримки та розширення в міру зростання проекту. Прийняття цих практик також може сприяти уникненню рефакторингу коду та гнучкій або адаптивній розробці програмного забезпечення [22].

Обрані програмні рішення дозволили створити кодову базу, яка дозволяє без зайвих зусиль модифікувати програмний комплекс, шляхом додавання нових програмних рішень без необхідності змінювати вже існуючий функціонал системи.

4 ДОСЛІДЖЕННЯ РОЗГОРТАННЯ ПРОГРАМНОГО КОМПЛЕКСУ

Розгортання програмного продукту є останнім етапом розробки будь-якого проекту. Є декілька способів розгортання, а саме:

- Розгортання безпосередньо на сервері;
- розгортати у вигляді *docker* контейнеру.

Використовуючи розгортання безпосередньо на сервері, без використання додаткових інструментів, є досить простим процесом, але такий вид розгортання має певні нюанси і проблемні аспекти, а саме:

1. Потребує попереднього налаштування робочого оточення. Необхідно власноруч встановлювати залежні бібліотеки та *SDK*;
2. необхідно слідкувати за сумісністю системних бібліотек. Оновлення зазвичай відбувається під час оновлення основної системи;
3. проблеми з безпекою. При розгортанні проекту на основній системи, програмне рішення має доступ до будь-якої директорії системи;
4. труднощі при переміщенні проекту на новий сервер. При кожному переміщенні необхідно проводити налаштування робочого оточення з самого початку.

Більш оптимальною альтернативою першому способу є використання *docker* контейнерів.

Docker – це проект з відкритим вихідним кодом для автоматизації розгортання додатків у вигляді портативних, самодостатніх контейнерів, які можуть працювати в хмарі або локально.

Контейнери *Docker* можуть працювати будь-де, локально в центрі обробки даних клієнтів, у зовнішнього постачальника послуг або в хмарі. Контейнери зображень *Docker* можуть працювати на *Linux* і *Windows*. Однак образи *Windows* можуть працювати тільки на хостах *Windows*, а образи Linux можуть працювати на

хостах *Linux* і хостах *Windows* (поки що використовують віртуальну машину *Hyper-V Linux*), де хост означає сервер або віртуальну машину.

Розробники можуть використовувати середовища розробки на *Windows*, *Linux* або *macOS*. На комп'ютері розробника розробник запускає хост *Docker*, де розгортаються зображення *Docker*, включаючи додаток та його залежності. Розробники, які працюють на *Linux* або в *macOS*, використовують хост *Docker*, заснований на *Linux*, і вони можуть створювати образи лише для контейнерів *Linux*. Розробники, які працюють над *Windows*, можуть створювати образи як для Linux, так і для контейнерів *Windows* [23].

Для проведення дослідження розгортання програмного комплексу універсальної системи контролю ваги розглянемо обидва варіанта. Для початку розглянемо компоненти, які необхідні для функціонування програмну комплексу.

4.1.Компоненти розгортання

Для коректного функціонування програмного комплексу необхідно встановити додаткові інструменти, а саме:

- *Nginx* – це програмне забезпечення з відкритим кодом для вебобслуговування, зворотного проксі-сервера, кешування, балансування навантаження, потокового передавання медіа тощо. Він починався як вебсервер, розроблений для максимальної продуктивності та стабільності. На додаток до своїх можливостей *HTTP*-сервера, *NGINX* також може функціонувати як проксі-сервер для електронної пошти (*IMAP*, *POP3* та *SMTP*) та зворотний проксі-сервер і балансувальник навантаження для серверів *HTTP*, *TCP* та *UDP* [24];
- *Node.js* – це кросплатформне, середовище виконання JavaScript з відкритим вихідним кодом, яке виконує код JavaScript за межами веббраузера [25];
- *.NET SDK* – набір бібліотек і інструментів для створення і використання програмних продуктів на платформі *.NET*;

– *MariaDB* – база даних для зберігання конфігураційних даних;

Порівняння способів розпочнемо з першого варіанту – розгортання на серверній операційній системі.

4.2.Розгортання на серверній операційній системі

Порядок встановлення інструментів не має значення, а операційною системою за замовчення є *Linux*.

4.2.1. Інсталяція *proxy* – серверу

На рис. 4.1 зображено команди для встановлення серверу.

```
$ sudo apt update
$ sudo apt install nginx
```

Рисунок 4.1 – Встановлення вебсерверу

Першою командою ми оновлюємо перелік доступних програмних рішень у віддаленому репозиторії, а другою розпочинаємо встановлення необхідного програмного продукту.

Після встановлення, необхідно додати запуск вебсерверу під час завантаження системи. Команду для додавання сервісу вебсерверу наведено на рис. 4.2.

```
$ sudo systemctl enable nginx
```

Рисунок 4.2 – Налаштування автозапуску вебсерверу

Після встановлення і конфігурації сервісів вебсерверу, необхідно провести конфігурацію самого вебсерверу, шляхом редагування файлу, який знаходиться у наступній директорії – */usr/local/nginx/conf*.

За замовченням вебінтерфейс використовує 3000 порт, а програмні інтерфейси – 80 та 443.

Після налаштування *proxy* – серверу, розглянемо встановлення *Node.js*.

4.2.2. Інсталяція серверу вебінтерфейсу

Як зазначалось у першому розділі, вебінтерфейс створено з використанням бібліотеки *ReactJS*. Для функціонування цієї бібліотеки необхідно встановити серверну частину, а саме – *NodeJS*:18. Як і в пункті 4.2.1,в першу чергу необхідно оновити перелік програмного забезпечення, після чого використати команду наведену на рис. 4.3

```
$ sudo apt install nodejs npm
```

Рисунок 4.3 –Встановлення NodeJS

Команда на рис. 4.3 виконує одразу дві дії:

1. Встановлення серверу;
2. встановлення пакетного менеджеру *npm*. Пакетний менеджер за замовченням для *JavaScript*.

4.2.3. Інсталяція .NET

Для встановлення технології *.NET* можна використати два способи, а саме:

1. Встановити за допомогою скрипту, доступного на офіційному сайті корпорації *Microsoft* [26];
2. встановити у ручному режимі.

Розглянемо обидва варіанти. Розпочнемо з встановлення за допомогою скрипту.

Спочатку необхідно завантажити скрипт на сервер. Після чого додати права на виконання. Команда для змін прав наведено на рис. 4.4.

```
Bash

sudo chmod +x ./dotnet-install.sh
```

Рисунок 4.4 – Зміна прав для виконання скрипту

За замовченням, при виконанні скрипту інсталюється остання версія платформи та додаткових залежностей. За бажанням можна вказати необхідну версію. В нашому випадку використовується остання версія платформи.

Цей спосіб найпростіший, адже не потребує додаткових налаштувань.

При встановленні у ручному режимі, можна використати офіційний репозиторій операційної системи. Але у цьому випадку можуть бути певні обмеження. По-перше, при використанні операційних систем на базі *Debian*, ми не маємо можливість інсталювати останню версію платформи, адже додавання актуальних версій відбувається через деякий час після релізу. По-друге, для встановлення платформи необхідно мати процесор лише на архітектурі *x64*, версії для інших архітектур недоступні.

То ж, краще за все завантажити останню версію платформи і провести налаштування власноруч. Після завантаження платформи, необхідно провести налаштування системи. На рис. 4.5 наведено приклад конфігурації.

```bash
DOTNET_FILE=dotnet-sdk-7.0.100-linux-x64.tar.gz
export DOTNET_ROOT=$(pwd)/.dotnet

mkdir -p "$DOTNET_ROOT" && tar zxf "$DOTNET_FILE" -C "$DOTNET_ROOT"

export PATH=$PATH:$DOTNET_ROOT:$DOTNET_ROOT/tools
```

Рисунок 4.5 – Конфігурування системи

Наведенні команди можуть бути запущені з каталогу, де зберігається файл для вилучення середовища виконання. Це зробить команди *.NET CLI* доступними в терміналі та встановить необхідні змінні середовища.

4.2.4. Інсталяція *MariaDB*

Встановлення бази даних для проекту, має аналогічну послідовність, як встановлення proxy – серверу.

Спочатку оновлюємо системні компоненти, після чого інсталюємо базу даних, команду для встановлення наведено на рис. 4.6.

```
sudo apt install mariadb-server
```

Рисунок 4.6 – встановлення бази даних

Після встановлення бази даних необхідно додати сервіс бази даних до автозапуску.

Для нових інсталяцій *MariaDB* наступним кроком є запуск сценарію безпеки. Цей сценарій змінює деякі з менш безпечних параметрів за замовчуванням для

таких речей, як віддалені кореневі логіни та зразки користувачів. Для їх інсталяції необхідно виконати команду зображену на рис. 4.7

```
sudo mysql_secure_installation
```

Рисунок 4.7 –Встановлення модулів безпеки бази даних

Під час інсталювання побачите ряд підказок, де ви можете внести деякі зміни в параметри безпеки *MariaDB*. У першій підказці буде запропоновано ввести актуальний пароль *root* бази даних.

Після налаштувань бази даних, можна перевірити підключення до бази даних. За замовчення використовується порт 3306.

4.3.Розгортання за допомогою *Docker*

Для розгортання проекту за допомогою *docker*, в першу чергу, необхідно провести інсталяцію програмного рішення.

4.3.1. Інсталяція платформи *Docker*

Як і з платформою .NET, є декілька варіантів інсталювання, а саме:

– За допомогою репозиторію системи;

– за допомогою завантаженого пакету;

– за допомогою скрипту.

Інсталювання за допомогою репозиторію системи потребує додавання репозиторію *Docker* до системи. Для цього необхідно виконати команду, представлену на рис. 4.8.

```
sudo mkdir -m 0755 -p /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg

echo \
"deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Рисунок 4.8 – Інсталювання Docker

Результатом виконання команди буде можливість інсталяції *docker* за допомогою команди *apt*.

При інсталяції за допомогою пакетів, перший крок – завантажити пакети з офіційного сайту розробників. Для завантаження необхідно вказати версію продукту та цільову архітекту.

Для коректного функціонування, необхідно завантажити декілька файлів, а саме:

– containerd.io;
– docker-ce;
– docker-ce-cli;
– docker-buildx-plugin;
– docker-compose-plugin.

Без цих пакетів використання *docker* буде неможливе. Для встановлення завантажених пакетів необхідно виконати команду, приклад якої наведено на рис. 4.9.

```
sudo dpkg -i ./containerd.io_<version>_<arch>.deb \
./docker-ce_<version>_<arch>.deb \
./docker-ce-cli_<version>_<arch>.deb \
./docker-buildx-plugin_<version>_<arch>.deb \
./docker-compose-plugin_<version>_<arch>.deb
```

Рисунок 4.9 – Встановлення Docker

Останній доступний спосіб встановлення – встановлення за допомогою скрипта. Перший крок – завантажити скрипт на сервер, за допомогою команди на рис. 4.10.

```
curl -fsSL https://get.docker.com -o get-docker.sh
sudo sh ./get-docker.sh --dry-run
```

Рисунок 4.10 – Встановлення Docker за допомогою скрипта

Після виконання скрипту, *docker* буде інстальовано у систему.

4.3.2. Інсталяція модулю *Docker Compose*

Так, як універсальний програмний комплекс складається з декількох частин, то для зручності розгортання використовуються додаткові інструменти. *Docker Compose* – це інструмент, який дозволяє запускати багатоконтейнерні прикладні середовища на основі визначень, встановлених у файлі YAML. Він використовує

визначення служб для створення повністю конфігурованих середовищ з кількома контейнерами, які можуть спільно використовувати мережі та томи даних.

Для інсталювання необхідно виконати команду, наведену на рис 4.11.

```
curl -SL https://github.com/docker/compose/releases/download/v2.3.3/docker-compose-linux-x86_64 -o ~/.docker/cli-plugins/docker-compose
```

Рисунок 4.11 – Інсталювання інструменту *Docker Compose*

В результаті виконання команди, буде завантажено модуль до директорії *docker*, що надасть можливість викликати інструмент з будь-якої директорії.
Для розгортання проекту за допомогою *docker*, необхідно створити файл з назвою *Dockerfile*. Розглянемо файл розгортання кожного із проектів.

4.3.3. Файл розгортання вебінтерфейсу

Розгортання вебінтерфейсу не потребує додаткових директорій чи модулів, тому структура файлу буде досить простою. Приклад файлу розгортання наведено на рис. 4.12.

```
FROM node:18-alpine

ENV PYTHONUNBUFFERED=1
RUN apk add --update --no-cache python3 alpine-sdk && ln -sf python3 /usr/bin/python
RUN python3 -m ensurepip
RUN pip3 install --no-cache --upgrade pip setuptools

WORKDIR /app
COPY . .
ENV GENERATE_SOURCEMAP=false
RUN npm install
RUN npm run build

RUN npm config set proxy http://uniplatform_backend:80

ENTRYPOINT ["npm", "run","start"]
```

Рисунок 4.12 – Файл розгортання вебінтерфейсу

Перший крок – зазначити, яке оточення буде використовувати контейнер. У нашому випадку, необхідне оточення з встановленим *NodeJS* версії 18 на базу операційної системи *Alpine Linux*.

Наступний крок – інсталяція пакетів *python* у систему для можливості використання пакетного менеджеру *pip*. Вказаний пакетний менеджер необхідний для інсталювання залежностей *ReactJS* проекту.

Після інсталювання необхідних пакетів та залежностей необхідно скопіювати файли вебінтерфейсу до контейнеру, виконати інсталювання залежностей та створити збірку. Останній крок – запуск створеної збірки.

Розглянемо файл розгортання для серверної частини універсального програмного комплексу системи контролю ваги.

4.3.4. Файл розгортання серверної частини

Файл розгортання серверної частини, майже не відрізняється від процесу розгортання вебінтерфейсу. Текст файлу наведено на рис. 4.13.

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:7.0 AS build
WORKDIR /src
COPY ["API.csproj", "."]
RUN dotnet restore "./API.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "API.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "API.csproj" -c Release -o /app/publish /p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "API.dll"]
```

Рисунок 4.13 – Розгортання серверної частини

Файл розгортання *.NET* проекту має ті ж самі кроки, що і файл розгортання *NodeJS*. Після вказання операційної системи нам необхідно інсталювати оточення *.NET* та *SDK*, для коректного виконання проекту. Створити збірку проекту та виконати команду для початку виконання.

4.3.5. Файл розгортання багатоконтейнерного середовища

Як зазначалось у пункті 4.3.2, розгортання проекту відбувається за допомогою *docker compose*. Для використання інструменту, необхідно створити файл з конфігурацією. Так, як файл конфігурації є досить громіздким, розглянемо кожен сервіс окремо.

Розпочнемо огляд з сервісу *proxy* – серверу.

На рис. 4.14 наведено текст створення сервісу для *docker*.

```
nginx:
  image: nginx:alpine
  restart: always
  depends_on:
    - uniplatform_backend
    - uniplatform_frontend
  ports:
    - "80:80"
    - "443:443"
  volumes:
    - ./uniplatform-nginx.conf:/etc/nginx/conf.d/default.conf
```

Рисунок 4.14 – Опис сервісу proxy – серверу

Створення будь-якого сервісу розпочинається із вказачння назви необхідного образу, у нашому випадку – *nginx* останньої версії. Усі інші параметри є додатковими, але їх наявність значно полегшує роботу з сервісом. Розглянемо кожен із них:

- *Restart* – відповідє за перезавантаження сервісу у разі помилки;
- *Depends_on* – гарантує запуск сервісу після зазначених сервісів;
- *Ports* – використовується для перенаправлення внутрішниіх портів сервісу на порти хостової систем, відбувається процес мапінгу;
- *Volumes* – слугує для підключення зазначеної директорії до сервісу.

Так, як контейнер *Nginx* є доступним на платформі *Docker Hub*, яка використовується для публікації контейнерів, то ми можемо не виконувати процес створення збірки, достатньо завантажити вже готовий образ.

Розглянемо процес запуску сервісу, який відповідає за серверну частину проекту.

На рис. 4.15 наведено текст створення сервісу серверної частини.

```
uniplatform_backend:
  restart: always
  build: ../API/
  volumes:
    - /dev:/dev
```

Рисунок 4.15 – Сервіс серверної частини

Параметр *restart* і *volume* було розглянуто вище, параметр *build* використовується для вказання шляху до *Dockerfile*. Після запуску розгортання,

docker compose знаходить файл з назвою *Dockerfile* і виконує інструкції зазначені у файлі. Та ж сама логіка розгортання і вебінтерфейсу. Текст створення сервісу наведено на рис. 4.16.

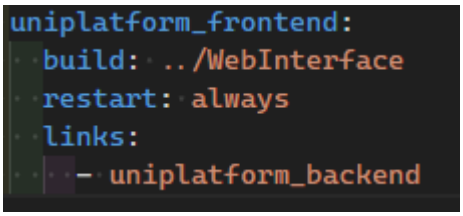

Рисунок 4.16 – Сервіс вебінтерфейсу

Останній елемент системи – база даних. Текст створення сервісу бази даних наведено на рис.4.17.

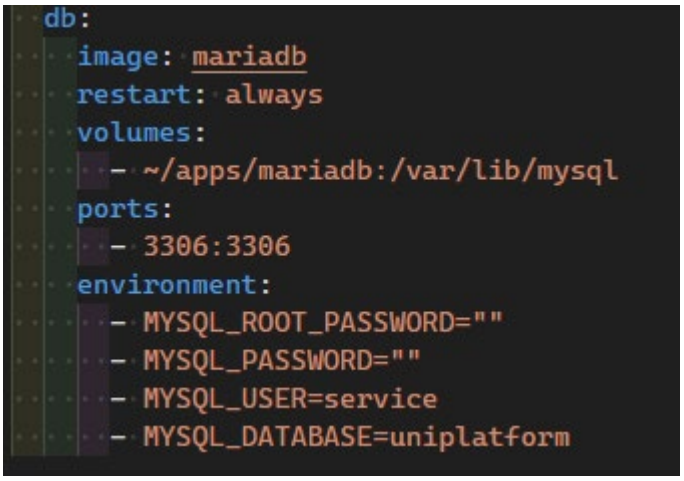

Рисунок 4.17 – Сервіс бази даних

Останній параметр, який не було розглянуто – *environment*. Цей параметр використовується для задання значень змінних оточення. В нашому прикладі, використано зміні для задання користувачів бази даних та назва бази до якої буде автоматичне підключення. Зберігання файлів бази даних відбувається на основній системі, що надає можливість змінювати образ бази даних без втрати записів.

4.4.Порівняння способів розгортання

І перший, і другий варіант надають можливість проведення останнього етапу розробки універсального програмного комплексу системи контролю ваги – розгортання. Для кінцевого користувача немає різниці, як саме було проведено цей етап, але, виконавши дослідження цього питання, можна зробити наступні

висновки – розгортання за допомогою *docker* є більш оптимальний і раціональним варіантом.

Розгортання з використанням *Docker* надає цілий ряд переваг та спрощень при роботі. Перша перевага – це надійність, що у вас така ж установка, як і у вашої команди. Це чудово з кількох причин: є так багато цінності в тому, щоб мати можливість працювати з припущенням, що вся ваша команда використовує однакові налаштування, і це дозволяє запускати сценарії та процеси, які дадуть можливість всій вашій команді розробників виконувати загальні операції за допомогою простої команди.

Друга перевага – за допомогою *Docker* ви можете легко ізолювати та усунути проблеми з навколишнім середовищем у своїй команді, не знаючи, як налаштований сервер.

Третя перевага – за допомогою інфраструктури *Docker* ви можете легко перенести середу (з деякими невеликими змінами утиліт) в систему вибору *CI*. Більшість відомих рішень *CI*, доступних сьогодні, добре інтегруються з *Docker*.

Остання і одна із найважливіших переваг Docker при роботі із промисловим програмним забезпеченням – стабільність. *Docker* заснований на *Linux* і, як такий, має ядро *Linux* у кожному контейнері, незалежно від системи, на якій воно працює. Незважаючи на те, що *Docker* часто оновлюється, середовище залишається стабільним у будь-якій системі або пристрої. Немає необхідності раптово відкочуватися до попереднього оновлення або панікувати через непередбачені проблеми з сумісністю.

Кожна із розглянутих переваг значно спрощує процес супроводу програмного продукту, його підтримки та оновлення. При використанні розгортання за допомогою Docker, ми можемо біти впевненні, що програмний комплекс працює в одному і тому ж середовищі на будь-якому пристрої, використовує одні і ті ж самі версії залежностей та додаткових програмних модулів.

Висновки до розділу 4

Як і бідь-який із розглянутих етапів, процес розгортання є важливою частиною створення програмного продукту. У четвертому розділі було досліджено декілька варіантів розгортання універсального програмного комплексу системи контролю ваги.

Проаналізувавши кожен із варіантів, було зроблено висновок, що розгортання програмного комплексу, шляхом запуску безпосередньо на основній системі не є оптимальним та пріоритетним варіантом через ряд недоліків, а саме:

- Залежність від середовища виконання;
- можливі конфлікти залежностей сервісів, що може призвести до появи помилок та виключень;
- важка переносимість між серверами та пристроями, що потребує повторень інструкції розгортання;
- доступ до будь-якої директорії системи, що порушує правила безпеки.

Використання платформи *Docker* вирішує усі зазначенні проблеми. При використанні цього способу розгортання, ми має можливість гнучко конфігурувати середовище виконання, встановлювати необхідні залежності для кожного контейнеру окремо. Один конфігураційний файл дозволяє виконати розгортання усіх контейнерів на будь-якому пристрої, що надає можливість використовувати декілька апаратних платформ для розгортання, а підключення лише необхідних директорій основної операційної системи дозволяє значно підвищити безпеку використання програмного комплексу.

ВИСНОВКИ

Україна є одним із найважливіших світових виробників зерна. Країна вирощує та експортує переважно пшеницю, кукурудзу та ячмінь. За даними Європейської комісії, на Україну припадає 10% світового ринку пшениці, 15% – кукурудзи, 13% – ячменю. Жодна компанія, яка займається культурами, не може працювати без вагових комплексів та програмних рішень для ведення бухгалтерської діяльності. Нині обмін даними між ваговими комплексами та бухгалтерськими системами відбувається у ручному режимі. Це спричиняє появу помилок, наприклад, використання невірної ваги в системі обліку. Одним із варіантів вирішення цієї проблеми є розробка універсального програмного комплексу (УПК), який надавав би перелік стандартизованих прикладних програмних інтерфейсів для автоматичного запиту ваги. На ринку вагових комплексів представлені програмні системи різної якості. Зазвичай, вони є вендорнозалежними, що унеможливлює використання таких програмних рішень для уніфікації програмних систем з єдиним інтерфейсом доступу.

УПК для отримання ваги є важливою складовою для забезпечення процесу автоматизації типових бізнес-процесів ринку агрокультур. Слід зауважити, що остаточно проблема не вирішена. Створення прототипу універсального програмного комплексу системи контролю ваги є першим етапом для створення повноцінного конкурентоспроможного УПК, який можна буде використовувати в реальному секторі економіці. Створений програмний продукт потребує детального тестування на реальних пристроях і може бути удосконалений в залежності від більш конкретизованих цілей та сценаріїв використання.

Вектором розвитку для програмного комплексу є впровадження підтримки периферійних пристроїв різної групи призначення, підтримка гнучких алгоритмів автоматичної фіксації ваги, підтримка фіксації номерів автомобілів з подальшою фіксацією отриманої інформації.

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАННЯ

1) ASP.NET Applications: вебсайт. URL: https://www.koombea.com/blog/asp-net-applications/ (дата звернення 20.12.2022)
2) What is C#?. Twilio: вебсайт. URL: https://www.twilio.com/docs/glossary/what-is-csharp (дата звернення 21.12.2022)
3) What is MariaDB?. Techtarget: вебсайт. URL: https://www.techtarget.com/searchdatamanagement/definition/MariaDB (дата звернення 22.12.2022)
4) What is Entity Framework?. Entityframeworktutorial: вебсайт. URL: https://www.entityframeworktutorial.net/what-is-entityframework.aspx (дата звернення 22.12.2022)
5) Functional Modelling in object oriented analysis and design - GeeksforGeeks. GeeksforGeeks: вебсайт. URL: https://www.geeksforgeeks.org/functional-modelling-in-object-oriented-analysis-and-design/ (дата звернення 23.12.2022)
6) What is a Data Flow Diagram. Lucidchart: вебсайт. URL: https://www.lucidchart.com/pages/data-flow-diagram (дата звернення 24.12.2022)
7) Artifact: Business Service / Information Diagram. PubsOpengroup: вебсайт. URL: https://pubs.opengroup.org/architecture/togaf90-doc/epf/TOGAF9/workproducts/Business%20Service%20Information%20Diagram_61BA313E.html (дата звернення 24.12.2022)
8) What is Use Case Diagram?. VisualParadigm: вебсайт. URL: https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-use-case-diagram/ (дата звернення 26.12.2022)

9) Microservice Architecture - Introduction: вебсайт. URL: https://www.tutorialspoint.com/microservice_architecture/microservice_architecture_introduction.htm#:~:text=Microservice%20is%20the%20process%20of%20implementing%20Service-oriented%20Architecture,will%20be%20sub-divided%20into%20small%2C%20interconnected%20business%20units. (дата звернення 10.02.2023)
10) :root - CSS: Cascading Style Sheets | MDN: вебсайт. URL: https://developer.mozilla.org/en-US/docs/Web/CSS/:root?retiredLocale=tr (дата звернення 10.02.2023)
11) Dependency injection – .NET | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection (дата звернення 11.02.2023)
12) What is an API Endpoint? | API Endpoint Definition | RapidAPI: вебсайт. URL: https://rapidapi.com/blog/api-glossary/endpoint/#:~:text=In%20simple%20terms%2C%20an%20API%20endpoint%20is%20the,need%20from%20a%20server%20to%20perform%20their%20task. (дата звернення 11.02.2023)
13) What is an API Endpoint? | API Endpoint Definition | RapidAPI: вебсайт. URL: https://kinsta.com/knowledgebase/what-is-an-http-request/ (дата звернення 11.02.2023)
14) Asynchronous programming in C# | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/ (дата звернення 11.02.2023)
15) Generic Classes –C# Programming Guide| Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/generics/generic-classes (дата звернення 11.02.2023)
16) Extension Methods - C# Programming Guide | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/extension-methods (дата звернення 11.02.2023)

17) Data Transfer Object DTO Definition and Usage | Okta: вебсайт. URL: https://www.okta.com/identity-101/dto/ (дата звернення 11.02.2023)
18) Migrations Overview - EF Core | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli (дата звернення 11.02.2023)
19) System.IO Namespace | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/api/system.io?view=net-7.0 (дата звернення 11.02.2023)
20) .NET Standard | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/standard/net-standard?tabs=net-standard-1-0 (дата звернення 11.02.2023)
21) Introduction to GPIO - General Purpose I/O - NerdyElectronics: вебсайт. URL: https://nerdyelectronics.com/introduction-to-gpio/ (дата звернення 11.02.2023)
22) SOLID: The First 5 Principles of Object Oriented Design | DigitalOcean: вебсайт. URL: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design (дата звернення 11.02.2023)
23) What is Docker? | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/container-docker-introduction/docker-defined (дата звернення 12.02.2023)
24) What is Docker? - NGINX: вебсайт. URL: https://www.nginx.com/resources/glossary/nginx/ (дата звернення 12.02.2023)
25) What is Node.js? A beginner's introduction to JavaScript runtime: вебсайт. URL: https://www.educative.io/blog/what-is-nodejs (дата звернення 12.02.2023)
26) Install .NET on Linux without using a package manager - .NET | Microsoft Learn: вебсайт. URL: https://learn.microsoft.com/en-

us/dotnet/core/install/linux-scripted-manual#scripted-install (дата звернення 12.02.2023)

27) How To Install and Use Docker Compose on Ubuntu 22.04 | DigitalOcean: вебсайт. URL: https://www.digitalocean.com/community/tutorials/how-to-install-and-use-docker-compose-on-ubuntu-22-04 (дата звернення 12.02.2023)