ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Розробка програмного забезпечення для аналізу та управління процесами тайм-менеджменту»

на здобуття освітнього ступеня бакалавра
зі спеціальності 121 Інженерія програмного забезпечення
освітньо-професійної програми «Інженерія програмного забезпечення»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Ілля НОВОХАЦЬКИЙ
(підпис)

Виконав: здобувач вищої освіти групи ПД-43
Ілля НОВОХАЦЬКИЙ

Керівник: Богдан ХУДІК
доктор філософії (PhD)

Рецензент: _____

Київ 2026

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Інженерії програмного забезпечення

Ступінь вищої освіти Бакалавр

Спеціальність 121 Інженерія програмного забезпечення

Освітньо-професійна програма «Інженерія програмного забезпечення»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Інженерії програмного забезпечення
___ Ірина ЗАМРІЙ
« ____ » _____ 2026 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Новохацькому Іллі Олеговичу

1. Тема кваліфікаційної роботи: «Розробка програмного забезпечення для аналізу та управління процесами тайм-менеджменту»
керівник кваліфікаційної роботи доктор філософії (PhD) Богдан ХУДІК,
затверджені наказом Державного університету інформаційно-комунікаційних технологій від «20» лютого 2026 року № 51.

2. Строк подання кваліфікаційної роботи «29» травня 2026 року.

3. Вихідні дані до кваліфікаційної роботи: теоретичні відомості про Agile методології планування та методи персонального тайм-менеджменту; принципи проєктування розподілених клієнт-серверних архітектур і реляційних баз даних; технічна документація екосистеми .NET, СУБД PostgreSQL, а також специфікації протоколів криптографічного захисту JWT та інтеграції зовнішніх сервісів Google OAuth2.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

1. Дослідження методів персонального планування та аналіз можливостей адаптації гнучких методологій (Agile) для індивідуального управління часом.

2. Проєктування розподіленої інфокомунікаційної системи для ітеративного планування та аналізу особистої продуктивності.

3. Програмна реалізація та опис функціонування клієнт-серверного застосунку на базі екосистеми .NET.

4. Тестування та верифікація роботи програмного комплексу

5. Перелік графічного матеріалу: *презентація*

1. Аналіз аналогів.
2. Вимоги до програмного забезпечення: функціональні вимоги.
3. Вимоги до програмного забезпечення: нефункціональні вимоги
4. Програмні засоби реалізації.
5. Діаграма прецедентів.
6. Діаграма архітектурної взаємодії
7. Діаграма класів
8. ER-діаграма бази даних.
9. Діаграма класів.
10. Екранні форми.
11. Відеодемонстрація застосунку
12. Апробація результатів дослідження

6. Дата видачі завдання «21» лютого 2026 року.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Підбір та аналіз науково-технічної літератури	21.02-28.02.2026	
2	Аналіз та дослідження існуючих аналогів	01.03-07.03.2026	
3	Обґрунтувати вибір засобів розробки: екосистема .NET9, клієнт-серверна архітектура	08.03-15.03.2026	
4	Розробити об'єктно-реляційну модель предметної області, визначити зв'язки між сутностями.	16.03-22.03.2026	
5	Розробити серверну бізнес-логіку, механізми управління завданнями та спринтами, а також створити графічний інтерфейс.	23.03-15.04.2026	
6	Реалізувати гібридну систему авторизації та забезпечити криптографічну ізоляцію персональної інформації користувачів.	16.04-22.04.2026	

7	Провести тестування клієнту та серверу окремо за допомогою Junit та інтеграційних тестів, провести тестування клієнт-серверної взаємодії.	23.04-30.04.2026	
8	Оформлення роботи: вступ, висновки, реферат	01.05-10.05.2026	
9	Розробка демонстраційних матеріалів	11.05-22.05.2026	
10	Попередній захист роботи	11.05-22.05.2026	
11	Подання кваліфікаційної роботи	23.05-29.05.2026	

Здобувач вищої освіти _____ Ілля НОВОХАЦЬКИЙ
(підпис)

Керівник
кваліфікаційної роботи _____ Богдан ХУДІК
(підпис)

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття освітнього ступеня бакалавра: 96 стор., 1 табл. , 16 рис. , 21 джерело.

Мета роботи – покращення управління процесами тайм-менеджменту користувачів шляхом програмної адаптації корпоративних гнучких методологій (Agile) для індивідуального використання.

Об'єкт дослідження – процеси розробки програмних систем для організації, планування та управління часом користувачів.

Предмет дослідження – методи та програмні засоби (зокрема, архітектура «клієнт-сервер», технології .NET та фреймворк .NET MAUI) для реалізації застосунку для тайм-менеджменту.

Короткий зміст роботи: В роботі проаналізовано когнітивні особливості процесу персонального планування та недоліки існуючих програмних засобів, зокрема Jira, Todoist, Trello, Toggl Track та Microsoft To Do. Спроєктовано нормалізовану об'єктно-реляційну базу даних та трирівневу клієнт-серверну архітектуру. Розроблено алгоритм роботи застосунку та програмно реалізовано ключові функціональні можливості: безпечну гібридну авторизацію (включно з Google OAuth2) , управління життєвим циклом завдань за допомогою просторового інтерфейсу Drag-and-Drop, планування робочого навантаження часовими ітераціями, автоматичну генерацію зведеної аналітики продуктивності, а також асинхронну відправку Push-сповіщень через фонові черги. Проведено модульне та інтеграційне тестування серверного API і клієнтського застосунку за допомогою фреймворку xUnit. В роботі використано платформу .NET 9 (фреймворки ASP.NET Core та .NET MAUI) , СУБД PostgreSQL у зв'язці з Entity Framework Core , технологію SignalR (WebSockets) та платформу Docker для розгортання.

Сферою використання застосунку є організація індивідуального робочого процесу IT-спеціалістів, фрілансерів та студентів для запобігання когнітивному перевантаженню та підвищення особистої продуктивності.

КЛЮЧОВІ СЛОВА: ТАЙМ-МЕНЕДЖМЕНТ, AGILE, СПРИНТ, КЛІЄНТ-СЕРВЕРНА АРХІТЕКТУРА, .NET MAUI, ASP.NET CORE, POSTGRESQL, JWT, DOCKER.

ЗМІСТ

ВСТУП

Актуальність: Сучасний етап розвитку інформаційного суспільства характеризується безпрецедентним зростанням обсягів даних та інтенсифікацією робочих процесів. В умовах глобальної цифровізації, переходу до гібридних форматів роботи та безперервного потоку комунікацій, здатність людини ефективно керувати власним часом стає критичним фактором не лише професійного успіху, але й збереження психологічного здоров'я. Нездатність структурувати робоче навантаження призводить до явища когнітивного перевантаження, прокрастинації та хронічного стресу.

Незважаючи на велику кількість програмних продуктів для ведення списків справ, більшість із них мають суттєві архітектурні та концептуальні недоліки. Вони або пропонують занадто примітивні лінійні списки, які не дозволяють гнучко пріоритезувати завдання, або, навпаки, є громіздкими корпоративними системами (на кшталт Jira), які вимагають надмірного часу на адміністрування та не підходять для персонального використання. Крім того, багатьом існуючим рішенням бракує інструментів для глибокої аналітики власної продуктивності та підтримки ітеративного планування, адаптованого під індивідуальні потреби.

Об'єктом дослідження є процеси розробки програмних систем для організації, планування та управління часом користувачів

Предметом дослідження є методи та програмні засоби (зокрема, архітектура «клієнт-сервер», технології .NET та фреймворк .NET MAUI) для реалізації застосунку для тайм-менеджменту.

Метою роботи є покращення управління процесами тайм-менеджменту користувачів.

Для досягнення поставленої мети першочергово було проведено глибокий аналіз існуючих програмних рішень у сфері персонального тайм-менеджменту та управління завданнями, що дозволило сформулювати вичерпний комплекс функціональних і нефункціональних вимог до проєктуємої системи. Наступним

кроком став вибір сучасного інженерного стеку технологій: кросплатформний фреймворк .NET MAUI (XAML, C#) з архітектурним патерном MVVM для розробки клієнтського рівня, платформа ASP.NET Core (.NET 9) для реалізації централізованого серверного REST API, транзакційна СУБД PostgreSQL у поєднанні з Entity Framework Core для надійного збереження даних, а також механізми JWT та SignalR для забезпечення безпеки і двосторонньої комунікації. На фінальному етапі розробки було протестовано надійність клієнт-серверної взаємодії, швидкодію нативного графічного інтерфейсу на операційних системах macOS та Windows, а також коректність виконання закладених бізнес-алгоритмів і фонових процесів.

Наукова новизна роботи полягає в удосконаленні моделі персонального тайм-менеджменту шляхом програмної адаптації корпоративних agile методологій для індивідуального використання. Вперше для подібних персональних систем запропоновано оптимізований підхід до управління життєвим циклом завдань, який базується на гнучкій маршрутизації робочого навантаження та просторовому розподілі активностей у межах суворих часових ітерацій. Також набула подальшого розвитку архітектурна модель розробки розподілених клієнт-серверних рішень завдяки ефективній комбінації декларативного проєктування нативних інтерфейсів, stateless-архітектури бекенду та повнодуплексних каналів зв'язку. Усе це дозволило розробити комплексну інфокомунікаційну систему, яка забезпечує миттєву транзакційну синхронізацію даних у режимі реального часу без надмірного споживання апаратних ресурсів пристрою користувача.

Практична значущість одержаних результатів полягає в тому, що розроблена інфокомунікаційна система є повністю функціональним програмним продуктом, готовим до впровадження та повсякденного використання. Її застосування дозволяє кінцевим користувачам ефективно боротися з когнітивним перевантаженням, раціонально структурувати робочі процеси та підвищувати особисту ефективність за допомогою адаптованих методів ітеративного планування.

1 АНАЛІЗ МЕНЕДЖМЕНТУ ЧАСУ

1.1. Загальна характеристика предметної галузі та актуальність задачі

У сучасному інформаційно орієнтованому суспільстві тайм-менеджмент трансформувався з простої побутової навички у критично важливий інструмент забезпечення професійної та особистої ефективності. В умовах вічної цифровізації та безперервного потоку даних головне завдання менеджменту часу полягає у забезпеченні користувача засобами для структурування робочого навантаження, гнучкої пріоритезації задач та об’єктивного аналізу витраченого часу.

Згідно з дослідженнями у сфері ІПЗ та психології, сучасні спеціалісти (зокрема розробники програмного забезпечення, системні адміністратори, фрілансери, а також студенти ІТ-спеціальностей) щоденно стикаються з проблемою критично високого розумового навантаження. Специфіка інтелектуальної праці вимагає глибокого фокусування, проте реалії робочого процесу часто змушують до мультитаскінгу, до перестрибувань з однієї задачі на іншу. Постійне перемикання контексту між різноплановими завданнями, проєктами та комунікаційними каналами призводить до явища «залишку уваги», що суттєво виснажує нервову систему та знижує загальну продуктивність на 20-40% [1]. Відповідно, виникає гостра потреба у використанні спеціалізованих засобів, які б автоматизували процес управління життєвим циклом завдань і знімали частину рутинного навантаження з користувача.

Хоча проблема управління часом має глобальний характер і тягнеться за суспільством вже не перший рік, традиційні підходи до її вирішення демонструють свою неспроможність в умовах багатозадачності. Класичні інструменти, такі як паперові планувальники або прості електронні нотатки, є статичними та неефективними для ведення складних багатоетапних проєктів, серед невідсортованих справ легко загубитись, а величезна «хмара» завдань, нависаюча над користувачем, призводить до втрати мотивації та підвищення ризику

прокрастинації. Ці засоби страждають від відсутності структурних зв'язків, що унеможливлює логічне групування завдань за категоріями або робочими циклами з подальшим аналізом зробленого. Крім того, виникає складність контролю дедлайнів через відсутність автоматичних механізмів відстеження протермінованих завдань, що суттєво підвищує ризик зриву графіків. Ще одним критичним недоліком є неможливість аналітики, оскільки відсутність інструментів для збору статистичних даних блокує проведення ретроспективного аналізу діяльності та об'єктивної оцінки власної продуктивності.

1.2. Аналіз цільової аудиторії та її специфічних потреб

Розробка ефективного ПЗ вимагає чіткого розуміння специфіки цільової аудиторії та її поведінкових патернів. У контексті даного дослідження головними користувачами виступають індивідуальні розробники, ІТ-спеціалісти, фрілансери та студенти технічних спеціальностей. Фундаментальною особливістю цієї групи є те, що користувач виступає одночасно у двох ролях: як «менеджер», який здійснює стратегічне планування та розподіл обсягу робіт, і як безпосередній «виконавець», який ці задачі реалізує. Таке поєднання часто призводить до внутрішнього конфлікту інтересів та емоційного вигорання, оскільки процеси планування та виконання вимагають різних підходів [2].

Для вирішення цієї суперечності сучасна концепція управління часом вимагає впровадження ітеративного планування. Перехід від нескінченних лінійних списків справ до методології дозволяє розбивати глобальні цілі на короткі, чітко обмежені в часі робочі цикли у формі індивідуальних спринтів. Це не лише забезпечує кращий контроль прогресу, але й створює психологічний ефект завершеності наприкінці кожної ітерації. Важливою складовою такого підходу є інтелектуальна аналітика замість ручного мікроменеджменту. Система має фоново фіксувати життєвий цикл завдань та аналізувати дотримання дедлайнів. Такий об'єктивний збір статистики дозволяє оцінювати періоди максимальної концентрації та продуктивності користувача, допомагаючи йому адаптувати

подальше навантаження на основі реальних даних, а не суб'єктивних відчуттів.

Важливим критерієм успішності та користі застосунку є прагнення до мінімізації кроків для створення завдання. Програмний продукт повинен мати інтуїтивно зрозумілий, візуально розвантажений інтерфейс, який виключає необхідність заповнення складних форм для базових операцій із завданнями. Взаємодія з робочим простором має відбуватися максимально природно, що дозволяючи змістити фокус уваги з адміністрування системи безпосередньо на виконання роботи.

Усе вищезазначене має обов'язково підкріплюватися надійною технологічною базою, що гарантує інтеграцію кросплатформеності. Враховуючи динамічний ритм життя цільової аудиторії, архітектура системи повинна забезпечувати безперервний доступ до списку справ з різних пристроїв та мати стабільну синхронізацію даних. Це гарантує цілісність інформації та повну незалежність користувача від конкретного апаратного забезпечення чи локального робочого місця.

1.3. Огляд та аналіз існуючих підходів та програмних засобів

Аналіз ринку програмного забезпечення показує, що наявні рішення можна розділити на дві великі категорії, кожна з яких має суттєві недоліки для обраної цільової аудиторії.

Категорія 1: Корпоративні Agile-системи (Jira, Azure DevOps, Asana). Дані системи де-факто є стандартом для управління командами у великих підприємствах. Вони підтримують методології Scrum, Kanban, мають потужну аналітику та можливості інтеграції. Перш за все, ці системи є надмірно перевантаженими. Поріг входження є дуже високим. Індивідуальний користувач витрачає непропорційно багато часу на налаштування робочих просторів, адміністрування статусів та переходів, замість того, щоб безпосередньо виконувати завдання. Крім того, більшість з них є важкими веб-додатками, що споживають значні ресурси комп'ютера.

Категорія 2: Прості менеджери завдань (Todoist, Microsoft To Do, Apple Reminders). Ці додатки орієнтовані на широкого споживача. Їхньою перевагою є мінімалістичний дизайн та висока швидкість роботи. Недоліком для професійного використання у таких системах є відсутність підтримки будь-яких методологій менеджменту, зокрема Agile.

У наукових статтях, присвячених особистій ефективності, наприклад, у працях з адаптації методології Scrum для індивідуального використання - «Personal Scrum», зазначається, що поєднання списку завдань із чіткими часовими рамками дає найкращий результат [3]. Проте жоден із популярних мас-маркет додатків не реалізує цю парадигму на належному рівні.

1.4 Аналіз програмного забезпечення для менеджменту часу

Для формування чітких вимог до функціоналу, було проведено дослідження сучасного ринку інструментальних засобів тайм-менеджменту. Ринок програмного забезпечення для менеджменту часу надзвичайно насичений, проте більшість існуючих рішень концептуально орієнтовані або на колективну роботу у великих корпораціях, або на примітивне побутове використання. Метою аналізу є оцінка того, наскільки наявні на ринку інструменти здатні задовольнити специфічні потреби індивідуального користувача в ітеративному плануванні. Оцінювання кожного інструмента проводилося за такими ключовими споживчими критеріями, як підтримка концепції agile, наявність автоматизованих інструментів збору статистики щодо власної продуктивності та успішності дотримання дедлайнів, рівень когнітивного навантаження на користувача та загальна зручність використання програми у щоденному робочому процесі.

1.4.1. Atlassian Jira

Програмний комплекс Jira від компанії Atlassian є світовим стандартом для управління проєктами за методологіями Agile, Scrum та Kanban [4]. Цей продукт створений для забезпечення прозорості робочого процесу у великих командах. З

точки зору функціоналу, Jira пропонує значний набір інструментів: користувач може створювати масштабні цілі - епіки, розбивати їх на менші цілі – історії, та конкретні задачі, формувати беклог і планувати спринти з чіткими часовими рамками.

Головною і найсильнішою стороною Jira є її модуль звітності. Програма автоматично генерує графіки продуктивності команди (наприклад, Burndown charts), що дозволяє візуально оцінити, чи встигають розробники закрити всі задачі до кінця спринту. Система жорстко контролює дедлайни та дозволяє налаштувати складні фільтри для пошуку прострочених завдань.

Однак, ключова перевага Jira стає її найголовнішим недоліком при спробі індивідуального використання. Інтерфейс програми є надзвичайно перевантаженим. Для простого користувача створення однієї задачі перетворюється на «похід до продуктового»: необхідно заповнити безліч полів (тип, пріоритет, виконавець, оцінка складності, компоненти). Крім того, переміщення задачі по етапах виконання вимагає постійної зміни її статусів вручну. Усе це створює колосальне когнітивне навантаження - замість того, щоб просто виконувати роботу, індивідуальний користувач змушений постійно «звітувати» перед системою, витрачаючи дорогоцінний час на адміністрування карток.

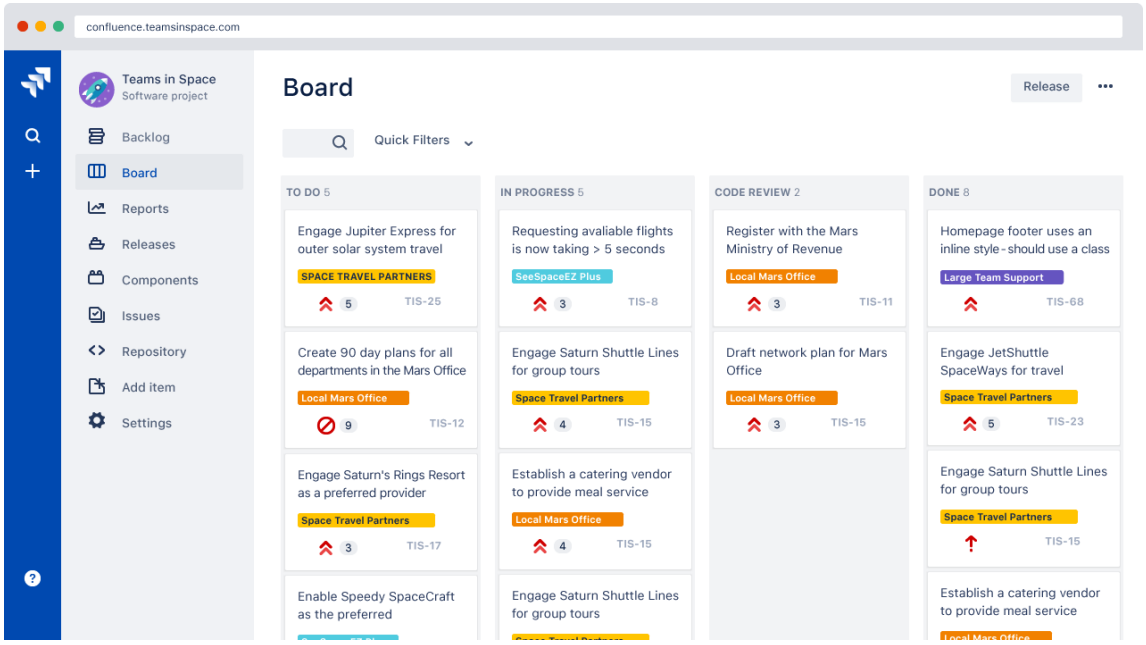


Рис.1.1 Екранні форми застосунку Jira

1.4.2 Todoist

Todoist - це один із найпопулярніших споживчих сервісів для управління особистими завданнями, який орієнтований на філософію максимального розвантаження пам’яті, запропоновану Девідом Алленом [5]. Програма ідеально підходить для фіксації щоденних справ, структурування їх за папками та швидкого встановлення кінцевих термінів.

Todoist вирізняється зразковим користувацьким досвідом[6]. Інтерфейс є мінімалістичним, інтуїтивно зрозумілим і працює миттєво. Найбільш зручною функцією є розпізнавання природної мови. Користувачу достатньо надрукувати «здати проєкт у п'ятницю о 18:00», і програма сама зрозуміє текст та встановить відповідний дедлайн. Це зводить до абсолютного мінімуму час на створення завдання.

З погляду серйозного професійного планування Todoist має суттєві прогалини. По-перше, користувач не може згрупувати завдання у спринти - робочий процес виглядає як нескінченний лінійний список, що психологічно тисне на споживача. По-друге, програма вкрай слабо працює зі статистикою порушення дедлайнів. Якщо завдання прострочено, воно просто підсвічується червоним кольором. Користувач не може в кінці місяця відкрити звіт і побачити, що він, наприклад, "успішно завершив 80% завдань, а 20% виконав із запізненням". Відсутність ретроспективної аналітики не дозволяє користувачу робити висновки щодо власної продуктивності та покращувати навички планування.

Система дозволяє встановлювати дедлайни для кожної картки, додавати вкладення та чек-лісти.

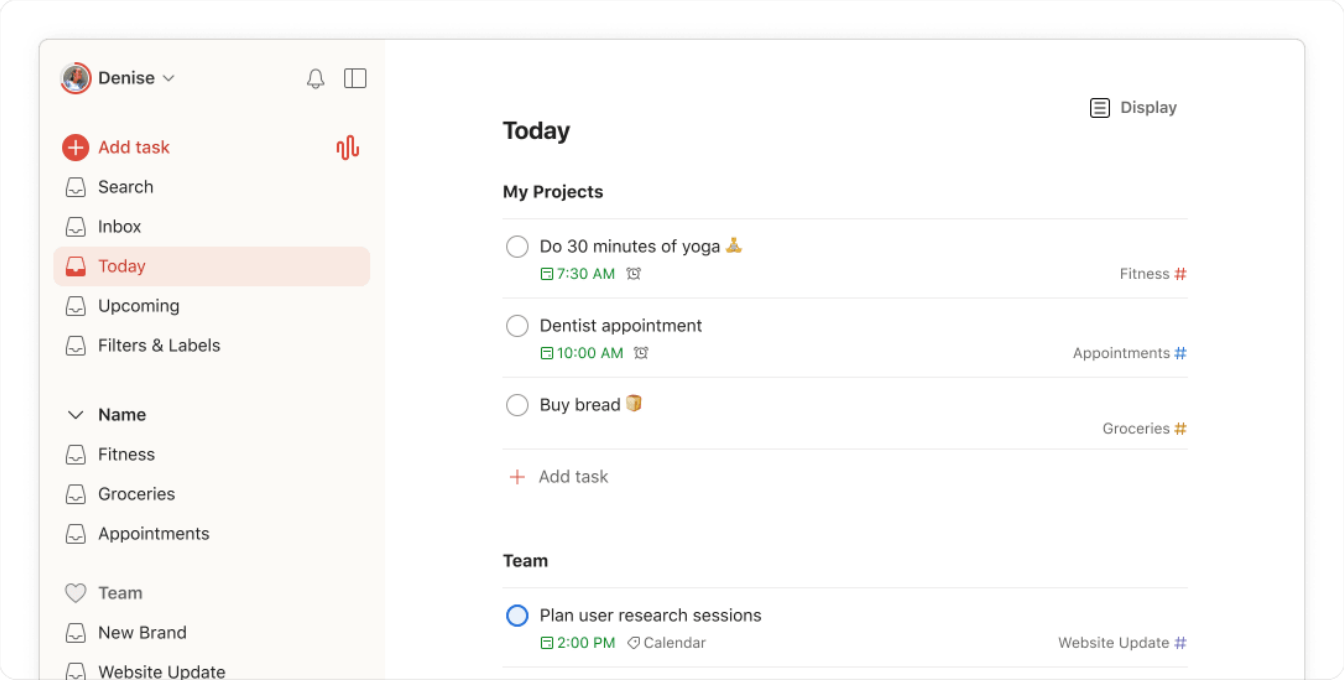


Рис.1.2 Екранні форми застосунку Todoist

1.4.3 Trello

Trello - це інструмент управління завданнями, який популяризував серед широкого кола користувачів парадигму канбан-дощок [7].

Споживачі обирають Trello завдяки його наочності. Процес планування нагадує наклеювання стікерів на коркову дошку. Це дуже зручно для відстеження задач, які проходять послідовні стадії виконання (наприклад: "Ідея", "В роботі", "На перевірці", "Готово"). Дошки є дуже гнучкими: користувач сам визначає, як називатимуться колонки та за якими правилами переміщуватимуться картки.

Головною проблемою Trello є швидка втрата контролю над інформацією при зростанні масштабів проєкту. Коли карток стає занадто багато, дошка перетворюється на візуальний хаос. Trello концептуально не підтримує планування спринтами (користувачі змушені імітувати це, створюючи колонки з назвами "Спринт 1", "Спринт 2", що є незручним). Також програма не надає зведеної автоматичної статистики: користувачу важко швидко оцінити загальний обсяг прострочених завдань по всіх дошках одночасно без використання платних розширень.

Рис.1.3 Екранні форми застосунку Trello

1.4.4 Toggl Track

Toggl Track представляє категорію програмного забезпечення, що фокусується виключно на відстеженні активності користувача [8]. Він популярний серед фахівців, які працюють з погодинною оплатою. Користувач створює сутність активності, прив'язує її до певного проєкту і фіксує проміжки часу, витрачені на її виконання.

Інтерфейс надання статистики в Toggl є одним із найкращих на ринку. Користувач отримує повні, деталізовані та візуально привабливі звіти про свою продуктивність. Можна чітко побачити, на які проєкти пішло найбільше зусиль протягом тижня.

Toggl є інструментом реактивним, а не проактивним. У ньому принципово неможливо скласти план на тиждень, створити спринт чи встановити дедлайн. Через це користувач змушений вести подвійний облік: планувати завдання, наприклад, у Todoist, а результати фіксувати в Toggl. Окрім того, Toggl вимагає постійного мікроменеджменту - необхідність вручну запускати та зупиняти лічильники, вписувати періоди активності постфактум, що психологічно виснажує і відволікає від основної роботи.

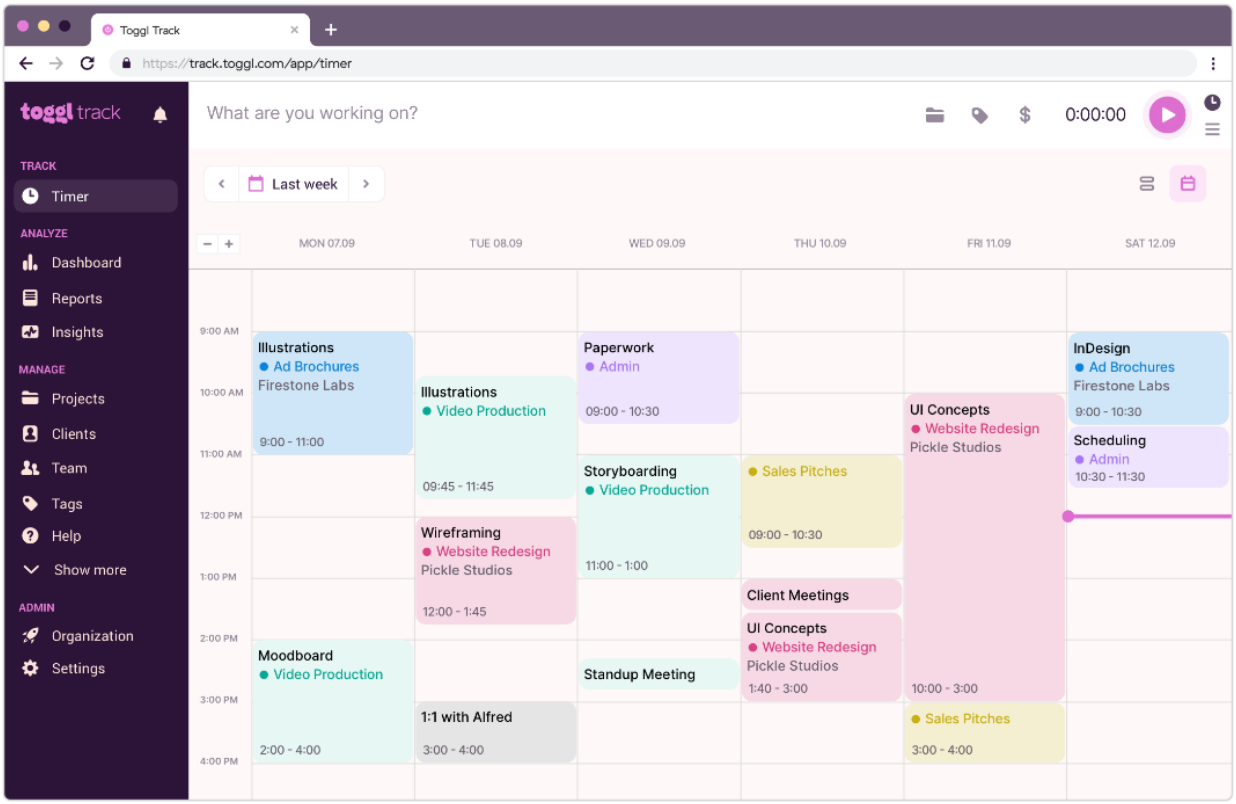


Рис.1.4 екранні форми застосунку Toogl Track

1.4.5 Microsoft To Do

Microsoft To Do - це сервіс для ведення переліку справ, який є частиною екосистеми Microsoft та встановлюється за замовчуванням на багатьох пристроях. Програма орієнтована на максимально просте, побутове використання - складання списків покупок, фіксацію дрібних доручень або створення плану на поточний день. Повна безкоштовність, відсутність реклами та глибока інтеграція з сервісами Microsoft робить To Do приємним доповнення до інших програмних рішень.

Головною фічею для користувача є екран «Мій день» - чистий аркуш, який щоранку оновлюється, дозволяючи користувачеві самостійно вибрати із загального пулу завдань ті, на яких він хоче сфокусуватися саме сьогодні. Але, з іншого боку, функціонал програми є занадто примітивним для потреб професійного планування. Тут неможливо реалізувати Agile методології. Інтерфейс не дозволяє побачити картину проєкту в цілому. Статистика продуктивності чи дотримання дедлайнів відсутня повністю: щойно користувач ставить галочку «Виконано», завдання

зникає в загальному архіві, і проаналізувати його в кінці місяця для оцінки власної ефективності практично неможливо.

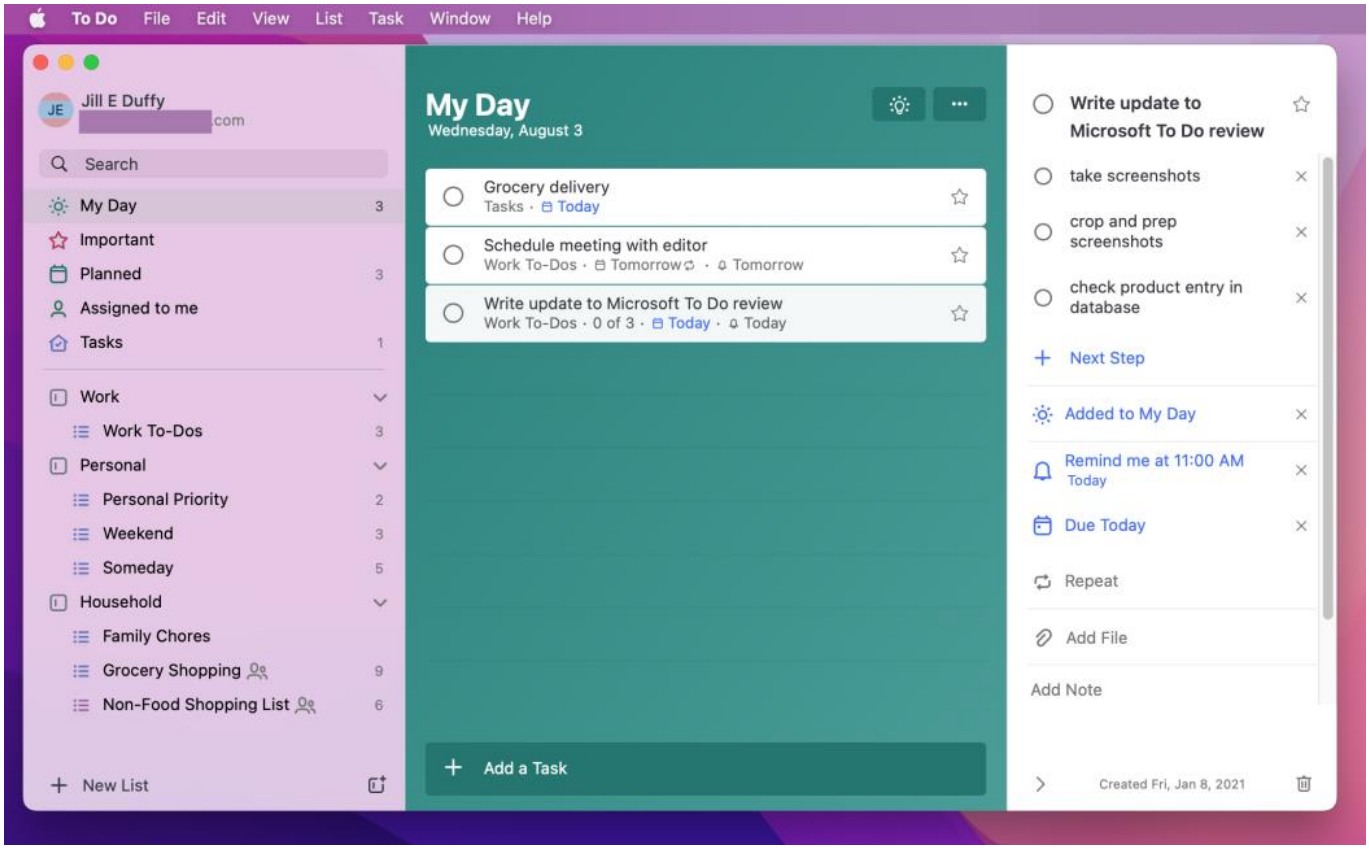


Рис.1.5 екранні форми застосунку Microsoft To Do

Для систематизації результатів дослідження та обґрунтування вимог до розроблюваного застосунку було сформовано систему критеріїв. Ці критерії відображають ключові потреби сучасного споживача: можливість розбивати велику роботу на етапи, автоматично отримувати аналітику своєї продуктивності, відстежувати порушення дедлайнів та користуватися програмою без зайвого навантаження. Результати порівняльного аналізу зведено у таблицю 1.1. Для демонстрації конкурентних переваг у таблиці також наведено прогнозовані характеристики майбутнього проєкту.

Таблиця 1.1

Результати порівняльного аналізу

Критерій оцінювання	Jira	Todoist	Trello	Toggl	Microsoft To Do	«DeepLone»
Можливість планувати роботу спринтами	+	-	-	-	-	+
Автоматична статистика закриття завдань	+	-	-	+	-	+
Відстеження та аналіз прострочених дедлайнів	+	+	+	-	+	+
Відсутність ручного мікроменеджменту	-	+	+	-	+	+
Кроки до створення карток	5+	2	3	4	2	2

1.5. Специфіка архітектурної побудови та синхронізації даних у системах планування

Важливим аспектом є вивчення способів організації взаємодії між користувачем та даними. У сучасних умовах, коли робочий процес фахівця може відбуватися на різних пристроях, критичного значення набуває модель збереження та синхронізації інформації.

Можна виділити два основні підходи до побудови таких систем:

Локальні (автономні) системи: забезпечують високу швидкість роботи та конфіденційність, оскільки дані не залишають пристрою користувача. Проте вони мають суттєве обмеження - відсутність доступу до списку завдань та журналів часу з інших платформ, що є неприпустимим для динамічного робочого середовища.

Клієнт-серверні системи з хмарною синхронізацією: дозволяють зберігати

централізовану базу даних на сервері. Це забезпечує цілісність інформації та можливість доступу до актуального стану справ з будь-якої точки світу.

Окрему увагу слід приділити питанню використання веб-технологій. Це дозволяє швидко розробляти кросплатформенні рішення, проте вони часто мають надмірні вимоги до апаратних ресурсів. Десктопні додатки, навпаки, забезпечують кращий відгук інтерфейсу та швидкодію. Таким чином, при виборі моделі реалізації пріоритет повинен надаватися технологіям, що поєднують кросплатформеність із продуктивністю та надійними механізмами API-взаємодії.

1.6. Перспективи та напрямки вдосконалення систем персонального менеджменту

Стрімкий розвиток технологій диктує нові вимоги до архітектури та функціонального наповнення систем персонального тайм-менеджменту. Сучасні тенденції свідчать про те, що еволюція таких програмних продуктів рухається від створення пасивних інструментів фіксації даних до розробки проактивних цифрових асистентів. Одним із найперспективніших напрямків удосконалення є інтелектуалізація систем за рахунок впровадження алгоритмів машинного навчання та штучного інтелекту. Це відкриває можливості для переходу до предиктивної аналітики, коли застосунок здатний не лише збирати історичні дані про виконання завдань, але й прогнозувати ймовірність зриву дедлайнів на основі попереднього досвіду користувача [9]. Алгоритмічний аналіз індивідуального темпу роботи дозволить системі автоматично генерувати рекомендації щодо оптимального розподілу навантаження, попереджаючи емоційне вигорання та перевтому ще до їх фактичного настання.

Іншим важливим вектором розвитку є глибока інтеграція систем персонального планування із зовнішніми цифровими екосистемами. Оскільки цільовою аудиторією виступають IT-спеціалісти, розробники та студенти технічних спеціальностей, ізольованість менеджера завдань стає його критичним недоліком. Перспективним вбачається створення безшовних інтеграцій із

середовищами розробки, системами контролю версій та комунікаційними платформами. Це дозволить автоматизувати процес зміни статусів завдань: наприклад, коміт у репозиторії може автоматично переводити відповідну задачу у статус виконаної на дошці спринту. Така синергія інструментів кардинально знизить рутину, мінімізує кількість "ручних" маніпуляцій з інтерфейсом та дозволяє користувачеві залишатися у стані потоку, не відволікаючись на мікроменеджмент [10].

Окремої уваги заслуговує вдосконалення архітектурних та інтерфейсних рішень. З точки зору UX/UI, майбутнє належить просторовим інтерфейсам, де візуалізація завдань відбувається на основі їх семантичної ваги та терміновості. Застосування гнучких механізмів, таких як Drag-and-Drop з миттєвим перерахунком параметрів бази даних, є лише першим кроком до створення повністю імерсивного робочого середовища.

Підсумовуючи, можна стверджувати, що подальший розвиток галузі лежить у площині зменшення когнітивного навантаження на користувача. Системи персонального менеджменту нового покоління повинні брати на себе всю математичну, аналітичну та організаційну роботу, залишаючи людині виключно функцію прийняття стратегічних рішень та безпосереднього виконання творчих або інженерних задач. Розробка рішень, які відповідають цим критеріям, є не лише технічним викликом, але й важливою соціально-економічною потребою сучасного та майбутнього суспільства.

2 ОГЛЯД ЗАСОБІВ РЕАЛІЗАЦІЇ ДОДАТКУ ДЛЯ МЕНЕДЖМЕНТУ ЧАСУ

2.1. Архітектурні підходи до розробки сучасного програмного забезпечення

Сучасна індустрія розробки програмного забезпечення пропонує декілька фундаментальних архітектурних підходів для створення користувацьких застосунків. За способом організації зберігання та обробки даних виділяють локальні та клієнт-серверні системи. Локальні додатки передбачають збереження всієї бізнес-логіки та бази даних безпосередньо на фізичному пристрої користувача. Хоча такий підхід забезпечує незалежність від інтернет-з'єднання та високу конфіденційність, він має суттєві недоліки, зокрема неможливість доступу до власних завдань з інших пристроїв та високий ризик незворотної втрати даних у разі апаратного збою комп'ютера. Натомість у клієнт-серверних системах дані зберігаються в централізованій базі на сервері, а клієнтський додаток виступає лише засобом їх відображення та відправляє запити на зміну через API. Це гарантує структурну цілісність інформації та її безперервну синхронізацію між усіма пристроями користувача.

Важливим аспектом проєктування є також вибір технології реалізації клієнтської частини. Класичні веб-додатки працюють виключно у браузері і не потребують встановлення, але жорстко залежать від середовища виконання і не можуть повноцінно використовувати апаратні потужності комп'ютера. Альтернативним популярним рішенням є гібридні десктопні додатки. По суті, вони є веб-сайтами, запакованими у власний вбудований браузер, що працює як окрема програма. Головним архітектурним недоліком таких гібридних рішень є надмірне споживання оперативної пам'яті, що робить їх занадто «важкими» для системи та призводить до зниження загальної продуктивності.

Противагою цьому виступають сучасні кросплатформені додатки, які компілюються безпосередньо в машинний код під конкретну операційну систему.

Їхньою беззаперечною перевагою є максимальна швидкість роботи, плавність інтерфейсу та економне використання системних ресурсів. З огляду на наведені характеристики та для успішної реалізації поставлених завдань, розроблювана система проєктується на основі клієнт-серверної архітектури з використанням десктопного клієнта. Такий підхід вимагає залучення комплексу інструментальних засобів.

2.2. Засоби реалізації серверної логіки

Ефективна реалізація серверної логіки вимагає ретельного підбору технологічного стека, який здатен забезпечити високу продуктивність, безпеку даних та надійну кросплатформену взаємодію з клієнтським застосунком. Оскільки архітектура проєкту передбачає інтенсивний обмін даними (маніпуляції з об'єктами завдань, синхронізація спринтів, динамічне обчислення статистичних метрик), для розробки бекенду було обрано екосистему технологій від компанії Microsoft. Такий вибір зумовлений високою надійністю компонентів, їхньою глибокою інтеграцією між собою та можливістю використання уніфікованого підходу до написання коду.

2.2.1. JetBrains Rider

Як основний інструментарій для написання, тестування та дебагінгу програмного коду було обрано середовище JetBrains Rider. У ньому наявний потужний вбудований рушій статичного аналізу коду на базі технологій ReSharper, який забезпечує автоматичне виявлення синтаксичних та архітектурних вразливостей, а також пропонує інтелектуальні механізми рефакторингу в режимі реального часу. Крім того, Rider гарантує безшовну інтеграцію з інструментарієм .NET 9 на операційних системах сімейства macOS. Це має критичне значення для проєкту, оскільки дозволяє в єдиному ізольованому робочому просторі здійснювати розробку, профілювання продуктивності та дебагінг як бекенд-частини, так і клієнтського застосунку [11].

2.2.2 C#

C# виступає базовою мовою для розробки як серверної, так і клієнтської частин системи. Це сучасна об’єктно-орієнтована, строго типізована мова програмування, яка забезпечує високий рівень безпеки типів та суттєво знижує ймовірність виникнення критичних помилок ще на етапі компіляції [12]. Використання C# як єдиної мови дозволяє значно оптимізувати процес створення продукту: це дає змогу перевикористовувати спільні моделі даних (Data Transfer Objects) між клієнтом і сервером та уникнути необхідності перемикання когнітивного контексту розробника.

2.2.3 .NET 9

.NET 9 використовується як інфраструктурний фундамент для бекенду. Це найактуальніша стабільна версія кросплатформеного середовища виконання від Microsoft. Її ключові архітектурні вдосконалення включають глибоку оптимізацію використання оперативної пам'яті та підвищення загальної продуктивності обчислень. Ці оптимізаційні характеристики є критично важливими для швидкої та безперебійної обробки великої кількості паралельних запитів від клієнтських пристроїв без надмірного навантаження на апаратну частину сервера [13].

2.2.4. ASP.NET Core

ASP.NET Core застосовується для безпосередньої побудови програмного інтерфейсу системи. Цей інструмент дозволяє створювати надійну архітектуру Stateless API, що робить серверну частину легко масштабованою [14]. До головних особливостей фреймворку належить наявність вбудованих механізмів впровадження залежностей та інструментів маршрутизації. Завдяки цим засобам реалізується система контролерів, які відповідають за обробку ключової бізнес-логіки: управління життєвим циклом спринтів, збереження просторових координат завдань та формування масивів статистичних даних.

2.3. Інструменти проєктування бази даних та ORM

Збереження, структурування та швидка обробка інформації є фундаментальною складовою будь-якої інфокомунікаційної системи. Логіка цього проєкту побудована на чітких ієрархічних зв'язках між сутностями (наприклад, один користувач володіє багатьма категоріями та спринтами, а кожен спринт містить множину завдань), оптимальним рішенням є використання класичної реляційної моделі даних.

2.3.1. PostgreSQL

Як фізичне сховище інформації було обрано PostgreSQL - одну з найбільш надійних, масштабованих та потужних об'єктно-реляційних СУБД з відкритим кодом. Її вибір обґрунтовується повною гарантією транзакційної цілісності та суворою відповідністю стандартам ACID (атомарність, узгодженість, ізольованість, довговічність). Крім того, архітектура PostgreSQL оптимізована для ефективної обробки складних агрегаційних запитів, що є критично необхідним для безперебійної генерації зведеної статистики продуктивності користувача в режимі реального часу [15].

2.3.2. Entity Framework Core

З метою абстрагування бізнес-логіки від специфіки написання низькорівневих SQL-запитів та прискорення загального процесу розробки, застосовується сучасний ORM-фреймворк Entity Framework Core. Цей інструмент дозволяє розробнику взаємодіяти з базою даних не через мову запитів, а за допомогою строгих C#-класів, автоматично транслюючи об'єктно-орієнтований код в оптимізований SQL [16]. Це не лише підвищує читабельність коду, але й унеможливлює виникнення вразливостей типу SQL-ін'єкцій.

2.3.3. Принцип Code First

Під час роботи з базою даних у межах проєкту застосовується підхід Code First. Згідно з цією парадигмою, спочатку розробляється абстрактна доменна

модель безпосередньо у коді застосунку, після чого інструментарій EF Core автоматично генерує програмні міграції. Ці міграції самостійно конвертують зміни у C#-класах у відповідні SQL-команди для створення або оновлення структури таблиць. Такий підхід гарантує абсолютну синхронізацію між програмним кодом бекенду та фізичною схемою бази даних, а також дозволяє зручно застосовувати системи контролю версій для відстеження еволюції структури БД.

2.4. Технології реалізації клієнтського інтерфейсу

Як показав аналіз сучасних архітектурних підходів, використання гібридних веб-технологій для розробки десктопних застосунків неминуче призводить до проблеми надмірного споживання оперативної пам'яті та процесорного часу. Враховуючи вимоги до ресурсоефективності та необхідність забезпечення максимально плавного користувацького досвіду, для розробки клієнтської частини системи було обрано нативний підхід.

2.4.1. .NET MAUI

.NET Multi-platform App UI виступає базовим фреймворком для створення клієнтського застосунку. Ця технологія дозволяє розробляти кросплатформені додатки, маючи єдину уніфіковану кодову базу на мові C# [17]. Фундаментальною відмінною рисою фреймворку є те, що компоненти інтерфейсу не малюються поверх веб-рушія, а компілюються у специфічні для кожної операційної системи нативні елементи керування (наприклад, з використанням технології Mac Catalyst для середовища macOS). Завдяки цьому додаток має миттєвий відгук інтерфейсу, органічно виглядає в системі та не перевантажує апаратні потужності комп'ютера користувача

2.4.2. XAML

Застосовується для безпосереднього проєктування візуальної складової системи. Використання цієї мови дозволяє описувати ієрархію візуальних

компонентів декларативним шляхом, забезпечуючи суворе та чітке відокремлення графічного інтерфейсу від базової бізнес-логіки застосунку [18]. Це суттєво спрощує підтримку коду та процес створення складних інтерактивних зон, таких як панелі завдань із підтримкою вільного переміщення об'єктів «Drag-and-Drop».

2.4.3. Архітектурний паттерн MVVM

Model-View-ViewModel використовується для забезпечення надійної, модульної структури клієнтського коду та керування станом програми. Цей патерн реалізується за допомогою системи двосторонніх прив'язок даних. Завдяки цьому механізму система дозволяє автоматично оновлювати візуальні компоненти на екрані (наприклад, списки активних завдань, кошик, або статистичні графіки) безпосередньо у момент зміни відповідних даних у системі. Це повністю виключає необхідність прямого маніпулювання елементами інтерфейсу в коді, що робить архітектуру застосунку більш безпечною та стійкою до помилок [19].

2.5. Спеціалізовані бібліотеки та засоби безпеки

Оскільки розроблювана система передбачає багатокористувацький доступ через глобальну мережу Інтернет, критичним аспектом архітектурного проєктування є забезпечення надійного захисту персональних даних та безпеки інформаційного обміну між клієнтом і сервером.

2.5.1. JSON Web Token

JWT використовується як базовий механізм автентифікації та перевірки цілісності даних. Відповідно до цього стандарту [20], після успішної перевірки облікових даних (наприклад, логіну та хешованого паролю) сервер генерує криптографічно підписаний токен. Клієнтський застосунок зберігає цей маркер і автоматично додає його у заголовки (HTTP Headers) кожного наступного запиту до сервера. Такий підхід ідеально відповідає концепції архітектури без збереження стану (Stateless API), оскільки серверу не потрібно утримувати сесію в оперативній

пам'яті - уся необхідна інформація для верифікації прав доступу вже міститься всередині самого токена. Це усуває необхідність передавати пароль при кожній дії та надійно захищає систему.

2.5.2. OAUTH2

Як додатковий рівень безпеки та засіб для майбутнього масштабування, архітектура системи орієнтована на підтримку протоколу OAuth 2.0. Це відкритий галузевий фреймворк, який дозволяє користувачам надавати стороннім застосункам обмежений доступ до своїх захищених ресурсів без необхідності розкриття власних облікових даних. Впровадження цього стандарту є особливо актуальним з огляду на необхідність інтеграції системи із зовнішніми цифровими екосистемами (наприклад, реалізація єдиного входу Single Sign-On через Google або GitHub). Протокол OAuth 2.0 органічно доповнює JWT, виступаючи надійним транспортним каркасом для безпечної видачі та оновлення маркерів доступу (Access Tokens) і маркерів оновлення (Refresh Tokens)[21].

3 ПРОЄКТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ТАЙМ-МЕНЕДЖМЕНТУ

3.1 Вимоги до системи

Аналіз особливостей процесу персонального тайм-менеджменту та існуючих засобів для ведення списків справ дозволяє сформулювати вимоги до майбутнього програмного продукту. Програма для менеджменту часу та продуктивності повинна бути реалізована на основі клієнт-серверної архітектури та забезпечувати наступні ключові функції.

3.1.1. Функціональні вимоги

1) Створення та управління Спринтами (встановлення назви, дати початку та дати завершення робочого циклу).

2) внесення в систему даних про нові завдання з обов'язковою можливістю встановлення чітких дедлайнів.

3) можливість прив'язки створених завдань до конкретного Спринту.

4) засоби управління життєвим циклом завдання (редагування опису, зміна статусів виконання, видалення).

5) відстеження порушення дедлайнів для кожного активного завдання.

6) можливість розрахунку підсумкових даних щодо продуктивності користувача за обраний період (співвідношення успішно закритих та прострочених завдань).

7) можливість представлення зібраної статистичної інформації у вигляді наочних графіків та діаграм.

3.1.2. Нефункціональні вимоги

1) Доступ до REST API здійснюється виключно за валідними JWT-токенами. Паролі зберігаються у вигляді хешів, згенерованих за алгоритмом PBKDF2 (210 000 ітерацій) із додаванням унікальної 16-байтової солі для захисту від брутфорсу.

2) Токени пристроїв для Push-сповіщень повинні реєструватися та зберігатися у безпечному середовищі на стороні сервера.

3) Відправка Push-сповіщень має відбуватися асинхронно у фоновому режимі за допомогою воркера.

4) Серверна інфраструктура розгортається у середовищі Docker, для оптимізації образу API багатоетапна перевірка, а для бази даних налаштування механізму healthcheck для запобігання критичних помилок під час запуску сервісів.

5) Рабезпечення безперервної синхронізації даних між клієнтським додатком та централізованою базою даних PostgreSQL через REST API.

6) Проєкт повинен мати чітку модульну структуру: логічне розділення на клієнтську та серверну частини.

7) Для взаємодії з базою даних має використовуватись сучасний ORM (Entity Framework Core), що спростить підтримку та міграції схеми бази даних.

3.2. Сценарії взаємодії

Функціональні можливості розробленої системи управління завданнями найповніше розкриваються через опис логіки взаємодії користувача з програмним комплексом. Ця взаємодія охоплює кілька ключових бізнес-процесів, які забезпечують безперервний цикл персонального тайм-менеджменту.

Першим і фундаментальним етапом роботи є процес ідентифікації та входу в систему. Відкриваючи десктопний застосунок, користувач обирає між класичною реєстрацією (за допомогою електронної пошти та пароля) та швидкою авторизацією через обліковий запис Google. У разі використання стороннього сервісу застосунок перенаправляє користувача на безпечну сторінку підтвердження. Після успішної перевірки особи серверна частина створює

захищену сесію. Це дозволяє користувачеві безперешкодно та безпечно працювати у своєму головному робочому просторі (дашборді) без необхідності повторного введення облікових даних під час наступних сеансів.

Отримавши доступ до системи, користувач переходить до безпосереднього управління робочим навантаженням. Через зручну форму створюються нові завдання: користувач вказує назву, розгорнутий опис, обирає категорію, встановлює рівень пріоритетності та чіткі часові межі (дедлайни). Після збереження система автоматично фіксує ці дані. Створене завдання з'являється у загальному робочому списку зі стартовим статусом, який у процесі виконання може динамічно змінюватися: від початкового етапу очікування до активної роботи та фінального завершення.

Для забезпечення гнучкості та структурованості робочого процесу система пропонує механізм ітеративного планування. Користувач має можливість формувати робочі цикли - спринти, задаючи для них унікальні назви та часові межі від старту до завершення. Після створення спринту користувач виокремлює необхідні завдання із дашборду та призначає їх на цей робочий період. Програма автоматично формує ізольовану дошку, на якій відображаються виключно ті завдання, що підлягають виконанню в межах поточної ітерації, дозволяючи сфокусувати увагу на головних цілях.

Завершальним етапом взаємодії є аналіз особистої продуктивності, який базується на історії виконання завдань. При переході до аналітичного розділу система автоматично збирає та обробляє накопичену статистику. Серверна частина підраховує загальну кількість виконаних завдань, оцінює ефективність закриття спринтів та аналізує дотримання дедлайнів. Опрацьована інформація передається у клієнтський застосунок, де перетворюється на наочні візуальні елементи - графіки, діаграми та зведені показники. Це забезпечує користувача зручним інструментарієм для об'єктивної оцінки власної ефективності та планування подальшого навантаження.

Також користувач може налаштовувати свій дашборд під себе – картки завдань розміщуються користувачем за своїм бажанням, також в нього є можливість змінити кольорову тему застосунку та скинути положення карток.

Діаграму прецедентів наведено на рисунку 3.1.

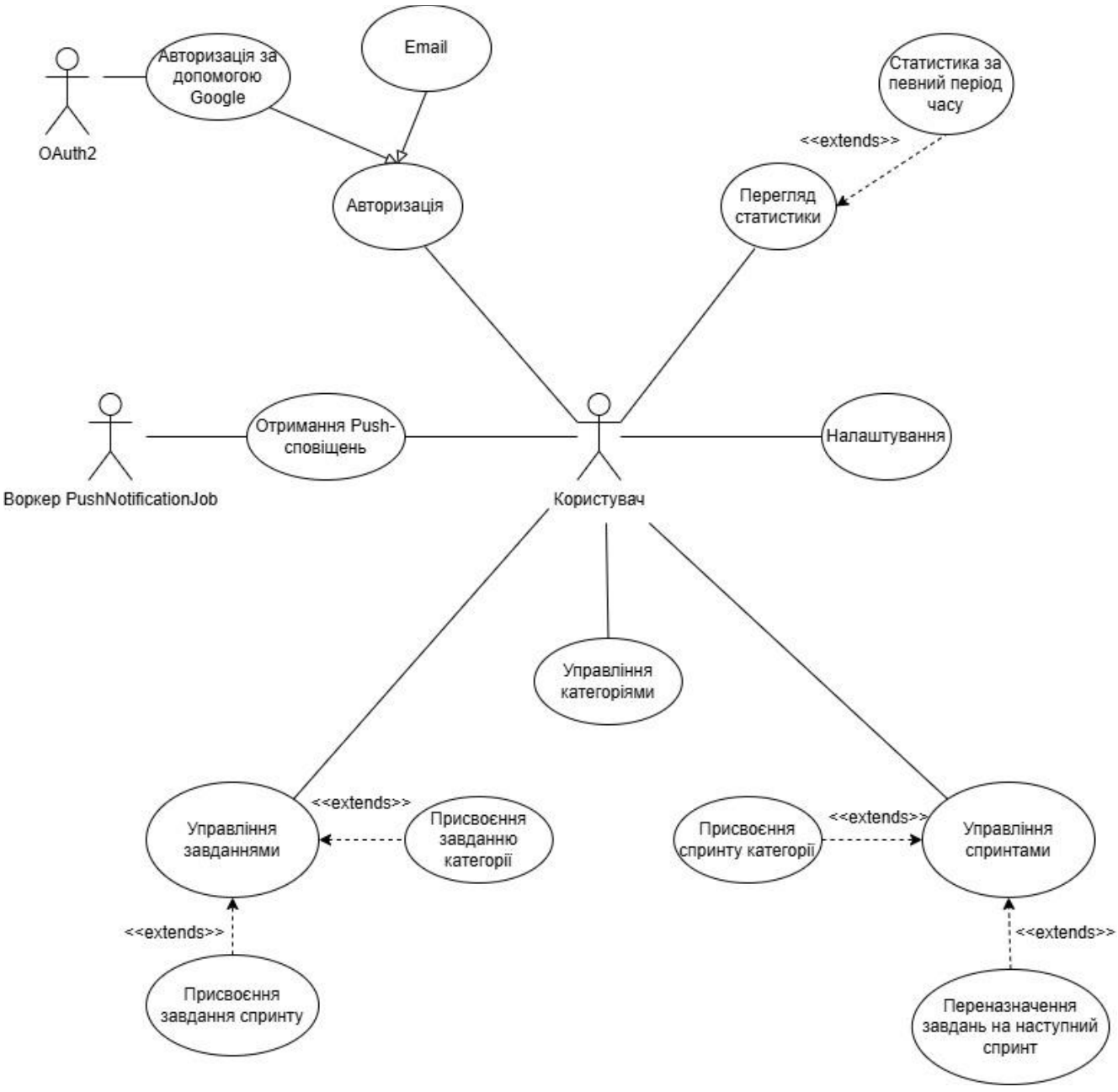


Рис. 3.1. Діаграма прецедентів

3.3 Проєктування архітектури

Проєкт реалізує класичну трирівневу клієнт-серверну парадигму. Такий архітектурний підхід обрано з метою забезпечення максимальної масштабованості та чіткого розмежування зон відповідальності між підсистемами збереження інформації, обробки бізнес-правил та візуалізації даних.

Центральним вузлом системи (серверним рівнем) виступає REST API додаток, розроблений на базі фреймворку ASP.NET Core під управлінням платформи .NET 9. Цей компонент бере на себе виконання всієї бізнес-логіки: від криптографічної перевірки ідентичності користувачів за допомогою JWT-маркерів до організації життєвого циклу спринтів та генерації аналітичних звітів. Окрім обробки синхронних запитів через систему контролерів, серверна частина містить інтегровані фонові служби для асинхронної взаємодії з нативними провайдерами Push-сповіщень, зокрема APNs для екосистеми Apple та WNS для середовища Windows.

За надійність збереження стану системи відповідає рівень доступу до даних, фізичною основою якого є транзакційна реляційна СУБД PostgreSQL. Для ефективної інтеграції між об'єктно-орієнтованим кодом бекенду та реляційною структурою бази застосовано технологію об'єктно-реляційного відображення Entity Framework Core. Її використання абстрагує розробника від ручного написання SQL-інструкцій, гарантуючи безпеку операцій та значно спрощуючи процес еволюції структури даних завдяки механізму автоматизованих програмних міграцій.

Безпосередня взаємодія з кінцевим користувачем здійснюється на клієнтському рівні, який представлений десктопним застосунком для ОС macOS та Windows, створеним за допомогою технології .NET MAUI. Графічний інтерфейс проєктується повністю ізольовано від внутрішньої логіки програми завдяки використанню декларативної мови розмітки XAML та дотриманню архітектурного патерну MVVM. З метою підвищення модульності та полегшення модульного

тестування, уся логіка формування мережевих запитів до віддаленого сервера інкапсульована у відокремлену спільну бібліотеку ядра (Core).

Інтеграційна взаємодія між нативним клієнтом та сервером побудована на комбінованій транспортній моделі. Базовий обмін інформацією відбувається через канали зв'язку HTTPS за стандартами REST, де об'єкти серіалізуються у легкий текстовий формат JSON для подальшого парсингу клієнтом. Водночас, для реалізації механізмів миттєвого реагування (наприклад, для відправки внутрішньосистемних сповіщень) впроваджено технологію WebSockets за допомогою бібліотеки SignalR. Вона створює безперервний дуплексний канал зв'язку, дозволяючи серверу оновлювати стан інтерфейсу застосунку в режимі реального часу, минаючи обмеження традиційної моделі «запит-відповідь».

Взаємодію між компонентами системи представлено на рисунку 3.2.

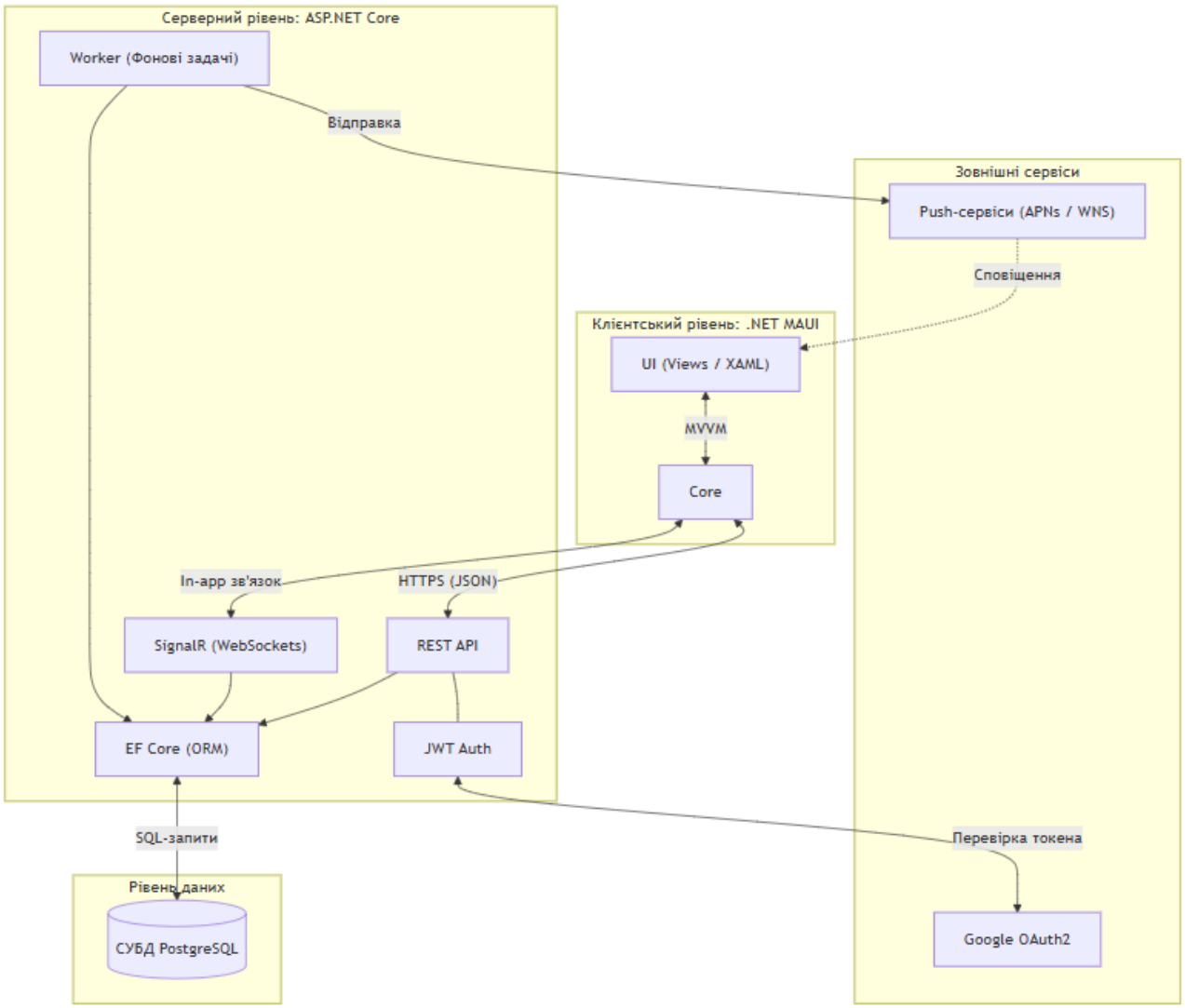


Рис.3.2. Діаграма архітектурної взаємодії

3.4. Структура класів

Проєктування предметної області розробленої системи базується на принципах об'єктно-орієнтованого підходу. Для забезпечення прозорої трансляції об'єктної моделі застосунку в реляційну структуру бази даних використовується технологія об'єктно-реляційного відображення. Усі сутності предметної області представлені класами, властивості яких безпосередньо відображаються на поля відповідних таблиць. Фундаментальним вузлом архітектури виступає сутність користувача, оскільки життєвий цикл більшості інших об'єктів ініціюється виключно в межах його контексту. Цей клас інкапсулює персональні дані та

підтримує гібридну модель безпеки: він містить як криптографічні артефакти для класичної авторизації, так і спеціальні ідентифікатори для інтеграції з зовнішніми провайдерами делегованого доступу за протоколом OAuth2.

Основою бізнес-логіки є підсистема управління робочим навантаженням, центральним елементом якої є сутність завдання. Цей клас зберігає не лише базову текстову інформацію, але й вичерпні метадані про життєвий цикл процесу: поточний статус виконання, рівень пріоритетності, а також часові позначки запланованого старту, суворих дедлайнів та фактичного завершення. Для логічної сегментації завдань реалізовано сутність категорії, яка додатково містить візуальні атрибути для ідентифікації в інтерфейсі. Організація роботи за гнучкими методологіями забезпечується класом спринту, що визначає часові межі ітерації, її глобальну мету та акумулює пул запланованих завдань.

Для взаємодії з користувачем розроблено підсистему асинхронних сповіщень. Вона оперує сутностями токену девайсу, які зберігають унікальні ідентифікатори та типи платформ клієнтських пристроїв для доставки повідомлень. Безпосередня розсилка керується відповідним класом, що функціонує як завдання для фонової черги: він містить контент повідомлення, графік відправки, лічильники спроб доставки та механізми фіксації помилок, маючи при цьому можливість посилатися на конкретні завдання чи спринти для швидкого контекстного переходу.

Паралельно з цим функціонує аналітичне ядро системи, представлене класами статистики. Вони відповідають за агрегацію показників ефективності - від підрахунку закритих завдань і витраченого часу в межах одного спринту до щоденної калькуляції комплексного індексу продуктивності користувача.

Цілісність даних на рівні реляційної бази підтримується завдяки суворому проєктуванню відношень між класами.

В архітектурі домінує зв'язок типу «композиція»: сутність користувача повністю володіє своїми завданнями, спринтами, статистикою та зареєстрованими пристроями. Відповідно, при видаленні профілю спрацьовує механізм каскадного

видалення, який автоматично очищає всі пов'язані артефакти; аналогічно завдання повністю володіє власними записами обліку часу.

Натомість для побудови гнучких зв'язків використовується «агрегація»: прив'язка завдання до певної категорії або спринту є опціональною і реалізується через ключі, що допускають нульове значення.

Це гарантує гнучкість системи, дозволяючи завданню існувати в загальному беклозі без прив'язки до ітерації, а також запобігає втраті завдань у разі видалення категорії, до якої вони належали. Діаграму класів представлено на рисунку 3.3.

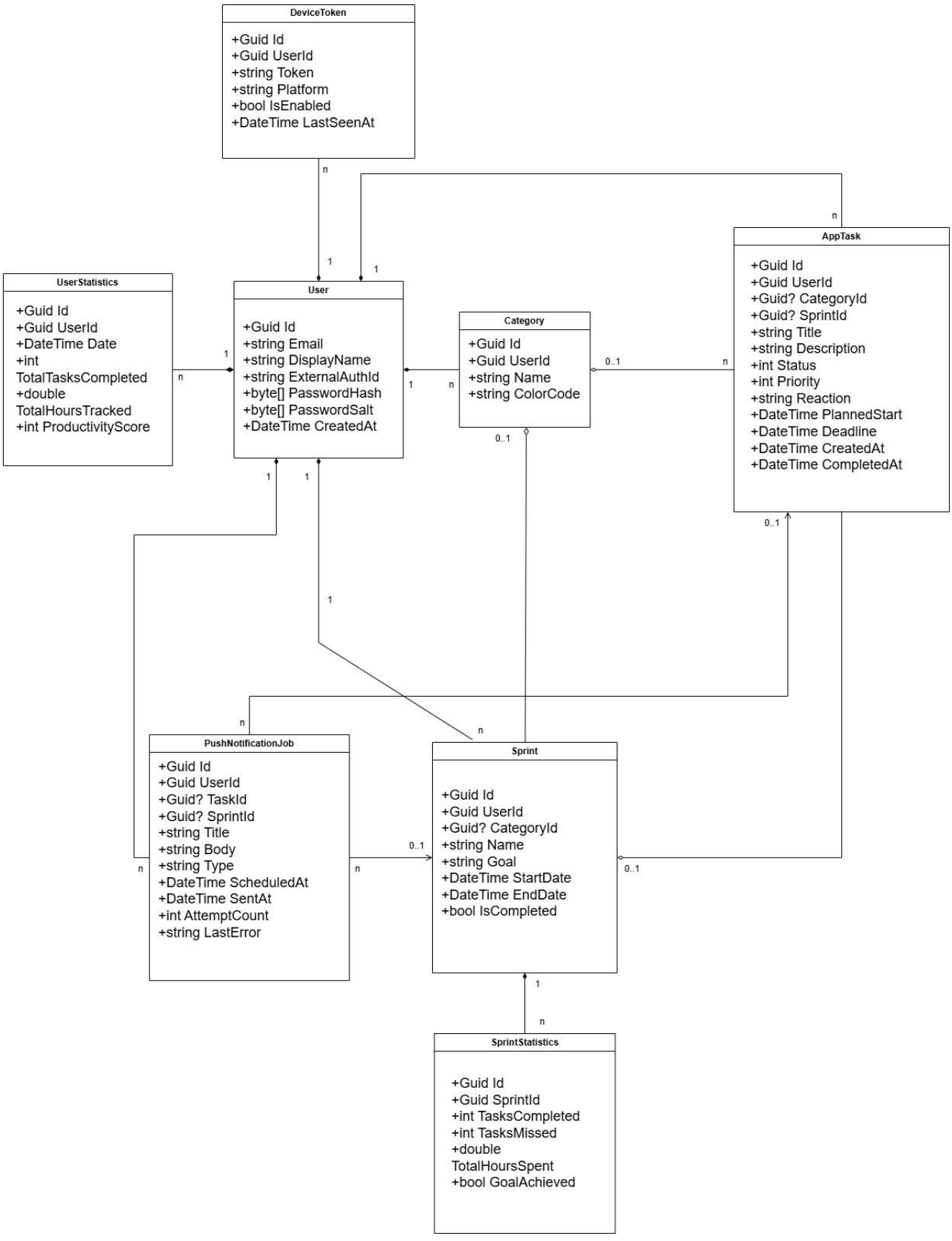


Рис. 3.3. Діаграма класів

3.5. Проєктування бази даних

Фізична реалізація сховища даних спроєктована з дотриманням принципів реляційної нормалізації та суворих вимог до ACID. Для гарантування глобальної унікальності записів та унеможливлення колізій при масштабуванні системи, як первинний ключ в усіх взаємопов'язаних таблицях використовується тип даних UUID.

Сама структура логічно декомпозована на кілька функціональних доменів, які тісно взаємодіють між собою через механізми зовнішніх ключів. База даних містить 8 таблиць.

- Таблиця Users (Користувачі) Зберігає облікові записи та дані для автентифікації.

Email (varchar): Електронна пошта користувача. Має обмеження UNIQUE INDEX для запобігання дублюванню.

DisplayName (varchar): Ім'я користувача для відображення в інтерфейсі.

ExternalAuthId (varchar): Ідентифікатор стороннього провайдера (Google OAuth2).

PasswordHash / PasswordSalt (bytea): Бінарні поля для безпечного зберігання криптографічних хешів та солі паролів.

CreatedAt (timestamp): Час створення профілю.

- Таблиця Tasks (Завдання) Основна таблиця системи, що зберігає операційні дані робочого процесу.

UserId (uuid) [FK]: Обов'язковий зовнішній ключ - власник завдання (пов'язаний з Users). Зв'язок налаштовано як ON DELETE CASCADE.

CategoryId (uuid) [FK]: Необов'язковий зовнішній ключ (Nullable) для зв'язку з Categories. Обмеження: ON DELETE SET NULL.

SprintId (uuid) [FK]: Необов'язковий зовнішній ключ (Nullable) для зв'язку зі Sprints (ON DELETE SET NULL).

Title (varchar) / Description (text): Назва та детальний опис завдання.

Status (int) / Priority (int): Числові переліки (Enums) для відстеження стану

виконання (ToDo, InProgress, Done) та пріоритету (Low, Medium, High).

PlannedStart / Deadline / CompletedAt (timestamp): Дати життєвого циклу завдання (можуть бути NULL).

- Таблиці організації процесу (Categories та Sprints) Призначені для структурування списку завдань користувача (UserId [FK]).

Categories: Зберігає назву (Name) та шістнадцятковий код кольору (ColorCode: varchar) для візуального відокремлення в інтерфейсі.

Sprints: Реалізує підтримку ітераційного підходу. Містить обов'язкові часові рамки StartDate та EndDate, текстову мету Goal та прапорець завершення IsCompleted (boolean).

- Інфраструктурні таблиці сповіщень (DeviceTokens та PushNotificationJobs)

DeviceTokens: Реєструє пристрої користувача для розсилки повідомлень. Містить поля Token (varchar) та Platform (varchar: ios, windows, macos).

PushNotificationJobs: Черга повідомлень для фонового воркера. Містить текст сповіщення (Title, Body), тип (Type: varchar), запланований час (ScheduledAt) та кількість спроб відправки (AttemptCount: int). Опціонально зв'язана із завданнями (TaskId) чи спринтами (SprintId).

- Аналітичні таблиці (UserStatistics та SprintStatistics) Зберігають агреговані (підраховані) дані для миттєвої видачі статистики клієнтському додатку без необхідності складних SQL-запитів GROUP BY у реальному часі.

SprintStatistics: Зв'язана зі спринтом (SprintId [FK]). Зберігає кількість закритих (TasksCompleted: int) та пропущених завдань, а також сумарні години (TotalHoursSpent: double precision).

UserStatistics: Щоденний зріз по користувачу. Містить поле Date (timestamp), кількість виконаних за день завдань та обчислений рейтинг продуктивності (ProductivityScore: int).

ER-діаграму бази даних зображено на рисунку 3.4.

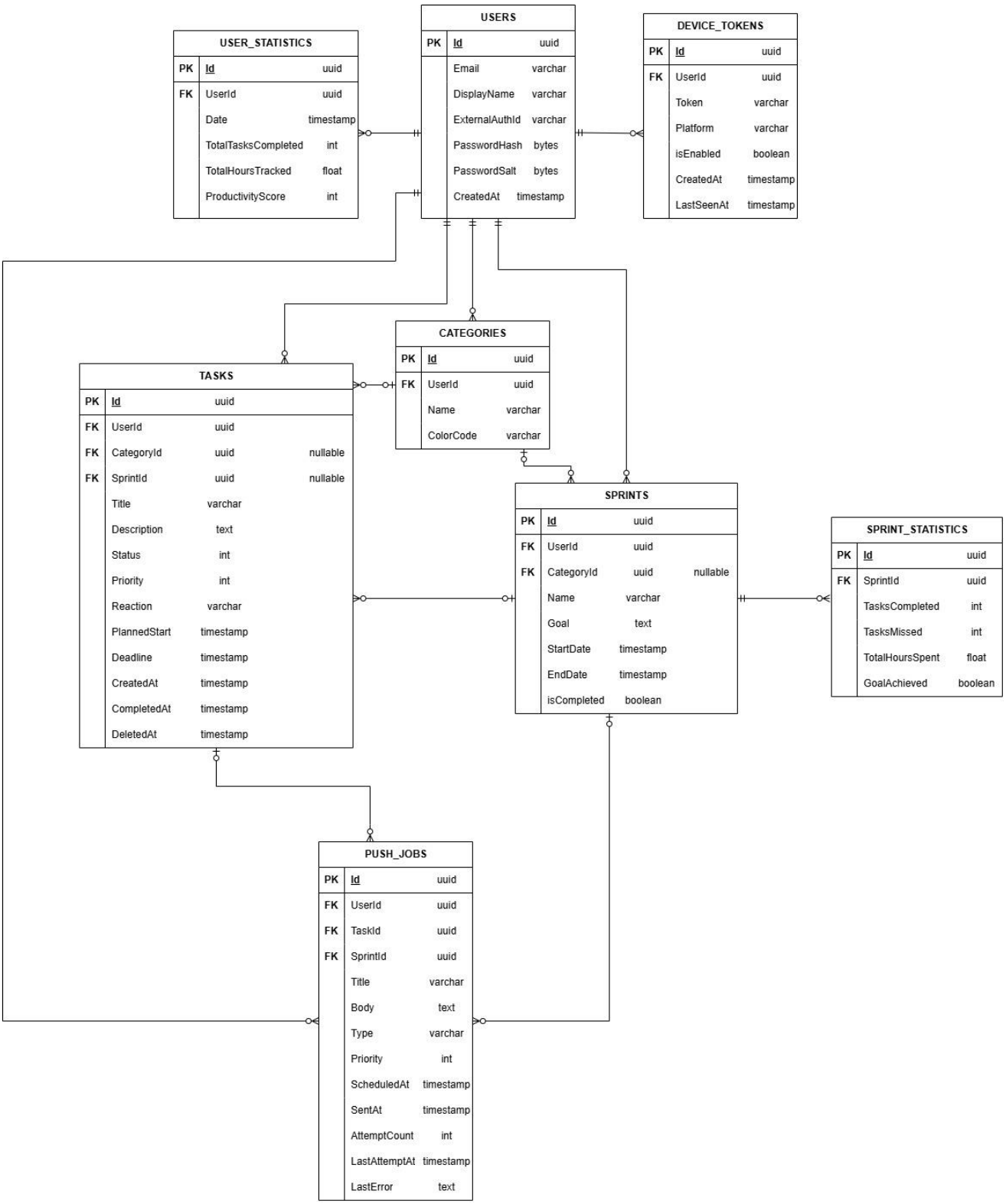


Рис.3.4. ER-діаграма бази даних

4 РЕАЛІЗАЦІЯ ЗАСТОСУНКУ ТА ТЕСТУВАННЯ

4.1. Реалізація функціоналу

Практична реалізація функціональних вимог системи ґрунтується на архітектурній моделі клієнт-сервер. Серверний рівень займає головну роль, повністю інкапсулюючи бізнес-логіку, механізми суворої валідації вхідних даних та взаємодію з базою даних через систему незалежних контролерів і фонових служб. Натомість, клієнтський застосунок бере на себе відповідальність за презентаційну логіку: управління локальним станом вікон, рендеринг графічного інтерфейсу та забезпечення миттєвого відгуку на дії користувача безпосередньо на його пристрої.

Задля забезпечення високої зв'язності всередині компонентів та низької залежності між ними, кодову базу проєкту логічно декомпозовано на чотири ключові функціональні модулі:

- Модуль безпеки та управління доступом. Гарантує криптографічну ізоляцію користувацьких даних завдяки надійним механізмам автентифікації та авторизації. Модуль підтримує гібридний конвеєр ідентифікації: від класичної перевірки хешованих облікових даних до делегованого доступу за протоколом OAuth2. Успішна ідентифікація завершується створенням захищеної сесії для безпечної обробки всіх подальших API-запитів.

- Модуль організації завдань(Core Logic). Виступає центральним ядром системи, що регламентує весь життєвий цикл робочого навантаження: від ініціалізації сутності до її архівування. Підсистема реалізує алгоритми логічної категоризації, гнучкого планування (Agile-орієнтовані спринти) та автоматизованої маршрутизації завдань, зокрема механізми перенесення протермінованих тасок між суміжними ітераціями.

- Модуль аналітики. Відповідає за безперервний моніторинг активності користувача. Реалізує високоточний хронометраж робочих сесій із подальшою агрегацією первинних даних на боці сервера. Математичний апарат модуля

трансформує розрізнені часові мітки у зведені статистичні індикатори, які доставляються клієнту для візуалізації.

- Модуль фонових процесів та багатоканальних сповіщень. Комплексна інфраструктурна підсистема для проактивної доставки нагадувань про дедлайни. Використовує комбіновану транспортну модель: інтеграцію з нативними чергами ОС для гарантованої відправки зовнішніх повідомлень (навіть коли додаток закрито), а також механізм повнодуплексного зв'язку для миттєвого оновлення стану інтерфейсу під час активного сеансу роботи користувача.

4.2. Реалізація модулю безпеки

Модуль безпеки відповідає за захист персональних даних користувачів та розмежування прав доступу до ресурсів бази даних. Архітектура безпеки побудована навколо контролера AuthController, який реалізує гібридну модель автентифікації: систему підтримує як класичну реєстрацію за допомогою електронної пошти та пароля, так і сучасний безпечний вхід через сторонніх провайдерів ідентифікації. Взаємодія клієнтського додатку з захищеними маршрутами API відбувається за стандартом Stateless Authentication з використанням токенів доступу.

4.2.1. Реєстрація та захист паролів

При використанні класичної системи реєстрації система ніколи не зберігає паролі користувачів у відкритому вигляді. Для забезпечення найвищого рівня безпеки реалізовано спеціальний клас AuthCrypto, який використовує алгоритм Pbkdf2 з 210 000 ітераціями хешування за стандартом SHA256. Процес захисту пароля починається з генерації унікальної 16-байтової криптографічної "солі" за допомогою RandomNumberGenerator. Після цього оригінальний пароль, об'єднаний з сіллю, проходить через сотні тисяч ітерацій алгоритму Pbkdf2.

Такий підхід робить процес підбору паролів зловмисниками математично та економічно недоцільним. Згенеровані бінарні масиви солі та хешу (byte[])

повертаються в контролер для збереження в базі даних. У разі спроби авторизації система використовує метод VerifyPassword та спеціальну функцію FixedTimeEquals для порівняння хешів, що захищає сервер від атак за часом виконання:

```
public static (byte[] salt, byte[] hash) HashPassword(string
password)
{
     var salt = RandomNumberGenerator.GetBytes(SaltSize);
     var hash = Rfc2898DeriveBytes.Pbkdf2(
                    password,
                    salt,
                    Pbkdf2Iterations,
                    HashAlgorithmName.SHA256,
                    HashSize);
     return (salt, hash);
}

public static bool VerifyPassword(string password, byte[] salt,
byte[] expectedHash)
{
     var hash = Rfc2898DeriveBytes.Pbkdf2(
                    password,
                    salt,
                    Pbkdf2Iterations,
                    HashAlgorithmName.SHA256,
                    expectedHash.Length);
     return CryptographicOperations.FixedTimeEquals(hash,
expectedHash);
}
```

4.2.2. OAuth2

Для зручності користувачів та подальшого розширення проєкту реалізовано механізм швидкого входу через обліковий запис Google. Клієнтський додаток самостійно отримує ідентифікаційний токен від сервісів Google і передає його на бекенд. Серверна реалізація, представлена у методі Google контролера, зчитує конфігураційні параметри, підтримуючи одночасно кілька ідентифікаторів клієнта (наприклад, для iOS, MacCatalyst та Web). Сервер ініціалізує HTTP-клієнт і відправляє отриманий токен на офіційний ендпоінт Google з використанням безпечного екранування даних в URL.

Отримавши десеріалізовану відповідь у вигляді об'єкта GoogleTokenInfo, система виконує жорстку перевірку цільової аудиторії (поля aud та azp). Сервер перевіряє, чи співпадає токен з дозволеними Client ID нашого додатку, що гарантує захист від використання токенів, викрадених з інших систем. Після успішної валідації з токена витягується унікальний ідентифікатор профілю Google (sub), за яким у базі даних виконується пошук існуючого профілю через Entity Framework. Якщо профілю немає, відбувається його автоматична реєстрація зі збереженням нормалізованої пошти; якщо профіль існує - система синхронізує його базові поля:

```
[AllowAnonymous]
[HttpPost(«google»)]
public async Task<ActionResult<AuthResponseDto>>
Google([FromBody] GoogleAuthRequestDto dto)
{
     if (string.IsNullOrWhiteSpace(dto.IdToken)) return
BadRequest(«idToken is required»);

     var googleClientId = _config[«Google:ClientId»];
     if (string.IsNullOrWhiteSpace(googleClientId))
     {
          return StatusCode(500, «Missing config:
Google:ClientId»);
     }

     // Allow multiple OAuth client ids (e.g., iOS/MacCatalyst +
Web) via comma/semicolon separated config.
     Var allowedClientIds = googleClientId
```

```
                              .Split(new[] { ‘,’, ‘;’, ‘\n’ },
StringSplitOptions.RemoveEmptyEntries |
StringSplitOptions.TrimEntries)
                              .Where(x =>
!string.IsNullOrWhiteSpace(x))
                              .ToHashSet(StringComparer.Ordinal);

     using var http = new HttpClient();
     var tokenInfoUrl =
$»https://oauth2.googleapis.com/tokeninfo?id_token={Uri.EscapeDa
taString(dto.IdToken)}»;
     var tokenInfo = await
http.GetFromJsonAsync<GoogleTokenInfo>(tokenInfoUrl);
     if (tokenInfo == null) return Unauthorized(«Invalid Google
token»);
     if (string.IsNullOrWhiteSpace(tokenInfo.email)) return
Unauthorized(«Google token does not contain email»);

     // tokeninfo returns «aud» (audience). For some flows
Google may also include «azp».
     // We accept if either matches one of the configured client
ids.
     Var audOk = !string.IsNullOrWhiteSpace(tokenInfo.aud) &&
allowedClientIds.Contains(tokenInfo.aud);
     var azpOk = !string.IsNullOrWhiteSpace(tokenInfo.azp) &&
allowedClientIds.Contains(tokenInfo.azp);
     if (!audOk && !azpOk)
     {
          // Helpful diagnostics during development.
          Return Unauthorized(
                    $»Invalid audience. Aud={tokenInfo.aud ??
«<null>»} azp={tokenInfo.azp ?? «<null>»}
allowed=[{string.Join(«, «, allowedClientIds)}]»);
     }

     // Upsert user
     var externalAuthId = tokenInfo.sub ?? string.Empty;
     var user = await _context.Users.FirstOrDefaultAsync(u =>
u.ExternalAuthId == externalAuthId);
     if (user == null)
     {
          user = new User
          {
               ExternalAuthId = externalAuthId,
```

```
                Email =
AuthCrypto.NormalizeEmail(tokenInfo.email),
                DisplayName = tokenInfo.name ?? tokenInfo.email ??
«User»,
                PasswordHash = null,
                PasswordSalt = null
            };
            _context.Users.Add(user);
            await _context.SaveChangesAsync();
        }
        else
        {
            // Sync basic fields if changed
            user.Email =
AuthCrypto.NormalizeEmail(tokenInfo.email);
            if (!string.IsNullOrWhiteSpace(tokenInfo.name))
user.DisplayName = tokenInfo.name;
            await _context.SaveChangesAsync();
        }

        return Ok(ToAuthResponse(user));
}
```

4.2.3. JWT-токени

Після успішного підтвердження особи користувача система видає йому ключ доступу у вигляді JWT-токена, який формується у методі ToAuthResponse. Алгоритм генерації починається із завантаження секретного ключа (Jwt:SigningKey) з конфігурації сервера. На його основі створюється криптографічний об'єкт SymmetricSecurityKey та визначаються параметри SigningCredentials за допомогою симетричного алгоритму HmacSha256. Далі формується Payload, до якого записуються відкриті ідентифікаційні дані: внутрішній ідентифікатор користувача у системі, його електронна пошта та ім'я. Згенерований об'єкт JwtSecurityToken ініціалізується цими заявами, цифровим підписом та встановленим терміном дії у 7 днів. Після цього клас JwtSecurityTokenHandler конвертує об'єкт у стандартний строковий формат токена, який повертається клієнту у складі DTO-об'єкта для використання в заголовках наступних запитів:

```
    private AuthResponseDto ToAuthResponse(User user)
    {
     var signingKey = _config["Jwt:SigningKey"];
     if (string.IsNullOrWhiteSpace(signingKey))
          throw new InvalidOperationException("Missing config:
Jwt:SigningKey");

     var key = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(signingKey));
     var creds = new SigningCredentials(key,
SecurityAlgorithms.HmacSha256);
     var claims = new List<Claim>
     {
          new(ClaimTypes.NameIdentifier, user.Id.ToString()),
          new(ClaimTypes.Email, user.Email),
          new("name", user.DisplayName)
     };

     var jwt = new JwtSecurityToken(
         claims: claims,
         expires: DateTime.UtcNow.AddDays(7),
         signingCredentials: creds);

     var accessToken = new
JwtSecurityTokenHandler().WriteToken(jwt);

     return new AuthResponseDto
     {
          UserId = user.Id,
          Email = user.Email,
          DisplayName = user.DisplayName,
          AccessToken = accessToken
     };
    }
```

4.3. Реалізація модулю організації завдань

Цей модуль забезпечує повний життєвий цикл робочого процесу користувача. Архітектура модуля розділена на два основні контролери: TasksController для індивідуального управління кожним завданням та

SprintsController для масової організації завдань у часові ітерації (спринти) відповідно до гнучких методологій Agile.

4.3.1. Створення та управління завданнями

Процес створення нового завдання містить кілька рівнів валідації та інтеграції з іншими модулями системи. Метод контролера спочатку витягує ідентифікатор поточного авторизованого користувача з токена доступу. Якщо користувач вирішив одразу прив'язати завдання до певної категорії або спринту (передавши CategoryId або SprintId), сервер звертається до бази даних, щоб підтвердити, що ці об'єкти існують і дійсно належать цьому користувачеві. Це унеможливлює несанкціоновану прив'язку даних до чужих проєктів.

Після успішної валідації DTO-модель, отримана від клієнта, трансформується у сутність бази даних AppTask. При цьому всі дати обов'язково конвертуються у час UTC для уникнення проблем із часовими поясами. Щойно завдання зберігається в базі даних через Entity Framework Core, контролер автоматично викликає сервіс IPushScheduler, який ставить у чергу розклад фонових нагадувань про наближення дедлайну щойно створеного завдання. У фіналі сервер збирає об'єкт відповіді з присвоєним унікальним ідентифікатором та повертає його клієнту:

```
    [HttpPost]
    public async Task<ActionResult<AppTaskResponseDto>>
CreateTask([FromBody] AppTaskCreateDto dto)
    {
        var userId = UserContext.GetRequiredUserId(User);

        if (dto.CategoryId.HasValue)
        {
            var categoryOk = await
_context.Categories.AnyAsync(c => c.Id == dto.CategoryId &&
c.UserId == userId);
            if (!categoryOk) return BadRequest("Категорію не
знайдено");
        }
```

```
          if (dto.SprintId.HasValue)
          {
               var sprintOk = await _context.Sprints.AnyAsync(s
=> s.Id == dto.SprintId && s.UserId == userId);
               if (!sprintOk) return BadRequest("Спринт не
знайдено");
          }


          var newTask = new AppTask
          {
               Title = dto.Title,
               Description = dto.Description,
               CategoryId = dto.CategoryId,
               SprintId = dto.SprintId,
               PlannedStart = dto.PlannedStart,
               Deadline = dto.Deadline.ToUniversalTime(),
               Status = dto.Status,
               Priority = dto.Priority,

               UserId = userId
          };


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

          await
_pushScheduler.EnsureNextTaskReminderAsync(userId, newTask,
HttpContext.RequestAborted);


          var responseDto = new AppTaskResponseDto
          {
               Id = newTask.Id,
               Title = newTask.Title,
               Description = newTask.Description,
               Reaction = newTask.Reaction,
               CategoryId = newTask.CategoryId,
               SprintId = newTask.SprintId,
               PlannedStart = newTask.PlannedStart,
               Deadline = newTask.Deadline,
               CreatedAt = newTask.CreatedAt,
               CompletedAt = newTask.CompletedAt,
```

```
            Status = newTask.Status,
            Priority = newTask.Priority
        };

        return CreatedAtAction(nameof(GetTasks), new { id =
newTask.Id }, responseDto);
    }
```

4.3.2. Створення та управління спринтами

Найскладнішою та найкориснішою функцією цього блоку є алгоритм RolloverSprint, який автоматизує процес закриття поточного етапу роботи та планування наступного. Коли користувач ініціює цей процес, сервер знаходить активний спринт за його ідентифікатором, позначає його як успішно завершений (IsCompleted = true) та негайно скасовує всі заплановані системні Push-сповіщення, пов'язані з його дедлайном. Одразу після цього створюється абсолютно новий спринт із параметрами (назвою, метою, новими датами), які клієнт передав у тілі запиту. Найважливішим етапом є міграція даних: система виконує запит до таблиці завдань, щоб знайти всі задачі зі старого спринту, статус яких не дорівнює "Виконано" (Status != 2). Для кожної знайденої відкритої задачі сервер автоматично змінює зовнішній ключ SprintId на ідентифікатор щойно створеного спринту та синхронізує дедлайн задачі з датою закінчення нової ітерації. Після успішного збереження всіх змін у базі формується детальна статистична відповідь для клієнтського додатку:

```
[HttpPost("{id}/rollover")]
public async Task<ActionResult<SprintResponseDto>>
RolloverSprint(Guid id, [FromBody] SprintCreateDto dto)
{
    var userId = UserContext.GetRequiredUserId(User);

    var sprint = await _context.Sprints.FindAsync(id);
    if (sprint == null) return NotFound("Спринт не знайдено");
    if (sprint.UserId != userId) return Forbid();
```

```
        sprint.IsCompleted = true;
        await CancelPendingSprintDeadlinePushesAsync(id);

        var newSprint = new Sprint
        {
            Name = dto.Name,
            Goal = dto.Goal,
            StartDate = NormalizeClientUtc(dto.StartDate),
            EndDate = NormalizeClientUtc(dto.EndDate),
            IsCompleted = false,
            CategoryId = dto.CategoryId,
            UserId = userId
        };

        _context.Sprints.Add(newSprint);

        var openTasks = await _context.Tasks
                            .Where(t => t.UserId == userId &&
t.DeletedAt == null && t.SprintId == id && t.Status != 2)
                            .ToListAsync();

        foreach (var task in openTasks)
        {
            task.SprintId = newSprint.Id;
            task.Deadline = newSprint.EndDate;
        }

        await _context.SaveChangesAsync();

        return Ok(new SprintResponseDto
        {
            Id = newSprint.Id,
            Name = newSprint.Name,
            Goal = newSprint.Goal,
            StartDate = MarkUtc(newSprint.StartDate),
            EndDate = MarkUtc(newSprint.EndDate),
            IsCompleted = newSprint.IsCompleted,
            CategoryId = newSprint.CategoryId,
            TotalTasks = openTasks.Count,
            DoneTasks = 0,
            OpenTasks = openTasks.Count,
            Tasks = new List<AppTaskResponseDto>()
        });
    }
```

4.4. Реалізація клієнтського застосунку

Архітектура застосунку строго дотримується патерну MVVM, що забезпечує чітке розділення графічного інтерфейсу та бізнес-логіки. Для мінімізації зв'язності коду між компонентами активно застосовується механізм впровадження залежностей (Dependency Injection).

4.4.1. Мережева взаємодія та ін’єкція токенів авторизації

Для комунікації з серверним REST API використовується централізований HTTP-клієнт. Щоб не дублювати логіку додавання JWT-токена до кожного окремого мережевого запиту, у системі реалізовано спеціальний перехоплювач на базі класу DelegatingHandler. Цей компонент автоматично втручається в життєвий цикл кожного вихідного HTTP-запиту. Спочатку він асинхронно звертається до інтерфейсу безпечного локального сховища пристрою (IAuthTokenStore), щоб витягти збережений раніше токен доступу. Якщо токен існує, перехоплювач автоматично формує стандартний заголовок Authorization з префіксом Bearer та прикріплює його до запиту перед остаточною відправкою на сервер. Такий архітектурний підхід робить мережевий шар додатку абсолютно прозорим для розробника, оскільки всі інші класи-сервіси можуть відправляти запити до API, не турбуючись про налаштування авторизації:

```
public sealed class AuthorizationMessageHandler : DelegatingHandler
{
	private readonly IAuthTokenStore _tokens;

	public AuthorizationMessageHandler(IAuthTokenStore tokens)
	{
		_tokens = tokens;
	}

	protected override async Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken cancellationToken)
```

```
        {
                var token = await
_tokens.GetTokenAsync(cancellationToken).ConfigureAwait(false);
                if (!string.IsNullOrWhiteSpace(token))
                        request.Headers.Authorization = new
AuthenticationHeaderValue("Bearer", token);

                return await base.SendAsync(request,
cancellationToken).ConfigureAwait(false);
        }
}
```

4.4.2. Обробка бізнес-логіки

Взаємодія інтерфейсу користувача (View) з моделями даних реалізована виключно через класи ViewModel. Для скорочення шаблонного коду використовується бібліотека CommunityToolkit.Mvvm, яка автоматично генерує необхідні властивості та команди на етапі компіляції (завдяки атрибутам на кшталт [RelayCommand]). Яскравим прикладом цього архітектурного підходу є LoginViewModel, що відповідає за авторизацію. Метод ініціалізації входу через Google отримує ідентифікаційний токен від нативних сервісів операційної системи та передає його до API-клієнта.

Уся ця бізнес-логіка загорнута у спеціальний метод-делегат RunAuthFlowAsync. Цей метод виконує кілька критично важливих завдань безпеки та стабільності: по-перше, він гарантує, що операція входу не буде запущена двічі одночасно (блокування через прапорець IsBusy); по-друге, він ініціалізує токен скасування (CancellationTokenSource) з тайм-аутом у 60 секунд для уникнення "зависань" додатку при поганому інтернет-з'єднанні; по-третє, він перехоплює будь-які мережеві помилки (через блок try-catch) і обов'язково повертає оновлення інтерфейсу - наприклад, виведення тексту помилки або приховування індикатора завантаження - виключно в головний потік виконання додатку (MainThread.BeginInvokeOnMainThread). Це гарантує, що додаток не завершить роботу аварійно через спробу змінити графічний інтерфейс із фонового обчислювального потоку.

```
    [RelayCommand]
private async Task SignInWithGoogleAsync()
{
    await RunAuthFlowAsync(async ct =>
    {
        CrashLogger.Write("Auth", "Google sign-in started");
        var idToken = await
_google.GetGoogleIdTokenAsync(ct).ConfigureAwait(false);
        CrashLogger.Write("Auth", $"Google id_token received
(len={idToken.Length})");

        var auth = await
_api.AuthenticateWithGoogleAsync(idToken,
ct).ConfigureAwait(false);
        await FinishAuthAsync(auth, ct).ConfigureAwait(false);
    }).ConfigureAwait(false);
}

private async Task RunAuthFlowAsync(Func<CancellationToken,
Task> action)
{
    if (IsBusy) return;

    MainThread.BeginInvokeOnMainThread(() =>
    {
        Error = null;
        IsBusy = true;
    });

    try
    {
        using var cts = new
CancellationTokenSource(TimeSpan.FromSeconds(60));
        await action(cts.Token).ConfigureAwait(false);
    }
    catch (Exception ex)
    {
        CrashLogger.Write("AuthError", ex.ToString());
        MainThread.BeginInvokeOnMainThread(() =>
                                            Error =
string.IsNullOrWhiteSpace(ex.Message) ? ex.ToString() :
ex.Message);
    }
    finally
```

```
    {
        MainThread.BeginInvokeOnMainThread(() => IsBusy =
false);
    }
}
```

4.4.3. Взаємодія користувача з системою

Взаємодія користувача з графічним інтерфейсом додатку побудована на декларативній розмітці XAML, яка тісно інтегрована з патерном MVVM за допомогою механізму Data Binding. Розглянемо цей підхід на прикладі головного робочого простору - віртуальної дошки із завданнями. Замість жорсткого маніпулювання UI-елементами через код, система використовує BindableLayout всередині контейнера вільного позиціонування AbsoluteLayout. Директива ItemsSource="{Binding VisibleStickyTasks}" змушує додаток автоматично генерувати візуальні компоненти карток завдань (StickyNoteView) на основі колекції об'єктів з ViewModel. Якщо додати або видалити завдання у колекції, інтерфейс миттєво синхронізується і перемалюється сам.

Складна взаємодія, як-от перетягування карток по екрану або обробка кліків, реалізована через гнучку систему жестів GestureRecognizers. Наприклад, PointerGestureRecognizer постійно відстежує рух курсора чи пальця (PointerMoved) для переміщення нотатки, тоді як TapGestureRecognizer прив'язаний до команди CloseDropPanelsCommand, яка інкапсульована у ViewModel і відповідає за приховування всіх бічних панелей при кліку по порожньому місцю дошки. Окремо варто відзначити динамічне позиціонування: завдяки прив'язці LayoutBounds="{Binding LayoutBounds}", саме ViewModel обчислює пропорційні координати кожної картки на екрані, а XAML лише відображає їх. Також інтерфейс дошки повністю підтримує автоматичну адаптацію до системної теми (світлої чи темної) за допомогою вбудованого розширення AppThemeBinding, що дозволяє змінювати кольори фону "на льоту".

```
<AbsoluteLayout
    x:Name="StickyCanvas"
```

```
        Margin="36,36,36,36"
        BackgroundColor="Transparent"
        SizeChanged="OnCanvasSizeChanged"
        ZIndex="80">

        <AbsoluteLayout.GestureRecognizers>
            <TapGestureRecognizer                Command="{Binding
CloseDropPanelsCommand}" />
            <PointerGestureRecognizer
PointerMoved="OnStickyBoardPointerMoved" />
        </AbsoluteLayout.GestureRecognizers>

        <BindableLayout.ItemsSource>
            <Binding Path="VisibleStickyTasks" />
        </BindableLayout.ItemsSource>

        <BindableLayout.ItemTemplate>
            <DataTemplate x:DataType="localModels:StickyTask">
                <controls:StickyNoteView
                    AbsoluteLayout.LayoutBounds="{Binding
LayoutBounds}"

AbsoluteLayout.LayoutFlags="PositionProportional"
                    Host="{Binding            Source={x:Reference
ThisWorkspace}, Path=BindingContext}" />
            </DataTemplate>
        </BindableLayout.ItemTemplate>
    </AbsoluteLayout>
```

4.5. Реалізація модулю аналітики

Модуль аналітики відіграє ключову роль у підтримці мотивації користувача, агрегуючи великі масиви даних про його активність у зрозумілі метрики. Серцем цієї системи є StatsController, який обробляє різноманітні запити: від формування загального резюме профілю до побудови детальних часових рядів (погодинних або щоденних графіків). Розглянемо базовий алгоритм обчислення загальної статистики користувача у методі GetUserStats. Клієнтський додаток передає бажаний діапазон дат, який сервер одразу нормалізує до формату UTC, щоб повністю уникнути технічних розбіжностей між часовими поясами пристроїв.

Потім система використовує функціонал Entity Framework Core для формування оптимізованих запитів. Замість того, щоб завантажувати всі завдання в оперативну пам'ять сервера і рахувати їх там, підрахунок кількості завдань за різними статусами (ToDo, InProgress, Done) виконується безпосередньо на боці бази даних через асинхронні виклики CountAsync.

Після отримання ідентифікаторів актуальних завдань, сервер витягує всі пов'язані записи відстеження часу та ітеративно підраховує сумарну кількість витрачених годин.

Найцікавішою частиною цього алгоритму є елемент гейміфікації: система розраховує так званий "бал продуктивності" (Productivity Score) за допомогою спеціальної евристики. Згідно з цим підходом, кожне успішно завершене завдання приносить користувачу 10 базових балів, а кожна година сфокусованої роботи, офіційно відстеженої таймером, додає ще один бал. Сформована агрегована статистика повертається клієнту у вигляді оптимізованого DTO-об'єкта для миттєвої візуалізації на дашборді:

```
    [HttpGet("user")]
    public async Task<ActionResult<UserStatsResponseDto>>
GetUserStats([FromQuery] DateTime? from, [FromQuery] DateTime?
to)
    {
     var userId = UserContext.GetRequiredUserId(User);

     var fromUtc = (from ?? DateTime.UtcNow.Date.AddDays(-
7)).ToUniversalTime();
     var toUtc = (to ?? DateTime.UtcNow).ToUniversalTime();
     if (toUtc < fromUtc) return BadRequest("Invalid date
range");

     var tasksQuery = _db.Tasks.Where(t => t.UserId == userId);
     var activeTasksQuery = tasksQuery.Where(t => t.DeletedAt ==
null);

     var tasksTodo = await activeTasksQuery.CountAsync(t =>
t.Status == 0);
     var tasksInProgress = await activeTasksQuery.CountAsync(t
=> t.Status == 1);
```

```
    var tasksCompleted = await activeTasksQuery.CountAsync(t =>
t.Status == 2);

    var tasksCreated = await tasksQuery.CountAsync(t =>
t.CreatedAt >= fromUtc && t.CreatedAt <= toUtc);

    var taskIds = await activeTasksQuery.Select(t =>
t.Id).ToListAsync();
    var logs = await _db.TimeLogs
                .Where(l => taskIds.Contains(l.TaskId) &&
l.StartedAt >= fromUtc && l.StartedAt <= toUtc)
                .ToListAsync();

    var total = TimeSpan.Zero;
    foreach (var log in logs)
    {
        total += (log.EndedAt ?? DateTime.UtcNow) -
log.StartedAt;
    }

    var totalHours = Math.Round(total.TotalHours, 2);

    var productivity = (tasksCompleted * 10) +
(int)Math.Round(totalHours);

    return Ok(new UserStatsResponseDto
    {
        From = fromUtc,
        To = toUtc,
        TasksCompleted = tasksCompleted,
        TasksCreated = tasksCreated,
        TasksInProgress = tasksInProgress,
        TasksTodo = tasksTodo,
        TotalHoursTracked = totalHours,
        ProductivityScore = productivity
    });
   }
```

4.6. Реалізація модулю фонових сповіщень

Підсистема сповіщень розроблена за принципами повністю асинхронної архітектури та працює окремо від основних потоків користувацьких запитів, щоб

гарантувати миттєву відповідь API. Процес поділяється на два етапи: планування та фактичну відправку. Серцем системи планування є спеціальний сервіс PushScheduler, який викликається щоразу, коли завдання створюється або модифікується. Головний алгоритм зосереджений у методі EnsureNextTaskReminderAsync. Його робота починається з перевірки статусу: якщо користувач щойно відмітив завдання як виконане (статус 2), система негайно скасовує всі системні нагадування для цього ідентифікатора у базі даних бази даних, звільняючи ресурси.

Якщо ж завдання залишається в роботі, алгоритм розраховує часовий інтервал до наступного "пінгу", зважаючи на встановлений пріоритет (чим вищий пріоритет - тим частіші нагадування) та близькість до дедлайну. Критично важливою умовою бізнес-логіки є те, що час наступного сповіщення програмно не може перевищити сам дедлайн завдання. Алгоритм автоматично зміщує обчислений час відправки до кінцевого терміну, щоб гарантувати своєчасне попередження користувача, і динамічно змінює заголовок повідомлення на "Прострочено", якщо розсилка планується в останні 90 секунд.

На завершальному етапі система перевіряє таблицю черги PushNotificationJobs. Замість того, щоб безконтрольно плодити сотні нових повідомлень при кожній зміні тексту завдання, планувальник оптимізує роботу інфраструктури: він знаходить вже існуюче невідправлене повідомлення для цього завдання і просто оновлює його розклад та текст.

Завдяки такому підходу, окремий системний фоновий процес (PushJobWorker) завжди опрацьовує ідеально чисту чергу, що дозволяє йому миттєво доставляти повідомлення через нативні канали операційних систем (WNS, APNs) та WebSockets (SignalR):

```
public async Task EnsureNextTaskReminderAsync(Guid userId,
AppTask task, CancellationToken cancellationToken)
{
    if (task.Status == 2)
    {
```

```
          await CancelTaskRemindersAsync(userId, task.Id,
cancellationToken);
          return;
     }

     var interval =
TaskReminderSchedulingRules.GetInterval(task.Priority,
task.Deadline, DateTime.UtcNow);
     if (interval == null) return;

     var now = DateTime.UtcNow;
     var nextAt = now.Add(interval.Value);

     if (task.Deadline.HasValue)
     {
          var dl = task.Deadline.Value;
          if (dl <= now)
               return;

          if (nextAt > dl)
               nextAt = dl;
     }

     if (nextAt <= now)
          nextAt = now.AddSeconds(15);

     var existing = await _db.PushNotificationJobs
                    .Where(j => j.UserId == userId && j.TaskId
== task.Id && j.SentAt == null && j.Type == "task_reminder")
                    .OrderByDescending(j => j.ScheduledAt)
                    .FirstOrDefaultAsync(cancellationToken);

     var titlePrefix = "Нагадування";
     if (task.Deadline.HasValue && nextAt >=
task.Deadline.Value.AddSeconds(-90))
          titlePrefix = "Прострочено";

     var title = $"{titlePrefix}: {task.Title}";
     var body = task.Deadline.HasValue
                ? $"Дедлайн: {task.Deadline.Value:yyyy-MM-dd
HH:mm} (UTC)"
                : "Не забудь про задачу";

     if (existing == null)
```

```
    {
        _db.PushNotificationJobs.Add(new PushNotificationJob
        {
            UserId = userId,
            TaskId = task.Id,
            ScheduledAt = nextAt,
            Title = title,
            Body = body,
            Priority = task.Priority,
            Type = "task_reminder"
        });
    }
    else
    {
        existing.ScheduledAt = nextAt;
        existing.Title = title;
        existing.Body = body;
        existing.Priority = task.Priority;
    }

    await _db.SaveChangesAsync(cancellationToken);
}
```

4.7. Реалізація тестування системи

Для забезпечення стабільності та надійності застосовано комплексну стратегію тестування з використанням фреймворку xUnit. Архітектура тестів чітко розділена на ізольовані рівні: модульні Unit та інтеграційні тести для бекенду, а також спеціалізовані тести для клієнтського додатку.

4.7.1. Модульне тестування серверної логіки

Модульні тести API відповідають за перевірку чистої бізнес-логіки системи, яка не залежить від зовнішніх сервісів, баз даних або мережі. Чудовим прикладом такого підходу є тестування криптографічного модуля AuthCrypto. Замість того, щоб писати окремий тест для кожного можливого варіанту електронної пошти, використовується атрибут [Theory], який дозволяє передавати масиви тестових даних (через [InlineData]) в один і той самий метод. Це гарантує, що функція

нормалізації пошти правильно обробляє зайві пробіли та великі літери за всіма можливими сценаріями.

Також перевіряється найголовніший криптографічний процес: тест генерує унікальну сіль та хеш для пароля, а потім через атрибут [Fact] імітує спробу авторизації, перевіряючи, чи метод VerifyPassword успішно розпізнає правильний пароль і відхиляє помилковий. Оскільки ці тести не звертаються до диска чи бази даних, вони виконуються миттєво:

```
[Trait("Category", "Unit")]
public sealed class AuthCryptoTests
{
    [Theory]
    [InlineData("  A@B.COM ", "a@b.com")]
    [InlineData(null, "")]
    public void NormalizeEmail_trims_and_lowercases(string? input, string expected)
    {
        Assert.Equal(expected, AuthCrypto.NormalizeEmail(input));
    }
    [Fact]
    public void HashPassword_and_VerifyPassword_roundtrip()
    {
        var (salt, hash) = AuthCrypto.HashPassword("correct horse battery staple");
        Assert.True(AuthCrypto.VerifyPassword("correct horse battery staple", salt, hash));
    }
}
```

4.7.2. Інтеграційне тестування API

На відміну від модульних, інтеграційні тести перевіряють роботу всієї системи в комплексі - від HTTP-запиту до збереження даних. Для цього використовується інструмент WebApplicationFactory, який автоматично піднімає повноцінний тестовий сервер у пам'яті разом з ізольованою базою даних. У наведеному нижче прикладі інтеграційного тесту емулюється повний користувацький сценарій керування завданням. Спочатку система автоматично реєструє нового віртуального клієнта (RegisterClientAsync) та отримує дійсний JWT-токен доступу. Далі тестовий клієнт відправляє POST-запит на створення нового завдання і перевіряє, чи сервер повернув код успішного виконання (EnsureSuccessStatusCode). Найважливіша перевірка відбувається на етапі видалення: клієнт відправляє DELETE-запит, після чого знову запитує весь список завдань. Тест за допомогою Assert.Empty переконується, що механізм Soft Delete спрацював правильно, і видалене завдання більше не повертається у загальному списку:

```
[Trait("Category", "Integration")]
[Collection("ApiIntegration")]
public sealed class TasksIntegrationTests
{
    private readonly ApiIntegrationFixture _fx;

    public TasksIntegrationTests(ApiIntegrationFixture fx) =>
_fx = fx;

    [Fact]
    public async Task
DeleteTask_soft_deletes_and_excludes_from_list()
    {
        var (client, _) = await
RegisterClientAsync(_fx.Factory);
        var deadline = DateTime.UtcNow.AddDays(1);
        var createResp = await
client.PostAsJsonAsync("api/tasks", new AppTaskCreateDto
        {
```

```
            Title = "Del",
            Deadline = deadline,
            Status = 0,
            Priority = 1
        }, JsonConfig.Options);
        createResp.EnsureSuccessStatusCode();
        var created = await
createResp.Content.ReadFromJsonAsync<AppTaskResponseDto>(JsonCon
fig.Options);
        Assert.NotNull(created);

        var delResp = await
client.DeleteAsync($"api/tasks/{created.Id}");
        delResp.EnsureSuccessStatusCode();

        var listResp = await client.GetAsync("api/tasks");
        listResp.EnsureSuccessStatusCode();
        var list = await
listResp.Content.ReadFromJsonAsync<List<AppTaskResponseDto>>(Jso
nConfig.Options);
        Assert.Empty(list!);
    }
}
```

4.7.3. Тестування клієнтського застосунку

Тестування клієнтської частини вимагає особливого підходу, оскільки клієнтський додаток не повинен відправляти реальні запити на сервер під час проходження тестів. Для вирішення цієї проблеми розроблено спеціальний клас-заглушку RecordingHandler, який успадковує HttpMessageHandler.

Цей перехоплювач вбудовується безпосередньо у тестовий HttpClient. Коли ApiClient намагається відправити HTTP-запит (наприклад, на авторизацію), RecordingHandler миттєво перехоплює його, зберігає всі параметри запиту у властивість LastRequest і миттєво повертає наперед запрограмовану (фейкову) відповідь у форматі JSON.

Далі фреймворк тестування перевіряє, чи правильно клієнтський клас сформував URL-адресу (http://localhost/api/auth/login) та чи коректно він десеріалізував отримані з фейкової відповіді дані (наприклад, електронну пошту).

Такий механізм гарантує 100% покриття мережевої логіки без жодного реального звернення до бекенду:

```
    public sealed class ApiClientTests
    {
        private sealed class RecordingHandler :
HttpMessageHandler
        {
            public HttpRequestMessage? LastRequest { get;
private set; }
            private readonly Func<HttpRequestMessage,
HttpResponseMessage> _respond;

            public RecordingHandler(Func<HttpRequestMessage,
HttpResponseMessage>? respond = null)
            {
                _respond = respond ?? (_ => new
HttpResponseMessage(HttpStatusCode.OK)
                {
                    Content = new StringContent("[]")
                });
            }

            protected override Task<HttpResponseMessage>
SendAsync(HttpRequestMessage request, CancellationToken
cancellationToken)
        {
            LastRequest = request;
            return Task.FromResult(_respond(request));
        }
    }
```

```
    [Fact]
    public async Task LoginWithEmailAsync_posts_to_login_route()
    {
        var inner = new RecordingHandler(_ => new
HttpResponseMessage(HttpStatusCode.OK)
        {
            Content = new StringContent("""{"userId":"00000000-
0000-0000-0000-
000000000001","email":"a@b.c","displayName":"A","accessToken":"j
wt"}""")
        });
        using var http = new HttpClient(inner) { BaseAddress =
new Uri("http://localhost/") };
        var api = new ApiClient(http);
        var dto = await api.LoginWithEmailAsync("a@b.c",
"secret");

        Assert.Equal("a@b.c", dto.Email);
        Assert.Equal("http://localhost/api/auth/login",
inner.LastRequest!.RequestUri!.ToString());
    }
}
```

4.8. Розгортання серверу

Процес розгортання та доставки серверної частини застосунку (REST API та бази даних PostgreSQL) повністю автоматизовано за допомогою технологій контейнеризації Docker та локальної оркестрації Docker Compose. Цей підхід забезпечує абсолютну ідентичність середовищ розробки та продуктиву, назавжди усуваючи класичну проблему "на моєму комп'ютері все працювало".

Головний бекенд-додаток пакується у контейнер за допомогою так званої багатоетапної збірки у Dockerfile: на першому етапі використовується "важкий" базовий образ .NET 9.0 SDK для відновлення залежностей, компіляції та публікації вихідного коду, а на фінальному етапі готовий скомпільований бінарний файл переноситься у мінімалістичний середовищний образ aspnet:9.0. Це рішення дозволяє суттєво зменшити розмір кінцевого контейнера, прискорити його

завантаження та значно підвищити рівень безпеки, оскільки в контейнері фізично відсутні інструменти збірки.

Для об'єднання бази даних та серверного API в єдину віртуальну мережу використовується декларативний конфігураційний файл docker-compose.yml.

Архітектурно розгортання складається з двох тісно пов'язаних сервісів: образу postgres для безпечного зберігання інформації та власне бекенд-сервісу api. Однією з найпоширеніших критичних проблем при синхронному розгортанні кількох контейнерів є те, що бекенд може спробувати встановити з'єднання з базою даних ще до того, як її рушій повністю ініціалізується в пам'яті сервера, що миттєво призведе до крашу всього API.

Для елегантного вирішення цієї гонки процесів застосовано механізм моніторингу працездатності healthcheck. Він налаштований так, що кожні 5 секунд відправляє низькорівневу термінальну команду pg_isready безпосередньо в контейнер з PostgreSQL.

Завдяки директиві depends_on: condition: service_healthy, інструмент Docker Compose примусово призупиняє запуск контейнера з API доти, поки база даних не відповість на цей "пінг" і не підтвердить свою абсолютну готовність приймати TCP-з'єднання. Крім того, для надійного збереження даних користувачів між перезапусками, оновленнями або аварійними зупинками серверів до контейнера бази підключено захищений постійний іменований том pgdata.

4.9. Демонстрація результатів

Підсумковим етапом розробки системи є перевірка її працездатності та візуальна демонстрація реалізованого графічного інтерфейсу.

Точкою входу для користувача є екран ідентифікації (рис. 3.1). Інтерфейс вікна спроєктовано мінімалістично: він містить класичну форму для введення електронної пошти та пароля, а також елемент для швидкої авторизації через сервіси облікових записів Google. Після успішної перевірки облікових даних та

отримання криптографічного токена застосунок впускає користувача до його робочого простору.

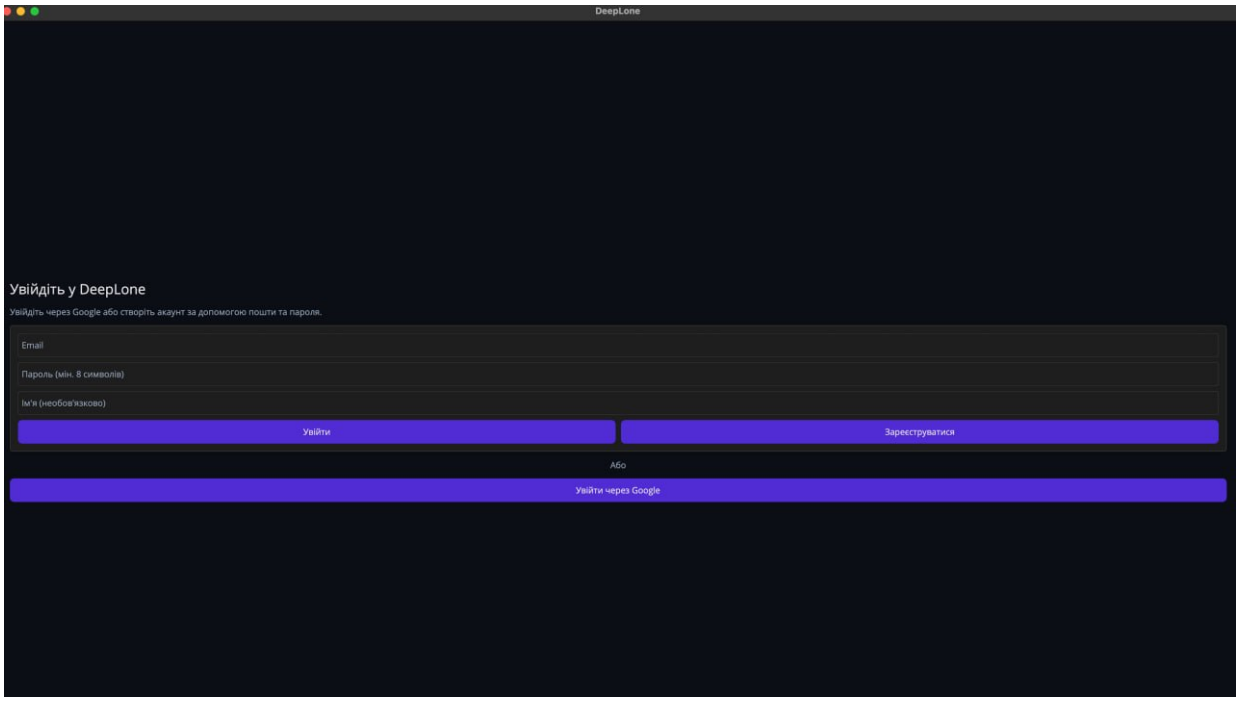


Рис. 4.1. Екран реєстрації та авторизації

Головне вікно системи розроблене з урахуванням принципів зменшення когнітивного навантаження (рис. 4.2). Центральну частину екрана займає інтерактивний простір управління завданнями. Користувач має можливість створити нові записи через форму, вказуючи назву, опис, дедлайн та частоту відправки нагадувань (рис. 4.3). Завдяки реалізації концепції просторового інтерфейсу та механізму «Drag-and-Drop», присвоєння завданню категорії чи спринту здійснюється шляхом інтуїтивного перетягування прилипали.(рис. 4.4). Також користувач може назначити завданню особливий значок та редагувати його подробиці, натиснувши кнопку з олівцем. Користувач назначає завдання виконаним натиснувши по ньому 2 рази.

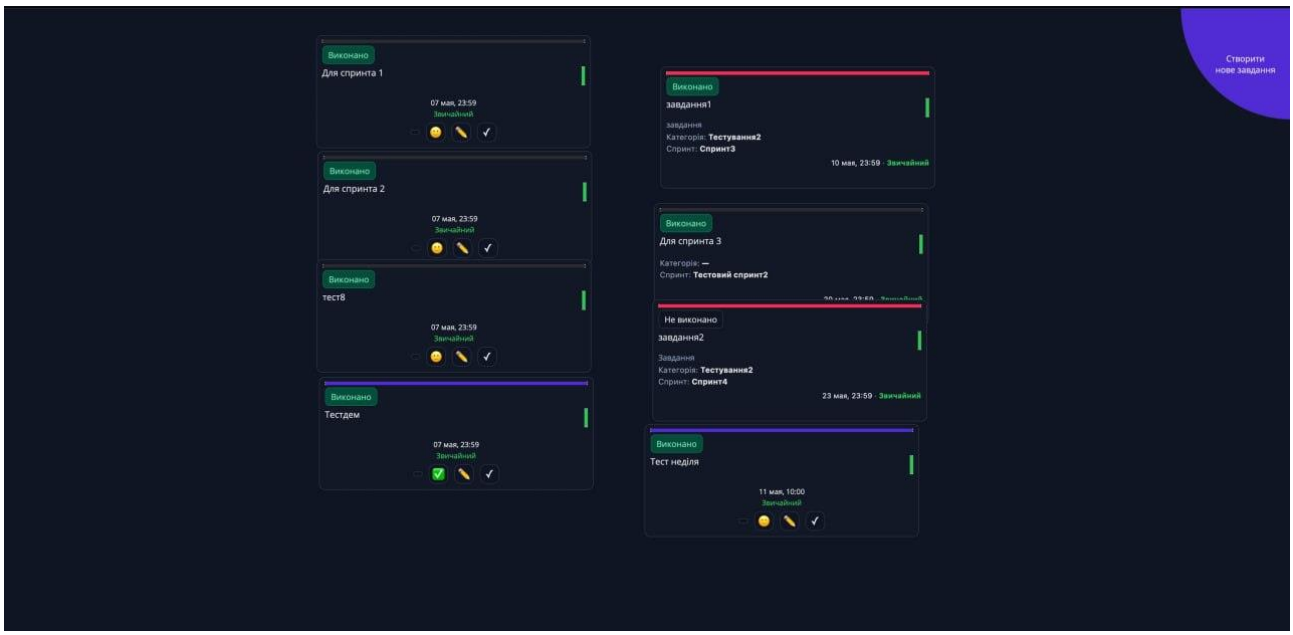


Рис. 4.2. Дошка користувача та модальна кнопка «Створити нове завдання»

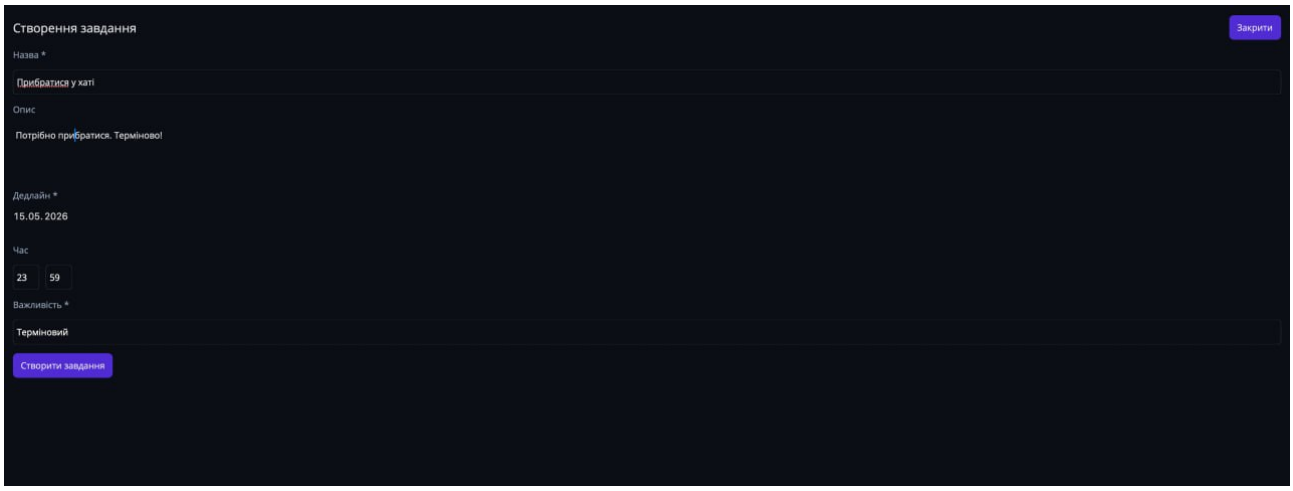


Рис. 4.3. Форма створення завдання

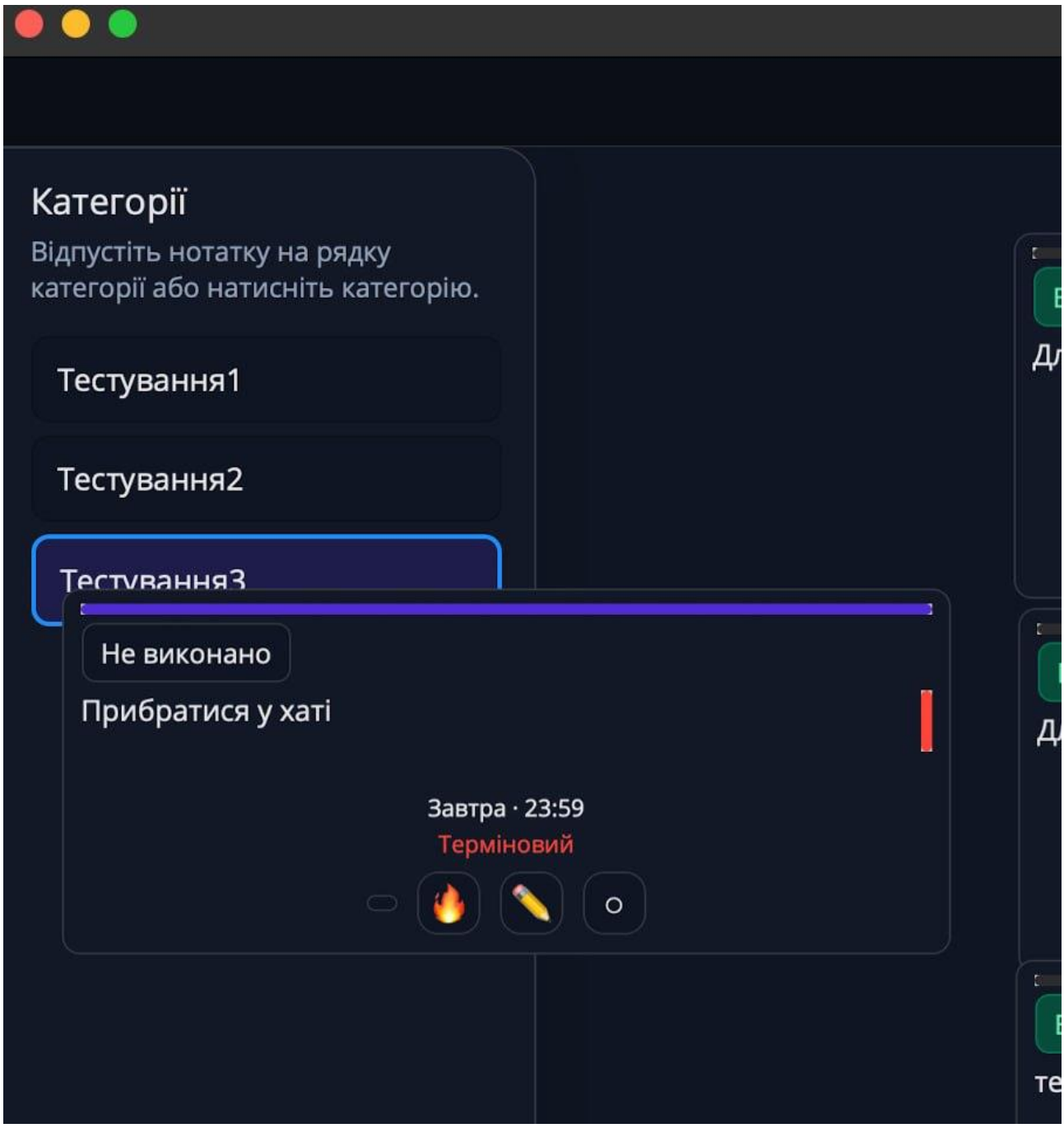


Рис. 4.4. Призначення завданню категорії за допомогою «Drag-and-Drop»

Для організації довгострокового планування за agile методологіями розроблено розділ управління робочими ітераціями (рис. 4.5). Інтерфейс дозволяє створювати нові спринти, задаючи для них конкретні часові межі та мету (рис. 4.6). Завдання призначаються за тим ж принципом, як з категоріями.

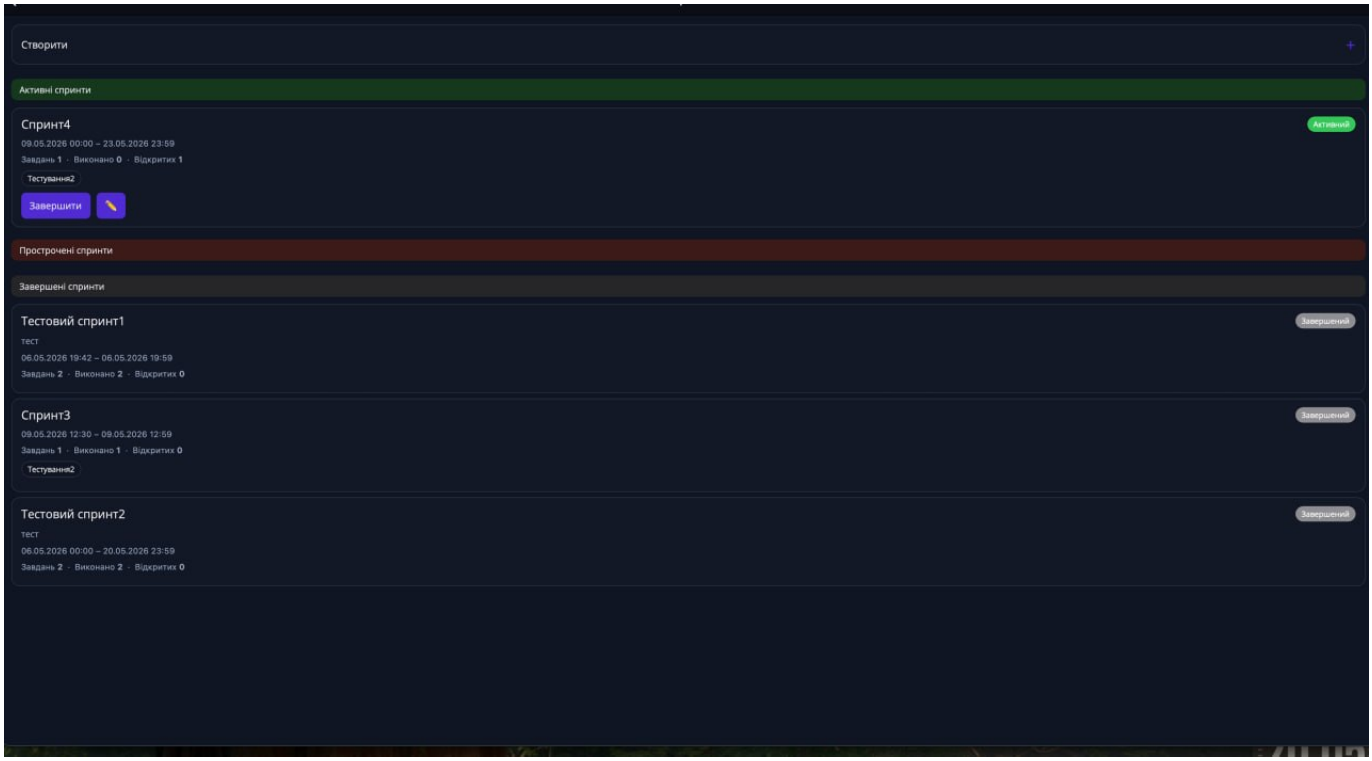


Рис. 4.5. Меню спринтів

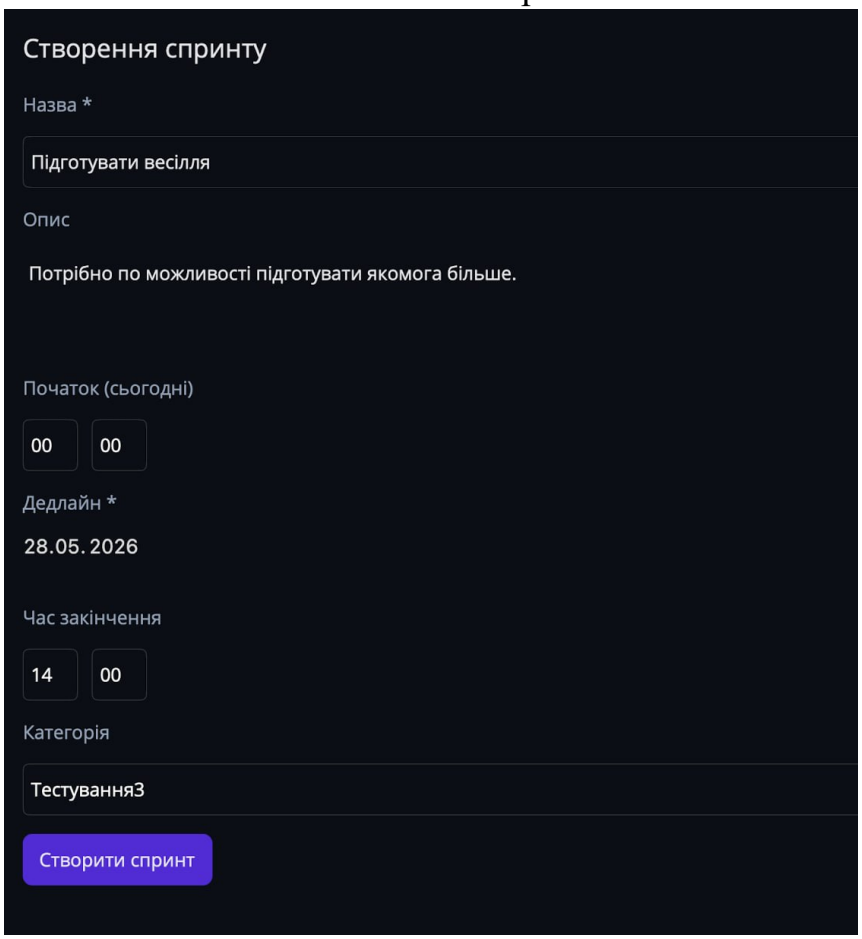


Рис. 4.6. Створення нового спринту

У меню аналітики користувач може наглядно побачити свою успішність, у вигляду графіків. Інтерфейс демонструє агреговані метрики: співвідношення успішно виконаних та прострочених завдань та загальний витрачений час (рис. 4.7).

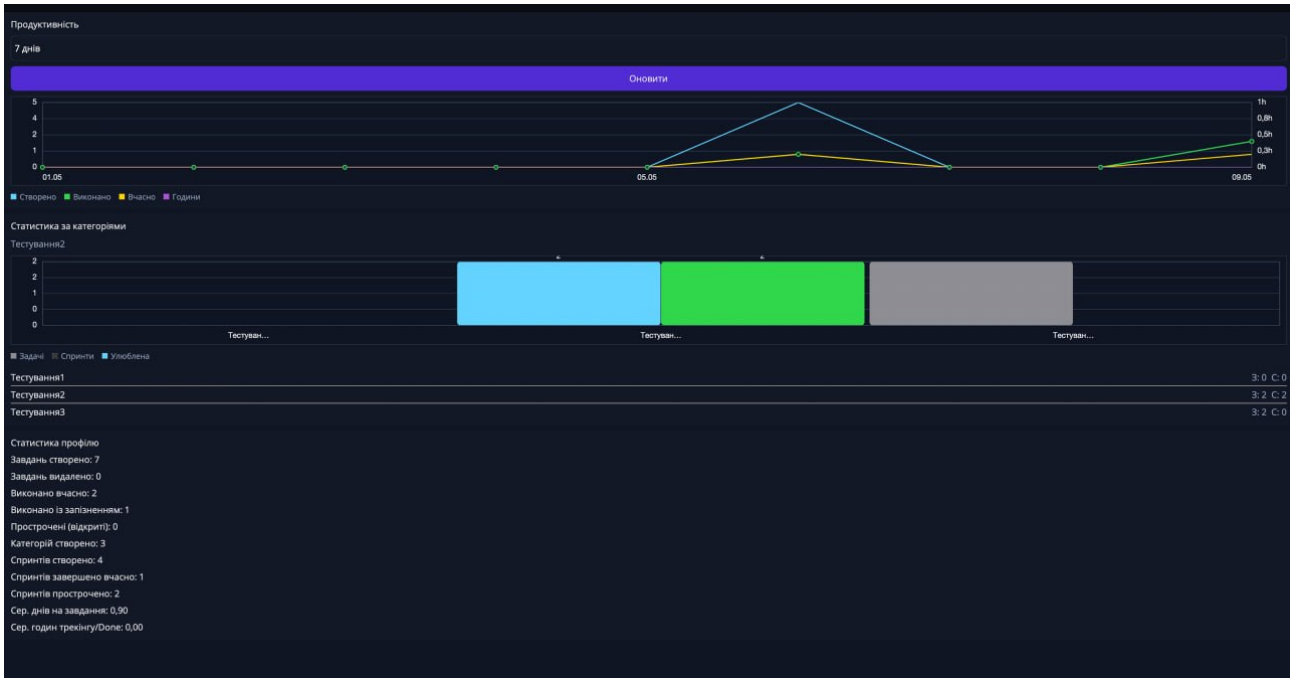


Рис. 4.7. Меню аналітики

ВИСНОВКИ

У рамках виконання кваліфікаційної роботи було успішно спроєктовано та розроблено систему персонального тайм-менеджменту «DeepLone». За результатами проведених досліджень та етапів розробки можна зробити наступні висновки:

1. дослідження когнітивних особливостей персонального тайм-менеджменту та аналіз недоліків існуючих рішень дозволили сформулювати чіткі функціональні та нефункціональні вимоги до програмного продукту. Визначено необхідність інтеграції елементів agile методологій для індивідуального планування, що стало концептуальною основою бізнес-логіки системи;

2. обґрунтовано вибір трирівневої клієнт-серверної моделі. На рівні доступу до даних розроблено оптимізовану реляційну схему під управлінням СУБД PostgreSQL із застосуванням технології об'єктно-реляційного відображення. Використання суворих зовнішніх ключів, механізмів каскадного видалення та опціональної агрегації дозволило гарантувати повну транзакційну цілісність даних при збереженні гнучкості маршрутизації завдань;

3. безумовне розділення логіки забезпечено шляхом створення REST API на базі платформи ASP.NET Core . Серверна частина інкапсулювала в собі всі критичні процеси: гібридну криптографічну авторизацію з підтримкою JWT та делегованого доступу Google OAuth2, управління життєвим циклом спринтів, агрегацію статистичних даних та роботу з фоновими чергами для асинхронної розсилки Push-сповіщень. Для забезпечення максимальної інтерактивності імплементовано механізм повнодуплексного зв'язку через WebSockets;

4. графічний інтерфейс користувача реалізовано у вигляді нативного десктопного застосунку для операційних систем macOS та Windows за допомогою фреймворку .NET MAUI. Суворе дотримання архітектурного патерну MVVM та використання декларативної розмітки XAML забезпечило високу ремонтопридатність коду. Впровадження концепції просторового інтерфейсу та

механізмів «Drag-and-Drop» дозволило мінімізувати когнітивне навантаження на користувача;

5. система забезпечує замкнений цикл управління робочим навантаженням: від реєстрації та створення завдань із дедлайнами до їх об'єднання в ітеративні спринти та точного хронометражу витраченого часу. Вбудований аналітичний модуль автоматично генерує зведені метрики ефективності, забезпечуючи користувача наочним інструментарієм для оцінки власної продуктивності;

6. етап тестування та демонстрація результатів підтвердили стабільність системи. Мережева взаємодія виконується без критичних затримок, інтерфейс працює плавно, а вся закладена бізнес-логіка функціонує коректно;

7. мета дипломної роботи досягнута в повному обсязі. Розроблений програмний продукт відповідає сучасним стандартам інженерії програмного забезпечення і є готовим до практичного використання як ефективний інструмент підвищення особистої продуктивності.

Робота пройшла апробацію. За її результатами було опубліковано наступні тези доповідей:

1. Новохацький І.О Худік Б.О Розробка застосунку для менеджменту часу на основі фреймворку .NET MAUI. Матеріали II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.498-99.

2. Новохацький І.О., Худік Б.О. Проектування та розробка програмного забезпечення на базі екосистеми .NET 9. Матеріали VII Всеукраїнську науково-технічну конференцію «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Збірник тез. К.: ДУІКТ, 2026. С.453-456.

ПЕРЕЛІК ПОСИЛАНЬ

1. Роль цифрових інструментів тайм-менеджменту в бізнес-плануванні / Електронне наукове фахове видання Хмельницького національного університету. – 2023. – URL: https://dsim.khmnu.edu.ua/index.php/dsim/article/download/432/430

2. Winters T., Manshreck T., Wright H. Software Engineering at Google: Lessons Learned from Programming Over Time. Sebastopol : O'Reilly Media, 2020. 602 p.

3. Schwaber K., Sutherland J. The Scrum Guide. The Definitive Guide to Scrum: The Rules of the Game. – 2020. – URL: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf

4. Jira Software Guide: Agile project management tools [Електронний ресурс] / Atlassian. – URL: https://www.atlassian.com/software/jira/guides

5. Аллен Д. Як упорядкувати справи. Мистецтво продуктивності без стресу / Девід Аллен. – Київ : Фабула, 2019. – 416 с.

6. Getting Things Done (GTD) with Todoist [Електронний ресурс] / Doist. – URL: https://todoist.com/productivity-methods/getting-things-done

7. Trello: A visual tool for empowering your team [Електронний ресурс]. – URL: https://trello.com/tour

8. Toggl Track: Time tracking software for teams [Електронний ресурс]. – URL: https://toggl.com/track/

9. Nieto-Rodriguez A., Viana R. How AI Will Transform Project Management. Harvard Business Review. 2023. URL: https://hbr.org/2023/02/how-ai-will-transform-project-management.

10. Noda A., Storey M. A., Forsgren N., Greiler M. DevEx: What Actually Drives Productivity. ACM Queue. 2023. Vol. 21, № 2. P. 26–44.

11. JetBrains Rider: Fast & powerful cross-platform .NET IDE. JetBrains : офіційний сайт. 2025. URL: https://www.jetbrains.com/rider/.

12. Офіційна документація мови програмування C# [Електронний ресурс] / Microsoft. – URL: https://learn.microsoft.com/uk-ua/dotnet/csharp/

13. What's new in .NET 9: Performance and Core Updates [Електронний ресурс] / Microsoft. – URL: https://learn.microsoft.com/en-us/dotnet/core/whats-new/dotnet-9

14. Freeman A. Pro ASP.NET Core 7 / Adam Freeman. – Apress, 2022. – 1184 p.

15. PostgreSQL 16 Official Documentation [Електронний ресурс] / The PostgreSQL Global Development Group. – URL: https://www.postgresql.org/docs/16/index.html

16. Entity Framework Core Overview: Object-Relational Mapper for .NET [Електронний ресурс] / Microsoft. – URL: https://learn.microsoft.com/en-us/ef/core/

17. Офіційна документація .NET MAUI (Multi-platform App UI) [Електронний ресурс] / Microsoft. – URL: https://learn.microsoft.com/uk-ua/dotnet/maui/

18. eXtensible Application Markup Language (XAML) in .NET MAUI. Microsoft Learn. 2024. URL: https://learn.microsoft.com/en-us/dotnet/maui/xaml/ (дата звернення: 08.05.2026).

19. Model-View-ViewModel (MVVM) in .NET MAUI [Електронний ресурс] / Microsoft. – URL: https://learn.microsoft.com/en-us/dotnet/architecture/maui/mvvm

20. JSON Web Token (JWT) Standard (RFC 7519) [Електронний ресурс] / Internet Engineering Task Force. – URL: https://datatracker.ietf.org/doc/html/rfc7519

21. What is OAuth 2.0? Fundamental Concepts of the Authorization Framework. Auth0 by Okta. 2025. URL: https://auth0.com/intro-to-iam/what-is-oauth-2 (дата звернення: 08.05.2026).

ДОДАТОК А. ДЕМОНАСТРАЦІЙНІ МАТЕРІАЛИ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

КАФЕДРА ІНЖЕНЕРІЇ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розробка програмного забезпечення для аналізу та управління процесами тайм-менеджменту

Виконав студент 4 курсу
групи ПД-43
Новохацький Ілля Олегович
Керівник роботи
доктор філософії (PhD) Худік Богдан Олександрович

Київ – 2026

МЕТА, ОБ’ЄКТ ТА ПРЕДМЕТ ДОСЛІДЖЕННЯ

Мета роботи: покращення управління процесами тайм-менеджменту користувачів.

Об’єкт дослідження: процес розробки програмних систем для організації, планування та управління часом користувачів.

Предмет дослідження: методи та програмні засоби для реалізації застосунку для тайм-менеджменту.

ЗАДАЧІ КВАЛІФІКАЦІЙНОЇ РОБОТИ

1. Дослідити існуючі системи управління завданнями та проаналізувати можливості адаптації agile методології для індивідуального планування.
2. Визначити функціональні та нефункціональні вимоги до розподіленої клієнт-серверної архітектури, сформувати критерії безпеки та швидкодії.
3. Обґрунтувати вибір засобів розробки: екосистема .NET для серверу-клієнту та СУБД PostgreSQL для збереження даних.
4. Розробити об'єктно-реляційну модель предметної області, визначити зв'язки між сутностями.
5. Розробити серверну бізнес-логіку, механізми управління завданнями та спринтами, а також створити графічний інтерфейс.
6. Реалізувати гібридну систему авторизації (класична + Google OAuth2) та забезпечити криптографічну ізоляцію персональної інформації користувачів.
7. Провести тестування клієнту та серверу окремо за допомогою Junit та інтеграційних тестів, провести тестування клієнт-серверної взаємодії загалом.

3

АНАЛІЗ АНАЛОГІВ

Критерій оцінювання	**Jira**	**Todoist**	**Trello**	**Toggl**	**Microsoft To Do**	**«DeepLone»**
Можливість планувати роботу спринтами	+	-	-	-	-	+
Автоматична статистика закриття завдань	+	-	-	+	-	+
Відстеження та аналіз прострочених дедлайнів	+	+	+	-	+	+
Відсутність ручного мікроменеджменту	-	+	+	-	+	+
Кроки до створення карток	5+	2	3	4	2	2

4

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Функціональні вимоги

1. Створення та управління Спринтами (встановлення назви, дати початку та дати завершення робочого циклу);
2. внесення в систему даних про нові завдання з обов'язковою можливістю встановлення чітких дедлайнів;
3. можливість прив'язки створених завдань до конкретного Спринту;
4. засоби управління життєвим циклом завдання (редагування опису, зміна статусів виконання, видалення);
5. можливість представлення зібраної статистичної інформації у вигляді наочних графіків та діаграм.

ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Нефункціональні вимоги

1. Доступ до REST API здійснюється виключно за валідними JWT-токенами. Паролі зберігаються у вигляді хешів, згенерованих за алгоритмом PBKDF2 (210 000 ітерацій) із додаванням унікальної 16-байтової солі для захисту від брутфорсу;
2. токени пристроїв для Push-сповіщень повинні реєструватися та зберігатися у середовищі на стороні сервера;
3. відправка Push-сповіщень має відбуватися асинхронно у фоновому режимі;
4.Серверна інфраструктура розгортається у середовищі Docker, для оптимізації образу API багатоетапна перевірка, а для бази даних налаштування механізму healthcheck для запобігання критичних помилок під час запуску сервісів;
5. забезпечення безперервної синхронізації даних між клієнтським додатком та централізованою базою даних PostgreSQL через REST API.

ПРОГРАМНІ ЗАСОБИ РЕАЛІЗАЦІЇ

JetBrains Rider

SignalR

PostgreSQL

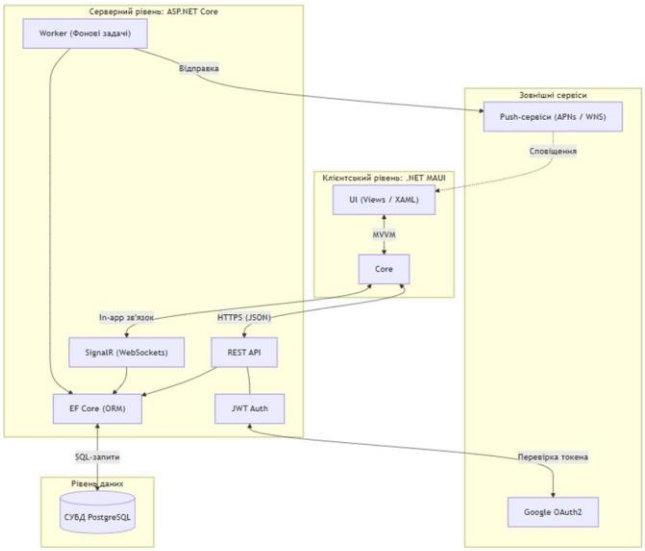
Docker

7

ДІАГРАМА ПРЕЦЕДЕНТІВ

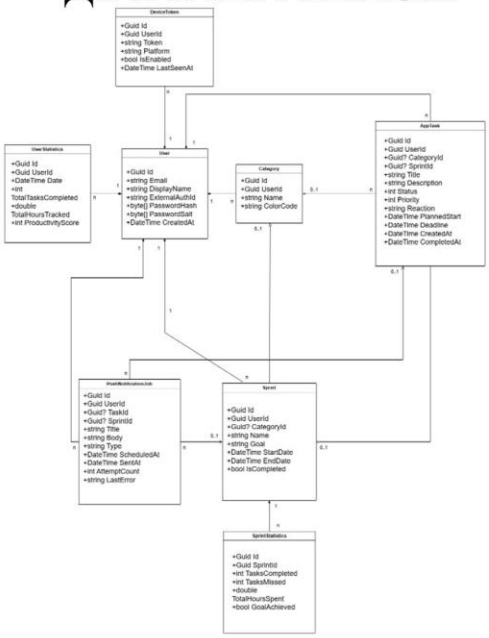
8

ДІАГРАМА АРХІТЕКТУРНОЇ ВЗАЄМОДІЇ



9

ДІАГРАМА КЛАСІВ

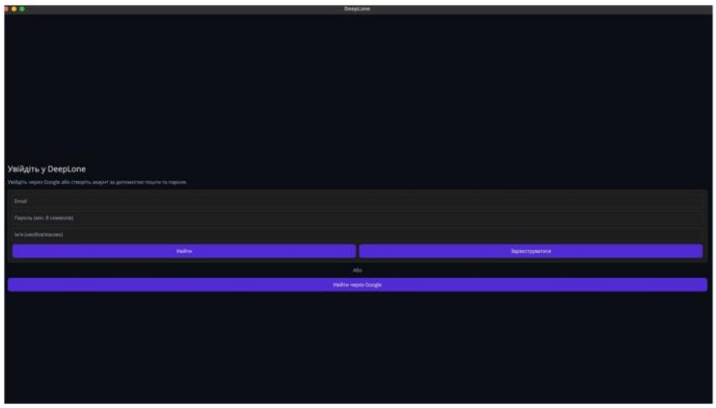


10

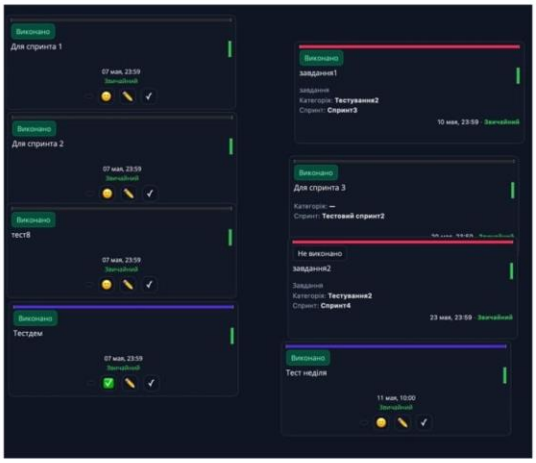
ER-діаграма бази даних

11

ЕКРАННІ ФОРМИ

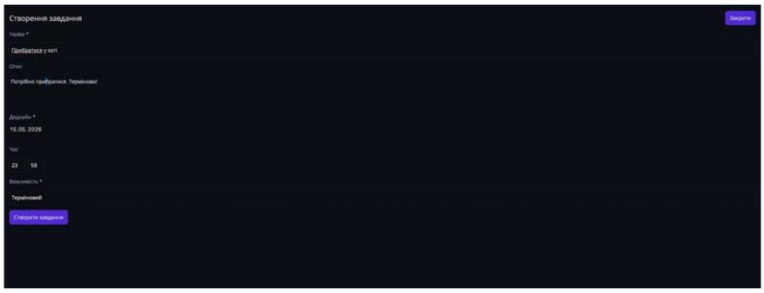


Екран реєстрації та авторизації

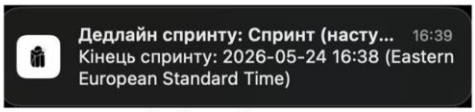


Дошка користувача

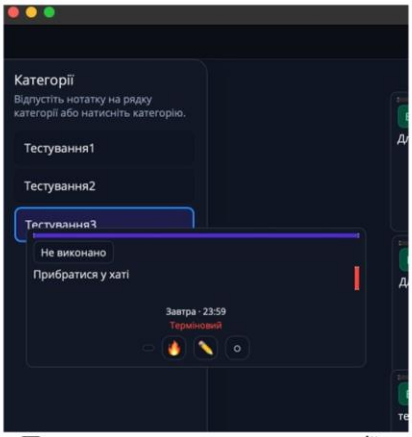
ЕКРАННІ ФОРМИ



Форма створення завдання.



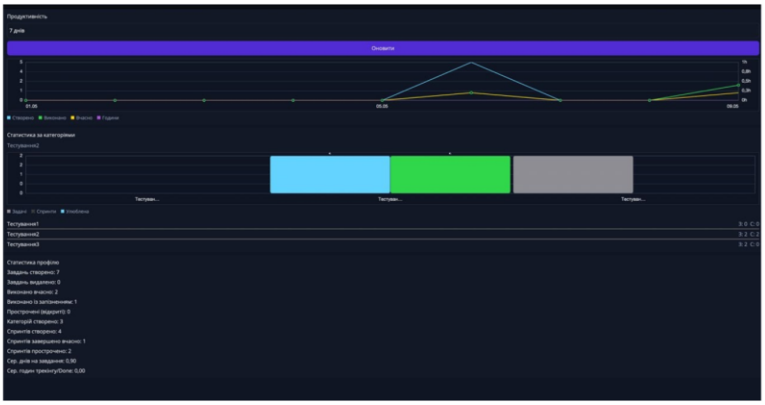
Push-повідомлення



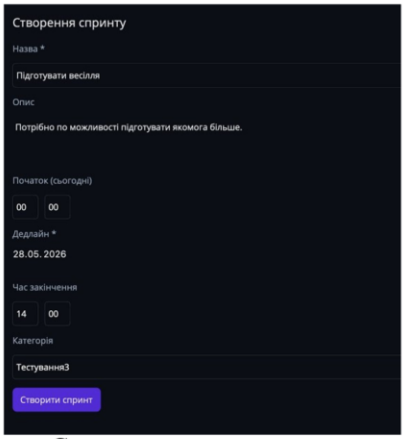
Призначення завданню категорії за допомогою «Drag-and-Drop»

13

ЕКРАННІ ФОРМИ



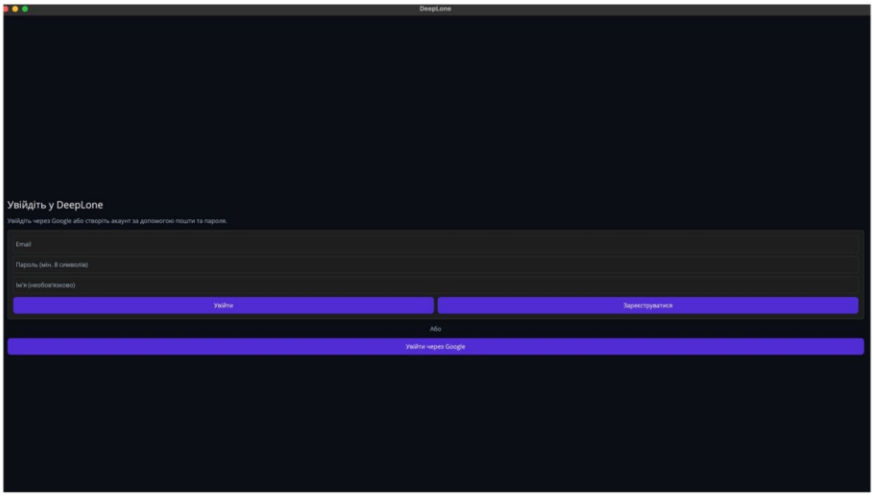
Меню аналітики



Створення нового спринту.

14

ВІДЕОДЕМОНСТРАЦІЯ ЗАСТОСУНКУ



АПРОБАЦІЯ РЕЗУЛЬТАТІВ ДОСЛІДЖЕННЯ

1. Новохацький І.О Худік Б.О Розробка застосунку для менеджменту часу на основі фреймворку .NET MAUI. Матеріали II Всеукраїнська науково-технічна конференція «Виклики та рішення в програмній інженерії», 26 листопада 2025 року, Київ, Державний університет інформаційно-комунікаційних технологій. Збірник тез. К.: ДУІКТ, 2025. С.498-99.
2. Новохацький І.О., Худік Б.О. Проектування та розробка програмного забезпечення на базі екосистеми .NET 9. Матеріали VII Всеукраїнську науково-технічну конференцію «Застосування програмного забезпечення в інформаційно-комунікаційних технологіях», 23 квітня 2026 року, Київ, Збірник тез. К.: ДУІКТ, 2025. С.453-456.

ВИСНОВКИ

1. Проведено аналіз методів персонального тайм-менеджменту та обґрунтовано доцільність адаптації Agile для індивідуального планування. Це дозволило перейти від лінійних списків до гнучких спринтів, знижуючи когнітивне навантаження на користувача.
2. Сформовано технічні вимоги до системи, що передбачають ізольовану клієнт-серверну архітектуру.
3. Обґрунтовано вибір технологій: екосистема .NET та СУБД PostgreSQL з Entity Framework Core. Такий стек розробки забезпечив швидкодію клієнта та транзакційну цілісність даних.
4. Спроєктовано трирівневу архітектуру й базу даних, реалізовано алгоритми управління завданнями, спринтами та розрахунку продуктивності.
5. Здійснено програмну реалізацію серверної логіки з фоновими чергами й SignalR та клієнтського застосунку з Drag-and-Drop.
6. Забезпечено безпеку даних завдяки гібридній системі авторизації через токени JWT та сервіси Google OAuth2.
7. Проведено комплексне тестування системи, яке підтвердило коректність мережевої взаємодії, точність аналітики та швидкодію. Успішна верифікація всіх модулів повністю доводить готовність застосунку до експлуатації.

ДОДАТОК Б. ЛІСТИНГ ОСНОВНИХ МОДУЛІВ

Program.cs

```
using DeepLone.api.Config;
using DeepLone.api.Entities;
using DeepLone.api.Push;
using Microsoft.AspNetCore.Authentication.JwtBearer;
using Microsoft.EntityFrameworkCore;
using Microsoft.IdentityModel.Tokens;
using Scalar.AspNetCore;
using System.Text;

DotEnv.Load(Path.Combine(AppContext.BaseDirectory, ".env"));
DotEnv.Load(Path.Combine(Directory.GetCurrentDirectory(),
".env"));

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddDbContext<AppDbContext>(options =>

options.UseNpgsql(builder.Configuration.GetConnectionString("
DefaultConnection")));

builder.Services.AddSingleton<IFcmSender, FcmSender>();
builder.Services.AddHttpClient();
builder.Services.AddSingleton<IWnsSender, WnsSender>();
builder.Services.AddSingleton<IApnsSender, ApnsSender>();
builder.Services.AddScoped<IPushScheduler, PushScheduler>();
if (!builder.Environment.IsEnvironment("Testing"))
{
   builder.Services.AddHostedService<PushJobWorker>();
}

builder.Services.AddSignalR();

builder.Services.AddAuthentication(JwtBearerDefaults.Authentic
ationScheme)
   .AddJwtBearer(options =>
   {
      var signingKey = builder.Configuration["Jwt:SigningKey"];
      if (string.IsNullOrWhiteSpace(signingKey))
      {
         throw new InvalidOperationException("Missing config:
Jwt:SigningKey");
      }

      options.TokenValidationParameters = new
TokenValidationParameters
      {
         ValidateIssuer = false,
         ValidateAudience = false,
         ValidateLifetime = true,
         ValidateIssuerSigningKey = true,
         IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes(signingKey)),
         ClockSkew = TimeSpan.FromMinutes(2)
      };

      options.Events = new JwtBearerEvents
      {
         OnMessageReceived = context =>
         {
            var accessToken =
context.Request.Query["access_token"];
            var path = context.HttpContext.Request.Path;
            if (!string.IsNullOrEmpty(accessToken) &&
               path.StartsWithSegments("/hubs"))
            {
               context.Token = accessToken;
            }

            return Task.CompletedTask;
         }
      };
   });

builder.Services.AddAuthorization();

builder.Services.AddControllers();

builder.Services.AddOpenApi();

var app = builder.Build();

using (var scope = app.Services.CreateScope())
{
   var db =
scope.ServiceProvider.GetRequiredService<AppDbContext>();
   var pending = await
db.Database.GetPendingMigrationsAsync();
   var applied = await db.Database.GetAppliedMigrationsAsync();
   Console.WriteLine($"[DB] Applied migrations: {string.Join(",
", applied)}");
   Console.WriteLine($"[DB] Pending migrations: {string.Join(",
", pending)}");

   db.Database.Migrate();

   try
   {
      static async Task<bool> HasColumnAsync(AppDbContext
db, string table, string column)
      {
         var list = await db.Database.SqlQueryRaw<string>(
            """
            SELECT column_name
            FROM information_schema.columns
            WHERE table_schema = 'public' AND table_name =
{0} AND column_name = {1}
            """,
            table,
            column).ToListAsync();
         return list.Count > 0;
      }

      if (!await HasColumnAsync(db, "Sprints", "CategoryId"))
      {
         Console.WriteLine("[DB] Patching: add
Sprints.CategoryId");
```

```
        await db.Database.ExecuteSqlRawAsync("""ALTER
TABLE public."Sprints" ADD COLUMN "CategoryId" uuid
NULL;""");
        await db.Database.ExecuteSqlRawAsync("""CREATE
INDEX IF NOT EXISTS "IX_Sprints_CategoryId" ON
public."Sprints" ("CategoryId");""");
        await db.Database.ExecuteSqlRawAsync(
            """ALTER TABLE public."Sprints" ADD
CONSTRAINT "FK_Sprints_Categories_CategoryId" FOREIGN
KEY ("CategoryId") REFERENCES public."Categories" ("Id")
ON DELETE SET NULL;""");
    }

    if (!await HasColumnAsync(db, "Tasks", "CreatedAt"))
    {
        Console.WriteLine("[DB] Patching: add
Tasks.CreatedAt");
        await db.Database.ExecuteSqlRawAsync("""ALTER
TABLE public."Tasks" ADD COLUMN "CreatedAt" timestamp
with time zone NOT NULL DEFAULT (NOW());""");
    }

    if (!await HasColumnAsync(db, "Tasks", "CompletedAt"))
    {
        Console.WriteLine("[DB] Patching: add
Tasks.CompletedAt");
        await db.Database.ExecuteSqlRawAsync("""ALTER
TABLE public."Tasks" ADD COLUMN "CompletedAt"
timestamp with time zone NULL;""");
    }

    if (!await HasColumnAsync(db, "Tasks", "DeletedAt"))
    {
        Console.WriteLine("[DB] Patching: add
Tasks.DeletedAt");
        await db.Database.ExecuteSqlRawAsync("""ALTER
TABLE public."Tasks" ADD COLUMN "DeletedAt" timestamp
with time zone NULL;""");
        await db.Database.ExecuteSqlRawAsync("""CREATE
INDEX IF NOT EXISTS "IX_Tasks_UserId_DeletedAt" ON
public."Tasks" ("UserId", "DeletedAt");""");
    }

    if (!await HasColumnAsync(db, "Tasks", "Reaction"))
    {
        Console.WriteLine("[DB] Patching: add Tasks.Reaction");
        await db.Database.ExecuteSqlRawAsync("""ALTER
TABLE public."Tasks" ADD COLUMN "Reaction" text
NULL;""");
```

TaskController.cs

```
using DeepLone.api.Auth;
using DeepLone.api.DTOs;
using DeepLone.api.Entities;
using DeepLone.api.Push;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace DeepLone.api.Controllers;

[ApiController]
// Explicit lowercase route so clients and reverse proxies always
match (not api/Tasks).
```

```
    }

    var sprintCols = await db.Database.SqlQueryRaw<string>(
        """
        SELECT column_name
        FROM information_schema.columns
        WHERE table_schema = 'public' AND table_name =
'Sprints'
        ORDER BY ordinal_position
        """)
      .ToListAsync();
    Console.WriteLine($"[DB] public.\"Sprints\" columns:
{string.Join(", ", sprintCols)}");

    var taskCols = await db.Database.SqlQueryRaw<string>(
        """
        SELECT column_name
        FROM information_schema.columns
        WHERE table_schema = 'public' AND table_name =
'Tasks'
        ORDER BY ordinal_position
        """)
      .ToListAsync();
    Console.WriteLine($"[DB] public.\"Tasks\" columns:
{string.Join(", ", taskCols)}");
  }
  catch (Exception ex)
  {
    Console.WriteLine($"[DB] Column check failed:
{ex.GetType().Name}: {ex.Message}");
  }
}

app.MapOpenApi();
app.MapScalarApiReference();

app.UseAuthentication();
app.UseAuthorization();

app.MapControllers();
app.MapHub<NotificationsHub>("/hubs/notifications");

app.Run();

public partial class Program
{
}
```

```
[Route("api/tasks")]
[Authorize]
public class TasksController : ControllerBase
{
  private readonly AppDbContext _context;
  private readonly IPushScheduler _pushScheduler;

  // Впровадження залежностей (Dependency Injection).
  // ASP.NET сам передасть сюди підключення до нашої БД.
  public TasksController(AppDbContext context, IPushScheduler
pushScheduler)
  {
    _context = context;
    _pushScheduler = pushScheduler;
  }
```

```
    [HttpGet]
    public async
Task<ActionResult<IEnumerable<AppTaskResponseDto>>>
GetTasks()
    {
        var userId = UserContext.GetRequiredUserId(User);

        var tasks = await _context.Tasks.Where(t => t.UserId ==
userId && t.DeletedAt == null).ToListAsync();
        var taskIds = tasks.Select(t => t.Id).ToList();

        var allTimeLogs = await _context.TimeLogs.Where(l =>
taskIds.Contains(l.TaskId)).ToListAsync();

        var response = tasks.Select(t =>
        {
            var taskLogs = allTimeLogs.Where(l => l.TaskId ==
t.Id).ToList();

            TimeSpan totalTime = TimeSpan.Zero;
            bool isRunning = false;

            foreach (var log in taskLogs)
            {
                if (log.EndedAt.HasValue)
                {
                    totalTime += log.EndedAt.Value - log.StartedAt;
                }
                else
                {
                    isRunning = true;
                    totalTime += DateTime.UtcNow - log.StartedAt;
                }
            }

            return new AppTaskResponseDto
            {
                Id = t.Id,
                Title = t.Title,
                Description = t.Description,
                Reaction = t.Reaction,
                CategoryId = t.CategoryId,
                SprintId = t.SprintId,
                PlannedStart = t.PlannedStart,
                Deadline = t.Deadline,
                CreatedAt = t.CreatedAt,
                CompletedAt = t.CompletedAt,
                Status = t.Status,
                Priority = t.Priority,

                TrackedTime =
$"{(int)totalTime.TotalHours:D2}:{totalTime.Minutes:D2}:{total
Time.Seconds:D2}",
                IsTimerRunning = isRunning
            };
        }).ToList();

        return Ok(response);
    }

    [HttpGet("{id:guid}")]
    public async Task<ActionResult<AppTaskResponseDto>>
GetTask(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var t = await _context.Tasks
            .AsNoTracking()
            .FirstOrDefaultAsync(x => x.Id == id && x.UserId ==
userId && x.DeletedAt == null);

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

        var taskLogs = await _context.TimeLogs.Where(l =>
l.TaskId == t.Id).ToListAsync();

        TimeSpan totalTime = TimeSpan.Zero;
        var isRunning = false;
        foreach (var log in taskLogs)
        {
            if (log.EndedAt.HasValue)
                totalTime += log.EndedAt.Value - log.StartedAt;
            else
            {
                isRunning = true;
                totalTime += DateTime.UtcNow - log.StartedAt;
            }
        }

        return Ok(new AppTaskResponseDto
        {
            Id = t.Id,
            Title = t.Title,
            Description = t.Description,
            Reaction = t.Reaction,
            CategoryId = t.CategoryId,
            SprintId = t.SprintId,
            PlannedStart = t.PlannedStart,
            Deadline = t.Deadline,
            CreatedAt = t.CreatedAt,
            CompletedAt = t.CompletedAt,
            Status = t.Status,
            Priority = t.Priority,
            TrackedTime =
$"{(int)totalTime.TotalHours:D2}:{totalTime.Minutes:D2}:{total
Time.Seconds:D2}",
            IsTimerRunning = isRunning
        });
    }


    [HttpGet("trash")]
    public async
Task<ActionResult<IEnumerable<AppTaskResponseDto>>>
GetTrashTasks()
    {
        var userId = UserContext.GetRequiredUserId(User);

        var tasks = await _context.Tasks
            .AsNoTracking()
            .Where(t => t.UserId == userId && t.DeletedAt != null)
            .OrderByDescending(t => t.DeletedAt)
            .ToListAsync();
        var taskIds = tasks.Select(t => t.Id).ToList();
        var allTimeLogs = taskIds.Count == 0
            ? new List<TimeLog>()
            : await _context.TimeLogs.Where(l =>
taskIds.Contains(l.TaskId)).ToListAsync();
```

```
        var response = tasks.Select(t =>
        {
            var taskLogs = allTimeLogs.Where(l => l.TaskId ==
t.Id).ToList();
            TimeSpan totalTime = TimeSpan.Zero;
            var isRunning = false;
            foreach (var log in taskLogs)
            {
                if (log.EndedAt.HasValue)
                    totalTime += log.EndedAt.Value - log.StartedAt;
                else
                {
                    isRunning = true;
                    totalTime += DateTime.UtcNow - log.StartedAt;
                }
            }

            return new AppTaskResponseDto
            {
                Id = t.Id,
                Title = t.Title,
                Description = t.Description,
                Reaction = t.Reaction,
                CategoryId = t.CategoryId,
                SprintId = t.SprintId,
                PlannedStart = t.PlannedStart,
                Deadline = t.Deadline,
                CreatedAt = t.CreatedAt,
                CompletedAt = t.CompletedAt,
                DeletedAt = t.DeletedAt,
                Status = t.Status,
                Priority = t.Priority,
                TrackedTime =
$"{(int)totalTime.TotalHours:D2}:{totalTime.Minutes:D2}:{total
Time.Seconds:D2}",
                IsTimerRunning = isRunning
            };
        }).ToList();

        return Ok(response);
    }


    [HttpDelete("trash/all")]
    public async Task<IActionResult> PurgeAllTrashTasks()
    {
        var userId = UserContext.GetRequiredUserId(User);

        var trashed = await _context.Tasks
            .Where(t => t.UserId == userId && t.DeletedAt != null)
            .ToListAsync();

        if (trashed.Count == 0)
            return NoContent();

        foreach (var t in trashed)
            await _pushScheduler.CancelTaskRemindersAsync(userId,
t.Id, HttpContext.RequestAborted);

        _context.Tasks.RemoveRange(trashed);
        await _context.SaveChangesAsync();

        return NoContent();
    }

    [HttpPost]
    public async Task<ActionResult<AppTaskResponseDto>>
CreateTask([FromBody] AppTaskCreateDto dto)
    {
        var userId = UserContext.GetRequiredUserId(User);

        if (dto.CategoryId.HasValue)
        {
            var categoryOk = await _context.Categories.AnyAsync(c
=> c.Id == dto.CategoryId && c.UserId == userId);
            if (!categoryOk) return BadRequest("Категорію не
знайдено");
        }

        if (dto.SprintId.HasValue)
        {
            var sprintOk = await _context.Sprints.AnyAsync(s => s.Id
== dto.SprintId && s.UserId == userId);
            if (!sprintOk) return BadRequest("Спринт не знайдено");
        }


        var newTask = new AppTask
        {
            Title = dto.Title,
            Description = dto.Description,
            CategoryId = dto.CategoryId,
            SprintId = dto.SprintId,
            PlannedStart = dto.PlannedStart,
            Deadline = dto.Deadline.ToUniversalTime(),
            Status = dto.Status,
            Priority = dto.Priority,

            UserId = userId
        };



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

        await
_pushScheduler.EnsureNextTaskReminderAsync(userId,
newTask, HttpContext.RequestAborted);

        var responseDto = new AppTaskResponseDto
        {
            Id = newTask.Id,
            Title = newTask.Title,
            Description = newTask.Description,
            Reaction = newTask.Reaction,
            CategoryId = newTask.CategoryId,
            SprintId = newTask.SprintId,
            PlannedStart = newTask.PlannedStart,
            Deadline = newTask.Deadline,
            CreatedAt = newTask.CreatedAt,
            CompletedAt = newTask.CompletedAt,
            Status = newTask.Status,
            Priority = newTask.Priority
        };

        return CreatedAtAction(nameof(GetTasks), new { id =
newTask.Id }, responseDto);
    }
```

```
    public async Task<IActionResult> SetReaction(Guid id,
[FromBody] TaskReactionDto dto)
    {
        var userId = UserContext.GetRequiredUserId(User);
        var task = await _context.Tasks.FindAsync(id);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.UserId != userId) return Forbid();
        if (task.DeletedAt != null) return NotFound("Задачу не
знайдено");

        var reaction = string.IsNullOrWhiteSpace(dto.Reaction) ?
null : dto.Reaction.Trim();
        if (reaction != null && reaction.Length > 16)
            return BadRequest("Реакція занадто довга");

        task.Reaction = reaction;
        await _context.SaveChangesAsync();
        return NoContent();
    }
/
    [HttpPost("{id}/start")]
    public async Task<ActionResult<TimeLogResponseDto>>
StartTaskTimer(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var task = await _context.Tasks.FindAsync(id);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.UserId != userId) return Forbid();


        var activeLog = await _context.TimeLogs
            .FirstOrDefaultAsync(t => t.TaskId == id && t.EndedAt
== null);

        if (activeLog != null) return BadRequest("Таймер для цієї
задачі вже працює!");
        var newLog = new TimeLog
        {
            TaskId = id,
            StartedAt = DateTime.UtcNow // Обов'язково в UTC!
        };

        _context.TimeLogs.Add(newLog);

        // Автоматично міняємо статус задачі на 1 (In Progress)
        task.Status = 1;

        await _context.SaveChangesAsync();

        return Ok(new TimeLogResponseDto
        {
            Id = newLog.Id,
            TaskId = newLog.TaskId,
            StartedAt = newLog.StartedAt,
            EndedAt = newLog.EndedAt
        });
    }


    [HttpPost("{id}/stop")]
    public async Task<ActionResult<TimeLogResponseDto>>
StopTaskTimer(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);


        var task = await _context.Tasks.FindAsync(id);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.UserId != userId) return Forbid();

        var activeLog = await
_context.TimeLogs.FirstOrDefaultAsync(t => t.TaskId == id &&
t.EndedAt == null);

        if (activeLog == null) return BadRequest("Немає активного
таймера для цієї задачі");

        activeLog.EndedAt = DateTime.UtcNow;

        await _context.SaveChangesAsync();

        return Ok(new TimeLogResponseDto
        {
            Id = activeLog.Id,
            TaskId = activeLog.TaskId,
            StartedAt = activeLog.StartedAt,
            EndedAt = activeLog.EndedAt
        });
    }
    [HttpGet("{id}/time")]
    public async
Task<ActionResult<TimeSummaryResponseDto>>
GetTaskTimeSummary(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var task = await _context.Tasks.FindAsync(id);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.UserId != userId) return Forbid();


        var logs = await _context.TimeLogs
            .Where(t => t.TaskId == id)
            .ToListAsync();

        TimeSpan totalTime = TimeSpan.Zero;
        bool isRunning = false;

        foreach (var log in logs)
        {
            if (log.EndedAt.HasValue)
            {
                // Якщо таймер був зупинений, додаємо різницю між
кінцем і початком
                totalTime += log.EndedAt.Value - log.StartedAt;
            }
            else
            {
                // Якщо таймер працює ПРЯМО ЗАРАЗ, рахуємо час
від старту до поточної секунди
                isRunning = true;
                totalTime += DateTime.UtcNow - log.StartedAt;
            }
        }

        string formatted =
$"{(int)totalTime.TotalHours:D2}:{totalTime.Minutes:D2}:{total
Time.Seconds:D2}";
```

```
        return Ok(new TimeSummaryResponseDto
        {
            TaskId = id,
            TotalMinutes = Math.Round(totalTime.TotalMinutes, 2),
            FormattedTime = formatted,
            IsRunning = isRunning
        });
    }

    [HttpPut("{id}")]
    public async Task<IActionResult> UpdateTask(Guid id,
[FromBody] AppTaskCreateDto dto)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var task = await _context.Tasks.FindAsync(id);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.UserId != userId) return Forbid();
        if (task.DeletedAt != null) return NotFound("Задачу не
знайдено");

        if (dto.CategoryId.HasValue)
        {
            var categoryOk = await _context.Categories.AnyAsync(c
=> c.Id == dto.CategoryId && c.UserId == userId);
            if (!categoryOk) return BadRequest("Категорію не
знайдено");
        }

        if (dto.SprintId.HasValue)
        {
            var sprintOk = await _context.Sprints.AnyAsync(s => s.Id
== dto.SprintId && s.UserId == userId);
            if (!sprintOk) return BadRequest("Спринт не знайдено");
        }


        task.Title = dto.Title;
        task.Description = dto.Description;
        if (task.Status != 2 && dto.Status == 2)
            task.CompletedAt = DateTime.UtcNow;
        else if (task.Status == 2 && dto.Status != 2)
            task.CompletedAt = null;

        task.Status = dto.Status;
        task.Priority = dto.Priority;

        task.SprintId = dto.SprintId;
        task.CategoryId = dto.CategoryId;

        передаються)
        task.PlannedStart = dto.PlannedStart?.ToUniversalTime();
        task.Deadline = dto.Deadline.ToUniversalTime();

        await _context.SaveChangesAsync();

        await
_pushScheduler.EnsureNextTaskReminderAsync(userId, task,
HttpContext.RequestAborted);

        return NoContent(); // Повертає 204 (Успіх без тіла
відповіді)
```

MauiProgram.cs

```
    }


    public async Task<IActionResult> DeleteTask(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var task = await _context.Tasks.FirstOrDefaultAsync(t =>
t.Id == id && t.UserId == userId);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.DeletedAt != null) return NoContent();

        task.DeletedAt = DateTime.UtcNow;
        await _context.SaveChangesAsync();

        await _pushScheduler.CancelTaskRemindersAsync(userId,
id, HttpContext.RequestAborted);

        return NoContent();
    }

    [HttpPost("{id}/restore")]
    public async Task<IActionResult> RestoreTask(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var task = await _context.Tasks.FirstOrDefaultAsync(t =>
t.Id == id && t.UserId == userId);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.DeletedAt == null) return BadRequest("Задача не в
кошику");

        task.DeletedAt = null;
        await _context.SaveChangesAsync();

        await
_pushScheduler.EnsureNextTaskReminderAsync(userId, task,
HttpContext.RequestAborted);

        return NoContent();
    }

    [HttpDelete("{id}/purge")]
    public async Task<IActionResult> PurgeTask(Guid id)
    {
        var userId = UserContext.GetRequiredUserId(User);

        var task = await _context.Tasks.FirstOrDefaultAsync(t =>
t.Id == id && t.UserId == userId);
        if (task == null) return NotFound("Задачу не знайдено");
        if (task.DeletedAt == null) return BadRequest("Спочатку
перемістіть задачу в кошик");

        await _pushScheduler.CancelTaskRemindersAsync(userId,
id, HttpContext.RequestAborted);

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

        return NoContent();
    }
}
```

```
using Deeplone.client.Configuration;
using Deeplone.client.Services;
using Deeplone.client.Services.Diagnostics;
using Deeplone.client.Services.Notifications;
using Deeplone.client.ViewModels;
using Deeplone.client.Views;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Logging;
using Microsoft.Maui.Controls.Hosting;
using Microsoft.Maui.Hosting;

namespace Deeplone.client;

public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        CrashLogger.Init();

        // Try common layouts, but don't rely on the working
directory being the repo root.
        var cwd = Directory.GetCurrentDirectory();
        var baseDir = AppContext.BaseDirectory;
        var appData = FileSystem.AppDataDirectory;
        var cache = FileSystem.CacheDirectory;
        // Preferred locations for sandboxed apps
(MacCatalyst/packaged):
        DotEnvLoader.LoadIfPresent(Path.Combine(appData,
".env"));
        DotEnvLoader.LoadIfPresent(Path.Combine(cache, ".env"));

#if !(MACCATALYST || IOS)
        // Developer-friendly local layouts (non-sandboxed):
        DotEnvLoader.LoadIfPresent(Path.Combine(cwd,
"DeepLone.api", ".env"));
        DotEnvLoader.LoadIfPresent(Path.Combine(cwd, ".env"));
        DotEnvLoader.LoadIfPresent(Path.Combine(baseDir,
".env"));
        DotEnvLoader.LoadFromParents(cwd, ".env",
Path.Combine("DeepLone.api", ".env"));
        DotEnvLoader.LoadFromParents(baseDir, ".env",
Path.Combine("DeepLone.api", ".env"));
#endif
        BundledClientConfig.TryApplyDefaults();

        try
        {
            var cid =
Environment.GetEnvironmentVariable("Google__ClientId") ??
string.Empty;
            CrashLogger.Write("Env", $"Google__ClientId
len={cid.Length}\nLoaded:\n{string.Join("\n",
DotEnvLoader.LoadedEnvFiles.Select(p => "- " + p))}");
        }
        catch { }

        var builder = MauiApp.CreateBuilder();
        builder
            .UseMauiApp<App>()
            .ConfigureFonts(fonts =>
            {
                fonts.AddFont("OpenSans-Regular.ttf",
"OpenSansRegular");
                fonts.AddFont("OpenSans-Semibold.ttf",
"OpenSansSemibold");
            });

        builder.Services.AddSingleton<IThemeService,
ThemeService>();
        builder.Services.AddSingleton<IUserProfileStore,
UserProfileStore>();
        builder.Services.AddSingleton<IAuthTokenStore,
AuthTokenStore>();
        builder.Services.AddSingleton<StickyPositionStore>();
        builder.Services.AddSingleton<INotificationsClient,
NotificationsClient>();
        builder.Services.AddSingleton<IDevicePushRegistrar,
DevicePushRegistrar>();
        builder.Services.AddSingleton<ISystemNotificationService,
SystemNotificationService>();

builder.Services.AddSingleton<IDeepLinkNavigationService,
DeepLinkNavigationService>();
        builder.Services.AddSingleton<IRootPageService,
RootPageService>();
        builder.Services.AddSingleton<IGoogleAuthService,
GoogleAuthService>();


builder.Services.AddTransient<AuthorizationMessageHandler>();

builder.Services.AddTransient<UnauthorizedRedirectHandler>();
        builder.Services.AddHttpClient<ApiClient>(c => {
c.BaseAddress = new Uri(ApiConstants.BaseUrl); })

.AddHttpMessageHandler<AuthorizationMessageHandler>()

.AddHttpMessageHandler<UnauthorizedRedirectHandler>();

        builder.Services.AddSingleton<WorkspaceViewModel>();
        builder.Services.AddSingleton<AppShell>();
        builder.Services.AddSingleton<App>();

        builder.Services.AddTransient<LoginViewModel>();
        builder.Services.AddTransient<LoginPage>();

        builder.Services.AddSingleton<CategoriesViewModel>();
        builder.Services.AddSingleton<SprintsViewModel>();
        builder.Services.AddTransient<SprintsPage>();
        builder.Services.AddTransient<SettingsViewModel>();
        builder.Services.AddSingleton<StatisticsViewModel>();
        builder.Services.AddTransient<StatisticsPage>();
        builder.Services.AddTransient<TrashViewModel>();

builder.Services.AddTransient<TaskCreateFormViewModel>();
        builder.Services.AddTransient<TaskCreateModalPage>();

builder.Services.AddTransient<CategoryCreateViewModel>();

builder.Services.AddTransient<CategoryCreateModalPage>();
        builder.Services.AddTransient<SprintCreateViewModel>();
        builder.Services.AddTransient<SprintCreateModalPage>();
        builder.Services.AddTransient<SettingsModalPage>();

#if DEBUG
        builder.Logging.AddDebug();
#endif

        return builder.Build();
    }
```