

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО”

Навчально-науковий інститут атомної та теплової енергетики

Кафедра цифрових технологій в енергетиці

“До захисту допущено”

Завідувач кафедри ЦТЕ

Наталія АУШЕВА

“_____” _____ 2026 р.

**Дипломна робота
на здобуття ступеня бакалавр**

За освітньою програмою “Цифрові технології в енергетиці”

Спеціальності 122 “Комп’ютерні науки”

на тему: “Корпоративна вебсистема управління робочими змінами та персоналом”

Виконав: студент 4 курсу, групи ТР-25

КАРПЕНКО Дмитро Ігорович

(прізвище, ім’я, по батькові)

_____ (підпис)

Керівник: доцент каф. цифрових технологій в енергетиці
к.ф.-м.н., доцент, ДОНЕЦЬ Андрій Георгійович

(посада, науковий ступінь, вчене звання, прізвище, ім’я, по батькові)

_____ (підпис)

Рецензент: доцент кафедри ТАЕ
к.т.н., доцент СІРИЙ Олександр Анатолійович

(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище, ім’я, по батькові)

_____ (підпис)

Нормоконтролер: асистент Здор К.А.

(посада, ім’я, ПРІЗВИЩЕ)

_____ (підпис)

Засвідчую, що у цій дипломній роботі
немає запозичень з праць інших авторів
без відповідних посилань.

Студент

_____ (підпис)

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ АТОМНОЇ ТА ТЕПЛОВОЇ ЕНЕРГЕТИКИ

Кафедра ЦИФРОВИХ ТЕХНОЛОГІЙ В ЕНЕРГЕТИЦІ

Рівень вищої освіти – перший (бакалаврський)

спеціальність 122 «Комп'ютерні науки»

Освітньо-професійна програма «Цифрові технології в енергетиці»

ЗАТВЕРДЖУЮ

Завідувач кафедри ЦТЕ

Наталія АУШЕВА

«___» _____ 2026 р.

ЗАВДАННЯ

на дипломну роботу студенту

КАРПЕНКУ Дмитру Ігоровичу

(прізвище, ім'я, по батькові)

1. Тема роботи «Корпоративна вебсистема управління робочими змінами та персоналом»

Науковий керівник Донець Андрій Георгійович, доцент

(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «___» _____ 2026р. №

2. Термін подання студентом роботи **08.06.2026 року**

3. Вихідні дані до роботи: мова програмування C#, фреймворк ASP.NET Core, мова програмування JavaScript/TypeScript, бібліотека React, СКБД Microsoft SQL Server, технологія об'єктно-реляційного відображення та міграцій Entity Framework Core, інструмент збирання Vite, CSS-фреймворк Bootstrap 5, механізм автентифікації JWT, бібліотеки FullCalendar, Chart.js та ClosedXML, фреймворк модульного тестування xUnit.

4. Зміст роботи:

- 1) провести аналіз існуючих підходів, методів та алгоритмів для управління робочими змінами та персоналом на підприємствах та визначити вимоги до розроблюваної системи;
- 2) провести аналіз сучасних програмних засобів, технологій та платформ для реалізації вебсистеми;
- 3) аргументувати вибір інструментів, технологій та алгоритмів для реалізації програмного забезпечення вебсистеми;
- 4) спроектувати архітектуру програмного забезпечення вебсистеми;

- 5) розробити програмне забезпечення вебсистеми управління робочими змінами та персоналом.
5. Орієнтовний перелік ілюстративного матеріалу: діаграми прецедентів, що демонструють взаємодію користувачів з системою; схеми бази даних, знімки екрану інтерфейсу користувача, приклади згенерованих звітів.
6. Дата видачі завдання «19» вересня 2025 р.

Календарний план

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів дипломної роботи	Примітка
1	Затвердження теми роботи	19.09.25	Виконано
2	Вивчення та аналіз задачі	20.10.25-20.11.25	Виконано
3	Розробка архітектури та загальної структури системи	01.12.25-07.01.26	Виконано
4	Розробка серверної та клієнтської частин	10.01.26-15.03.26	Виконано
5	Програмна реалізація системи	21.04.26-11.05.26	Виконано
6	Оформлення пояснювальної записки	19.05.26-26.05.26	Виконано
7	Захист програмного продукту	13.05.26	Виконано
8	Передзахист	03.06.26	Виконано
9	Подання готової роботи на кафедру	08.06.26	Виконано
10	Захист	15.06.26	Виконано

Студент

(підпис)

Дмитро КАРПЕНКО

(ім'я, ПРІЗВИЩЕ)

Науковий керівник

(підпис)

Андрій ДОНЕЦЬ

(ім'я, ПРІЗВИЩЕ)

АНОТАЦІЯ

Дипломна робота виконана обсягом XX сторінок, містить XX зображень, XX таблиць, XX джерел у переліку посилань.

Метою роботи є проєктування та розробка спеціалізованої корпоративної вебсистеми, спрямованої на цифрову трансформацію процесів управління трудовими ресурсами підприємства. Створений застосунок автоматизує рутинні процеси розподілу робочого часу, ліквідує проблеми часових накладань робочих змін та забезпечує прозорість підрахунку відпрацьованих годин.

Програмний продукт реалізує рольову модель доступу з поділом на адміністраторів та працівників. Основний функціонал системи включає: виконання CRUD-операцій з робочими змінами через інтерактивний календар, автоматичне блокування часових колізій на рівні бази даних, управління обліковими записами, розсилку глобальних інформаційних повідомлень, а також візуалізацію аналітики та автоматичну генерацію звітів про продуктивність у форматі Excel.

Систему побудовано за архітектурним підходом Single Page Application з дотриманням принципів Clean Architecture. Серверна частина реалізована мовою програмування C# на базі платформи ASP.NET Core, з використанням Entity Framework Core та СУБД MS SQL Server. Взаємодія між клієнтом та сервером здійснюється через REST API.

Клієнтська частина застосунку розроблена мовою TypeScript з використанням бібліотеки React. Глобальна обробка HTTP-запитів покладена на клієнт Axios. Візуальний інтерфейс побудовано за допомогою фреймворку Bootstrap 5, а для інтерактивних елементів застосовано бібліотеки FullCalendar та Chart.js. Надійність серверної бізнес-логіки підтверджена модульним тестуванням за допомогою фреймворку xUnit та ізольованої бази даних в оперативній пам'яті.

Ключові слова: УПРАВЛІННЯ ПЕРСОНАЛОМ, ВЕБСИСТЕМА, ASP.NET CORE, REACT, TYPESCRIPT, CLEAN ARCHITECTURE, REST API, JWT.

ANNOTATION

The diploma thesis consists of XX pages, contains XX figures, XX tables, and XX references.

The purpose of the work is the design and development of a specialized corporate web system aimed at the digital transformation of human resource management processes. The created application automates the routine processes of working time allocation, eliminates the problems of shift overlapping, and ensures the transparency of worked hours calculation.

The software product implements a Role-Based Access Control model dividing users into administrators and employees. The main functionality of the system includes: execution of CRUD operations with work shifts via an interactive calendar, automatic blocking of time collisions at the database level, user account management, distribution of global notification messages, as well as analytics visualization and automatic generation of productivity reports in Excel format.

The system is built on a Single Page Application architectural approach complying with Clean Architecture principles. The backend is implemented in C# programming language based on the ASP.NET Core platform, utilizing Entity Framework Core and MS SQL Server DBMS. Client-server interaction is carried out via a REST API with endpoint protection using stateless JWT authentication combined with secure HTTP-only cookies.

The client side of the application is developed in TypeScript using the React library. Global handling of HTTP requests is managed by the Axios client. The visual interface is built with the Bootstrap 5 framework, while FullCalendar and Chart.js libraries are used for interactive elements. The reliability of the server business logic is confirmed by unit testing using the xUnit framework and an isolated in-memory database.

Key words: HUMAN RESOURCE MANAGEMENT, WEB SYSTEM, ASP.NET CORE, REACT, TYPESCRIPT, CLEAN ARCHITECTURE, REST API, JWT.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ	7
ВСТУП	8
1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ.....	10
1.1 Огляд існуючих рішень та їх недоліки	10
1.2 Постановка задачі розробки.....	11
1.2.1 Функціональні вимоги до системи.....	12
1.2.2 Нефункціональні вимоги.....	13
1.2.3 Обґрунтування доцільності розробки	14
2 МЕТОДИ ТА ЗАСОБИ РОЗРОБКИ.....	15
2.1 Обґрунтування вибору технологічного стеку	15
2.1.1 Вибір платформи серверної частини	15
2.1.2 Вибір бази даних та ORM	16
2.1.3 Вибір технологій клієнтської частини.....	18
2.2 Проєктування архітектури за принципами Clean Architecture	19
2.3 Проєктування бази даних	21
2.4 Проєктування REST API та механізмів авторизації.....	23
3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ	26
3.1 Структура проєкту та розподіл на шари.....	26
3.2 Реалізація серверної логіки	32
3.3 Реалізація клієнтського застосунку.....	34
3.4 Реалізація ключових модулів.....	36
3.4.1 Модуль авторизації та управління доступом	36

3.4.2 Інтерактивний календар та планування змін	39
3.4.3 Модуль статистики та генерації звітів.....	42
4 ТЕСТУВАННЯ ТА ВЗАЄМОДІЯ З СИСТЕМОЮ	45
4.1 Модульне тестування за допомогою xUnit	45
4.2 Опис інтерфейсу.....	47
ВИСНОВКИ.....	53
СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ.....	55
ДОДАТОК А.....	57
ДОДАТОК Б	67

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

API — Application Programming Interface — прикладний програмний інтерфейс; набір інструментів та функцій, що дозволяє різним програмним компонентам взаємодіяти між собою.

CRUD — Create, Read, Update, Delete — акронім, що позначає чотири базові функції управління даними в інформаційних системах: створення, читання, модифікацію та видалення.

DTO — Data Transfer Object — об'єкт передачі даних; шаблон проєктування, що використовується для безпечної та оптимізованої передачі даних між шарами програми або клієнтом і сервером.

GUI — Graphical User Interface — графічний інтерфейс користувача; система засобів для взаємодії користувача з програмою за допомогою візуальних елементів.

JWT — JSON Web Token — відкритий інтернет-стандарт (RFC 7519) для безпечної передачі інформації між сторонами у вигляді зашифрованого JSON-об'єкта; використовується для безстанової автентифікації.

LINQ — Language Integrated Query — технологія від Microsoft, що інтегрує можливості виконання запитів до баз даних безпосередньо у синтаксис мови програмування C#.

ORM — Object-Relational Mapping — об'єктно-реляційне відображення; технологія програмування, яка пов'язує таблиці реляційних баз даних з класами об'єктно-орієнтованих мов.

REST — Representational State Transfer — архітектурний стиль проєктування розподілених застосунків та вебсервісів, що базується на використанні стандартних методів протоколу HTTP.

SPA — Single Page Application — односторінковий застосунок; вебзастосунок, який завантажує єдиний HTML-документ і динамічно оновлює його вміст без повного перезавантаження сторінки під час взаємодії з користувачем.

ВСТУП

У сучасних умовах глобальної цифровізації та стрімкого розвитку інформаційних технологій ефективне управління трудовими ресурсами стає одним із головних критеріїв конкурентоспроможності будь-якого підприємства. Людський ресурс є найціннішим активом компанії, а раціональне планування робочого часу безпосередньо впливає на загальну продуктивність та фінансові показники бізнесу [1, 2].

Актуальність теми.

Традиційні підходи до управління розкладом, такі як використання паперових журналів або неспеціалізованих електронних таблиць, втратили свою ефективність. Вони характеризуються високою часткою ручної праці, що неминуче призводить до помилок в організації робочого процесу, часових накладань робочих змін та значних складнощів у точному підрахунку відпрацьованих годин для нарахування заробітної плати. З іншого боку, впровадження готових глобальних ERP-систем або хмарних SaaS-рішень часто супроводжується невиправдано високими фінансовими витратами, складністю інтеграції та неможливістю адаптації під специфічні внутрішні процеси конкретного підприємства [3, 4].

У зв'язку з цим, розробка власної, оптимізованої корпоративної вебсистеми, яка б забезпечувала єдиний цифровий простір для адміністраторів (менеджерів) та співробітників, є надзвичайно актуальним завданням. Таке рішення дозволить автоматизувати процес планування, ліквідувати вплив людського фактора при складанні графіків та забезпечити абсолютну прозорість комунікації і звітності на підприємстві.

Мета дослідження.

Метою дипломної роботи є проєктування та комплексна програмна реалізація спеціалізованої корпоративної вебсистеми, яка призначена для цифровізації та оптимізації процесів управління трудовими ресурсами підприємства. Головний акцент робиться на створенні єдиного захищеного

інформаційного середовища, що дозволяє повністю автоматизувати планування робочих змін, забезпечити оперативну корпоративну комунікацію та впровадити надійні механізми збору, обробки й аналізу статистики продуктивності персоналу для подальшого прийняття ефективних управлінських рішень.

Методи дослідження.

При виконанні роботи застосовувалися методи системного аналізу, принципи реляційного моделювання баз даних, методи об'єктно-орієнтованого програмування та архітектурні патерни, а також методи автоматизованого модульного тестування.

Практичне значення отриманих результатів.

Практичне значення отриманих результатів полягає у створенні повністю готового до впровадження програмного продукту. Розроблена вебсистема дозволяє підприємству відмовитись від ручних методів планування, забезпечує безпомилковий розподіл робочих змін, спрощує корпоративну комунікацію за допомогою системи повідомлень та значно пришвидшує процес розрахунку заробітної плати завдяки автоматизованій генерації Excel-звітів про відпрацьовані години. Модульна архітектура системи закладає міцний фундамент для її подальшого масштабування та інтеграції в IT-інфраструктуру підприємства.

Апробація результатів.

Результати роботи були представлені на студентській науковій конференції «Сучасні проблеми наукового забезпечення енергетики: безпека, стійкість, IT та екологічний моніторинг», секція №10 «Інформаційні технології та комп'ютерне моделювання» (доповідь «Корпоративна веб-система управління робочими змінами та персоналом», квітень 2026 р.).

1 АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ПОСТАНОВКА ЗАДАЧІ

Ефективне управління трудовими ресурсами є одним із найважливіших факторів успішного функціонування будь-якого сучасного підприємства. В умовах динамічного бізнес-середовища процес планування робочого часу, розподілу змін та ведення обліку відпрацьованих годин вимагає високої точності та оперативності. Традиційні підходи до вирішення цих завдань поступово втрачають свою актуальність, поступаючись місцем спеціалізованим інформаційним системам. Цифровізація цих процесів дозволяє мінімізувати вплив людського фактора, оптимізувати комунікацію між керівництвом та персоналом, а також забезпечити прозорість нарахування заробітної плати на основі реальної статистики продуктивності.

1.1 Огляд існуючих рішень та їх недоліки

На сучасному ринку програмного забезпечення існує значна кількість інструментів для управління персоналом та робочими графіками. Умовно їх можна поділити на три основні категорії: ручні методи обліку (електронні таблиці), глобальні корпоративні ERP-системи та хмарні SaaS-рішення для планування. Детальний аналіз кожної з категорій виявляє низку суттєвих недоліків, які ускладнюють їх використання для конкретних бізнес-потреб.

Використання електронних таблиць (наприклад, Microsoft Excel або Google Sheets) залишається найпоширенішим, але найменш ефективним методом. Головним недоліком такого підходу є відсутність автоматизації: координатору доводиться вручну відслідковувати накладання робочих змін, що неминуче призводить до логічних помилок. Крім того, електронні таблиці не мають вбудованих механізмів сповіщення працівників про зміни в графіку, не

забезпечують належного розмежування прав доступу (працівник може випадково видалити дані) та ускладнюють процес збору статистики за тривалі періоди.

Глобальні корпоративні системи, такі як SAP SuccessFactors або Oracle HCM Cloud, пропонують вичерпний функціонал для управління ресурсами. Однак їх впровадження супроводжується величезними капіталовкладеннями, необхідністю тривалого навчання персоналу та складною інтеграцією з існуючою інфраструктурою. Для підприємств малого та середнього бізнесу використання таких систем є економічно недоцільним, оскільки більшість їхнього функціоналу залишатиметься незатребуваною.

Спеціалізовані хмарні сервіси (SaaS), такі як Sling, Deputy або Homebase, є зручнішими альтернативами. Проте вони мають інший ряд обмежень. По-перше, більшість із них використовують модель оплати за кожного активного користувача (pay-per-user), що призводить до значних щомісячних витрат при масштабуванні штату. По-друге, дані підприємства зберігаються на сторонніх серверах, що може суперечити внутрішнім політикам інформаційної безпеки та конфіденційності компанії. По-третє, такі рішення є універсальними і важко піддаються глибокій кастомізації під специфічні внутрішні процеси конкретної організації.

1.2 Постановка задачі розробки

Враховуючи проаналізовані недоліки існуючих рішень, виникає потреба у створенні власної, незалежної корпоративної вебсистеми для управління розкладом та трудовими ресурсами.

Головною задачею є розробка клієнт-серверного застосунку, який забезпечить єдиний цифровий простір для взаємодії адміністраторів та співробітників підприємства. Система повинна автоматизувати рутинні процеси планування, гарантувати логічну цілісність розкладу (недопущення накладок), забезпечувати надійне зберігання даних та генерувати аналітичну звітність у

зручному форматі. Взаємодія з системою має відбуватися через сучасний, інтуїтивно зрозумілий веб-інтерфейс, адаптований під різні типи пристроїв.

1.2.1 Функціональні вимоги до системи

Для досягнення поставленої мети, система повинна задовольняти чітко визначений перелік функціональних вимог, розділених відповідно до ролей користувачів (рисунок 1.1).

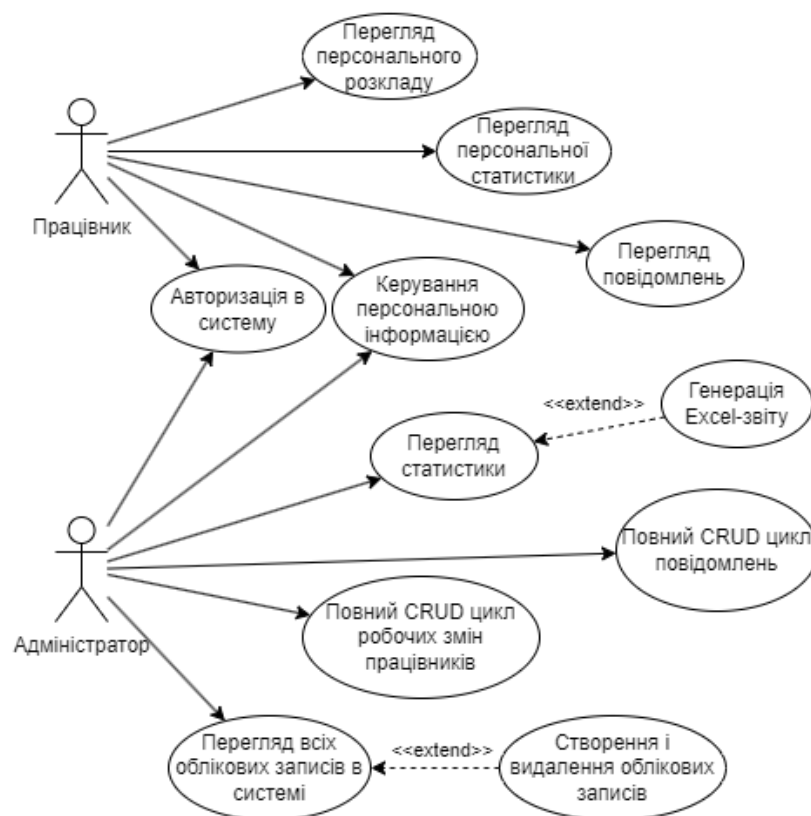


Рисунок 1.1 – Діаграма прецедентів системи управління трудовими ресурсами

Для ролі «Адміністратор» система повинна забезпечувати:

- Можливість реєстрації нових працівників у системі, редагування їх даних та звільнення (видалення профілю).

- Можливість створення, редагування та видалення робочих змін для будь-якого працівника через інтерактивний календар.
- Автоматичне блокування спроб призначення працівнику двох або більше змін, які перетинаються у часі.
- Можливість створення глобальних інформаційних повідомлень для всього колективу, а також їх видалення.
- Перегляд загальної статистики відпрацьованих годин, візуалізація топ-працівників та завантаження деталізованих звітів у форматі електронних таблиць Excel за обраний період.
- Для ролі «Працівник» система повинна забезпечувати:
- Доступ до власного графіка роботи через інтерактивний календар у режимі "тільки для читання".
- Автоматичне відображення кількості відпрацьованих годин за поточний розрахунковий період.
- Перегляд глобальних повідомлень від адміністрації компанії.
- Можливість самостійної зміни пароля доступу до системи.

1.2.2 Нефункціональні вимоги

Окрім безпосереднього виконання бізнес-функцій, система повинна відповідати ряду нефункціональних вимог, що визначають її якість, надійність та зручність експлуатації:

- Доступ до системи повинен надаватися виключно авторизованим користувачам. Паролі повинні зберігатися у базі даних виключно у вигляді незворотних криптографічних хешів. Механізм обміну даними між клієнтом та сервером повинен бути захищеним від міжсайтового скриптингу (XSS) та підробки запитів.

- Архітектура застосунку повинна мінімізувати кількість перезавантажень сторінки. Обчислення важкої аналітики має відбуватися на рівні бази даних, щоб не перевантажувати клієнтські пристрої.
- Система має бути спроектована таким чином, щоб додавання нових модулів або збільшення кількості працівників у базі даних не вимагало переписування існуючого ядра бізнес-логіки.
- Графічний інтерфейс повинен бути мінімалістичним, не перевантаженим зайвими елементами. Всі дії користувача (успішні чи помилкові) повинні супроводжуватись чіткими візуальними сповіщеннями.

1.2.3 Обґрунтування доцільності розробки

Розробка власного спеціалізованого рішення є економічно та стратегічно обґрунтованою. По-перше, впровадження такої системи ліквідує регулярні витрати на підписку, характерні для SaaS-продуктів. Підприємство стає повноправним власником програмного забезпечення і може розгорнути його на власних локальних або хмарних серверах, забезпечивши повний контроль над комерційною та персональною інформацією.

По-друге, перехід від ручного планування до автоматизованої цифрової системи дозволяє керівникам вивільнити значний обсяг робочого часу, що раніше витрачався на складання графіків та підрахунок годин. Виключення людського фактора при обліку робочого часу гарантує справедливе нарахування заробітної плати, що, у свою чергу, підвищує рівень лояльності працівників до компанії. Повний доступ до вихідного коду створює надійний фундамент для майбутніх інтеграцій, наприклад, із бухгалтерськими системами або системами контролю доступу, що робить дану розробку перспективною інвестицією в розвиток ІТ-інфраструктури підприємства.

2 МЕТОДИ ТА ЗАСОБИ РОЗРОБКИ

Створення сучасної корпоративної вебсистеми для управління трудовими ресурсами вимагає уважного підбору інструментів. Обрані технології повинні забезпечувати високу продуктивність, масштабованість, надійність і безпеку обробки даних.

Для реалізації застосунку обрано архітектурний підхід на основі Single Page Application (SPA). Тут клієнтська частина працює незалежно від серверної, обмінюючись даними через REST API. Такий розподіл обов'язків дозволяє розробляти, тестувати та розгортати обидві частини системи окремо.

2.1 Обґрунтування вибору технологічного стеку

Вибір технологічного стеку є одним із найважливіших етапів життєвого циклу розробки програмного забезпечення. Для того, щоб система відповідала визначеним функціональним та нефункціональним вимогам, було проведено аналіз доступних на ринку рішень.

2.1.1 Вибір платформи серверної частини

Серверна частина відповідає за реалізацію бізнес-логіки, взаємодію з базою даних, обробку запитів від клієнта та забезпечення безпеки. Для реалізації серверної частини було обрано платформу .NET (зокрема фреймворк ASP.NET Core) та об'єктно-орієнтовану мову програмування C# [5, 6, 7].

ASP.NET Core має суттєві переваги над конкурентами в сфері корпоративної розробки, а саме те, що сучасні версії .NET демонструють одні з найкращих показників швидкодії серед вебфреймворків. Крім того, кросплатформність ядра дозволяє розгортати застосунок як на Windows серверах, так і в середовищах Linux

або у Docker контейнерах. Мова C# має строгу типізацію, що дозволяє виявляти значну кількість помилок ще на етапі компіляції. Це критично важливо для корпоративних систем, де помилки в бізнес-логіці (наприклад, при розрахунку робочих годин) можуть призвести до фінансових втрат. Також ASP.NET Core надає потужний контейнер впровадження залежностей (Dependency Injection). Це дозволяє уникнути сильної зв'язності коду, спрощує архітектуру та робить систему легкою для модульного тестування. Фреймворк містить готові вбудовані механізми для реалізації безпеки, зокрема Microsoft.AspNetCore.Identity для управління користувачами та ролями, а також надійні засоби для роботи з JWT (JSON Web Token) [8, 9]. Це дозволило швидко та безпечно реалізувати модулі авторизації та розмежування прав доступу. ASP.NET Core надає інтуїтивно зрозумілий механізм маршрутизації та контролерів для створення сучасних API. Відповіді на HTTP запити створюються у вигляді JSON-об'єктів, що є стандартом для взаємодії з сучасними веб-клієнтами. Додатковою перевагою є легка інтеграція зі Swagger для автоматичної генерації документації до API.

Як альтернативи розглядалися платформи Express або NestJS (Node.js) та Spring Boot (Java). Хоча Node.js чудово підходить для I/O-інтенсивних задач, строга типізація C# та багата стандартна бібліотека, роблять .NET кращим вибором для складної бізнес-логіки. Java зі Spring Boot має схожі з .NET характеристики, однак екосистема розробки від Microsoft забезпечує вищу швидкість розробки та нижчий поріг входження, що в свою чергу також зменшує час розробки.

2.1.2 Вибір бази даних та ORM

Забезпечення цілісного та безпечного зберігання інформації є критичним аспектом розробки будь-якої корпоративної системи. Оскільки предметна область застосунку оперує чітко структурованими даними (облікові записи працівників, розклад робочих змін, системні повідомлення), між якими існують жорсткі зв'язки,

було прийнято рішення використовувати реляційну систему управління базами даних.

Як основну базу даних було обрано Microsoft SQL Server [10]. Цей вибір зумовлений її високою надійністю, корпоративним рівнем продуктивності, відповідністю вимогам ACID (атомарність, узгодженість, ізолюваність, довговічність) та безшовною інтеграцією з екосистемою .NET. MS SQL Server забезпечує швидке виконання складних запитів, що є критично важливим для модулів генерації звітів та збору статистики.

Для взаємодії програмного коду серверної частини з базою даних застосовано технологію ORM (Object Relational Mapping). Використання ORM дозволяє абстрагуватися від написання низькорівневих SQL-запитів, оперуючи натомість об'єктами та класами безпосередньо в коді мовою C#. Як основний ORM-інструмент обрано Entity Framework Core (EF Core) [11].

Використання Entity Framework Core у цьому проєкті надає архітектурні та практичні переваги, як от інтеграція з LINQ (Language Integrated Query). EF Core дозволяє формувати запити до бази даних за допомогою синтаксису LINQ. Це робить запити строго типізованими, що дозволяє виявляти синтаксичні помилки під час компіляції. У межах проєкту, LINQ активно використовується для фільтрації робочих змін за датами та розрахунку продуктивності працівників безпосередньо на рівні бази даних. Також наша ORM автоматично використовує параметризовані запити для будь-яких операцій читання чи запису, що нівелює ризики атак типу SQL-ін'єкцій.

Альтернативним варіантом ORM був Dapper. Однак, попри те, що ця мікро-ORM може демонструвати дещо вищу швидкість виконання окремих запитів, він вимагає ручного написання SQL-коду, що сповільнює розробку та ускладнює підтримку проєкту. EF Core є оптимальним компромісом між продуктивністю, широким функціоналом та швидкістю розробки.

2.1.3 Вибір технологій клієнтської частини

Для реалізації клієнтської частини було обрано архітектуру Single Page Application (SPA), що забезпечує високу швидкість взаємодії користувача з системою. У такому підході вебсторінка завантажується лише один раз, а подальше оновлення даних відбувається асинхронно у фоновому режимі без повного перезавантаження сторінки.

Основною мовою програмування клієнтського застосунку виступає TypeScript [12, 13]. На відміну від класичного JavaScript, TypeScript додає строгу статичну типізацію. Це дозволяє описувати інтерфейси об'єктів, які повністю відповідають DTO-моделям (Data Transfer Objects) серверної частини. Такий підхід гарантує цілісність даних на всіх етапах їх передачі, дозволяє виявляти помилки доступу до властивостей ще на етапі написання коду, а також забезпечує повну підтримку з боку середовища розробки.

Головним інструментом для побудови інтерфейсу користувача було обрано бібліотеку React [14, 15]. Вибір на користь цієї бібліотеки обґрунтовується її компонентним підходом, що дозволяє розбивати складний інтерфейс на дрібні, ізольовані та перевикористовувані компоненти. Наприклад, модальні вікна створення змін чи генерації звітів реалізовані як окремі незалежні модулі, що спрощує підтримку та тестування коду. Висока продуктивність React забезпечується концепцією Virtual DOM (Virtual Document Object Model). Бібліотека відслідковує зміни у стані застосунку і мінімізує кількість дорогих операцій прямої взаємодії з реальним деревом документа. Це особливо важливо для розроблюваної системи під час рендерингу складних інтерактивних елементів, таких як розклад робочих змін. Крім того, декларативний стиль React значно спрощує прогнозованість поведінки інтерфейсу. Альтернативні рішення, такі як Angular, пропонують більш монолітну та жорстку структуру, тоді як гнучкість React дозволила підібрати інструментарій індивідуально саме під потреби конкретного проєкту.

Візуальне оформлення та адаптивність застосунку покладається на CSS-фреймворк Bootstrap 5 [16]. Він забезпечує готовий набір стандартизованих компонентів (навігаційні панелі, кнопки, форми, модальні вікна), що дозволяє суттєво скоротити час на верстку та зосередитися безпосередньо на бізнес-логіці системи.

Для забезпечення повноцінної роботи застосунку базова екосистема React була доповнена рядом спеціалізованих бібліотек. Навігація між сторінками реалізована за допомогою react-router-dom [17]. Для роботи з формами та валідації введених користувачем даних використовується react-hook-form [18], що мінімізує кількість перемальовувань компонентів. Асинхронні HTTP-запити до серверного REST API виконуються через сучасний клієнт Axios [19], який дозволяє зручно налаштовувати перехоплювачі запитів (interceptors) для глобальної обробки помилок та управління токенами доступу. Візуалізація ключових даних системи (графіки, розклад) реалізована за допомогою спеціалізованих бібліотек Chart.js [20] та FullCalendar [21].

2.2 Проєктування архітектури за принципами Clean Architecture

Основою проєктування серверної частини став архітектурний підхід Clean Architecture (рисунок 2.1) [22]. Головна ідея підходу полягає в розподілі програмного забезпечення на шари, щоб бізнес-логіка застосунку залишалася незалежною від зовнішніх фреймворків, баз даних та інтерфейсів користувача. Фундаментальним правилом такої архітектури є те, що залежності в коді повинні бути спрямовані виключно всередину, у бік доменних сутностей та бізнес-правил.

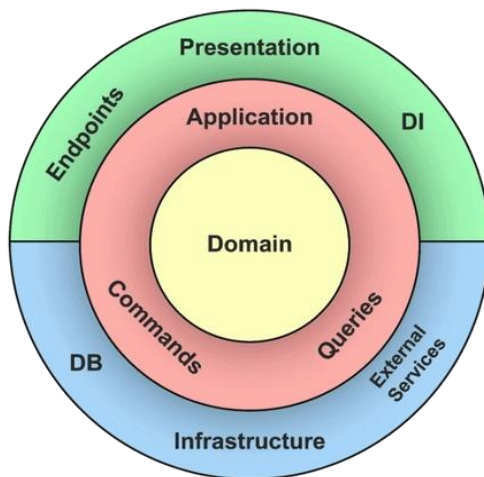


Рисунок 2.1 – Схема чистої архітектури

Для практичної реалізації такого принципу, рішення (Solution) було фізично розділено на три незалежні проєкти: шар ядра (Core), шар інфраструктури (Infrastructure) та шар представлення (WebAPI).

Центральним елементом є шар ядра системи. Цей проєкт не повинен мати жодних зовнішніх залежностей від інших шарів. У ньому зберігаються головні сутності (Entity), об'єкти для передачі даних (DTO), мапери (mappers) та інтерфейси сервісів (ServiceContracts). Згідно з принципами інверсії залежностей (Dependency Inversion), високорівнева бізнес-логіка не має залежати від низькорівневих деталей (наприклад база даних), обидві повинні залежати від абстракції, тому ядро не знає як дані будуть зберігатися чи звідки братися, а лише описує, що система має робити [23].

Шар інфраструктури безпосередньо посилається на шар ядра і містить конкретні реалізації описаних там абстракцій. Саме на цьому рівні знаходиться контекст бази даних, налаштування EF Core та файли міграцій. Крім того, на цьому шарі реалізовані класи сервісів які взаємодіють з базою даних чи сторонніми бібліотеками. Якщо виникне необхідність змінити СУБД на будь-яку іншу, то змінюватиметься лише цей шар, вся бізнес-логіка лишиться.

Головна функція шару представлення — обробка HTTP-запитів від клієнтського застосунку, маршрутизація, перевірка автентифікації користувачів за допомогою JWT та форматування відповідей. Контролери не містять жодної бізнес-логіки, вони взаємодіють із системою виключно через інтерфейси тобто посилаються лише на шар ядра.

Всі шари зв'язуються в одну систему на етапі запуску застосунку за допомогою вбудованого в ASP.NET Core контейнера впровадження залежностей. Завдяки такому рішенню система є дуже гнучкою, легко масштабується і легко покривається модульними тестами адже будь-яку залежність легко підмінити.

2.3 Проєктування бази даних

Для забезпечення надійного зберігання даних було спроектовано реляційну базу даних з дотриманням правил нормалізації (3 нормальна форма), це дозволяє уникнути дублювання даних та аномалій при оновленні. Взаємодія з БД реалізована з допомогою EF Core.

Найзручнішим підходом до створення бази даних є code first підхід (спочатку код), це означає, що ORM сама створює SQL-запити на основі написаного коду і класів сутностей. Також це дозволяє зберігати всі зміни у файлах міграцій які також додаються до Git-репозиторію і це дає можливість дуже легко розгортати проєкт де потрібно, адже, щоб всі зміни застосувались потрібен запуск лише однієї команди.

Головною сутністю системи є користувач, реалізований класом ApplicationUser, він наслідується від базового класу IdentityUser<Guid>, який реалізований в ASP.NET Core Identity. Це дозволяє автоматично генерувати стандартний набір таблиць необхідних для керування користувачами, але не виключає можливості додавання власних таблиць чи полів в них.

Сутність робочої зміни відповідає за облік робочого часу. Ця таблиця містить поля початку і кінця зміни. Зовнішній ключ EmployeeID який посилається на

таблицю користувачів і вказує працівника цієї зміни має зв'язок «один до багатьох». Оскільки цей зв'язок на рівні ORM реалізовано через навігаційну властивість `Employee`, це дає можливість `Entity Framework Core` автоматично завантажувати дані працівника при отриманні інформації про зміну за допомогою вбудованого методу `Include`. І це лише один з прикладів.

Таблиця повідомлень призначена для розсилки загальних інформаційних повідомлень (наприклад, оголошень від адміністрації) для всіх працівників підприємства. Сутність містить текстові поля `Title` (заголовок) та `Message` (вміст повідомлення), а також поле `CreatedAt`, що фіксує точний час створення запису.

Спроектована структура бази даних (рисунок 2.1) повністю задовольняє вимоги до управління трудовими ресурсами підприємства.

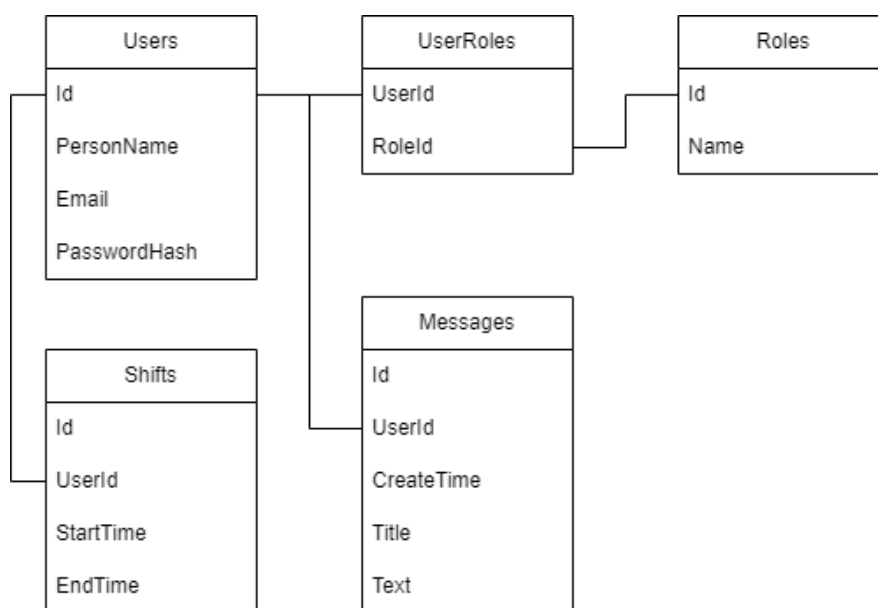


Рисунок 2.1 – Структура бази даних

Завдяки правильно налаштованим зв'язкам на рівні ORM система гарантує коректне каскадне оновлення та видалення залежних записів, надійно запобігаючи появі структурних аномалій у базі.

2.4 Проєктування REST API та механізмів авторизації

Оскільки архітектура застосунку передбачає чітке розділення на клієнтську та серверну частини, єдиним способом їхньої взаємодії є використання API (Application Programming Interface). Для реалізації цієї взаємодії було обрано підхід REST (Representational State Transfer), він базується на використанні стандартних методів HTTP-протоколу для маніпуляції ресурсами системи.

Кожен ресурс має власний контролер, який обробляє відповідні запити. Наприклад, для роботи з робочими змінами використовується маршрут `/api/shifts`, де HTTP-метод GET повертає список змін, POST – створює нову зміну, PUT – оновлює існуючу, а DELETE – видаляє її. Обмін даними між клієнтом та сервером відбувається виключно у форматі JSON (JavaScript Object Notation), що є галузевим стандартом, який легко обробляється як на стороні сервера, так і на стороні клієнта.

З огляду на специфіку корпоративної системи, де обробляються персональні дані співробітників та комерційна інформація підприємства, питання інформаційної безпеки та розмежування доступу є критично важливими. Будь-який неавторизований доступ до API має бути заблокований. Для реалізації цього завдання було впроваджено механізм автентифікації на основі JWT (JSON Web Token).

На відміну від класичних сесій, де сервер змушений зберігати стан кожного користувача в оперативній пам'яті або базі даних, JWT дозволяє серверу залишатися повністю незалежним. Після успішної перевірки логіна та пароля, сервер генерує унікальний зашифрований токен. Цей токен містить так звані Claims – всю потрібну інформацію про користувача. У розробленій системі токен містить унікальний ідентифікатор користувача, його електронну пошту, ім'я та роль.

Згенерований JWT-токен не повинен повертатись у відкритому вигляді в тілі відповіді для збереження в `localStorage`, це робить його вразливим до XSS-атак (Cross-Site Scripting). Натомість сервер відправляє токен клієнту у вигляді HTTP-заголовка, створюючи захищений файл cookie (через параметри `HttpOnly`). Такий підхід робить токен недоступним для читання з клієнтського JavaScript,

автоматично додаючи його до кожного наступного запиту до API через клієнт Axios.

Додатково, на стороні клієнтського застосунку розроблено спеціальні компоненти-маршрутизатори `RequireAuth` та `RequireAdmin`, які захищають візуальні сторінки від несанкціонованого доступу та перенаправляють користувача на сторінку входу у разі відсутності прав чи закінчення терміну дії токена.

На рівні із захистом маршрутів та автентифікацією через токени, не менш важливим аспектом є криптографічний захист облікових даних на рівні бази даних та налаштування безпеки мережевої взаємодії. Зберігання паролів користувачів у базі даних у відкритому текстовому вигляді є грубим порушенням стандартів інформаційної безпеки, оскільки у разі компрометації бази даних (витоку) злоумисники отримають миттєвий доступ до всіх облікових записів.

Для вирішення цієї проблеми у розробленій системі застосовано вбудований інструментарій безпеки фреймворку ASP.NET Core Identity. Програмна реалізація гарантує, що оригінальний пароль користувача ніколи не потрапляє до бази даних. Замість цього використовується одностороннє криптографічне хешування. Механізм хешування перетворює пароль довільної довжини на унікальний рядок фіксованої довжини (хеш-суму). Оскільки ця математична функція є незворотною, отримати оригінальний пароль із хешу неможливо.

Сучасні кібератаки часто використовують метод перебору за словником або заздалегідь згенеровані таблиці для злому хешів. Щоб нівелювати цю загрозу, система Identity автоматично застосовує технологію «додавання солі» (Salting). При реєстрації нового працівника, система генерує криптографічно стійкий випадковий набір даних, який додається до пароля перед його хешуванням. Ця «сіль» зберігається у базі даних разом із хешем. Таким чином, навіть якщо два користувачі створять абсолютно однакові паролі (наприклад, `Password123!`), їхні хеш-суми у базі даних будуть кардинально відрізнятися, що робить використання таких таблиць марним. Алгоритмом хешування за замовчуванням в актуальних версіях .NET є PBKDF2 (Password-Based Key Derivation Function 2) з

використанням HMAC-SHA256, який виконує десятки тисяч ітерацій хешування, щоб штучно сповільнити процес підбору пароля методом грубої сили (Brute-force).

Окрім захисту даних під час зберігання, система потребує захисту даних під час їх передачі та налаштування політик мережевої взаємодії. Оскільки клієнтський застосунок та серверна частина фізично розгорнуті на різних доменах або портах (наприклад, localhost:5173 для фронтенду та localhost:5267 для бекенду), браузері автоматично блокують будь-які HTTP-запити між ними. Це стандартний механізм безпеки браузерів, відомий як «Політика єдиного походження», який запобігає виконанню шкідливих скриптів зі сторонніх сайтів.

Для легітимної взаємодії між клієнтом та сервером у файлі конфігурації Program.cs було спроектовано та налаштовано політику CORS (Cross-Origin Resource Sharing) [24]. Налаштована політика явно вказує серверу приймати запити виключно від домену клієнтського застосунку (WithOrigins).

Також заслуговує уваги налаштування передачі облікових даних у конфігурації CORS (параметр AllowCredentials). Як зазначалося раніше, JWT-токен передається клієнту у вигляді захищеного файлу cookie. Однак, за стандартами безпеки, браузері блокують відправку файлів cookie у крос-доменних запитах. Дозвіл AllowCredentials на стороні сервера у комбінації з налаштуванням withCredentials: true у клієнті Axios утворюють захищений тунель, який дозволяє системі безпечно обмінюватися JWT-токеном при кожному запиті, не порушуючи при цьому загальних політик SOP.

Спроектований комплекс заходів забезпечує високий рівень захищеності корпоративної інформації та відповідає сучасним стандартам веброзробки.

3 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Практична реалізація спроектованої системи вимагала чіткої організації робочого простору та дотримання обраних архітектурних патернів безпосередньо у вихідному коді. Процес розробки був розділений на два паралельні напрямки: написання серверного програмного інтерфейсу (REST API) за допомогою інструментарію .NET SDK та створення інтерактивного клієнтського вебзастосунку на базі екосистеми Node.js та збирача Vite [25]. Для забезпечення високої якості коду, його читабельності та легкості подальшого супроводу, особлива увага приділялася фізичному розподілу файлів та модулів по відповідних директоріях.

3.1 Структура проєкту та розподіл на шари

Серверна частина системи організована у вигляді єдиного рішення WorkviaSolution, яке розділене на чотири окремі проєкти. Такий поділ на рівні файлової системи C# гарантує неможливість порушення архітектурних меж, оскільки залежності між проєктами налаштовуються строго однонаправлено через файли конфігурації .csproj.

Workvia.Core – це проєкт типу Class Library, що виступає ядром системи. Він не має залежностей від інших проєктів рішення. Внутрішня структура директорій цього проєкту структурована таким чином (рисунок 3.1):

- Entities – містить класи доменних сутностей (ApplicationUser, Shift, Notification і тд.), які безпосередньо відображають таблиці бази даних.
- DTO (Data Transfer Objects) – об'єкти передачі даних. Тут описані класи (наприклад, ShiftRequestDTO, LoginDTO), які використовуються для прийому даних від клієнта та відправки відповідей. Використання DTO дозволяє приховати внутрішню структуру бази даних та захистити систему від атак типу Over-posting.

- ServiceContracts – директорія з інтерфейсами (наприклад, IShiftService, IStatisticsService), які описують контракти для виконання бізнес-логіки.
- Mappers – містить статичні класи-розширення (ShiftMappingExtensions) для перетворення об'єктів Entity у DTO і навпаки.
- Services – хоча в своїй суті Clean Architecture виключає створення сервісів напряду в шарі ядра, специфічні сервіси все там існують. Вони взаємодіють з даними не напряду через контекст бази даних, а через абстракції UserManager, надані фреймворком. Натомість сервіси, які безпосередньо формують SQL-запити через ApplicationDbContext, або працюють зі специфічними файловими форматами через сторонні бібліотеки, винесені у окремий шар.

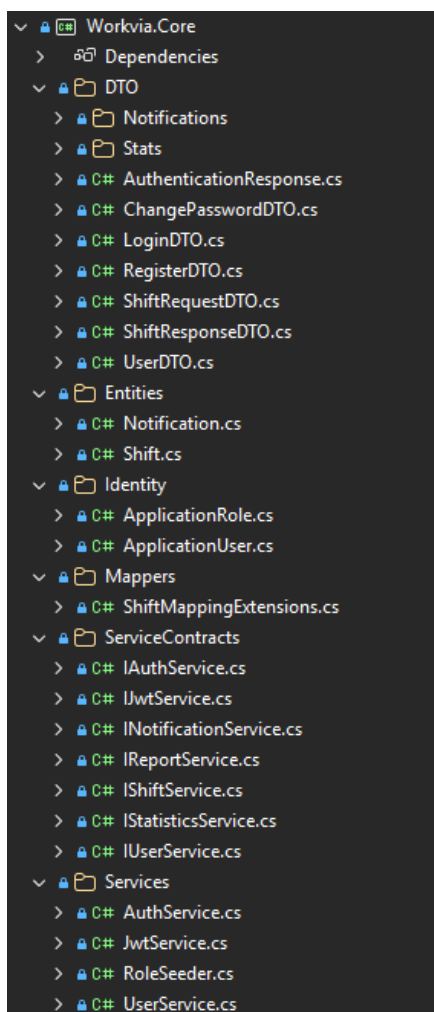


Рисунок 3.1 – Структура проєкту Workvia.Core

Проект інфраструктури, `Workvia.Infrastructure` (рисунок 3.2), відповідає за доступ до даних та зовнішні інтеграції. Він посилається виключно на `Workvia.Core`.

- `DatabaseContext` – містить клас `ApplicationDbContext`, який налаштовує зв'язок сутностей з Entity Framework Core.
- `Migrations` – автоматично згенеровані файли, що описують історію змін схеми бази даних.
- `Services` – містить конкретні реалізації інтерфейсів ядра (наприклад, `ShiftService.cs`, `ReportService.cs`), де прописана логіка взаємодії з базою даних через LINQ-запити.

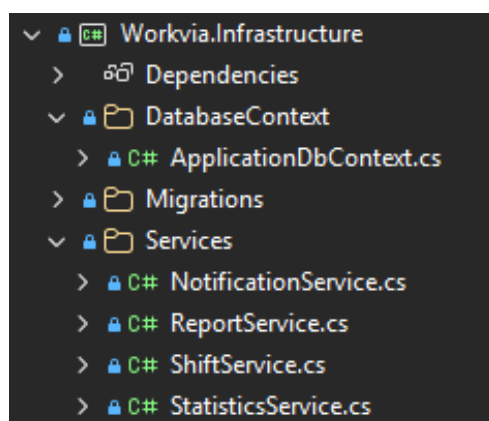


Рисунок 3.2 – Структура проєкту `Workvia.Infrastructure`

Виконуваний ASP.NET Core застосунок, `Workvia.WebAPI`, шар представлення (рисунок 3.3). Він слугує точкою входу в систему. Папка `Controllers` містить класи контролерів (`ShiftsController`, `AccountController` і тд.), які приймають HTTP-запити, викликають відповідні сервіси через їхні інтерфейси та повертають HTTP-відповіді. Файл `Program.cs` відіграє критичну роль конфігуратора системи. Саме тут налаштовується підключення до бази даних, параметри JWT-автентифікації, політики CORS, а також реєструються всі залежності для вбудованого DI-контейнера.

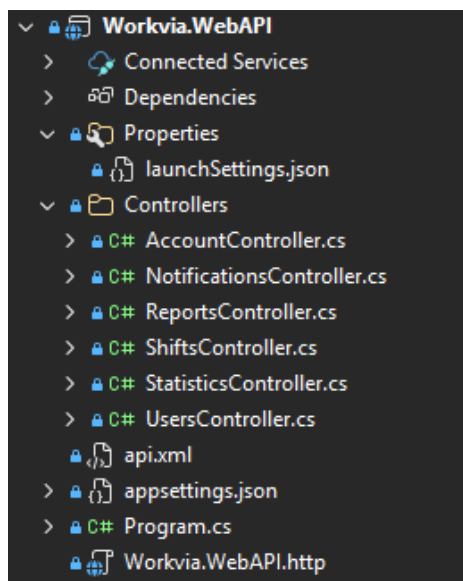


Рисунок 3.3 – Структура проєкту Workvia.WebAPI

Проект Workvia.Tests створений для написання модульних тестів за допомогою фреймворку xUnit, що дозволяє ізольовано перевіряти коректність роботи бізнес-сервісів.

Клієнтська частина застосунку розроблена з використанням парадигми Feature Based Folder Structure (структура папок на основі функціоналу). Замість традиційного групування за типом файлів (всі компоненти разом, всі стилі разом), код згруповано за бізнес-доменами (рисунок 3.4).

Основний код розміщено в директорії src:

- components – містить глобальні UI-компоненти, які перевикористовуються по всій системі незалежно від контексту (Modal, Navbar, DateTimePicker).
- core/guards – компоненти, що відповідають за захист маршрутів на стороні клієнта (RequireAuth.tsx, RequireAdmin.tsx).
- features – ядро клієнтської архітектури. Директорія розділена на логічні модулі: admin, employee, auth та shared. Кожен модуль містить власні сторінки, специфічні компоненти та макети, що робить код високо модульним.

- services – файли (наприклад, shift.service.ts, auth.service.ts), які інкапсулюють логіку формування HTTP-запитів до бекенду. Тут також знаходиться файл api.ts, який містить налаштований екземпляр клієнта Axios.
- types – директорія з описами інтерфейсів TypeScript, які строго відповідають структурі DTO з серверної частини, забезпечуючи типізацію під час компіляції клієнтського застосунку.

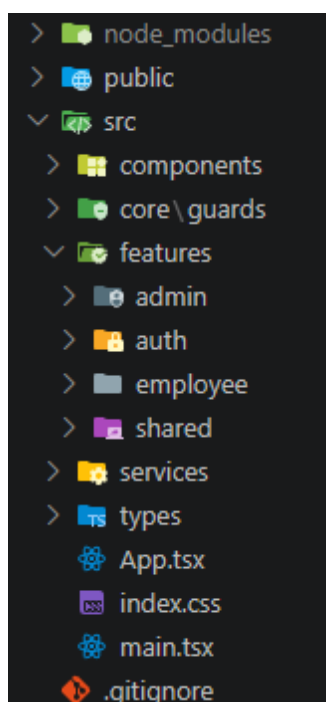


Рисунок 3.4 – Структура клієнтського застосунку

Такий строгий розподіл відповідальності як на серверній, так і на клієнтській сторонах дозволив уникнути сильної зв'язності коду та забезпечив можливість паралельної розробки різних модулів системи.

Для системного розуміння архітектури розробленого програмного продукту та візуалізації взаємозв'язків між описаними вище проєктами, було побудовано UML-діаграму компонентів (рисунок 3.5). Ця діаграма наочно демонструє напрямки потоків управління та фізичні залежності між модулями системи відповідно до парадигми Clean Architecture.

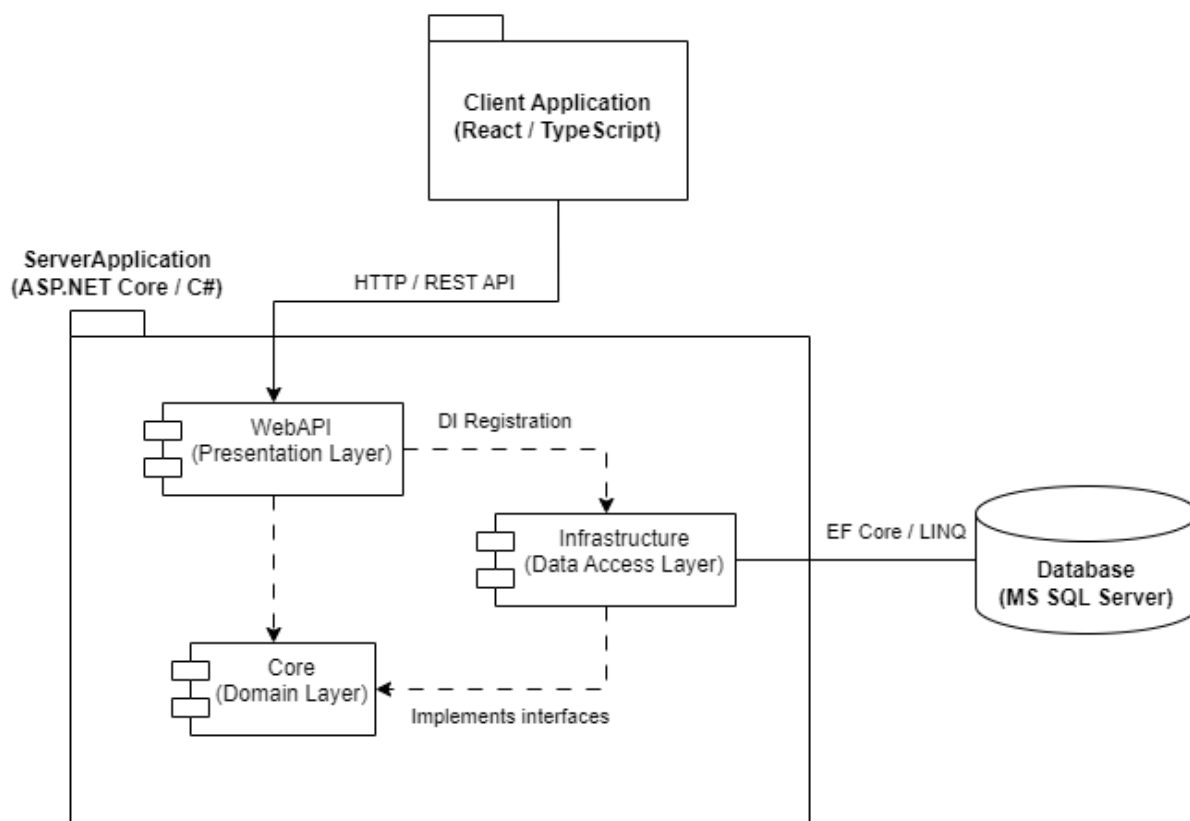


Рисунок 3.5 – UML-діаграма компонентів та взаємодії шарів системи

Як видно з діаграми, центральним та повністю незалежним вузлом системи є компонент Workvia.Core. Згідно з правилом інверсії залежностей (Dependency Inversion), він не має жодних вихідних зв'язків з іншими пакетами, що гарантує ізолюваність бізнес-логіки від зовнішніх технологій.

Компонент Workvia.Infrastructure безпосередньо залежить від Workvia.Core, оскільки його головною задачею є реалізація контрактів (інтерфейсів), описаних у ядрі. З іншого боку, інфраструктурний шар має зовнішню залежність від СУБД MS SQL Server, з якою він спілкується за допомогою провайдера Entity Framework Core.

Компонент Workvia.WebAPI виступає фасадом серверної частини. Він має залежності як від Workvia.Core (для доступу до DTO та інтерфейсів при обробці запитів), так і від Workvia.Infrastructure. Залежність від інфраструктури необхідна

виключно на етапі запуску застосунку для реєстрації конкретних реалізацій сервісів у контейнері впровадження залежностей (DI Container).

Взаємодія між клієнтським компонентом Client Application (React) та серверною частиною є слабозв'язаною. Клієнт не має прямого доступу до коду чи бази даних сервера, комунікація відбувається виключно через мережевий протокол HTTP за допомогою викликів кінцевих точок REST API. Такий компонентний розподіл забезпечує високу модульність: будь-який компонент (наприклад, клієнтський застосунок або СУБД) може бути замінений й або оновлений без необхідності переписування ядра системи.

3.2 Реалізація серверної логіки

Життєвий цикл будь-якої операції на стороні сервера ініціюється отриманням HTTP-запиту від клієнтського застосунку. Процес обробки побудований у вигляді конвеєра (pipeline) проміжного програмного забезпечення (middleware), налаштованого у файлі Program.cs (рисунок 3.6). Перед тим як запит досягне бізнес-логіки, він проходить через механізми маршрутизації (Routing), політики єдиного походження (CORS) та перевірку автентифікації. Якщо маршрут вимагає наявності прав доступу (наприклад, маршрути, позначені атрибутом [Authorize]), система автоматично вилучає JWT-токен із файлів cookie запиту, розшифровує його та перевіряє валідність підпису і термін дії.

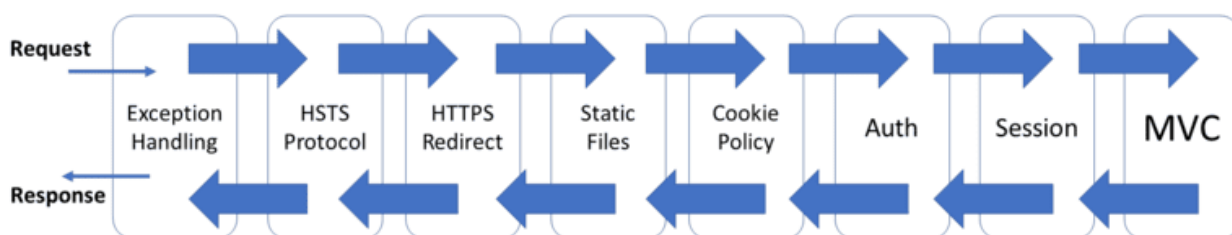


Рисунок 3.6 – Конвеєр обробки запитів до сервера

Після успішної авторизації запит передається до відповідного контролера. Відповідно до концепції «тонких контролерів», класи в теці `Controllers` позбавлені складної логіки обчислень. Їхньою головною задачею є прийом вхідних параметрів, ініціалізація виклику відповідного сервісу та форматування HTTP-відповіді з правильним статус-кодом (наприклад, 200 OK, 201 Created, 400 BadRequest або 404 NotFound). Усі зовнішні залежності передаються в контролери через конструктор за допомогою механізму впровадження залежностей (Dependency Injection), який реєструє сервіси з життєвим циклом `Scoped` (один екземпляр сервісу на один HTTP-запит).

Важливим етапом обробки вхідних даних є їхня десеріалізація та валідація. Для обміну даними між клієнтом та сервером розроблено набір об'єктів передачі даних (DTO), таких як `ShiftRequestDTO` або `LoginDTO`. Використання DTO відіграє ключову роль у забезпеченні безпеки: клієнт не має прямого доступу до сутностей бази даних, що унеможлиблює модифікацію прихованих полів. Валідація даних виконується автоматично завдяки атрибутам анотацій даних, таких як `[Required]` чи `[EmailAddress]`. Крім того, для складніших перевірок реалізовано інтерфейс `IValidatableObject`. Наприклад, у класі `ShiftRequestDTO` реалізовано перевірку, яка гарантує, що час закінчення зміни (`EndTime`) завжди є більшим за час її початку (`StartTime`). Якщо дані не проходять валідацію, контролер миттєво повертає помилку, не завантажуючи рівень бази даних.

Основна бізнес-логіка виконується на рівні сервісів інфраструктурного шару. Коли контролер викликає метод сервісу, наприклад `CreateShiftAsync` у класі `ShiftService`, сервіс спочатку виконує специфічні перевірки бізнес-правил. Зокрема, викликається метод `IsOverlapAsync`, який робить запит до бази даних за допомогою `Entity Framework Core` для перевірки того, чи не перетинається нова робоча зміна з уже існуючими змінами для конкретного працівника.

Якщо всі перевірки пройдені успішно, вхідний DTO об'єкт необхідно трансформувати у доменну сутність для збереження у базі даних. Для цього у ядрі системи реалізовано статичний клас `ShiftMappingExtensions`, який містить методи-розширення для мапінгу об'єктів. Метод `ToEntity` створює екземпляр класу `Shift`,

який потім додається до контексту бази даних `ApplicationDbContext` і зберігається методом `SaveChangesAsync()`.

З метою забезпечення високої пропускнуєї здатності сервера, абсолютно всі операції введення-виведення (I/O), такі як звернення до бази даних або генерація звітів, реалізовані з використанням асинхронного програмування на базі ключових слів `async/await` та типів `Task`. Це дозволяє звільняти потоки вебсервера під час очікування відповіді від СУБД, що значно підвищує масштабованість системи під навантаженням.

Після успішного виконання операції в базі даних, створена або оновлена сутність знову мапиться у вихідний формат (наприклад, за допомогою методу `ToResponseDTO`) і повертається контролеру, який серіалізує її у формат JSON і відправляє клієнту.

3.3 Реалізація клієнтського застосунку

Архітектура клієнтського застосунку базується на використанні функціональних компонентів React та системи хуків, що забезпечує миттєве оновлення інтерфейсу користувача у відповідь на зміну даних.

Організація переходів між сторінками без перезавантаження веббраузера реалізована за допомогою бібліотеки `react-router-dom`. Головний файл маршрутизації (`App.tsx`) описує ієрархічне дерево доступних шляхів (routes). Маршрути логічно поділені на публічні (наприклад, сторінка входу `/login`) та захищені. Захист маршрутів реалізується за допомогою спеціальних компонентів-обгорт `RequireAuth` та `RequireAdmin`. При спробі переходу на будь-яку захищену сторінку, ці компоненти звертаються до `AuthService` для перевірки наявності та валідності токена у локальному сховищі. Якщо користувач не авторизований, компонент блокує рендеринг цільової сторінки та примусово перенаправляє його на сторінку логіну. Аналогічно, `RequireAdmin` перевіряє наявність ролі «Admin» для доступу до панелі керування користувачами та розкладом.

Для побудови загального каркаса сторінок застосовується концепція макетів (Layouts). Компоненти `AdminLayout` та `EmployeeLayout` містять статичні елементи навігації (бічну панель), тоді як динамічний контент конкретної сторінки рендериться у спеціальному компоненті `<Outlet />`. Це дозволяє уникнути дублювання коду навігаційного меню в кожному окремому файлі сторінки.

Життєвий цикл отримання та відображення даних на сторінках стандартизовано. При використанні компонента (наприклад, сторінки `Users.tsx`), за допомогою хука `useEffect` ініціюється асинхронний запит до серверного API для отримання масиву об'єктів. До моменту отримання відповіді від сервера, локальний стан компонента, керований хуком `useState`, містить прапорець завантаження (`isLoading = true`), що дозволяє відобразити користувачеві індикатор очікування. Після успішного завантаження даних, стан оновлюється отриманим масивом, що викликає автоматичне перемальовування компонента та відображення таблиці або графіка.

Окремої уваги заслуговує підсистема взаємодії з REST API, яка інкапсульована у файлі `api.ts`. Замість використання стандартного `Fetch API`, застосовано бібліотеку `Axios`, що дозволило створити глобальний конфігурований клієнт. Ключовим налаштуванням екземпляра `Axios` є параметр `withCredentials: true`, який автоматично прикріплює до кожного вихідного запиту захищені HTTP-only файли `cookie` (що містять JWT-токен), забезпечуючи безперебійну авторизацію.

Для централізованої обробки мережових та серверних помилок реалізовано механізм перехоплювачів відповідей (`Response Interceptors`). Завдяки цьому архітектурному рішенню, компонентам візуального інтерфейсу не потрібно містити блоки `try-catch` для аналізу кодів помилок кожного окремого HTTP-запиту. Перехоплювач глобально аналізує статус-код відповіді від сервера:

- У разі отримання статусу 400 (`Bad Request`), перехоплювач парсить тіло відповіді, витягує масив помилок валідації (наприклад, повідомлення про те, що електронна пошта вже існує) та виводить першу з них користувачеві за допомогою спливаючого повідомлення (`toast-сповіщення` з бібліотеки `react-toastify`).

- Статус 401 (Unauthorized) генерує повідомлення про необхідність повторної авторизації.
- Статус 403 (Forbidden) інформує про відсутність прав доступу до ресурсу.
- Статус 500 (Internal Server Error) перехоплюється як критичний збій сервера з відповідним візуальним сповіщенням.

Такий підхід гарантує єдиний стандарт інформування користувача про помилки по всьому застосунку, значно розвантажує компоненти від зайвої логіки та робить код легко підтримуваним.

3.4 Реалізація ключових модулів

Розробка програмної системи велася за модульним принципом, де кожен складний бізнес-процес ізольовано у відповідний логічний блок. Взаємодія між клієнтським застосунком та серверною частиною під час виконання конкретних завдань вимагає синхронізованої роботи UI-компонентів, клієнтських сервісів та бекенд-контролерів. Нижче детально розглянуто практичну реалізацію найбільш критичних модулів розробленої системи.

3.4.1 Модуль авторизації та управління доступом

Процес ідентифікації та автентифікації користувача починається з компонента Login.tsx. Для побудови форми входу та управління її станом використано бібліотеку react-hook-form. Її головна перевага полягає у використанні uncontrolled компонентів, що дозволяє проводити строгу клієнтську валідацію введених даних (перевірка формату електронної пошти за допомогою регулярних виразів, перевірка на порожні поля) без виклику ресурсомісткого перемальовування всього компонента при кожному натисканні клавіші.

Після генерації токена, контролер формує відповідь об'єктом `AuthResponse`. Як зазначалося під час проєктування, сам секретний токен пакується у захищений заголовок `Set-Cookie`. Однак клієнтському застосунку для правильного відображення інтерфейсу (наприклад, імені користувача у навігаційній панелі або доступу до адмін-меню) потрібні метадані. Тому сервер додатково повертає JSON-об'єкт, що містить ім'я, роль та час закінчення дії сесії.

Цей несекретний об'єкт метаданих зберігається у локальному сховищі браузера (`localStorage` під ключем `authData`). Клієнтський `AuthService` використовує ці дані для синхронної перевірки прав: метод `isLoggedIn()` перевіряє, чи не минув час `expiration`, а метод `isAdmin()` читає поле `role`.

Весь процес автентифікації користувача зображено на рисунку 3.8.

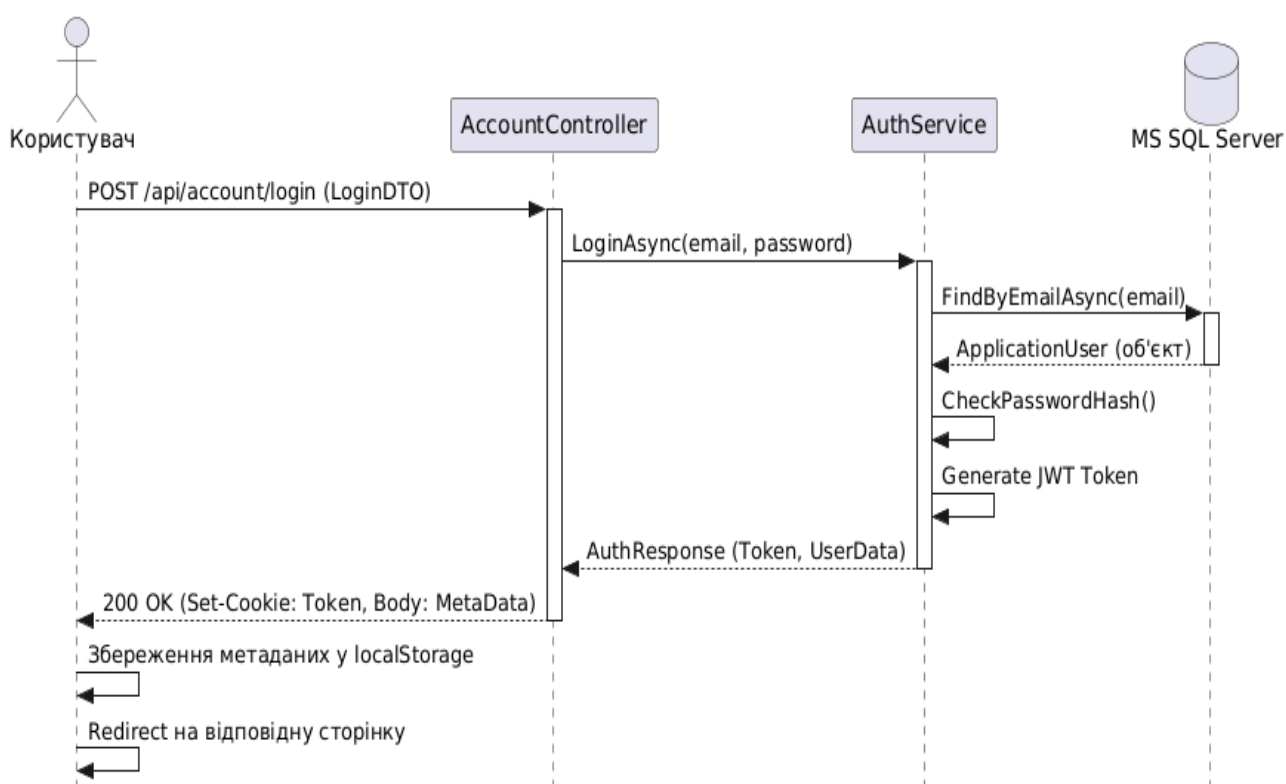


Рисунок 3.8 – Діаграма послідовності процесу автентифікації користувача

На базі цих методів реалізовано умовний рендеринг (`Conditional Rendering`) у користувацькому інтерфейсі. Наприклад, у компоненті `App.tsx` після успішного

входу відбувається перевірка: якщо `AuthService.isAdmin()` повертає `true`, користувача автоматично перенаправляє на маршрут `/admin` (з доступом до керування розкладом та генерації звітів), в іншому випадку на маршрут `/employee` (де доступний лише персональний графік). Аналогічним чином працює компонент `Navbar.tsx`, який динамічно відображає ім'я поточного користувача та кнопку «Logout», якщо користувач авторизований у системі.

3.4.2 Інтерактивний календар та планування змін

Модуль робочих змін є основним інструментом для розподілу навантаження та візуального контролю графіків співробітників. Візуалізація розкладу на клієнтській стороні реалізована за допомогою бібліотеки `FullCalendar`. Для забезпечення максимальної інформативності, компонент `AdminSchedule` налаштовано на роботу з кількома режимами перегляду (місяць, тиждень, день) завдяки інтеграції додаткових плагінів (рисунок 3.9).



Рисунок 3.9 – Вигляд заповненого календаря в режимі перегляду за тиждень

Отриманий від серверного API масив об'єктів типу Shift динамічно трансформується у формат подій (events), який вимагає бібліотека календаря. Для покращення користувацького досвіду та швидкого візуального розпізнавання змін різних працівників, у коді реалізовано допоміжну функцію `stringToColor`. Цей алгоритм приймає ім'я співробітника як рядок, обчислює його хеш-суму за допомогою побітових операцій та конвертує результат у шістнадцятковий (HEX) код кольору. Таким чином, кожен працівник автоматично отримує унікальний ідентифікаційний колір фону на графіку, який залишається незмінним при кожному рендерингу.

Управління даними (створення, оновлення, видалення) реалізовано через спливаючі модальні вікна `ShiftCreateModal` та `ShiftUpdateModal`. Введення часових проміжків відбувається за допомогою кастомного компонента `DateTimePicker`, який виступає обгорткою над бібліотекою `react-datepicker`.

Управління даними (створення, оновлення, видалення) реалізовано через спливаючі модальні вікна `ShiftCreateModal` та `ShiftUpdateModal`. Введення часових проміжків відбувається за допомогою кастомного компонента `DateTimePicker`, який виступає обгорткою над бібліотекою `react-datepicker`. Первинна перевірка коректності даних відбувається ще до відправки HTTP-запиту: логіка компонента перевіряє, щоб час завершення зміни обов'язково був більшим за час її початку.

Діаграма діяльності процесу призначення робочої зміни зображено на рисунку 3.10.

На стороні сервера (у `Workvia.WebAPI`) впроваджено багатоступеневий захист цілісності розкладу. Перший рівень захисту працює на етапі прив'язки моделі (Model Binding). Клас `ShiftRequestDTO` реалізує інтерфейс `IValidatableObject`, де програмно описано правило: якщо $EndTime \leq StartTime$, генерується помилка валідації (`ValidationResult`), і контролер автоматично повертає клієнту статус 400 Bad Request. Це гарантує, що жодні хибні хронологічні дані не потраплять до сервісного шару.

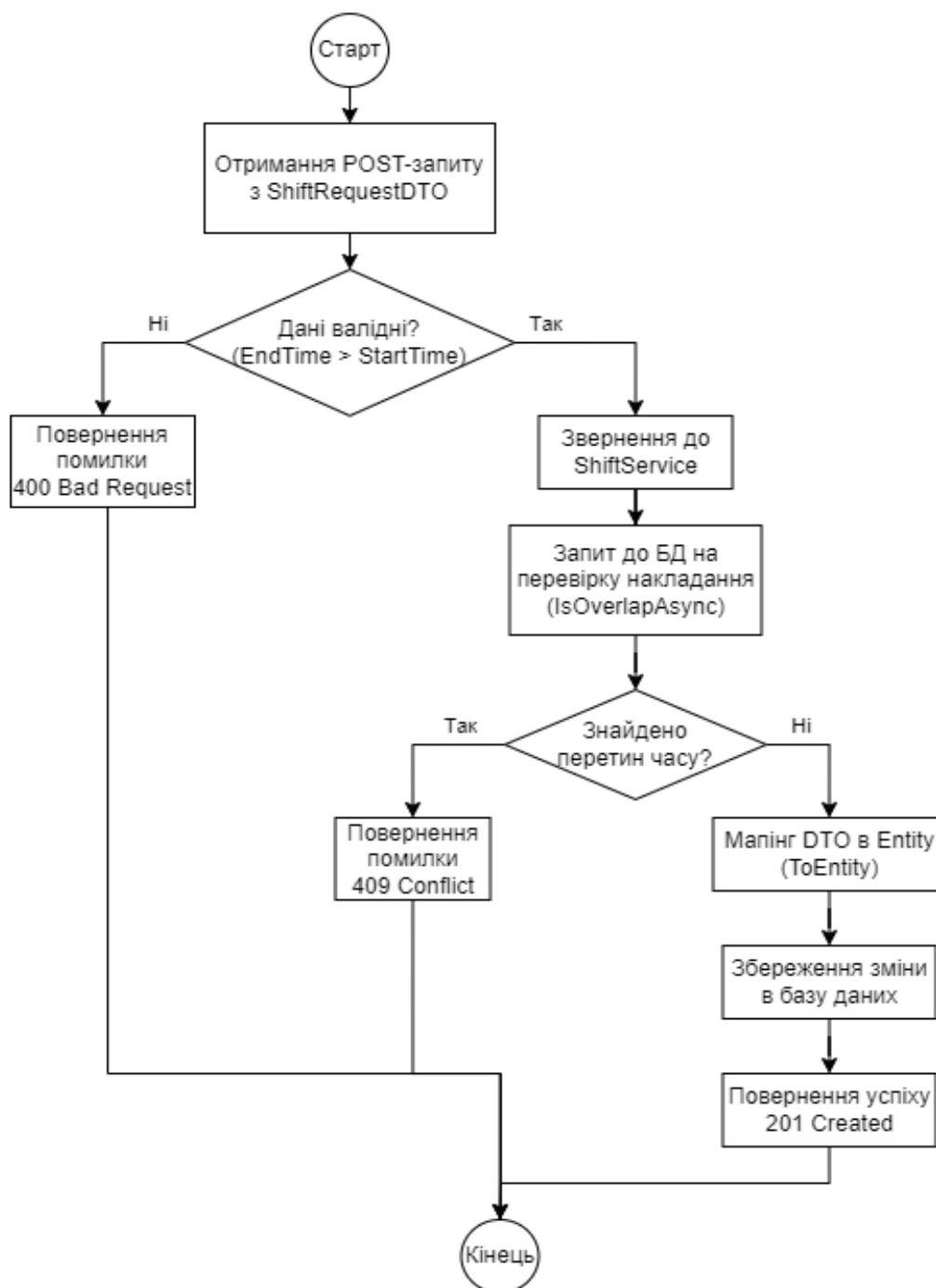


Рисунок 3.10 – Блок-схема діяльності процесу призначення робочої зміни

Головна бізнес-логіка модуля зосереджена у класі ShiftService. Найбільш критичним завданням цього сервісу є запобігання ситуаціям, коли одному працівнику намагаються призначити дві різні робочі зміни, які перетинаються в часі. Для цього реалізовано приватний метод IsOverlapAsync. Метод виконує LINQ-запит до бази даних через Entity Framework Core, шукаючи записи для конкретного

EmployeeID, які задовольняють умову перетину відрізків: час початку існуючої зміни менший за час кінця нової, а час кінця існуючої зміни більший за час початку нової ($s.StartTime < end \ \&\& \ s.EndTime > start$). Під час операції оновлення зміни (PUT-запит), цей запит також ігнорує поточний ShiftID, щоб зміна не конфліктувала сама з собою.

Якщо метод виявляє перетин, операція запису переривається, і сервіс повертає кортеж з повідомленням про помилку «Shift overlaps with an existing shift». Контролер трансліює цю помилку у статус 409 Conflict, а глобальний перехоплювач Axios на клієнті виводить її адміністратору у вигляді toast-сповіщення. Такий надійний архітектурний ланцюжок повністю захищає базу даних від логічних аномалій при плануванні ресурсів.

3.4.3 Модуль статистики та генерації звітів

Модуль збору статистики та формування звітності є ключовим інструментом для аналізу ефективності використання трудових ресурсів підприємства. Реалізація цього модуля вимагала об'ємних обчислень, які для забезпечення оптимальної швидкодії були перенесені з оперативної пам'яті сервера безпосередньо на рівень бази даних.

Бізнес-логіка розрахунку показників інкапсульована у класі StatisticsService. Замість того, щоб завантажувати всі записи робочих змін у пам'ять і обробляти їх засобами мови C#, обчислення виконуються на стороні СКБД Microsoft SQL Server за допомогою LINQ-запитів. Наприклад, для підрахунку загальної кількості відпрацьованих годин або визначення топ-5 найпродуктивніших працівників застосовується вбудована функція EF.Functions.DateDiffMinute. Вона дозволяє Entity Framework Core трансліювати код C# у нативний SQL-запит із використанням функції обчислення різниці в хвилинах між часом початку та завершення зміни. Далі застосовуються методи групування (GroupBy), сумування (SumAsync) та сортування за спаданням (OrderByDescending). Такий підхід суттєво економить

ресурси сервера та зменшує обсяг даних, що передаються мережею між базою даних та API.

Для візуального представлення аналітичних даних на клієнтській стороні інтегровано бібліотеку Chart.js (рисунок 3.11). Отримані від сервера JSON-масиви з показниками продуктивності динамічно конвертуються у конфігураційні об'єкти для графіків. Статистика топ-5 працівників рендериться у вигляді стовпчастої діаграми, де віссю X виступають імена співробітників, а віссю Y — відпрацьовані години. Розподіл кількості робочих змін по днях тижня реалізовано за допомогою кільцевої діаграми. Використання хуків `useEffect` та `useState` забезпечує паралельне завантаження цих незалежних наборів даних, що пришвидшує загальне відображення сторінки панелі керування.

Особливої уваги заслуговує алгоритм експорту даних у формат електронних таблиць Excel. Оскільки генерація складних файлів на стороні клієнта за допомогою JavaScript може призвести до зависання браузера при великих обсягах даних, цей процес повністю делеговано серверу [26]. У класі `ReportService` застосовується бібліотека `ClosedXML`, яка дозволяє працювати з форматом `OpenXML` без необхідності встановлення пакету Microsoft Office на сервері.

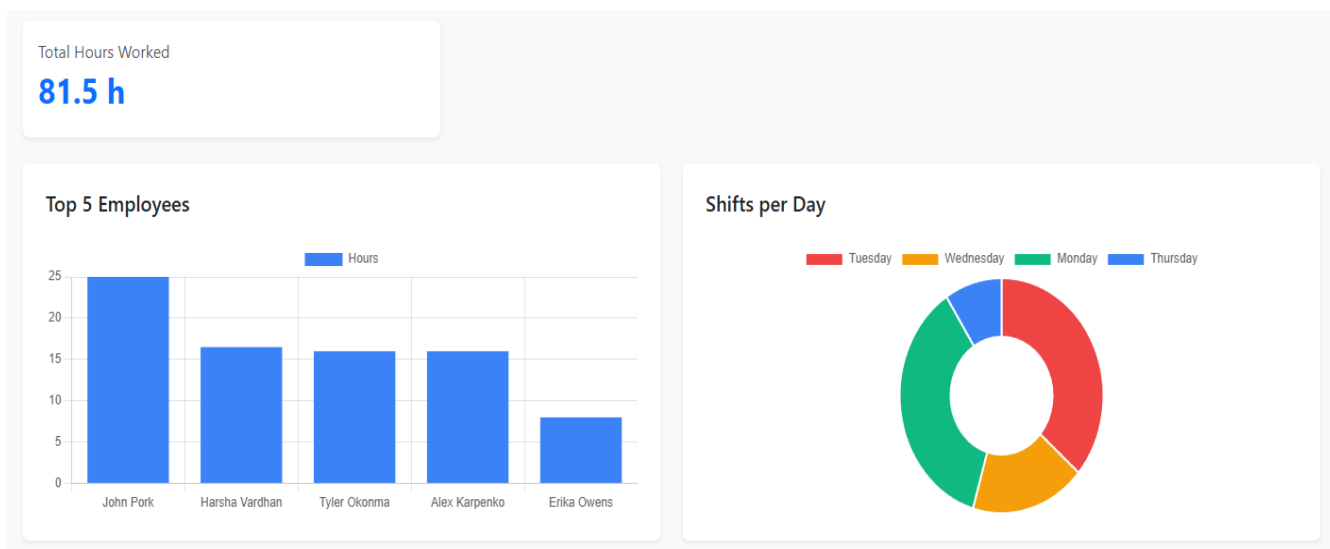


Рисунок 3.11 – Вигляд сторінки статистики

Алгоритм генерації звіту створює віртуальну книгу, що складається з двох аркушів. Перший аркуш «All shifts» (рисунок 3.12) містить детальну хронологію всіх відпрацьованих змін за обраний період, а другий «Employee summary», згруповані дані з підрахунком загальних годин для кожного працівника. Бібліотека також виконує базове стилізування таблиці, зокрема виділення заголовків напівжирним шрифтом та кольором фону. Сформований документ повертається контролеру як масив байтів.

	A	B	C	D	E
1	Employee name	Email	Shift start	Shift end	Hours worked
2	Tyler Okonma	tyler@example.com	20.04.2026 00:00	20.04.2026 08:00	8
3	John Pork	john@example.com	20.04.2026 08:00	20.04.2026 16:00	8
4	Harsha Vardhan	harsha@example.com	20.04.2026 13:00	20.04.2026 21:30	8,5
5	Alex Karpenko	alex@example.com	20.04.2026 16:00	21.04.2026 00:00	8
6	Tyler Okonma	tyler@example.com	21.04.2026 00:00	21.04.2026 08:00	8
7	John Pork	john@example.com	21.04.2026 08:00	21.04.2026 16:00	8
8	Harsha Vardhan	harsha@example.com	21.04.2026 12:00	21.04.2026 20:00	8
9	Alex Karpenko	alex@example.com	21.04.2026 16:00	22.04.2026 00:00	8
10	Erika Owens	erika@example.com	22.04.2026 00:00	22.04.2026 08:00	8
11	John Pork	john@example.com	22.04.2026 08:00	22.04.2026 16:00	8

Рисунок 3.12 – Вигляд «All shifts» згенерованого звіту

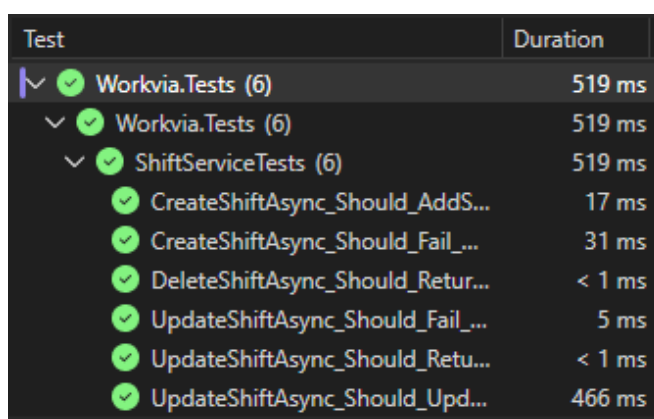
На стороні клієнтського застосунку обробка завантаження такого файлу реалізована за допомогою специфічного підходу. Сервіс ReportService.ts виконує GET-запит до API, вказуючи параметр «responseType: 'blob'». Це сигналізує бібліотеці Axios, що відповідь сервера є не текстовим JSON-об'єктом, а потоком бінарних даних. Отриманий blob (Binary large object) перетворюється на тимчасову локальну URL-адресу за допомогою методу window.URL.createObjectURL(). Після цього в DOM програмно створюється невидимий HTML-елемент <a>, йому призначається згенероване посилання та ім'я файлу, після чого імітується подія кліку (link.click()). Цей скрипт змушує браузер розпочати стандартний процес завантаження файлу на пристрій користувача. Після успішного завантаження тимчасове посилання та створений елемент знищуються для звільнення оперативної пам'яті браузера.

4 ТЕСТУВАННЯ ТА ВЗАЄМОДІЯ З СИСТЕМОЮ

Невіддільним етапом життєвого циклу розробки програмного забезпечення є верифікація реалізованого функціоналу та оцінка зручності взаємодії кінцевого користувача з готовим продуктом. Якість розробленої корпоративної системи забезпечується комбінацією автоматизованого тестування критичних вузлів бізнес-логіки та продуманого, інтуїтивно зрозумілого графічного інтерфейсу, що мінімізує поріг входження для нових працівників.

4.1 Модульне тестування за допомогою xUnit

Для гарантування надійності серверної частини та уникнення помилок у проєкті реалізовано модульне тестування (рисунок 4.1). Як основний інструмент тестування обрано фреймворк xUnit, який є сучасним стандартом для екосистеми .NET.



Test	Duration
✓ Workvia.Tests (6)	519 ms
✓ Workvia.Tests (6)	519 ms
✓ ShiftServiceTests (6)	519 ms
✓ CreateShiftAsync_Should_AddS...	17 ms
✓ CreateShiftAsync_Should_Fail_...	31 ms
✓ DeleteShiftAsync_Should_Retur...	< 1 ms
✓ UpdateShiftAsync_Should_Fail_...	5 ms
✓ UpdateShiftAsync_Should_Retu...	< 1 ms
✓ UpdateShiftAsync_Should_Upd...	466 ms

Рисунок 4.1 – Результати успішного виконання модульних тестів у Test Explorer

Головним викликом при тестуванні сервісів є необхідність ізоляції бізнес-логіки від реальної бази даних MS SQL Server. Виконання тестів на реальній базі даних є повільним і може призводити до небажаних мутацій даних. Для вирішення

цієї проблеми у проєкті Workvia.Tests використується бібліотека Microsoft.EntityFrameworkCore.InMemory. Так ми замінюємо основну БД на легку, безпосередньо в оперативній пам'яті ізольовано для кожного тестового сценарію. Завдяки допоміжному методу GetDbContext, кожен тест отримує чистий контекст з унікальним ідентифікатором, що дозволяє виконувати всі тести паралельно без конфліктів за ресурси.

Архітектура написання тестових методів базується на загальноприйнятому патерні AAA (Arrange, Act, Assert):

- Arrange (Підготовка) — створюється тестове оточення. У базу даних в пам'яті додаються фейкові записи про працівників (сутності ApplicationUser) та їхні існуючі робочі зміни. Також формується об'єкт запиту (ShiftRequestDTO).
- Act (Виконання) — відбувається виклик методу сервісу, який необхідно протестувати (наприклад, CreateShiftAsync або UpdateShiftAsync), із передачею підготовлених даних.
- Assert (Перевірка) — аналізується отриманий результат. Використовуються методи класу Assert (наприклад, Assert.True, Assert.Equal), щоб підтвердити, що фактична поведінка системи збігається з очікуваною.

Найбільша увага при покритті тестами була приділена алгоритму перевірки робочих змін. Реалізовано кілька критичних тестових сценаріїв:

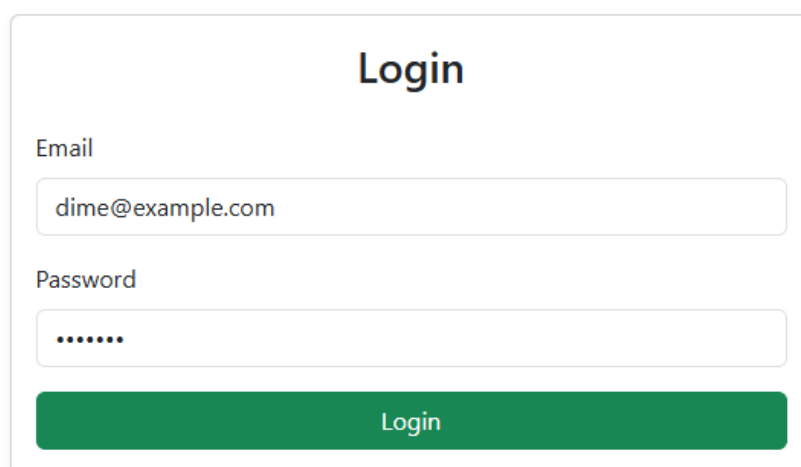
- CreateShiftAsync_Should_AddShiftToDatabase – перевіряє правильне виконання, підтверджуючи, що коректний запит успішно створює запис у базі даних і повертає ім'я працівника.
- CreateShiftAsync_Should_Fail_When_Overlap – перевіряє захисний механізм. У тестову базу додається зміна (наприклад, з 10:00 до 18:00), після чого робиться спроба створити нову зміну для того ж працівника з 17:00 до 20:00. Тест успішно проходить лише тоді, коли сервіс блокує операцію та повертає повідомлення «Shift overlaps with an existing shift».
- UpdateShiftAsync_Should_Update_When_Valid – перевіряє здатність системи коректно перепризначати робочу зміну на іншого працівника зі зміною часових рамок, підтверджуючи мутацію запису в базі.

Використання модульного тестування доводить, що ключовий алгоритм планування ресурсів працює безпомилково і надійно захищає систему від логічних колізій.

4.2 Опис інтерфейсу

Графічний інтерфейс користувача (GUI) спроектовано з акцентом на мінімалізм, швидкодію та адаптивність. Завдяки використанню фреймворку Bootstrap 5, застосунок коректно відображається як на настільних комп'ютерах, так і на мобільних пристроях. Взаємодія з системою розділена на два основні робочі простори залежно від ролі авторизованого користувача: панель адміністратора та панель працівника.

Точкою входу в систему є екран авторизації. Форма входу містить поля для введення електронної пошти та пароля (рисунок 4.2). Інтерфейс інтерактивно реагує на дії користувача: у разі введення некоректних даних (наприклад, неправильного формату пошти або хибного пароля), поля миттєво підсвічуються червоним кольором із відповідним текстовим підтвердженням помилки, не чекаючи перезавантаження сторінки.



The image shows a login form titled "Login" in a dark blue font. Below the title, there are two input fields. The first field is labeled "Email" and contains the text "dime@example.com". The second field is labeled "Password" and contains seven dots, indicating a masked password. Below these fields is a green button with the text "Login" in white. The entire form is enclosed in a light gray border with rounded corners.

Рисунок 4.2 — Сторінка авторизації користувача

Після успішної авторизації адміністратор потрапляє на головну інформаційну панель (Dashboard). Цей екран надає загальне уявлення про ефективність роботи підприємства за останні 30 днів. На сторінці розташовані віджети із загальною кількістю відпрацьованих годин, стовпчаста діаграма топ-5 найпродуктивніших працівників та кільцева діаграма розподілу навантаження по днях тижня (рисунк 4.3). Тут також розміщена кнопка «Generate Report», яка викликає спливаюче вікно для вибору часового періоду і подальшого завантаження Excel-звіту.

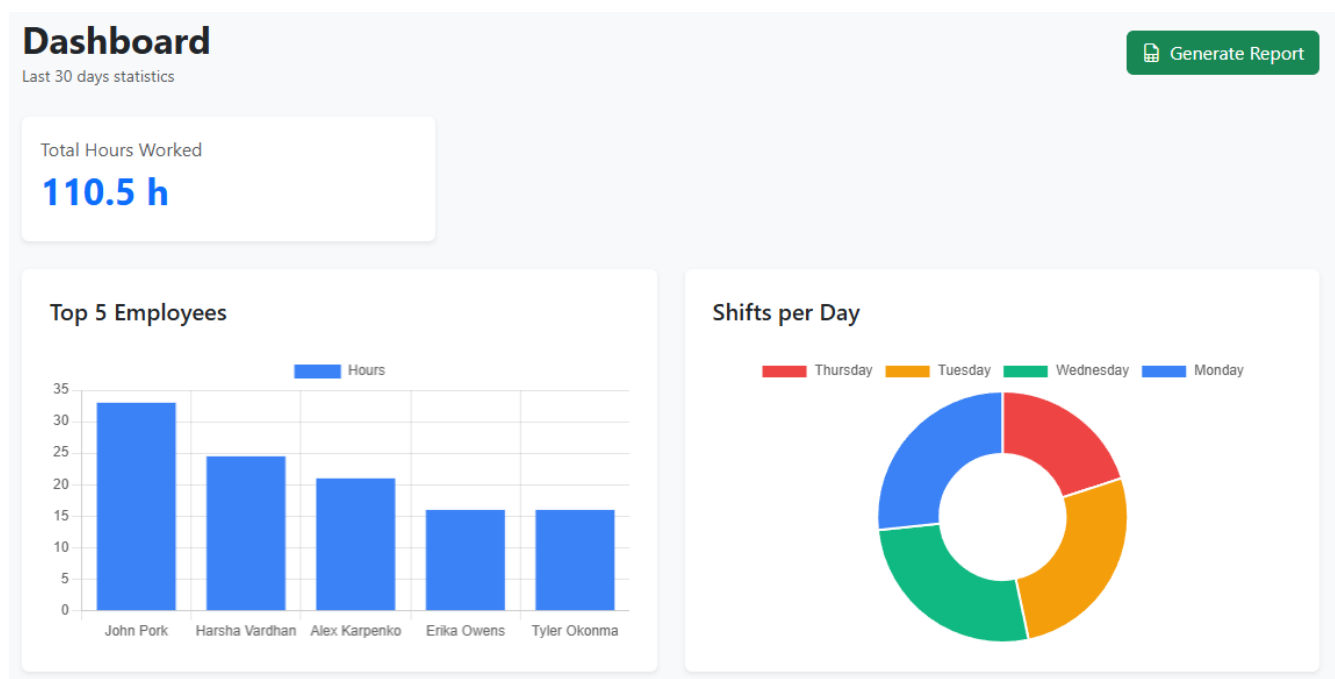


Рисунок 4.3 – Інформаційна панель адміністратора

Ключовим розділом системи є розклад робочих змін (Shifts). Інтерфейс цього розділу представлений повноцінним інтерактивним календарем. Кожна робоча зміна відображається у вигляді кольорового блоку, що містить ім'я працівника та часові рамки. Для зручності адміністратор може перемикаати вигляд календаря між місяцем, тижнем та конкретним днем. При натисканні на кнопку «Add Shift» або на існуючу зміну, відкривається модальне вікно (Modal). Використання модальних

вікон для CRUD-операцій дозволяє користувачеві зберігати контекст поточної сторінки, що є кращою практикою UX. Вікно містить випадаючий список з іменами працівників та поля для вибору дати і часу (DateTimePicker).

The screenshot shows a modal window titled "Edit Shift" with a close button (X). Inside the modal, there is a form with the following fields:

- Employee:** A dropdown menu showing "Alex Karpenko (alex@example.com)".
- Start Time:** A text input field containing "22.04.2026 16:00".
- End Time:** A text input field containing "22.04.2026 21:00".

Below the form, there is a calendar and a time picker. The calendar shows the month of April 2026, with the 22nd highlighted. The time picker shows a list of times from 19:30 to 22:30, with 21:00 selected. A red "Delete" button is on the left, and a blue "Changes" button is on the right.

Рисунок 4.4 – Модальне вікно редагування зміни

Для управління персоналом створено вкладку «Users» (рисунок 4.5). Він представляє собою табличний інтерфейс із переліком усіх зареєстрованих працівників. Таблиця містить кнопки для редагування облікових даних працівника або його видалення з системи. Реєстрація нового співробітника також відбувається через зручну модальну форму, де адміністратор вказує ім'я, пошту та первинний пароль.

Employees

 Register new employee

Name	Email	Actions	
John Pork	john@example.com		
Tyler Okonma	tyler@example.com		
Alex Karpenko	alex@example.com		
Harsha Vardhan	harsha@example.com		
Erika Owens	erika@example.com		

Рисунок 4.5 – Сторінка користувачів

На сторінці «Notifications» адміністратори можуть створювати нові повідомлення (рисунок 4.6) для всіх користувачів системи, що дозволяє централізовано повідомляти про будь які важливі події.

Create Notification

×

Title

Test title

Message

maximus mi vitae iaculis. Morbi eleifend augue dapibus posuere consectetur. Mauris faucibus dui eget leo malesuada, at porttitor libero luctus. Aenean egestas est vel velit auctor dignissim.

Cancel

Send

Рисунок 4.6 – Модальне вікно створення повідомлення

Інтерфейс працівника має власну навігаційну панель та ізольований набір маршрутів, що гарантує захист корпоративних даних від випадкового чи навмисного пошкодження. Взаємодія зводиться до двох основних модулів.

Головною сторінкою для працівника є «My Schedule» (рисунок 4.7). На відміну від адміністративної панелі, інтерактивний календар тут працює у режимі тільки для читання. Це означає, що працівник може зручно переглядати свій графік на місяць чи тиждень, але не має технічної можливості самовільно перемістити, скоротити або видалити призначену йому зміну. У верхній частині екрана виводиться персональна статистика відпрацьованих годин за поточний місяць.

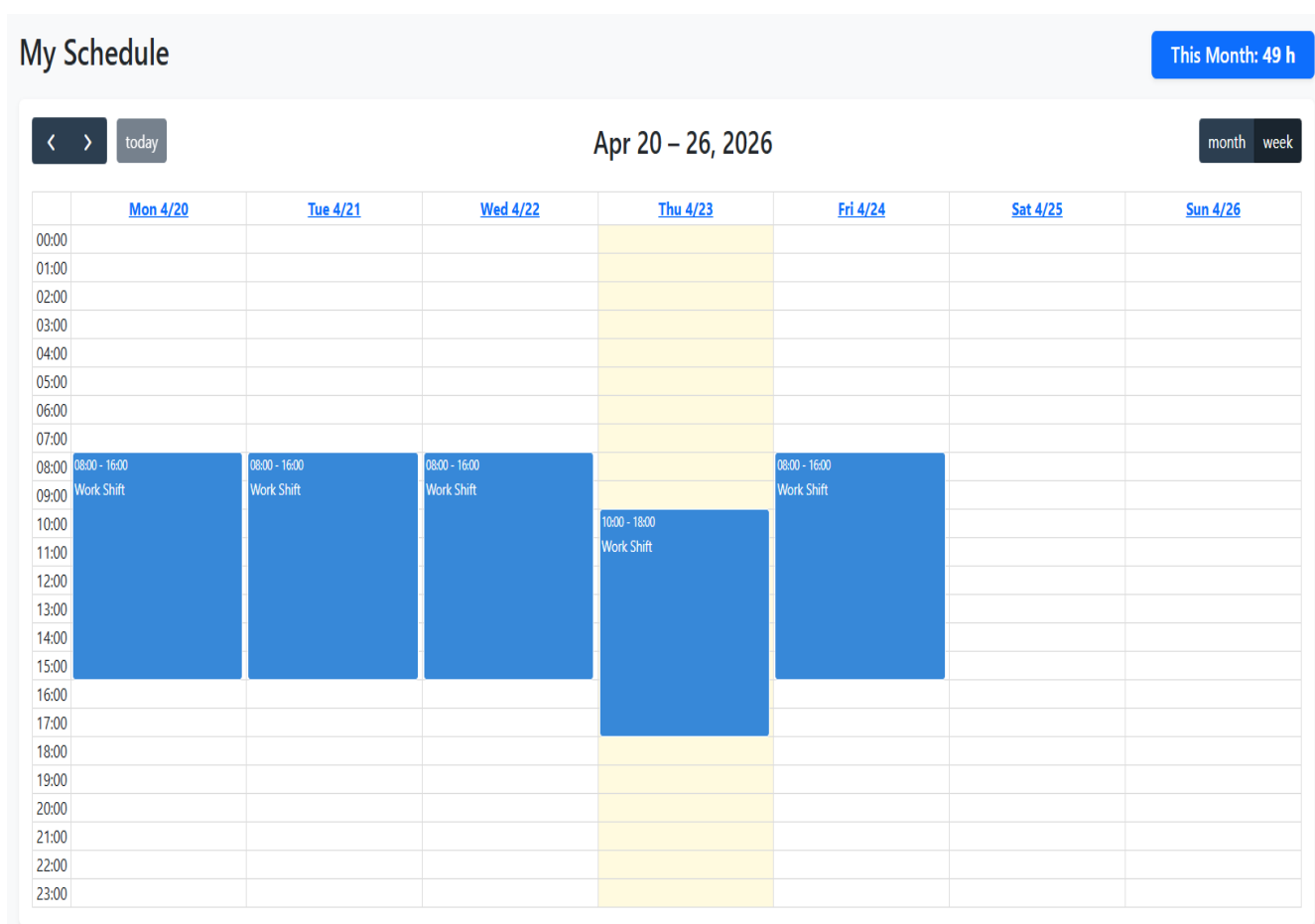


Рисунок 4.7 – Вигляд сторінки My Schedule

Для працівника інтерфейс вкладки повідомлень приховує кнопку створення нового повідомлення та іконки видалення біля кожного запису. Працівник бачить лише хронологічно відсортований список оголошень із чітким зазначенням дати та часу публікації, що забезпечує своєчасне інформування про важливі події компанії (рисунок 4.8).

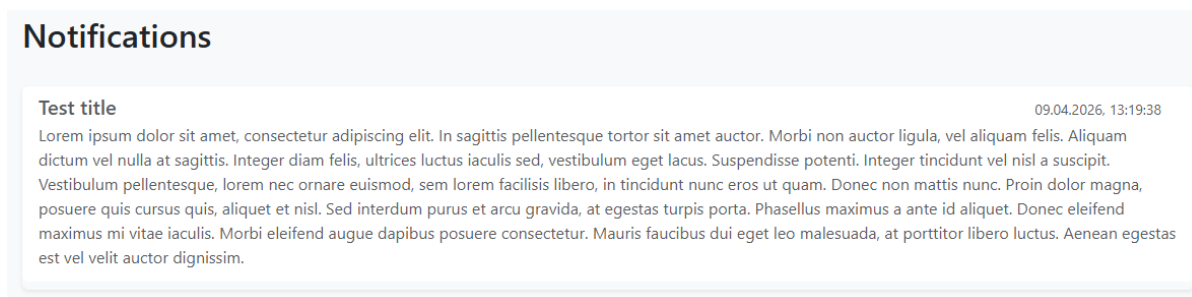


Рисунок 4.8 – Сторінка повідомлень користувача

Існує спільна для всіх користувачів вкладка «Preferences», що дозволяє змінювати пароль (рисунок 4.9).

A screenshot of a 'Change password' modal window. The window has a title bar with the text 'Change password' and a close button (X). Inside the window, there are three input fields: 'Current password', 'New password', and 'Confirm new password'. Each field contains a series of dots representing masked text. The 'Confirm new password' field is currently active, indicated by a blue border and a cursor. At the bottom right of the window is a green button labeled 'Change'.

Рисунок 4.9 – Модальне вікно зміни паролю

Завдяки глобальній обробці відповідей сервера (через бібліотеку react-toastify), будь-яка успішна дія (наприклад, зміна пароля) або помилка супроводжується акуратним спливаючим сповіщенням у нижньому правому куті екрана. Це забезпечує постійний і зрозумілий зворотний зв'язок між системою та користувачем, незалежно від його ролі.

ВИСНОВКИ

У ході виконання дипломної роботи було успішно вирішено актуальне завдання — спроектовано та розроблено спеціалізовану корпоративну вебсистему для управління трудовими ресурсами підприємства. Створений програмний продукт дозволяє автоматизувати рутинні процеси планування робочих змін, ведення обліку відпрацьованого часу та комунікації з персоналом.

На етапі аналізу предметної області було досліджено існуючі підходи до управління розкладом. Виявлено, що використання ручних методів або універсальних хмарних рішень не повною мірою задовольняє потреби підприємств через високий ризик помилок, проблеми з масштабованістю, або ж через потенційні загрози конфіденційності даних. На основі цих висновків було сформовано чіткі вимоги до власної системи з рольовим розподілом доступу між адміністраторами та працівниками.

Для реалізації поставлених завдань було обрано сучасний технологічний стек та застосовано архітектурний підхід Single Page Application (SPA). Серверну частину побудовано на базі платформи ASP.NET Core з використанням мови C#, а взаємодія з реляційною базою даних Microsoft SQL Server реалізована через ORM Entity Framework Core. Застосування принципів Clean Architecture забезпечило гнучкість, масштабованість та незалежність бізнес-логіки від зовнішніх інфраструктурних рішень. Клієнтська частина розроблена мовою TypeScript за допомогою бібліотеки React, що гарантує високу швидкодію та «реактивність» інтерфейсу. Безпека взаємодії між клієнтом та сервером надійно захищена механізмом безстанової JWT-автентифікації.

Під час програмної реалізації було успішно створено всі ключові модулі системи. Розроблено надійний алгоритм запобігання часовим колізіям, який на рівні інфраструктурного шару бази даних блокує спроби призначення працівнику змін, що перетинаються в часі. Створено модуль аналітики, який виконує агрегаційні обчислення безпосередньо засобами СУБД та візуалізує дані за

допомогою інтерактивних графіків. Також впроваджено функціонал автоматичної генерації та завантаження електронних Excel-звітів про продуктивність персоналу.

Якість та надійність розробленої серверної логіки підтверджена за допомогою модульного тестування критичних алгоритмів з використанням ізольованої бази даних в оперативній пам'яті. Графічний інтерфейс застосунку є ергономічним, адаптивним та містить суворе розмежування можливостей: адміністратори отримали повний контроль над даними за допомогою інтерактивного календаря та модальних вікон, тоді як працівникам надано зручний та безпечний простір для перегляду персонального графіка, статистики та корпоративних повідомлень.

Отже, впровадження розробленої системи дозволяє керівництву підприємства оптимізувати процес розподілу людського ресурсу, мінімізувати вплив людського фактора та забезпечити абсолютну прозорість нарахування робочих годин. Мета дипломної роботи досягнута повною мірою. Отриманий програмний продукт є повністю працездатним, відповідає всім початковим вимогам і створює надійний фундамент для подальшого розширення та інтеграції з іншими корпоративними інформаційними системами.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Крушельницька О. В., Мельничук Д. П. Управління персоналом : навч. посіб. 2-ге вид., перероб. й доп. Київ : Кондор, 2005. 308 с.
2. Савченко В. А. Управління розвитком персоналу : навч. посіб. Київ : КНЕУ, 2004. 351 с.
3. Балабанова Х. В. Системи управління трудовими ресурсами підприємства в умовах цифровізації. Економіка та управління підприємствами. 2023. № 4. С. 45–51.
4. Гавриленко Т. В., Бойко О. В. Проєктування інформаційних систем управління персоналом та розкладом. Вісник комп'ютерних технологій. 2024. № 12. С. 112–118.
5. Overview of ASP.NET Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/> (дата звернення: 10.05.2026).
6. Price M. J. C# 12 and .NET 8 – Modern Cross-Platform Development Fundamentals. 8th ed. Birmingham : Packt Publishing, 2023. 822 p.
7. Freeman A. Pro ASP.NET Core 7: Develop Cloud-Ready Web Applications. 10th ed. New York : Apress, 2023. 1200 p.
8. Introduction to Identity on ASP.NET Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/aspnet/core/security/authentication/identity> (дата звернення: 11.05.2026).
9. Introduction to JSON Web Tokens. JWT.IO. URL: <https://jwt.io/introduction> (дата звернення: 12.05.2026).
10. Ben-Gan A. T-SQL Fundamentals. 3rd ed. Redmond : Microsoft Press, 2016. 432 p.
11. Introduction to Entity Framework Core. Microsoft Learn. URL: <https://learn.microsoft.com/en-us/ef/core/> (дата звернення: 14.05.2026).
12. TypeScript: Typed JavaScript at Any Scale. TypeScript. URL: <https://www.typescriptlang.org/> (дата звернення: 12.05.2026).

13. Cherny B. Programming TypeScript: Making Your JavaScript Applications Scale. Sebastopol : O'Reilly Media, 2019. 320 p.
14. React – A JavaScript library for building user interfaces. React. URL: <https://react.dev/> (дата звернення: 15.05.2026).
15. Banks A., Porcello E. Learning React: Modern Patterns for Developing React Apps. 2nd ed. Sebastopol : O'Reilly Media, 2020. 336 p.
16. Bootstrap – The most popular HTML, CSS, and JS library in the world. Bootstrap. URL: <https://getbootstrap.com/> (дата звернення: 15.05.2026).
17. React Router: Declarative Routing for React.js. React Router. URL: <https://reactrouter.com/> (дата звернення: 16.05.2026).
18. React Hook Form – Performant, flexible and extensible forms with easy-to-use validation. React Hook Form. URL: <https://react-hook-form.com/> (дата звернення: 16.05.2026).
19. Axios: Promise based HTTP client for the browser and node.js. Axios. URL: <https://axios-http.com/> (дата звернення: 16.05.2026).
20. Chart.js – Open source HTML5 Charts for your website. Chart.js. URL: <https://www.chartjs.org/> (дата звернення: 18.05.2026).
21. FullCalendar – JavaScript Event Calendar. FullCalendar. URL: <https://fullcalendar.io/> (дата звернення: 18.05.2026).
22. Мартін Р. Ч. Чиста архітектура: мистецтво розробки програмного забезпечення / пер. з англ. Київ : Фабула, 2019. 368 с.
23. Evans E. Domain-Driven Design: Tackling Complexity in the Heart of Software. Boston : Addison-Wesley Professional, 2003. 560 p.
24. Cross-Origin Resource Sharing (CORS). MDN Web Docs. URL: <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS> (дата звернення: 20.05.2026).
25. Vite – Next Generation Frontend Tooling. Vite. URL: <https://vitejs.dev/> (дата звернення: 20.05.2026).
26. Flanagan D. JavaScript: The Definitive Guide. 7th ed. Sebastopol : O'Reilly Media, 2020. 706 p.

ДОДАТОК А

«Корпоративна веб-система управління робочими змінами та персоналом»

НТУУ «КПІ ім. Ігоря Сікорського»

НН ІАТЕ

Кафедра ЦТЕ

Група ТР-25

Реалізація сервісу робочих змін та конфігурація серверу

Мова програмування: С#

Аркушів 9

```
using Microsoft.EntityFrameworkCore;
using System;
using System.Collections.Generic;
using System.Text;
using Workvia.Core.DTO;
using Workvia.Core.Mappers;
using Workvia.Infrastructure.DatabaseContext;
using Workvia.Core.ServiceContracts;

namespace Workvia.Infrastructure.Services
{
    public class ShiftService : IShiftService
    {
        private readonly ApplicationDbContext _context;

        public ShiftService(ApplicationDbContext context)
        {
            _context = context;
        }

        public async Task<IEnumerable<ShiftResponseDTO>> GetAllShiftsAsync()
        {
            var shifts = await _context.Shifts
                .Include(s => s.Employee)
                .ToListAsync();

            return shifts.Select(s => s.ToResponseDTO()).ToList();
        }
    }
}
```

```

        public async Task<IEnumerable<ShiftResponseDTO>>
GetShiftsByEmployeeAsync(Guid employeeId)
    {
        var shifts = await _context.Shifts
            .Where(shift => shift.EmployeeID == employeeId)
            .Include(s => s.Employee)
            .ToListAsync();

        return shifts.Select(s => s.ToResponseDTO());
    }

    public async Task<ShiftResponseDTO?> GetShiftByIdAsync(Guid id)
    {
        var shift = await _context.Shifts
            .Include(s => s.Employee)
            .FirstOrDefaultAsync(s => s.ShiftID == id);

        return shift?.ToResponseDTO();
    }

    public async Task<(bool Succeeded, string Error, ShiftResponseDTO? Shift)>
CreateShiftAsync(ShiftRequestDTO shiftRequestDTO)
    {
        if (await IsOverlapAsync(shiftRequestDTO.EmployeeID,
shiftRequestDTO.StartTime, shiftRequestDTO.EndTime))
        {
            return (false, "Shift overlaps with an existing shift", null);
        }

        shiftRequestDTO.ShiftID = Guid.NewGuid();
    }

```

```

var shiftEntity = shiftRequestDTO.ToEntity();

_context.Shifts.Add(shiftEntity);
await _context.SaveChangesAsync();

await _context.Entry(shiftEntity).Reference(s => s.Employee).LoadAsync();

return (true, string.Empty, shiftEntity.ToResponseDTO());
}

public async Task<(bool Succeeded, string Error)> UpdateShiftAsync(Guid id,
ShiftRequestDTO shiftDTO)
{
    if (id != shiftDTO.ShiftID)
        return (false, "Mismatch ID");

    var existingShift = await _context.Shifts.FindAsync(id);
    if (existingShift == null)
        return (false, "Shift not found");

    if (await IsOverlapAsync(shiftDTO.EmployeeID, shiftDTO.StartTime,
shiftDTO.EndTime, id))
    {
        return (false, "Shift overlaps with an existing shift");
    }

    existingShift.EmployeeID = shiftDTO.EmployeeID;
    existingShift.StartTime = shiftDTO.StartTime;
    existingShift.EndTime = shiftDTO.EndTime;

```

```

try
{
    await _context.SaveChangesAsync();
    return (true, string.Empty);
}
catch (DbUpdateConcurrencyException)
{
    return (false, "Concurrency Error");
}
}

```

```

public async Task<bool> DeleteShiftAsync(Guid id)
{
    var shift = await _context.Shifts.FindAsync(id);
    if (shift == null)
        return false;

    _context.Shifts.Remove(shift);
    await _context.SaveChangesAsync();
    return true;
}

```

```

private async Task<bool> IsOverlapAsync(Guid employeeId, DateTime start,
DateTime end, Guid? excludeShiftId = null)
{
    return await _context.Shifts.AnyAsync(s =>
        s.EmployeeID == employeeId &&
        s.ShiftID != excludeShiftId &&
        s.StartTime < end &&
        s.EndTime > start
    );
}

```

```

        );
    }
}
}

```

```

using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Authorization;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using Workvia.Core.Identity;
using Workvia.Core.Services;
using Workvia.Core.ServiceContracts;
using Workvia.Infrastructure.DatabaseContext;
using Workvia.Infrastructure.Services;

```

```

var builder = WebApplication.CreateBuilder(args);

```

```

// Filters

```

```

builder.Services.AddControllers(options =>

```

```

{

```

```

    options.Filters.Add(new ProducesAttribute("application/json"));

```

```

    options.Filters.Add(new ConsumesAttribute("application/json"));

```

```

    var policy = new AuthorizationPolicyBuilder().RequireAuthenticatedUser().Build();

```

```

    options.Filters.Add(new AuthorizeFilter(policy));
});

// Services
builder.Services.AddTransient<IJwtService, JwtService>();
builder.Services.AddScoped<IAuthService, AuthService>();
builder.Services.AddScoped<IUserService, UserService>();
builder.Services.AddScoped<IShiftService, ShiftService>();
builder.Services.AddScoped<IStatisticsService, StatisticsService>();
builder.Services.AddScoped<INotificationService, NotificationService>();
builder.Services.AddScoped<IReportService, ReportService>();

builder.Services.AddDbContext<ApplicationDbContext>(options =>
{
    options.UseSqlServer(builder.Configuration.GetConnectionString("Default"));
});

builder.Services.AddEndpointsApiExplorer();

// Swagger
builder.Services.AddSwaggerGen(options =>
{
    options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory,
"api.xml"));
});

// Identity
builder.Services.AddIdentity<ApplicationUser, ApplicationRole>(options =>

```

```

{
    options.Password.RequiredLength = 5;
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireDigit = true;
    options.Password.RequireUppercase = true;
    options.Password.RequireLowercase = true;
})

.AddEntityFrameworkStores<ApplicationDbContext>()
.AddDefaultTokenProviders()
.AddUserStore<UserStore<ApplicationUser, ApplicationRole,
ApplicationDbContext, Guid>>()
.AddRoleStore<RoleStore<ApplicationRole, ApplicationDbContext, Guid>>());

// Cors
builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(policy =>
    {
        policy.WithOrigins("http://localhost:5173")
            .AllowAnyHeader()
            .AllowAnyMethod()
            .AllowCredentials();
    });
});

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;

```

```

options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
}).AddJwtBearer(options =>
{
options.TokenValidationParameters = new TokenValidationParameters()
{
ValidateAudience = true,
ValidAudience = builder.Configuration["Jwt:Audience"],
ValidateIssuer = true,
ValidIssuer = builder.Configuration["Jwt:Issuer"],
ValidateLifetime = true,
ValidateIssuerSigningKey = true,
IssuerSigningKey = new
SymmetricSecurityKey(System.Text.Encoding.UTF8.GetBytes(builder.Configuration["
Jwt:Key"]))
};

options.Events = new JwtBearerEvents
{
OnMessageReceived = context =>
{
if (context.Request.Cookies.TryGetValue("access_token", out var token))
{
context.Token = token;
}
return Task.CompletedTask;
}
};
});

```

```
builder.Services.AddAuthorization();
var app = builder.Build();

// Configure the HTTP request pipeline.
if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseHsts();
app.UseHttpsRedirection();
app.UseRouting();
app.UseCors();
app.UseAuthentication();
app.UseAuthorization();

//Role seed
using (var scope = app.Services.CreateScope())
{
    await RoleSeeder.SeedRoles(scope.ServiceProvider);
}

app.MapControllers();

app.Run();
```

ДОДАТОК Б

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

СУЧАСНІ ПРОБЛЕМИ НАУКОВОГО ЗАБЕЗПЕЧЕННЯ ЕНЕРГЕТИКИ: БЕЗПЕКА, СТІЙКІСТЬ, ІТ ТА ЕКОЛОГІЧНИЙ МОНІТОРИНГ

XXIII Міжнародна науково-практична конференція
молодих вчених та студентів

До 40-річчя Чорнобильської катастрофи

17–22 квітня 2026 року, м. Київ



Київ-2026

УДК 004:005.95

¹ Бакалаврант 4 курсу Карпенко Д. І.

² Доц., к.ф.-м.н. Донець А. Г.

<https://scholar.google.com.ua/citations?user=IDeYM00AAAAJ>

³ КПІ ім. Ігоря Сікорського

КОРПОРАТИВНА ВЕБ-СИСТЕМА УПРАВЛІННЯ РОБОЧИМИ ЗМІНАМИ ТА ПЕРСОНАЛОМ

Постановка проблеми та її актуальність. У сучасних умовах управління персоналом вимагає автоматизації розподілення людського ресурсу. Використання застарілих методів призводить до помилок в організації робочого процесу, накладання змін та складнощів у підрахунку відпрацьованих годин. Веб-система управління персоналом має вирішувати ці проблеми шляхом створення єдиного цифрового простору для менеджерів та співробітників. Система має оптимізувати процес планування змін завдяки зручному інтерфейсу, мінімізувати помилки при складанні графіку та забезпечити прозорість нарахування робочих годин для співробітників.

Аналіз останніх досліджень. Інформаційне забезпечення менеджменту найчастіше реалізується за принципами «чистої архітектури» (Clean Architecture), що забезпечує гнучкість, тестованість та легкість підтримки коду [1].

Серверна частина створена з використанням платформи ASP.NET Core Web API. Розподілена на 3 рівні (Core, Infrastructure, WebApi). Використовується ORM Entity Framework Core в поєднанні з СУБД Microsoft SQL Server [2]. Автентифікація реалізується засобами ASP.NET Core Identity та JWT (JSON Web Token). Для зручного тестування і подальшого розширення функціоналу додаються тести з бібліотекою xUnit.

SPA (Single Page Application) реалізовано з використанням бібліотеки React [3]. З метою спрощення реалізації використовуються UI компоненти Bootstrap, FullCalendar та react-chartjs-2 (діаграми та графіки). Взаємодія з бекендом через HTTP запити побудована на основі бібліотеки Axios.

Формулювання мети. Метою та навіть необхідною умовою існування сучасних бізнес-організацій є існування спеціалізованої корпоративної веб-системи, спрямованої на цифрову трансформацію процесів управління трудовими ресурсами підприємства. На сучасному етапі економічного розвитку вимоги до таких систем не обмежуються лише ліквідацією недоліків, якісною візуалізацією та захистом. Головною концептуальною метою є розробка програмної логіки для розподілу часових ресурсів у режимі «нон-стоп» та можливості постійного автоматичного оновлення.

Основна частина. Логіка роботи системи поділяється на кілька основних функціональних модулів, доступ до яких регулюється підсистемою ролівої авторизації.

Механізм автентифікації та розмежування прав доступу. Робота системи розпочинається з процедури ідентифікації користувача. На основі введених облікових даних сервер генерує JWT-токен, який містить зашифровану інформацію про ідентифікатор користувача та його роль. Цей токен використовується для підписання кожного наступного HTTP-запиту, що забезпечує захист кінцевих точок API та розмежування доступу до функціоналу на рівні серверної логіки.

Модуль планування робочого часу. Ключовим елементом системи є інтерактивний календар. Адміністратор ініціює створення робочих змін, вказуючи працівника та часовий інтервал. Система автоматично виконує валідацію даних на стороні сервера: перевіряється коректність дат та наявність перетинів з вже існуючими змінами даного працівника. У разі виявлення накладання змін, транзакція відхиляється, а користувач отримує відповідне сповіщення. У свою чергу працівник може переглядати лише свої робочі зміни. Дані про зміни відображаються у вигляді подій на календарній сітці з можливістю перегляду в розрізі тижня або місяця.

Аналітичний модуль та статистика. Система в реальному часі опрацьовує дані про відпрацьовані зміни та формує статистичні звіти:

- загальний облік – розрахунок сумарної кількості відпрацьованих годин за обраний період.
- рейтингування – формування списку «Топ працівників» на основі навантаження.
- розподіл навантаження – візуалізація кількості змін по днях тижня за допомогою діаграм, що дозволяє виявляти пікові навантаження та оптимізувати штатний розклад.

Управління персоналом. Адміністративний модуль дозволяє здійснювати повний цикл CRUD-операцій над обліковими записами співробітників, що дозволяє оперативно актуалізувати базу даних персоналу відповідно до кадрових змін на підприємстві.

Підсистема корпоративних сповіщень. Для забезпечення оперативної комунікації в системі реалізовано механізм глобальних оголошень. Адміністратор має можливість створювати текстові повідомлення, які зберігаються в базі даних та автоматично транслиуються всім користувачам системи. Це забезпечує централізоване інформування персоналу про важливі події, зміни в графіку або організаційні розпорядження без необхідності використання сторонніх месенджерів.

Структура бази даних. Маємо наступні таблиці БД – робочих змін, користувачів та їх ролей і повідомлень. Основний акцент робиться на автоматизації планування робочих змін, забезпеченні прозорого та точного обліку робочого часу, а також на наданні інструментів для детального аналізу ефективності персоналу задля оптимізації управлінських рішень. Для досягнення поставленої мети необхідно вирішити наступний комплекс завдань:

1. Дослідити існуючі підходи до управління графіками роботи, виявити недоліки ручного планування та визначити ключові вимоги до автоматизованої системи.
2. Розробити архітектуру програмного забезпечення на основі принципів «Clean Architecture» для забезпечення масштабованості, а також спроектувати реляційну базу даних для зберігання інформації про користувачів, зміни та корпоративні оголошення.
3. Розробити підсистему автентифікації та авторизації користувачів, що забезпечить чіткий розподіл прав доступу та функціональних можливостей між ролями.
4. Створити інтерактивний календарний інтерфейс для візуалізації робочого навантаження, що дозволить адміністраторам оперативно створювати, редагувати та переглядати зміни як у розрізі всього підприємства, так і для окремих співробітників.
5. Розробити та впровадити програмну логіку для автоматичного виявлення та запобігання конфліктам у розкладі (часові накладання змін, перевищення лімітів робочого часу).
6. Реалізувати модуль генерації статистики, який візуалізує дані про продуктивність працівників та розподіл навантаження по днях тижня у вигляді графіків та діаграм, а також забезпечує формування звітів.
7. Впровадити функціонал глобальних корпоративних оголошень, що дозволить адміністрації централізовано інформувати персонал про важливі події безпосередньо через інтерфейс веб-додатку.
8. Провести перевірку коректності роботи всіх модулів системи, зокрема валідації даних та безпеки доступу до API.

Висновки. Веб-додаток, призначений для автоматизації процесів складання розкладу робочих змін, обліку робочого часу та управління людськими ресурсами, вирішує актуальну проблему неефективності застарілих методів планування, пропонуючи сучасне цифрове рішення з єдиним інформаційним простором, що забезпечує прозорість нарахування відпрацьованих годин та значно скорочує час, який витрачається менеджерами на складання графіків. Використання розробленого веб-додатку дозволяє оптимізувати бізнес-процеси підприємства, підвищити дисципліну та ефективність управління персоналом.

Перелік посилань:

1. ESPOSITO, Dino. Clean Architecture with .NET. Microsoft Press, 2024. 336 p.
2. Документація Entity Framework Core. URL: <https://learn.microsoft.com/en-us/ef/core/> (дата звернення: 25.02.2026).
3. Banks A., Porcello E. Learning React. Sebastopol, CA : O'reilly. 2020. 295 p.