МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

МАТЕМАТИЧНИЙ ФАКУЛЬТЕТ
Кафедра комп'ютерних наук

КВАЛІФІКАЦІЙНА РОБОТА БАКАЛАВРА

на тему: **«РОЗРОБКА ІНФОРМАЦІЙНОЇ СИСТЕМИ УПРАВЛІННЯ ВНУТРІШНІМИ РОБОЧИМИ ЗАВДАННЯМИ ТА АНАЛІЗУ ВИКОРИСТАННЯ РОБОЧОГО ЧАСУ СПІВРОБІТНИКАМИ ПІДПРИЄМСТВА»**

Виконав: студент 4 курсу, групи 6.1222-з
спеціальності 122 Комп'ютерні науки
(шифр і назва спеціальності)
освітньої програми Комп'ютерні науки
(назва освітньої програми)
А.О. Коваль
(ініціали та прізвище)
Керівник старший викладач кафедри комп'ютерних наук, Циммерман Г.А
(посада, вчене звання, науковий ступінь, прізвище та ініціали)
Рецензент професор кафедри програмної інженерії, к.ф.-м.н., доцент Кудін О.В.
(посада, вчене звання, науковий ступінь, прізвище та ініціали)

Запоріжжя 2026

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАПОРІЗЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет математичний

Кафедра комп'ютерних наук

Рівень вищої освіти бакалавр

Спеціальність 122 Комп'ютерні науки
(шифр і назва)

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
комп'ютерних наук, к.пед.н.,
доцент

Пшенична
О.С.
(підпис)

" ____ " _____ 2026 р.

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТОВІ

Ковалю Андрію Олексійовичу
(прізвище, ім'я та по-батькові)

1. Тема роботи Розробка інформаційної системи управління внутрішніми робочими завданнями та аналізу використання робочого часу співробітниками підприємства.

керівник роботи Циммерман Геннадій Анатолійович
(прізвище, ім'я та по-батькові, науковий ступінь, вчене звання)

затверджені наказом ЗНУ від « 02 » грудня 2025 року № 1742-с

2. Строк подання студентом роботи 15.05.2026

3. Вихідні дані до роботи 1. Постановка задачі.
2. Перелік літератури.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Постановка задачі.
2. Основні теоретичні відомості.

5. Перелік графічного матеріалу (з точним зазначенням обов'язкових креслень)
презентація

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1	Матвіїшина Н.В., доцент кафедри комп'ютерних наук, к.т.н.		
2	Матвіїшина Н.В., доцент кафедри комп'ютерних наук, к.т.н.		
3	Матвіїшина Н.В., доцент кафедри комп'ютерних наук, к.т.н.		
4	Матвіїшина Н.В., доцент кафедри комп'ютерних наук, к.т.н.		

7. Дата видачі завдання 05.12.2025

К А Л Е Н Д А Р Н И Й П Л А Н

№	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1.	Розробка плану роботи.	10.12.2025	
2.	Збір вихідних даних.	19.12.2025	
3.	Обробка методичних та теоретичних джерел.	22.12.2025	
4.	Розробка першого та другого розділу.	16.01.2026	
5.	Розробка третього та четвертого розділу.	27.03.2026	
6.	Оформлення та нормоконтроль	06.04.2026	
	кваліфікаційної роботи бакалавра.		
7.	Захист кваліфікаційної роботи.	29.05.2026	

Студент _____ А.О Коваль
(підпис) (ініціали та прізвище)

Керівник роботи _____ Г. А. Циммерман
(підпис) (ініціали та прізвище)

Нормоконтроль пройдено

Нормоконтролер _____ О.Г.Спиця
(підпис) (ініціали та прізвище)

РЕФЕРАТ

Кваліфікаційна робота «Розробка інформаційної системи управління внутрішніми робочими завданнями та аналізу використання робочого часу співробітниками підприємства» 74 с., 22 рис., 4 табл., 14 джерел, 2 додатки.

АВТОМАТИЗАЦІЯ БІЗНЕС-ПРОЦЕСІВ, АНАЛІЗ РОБОЧОГО ЧАСУ, БАЗА ДАНИХ, ІНФОРМАЦІЙНА СИСТЕМА, ОБЛІК РОБОЧОГО ЧАСУ, РОБОЧЕ ЗАВДАННЯ, УПРАВЛІННЯ ЗАВДАННЯМИ.

Об’єкт дослідження – процес планування, обліку та контролю виконання внутрішніх робочих завдань співробітників підприємства.

Предмет дослідження – автоматизація процесів управління завданнями та обліку робочого часу.

Мета роботи – розробка інформаційної системи, яка забезпечує постановку та розподіл задач між працівниками, фіксацію фактичного часу роботи, контроль статусів виконання, а також формування аналітичних звітів щодо ефективності використання робочого часу й навантаження персоналу.

Метод дослідження – аналітичний, практичний. У процесі виконання роботи застосовано підходи об’єктно-орієнтованого проєктування, моделювання бізнес-логіки та даних, проєктування реляційної бази даних, а також реалізацію програмного застосунку мовою C# на платформі .NET. Для зберігання інформації про користувачів, задачі, облік часу та звітність використано СКБД.

У результаті роботи створено прототип інформаційної системи для внутрішнього управління задачами та обліку робочого часу, що підтримує: створення й призначення задач, ведення журналу часу, зміну статусів виконання, фільтрацію та пошук, а також базову аналітику продуктивності.

SUMMARY

Bachelor's Qualifying Theses «Development of an information system for managing internal work tasks and analyzing the use of working time by company employees»: 74 pages, 22 figures, 4 tables, 14 references, 2 supplements.

BUSINESS PROCESS AUTOMATION, WORKING TIME ANALYSIS, DATABASE, INFORMATION SYSTEM, WORKING TIME ACCOUNTING, WORK TASK, TASK MANAGEMENT.

Object of research – the process of planning, accounting, and controlling the execution of internal work tasks of enterprise employees.

Subject of research – automation of task management processes and working time accounting.

Purpose of the work – to develop an information system that provides task assignment and distribution among employees, recording of actual working time, monitoring of execution statuses, and generation of analytical reports on the efficiency of working time usage and personnel workload.

Research methods – analytical and practical. The study applies object-oriented design approaches, business logic and data modeling, relational database design, and implementation of a software application using C# on the .NET platform. A database management system (DBMS) is used to store information about users, tasks, time tracking, and reporting.

Results – as a result of the work, a prototype of an information system for internal task management and working time tracking has been developed. The system supports task creation and assignment, time logging, status management, filtering and search, as well as basic performance analytics.

ЗМІСТ

Завдання на кваліфікаційну роботу 2
Реферат 4
Summary 5
Скорочення та умовні позначки 7
Вступ 9
1 Теоретичні основи 11
1.1 Системи управління завданнями 11
1.1.1 Визначення та ключові поняття 11
1.1.2 Класифікація систем управління 11
1.2 Облік та аналіз використання робочого часу 13
1.2.1 Методи обліку робочого часу 14
1.2.2 Різниця між методами обліку робочого часу 14
1.2.3 Ключові метрики оцінки використання робочого часу 15
1.2.4 Роль аналізу робочого часу в управлінських рішеннях 16
1.3 Архітектурні патерни для десктопних інформаційних систем 17
1.3.1 Клієнт-серверна архітектура 17
1.3.2 Переваги клієнт-серверної архітектури 19
1.3.3 Інтерфейс взаємодії між клієнтом та сервером 19
1.3.4 Патерн організації інтерфейсу користувача 20
1.4 Порівняння існуючих рішень 21
1.4.1 Огляд Jira 21
1.4.2 Огляд Asana 22
1.4.3 Огляд Trello 23
1.5 Постановка задачі на розробку інформаційної системи 24
2 Аналіз вимог та специфікація програмної системи 26
2.1 Бізнес-процеси та користувачі підприємства 26
2.2 Сценарій виконання бізнес-процесу 27

2.3 Функціональні вимоги 29
3 Проєктування системи 31
3.1 Високорівнена архітектура 31
3.1.1 Вибір технологічного стеку 31
3.1.2 Архітектура модулів системи 32
3.2 Проєктування бази даних 34
3.2.1 Вибір системи керування базою даних 34
3.2.2 Створення структури БД 35
3.3 Проєктування API 39
3.4 Архітектура клієнта MVVM 41
3.5 Проєктування підсистеми аналітики та експорту звітності 42
3.6 Проєктування підсистеми взаємодії у реальному часі 43
4 Реалізація системи 45
4.1 Реалізація серверної частини 45
4.1.1 Конфігурація 45
4.1.2 Реалізація доступу до БД 46
4.1.3 Реалізація REST-контролерів та бізнес-логіки 47
4.2 Реалізація клієнтської частини 49
4.2.1 Архітектура MVVM та механізм навігації 49
4.2.2 Сервіси клієнта та обробка даних 51
4.2.3 Реалізація взаємодії у реальному часі 52
4.3 Реалізація аналітики та експорту 53
4.3.1 Підрахунок продуктивності 53
4.3.2 Формування аналітичних моделей передачі даних 55
4.3.3 Формування звітів та графіків 55
4.4 Реалізація користувацького інтерфейсу 56
Висновки 65
Перелік посилань 67
Додаток А Серверна частина інформаційної системи 69
Додаток Б Діаграма архітектурних зв’язків у клієнтському додатку 74

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАЧКИ

БД	–	База даних
СКБД	–	Система керування базами даних
API	–	Application Programming Interface
HTTP	–	HyperText Transfer Protocol, протокол передавання гіпертексту
JWT	–	JSON Web Token, формат токена для безпечної передачі Інформації
REST	–	Representational State Transfer, архітектурний стиль вебсервісів
C#	–	мова програмування на платформі .NET.
WPF	–	Windows Presentation Foundation, графічна підсистема для побудови інтерфейсів користувача.
MVVM	–	Model-View-ViewModel, архітектурний патерн для розділення логіки та інтерфейсу.
EF Core	–	Entity Framework Core, об’єктно-реляційне відображення для роботи з базами даних.
SignalR	–	бібліотека для забезпечення обміну даними між сервером та клієнтом у реальному часі.
DTO	–	Data Transfer Object, об’єкт передачі даних.

ВСТУП

У сучасних умовах цифровізації бізнесу ефективність роботи підприємства значною мірою залежить від того, наскільки прозоро організовані внутрішні процеси: постановка задач, контроль виконання, розподіл навантаження та аналіз використання робочого часу. За відсутності єдиного інструменту управління завданнями компанії часто стикаються з проблемами дублювання робіт, втрати домовленостей, складності оцінки реального прогресу, а також неможливості об’єктивно проаналізувати витрати часу на типові операції. Це знижує продуктивність, ускладнює планування та може призводити до зайвих витрат [1].

Метою кваліфікаційної роботи є розробка прототипу інформаційної системи управління внутрішніми робочими завданнями та аналізу використання робочого часу співробітниками підприємства, яка забезпечить централізоване ведення задач, облік часу та формування аналітичних звітів для керівництва й команди. Тобто робота передбачає розв’язання комплексної архітектурної проблеми шляхом створення клієнт-серверної системи, яка забезпечуватиме зберігання, обробку та управління інформацією про внутрішні робочі завдання, співробітників, облік витраченого часу, статуси виконання та інші ключові елементи роботи підприємства. Необхідно спроектувати реляційну базу даних, розробити серверну частину у вигляді захищеного REST API (ASP.NET Core) та створити сучасний настільний клієнтський додаток (WPF) з використанням патерна MVVM.

Для реалізації поставленої задачі використано мову програмування C# та платформу .NET, а для збереження даних – реляційну систему керування базами даних PostgreSQL.

Розроблений прототип інформаційної системи у підсумку призначений для надання зручних засобів керування внутрішніми робочими завданнями та обліку робочого часу співробітників. Система орієнтована на підвищення

контрольованості виконання робіт завдяки централізованому зберіганню інформації про задачі, статуси їх виконання та витрачений час. Система призначена для спрощення аналізу завантаженості співробітників шляхом формування аналітичних звітів, а також для підтримки безпечної обробки даних завдяки механізмам авторизації та розмежування доступу.

1 ТЕОРЕТИЧНІ ОСНОВИ

1.1 Системи управління завданнями

1.1.1 Визначення та ключові поняття

Система управління завданнями – це програмне забезпечення для планування, організації, делегування та контролю виконання робіт. Вона об’єднує в собі різний функціонал, зокрема – управління завданнями окремих співробітників, завданням команд, звіти по завданнях, тайм-трекінг.

Тайм-трекінг (англ. Time tracking) – процес обліку та відстеження витрат робочого часу на виконання завдань.

У межах розроблюваної інформаційної системи використовуються наступні основні поняття:

– завдання – це найменша структурна одиниця роботи в системі, що має назву, опис, термін виконання та відповідального виконавця;
– проєкт – це групування завдань за спільною метою або напрямком діяльності;
– статус – це етап життєвого циклу завдання;
– пріоритет – важливість завдання відносно усіх інших;
– оцінка часу – прогнозований час, який необхідний для виконання завдання;
– часовий запис – фактичний інтервал часу, витрачений на завдання.

1.1.2 Класифікація систем управління

Системи управління завданнями можна класифікувати за кількома основними ознаками.

За цільовою аудиторією системи управління поділяються на персональні та командні:

– персональні системи орієнтовані на підвищення індивідуальної продуктивності користувача. До таких систем належать різні To-Do Lists сервіси, наприклад такі як Microsoft To Do;
– командні системи більш орієнтовані на організацію спільної роботи декількох користувачів, розподіл ресурсів та звітність. Прикладами таких систем є Jira, Asana тощо.

За моделлю розгортання системи управління поділяються на хмарні (SaaS) та локальні (On-premise/LAN):

– хмарні зберігаються на серверах провайдера;
– локальні встановлюються безпосередньо на сервери організації або в межах локальної мережі.

За методологією системи управління поділяються на Agile/Kanban та Класичні/Waterfall:

– у варіанті Agile/Kanban завдання відображаються у вигляді карток і проходять через різні етапи виконання;
– в Класичній/Waterfall версії робочий процес поділяється на послідовні етапи, що виконуються один за одним.

To-Do Lists – це прості системи управління задачами які надають можливість користувачу створювати список справ, які необхідно виконати, позначати завдання як виконані та, іноді, надавати різні пріоритети або кінцеві терміни виконання.

On-premise – це модель розгортання програмного забезпечення, при якій система встановлюється локально і працює на серверах або комп’ютерах самої компанії. Її ключовою перевагою є повний контроль та конфіденційність даних.

Agile – це методологія яка базується на гнучкому плануванні, поступовій розробці та постійному зворотному зв’язку між командою та замовником. Основною ідеєю Agile є швидке реагування на зміни вимог та поступове покращення продукту. Прикладом роботи за Agile є переміщення від Спринта до

Спринта, Спринт в Agile це невеликий цикл від 1 до 4 тижнів, за період, яких команда має розробити якусь частину програмного забезпечення, яка буде готова до застосування.

Kanban – метод управління робочими процесами який передбачає візуалізацію завдань на спеціальній дошці [4]. Ця дошка має чотири різних етапа виконання завдань, наприклад:

– до виконання (To Do);
– у процесі виконання (In Progress);
– призупинено (On hold);
– виконано (Done).

Переміщення між якими дозволяє краще контролювати процес виконання роботи та швидко розуміти поточний стан задач.

1.2 Облік та аналіз використання робочого часу

Ефективне використання робочого часу є одним із ключових факторів продуктивності як окремого співробітника, так і всієї організації. Облік робочого часу дозволяє отримати інформацію про те, скільки часу витрачається на виконання конкретних завдань, проєктів або робочих процесів. На основі цих даних можна проводити аналіз ефективності роботи, визначати перевантаження або зупинки (простої), а також оптимізувати розподіл завдань між співробітниками.

У сучасних інформаційних системах управління завданнями облік робочого часу часто реалізується за допомогою спеціальних інструментів тайм-трекінгу. Вони дозволяють фіксувати початок і завершення роботи над завданням, зберігати дані про витрачений час та формувати звіти для подальшого аналізу.

1.2.1 Методи обліку робочого часу

Є доцільним розглянути три практичні підходи до обліку робочого часу [5].

Першим методом є ручний облік робочого часу. Працівник самостійно вводить інформацію про витрачений час після виконання конкретного завдання. Наприклад, співробітник може вказати, що на виконання певного завдання він витратив дві години. Цей підхід є найпростішим у реалізації, але він дуже залежить від дисципліни користувача та точності введених даних.

Другим методом є автоматичний облік за допомогою таймера. Співробітник запускає таймер на початку виконання завдання та зупиняє його після завершення роботи. Система автоматично обчислює витрачений час і зберігає ці дані. Даний метод дозволяє отримувати більш точні результати, оскільки зменшує ризик помилок користувача під час введення.

Третім методом є моніторинг активності користувача. У цьому методі система також як і в другому методі – автоматично відстежує активність на комп’ютері, наприклад використання програм, відкритих файлів або часу роботи в різних застосунках, тощо.

Це дозволяє ще більше зменшити участь користувача у відстежуванні витраченого робочого часу, але й також збільшує складність такої системи.

1.2.2 Різниця між методами обліку робочого часу

Ручне введення часу є найпростішим і досить гнучким підходом, тому що запис можна додати у будь-який момент. На практиці його слабке місце очевидне: точність сильно залежить від пам’яті працівника та його дисципліни. Якщо запис робиться наприкінці дня, похибка зазвичай зростає.

Таймер дає точніші результати, адже система автоматично рахує інтервал між стартом і зупинкою роботи. Це зменшує кількість помилок у порівнянні з

ручним методом. Водночас користувачу потрібно постійно контролювати процес: вчасно запускати таймер, ставити його на паузу та завершувати.

Моніторинг активності який відстежує активність в окремих застосунках майже повністю автоматизує облік часу. Система сама фіксує, які програми використовуються і скільки часу триває робота. Такий підхід зручний з погляду збору даних, але має і недоліки: нижчий рівень приватності для працівника що може також негативно впливати на психологічний стан співробітника підчас виконання роботи, та більший обсяг даних, які потім потрібно обробляти й аналізувати.

1.2.3 Ключові метрики оцінки використання робочого часу

Ефективне управління робочим процесом неможливе без об’єктивного вимірювання результатів. Тому для аналізу ефективності використання робочого часу в сучасних інформаційниих системах застосовуються різні кількісні та якісні показники – метрики. До основних метрик належать:

- загальна кількість відпрацьованих годин це базовий кількісний показник який відображає сумарний час витрачений співробітником на виконання робочих завдань за певний період;
- рівень продуктивності це комплексна метрика що відображає узагальнений показник ефективності співробітника за певною формулою, на відміну від простого підрахунку кількості виконаних завдань;
- точність оцінки часу є метрикою що відображає здатність окремого співробітника або команди адекватно планувати власні ресурси, тобто порівняння запланованого часу із фактично витраченим;
- прострочені завдання це показник, що фіксує кількість або частку завдань, які не були завершені до дедлайну.

1.2.4 Роль аналізу робочого часу в управлінських рішеннях

Аналіз робочого часу у сучасних компаніях вже перестав бути інструментом жорсткого мікроменеджменту та перетворився на джерело об'єктивних даних, за допомогою яких з'являється можливість об'єктивно приймати стратегічні й тактичні управлінські рішення:

а) оптимізація розподілу ресурсів:
   1) аналіз робочого часу дозволяє виявляти хронічну, понаднормову роботу окремих працівників або також їхній простій;
   2) на основі отриманих даних можна більш об'єктивно перерозподіляти завдання між співробітниками для покращення загальної ефективності, або залучати нових фахівців;

б) оцінка рентабельності проєктів:
   1) зіставлення відпрацьованих годин із бюджетом проєкту дозволяє виявити приховані витрати часу;
   2) це створює підґрунтя для більш ефективного коригування бюджетом проєкту, зміною ціноутворення для клієнта або закриттям нерентабельних напрямків;

в) удосконалення процесів планування:
   1) аналіз систематичного відхилення фактичного часу надасть можливість менеджеру виправити помилки в плануванні;
   2) у подальшому це сприяє зміні підходів до оцінки строків виконання робочого завдання ще на етапі планування;

г) прийняття кадрових рішень:
   1) використання відповідних метрик зможуть захистити менеджмент від упередженості до окремих співробітників, бо дані здатні чітко показувати результативність окремих фахівців;
   2) це надає можливість об'єктивно підвищувати співробітників на посаді, направляти на навчання або звільняти;

д) запобігання професійному вигоранню:

1) аналітика дозволяє керівникам простіше виявляти співробітників, які тривалий час працюють понаднормово, ще до того, як їхня ефективність різко впаде;
2) надання співробітнику відпустки або зменшення обсягу робочих завдань.

1.3 Архітектурні патерни для десктопних інформаційних систем

Під час розробки сучасних інформаційних систем важливе значення має правильний вибір архітектури програмного забезпечення. Від вибраної архітектури значною мірою залежить масштабованість системи, зручність її підтримки та можливість подальшого розвитку [6].

У випадку десктопних інформаційних систем особливо важливо правильно організувати взаємодію між інтерфейсом користувача, бізнес-логікою та механізмами збереження даних. Для цього використовуються різні архітектурні підходи та патерни проєктування, які дозволяють розділити відповідальність між компонентами системи та зменшити залежність між окремими модулями.

1.3.1 Клієнт-серверна архітектура

Клієнт-серверна архітектура є одним із найпоширеніших способів побудови інформаційних систем. Суть такого підходу полягає в тому, що система поділяється на дві основні частини – клієнт і сервер, які взаємодіють між собою через мережу (рис. 1.1).

Клієнтська частина зазвичай відповідає за роботу з користувачем. Саме тут відображається інтерфейс програми, обробляються дії користувача та формуються запити до сервера. Сервер, у свою чергу, приймає ці запити, виконує необхідні обчислення або перевірки та працює з базою даних [3, 6].

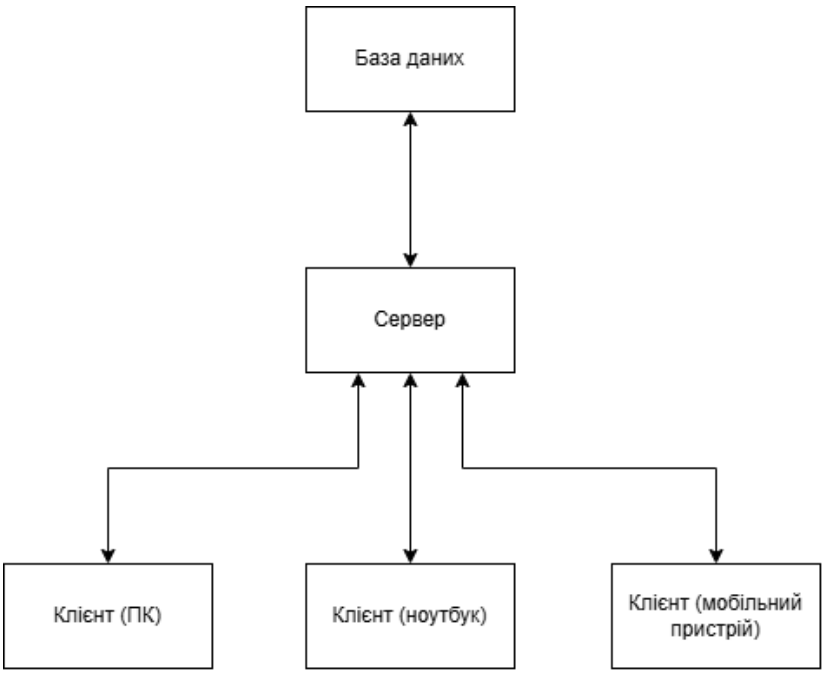


Рисунок 1.1 – Архітектура клієнт-серверної системи

Однією з основних переваг такого підходу є централізоване зберігання даних. Усі дані розміщуються на сервері, що робить їх простішими для адміністрування, резервного копіювання та захисту. Крім того, це дозволяє краще контролювати доступ до інформації.

Також не менш важливою особливістю є можливість одночасної роботи багатьох користувачів із системою. Кілька клієнтів можуть надсилати запити до одного сервера, який обробляє їх і повертає результати. Завдяки цьому система може використовуватися в організаціях або сервісах, де працює велика кількість людей.

Клієнт-серверна архітектура також сприяє підвищенню безпеки системи. Оскільки доступ до бази даних відбувається через сервер, саме на ньому можна реалізувати перевірку користувачів, обмеження прав доступу та інші механізми захисту.

Для порівняння, автономні програми працюють повністю на одному комп’ютері. У такому випадку і база даних, і логіка програми знаходяться на пристрої користувача. Такий підхід простіший у реалізації, однак він має певні

обмеження, наприклад складність організації спільної роботи кількох користувачів або синхронізації даних між різними пристроями.

1.3.2 Переваги клієнт-серверної архітектури

На відміну від автономних систем, де і інтерфейс, і база даних, і вся логіка знаходяться локально на комп'ютері користувача, клієнт-серверна архітектура вирішує низку критичних проблем для корпоративного сектору розробки.

Усі корпоративні дані (завдання, записи робочого часу, профілі співробітників) зберігаються в єдиному місці на сервері. Це унеможливлює розсинхронізацію даних між різними працівниками та значно спрощує процедури резервного копіювання і відновлення інформації у разі збоїв.

Серверні реляційні СКБД спеціально спроєктовані для ефективної обробки сотень паралельних транзакцій. Це дозволяє командам працювати над одними проєктами одночасно, не блокуючи роботу один одного.

1.3.3 Інтерфейс взаємодії між клієнтом та сервером

У більшості сучасних інформаційних систем для взаємодії між клієнтською та серверною частинами використовується REST API. Фактично він виступає інтерфейсом, через який компоненти системи обмінюються даними за допомогою HTTP-запитів.

REST API представляє собою набір правил та методів, які дозволяють різним частинам інформаційної системи обмінюватися даними за допомогою стандартних HTTP-запитів [11]. Основними операціями REST API є GET, POST, PUT та DELETE:

- GET використовується для отримання даних із сервера;
- POST застосовується для створення нового ресурсу;
- PUT / PATCH використовуються для оновлення існуючих даних;
- DELETE призначений для видалення ресурсу.

Важливою особливістю REST є розділення клієнтської та серверної частин системи. Клієнт не взаємодіє з базою даних напряму — усі операції виконуються через серверні API-методи. Це суттєво підвищує рівень безпеки системи та дозволяє контролювати доступ до інформації.

Використання REST API забезпечує більшу гнучкість під час розробки програмного забезпечення. Зокрема, клієнтська та серверна частини системи можуть розвиватися незалежно одна від одної, що значно спрощує модернізацію та масштабування функціоналу системи.

REST API є відносно простим у реалізації, добре документованим та єдиним компромісом що забезпечує необхідний рівень безпеки, і також широко підтримується різними програмними платформами.

Саме тому цей підхід часто використовується під час розробки інформаційних систем, зокрема настільних застосунків.

1.3.4 Патерн організації інтерфейсу користувача

MVVM (Model-View-ViewModel) – це один з найпопулярніших зараз патернів для реалізації інтерфейсу у настільних застосунків [7]. Цей патерн особливо часто використовується у WPF-застосунках, де механізм Data Binding – прив’язки даних, є базовою частиною фреймворку [12].

Основна його мета це відокремити інтерфейс застосунку від логіки застосунку шляхом прив’язки. Ця архітектура складається з трьох компонентів:

- Model відповідає за роботу з даними та бізнес-логіку програми. Дані самого застосунку, також іноді й бізнес логіка;
- View представляє користувацький інтерфейс. Тобто це весь інтерфейс, кнопки, поля, таблиці, тощо;
- ViewModel виступає проміжним шаром між Model та View. Це логіка інтерфейсу, команди та прив’язка даних.

Ключова особливість даного механізму прив’язки це можливість автоматичного відображення змін у інтерфейсі без необхідності його оновлювати.

Це працює таким чином – ViewModel отримує дані з моделі та оброблює їх для відображення у користувацькому інтерфейсі, й водночас може оброблювати дії користувача та передавати будь які відповідні користувацькі запити до моделі.

Його застосування дозволяє писати код більш структурованим а також сильно спрощує тестування логіки та підтримку застосунку.

1.4 Порівняння існуючих рішень

На сучасному ринку існують різноманітні програмні засоби, які надають можливість обліку та аналізу робочих завдань з використаним часом. Але основними та найвідомішими серед користувачів можна вважати наступні.

1.4.1 Огляд Jira

Jira – це система керування робочими процесами, яка дає змогу відстежувати роботу в різних сценаріях, розроблена компанією Atlassian Corporation (рис. 1.2). Вона допомагає розбивати роботу на керовані частини, призначати її потрібним виконавцям і проводити через налаштовуваний робочий процес до завершення. Також можна додавати статуси, пріоритети, коментарі та вкладення.

Дана система є найпотужнішою з усіх в порівнянні з рештою запропонованих рішень на ринку, але це є водночас недоліком й перевагою. З однієї сторони Jira надає можливість дуже глибоко налаштувати управління процесами у команді – особливо для розробників програмного забезпечення, з іншої сторони таке глибоке налаштування може стати викликом для

співробітників. По-перше в самому налаштуванні, й по-друге у використанні такої системи, наприклад, у сфері де це не є доречним.

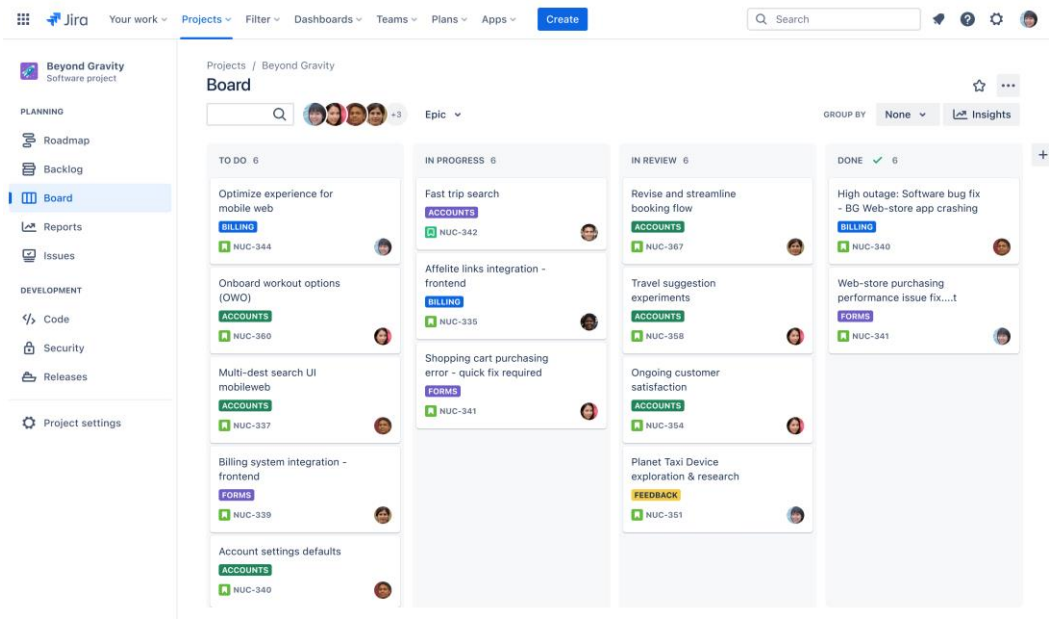


Рисунок 1.2 – Приклад застосунку Jira

1.4.2 Огляд Asana

Asana – це платформа для керування роботою, створена для проєктів, робочих процесів, цілей і команд. Вона дає змогу організовувати роботу команди, підтримувати актуальність інформації про завдання та працювати з різними поданнями проєкту, розроблена компанією Asana, Inc (рис. 1.3).

Наступна система вже є збалансованим рішенням серед запропонованих, З одного боку вона поєднує зручність використання та достатній рівень функціональності без перенавантаження для організації роботи команди й управління завданнями. З іншого боку вартість використання цієї системи може бути відносно високою, особливо для невеликих команд або компаній. Причиною є те, що частина розширених можливостей доступна лише у платних тарифах які коштують в двічі або навіть в тричі більше ніж у конкурентів.

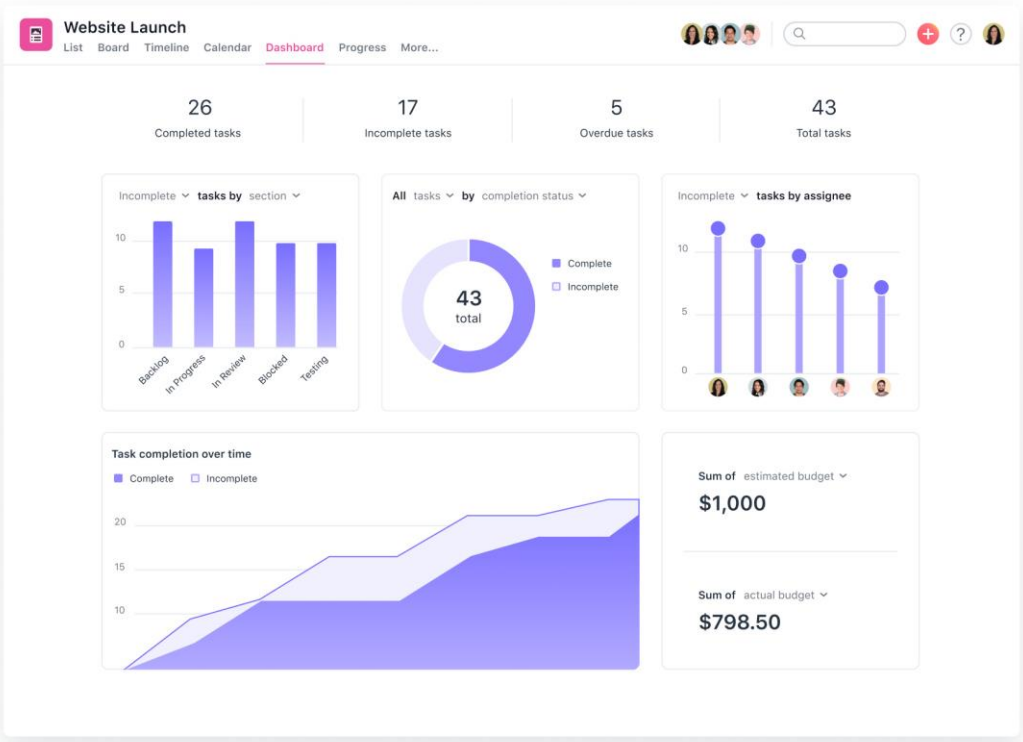


Рисунок 1.3 – Приклад застосунку Asana

1.4.3 Огляд Trello

Trello – це засіб для організації завдань і роботи, який дає змогу бачити проєкти під різними кутами, розроблений як і Jira компанією Atlassian Corporation (рис. 1.4). Він підтримує такі подання, як дошка, часовий план, таблиця, календар, панель показників, карта та подання робочого простору. Крім того, сервіс має вбудовану автоматизацію без програмування.

Дана система є найбільш спрощеним рішенням серед запропонованих. Головним пунктом Trello є – надання максимально простого та зрозумілого інтерфейсу, що дозволяє швидко організувати робочий процес без значних витрат часу на навчання. Водночас, така спрощеність обмежує можливості системи при роботі зі складними проєктами. Передусім, це проявляється у відсутності розвинених інструментів планування, крім того – у недостатній функціональності для управління командами та складними процесами.

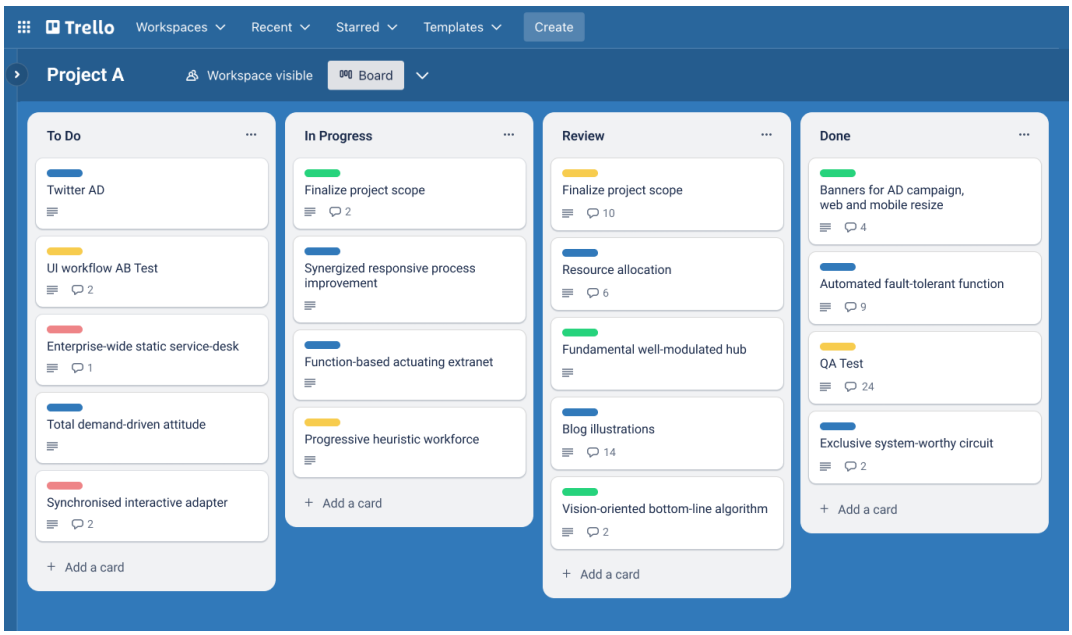


Рисунок 1.4 – Приклад застосунку Trello

1.5 Постановка задачі на розробку інформаційної системи

На основі проведеного аналізу предметної області, існуючих методологій управління, архітектурних патернів та огляду існуючих популярних програмних рішень, можна констатувати потребу у розробці спеціалізованої інформаційної системи для управління внутрішніми робочими завданнями та аналізу використання робочого часу співробітниками, що враховувала б специфіку малих та середніх підприємств.

Обґрунтування функціональних напрямків:

– розроблювана система повинна інтегрувати механізми Kanban-дошок для наочного керування станами завдань, забезпечуючи прозорість роботи команди [4];
– пріоритетним рішенням для системи обрано облік за допомогою таймера, що забезпечить оптимальний баланс між високою точністю даних та простотою використання для кінцевого співробітника;
– для закриття потреби в якісному менеджменті система має автоматично розраховувати ключові метрики, зокрема рівень продуктивності та

точність оцінки часу, що дозволить керівництву приймати обґрунтовані управлінські рішення;

- для забезпечення безпеки та централізації даних у корпоративній мережі за модель розгортання обрано On-premise що дозволить захистити чутливі та конфіденційні дані;
- взаємодія між компонентами повинна бути реалізована через REST API, що гарантує ізоляцію бази даних від клієнтської сторони;
- клієнтська частина має використовувати патерн MVVM [7], що дозволить створити продуктивний настільний інтерфейс із чітким розділенням бізнес-логіки та графічного представлення;

Огляд ринку показав відсутність збалансованого рішення для малих фірм (40–50 осіб), яке б не було надмірно складним, залежало від дорогої підписки або мало обмежену функціональність.

Отже, метою роботи є розробка інформаційної системи, яка поєднає простоту інтерфейсу з інструментами нативного тайм-трекінгу та аналітики, забезпечуючи при цьому економічну ефективність (за рахунок відмови від ліцензійних платежів) та конфіденційність при роботі в локальній мережі (LAN).

2 АНАЛІЗ ВИМОГ ТА СПЕЦИФІКАЦІЯ ПРОГРАМНОЇ СИСТЕМИ

2.1 Бізнес-процеси та користувачі підприємства

Перш за все на даному етапі розробки інформаційної системи необхідно ідентифікувати ключових користувачів системи то сформувати логіку їхньої взаємодії з програмою. На основі вже проведеного аналізу можна сказати що головним об’єктом автоматизації виступає процес управління робочим завданням. Тобто від його створення, до фіксації витраченого часу самим виконавцем – співробітником.

Для безпеки та розмежування відповідальності, в системі було виділено три ролі користувачів:

- адміністратор, це користувач з найбільшим рівнем доступу до Бази даних. Він відповідає за управління інфрастуктурою, блокування акаунтів, відновлення доступів та актуалізацію даних о співробітниках;
- менеджер, це особа яка відповідальна за управління процесами в команді або на підприємстві. Його обов’язками є створення нових проектів. Розбиття великих цілей на окремі завдання, призначення цих завдань конкретним виконавцям, а також планування часу на їх виконання;
- співробітник є звичайним користувачем інформаційної системи. Роль обмежена взаємодією лише з призначеними йому завданням та проєктами, але для більшої гнучкості співробітнику надана можливість для самого себе створити задачу. Основний сценарій співробітника полягає в змінені статусу завдання та здійснення моніторингу витраченого робочого часу за допомогою вбудованого таймеру.

2.2 Сценарій виконання бізнес-процесу

Маючи ідентифікованих користувачів можна сформувати алгоритм робочого процесу наступним чином:

- авторизований менеджер створює сутність «Завдання» в межах існуючого або новоствореного «Проєкту» та призначає відповідального за цю задачу співробітника;
- призначене «Завдання» миттєво з’являється у всіх співробітників для перегляду, а також у призначеного співробітника у робочому просторі для ініціації таймеру;
- розпочинаючи роботу, таймер для призначеної задачі активується співробітником на клієнтському застосунку починаючи локальний відлік часу, паралельно оновлюючи його статус на сервері;
- після завершення роботи, таймер зупиняється співробітником, в цей же момент клієнт формує запит до серверного API з даними про час початку та завершення робочої сесії;
- серверна частина виконує перевірку отриманих даних, прив’язує згенерований запис часу до відповідного завдання та зберігає інформацію в базі даних;
- за збереженими даними Менеджер або Адміністратор мають можливість у будь-який момент згенерувати аналітичний звіт, порівнюючи плановий час, закладений на виконання завдання, із фактичним часом, агрегованим із записів обліку робочого часу Співробітника.

Для формалізації та візуального представлення описаних функціональних вимог та сценаріїв використовуючи мову моделювання UML було створено діаграму прецедентів (Use Case Diagram) наведену на рисунку 2.1.

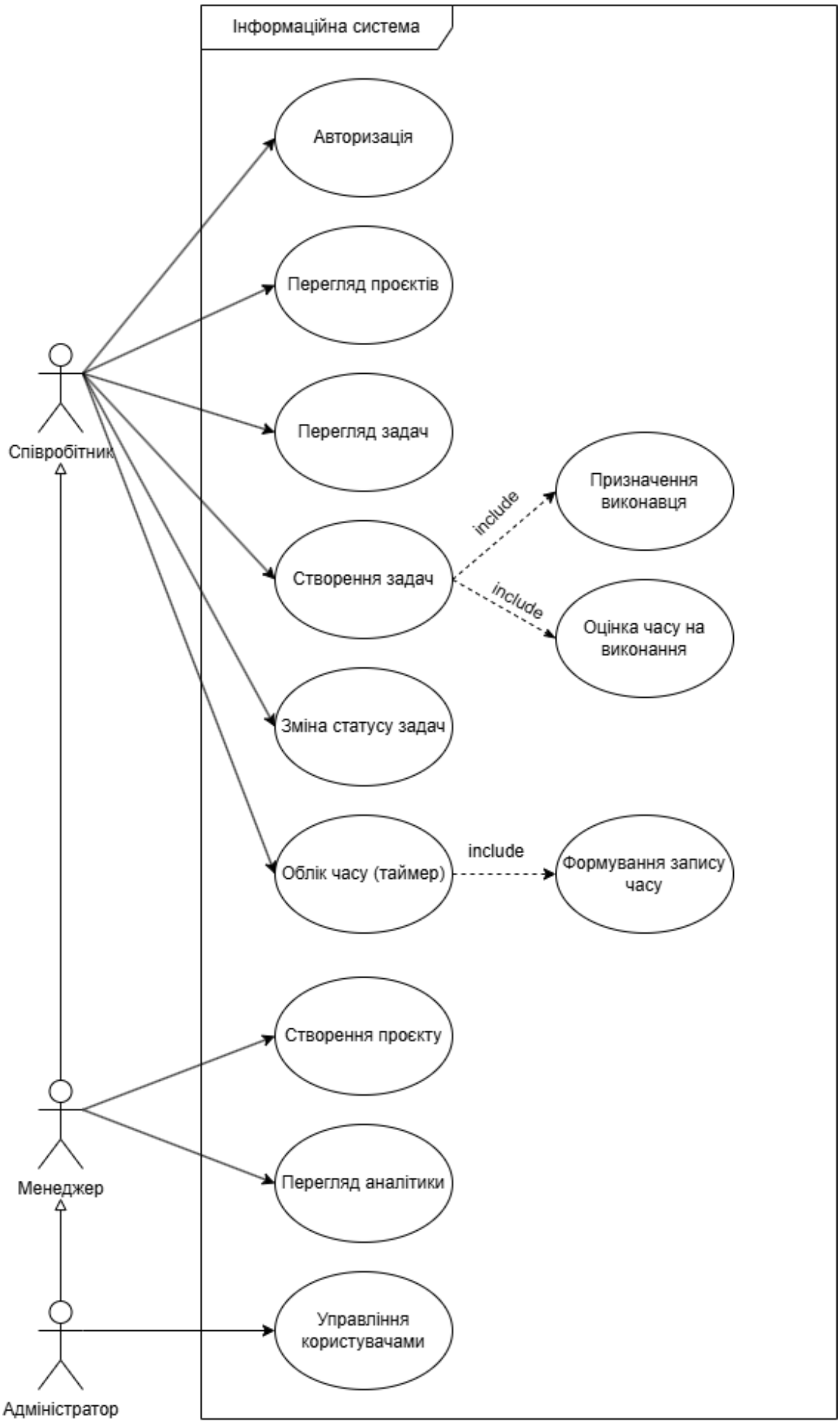


Рисунок 2.1 – Діаграма прецедентів

Таким чином створена діаграма прецедентів відображає ієрархічну побудову ролей в інформаційній системі, де Менеджер наслідує всі можливості Співробітника, а Адміністратор у свою чергу наслідує вже разом усі можливості Співробітника з Менеджером маючи додатково можливість управління користувачами.

Також потрібно зауважити що у випадку створення завдання Співробітником для самого себе, системою буде заблоковано призначення виконавця й автоматично закріплено за нею самого співробітника.

2.3 Функціональні вимоги

На основі змодельованого бізнес процесу й діаграми прецедентів наступним кроком необхідно сформувати набір функціональних вимог, який буде чітко відображати базовий набір можливості розроблюваної інформаційної системи (табл. 2.1).

Таблиця 2.1 – Список функціональних вимог

№	Компонент	Функціональна вимога	Ролі
1	Автентифікація	Реєстрація та безпечна авторизація користувачів (логін/пароль).	Усі
2	Автентифікація	Отримання списку доступних професій/посад на етапі реєстрації.	Усі
3	Проєкти	Створення, редагування характеристик та архівування проєктів.	Адміністратор, Менеджер
4	Завдання	Створення, редагування та видалення завдань, призначення виконавців, оцінка часу (Співробітник обмежений лише власними завданнями).	Усі
5	Завдання	Переведення завдання між статусами (To Do, In Progress, Done).	Усі

Продовження таблиці 2.1

6	Завдання	Перегляд, багатофакторний пошук та сортування списку завдань.	Усі
7	Облік часу	Фоновий облік витраченого часу за допомогою вбудованого таймера.	Усі
8	Облік часу	Ручне додавання запису часу до конкретного завдання.	Усі
9	Облік часу	Перегляд персонального журналу записів часу за поточний день.	Усі
10	Аналітика	Відображення Дошки на головній з персональними метриками (відпрацьований час, продуктивність).	Усі
11	Аналітика	Генерація аналітичного звіту (план/факт) в розрізі обраного співробітника.	Адміністратор, Менеджер
12	Аналітика	Генерація зведено-аналітичного звіту в розрізі обраного проєкту.	Адміністратор, Менеджер
13	Аналітика	Експорт згенерованих аналітичних звітів у формат Excel (.xlsx).	Адміністратор, Менеджер
14	Користувачі	Управління системними акаунтами: зміна ролі користувача, посади, деактивація доступу.	Адміністратор
15	Налаштування	Перемикання кольорової теми та мови інтерфейсу локального застосунку.	Усі
16	Синхронізація	Миттєве двостороннє оновлення даних між клієнтами у реальному часі.	Усі

Представлений у таблиці набір вимог повністю покриває потреби цільової аудиторії підприємства, забезпечуючи як зручне управління завданнями для ролей Менеджер та Адміністратор, так і точний облік витраченого часу на робочих місцях для ролі Співробітник.

3 ПРОЄКТУВАННЯ СИСТЕМИ

3.1 Високорівнена архітектура

3.1.1 Вибір технологічного стеку

Важливим кроком у проектуванні інформаційної системи також є вибір платформи та технологічного стеку який забезпечить необхідний рівень ефективності й масштабування у майбутньому.

В якості платформи для системи, що створюється, було обрано платформу .NET та мову C# [2] [8].

Такий вибір можна обґрунтувати наступними перевагами:

– по-перше, операційна система Windows є найбільш поширеною у світі серед настільних систем. За різними оцінками, кількість її встановлень перевищує кілька мільярдів, що робить її фактичним стандартом для користувацьких робочих станцій;
– по-друге, платформа .NET та мова C# були розроблені компанією Microsoft як частина програмної екосистеми Windows. Це забезпечує високий рівень інтеграції з операційною системою, стабільність роботи та відсутність потреби у встановленні додаткових сторонніх залежностей, окрім стандартного .NET Runtime;
– по-третє, використання C# і .NET як для клієнтської, так і для серверної частини дозволяє застосувати єдиний технологічний стек.

Окрім цього важливим фактором також є вбудовані в .NET, перевірені часом механізми впровадження залежностей, ORM-системи Entity Framework Core [13] та надійні бібліотеки для криптографії і автентифікації.

Усі перелічені переваги надають можливість значно спростити архітектуру системи, полегшити її підтримку та розвиток самого проєкту.

3.1.2 Архітектура модулів системи

На основі результатів, отриманих під час формування сценаріїв-прецедентів та аналізу бізнес-процесів, було визначено, що для реалізації інформаційної системи доцільно розділити відповідальність між окремими модулями за допомогою багатошарової архітектури (N-Layer Architecture), приклад наведено на рисунку 3.1 [6].

Тому на рівні середовища розробки Visual Studio рішення було розділено на п’ять основних зон відповідальності:

– моделі, це бібліотека базових класів, яка містить опис усіх доменних сутностей системи (Користувач, Завдання, Запис часу), а також DTO (Data Transfer Objects), що використовуються для передачі даних між шарами. Даний модуль є спільним і підключається до всіх інших проєктів;
– дані, містять контекст бази даних та конфігурацію зв’язків між таблицями для Entity Framework Core. Відповідає за комунікацію з СКБД;
– сервіси, це шар який інкапсулює логіку комунікації з сервером. Він містить HTTP-клієнти для відправки запитів та обробки відповідей. Виконує роль проміжного рівня між користувацьким інтерфейсом і мережею, приховуючи деталі реалізації API-викликів;
– серверна частина, це – головний виконуваний серверний додаток на базі ASP.NET Core. Здійснює прийом REST-запитів від клієнтів, обробку логіки, перевірку JWT-токенів та роботу з шаром доступу до даних;
– клієнт – настільний WPF-застосунок, побудований за патерном MVVM. Відповідає виключно за візуалізацію даних та обробку дій користувача.

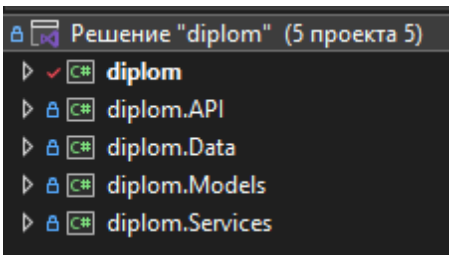


Рисунок 3.1 – Приклад N-Layer в Visual Studio

Графічне представлення модульних залежностей між описаними модулями та технологіями, що використовуються для зв’язку між ними, наведено на діаграмі (рис. 3.2).

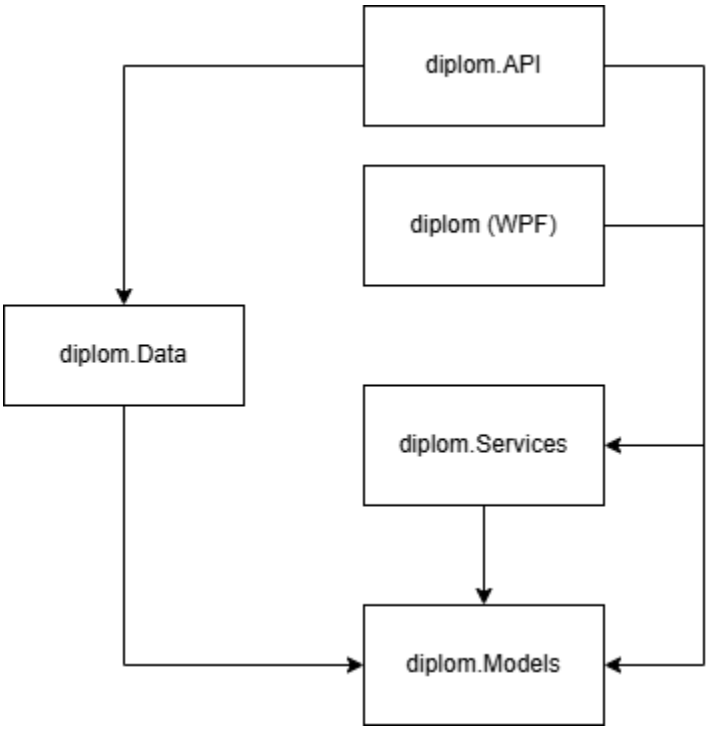


Рисунок 3.2 – Діаграма залежностей модулів

3.2 Проєктування бази даних

3.2.1 Вибір системи керування базою даних

Перед початком проєктування самої бази даних необхідно вирішити яка система керування базами даних з можливих варіантів на ринку, може у повній мірі задовільнити наші потреби.

Під час проєктування було створено порівняльну таблицю де розглядались чотири основні кандидати для використання в інформаційній системі (табл. 3.1).

Таблиця 3.1 – Порівняння доступних СКБД

Характеристика	PostgreSQL	MS SQL Server	MySQL	SQLite
Ліцензія та вартість	Повністю безкоштовна	Платна комерційна (безкоштовна версія до 10 ГБ)	Безкоштовна (Oracle)	Повністю безкоштовна
Масштабування та навантаження	Високе	Високе	Високе	Низьке
Кросплатформність	Linux, Windows, macOS, Docker	Windows, Linux (частково)	Linux, Windows	Усі платформи (локальний файл)
Підтримка в EF Core	Наявний провайдер Npgsql	Офіційний провайдер Microsoft	Наявні провайдери	Наявний провайдер для локальних БД
Цільове призначення	Складні транзакційні системи, аналітика	Корпоративні-рішення, корпоративні системи	Веб-додатки, середні навантаженя	Локальні застосунки, прототипи

Оцінювання характеристик у таблиці 3.1 виконано на основі аналізу офіційної документації відповідних СКБД, а також рівня їх підтримки у фреймворку Entity Framework Core [13] [10].

У межах цієї роботи під підтримкою в EF Core розуміється наявність провайдера, який забезпечує взаємодію застосунку з відповідною СКБД через Entity Framework Core. На основі порівняльного аналізу СКБД PostgreSQL було обрано з наступних причин [10]:

– повна безкоштовність без обмеження в об’ємі пам’яті в порівнянні з Microsoft SQL;
– підтримка багатокористувацької роботи, транзакцій, індексування та ефективного виконання SQL-запитів;
– наявність провайдеру Npgsql для ефективного застосування з EF Core.

3.2.2 Створення структури БД

Проєктування та розробка реляційної структури БД виконувалися за архітектурним підходом Code-First з використанням ORM-фреймворку Entity Framework Core [13].

Цей підхід дозволяє перенести відповідальність за управління базою даних на код додатку. Структурні таблиці реляційної БД описуються безпосередньо як C#-класи, а структурні обмеження для полів, наприклад – типи даних, довжини рядків, обов’язковість задаються за допомогою атрибутів анотації даних, таких як [Key], [Required], [MaxLength]. Фізична схема бази генерується та версіонується міграціями EF Core автоматично за командою розробника. Нижче наведено фрагмент програмного коду моделі TaskItem, яка описує сутність задач в системі:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
      namespace diplom.Models
      {
            public class TaskItem
            {
```

```
        [Key]
        public int Id { get; set; }
        [Required]
        [MaxLength(200)]
        public string Title { get; set; } = string.Empty;
        public string Description { get; set; } =
    string.Empty;
        public AppTaskStatus Status { get; set; } =
    AppTaskStatus.ToDo;
        public TaskPriority Priority { get; set; } =
    TaskPriority.Medium;
        public DateTime CreatedAt { get; set; } =
    DateTime.UtcNow;
        public DateTime? Deadline { get; set; }
        public double EstimatedHours { get; set; } = 0;
        [Required]
        public int ProjectId { get; set; }
        [ForeignKey("ProjectId")]
        public Project? Project { get; set; }
        public int? AssigneeId { get; set; }
        [ForeignKey("AssigneeId")]
        public User? Assignee { get; set; }
    }
}
```

У даному випадку використання атрибута Key вказує Entity Framework на те, що властивість Id має стати первинним ключем таблиці з автоматичним автоінкрементом. Атрибут ForeignKey встановлює навігаційну властивість для зовнішнього ключа.

Після визначення моделі сутності з використанням атрибутів Data Annotations, наступним кроком є необхідність сформувати фізичну структуру бази даних. При використанні підходу Code-First цей процес виконується автоматично за допомогою механізму міграцій у Entity Framework Core (рис. 3.3).

```
PS C:\Users\anko\source\repos\diplom> dotnet ef migrations script --project diplom.Data --startup-project diplom.API
Build started...
Build succeeded.
CREATE TABLE IF NOT EXISTS "__EFMigrationsHistory" (
    "MigrationId" character varying(150) NOT NULL,
    "ProductVersion" character varying(32) NOT NULL,
    CONSTRAINT "PK___EFMigrationsHistory" PRIMARY KEY ("MigrationId")
);
```

Рисунок 3.3 – Виконання міграції EF

На основі наведеної раніше моделі TaskItem Entity Framework Core формує SQL-скрипт для створення відповідної таблиці в базі даних. Фрагмент такого SQL-скрипту наведено нижче:

```
CREATE TABLE "Tasks" (
"Id" integer GENERATED BY DEFAULT AS IDENTITY,
"Title" character varying(200) NOT NULL,
"Description" text NOT NULL,
"Status" integer NOT NULL,
"Priority" integer NOT NULL,
"CreatedAt" timestamp without time zone NOT NULL,
"Deadline" timestamp without time zone,
"EstimatedHours" double precision NOT NULL,
"ProjectId" integer NOT NULL,
"AssigneeId" integer,
CONSTRAINT "PK_Tasks" PRIMARY KEY ("Id"),
CONSTRAINT "FK_Tasks_Projects_ProjectId" FOREIGN KEY ("ProjectId")
REFERENCES "Projects" ("Id") ON DELETE CASCADE,
CONSTRAINT "FK_Tasks_Users_AssigneeId" FOREIGN KEY ("AssigneeId")
REFERENCES "Users" ("Id")
);
CREATE INDEX "IX_Tasks_AssigneeId" ON "Tasks" ("AssigneeId");
CREATE INDEX "IX_Tasks_ProjectId" ON "Tasks" ("ProjectId");
```

Нижче перелічено усі сутності проєктованої інформаційної системи:

- користувачі (Users) – сутність, що зберігає облікові записи працівників та їхні профільні дані. Має зв’язок один-до-багатьох із сутностями завдань та записів часу, оскільки один користувач може виконувати багато завдань і створювати багато записів часу;
- проєкти (Projects) – сутність для групування завдань за окремими напрямами діяльності. Має зв’язок один-до-багатьох із завданнями;
- завдання (Tasks) – сутність, що містить інформацію про робочі завдання (назва, опис, статус, виконавець). Має зовнішні ключі до таблиць проєктів і користувачів, а також зв’язок один-до-багатьох із записами часу;
- записи часу (TimeLogs) – сутність для фіксації витраченого часу на виконання завдань. Містить посилання на завдання та користувача, реалізуючи зв’язки багато-до-одного.

Логічну структуру спроєктованої бази даних та зв’язки між сутностями системи деталізовано на ER-діаграмі, наведеній на рисунку 3.4.

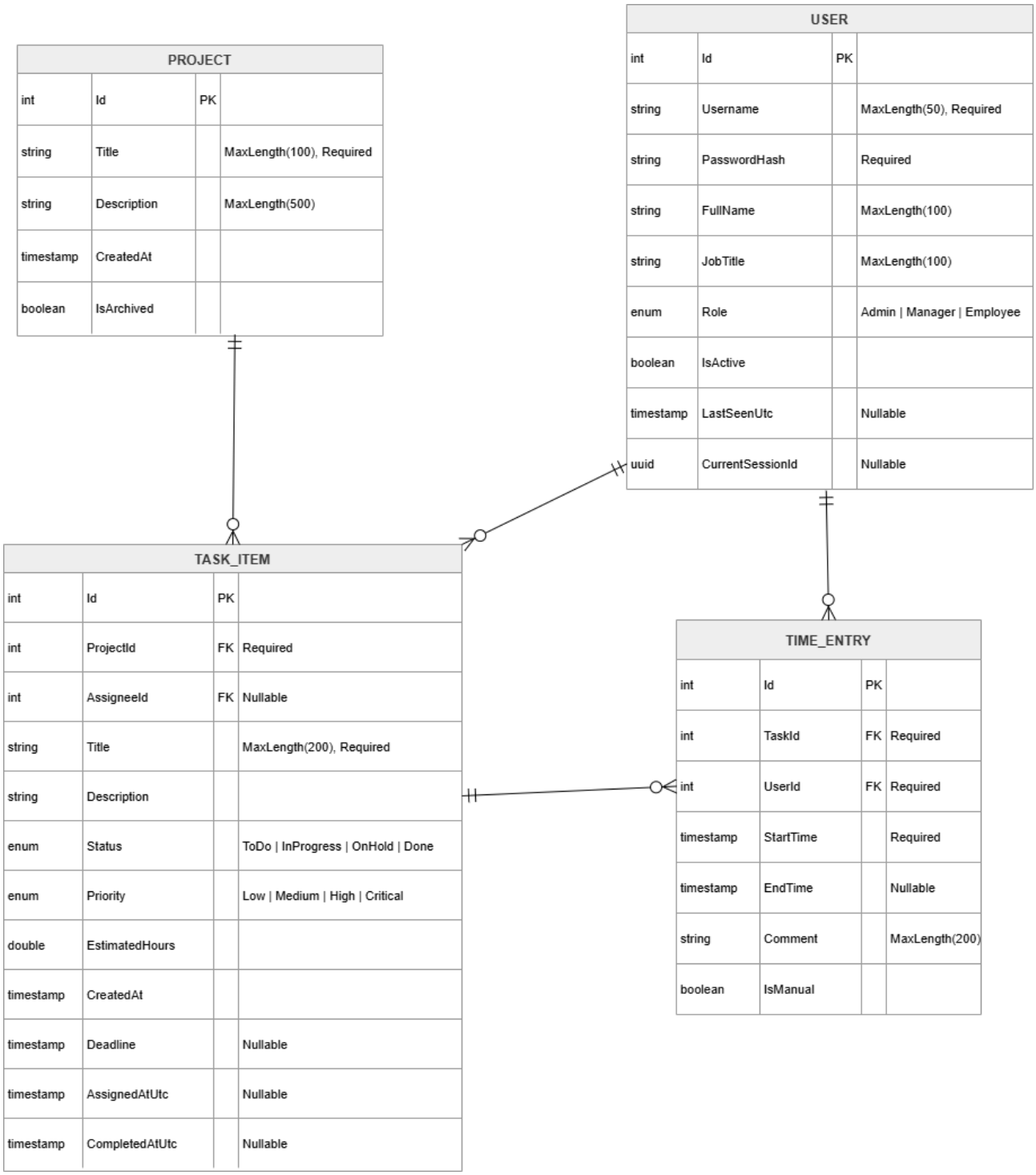


Рисунок 3.4 – ER-діаграма БД

Спроєктована база даних містить у собі загалом 4 таблиці:

– users містить інформацію про користувачів системи та використовується для зберігання облікових записів і профільних даних;
– projects призначена для зберігання інформації про проєкти, в межах яких групуються завдання;
– tasks містить дані про робочі завдання, включаючи їхні основні характеристики та зв’язки з проєктами і користувачами;
– timelogs використовується для фіксації витраченого часу на виконання завдань і містить посилання на відповідні завдання та користувачів.

3.3 Проєктування API

Взаємодія між клієнтським додатком (WPF) та серверною частиною побудована на базі архітектурного стилю REST поверх протоколу HTTP [11]. Такий вибір обґрунтований:

– по-перше універсальністю та стандартизацією, REST використовує стандартні HTTP-методи (GET, POST, PUT, DELETE), які мають чітку семантику. Що також залишає можливість масштабування у майбутньому;
– по-друге легкою вагою, Обмін даними здійснюється у текстовому форматі JSON, що дозволяє у свою чергу C# нативно та ефективно десеріалізувати об’єкти завдяки вбудованим можливостям.

Також не менш важливим компонентом захисту сучасних API є JSON Web Token який забезпечує безпеку та ідентифікацію клієнтів. На відміну від звичних сесій або Cookies, кожен HTTP запит містить маркер доступу, котрий криптографічно підписаний сервером. Це виключає необхідність постійно звертатися до БД для перевірки прав клієнта.

Відповідно до архітектури REST, програмний інтерфейс системи логічно поділено на ресурсо-орієнтовані контролери:

– AuthController – відповідає за автентифікацію, реєстрацію та видачу токенів;
– ProjectsController – керує життєвим циклом проєктів та формує аналітику їх виконання;
– TasksController – головний контролер для управління завданнями, зміни їх статусів та розподілу виконавців;
– TimeEntriesController – приймає сигнали системного таймера та обробляє ручні записи відпрацьованих годин;
– UsersController – адміністративний ресурс для налаштування системи розмежування доступу.

Завдяки результату аналізу функціональних вимог та діаграми прецедентів можна сформувати набір ключових кінцевих точок для управлінням завданнями та обліку часу у таблиці 3.2 а також для управлінням ресурсами та аналітикою у таблиці 3.3.

Таблиця 3.2 – API: Управління завданнями та облік часу

HTTP Метод	Маршрут (URL)	Призначення
POST	/api/auth/login	Вхід у систему та отримання JWT-токена
GET	/api/auth/me	Отримання даних поточного користувача та його ролі
GET	/api/tasks/my	Основний список задач поточного співробітника
POST	/api/tasks	Створення нової задачі
PUT	/api/tasks/{id}	Оновлення статусу, виконавця, дедлайну та інших полів задачі
POST	/api/timeentries	Запуск/додавання обліку часу по задачі
POST	/api/timeentries/stop-active	Зупинка активного таймера
GET	/api/timeentries/today	Перегляд відпрацьованого часу за сьогодні
GET	/api/projects/active	Перелік активних проєктів для роботи
GET	api/dashboard/stats	Ключові показники (KPI) на дашборді

Таблиця 3.3 – API: Управління ресурсами та аналітика

HTTP Метод	Маршрут (URL)	Роль	Призначення
GET	/api/users/assignable	Адміністратор, Менеджер	Отримати список користувачів для призначення виконавців
PUT	/api/users/{id}/role	Адміністратор	Змінити роль користувача
PUT	/api/users/{id}/activate	Адміністратор	Активувати користувача
PUT	/api/users/{id}/deactivate	Адміністратор	Деактивувати користувача
GET	/api/projects/{id}/analytics	Адміністратор, Менеджер	Перегляд аналітики по проєкту
GET	/api/users/{id}/analytics	Адміністратор, Менеджер	Перегляд аналітики по конкретному користувачу
POST	/api/projects	Адміністратор, Менеджер	Створити новий проєкт

3.4 Архітектура клієнта MVVM

Розробка настільного клієнтського застосунку (WPF) як було вирішено раніше базуватиметься на використанні архітектурного паттерна MVVM [7]. Що дозволить повністю відокремити візуальний інтерфейс користувача від бізнес-логіки та логіки презентації. У реалізації використано фреймворк CommunityToolkit.Mvvm, який забезпечує автоматичну генерацію коду для властивостей реактивного оновлення під час компіляції.

Архітектура клієнта структурно поділяється на три взаємозалежні рівні:

– представлення (View) – XAML-файли, що містять виключно опис візуальних елементів. Представлення не містять коду бізнес-логіки. А спілкування з логікою відбувається через механізм двостороннього прив’язування даних (Data Binding) ;

– моделі представлення (ViewModel) – класи-посередники, що інкапсулюють стан сторінки та команди. Навігація між екранами у застосунку побудована таким чином що головний об’єкт MainViewModel керує поточною сторінкою шляхом зміни активної ViewModel у своїй властивості CurrentView;
– сервіси (Services) – проміжні компоненти для абстракції інфраструктурних задач.

Детальну діаграму архітектурних зв’язків та залежностей між описаними шарами клієнтського застосунку наведено у Додатку Б.

3.5 Проєктування підсистеми аналітики та експорту звітності

Важливою вимогою до інформаційної системи авжеж є надання інструментів для моніторингу ефективності роботи працівників та стану проєктів. Для цього було спроєктовано підсистему генерації звітів чия робота складається з декількох етапів:

– збір даних на сервері – сервер отримує запити, робить вибірку з БД та рахує кількість відпрацьованих годин і завершених завдань;
– відображення клієнту – клієнт приймає ці дані та відображує зручні графіки у окремому вікні, для цього буде застосовано бібліотеку LiveCharts2 [14];
– експорт даних – для реалізації можливості збереження звітів, або для використання у інших видах роботи необхідно створити спеціальний сервіс експорту.

Сервіс експорту аналітики повинен генерувати два типи документів:

– звіт працівника з деталізацією відпрацьованого часу по днях;
– звіт проєкту який показує загальний прогрес виконання та список завдань, на які витрачено занадто багато часу.

Сам експорт звітів необхідно буде виконувати в .xlsx форматі, для реалізації цієї задачі навіть без встановленого Microsoft Office було обрано допоміжну бібліотеку ClosedXML [16].

3.6 Проєктування підсистеми взаємодії у реальному часі

У традиційній архітектурі REST API взаємодія завжди ініціюється клієнтським застосунком. Тобто, для того щоб дізнатися про появу нового завдання або зміну його статусу, клієнт мусить періодично надсилати запити на сервер. Такий підхід постійного опитування не лише може створити зайве навантаження на систему при її масштабуванні, а й унеможливлити створення повноцінного інтерактивного середовища.

Саме тому, замість реалізації періодичних запитів, було вирішено необхідність інтегрувати технологію відправки повідомлень у реальному часі на базі бібліотеки SignalR [9], яка працює поверх протоколу WebSockets. Архітектурно це реалізовано через встановлення постійного двостороннього з’єднання між WPF-клієнтами та серверним Хабом.

Використання SignalR у межах розроблюваної системи вирішить три ключові задачі:

– синхронізація між користувачами. Наприклад, якщо Менеджер призначив нову задачу, то сервер автоматично виконає оновлення інформації для усіх користувачів;
– внутрішня узгодженість інтерфейсу. Завдяки SignalR з’являється можливість реалізувати відстеження активного таймеру на інші сторінки, в режимі реального часу, що в рази збільшить відчуття живої системи;
– моніторинг присутності. Менеджери та Адміністратори зможуть мати можливість відстежувати актуальний статус працівників, тобто хто

зараз безпосередньо працює над чимось, хто просто онлайн, а кого немає на місці.

4 РЕАЛІЗАЦІЯ СИСТЕМИ

4.1 Реалізація серверної частини

Відповідно до розробленої в розділі 3 N-Layer архітектури, серверна частина інформаційної системи API відповідає за централізоване зберігання даних, обробку бізнес-логіки управління завданнями та автентифікацію. Розробка виконувалась на базі веб-фреймворку ASP.NET Core 8 [11].

4.1.1 Конфігурація

Точкою входу в серверний застосунок є файл Program.cs. Тут реалізовано налаштування усіх необхідних сервісів системи за допомогою вбудованого механізму впровадження залежностей. Приклад коду основних етапів конфігурації наведено нижче:

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>();

builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme
)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(jwtKey))
        };
    });

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.ReferenceHandler =
ReferenceHandler.IgnoreCycles;
```

```
        options.JsonSerializerOptions.Converters.Add(new
JsonStringEnumConverter());
    });

builder.Services.AddSignalR();

var app = builder.Build();

app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
app.MapHub<TimeHub>("/hubs/time");
```

Метод AddDbContext<AppDbContext>() підключає шар доступу до даних. Оскільки конфігурація з'єднання винесена у окремий клас, це забезпечує чистоту коду головного файлу.

Забезпеченням безпеки тобто JSON Web Token займається блок налаштування AddJwtBearer – визначає правила валідації вхідних токенів. Система перевіряє не лише підпис токена ValidateIssuerSigningKey, а й термін його дії ValidateLifetime, щоб уникнути можливості використання застарілого токену.

За маршрутизацію відповідальний метод MapHub<TimeHub> реєструється кінцева точка для веб-сокет з'єднань. Усі повідомлення про зміну статусів працівників та запуск таймерів будуть проходити саме через маршрут /hubs/time.

4.1.2 Реалізація доступу до БД

Відповідно до підходу Code-First, фізична структура бази даних повністю визначається програмним кодом через клас контексту AppDbContext. Фрагмент реалізації контексту БД наведено нижче:

```
public class AppDbContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Project> Projects { get; set; }
    public DbSet<TaskItem> Tasks { get; set; }
    public DbSet<TimeEntry> TimeLogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
```

```
        modelBuilder.Entity<Project>()
            .HasMany(p => p.Tasks)
            .WithOne(t => t.Project)
            .HasForeignKey(t => t.ProjectId)
            .OnDelete(DeleteBehavior.Cascade);
    }
}
```

Набір властивостей DbSet<T> відповідає за мапінг моделей C# у таблиці PostgreSQL. Кожне звернення до цих колекцій автоматично транслюється бібліотекою Entity Framework Core у відповідні SQL-запити.

У методі OnModelCreating() реалізовано правило «Один-до-багатьох» між проєктами та завданнями. Також увагу було приділено параметру DeleteBehavior.Cascade для того щоб при видаленні проєкту система автоматично очистила також усі пов’язані з ним завдання, що забезпечує цілісність даних на рівні СКБД.

4.1.3 Реалізація REST-контролерів та бізнес-логіки

Програмний інтерфейс системи логічно поділено на контролери, кожен з яких відповідає за свою частину роботи: завдання, облік часу чи проєкти.

Даний контролер забезпечує створення та оновлення завдань. Приклад реалізації методу створення завдань має такий вигляд:

```
[HttpPost]
public async Task<ActionResult<TaskItem>> Create([FromBody] TaskItem task)
{
    var userId = GetCurrentUserId();
    var isAdminOrManager = IsAdminOrManager();

    if (!isAdminOrManager) task.AssigneeId = userId;

    task.CreatedAt = DateTime.UtcNow;
    if (task.AssigneeId.HasValue) task.AssignedAtUtc = DateTime.UtcNow;
    task.CompletedAtUtc = task.Status == AppTaskStatus.Done ?
DateTime.UtcNow : null;

    _context.Tasks.Add(task);
    await _context.SaveChangesAsync();

    await _hub.Clients.All.SendAsync("TaskChanged", task.Id);
    return CreatedAtAction(nameof(GetById), new { id = task.Id }, task);
```

```
}
```

У методі реалізовано автоматичну перевірку ролей. Якщо систему використовує звичайний робітник, він не може призначити завдання іншій особі – система автоматично призначає його самого. Також завдяки SendAsync завдання з'являється у списках усіх користувачів миттєво.

Наступний контролер обробляє дані від системного таймера. Найважливішим є метод зупинки таймера, який фіксує результати роботи співробітника. Реалізація цього методу виглядає таким чином:

```
[HttpPost("stop-active")]
public async Task<ActionResult<TimeEntry>> StopActive([FromBody]
StopActiveRequest request)
{
    var userId = GetCurrentUserId();
    var entry = await _context.TimeLogs
        .Where(e => e.UserId == userId && e.EndTime == null)
        .OrderByDescending(e => e.StartTime)
        .FirstOrDefaultAsync();

    if (entry == null) return NotFound();

    entry.EndTime = DateTime.UtcNow;
    entry.Comment = request?.Comment ?? entry.Comment;
    entry.IsManual = false;

    await _context.SaveChangesAsync();
    await _hub.Clients.All.SendAsync("TimeEntryChanged", entry.TaskId,
userId);

    return Ok(entry);
}
```

Метод самостійно знаходить незавершену робочу сесію користувача і закриває її поточним серверним часом. Це забезпечує точність обліку, оскільки користувач не може підправити час вручну.

Цей контролер керує створенням проєктів та формує загальні звіти по них. Головною функцією є метод GetProjectAnalytics(), який збирає всі показники за певний період. Приклад методу GetProjectAnalytics() має такий вигляд:

```
[HttpGet("{id}/analytics")]
[Authorize(Roles = "Admin,Manager")]
public async Task<ActionResult<ProjectAnalyticsDto>>
GetProjectAnalytics(int id, [FromQuery] int days = 30)
{
```

```
    if (days < 7) days = 7;
    if (days > 90) days = 90;

    var toUtc = DateTime.UtcNow;
    var fromUtc = toUtc.Date.AddDays(-days + 1);

    var tasks = await _context.Tasks
        .Include(t => t.TimeEntries).Include(t => t.Assignee)
        .Where(t => t.ProjectId == id).ToListAsync();

    var logs = await _context.TimeLogs
        .Include(l => l.Task)
        .Where(l => l.Task != null && l.Task.ProjectId == id &&
l.EndTime.HasValue)
        .Where(l => l.EndTime!.Value >= fromUtc && l.EndTime!.Value <=
toUtc)
        .ToListAsync();

    return Ok(new ProjectAnalyticsDto
    {
        ProjectId = id,
        FromUtc = fromUtc,
        ToUtc = toUtc,
        WorkedHours = Math.Round(TimeSpan.FromTicks(logs.Sum(l =>
l.Duration.Ticks)).TotalHours, 2),
        TasksCompleted = tasks.Count(t => t.CompletedAtUtc.HasValue)
    });
}
```

Метод автоматично збирає всі записи часу, пов’язані з конкретним проєктом, та підсумовує відпрацьовані години. Результат повертається у вигляді спеціального об’єкта ProjectAnalyticsDto, який клієнтський додаток використовує для побудови графіків.

4.2 Реалізація клієнтської частини

Клієнтський додаток розроблено на основі технології WPF за архітектурою MVVM (Model-View-ViewModel).

4.2.1 Архітектура MVVM та механізм навігації

Логіка роботи інтерфейсу зосереджена у класах ViewModel. Для забезпечення автоматичного оновлення даних на екрані при їх зміні в коді, було використано базовий клас ObservableObject.

Головним компонентом інтерфейсу є MainViewModel, який відповідає за навігацію між розділами Дашборд тобто Головну, Завдання, Проєкти тощо. Приклад фрагменту класу MainViewModel наведено нижче:

```
public class MainViewModel : ObservableObject
{
    private object _currentView;
    public object CurrentView { get => _currentView; set =>
SetProperty(ref _currentView, value); }

    public ICommand NavigateCommand { get; }

    public MainViewModel()
    {
        CurrentView = new DashboardViewModel();
        NavigateCommand = new RelayCommand<string>(OnNavigate);
    }

    private void OnNavigate(string destination)
    {
        switch (destination)
        {
            case "Dashboard": CurrentView = new DashboardViewModel();
break;
            case "Tasks": CurrentView = new TasksViewModel(); break;
            case "Projects": CurrentView = new ProjectsViewModel();
break;
            case "TimeTracker": CurrentView = new TimeTrackerViewModel();
break;
        }
    }
     }
```

Властивість CurrentView використовує метод SetProperty [12]. Це дозволяє програмі повідомити графічному інтерфейсу про необхідність змінити вміст вікна без використання складного коду оновлення.

Метод OnNavigate змінює об’єкт у властивості CurrentView, а WPF автоматично підставляє відповідний візуальний шаблон View для обраної моделі представлення.

4.2.2 Сервіси клієнта та обробка даних

Для розділення логіки відображення ViewModels та інфраструктурних задач, було впроваджено декілька сервісів. Найважливішими з них є мережевий клієнта та сервіс управління часом.

Клас мережвого клієнту реалізований за шаблоном Singleton і відповідає за всі HTTP-запити до серверної частини. Головною його перевагою є не лише використання універсальних методів та автоматичне керування JWT-токеном, а й ефективність керування ресурсами тому що ми кожен раз перевикористовуємо наявне TCP з'єднання замість створювання нового кожен раз. Приклад універсального мережевого запиту:

```
public async Task<T?> GetAsync<T>(string endpoint)
{
    var response = await _http.GetAsync($"{BaseUrl}{endpoint}");
    await EnsureAuthorizedAsync(response);
    response.EnsureSuccessStatusCode();
    return await response.Content.ReadFromJsonAsync<T>(_jsonOptions);
}
```

Універсальний метод GetAsync<T> дозволяє суттєво скоротити обсяг коду. Замість того, щоб писати окремий метод для отримання завдань, а потім наприклад ще один окремий для проєктів, нам достатньо викликати цей один метод і передати очікуваний тип даних, наприклад GetAsync<TaskItem>, що поверне нам завдання.

Наступний сервіс зберігає глобальний стан активної задачі у програмі. Незалежно від того, між якими вікнами перемикається користувач, таймер буде продовжувати свою роботу у фоновому режимі, реалізація методу зупинки таймера має такий вигляд:

```
public async Task<TimeEntry?> StopActiveAsync(string comment = "Timer
session")
{
    if (!HasActiveSession) return null;

    TimeEntry? created = null;

    try
```

```
    {
        created = await _api.PostAsync<TimeEntry>("/api/timeentries/stop-
active", new
        {
            Comment = comment ?? string.Empty
        });
    }
    catch { }

    if (created == null)
    {
        created = await _dataService.CreateTimeEntryAsync(new TimeEntry
        {
            TaskId = ActiveTaskId!.Value,
            StartTime = ActiveStartTimeLocal!.Value.ToUniversalTime(),
            EndTime = DateTime.UtcNow,
            IsManual = false,
            Comment = comment ?? string.Empty
        });
    }

    ActiveTaskId = null;
    ActiveStartTimeLocal = null;
    return created;
}
```

При натисканні користувачем кнопки Стоп, сервіс перевіряє факт наявності активної сесії за допомогою HasActiveSession. Якщо сесія знайдена, то відправляється спеціальний POST-запит до API сервера до методу StopActive, розглянутого вище. Після успішної відповіді локальні змінні очищуються, і таймер зникає з екрану клієнта.

4.2.3 Реалізація взаємодії у реальному часі

Для забезпечення миттєвої синхронізації даних між різними клієнтами та відстеження статусу працівників "Онлайн", "Працює" було реалізовано клієнтський сервіс RealTimeService [9]. Його головна задача це підтримка фонового з’єднання із серверним Хабом, приклад підключення клієнта до SignalR наведено нижче:

```
public async Task EnsureStartedAsync()
{
    var url = $"{_api.BaseUrl.TrimEnd('/')}/hubs/time";
    _connection = new HubConnectionBuilder()
```

```
        .WithUrl(url,    o    =>    o.AccessTokenProvider    =    ()    =>
Task.FromResult(_api.Token)!)
        .WithAutomaticReconnect()
        .Build();

    _connection.On<int>("TaskChanged",              taskId              =>
TaskChanged?.Invoke(taskId));
    _connection.On<int, int>("TimeEntryChanged",
        (taskId, userId) => TimeEntryChanged?.Invoke(taskId, userId));

    await _connection.StartAsync();
}
```

Використання методу WithAutomaticReconnect() забезпечує у разі втрати інтернету автоматичні спроби відновити зв’язок із сервером без стороннього втручання користувача. Доступ до веб-сокетів захищено JWT-токеном, що допомагає запобігати несанкціонованому підключенню третіх осіб до каналу обміну даними підприємства.

Метод On забезпечує безперервне очікування сигналів від сервера. Тобто якщо менеджер додає нове завдання або колега запускає таймер, сервіс миттєво надсилає команду на оновлення списків в інтерфейсі.

4.3 Реалізація аналітики та експорту

4.3.1 Підрахунок продуктивності

Особливістю розробленої системи є наявність модуля інтелектуальної аналітики, який автоматично розраховує персональний рейтинг продуктивності працівників. Логіка розрахунків винесена в окремий клас ProductivityCalculator.

Для уникнення різких стрибків статистики, наприклад, коли працівник виконав лише одну задачу з однієї, що дасть 100% результату, порівняно з працівником, який виконав 9 з 10 задач, у системі застосовується метод байєсівського згладжування [15].

Окрім цього, алгоритм враховує точність планування часу та штрафує за прострочені дедлайни. Фрагмент ProductivityCalculator наведено нижче:

```
public static ProductivityResult CalculateForWeek(IEnumerable<TaskItem>
tasks, DateTime weekStart, DateTime weekEnd, Settings settings)
{

    var smoothed = (done + settings.PriorDoneTasks) / (total +
settings.PriorTotalTasks);

    var effMult = settings.EfficiencyMultiplierMin +
                  (settings.EfficiencyMultiplierMax -
settings.EfficiencyMultiplierMin) * efficiency;


    var overduePenalty = Math.Min(settings.OverdueMaxPenalty,
overdueCount * settings.OverduePerTaskPenalty);


    var score = smoothed * effMult * (1.0 - overduePenalty) * 100.0;

    return new ProductivityResult { ScorePercent = Math.Round(score, 1)
};
}
```

В наведеному фрагменті слід особливо приділити увагу параметрам PriorTotalTasks та PriorDoneTasks. Вони встановлюються за замовчуванням для усіх співробітників підприємства для більш об’єктивного підрахунку продуктивності, й авжеж можуть калібруватися на основі історичних даних підприємства. За замовчуванням PriorTotalTasks встановлено на 5 як помірна сила згладжування для малої вибірки, й PriorDoneTasks як базова доля виконання робочих завдань у 60%.

Параметр effMult лінійно перетворює показник efficiency у межах від 0 до 1 у коефіцієнт впливу на фінальний рейтинг, котрий за замовчуванням від 0.8 до 1.2. Тобто за низької точності планування підсумковий бал зменшується, а за високої – помірно збільшується.

Параметр overduePenalty обчислюється як добуток кількості прострочених задач на штраф за одну задачу, але з верхньою межею OverdueMaxPenalty. Кожне прострочення знижує рейтинг, але загальний штраф не стає необмеженим і не обнуляє результат повністю.

Фінальна оцінка формується як добуток цих трьох складових. Згладжений відсоток виконаних завдань множиться на коефіцієнт ефективності працівника.

Після цього від результату віднімається штрафний відсоток за кожну прострочену задачу.

4.3.2 Формування аналітичних моделей передачі даних

Для забезпечення безпеки та зменшення обсягу мережевого трафіку, результати роботи аналітичної системи не передаються клієнту у вигляді сирих записів з бази даних. Замість цього було спроєктовано спеціальні моделі передачі даних: UserAnalyticsDto та ProjectAnalyticsDto.

Вони виступають у ролі готових зведених звітів, які сервер будує перед відправкою:

- UserAnalyticsDto передає всю картину роботи конкретного співробітника. Вона містить глобальні статистику та масив щоденної аналітики. Що дозволяє легко побудувати графік продуктивності по днях, де видно, в які дні тижня працівник працює найефективніше;
- ProjectAnalyticsDto орієнтована на менеджмент. Вона агрегує дані всіх співробітників, які залучені до конкретного проєкту. Ключовою відмінністю цієї моделі є те, що вона фокусується на частці виконаних завдань до загальної кількості оцінених годин, що дозволяє менеджеру візуально відстежувати глобальний прогрес розробки та виявляти етапи, на яких команда накопичила технічний борг або прострочила дедлайни.

4.3.3 Формування звітів та графіків

Оброблені аналітичні дані інтерактивно відображаються у клієнтському додатку за допомогою бібліотеки LiveCharts2. Оскільки структура даних з сервера не завжди підходить для прямого малювання графіків, було реалізовано допоміжний клас ChartDataMapper, який перетворює серійні дані у формат

об’єктів для осей X та Y. Що дозволяє миттєво відстежувати активність та продуктивність співробітників у вигляді наочних діаграм.

Для цілей зовнішнього звітування, наприклад комунікації керівництва з бухгалтерією чи відділом кадрів, система також підтримує вивантаження результатів у формат .xlsx. Це було реалізовано за допомогою бібліотеки ClosedXML [16], що дозволяє формувати звіти без необхідності встановлення пакету Microsoft Office на серверному обладнанні. Приклад реалізації методу формування Excel-файлу має такий вигляд:

```
public static void ExportUserAnalyticsReport(UserAnalyticsDto dto, string
filePath)
{
    using var wb = new XLWorkbook();
    var wsSummary = wb.Worksheets.Add("Зведення");


    wsSummary.Cell(1, 1).Value = "Звіт аналітики користувача";
    wsSummary.Range(1, 1, 1, 4).Merge().Style
        .Font.SetBold()
        .Font.SetFontSize(16)
        .Fill.SetBackgroundColor(XLColor.FromHtml("#EAF2FF"));


    wsSummary.Cell(3, 1).Value = "Користувач";
    wsSummary.Cell(3, 2).Value = dto.FullName;
    wsSummary.Cell(7, 1).Value = "Відпрацьовано годин";
    wsSummary.Cell(7, 2).Value = dto.WorkedHours;

    wsSummary.Columns().AdjustToContents(); // Авторозширення стовпців
    wb.SaveAs(filePath);
}
```

Бібліотека ClosedXML дозволяє повністю керувати внутрішньою структурою файлу на рівні C#-коду. Система автоматично створює кілька аркушів всередині одного файлу, застосовує умовне форматування клітинок та розставляє відповідні стилі.

4.4 Реалізація користувацького інтерфейсу

Користувацький інтерфейс системи TimeFlow спроєктовано з використанням технології WPF (рис. 4.1). Головним елементом навігації є статична бічна панель, яка дозволяє швидко перемикатися між ключовими модулями системи.

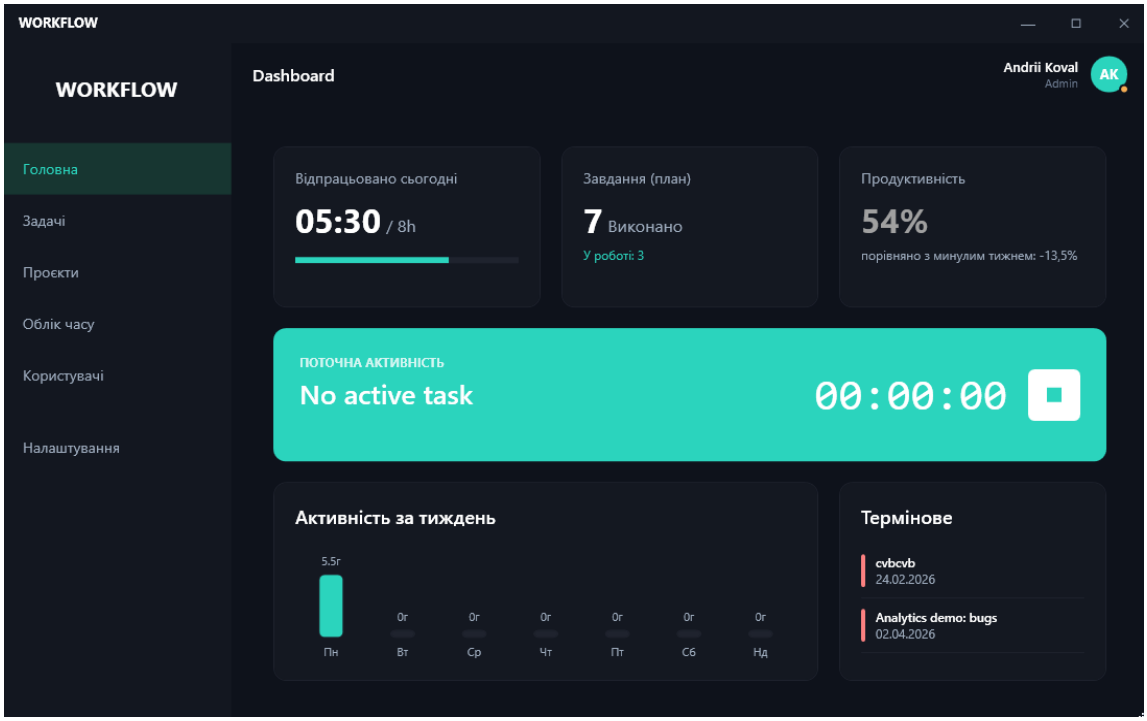


Рисунок 4.1 – Головна панель

Головна панель розроблена для швидкого моніторингу ключових показників користувача. Верхня частина екрану містить картки KPI, які в режимі реального часу відображають кількість відпрацьованих за сьогодні годин, загальну кількість завдань та кількість завдань, що потребують термінової уваги. У нижній частині розташовано інтерактивний графік тижневої активності, побудований на базі бібліотеки LiveCharts2.

Розділ завдань реалізовано у вигляді списку карток (рис. 4.2), кожна з яких містить коротку інформацію про дедлайн, пріоритет та виконавця.

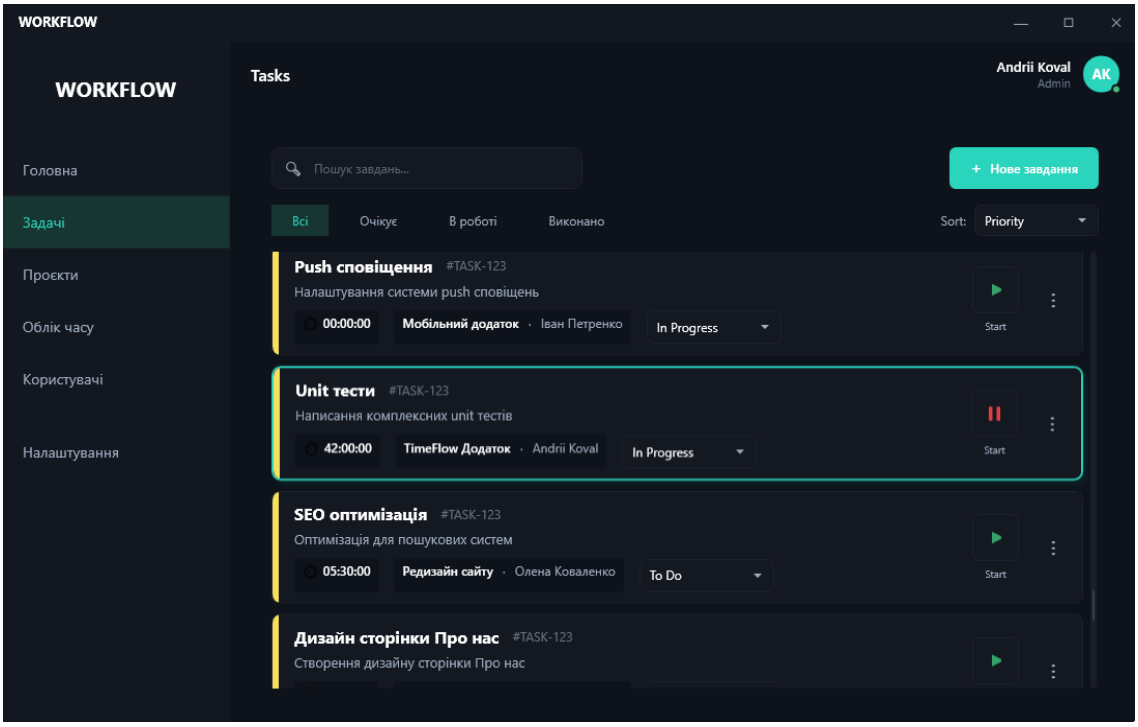


Рисунок 4.2 – Сторінка управління завданнями

Безпосередньо з картки завдання користувач може запустити таймер обліку часу одним кліком. Під час активної сесії таймер закріплюється у глобальному стані системи та продовжує відлік навіть при переході до інших розділів, що демонструє екран тайм-трекера (рис. 4.3).

У верхній частині вікна розташовано панель активного таймера, де користувач у реальному часі бачить, над якою задачею він працює та скільки годин і хвилин вже витрачено.

Нижче розташовується журнал активності за поточний день – деталізований перелік усіх завершених сесій роботи з позначенням фактично витраченого часу.

Для адміністраторів та менеджерів передбачено розділ Користувачі (рис. 4.4). Цей екран дозволяє керувати ролями співробітників та відображає їхній поточний статус присутності: Онлайн – зелений індикатор, з увімкненим таймером, Онлайн але без увімкненого таймеру – жовтий індикатор, або Офлайн – червоний.

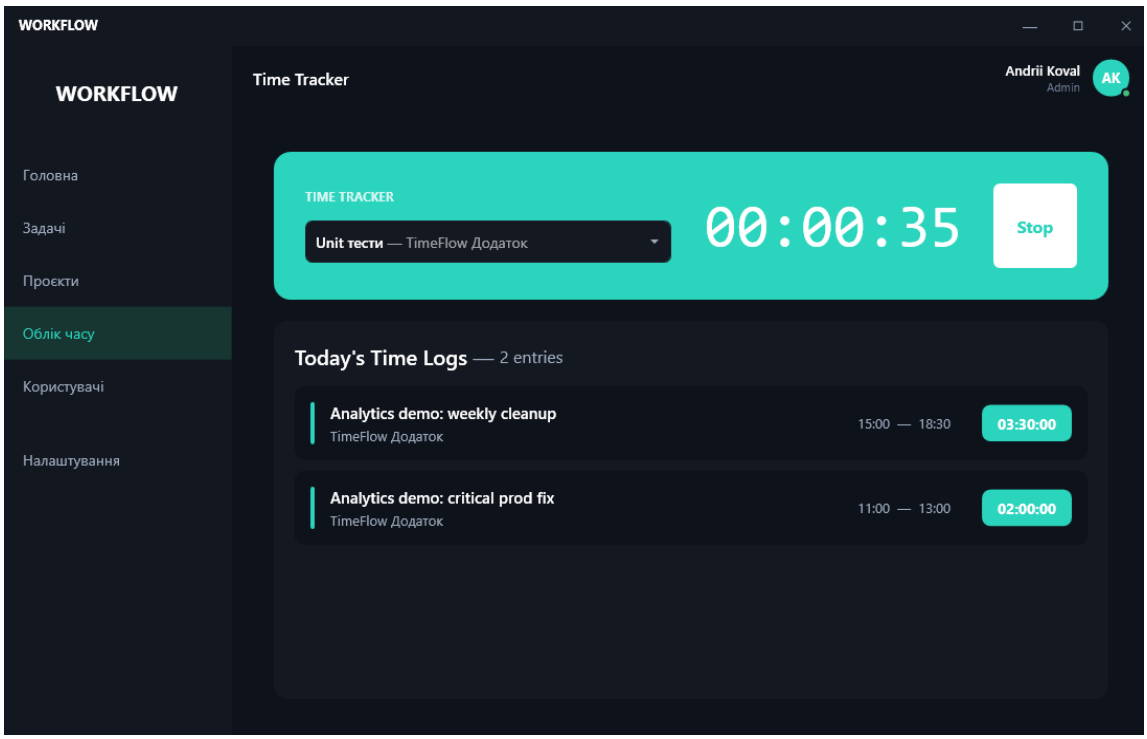


Рисунок 4.3 – Журнал робочого часу та активний таймер

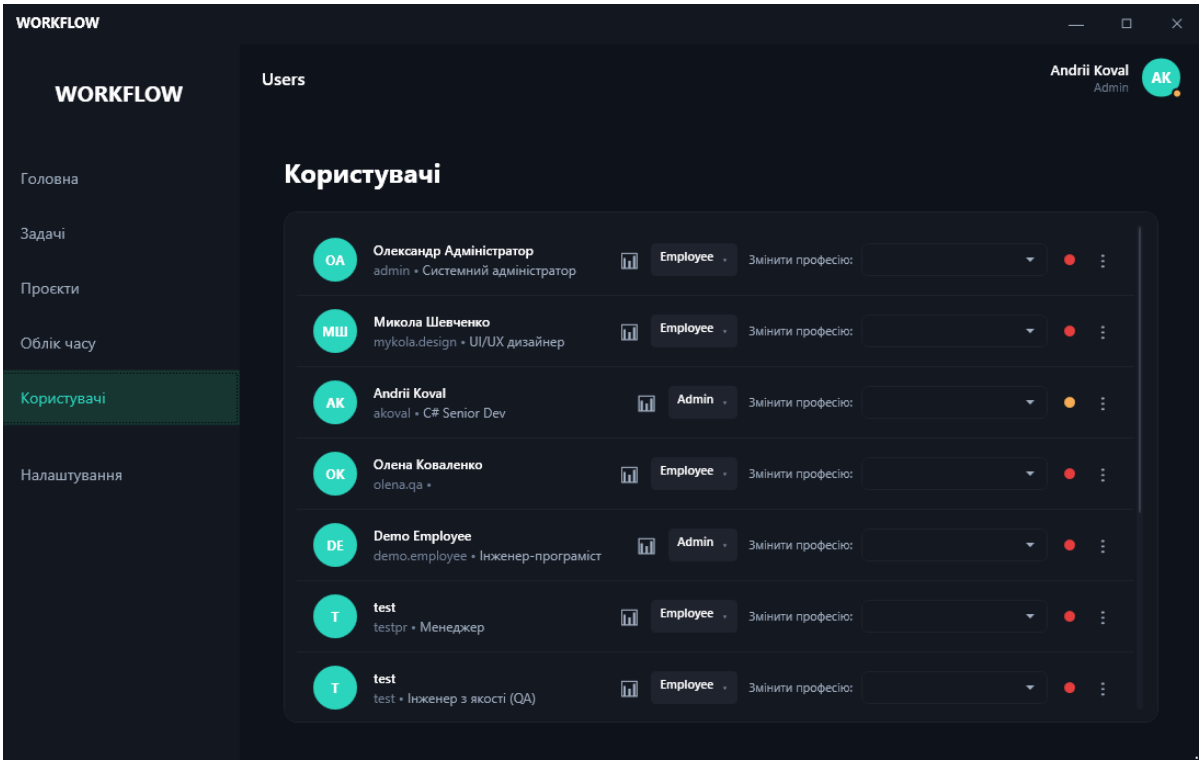


Рисунок 4.4 – Модуль керування персоналом

Найбільш складним візуальним компонентом є діалог розширеної аналітики працівника (рис. 4.5). Інтерфейс автоматично форматує дані, отримані з DTO-моделей. Зеленим кольором підсвічуються успішно виконані показники,

червоним – прострочені або негативні. З цього ж вікна менеджер або адміністратор може натиснути кнопку "Експорт в Excel" для збереження звіту на пристрій.

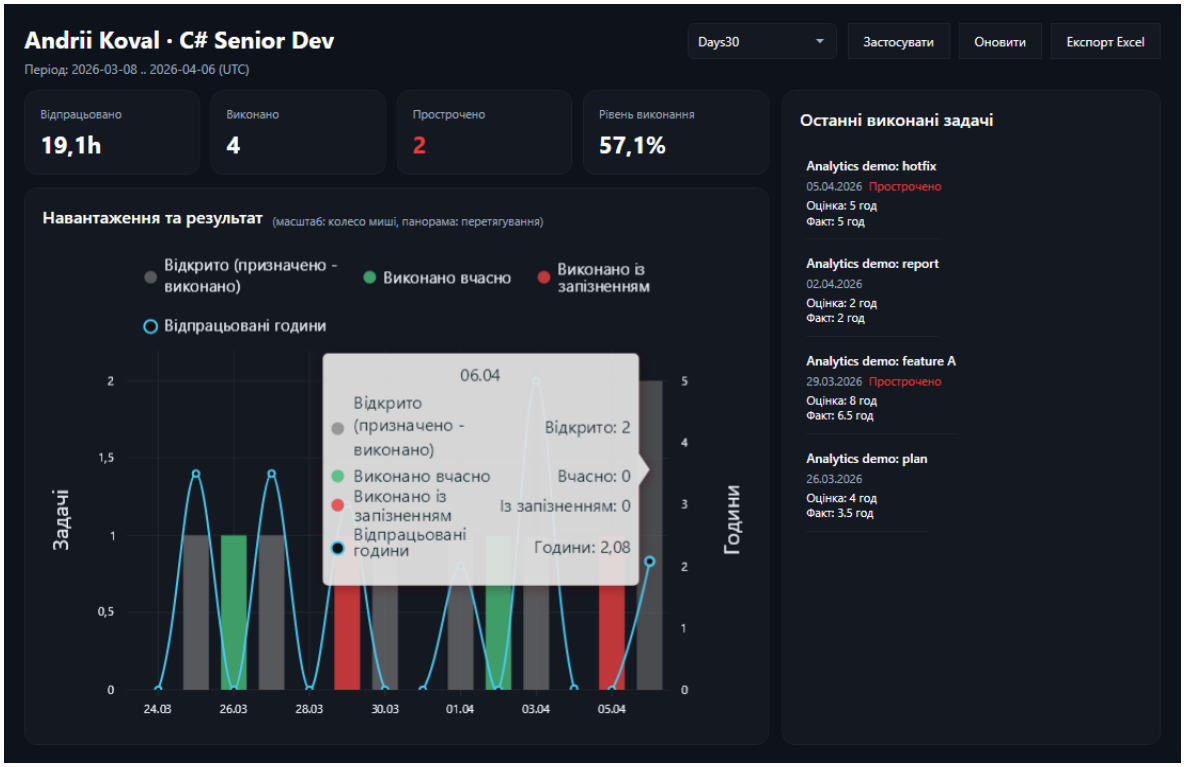
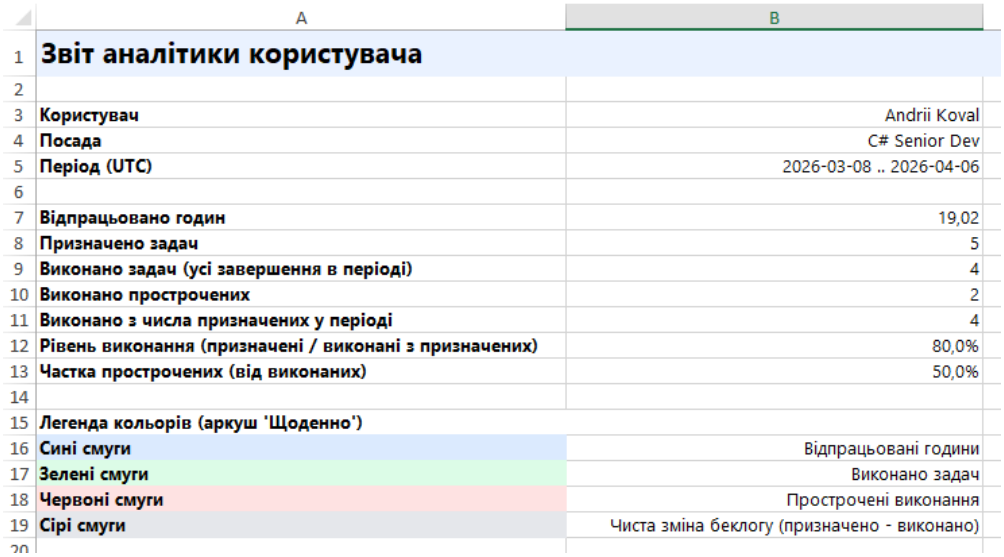


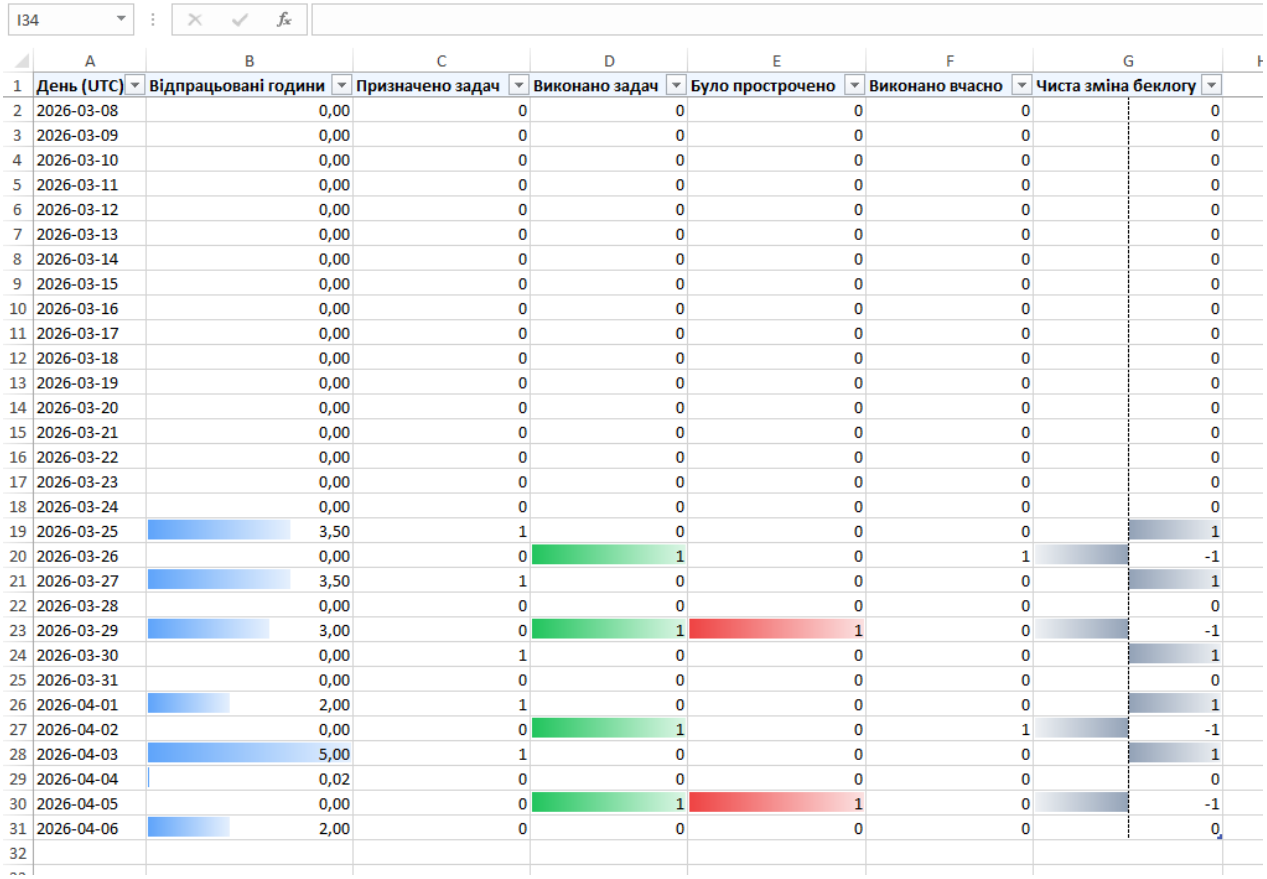
Рисунок 4.5 – Вікно аналітики та експорту звіту

Окрім візуальних графіків, тут також реалізовано збереження персональних звітів у форматі Excel – для відділу кадрів чи бухгалтерії, приклад наведено на рис. 4.6, 4.7 та 4.8.



	A	B
1	**Звіт аналітики користувача**	
2		
3	**Користувач**	Andrii Koval
4	**Посада**	C# Senior Dev
5	**Період (UTC)**	2026-03-08 .. 2026-04-06
6		
7	**Відпрацьовано годин**	19,02
8	**Призначено задач**	5
9	**Виконано задач (усі завершення в періоді)**	4
10	**Виконано прострочених**	2
11	**Виконано з числа призначених у періоді**	4
12	**Рівень виконання (призначені / виконані з призначених)**	80,0%
13	**Частка прострочених (від виконаних)**	50,0%
14		
15	**Легенда кольорів (аркуш 'Щоденно')**	
16	**Сині смуги**	Відпрацьовані години
17	**Зелені смуги**	Виконано задач
18	**Червоні смуги**	Прострочені виконання
19	**Сірі смуги**	Чиста зміна беклогу (призначено - виконано)

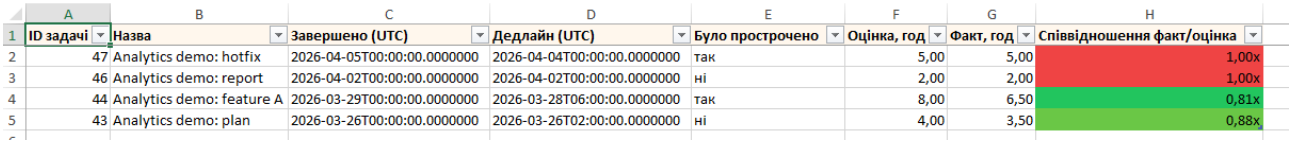
Рисунок 4.6 – Приклад Excel звіту по користувачу (Зведення)



I34

	A	B	C	D	E	F	G
1	**День (UTC)**	**Відпрацьовані години**	**Призначено задач**	**Виконано задач**	**Було прострочено**	**Виконано вчасно**	**Чиста зміна беклогу**
2	2026-03-08	0,00	0	0	0	0	0
3	2026-03-09	0,00	0	0	0	0	0
4	2026-03-10	0,00	0	0	0	0	0
5	2026-03-11	0,00	0	0	0	0	0
6	2026-03-12	0,00	0	0	0	0	0
7	2026-03-13	0,00	0	0	0	0	0
8	2026-03-14	0,00	0	0	0	0	0
9	2026-03-15	0,00	0	0	0	0	0
10	2026-03-16	0,00	0	0	0	0	0
11	2026-03-17	0,00	0	0	0	0	0
12	2026-03-18	0,00	0	0	0	0	0
13	2026-03-19	0,00	0	0	0	0	0
14	2026-03-20	0,00	0	0	0	0	0
15	2026-03-21	0,00	0	0	0	0	0
16	2026-03-22	0,00	0	0	0	0	0
17	2026-03-23	0,00	0	0	0	0	0
18	2026-03-24	0,00	0	0	0	0	0
19	2026-03-25	3,50	1	0	0	0	1
20	2026-03-26	0,00	0	1	0	1	-1
21	2026-03-27	3,50	1	0	0	0	1
22	2026-03-28	0,00	0	0	0	0	0
23	2026-03-29	3,00	0	1	1	0	-1
24	2026-03-30	0,00	1	0	0	0	1
25	2026-03-31	0,00	0	0	0	0	0
26	2026-04-01	2,00	1	0	0	0	1
27	2026-04-02	0,00	0	1	0	1	-1
28	2026-04-03	5,00	1	0	0	0	1
29	2026-04-04	0,02	0	0	0	0	0
30	2026-04-05	0,00	0	1	1	0	-1
31	2026-04-06	2,00	0	0	0	0	0
32							

Рисунок 4.7 – Приклад Excel звіту по користувачу (Щоденно)



	A	B	C	D	E	F	G	H
1	**ID задачі**	**Назва**	**Завершено (UTC)**	**Дедлайн (UTC)**	**Було прострочено**	**Оцінка, год**	**Факт, год**	**Співвідношення факт/оцінка**
2	47	Analytics demo: hotfix	2026-04-05T00:00:00.0000000	2026-04-04T00:00:00.0000000	так	5,00	5,00	1,00x
3	46	Analytics demo: report	2026-04-02T00:00:00.0000000	2026-04-02T00:00:00.0000000	ні	2,00	2,00	1,00x
4	44	Analytics demo: feature A	2026-03-29T00:00:00.0000000	2026-03-28T06:00:00.0000000	так	8,00	6,50	0,81x
5	43	Analytics demo: plan	2026-03-26T00:00:00.0000000	2026-03-26T02:00:00.0000000	ні	4,00	3,50	0,88x

Рисунок 4.8 – Приклад Excel звіту по користувачу (Останні)

Для структурування завдань у системі передбачено сторінку управління проєктами (рис. 4.9). Кожен проєкт представлено у вигляді інформативної картки, яка одразу показує кількісний прогрес, скільки завдань відведено на проєкт і скільки з них уже виконано.

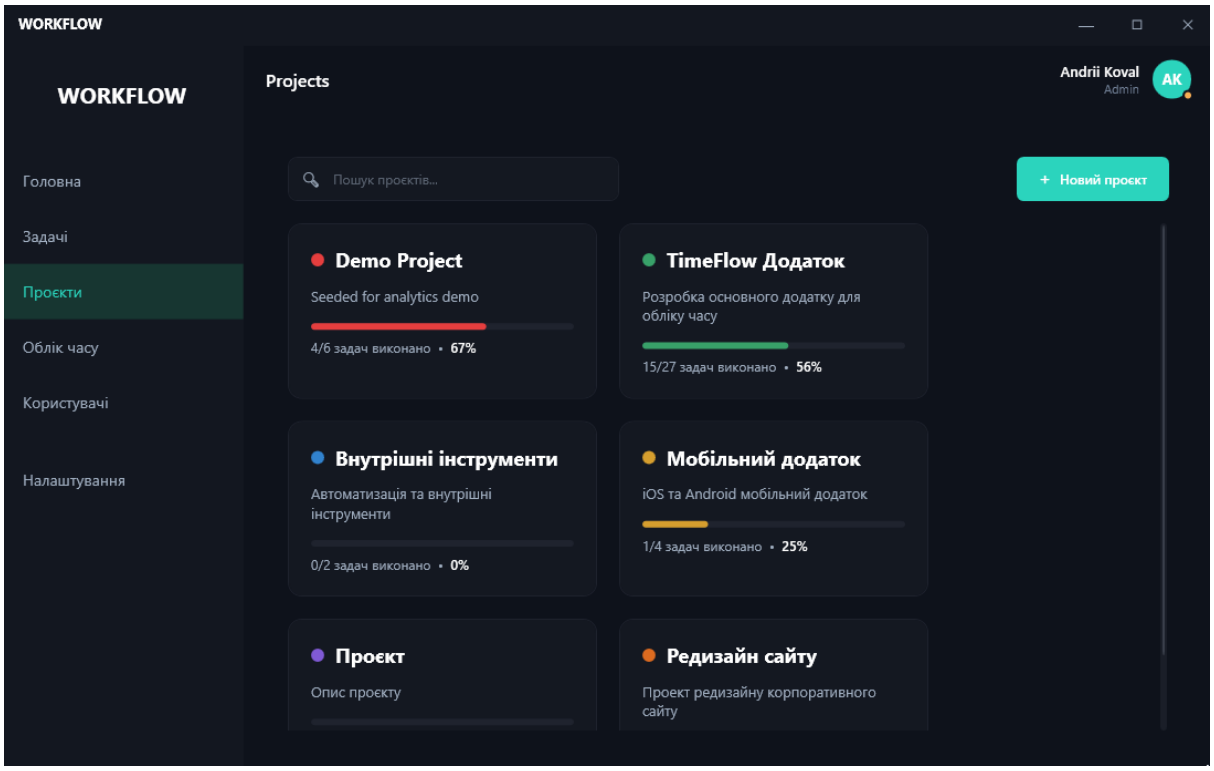


Рисунок 4.9 – Інтерфейс управління проєктами

Функцією цього модуля є можливість вивести аналітику по всьому проєкту за допомогою спеціального діалогового вікна (рис. 4.10). В аналітиці проєкту керівник бачить загальну кількість відпрацьованих годин усією командою, графік динаміки виконання завдань по днях та співвідношення виконаних і прострочених етапів.

Так само як і для персональної статистики, аналітику проєкту можна вивантажити в Excel-звіт. Згенерований .xlsx файл містить повну історію від зведення витрачених годин до переліку останніх виконаних завдань із зазначенням імен виконавців та порівнянням фактичного часу з попередньою оцінкою, приклад наведено на рис. 4.11 – 4.13.



Рисунок 4.10 – Вигляд вбудованої аналітики

	A	B	C
1	**Звіт аналітики проєкту**		
2			
3	**Проєкт**	TimeFlow Додаток	
4	**Період (UTC)**	2026-03-08 .. 2026-04-06	
5			
6	**Відпрацьовано годин**	38,02	
7	**Призначено задач**	10	
8	**Виконано задач**	8	
9	**Прострочено виконано**	4	
10			
11	**Легенда кольорів (аркуш 'Щоденно')**		
12	**Сині смуги**	Відпрацьовані години	
13	**Сірі смуги**	Призначено задач	
14	**Зелені смуги**	Виконано задач	
15	**Червоні смуги**	Прострочені виконання	
16			

Рисунок 4.11 – Приклад Excel звіту по проєкту (Зведення)

	A	B	C	D	E
1	День (UTC)	Відпрацьовані години	Призначено задач	Виконано задач	Було прострочено
5	2026-03-11	0,00	0	0	0
6	2026-03-12	0,00	0	0	0
7	2026-03-13	0,00	0	0	0
8	2026-03-14	0,00	0	0	0
9	2026-03-15	0,00	0	0	0
10	2026-03-16	0,00	0	0	0
11	2026-03-17	0,00	0	0	0
12	2026-03-18	0,00	0	0	0
13	2026-03-19	0,00	0	0	0
14	2026-03-20	0,00	0	0	0
15	2026-03-21	0,00	0	0	0
16	2026-03-22	0,00	0	0	0
17	2026-03-23	0,00	0	0	0
18	2026-03-24	0,00	0	0	0
19	2026-03-25	7,00	2	0	0
20	2026-03-26	0,00	0	2	0
21	2026-03-27	7,00	2	0	0
22	2026-03-28	0,00	0	0	0
23	2026-03-29	6,00	0	2	2
24	2026-03-30	0,00	2	0	0
25	2026-03-31	0,00	0	0	0
26	2026-04-01	4,00	2	0	0
27	2026-04-02	0,00	0	2	0
28	2026-04-03	10,00	2	0	0
29	2026-04-04	0,02	0	0	0
30	2026-04-05	0,00	0	2	2
31	2026-04-06	4,00	0	0	0

Рисунок 4.12 – Приклад Excel звіту по проєкту (Щоденно)

	A	B	C	D	E	F	G	H	I
1	ID задачі	Назва	Виконавець	Завершено (UTC)	Дедлайн (UTC)	Було прострочено	Оцінка, год	Факт, год	Співвідношення факт/оцінка
2	39	Analytics demo: hotfix	Іван Петренко	2026-04-05T00:00:00.0000000	2026-04-04T00:00:00.0000000	так	5,00	5,00	1,00x
3	47	Analytics demo: hotfix	Andrii Koval	2026-04-05T00:00:00.0000000	2026-04-04T00:00:00.0000000	так	5,00	5,00	1,00x
4	38	Analytics demo: report	Іван Петренко	2026-04-02T00:00:00.0000000	2026-04-02T00:00:00.0000000	ні	2,00	2,00	1,00x
5	46	Analytics demo: report	Andrii Koval	2026-04-02T00:00:00.0000000	2026-04-02T00:00:00.0000000	ні	2,00	2,00	1,00x
6	36	Analytics demo: feature A	Іван Петренко	2026-03-29T00:00:00.0000000	2026-03-28T06:00:00.0000000	так	8,00	6,50	0,81x
7	44	Analytics demo: feature A	Andrii Koval	2026-03-29T00:00:00.0000000	2026-03-28T06:00:00.0000000	так	8,00	6,50	0,81x
8	35	Analytics demo: plan	Іван Петренко	2026-03-26T00:00:00.0000000	2026-03-26T02:00:00.0000000	ні	4,00	3,50	0,88x
9	43	Analytics demo: plan	Andrii Koval	2026-03-26T00:00:00.0000000	2026-03-26T02:00:00.0000000	ні	4,00	3,50	0,88x
10									

Рисунок 4.13 – Приклад Excel звіту по проєкту (Останні)

ВИСНОВКИ

У ході виконання дипломної роботи було досліджено підходи до організації управління робочими завданнями та обліку робочого часу, а також проаналізовано сучасні інформаційні системи даного класу. Проведений аналіз показав, що існуючі рішення або є надмірно складними для використання в малих і середніх командах, або ж, навпаки, не надають достатнього рівня функціональності для повноцінного контролю та аналізу робочих процесів.

Розроблений прототип інформаційної системи дозволила досягти балансу між простотою використання та функціональною насиченістю. Основним результатом є створення інструменту, який не лише автоматизує управління завданнями, але й формує об’єктивне уявлення про ефективність роботи співробітників. На відміну від класичних систем, де акцент зроблено лише на організації задач, у розробленому рішенні ключову роль відіграє інтегроване відстеження часу та формування аналітики, що трансформує систему з інструменту контролю у засіб підтримки управлінських рішень.

У процесі дослідження було підтверджено, що використання таймера як основного методу обліку часу – повністю забезпечує оптимальне співвідношення між точністю даних та зручністю для користувача. Крім того, впровадження автоматизованого розрахунку метрик продуктивності дозволяє зменшити суб’єктивність оцінки праці та перейти до прийняття рішень на основі даних.

Практичне значення розробки полягає у можливості її застосування в умовах підприємств малого та середнього бізнесу, де важливими є як економічна доцільність, так і конфіденційність даних. Використання локальної моделі розгортання дозволяє невеликим організаціям повністю контролювати власні дані без залежності від сторонніх сервісів.

У ході виконання роботи також було накопичено низку ідей щодо подальшого розвитку системи. До найперспективніших напрямків можна

віднести – інтеграцію механізмів прогнозування строків виконання завдань на основі накопичених історичних даних, та розширення аналітичного модуля з використанням більш складних метрик та візуалізацій.

Отже, розроблене рішення демонструє, що ефективна система управління завданнями не обмежується лише фіксацією дій користувача, а виступає інструментом формування цифрової моделі робочого процесу, яка дозволяє виявляти закономірності, оптимізувати використання ресурсів та підвищувати загальну продуктивність організації.

ПЕРЕЛІК ПОСИЛАНЬ

1. Can digital transformation enhance labor productivity in enterprises. PLOS.One URL : https://journals.plos.org/plosone/article?id=10.1371/journal.pone.0325484 (дата звернення : 24.11.2025).

2. C# language documentation. URL : https://learn.microsoft.com/uk-ua/dotnet/csharp/ (дата звернення 25.11.2025).

3. Tanenbaum A. S., Wetherall D. Computer Networks. Pearson Education, 2011.

4. Kanban посібник. Atlassian Agile Guide. URL : https://www.atlassian.com/ agile/kanban (дата звернення : 27.11.2025).

5. All The Ways to Track Time in Toggl Track. Toggl Track Documentation. URL : https://toggl.com/track/ways-to-log-time/ (дата звернення : 27.11.2025).

6. Common web application architectures. Документація Microsoft: .NET Architecture. URL : https://learn.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures#clean-architecture (дата звернення : 02.02.2026).

7. WPF Apps With The Model-View-ViewModel Design Pattern. Документація Microsoft Learn. URL : https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern (дата звернення : 02.02.2026).

8. Introduction to the MVVM Toolkit. URL : https://learn.microsoft.com/en-us/dotnet/communitytoolkit/mvvm/ (дата звернення : 06.04.2026).

9. Overview of ASP.NET Core SignalR. URL : https://learn.microsoft.com/en-us/aspnet/core/signalr/introduction (дата звернення : 02.02.2026).

10. PostgreSQL 16.13 Documentation. PostgreSQL Global Development Group. URL : https://www.postgresql.org/docs/16/index.html (дата звернення : 07.02.2026).

11. ASP.NET Core fundamentals overview. URL : https://learn.microsoft.com/en-us/aspnet/core/fundamentals/ (дата звернення : 19.02.2026).

12. Data binding overview. Документація Microsoft Docs. URL : https://learn.microsoft.com/en-us/dotnet/desktop/wpf/data/ (дата звернення : 19.02.2026).

13. Entity Framework Core. Microsoft Learn. URL : https://learn.microsoft.com/en-us/ef/core/ (дата звернення : 19.02.2026).

14. Візуалізація даних за допомогою LiveCharts2. Офіційна документація LiveCharts. URL : https://livecharts.dev/ (дата звернення : 13.03.2026).

15. Manning C. D., Raghavan P., Schütze H. Introduction to Information Retrieval. Cambridge University Press, 2008.

16. Керівництво по роботі з Excel-файлами : Бібліотека ClosedXML. GitHub Wiki. URL : https://github.com/ClosedXML/ClosedXML/wiki (дата звернення : 15.03.2026).

ДОДАТОК А

Серверна частина інформаційної системи

А.1 Конфігурація серверного додатку

```
using diplom.Data;
using diplom.API.Services;
using diplom.API.Hubs;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using System.Text;
using System.Security.Claims;
using System.Text.Json.Serialization;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>();

builder.Services.AddScoped<IUserActivityService, UserActivityService>();

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

        options.Events = new JwtBearerEvents
        {
            OnTokenValidated = async context =>
            {
                var userIdClaim =
context.Principal?.FindFirst(ClaimTypes.NameIdentifier)?.Value;
                var sidClaim = context.Principal?.FindFirst("sid")?.Value
```

```
                    ??
context.Principal?.FindFirst(ClaimTypes.Sid)?.Value;
                if (!int.TryParse(userIdClaim, out var userId))
                {
                    context.Fail("Invalid user");
                    return;
                }

                var db =
context.HttpContext.RequestServices.GetRequiredService<AppDbContext>();
                var user = await db.Users.FindAsync(userId);
                if (user == null)
                {
                    context.Fail("Invalid user");
                    return;
                }

                var currentSid = user.CurrentSessionId?.ToString() ??
string.Empty;
                if (!string.Equals(sidClaim, currentSid,
StringComparison.Ordinal))
                {
                    context.Fail("Session revoked");
                }
            }
        };
    });

builder.Services.AddControllers()
    .AddJsonOptions(options =>
    {
        options.JsonSerializerOptions.ReferenceHandler =
ReferenceHandler.IgnoreCycles;
        options.JsonSerializerOptions.DefaultIgnoreCondition =
JsonIgnoreCondition.WhenWritingNull;
        options.JsonSerializerOptions.Converters.Add(new
JsonStringEnumConverter());
    });

builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.AddSecurityDefinition("Bearer", new
Microsoft.OpenApi.Models.OpenApiSecurityScheme
    {
        Description = "JWT Token: Bearer {token}",
        Name = "Authorization",
        In = Microsoft.OpenApi.Models.ParameterLocation.Header,
        Type = Microsoft.OpenApi.Models.SecuritySchemeType.ApiKey,
        Scheme = "Bearer"
    });

    c.AddSecurityRequirement(new
Microsoft.OpenApi.Models.OpenApiSecurityRequirement
    {
        {
            new Microsoft.OpenApi.Models.OpenApiSecurityScheme
            {
```

```
                Reference = new Microsoft.OpenApi.Models.OpenApiReference
                {
                    Type =
Microsoft.OpenApi.Models.ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            Array.Empty<string>()
        }
    });
});

builder.Services.AddCors(options =>
{
    options.AddDefaultPolicy(policy =>
    {
        policy.AllowAnyOrigin()
              .AllowAnyMethod()
              .AllowAnyHeader();
    });
});

builder.Services.AddSignalR();

var app = builder.Build();

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}

app.UseCors();
app.UseAuthentication();
app.UseAuthorization();
app.MapControllers();
app.MapHub<TimeHub>("/hubs/time");

app.Run();
```

А.2 Методи контролеру роботи с завданнями

```
[HttpPost]
        public async Task<ActionResult<TaskItem>> Create([FromBody]
TaskItem task)
        {
            var userId = GetCurrentUserId();
            var isAdminOrManager = IsAdminOrManager();

            if (!isAdminOrManager)
            {
                task.AssigneeId = userId;
            }
```

```
            task.CreatedAt = DateTime.UtcNow;
            if (task.AssigneeId.HasValue)
                task.AssignedAtUtc = DateTime.UtcNow;
            task.CompletedAtUtc = task.Status == AppTaskStatus.Done ?
DateTime.UtcNow : null;
            _context.Tasks.Add(task);
            await _context.SaveChangesAsync();

            var created = await _context.Tasks
                .Include(t => t.Project)
                .Include(t => t.TimeEntries)
                .FirstOrDefaultAsync(t => t.Id == task.Id);

            await _hub.Clients.All.SendAsync("TaskChanged", task.Id);

            return CreatedAtAction(nameof(GetById), new { id = task.Id },
created);
        }

        [HttpPut("{id}")]
        public async Task<ActionResult<TaskItem>> Update(int id,
[FromBody] TaskItem task)
        {
            var userId = GetCurrentUserId();
            var isAdminOrManager = IsAdminOrManager();

            var existing = await _context.Tasks.FindAsync(id);
            if (existing == null)
                return NotFound();

            if (!isAdminOrManager)
            {
                if (existing.AssigneeId != userId)
                    return Forbid();
            }

            existing.Title = task.Title;
            existing.Description = task.Description;

            if (existing.Status != task.Status)
            {
                if (task.Status == AppTaskStatus.Done)
                    existing.CompletedAtUtc = DateTime.UtcNow;
                else if (existing.Status == AppTaskStatus.Done)
                    existing.CompletedAtUtc = null;

                existing.Status = task.Status;
            }
            existing.Priority = task.Priority;
            existing.Deadline = task.Deadline;
            existing.EstimatedHours = task.EstimatedHours;

            if (isAdminOrManager)
            {
                existing.ProjectId = task.ProjectId;
                if (existing.AssigneeId != task.AssigneeId)
                {
```

```
                    existing.AssigneeId = task.AssigneeId;
                    existing.AssignedAtUtc = task.AssigneeId.HasValue ?
DateTime.UtcNow : null;
                }
            }

            await _context.SaveChangesAsync();
            await _hub.Clients.All.SendAsync("TaskChanged", existing.Id);
            return Ok(existing);
        }

        [HttpDelete("{id}")]
        public async Task<IActionResult> Delete(int id)
        {
            var userId = GetCurrentUserId();
            var isAdminOrManager = IsAdminOrManager();

            var task = await _context.Tasks.FindAsync(id);
            if (task == null)
                return NotFound();

            if (!isAdminOrManager)
            {
                if (task.AssigneeId != userId)
                    return Forbid();
            }

            _context.Tasks.Remove(task);
            await _context.SaveChangesAsync();
            await _hub.Clients.All.SendAsync("TaskChanged", task.Id);
            return NoContent();
        }
```

ДОДАТОК Б

Діаграма архітектурних зв’язків у клієнтському додатку

