Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп’ютерних наук

Допущено до захисту
Завідувач кафедри комп’ютерних наук,
доктор технічних наук, професор

Андрій БОНДАРЧУК
«01» червня 2026 р.

КВАЛІФІКАЦІЙНА РОБОТА

на здобуття освітнього ступеня «бакалавр»
Спеціальність 122 Комп’ютерні науки
Освітня програма 122.00.01 Інформатика

ТЕМА: МОДЕЛЮВАННЯ МІКРОСЕРВІСНОЇ АРХІТЕКТУРИ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ СИСТЕМИ УПРАВЛІННЯ ЧАСОМ (TIMETRACKER)

Виконав
Студент групи ІНб-2-22-4.0д
Колесниченко Іван Вячеславович
_____
(підпис)

Науковий керівник
доцент, к.п.н., ст.
Лілія Варченко-Троценко
_____
(підпис)

Київ – 2026

Київський столичний університет імені Бориса Грінченка
Факультет інформаційних технологій та математики
Кафедра комп'ютерних наук

«Затверджую»
Завідувач кафедри комп’ютерних наук,
доктор технічних наук, професор
Андрій БОНДАРЧУК
« 31 » 10 2025 р.

ЗАВДАННЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ БАКАЛАВРА
«Моделювання мікросервісної архітектури програмного забезпечення для системи управління часом (timetracker)»

Виконавець студент групи ІНб-2-22-4.0д,
спеціальності 122 Комп’ютерні науки,
освітньої програми 122.00.01 Інформатика
Колесниченко Іван В’ячеславович

1. Вихідні дані: Законодавство та нормативно-правові акти України у сфері інформаційних технологій; аналітичні дані ринку ПЗ для управління проєктами (Grand View Research); наукові публікації у галузі мікросервісної архітектури та веб-розробки; офіційна документація NestJS, Vue 3, TypeORM, PostgreSQL, Docker та JWT.
2. Основні завдання: здійснити огляд предметної області та порівняльний аналіз існуючих рішень для обліку робочого часу; сформулювати функціональні та нефункціональні вимоги; обрати та обґрунтувати технологічний стек і архітектурний патерн; спроектувати мікросервісну архітектуру з патерном API Gateway та структуру БД; розробити сервіси Gateway, Auth Service, Tracking Service та клієнтський SPA; провести модульне та функціональне тестування; оформити пояснювальну записку.
3. Пояснювальна записка: Обсяг – до 66 стор. формату А4 з дотриманням вимог стандарту і методичних рекомендацій кафедри.
4. Графічні матеріали: діаграма варіантів використання, діаграма компонентів архітектури, діаграма послідовності взаємодії компонентів, ER-діаграма БД, файлова структура проєкту, порівняльні діаграми аналізу рішень, скріншоти застосунку.
5. Додатки: лістинги конфігураційних файлів Docker Compose та ключових модулів системи.
6. Строк подання роботи на кафедру: «01» 06 2026 р.

Науковий керівник
к.п.н, доцент,
_____ Варченко-Троценко Л.О.
31.10.2025

Виконавець:
_____Колесниченко І.В.
31.10.2025

КАЛЕНДАРНИЙ ПЛАН

№	Етап виконання роботи	Термін виконання	Примітка
1	Формування теми кваліфікаційної роботи бакалавра та її затвердження	31.10.25	виконано
2	Огляд предметної області, порівняльний аналіз існуючих рішень (Toggl Track, Clockify, Jira) та написання першого розділу пояснювальної записки	01.11.25 – 11.01.26	виконано
3	Формулювання функціональних та нефункціональних вимог, проєктування мікросервісної архітектури з патерном API Gateway та вибір технологічного стеку	26.01.26 – 15.03.26	виконано
4	Проєктування ER-діаграми БД, розробка Gateway-сервісу та Auth Service з JWT-автентифікацією	16.03.26 – 31.03.26	виконано
5	Реалізація Tracking Service, розробка клієнтського SPA на Vue 3 та інтеграція компонентів системи	01.04.26 – 15.04.26	виконано
6	Написання другого розділу пояснювальної записки (архітектура, технологічний стек, проєктування)	16.04.26 – 30.04.26	виконано
7	Написання третього розділу пояснювальної записки (реалізація та тестування)	01.05.26 – 15.05.26	виконано
8	Подання роботи на перевірку, проходження антиплагіату, внесення правок наукового керівника	25.05.26 – 12.06.26	виконано
9	Захист кваліфікаційної роботи	16.06.26	виконано

«Затверджую»

Науковий керівник
к.п.н, доцент, Варченко-Троценко Л.О.

Індивідуальний план склав
Колесниченко І.В.

РЕФЕРАТ

Пояснювальна записка: 58 с., 3 розділи, 14 рисунків, 10 таблиць, 36 джерела.

Метою кваліфікаційної роботи є розробка та реалізація веб-застосунку TimeTracker для обліку робочого часу та управління задачами на основі мікросервісної архітектури з патерном API Gateway, що забезпечує ізоляцію компонентів, безпечну автентифікацію та відтворюване середовище розгортання.

Об'єкт дослідження –процес проєктування та реалізації програмного забезпечення для обліку робочого часу та управління задачами.

Предмет дослідження – методи та засоби моделювання мікросервісної архітектури веб-застосунку системи управління часом.

У першому розділі проведено огляд предметної області та порівняльний аналіз трьох популярних рішень –Toggl Track, Clockify та Jira –за сімома критеріями. Виявлено спільний системний недолік: жодне з рішень не забезпечує оптимального балансу між простотою використання, повноцінним управлінням задачами та централізованим обліком часу в єдиній системі без сторонніх плагінів.

У другому розділі сформульовано сім функціональних (Ф-1…Ф-7) та шість нефункціональних вимог (НФ-1…НФ-6), обрано мікросервісну архітектуру з патерном API Gateway, обґрунтовано технологічний стек: NestJS + TypeScript для серверних сервісів, Vue 3 + TypeScript для клієнтського інтерфейсу, TypeORM + PostgreSQL для роботи з даними, Docker + docker-compose для контейнеризації.

У третьому розділі описано реалізацію чотирьох компонентів системи (Gateway, Auth Service, Tracking Service, Frontend SPA), проведено модульне тестування Auth Service (47 тестів, Jest) з покриттям функцій 100% та операторів понад 90%, тестування Pinia-сховища (9 тестів, Vitest) з покриттям statements 97.14%, а також функціональне тестування 24 сценаріїв для всіх ендпоінтів системи. Верифіковано відповідність усім визначеним функціональним та нефункціональним вимогам.

Розроблена система може бути розгорнута на будь-якій машині з встановленим Docker однією командою без додаткового налаштування середовища та використана як основа для подальшого розширення функціональності.

Ключові слова: веб-застосунок, мікросервісна архітектура, облік робочого часу, API GATEWAY, NESTJS, VUE 3, TYPEORM, POSTGRESQL, DOCKER, JWT, автентифікація, модульне тестування.

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

CRUD (Create, Read, Update, Delete) – базові операції роботи з даними: створення, читання, оновлення, видалення

DI (Dependency Injection) – впровадження залежностей

ER-діаграма (Entity-Relationship diagram) – діаграма сутностей та зв'язків ESM (ECMAScript Modules) – стандарт модульної системи JavaScript

IDE (Integrated Development Environment) – інтегроване середовище розробки JWT (JSON Web Token) – стандарт відкритого токена доступу на основі JSON ORM (Object-Relational Mapping) – об'єктно-реляційне відображення

REST (Representational State Transfer) – архітектурний стиль взаємодії компонентів розподіленого застосунку

SPA (Single Page Application) – односторінковий застосунок

XSS (Cross-Site Scripting) – міжсайтовий скриптинг, тип вразливості веб застосунків

САГР (Compound Annual Growth Rate) – середньорічний темп зростання БД – база даних

ЗМІСТ

ВСТУП

Стрімкий розвиток інформаційних технологій та поширення дистанційних і гібридних моделей роботи зумовили зростання попиту на ефективні програмні рішення для обліку робочого часу та управління проєктами. Глобальний ринок відповідного програмного забезпечення оцінювався у понад 7 млрд доларів США у 2023 році та демонструє стабільне зростання із прогнозованим середньорічним темпом понад 10% до 2030 року. Основними рушіями цього зростання є цифровізація бізнес-процесів, масове впровадження Agile методологій та збільшення кількості розподілених команд, для яких прозорість витрат робочого часу є критично важливою умовою ефективного управління.

Попри широкий вибір існуючих рішень, більшість із них мають суттєві обмеження. Спеціалізовані інструменти обліку часу (Toggl Track, Clockify) не забезпечують повноцінного управління задачами та ієрархічними структурами проєктів. Системи управління проєктами (Jira) потребують додаткових платних плагінів для якісного time tracking та мають високий поріг входу для нетехнічних користувачів. Жодне з поширених рішень не поєднує в єдиній інтуїтивній платформі централізований облік часу, управління задачами та сучасну масштабовану архітектуру без залежності від сторонніх модулів.

Окремої уваги заслуговує архітектурний аспект проблеми. Більшість наявних рішень побудована на монолітній архітектурі, яка ускладнює незалежне масштабування компонентів, підвищує ризики при оновленні окремих модулів та обмежує гнучкість розгортання. Мікросервісна архітектура, яка набула широкого поширення в індустрії, дозволяє усунути ці недоліки шляхом декомпозиції системи на незалежні сервіси з чітко визначеними зонами відповідальності. Однак питання практичної реалізації мікросервісного підходу для систем обліку робочого часу залишається недостатньо дослідженим у контексті навчальних та малих комерційних проєктів.

Таким чином, розробка веб-застосунку для обліку робочого часу на основі мікросервісної архітектури є актуальним науково-практичним завданням, що

поєднує теоретичне моделювання архітектурних рішень із їх практичною реалізацією.

Об'єктом дослідження є процес проєктування та реалізації програмного забезпечення для обліку робочого часу та управління задачами.

Предметом дослідження є методи та засоби моделювання мікросервісної архітектури веб-застосунку системи управління часом.

Метою кваліфікаційної роботи є розробка та реалізація веб-застосунку TimeTracker для обліку робочого часу та управління задачами на основі мікросервісної архітектури з патерном API Gateway, що забезпечує ізоляцію компонентів, безпечну автентифікацію та відтворюване середовище розгортання.

Для досягнення поставленої мети визначено такі завдання дослідження:

1. провести огляд предметної області та аналіз існуючих веб-рішень для обліку робочого часу і управління задачами за визначеною системою критеріїв;
2. визначити функціональні та нефункціональні вимоги до проектованої системи з урахуванням виявлених недоліків існуючих рішень;
3. обґрунтувати вибір технологічного стеку та інструментарію розробки на основі порівняльного аналізу альтернатив;
4. спроектувати мікросервісну архітектуру системи з патерном API Gateway, визначити структуру бази даних та потоки даних між компонентами;
5. спроектувати користувацький інтерфейс відповідно до принципів адаптивного дизайну та евристик зручності використання;
6. реалізувати систему у вигляді контейнеризованого стеку з чотирьох незалежних сервісів та провести її комплексне тестування;
7. верифікувати відповідність реалізованої системи визначеним функціональним та нефункціональним вимогам.

Методи дослідження. У роботі використано методи порівняльного аналізу для оцінювання існуючих рішень, методи системного аналізу для формування вимог та проєктування архітектури, методи об'єктно-орієнтованого та

компонентного проєктування для розробки структури системи, а також методи модульного та функціонального тестування для верифікації реалізованого рішення.

Практична значущість роботи полягає у розробці повністю функціональної мікросервісної системи обліку робочого часу з відкритим вихідним кодом, яка може бути розгорнута на будь-якій машині з встановленим Docker без додаткового налаштування середовища та використана як основа для подальшого розширення функціональності.

Структура роботи. Кваліфікаційна робота складається зі вступу, трьох розділів, загальних висновків та списку використаних джерел. Загальний обсяг основного тексту становить 66 сторінок. Робота містить 12 таблиць, 13 рисунків та 33 джерел у списку використаної літератури.

РОЗДІЛ 1

АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ ТА ІСНУЮЧИХ РІШЕНЬ

1.1 Огляд існуючих веб-рішень для обліку робочого часу та управління задачами

Сучасний ринок програмних рішень для обліку робочого часу (time tracking) та управління проєктами є важливою складовою галузі інформаційних технологій, особливо в умовах розвитку віддаленої роботи та гнучких моделей зайнятості. Згідно з аналітичними дослідженнями компанії Grand View Research [1], глобальний ринок програмного забезпечення для управління проєктами оцінювався у понад 7 млрд доларів США у 2023 році та демонструє стабільне зростання із прогнозованим середньорічним темпом понад 10% до 2030 року. Основними драйверами розвитку є цифровізація бізнес-процесів, поширення Agile-методологій та зростання кількості розподілених команд.

Облік робочого часу є ключовим елементом ефективного управління проєктами, оскільки дозволяє оцінювати продуктивність працівників, контролювати витрати ресурсів та оптимізувати планування задач. Сучасні системи time tracking інтегруються з інструментами управління проєктами, комунікаційними платформами та аналітичними сервісами, формуючи єдину екосистему управління діяльністю команди. Важливими тенденціями є автоматизація збору даних, використання аналітики продуктивності та інтеграція з хмарними сервісами.

Серед найбільш популярних веб-рішень у цій сфері можна виділити такі системи, як Toggl Track [2], Clockify [3] та Jira [4]. Вони представляють різні підходи до організації обліку часу та управління задачами, що дозволяє проаналізувати їх переваги та обмеження.

Сервіс Toggl Track є одним із найбільш відомих інструментів для обліку робочого часу. Він надає користувачам можливість фіксувати витрачений час у режимі реального часу, створювати проєкти та задачі, а також формувати

детальні звіти. Основною перевагою системи є простота використання та інтуїтивно зрозумілий інтерфейс. Крім того, сервіс підтримує інтеграцію з великою кількістю сторонніх інструментів, таких як Trello та Asana. Однак функціональність управління командою та складними проєктами є обмеженою, що робить його менш придатним для великих організацій.

Іншим популярним рішенням є Clockify – безкоштовний сервіс для відстеження часу, який підтримує необмежену кількість користувачів і проєктів. Платформа дозволяє створювати тайм-логи, генерувати звіти та аналізувати використання робочого часу. Перевагою є доступність та відсутність обмежень у базовій версії. Водночас інтерфейс системи може бути менш зручним для нових користувачів,

а деякі розширені функції (наприклад, розширена аналітика або інтеграції) доступні лише у платних тарифах.

Система Jira від компанії Atlassian є більш складним і потужним інструментом, орієнтованим на управління проєктами в IT-командах. Вона підтримує роботу з задачами, спринтами, беклогами та дозволяє інтегрувати облік часу безпосередньо у процес розробки. Jira широко використовується в Agile

середовищі та забезпечує високий рівень гнучкості налаштувань. Проте система має складний інтерфейс і вимагає часу для навчання, що може бути недоліком для невеликих команд або окремих користувачів.

Аналіз існуючих рішень показує, що, незважаючи на широкий функціонал, більшість сучасних систем мають певні обмеження. До них можна віднести: надлишкову складність інтерфейсу для початківців, відсутність чіткої інтеграції між управлінням задачами та обліком часу, а також залежність від платних підписок для отримання повного функціоналу. Крім того, багато сервісів орієнтовані або на індивідуальне використання, або на великі корпоративні команди, залишаючи поза увагою потреби невеликих груп користувачів.

Таким чином, існує потреба у створенні веб-застосунку, який поєднує простоту використання, базову функціональність управління проєктами та

ефективний облік робочого часу в межах єдиної системи.

Саме це завдання вирішується у рамках даної кваліфікаційної роботи шляхом розробки системи TimeTracker.

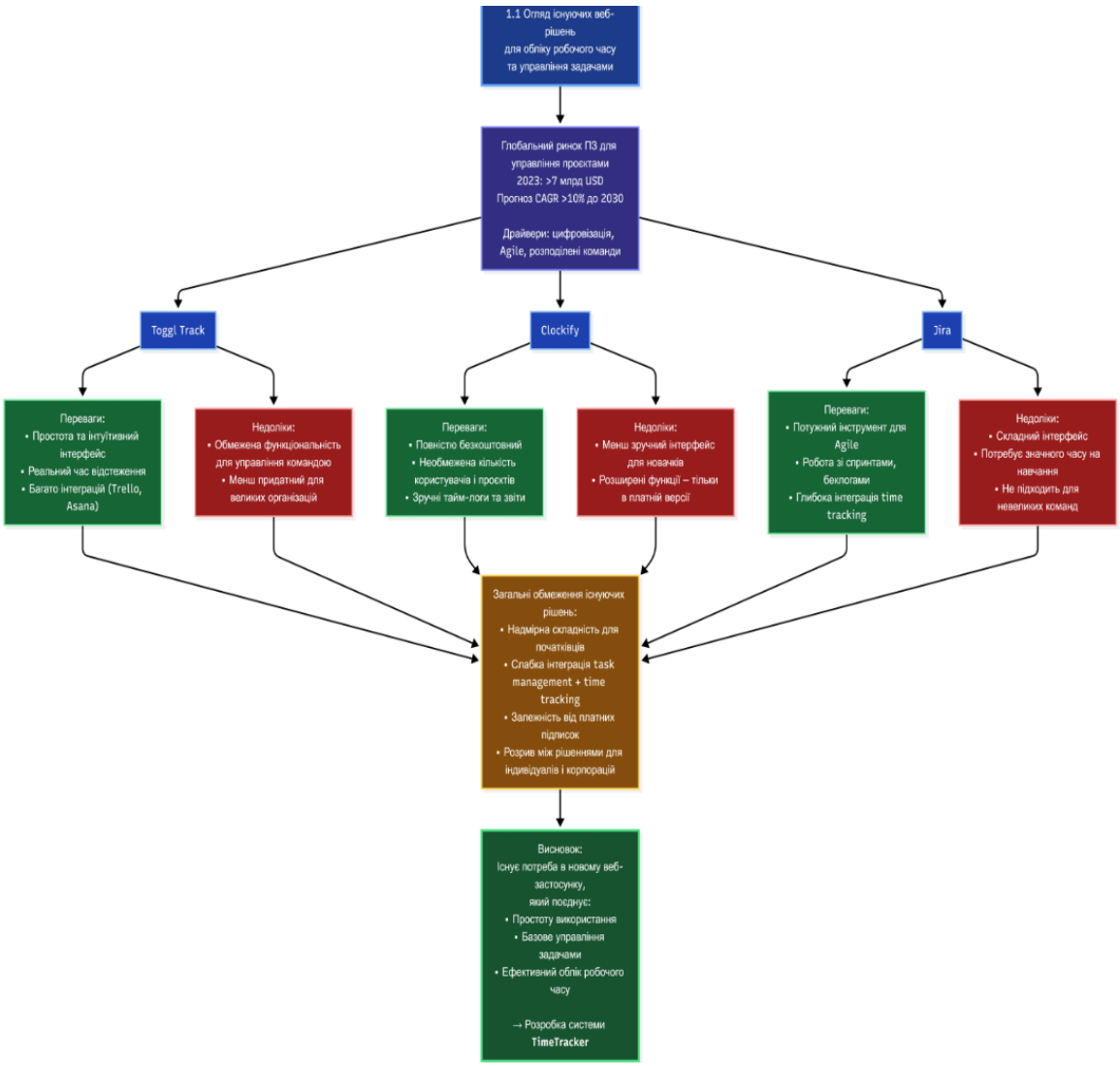

Рис. 1.1. Загальна структура розглянутих веб-рішень для обліку робочого часу та управління задачами

1.2 Опис і аналіз критеріїв порівняння веб-застосунків для обліку робочого часу та управління задачами

Для проведення об’єктивного порівняльного аналізу існуючих веб-застосунків для обліку робочого часу та управління задачами необхідно сформувати чітку систему критеріїв оцінювання. Вибір критеріїв є ключовим методологічним етапом, оскільки саме вони забезпечують обґрунтованість

висновків щодо сильних і слабких сторін рішень, а також доцільності розробки нового програмного продукту.

Відповідно до принципів системного аналізу, критерії мають бути незалежними, однозначно інтерпретованими та такими, що охоплюють основні аспекти взаємодії користувача з системою. При їх формуванні було враховано потреби двох основних груп користувачів:

Професійні користувачі (розробники, менеджери проєктів, IT-команди) для яких важливі точність обліку, гнучке управління задачами, потужна аналітика та інтеграції.

Звичайні користувачі та невеликі команди для яких пріоритетом є простота використання, швидке освоєння та відсутність надмірної складності. Додатково враховано рекомендації у сфері UX-дизайну, зокрема 10 heuristics usability Якоба Нільсена з Nielsen Norman Group [5], які акцентують увагу на зрозумілості системи, мінімізації когнітивного навантаження, видимості статусу та ефективності виконання задач. Також використано принципи task-oriented design, згідно з якими інтерфейс повинен бути орієнтований на реальні задачі користувача, а не лише на демонстрацію функціоналу.

Критерій 1. Наявність веб-інтерфейсу та адаптивності (так/ні) Сучасні користувачі працюють на різних пристроях. Повноцінний адаптивний (responsive) веб-інтерфейс забезпечує зручну роботу як з комп’ютера, так і з мобільних пристроїв без втрати функціональності. Відсутність адаптивності суттєво знижує доступність системи в умовах віддаленої та гібридної роботи.

Критерій 2. Формат обліку часу (ручний / таймер / змішаний) Цей критерій визначає гнучкість фіксації часу. Ручний формат дозволяє вводити дані постфактум, таймер – відстежувати час у реальному часі, а змішаний підхід

поєднує обидва способи. Згідно з практиками 2026 року, найбільш ефективним вважається саме змішаний формат з підтримкою автоматизованого трекінгу.

Критерій 3. Наявність системи управління проєктами і задачами (так/ні) Ефективний облік часу неможливий без контексту. Можливість створення

проєктів, задач, підзадач та ієрархічних структур дозволяє прив'язувати час до конкретних активностей і підвищує точність планування.

Критерій 4. Наявність аналітики та звітності (так/ні) Система повинна надавати зручні інструменти для аналізу продуктивності: звіти за проєктами, користувачами, періодами, бюджетом тощо. Відсутність якісної аналітики обмежує можливість прийняття обґрунтованих управлінських рішень.

Критерій 5. Інтеграція з іншими сервісами (так/ні або кількість інтеграцій) Сучасні інструменти працюють у єдиній цифровій екосистемі. Інтеграція зі Slack, GitHub, Trello, Asana, Google Calendar, Microsoft Teams та іншими сервісами дозволяє автоматизувати процеси та зменшити ручну роботу.

Критерій 6. Поріг входу та складність інтерфейсу (низький / середній / високий) Критерій оцінює, наскільки швидко новий користувач може почати продуктивну роботу. Низький поріг входу (проста навігація, інтуїтивний дизайн) є критично важливим для невеликих команд і фрилансерів, тоді як високий –часто характерний для потужних корпоративних рішень.

Критерій 7. Модель доступу / цінова політика (безкоштовна / freemium / платна) Фінансова доступність суттєво впливає на вибір. Безкоштовні та freemium моделі привабливі для стартапів і невеликих команд, тоді як повністю платні рішення зазвичай пропонують розширений функціонал, підтримку та безпеку рівня enterprise.

Сукупність цих критеріїв дозволяє комплексно оцінити веб-застосунки з трьох ключових перспектив:

1. Доступність та зручність використання (критерії 1, 6, 7);
2. Функціональні можливості (критерії 2, 3, 4);
3. Інтеграція в цифрову екосистему (критерій 5).

Таким чином, сформована система критеріїв створює надійну основу для порівняльного аналізу Toggl Track, Clockify та Jira. Вона допомагає виявити їх ключові недоліки (надмірна складність, фрагментованість функціоналу або обмеження безкоштовної версії) і обґрунтовує необхідність розробки власного веб-застосунку TimeTracker, який поєднуватиме простоту, достатній функціонал

управління задачами та ефективний облік робочого часу для різних категорій користувачів.

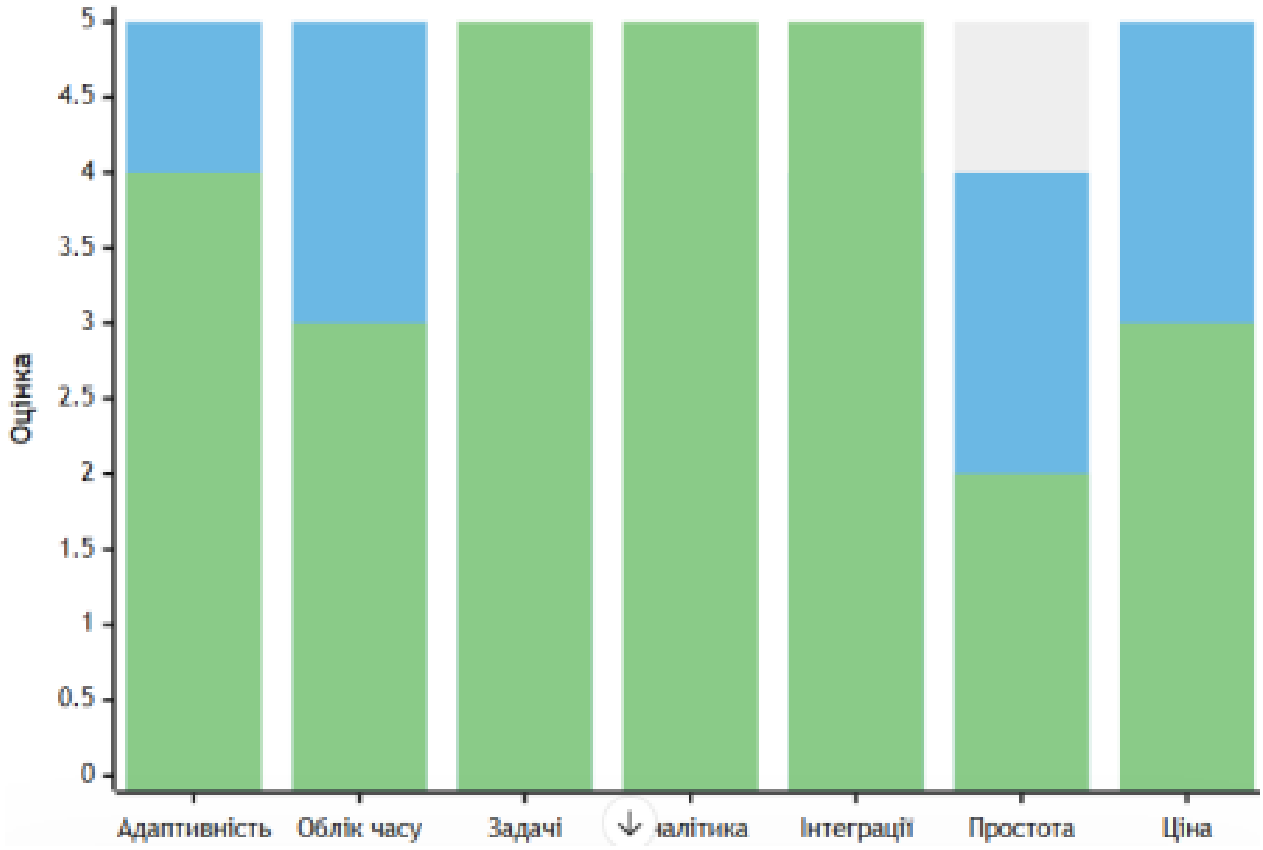

Рис 1.2. Порівняльна радарна діаграма веб-застосунків для обліку робочого часу за визначеними критеріями

1.3 Оцінка існуючих веб-застосунків для обліку робочого часу та управління задачами

На основі критеріїв, визначених у підрозділі 1.2 (адаптивний веб-інтерфейс, формат обліку часу, управління проєктами і задачами, аналітика та звітність, інтеграції з сервісами, простота використання, цінова доступність), проведено порівняльну оцінку трьох популярних веб-застосунків у сфері обліку робочого

часу та управління задачами: Toggl Track, Clockify та Jira (від Atlassian). Ці сервіси обрано через їхню широку популярність серед фрілансерів, малого бізнесу та команд розробки, а також значну кількість користувачів і відгуків. Для оцінювання застосовано шкалу змішаного типу:

для бінарних критеріїв – «Так» / «Ні»;

для якісних характеристик – градація «низький / середній / високий»; для кількісних аспектів (наприклад, рівень інтеграцій) – умовна оцінка за шкалою 1–5 (де 5 – максимум).

Результати оцінювання зведено до таблиці 1.1.

Таблиця 1.1

Порівняльна оцінка веб-застосунків для обліку робочого часу

Критерій	**Toggl Track**	**Clockify**	**Jira**
Адаптивний веб-інтерфейс	Так	Так	Так
Формат обліку часу	Змішаний	Змішаний	Частково
Управління проєктами і задачами	Частково	Так	Так
Аналітика та звітність	Так	Так	Так
Інтеграції з сервісами	Високий	Середній	Високий
Простота використання	Високий	Середній	Низький
Цінова доступність	Середня (freemium)	Висока (free)	Середня (freemium)

Аналіз даних таблиці 1.1 дозволяє зробити детальні узагальнені висновки щодо сильних і слабких сторін кожного рішення.

Сервіс Toggl Track позиціонується як зручний інструмент для індивідуальних користувачів і невеликих команд [6]. Він пропонує інтуїтивний інтерфейс з підтримкою таймера в реальному часі, ручного введення, автоматичного трекінгу активності на десктопі (Timeline), перегляду в календарі, списку та timesheet. Користувачі можуть призначати час на проєкти, клієнти, задачі та теги, а також використовувати Pomodoro-таймер [7]. Аналітика включає кастомні звіти про продуктивність, прибутковість,

завантаженість команди та тенденції (Insights). Сервіс підтримує понад 100 інтеграцій (включаючи Jira, Asana, Slack, Salesforce) через браузерні розширення та API [13] [14]. Відповідно до актуальних даних на 2026 рік, безкоштовний план підтримує до 5 користувачів з базовим функціоналом, план Starter – від $9 за користувача на місяць (при річній оплаті) або $10 при щомісячній, Premium – від $18–20 відповідно. Перевага – висока простота використання та швидкість роботи. Основний недолік – обмежене повноцінне управління задачами: відсутня глибока ієрархія проєктів (sub-tasks обмежені), спринти чи agile-борди, що знижує ефективність у складних командних проєктах з багатьма залежностями.

Платформа Clockify акцентує увагу на максимальній доступності та необмеженому використанні для великих команд [8]. Безкоштовний план дозволяє необмежену кількість користувачів, проєктів, задач і трекінгу часу (таймер, timesheet, kiosk, auto-tracker, календар). Підтримуються задачі як суб

проєкти, клієнти, теги, ставки (billable rates), базова аналітика (звіти про витрати часу, costs, productivity). Є підтримка invoicing, attendance, scheduling [9]. Відповідно до офіційних даних, платні плани мають таку структуру: Basic – $3.99/користувач на місяць (при річній оплаті), Standard – $5.49, Pro – $7.99, Enterprise – $11.99. За оцінками незалежних аналітиків, Clockify є одним із найбільш конкурентоспроможних рішень за співвідношенням ціни та функціоналу [15]. Однак інтерфейс вважається менш інтуїтивним і трохи

«перевантаженим» порівняно з Toggl, а розширені можливості аналітики та інтеграцій часто вимагають переходу на платну підписку. Управління задачами присутнє, але не таке глибоке, як у спеціалізованих PM-системах. Система Jira від Atlassian є потужним рішенням для управління проєктами, особливо в agile-середовищі (Scrum, Kanban boards, backlogs, sprints, roadmaps) [10]. Вона забезпечує повноцінну ієрархію задач (epics, stories, subtasks), workflows, автоматизацію, звітність (burndown charts, velocity тощо). Облік часу реалізовано через вбудоване логування worklogs (estimate vs actual), але це не центральна функція – для повноцінних timesheets, approvals, детальної аналітики

та billable rates зазвичай потрібні додаткові Marketplace-застосунки, зокрема Tempo Timesheets. Інтеграцій тисячі (Marketplace + API), включаючи Slack, GitHub, Confluence. Відповідно до актуальних цін станом на 2026 рік, хмарна версія Jira пропонує план Free (до 10 користувачів), Standard – $7.91/користувач на місяць, Premium – $14.54/користувач на місяць. Водночас слід враховувати, що у 2025 році Atlassian підвищила ціни на 5–20%, а реальна вартість володіння для середніх команд суттєво зростає через необхідність додаткових Marketplace-застосунків, вартість яких зазвичай складає $3–10 за користувача на місяць. Головні недоліки: високий поріг входу, складний інтерфейс (особливо для нетехнічних користувачів), перевантаженість налаштуваннями та необхідність додаткових плагінів для якісного time tracking. Загальний аналіз показує, що жоден із розглянутих веб-застосунків не забезпечує оптимального балансу між простотою використання, повноцінним управлінням задачами та ефективним централізованим обліком робочого часу в єдиній інтуїтивній системі.

Toggl Track – простий і зручний для індивідуального/невеликого командного трекінгу часу, але обмежений у глибокому управлінні проєктами та задачами (немає повноцінних agile-інструментів)

Clockify – максимально доступний (unlimited users у free), з хорошим базовим управлінням задачами, але компромісний за UX та глибиною аналітики без платної підписки.

Jira – найбільш потужний для управління проєктами та agile-процесами, але складний, перевантажений і з фрагментованим обліком часу, що часто вимагає сторонніх застосунків.

Рис 1.3. Порівняльна оцінка веб-застосунків

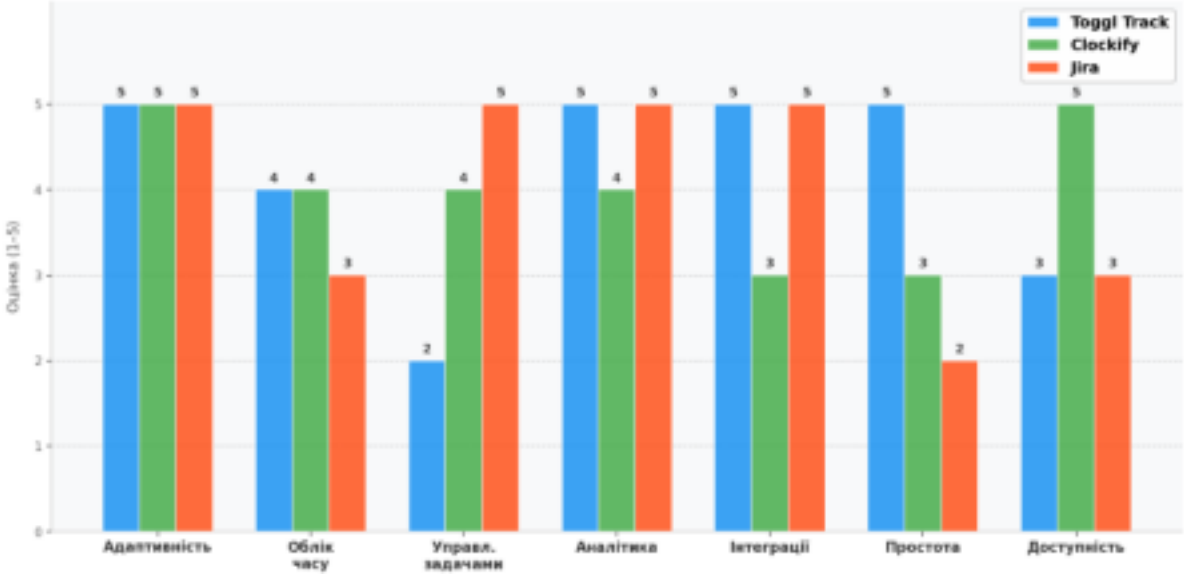

Рис 1.4. Оцінювання веб-застосунків за критеріями (групована стовпчаста діаграма)

Результати, відображені на рисунках 1.3 (порівняльна радарна діаграма) та 1.4 (стовпчаста діаграма оцінювання за критеріями), підтверджують ці висновки. Радарна діаграма ілюструє зміщення акцентів кожного сервісу: Toggl - у бік простоти, Clockify – доступності, Jira – функціональності. Стовпчаста діаграма підкреслює дисбаланс: системи з високою функціональністю (Jira) мають нижчу зручність, а прості інструменти (Toggl) – обмежений task management. Проведений аналіз виявив системні недоліки існуючих рішень для різних категорій користувачів (фрілансери, малі команди, середній бізнес, розробницькі відділи):

надмірна складність інтерфейсу та високий поріг входу (Jira); недостатній функціонал повноцінного управління проєктами та задачами в єдиній системі (Toggl Track);

обмеження безкоштовних версій щодо розширеної аналітики, інтеграцій чи approvals (Clockify у деяких аспектах);

фрагментація функціоналу (time tracking часто відокремлений від task management або вимагає додаткових модулів).

Ці недоліки формують чіткі вимоги до нового веб-застосунку: поєднання інтуїтивного дизайну, централізованого обліку часу (таймер + manual + auto) та повноцінного управління проєктами/задачами (ієрархія, статуси, залежності) без необхідності сторонніх плагінів.

Таким чином, розробка системи TimeTracker є обґрунтованою та актуальною. Вона спрямована на усунення зазначених обмежень шляхом створення єдиної платформи з: простим та інтуїтивним інтерфейсом, доступним для різних категорій користувачів. Повноцінним управлінням проєктами, задачами та ієрархією; ефективним обліком робочого часу (змішаний формат) з потужною, але не перевантаженою аналітикою.

Проєктування архітектури, вибір технологій та реалізація такого рішення розглядаються у наступних розділах дипломної роботи.

1.4 Висновки до розділу 1

У першому розділі проведено огляд предметної області, сформовано систему критеріїв оцінювання та виконано порівняльний аналіз існуючих веб-застосунків для обліку робочого часу та управління задачами.

Встановлено, що глобальний ринок програмного забезпечення для управління проєктами перевищує 7 млрд доларів США та демонструє стабільне зростання з прогнозованим середньорічним темпом понад 10% до 2030 року. Основними тенденціями є автоматизація збору даних про витрачений час, інтеграція з хмарними сервісами та поширення Agile-методологій у розподілених командах. Проаналізовано три найбільш популярні рішення у цій сфері – Toggl Track, Clockify та Jira, – які представляють різні підходи до організації обліку часу та управління задачами.

Сформовано систему з семи критеріїв порівняльного оцінювання, що охоплюють три ключові перспективи: доступність та зручність використання (адаптивний веб-інтерфейс, простота використання, цінова доступність), функціональні можливості (формат обліку часу, управління проєктами і задачами, аналітика та звітність) та інтеграцію в цифрову екосистему. При формуванні критеріїв враховано потреби двох цільових груп користувачів – професійних фахівців та невеликих команд – а також рекомендації Nielsen Norman Group щодо юзабіліті інтерфейсів.

За результатами порівняльного аналізу встановлено, що кожне з розглянутих рішень має суттєві обмеження: Toggl Track забезпечує високу простоту використання, але має обмежений функціонал управління проєктами та відсутність agile-інструментів; Clockify пропонує необмежений безкоштовний доступ, але поступається за якістю UX та глибиною аналітики; Jira є найпотужнішою системою для управління проєктами, однак відрізняється складним інтерфейсом, високим порогом входу та фрагментованим обліком часу, що вимагає підключення сторонніх Marketplace-застосунків.

Виявлені системні недоліки існуючих рішень – фрагментація функціоналу

між трекінгом часу та управлінням задачами, надмірна складність або недостатня функціональність – обґрунтовують доцільність розробки власного веб застосунку TimeTracker. Визначено ключові вимоги до нового рішення: інтуїтивний інтерфейс з низьким порогом входу, централізований змішаний облік робочого часу та повноцінне управління проєктами і задачами в єдиній системі без необхідності сторонніх інтеграцій. Реалізація цих вимог розглядається у наступних розділах роботи.

РОЗДІЛ 2

ПРОЄКТУВАННЯ СИСТЕМИ TIMETRACKER

2.1 Визначення основних критеріїв для вибору інструментарію розробки веб-застосунку

Вибір інструментарію для розробки веб-застосунку є важливим архітектурним рішенням, що визначає трудомісткість розробки, зручність підтримки, продуктивність кінцевого продукту та можливості його розширення. Для обґрунтованого вибору необхідно попередньо визначити систему критеріїв, яким має відповідати інструментарій у контексті конкретного проєкту.

При формуванні критеріїв враховувались характеристики проєкту: мікросервісна архітектура з чотирьох незалежних компонентів, розробка однією особою, необхідність контейнеризації через Docker, а також технічні вимоги, визначені у підрозділі 2.1. Окремо враховувались вимога НФ-3 щодо мікросервісної архітектури та вимога НФ-5 щодо контейнеризації – інструментарій повинен органічно вписуватись у цю модель розгортання.

За результатами аналізу сформульовано наступні критерії вибору інструментарію.

Критерій 1. Відповідність мікросервісній архітектурі. Інструментарій серверної частини повинен підтримувати модульну організацію коду, що дозволяє чітко розмежовувати зони відповідальності між сервісами. Використання монолітних фреймворків для проєкту з мікросервісною архітектурою є надмірним і суперечить принципу відповідності засобів задачі.

Критерій 2. Підтримка TypeScript. Оскільки система складається з кількох взаємодіючих сервісів із власними контрактами API, суворий контроль типів є критично важливим для забезпечення узгодженості інтерфейсів між компонентами та зниження кількості помилок під час інтеграції.

Критерій 3. Поріг входу та крива навчання. Для забезпечення ефективної одноосібної розробки перевага надається інструментам із низьким порогом

входу, широкою офіційною документацією та активною спільнотою. Це знижує час на освоєння інструменту і прискорює вирішення нетипових завдань на всіх рівнях стеку.

Критерій 4. Продуктивність та масштабованість. Обраний технологічний стек повинен забезпечувати достатню продуктивність для одночасної обробки запитів від кількох користувачів. Серверна частина повинна підтримувати неблокуючу асинхронну модель виконання, а клієнтська – мінімальний розмір бандлу для швидкого завантаження інтерфейсу.

Критерій 5. Інтеграція з ORM та базами даних. Інструментарій серверної частини повинен мати зрілі засоби для роботи з реляційними базами даних: підтримку міграцій, декларативний опис сутностей та зручний API для

виконання запитів. Це безпосередньо впливає на швидкість реалізації вимог Ф-2 – Ф-6.

Критерій 6. Підтримка контейнеризації. Інструментарій повинен мати офіційні або широко використовувані базові Docker-образи, підтримувати багатоетапну збірку (multi-stage build) для мінімізації розміру production-образів та коректно функціонувати в середовищі docker-compose.

Критерій 7. Доступність інструментів розробки. Середовище розробки повинно бути доступним безкоштовно, підтримувати підсвічування синтаксису, автодоповнення та налагодження коду для всіх використовуваних мов і фреймворків. Наявність розширень для IDE суттєво прискорює розробку у багатосервісному середовищі.

Визначені сім критеріїв охоплюють технічні, організаційні та практичні аспекти вибору інструментарію і забезпечують об'єктивну основу для порівняльного аналізу альтернатив у наступному підрозділі.

2.2 Вибір інструментарію для розробки веб-застосунку

На основі критеріїв, визначених у підрозділі 2.3, проведено порівняльний аналіз альтернативних варіантів інструментарію для кожної технологічної категорії: серверний фреймворк, мова програмування, frontend-фреймворк,

ORM, система управління базами даних, засоби контейнеризації та середовище розробки.

Серверний фреймворк. Для реалізації backend-сервісів (Gateway, Auth Service, Tracking Service) розглядались NestJS та Express.js. Express.js є мінімалістичним unopinionated фреймворком з найнижчим порогом входу та максимальною гнучкістю, однак не надає вбудованої структури для організації коду – відсутність модульної архітектури, dependency injection і вбудованої підтримки TypeScript суттєво ускладнює підтримку при зростанні кодової бази [22]. NestJS побудований поверх Express.js і надає opinionated архітектуру, натхненну

Angular: модулі, контролери, сервіси, вбудований DI-контейнер та першокласну підтримку мікросервісів, [23]. NestJS забезпечує на 40% швидший розвиток складних застосунків порівняно з Express.js завдяки структурі та суворому типізуванню. З огляду на критерії 1, 2 і 3 для проєкту обрано NestJS.

Мова програмування. Для серверної та клієнтської частини розглядались TypeScript та JavaScript. JavaScript є нативною мовою Node.js і браузера, однак відсутність статичної типізації у мікросервісній системі з кількома точками інтеграції суттєво підвищує ризик помилок на стику сервісів. TypeScript надає суворий контроль типів, виявлення помилок на етапі компіляції, підтримку декораторів (які є основою NestJS-архітектури) та значно кращу підтримку IDE [24]. З огляду на критерії 1 і 2 для всіх сервісів проєкту обрано TypeScript.

Frontend-фреймворк. Для реалізації клієнтського SPA розглядались Vue 3, React та Angular. Angular є надмірним для проєкту такого масштабу – його архітектура і кількість концепцій (RxJS, NgModule, декоратори) виправдані лише для великих enterprise-команд. React є найпопулярнішим фреймворком (~85 млн завантажень на тиждень) з потужною екосистемою, однак має складнішу криву навчання через hooks-модель та необхідність вибору між великою кількістю бібліотек управління станом [25].

Vue 3 зберігає перевагу у менших розмірах бандлу та швидкості гідратації, що робить його оптимальним вибором для застосунків, де важлива швидкість

завантаження. Monterail Vue 3 з runtime-only збіркою має розмір ~23 KB, тоді як React потребує мінімум ~47 KB. Composition API та офіційна бібліотека стану Pinia забезпечують зручний і типобезпечний підхід до управління глобальним станом. З огляду на критерії 3 і 4 обрано Vue 3.

На рисунку 2.1 подано порівняння розглянутих інструментів за визначеними критеріями.

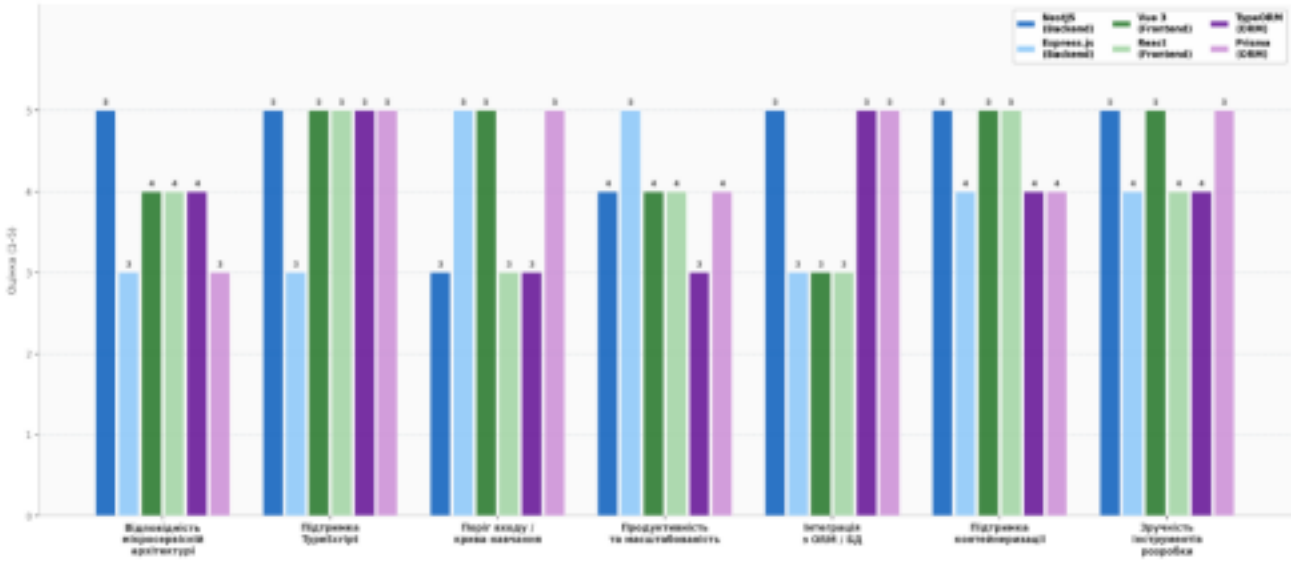

Рис 2.1. Порівняння інструментарію за критеріями вибору

ORM та система управління базами даних. Для роботи з базами даних розглядались TypeORM, Prisma та Drizzle. Drizzle є найшвидшим сучасним ORM (~7.4 KB), але вимагає глибоких знань SQL та не надає вбудованих засобів синхронізації схеми, що ускладнює початкову розробку [26]. Prisma надає найкращий developer experience з автогенерованим типобезпечним клієнтом та schema-first підходом, однак вимагає окремого кроку prisma generate та додаткового Rust-рушія [27]. TypeORM є де-факто стандартом для NestJS: decorator-based entities органічно поєднуються з NestJS-модулями, опція synchronize: true автоматично застосовує схему при старті (зручно для розробки), а каскадне видалення налаштовується декларативно на рівні сутностей. TypeORM є першокласним громадянином екосистеми NestJS та забезпечує найменший overhead при інтеграції. З огляду на критерії 5 і 1 обрано TypeORM з PostgreSQL.

PostgreSQL обрано серед реляційних СУБД як найбільш надійне рішення з відкритим кодом, що підтримує JSON, складні запити та добре задокументовано у поєднанні з TypeORM.

Засоби контейнеризації. Docker та docker-compose є галузевим стандартом для контейнеризації мікросервісних застосунків. Всі сервіси мають офіційні базові образи (node:alpine, postgres), а docker-compose дозволяє декларативно описати весь стек і запустити його однією командою docker-compose up -d. З огляду на вимогу НФ-5 обрано Docker + docker-compose.

Середовище розробки. Для написання та налагодження коду розглядались Visual Studio Code, WebStorm та Vim. WebStorm є потужним IDE з вбудованою підтримкою TypeScript та Node.js, однак є платним. Vim потребує значного часу на налаштування та не надає зручних засобів візуальної відладки. Visual Studio Code є безкоштовним редактором з відкритим кодом, що надає повноцінну підтримку TypeScript через вбудований Language Server, розширення для NestJS, Docker та Vue 3 (Volar), вбудований термінал та інтеграцію з Git [28]. З огляду на критерій 7 обрано Visual Studio Code.

Результати порівняльного аналізу інструментарію за визначеними критеріями зведені у таблиці 2.2.

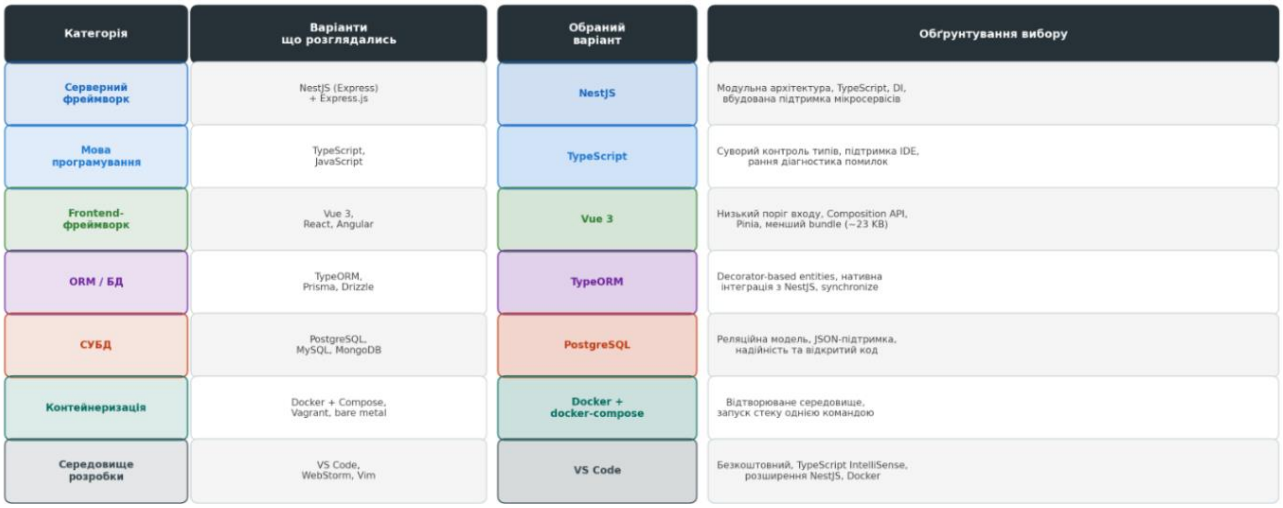

Рис 2.2. Підсумкова таблиця обраного технологічного стеку TimeTracker

Таблиця 2.2

Порівняльний аналіз інструментарію для розробки веб-застосунку TimeTracker

Категорія	**Варіанти що розглядались**	**Обраний варіант**	**Обґрунтування вибору**
Серверний фреймворк	NestJS, Express.js	NestJS	Модульна архітектура, DI, TypeScript, мікросервіси
Мова програмування	TypeScript, JavaScript	TypeScript	Суворі типи, декоратори, підтримка IDE
Frontend фреймворк	Vue 3, React, Angular	Vue 3	Низький поріг входу, Pinia, менший бандл
ORM	TypeORM, Prisma, Drizzle	TypeORM	Нативна інтеграція з NestJS, decorator entities
СУБД	PostgreSQL, MySQL, MongoDB	PostgreSQL	Реляційна модель, надійність, відкритий код
Контейнеризація	Docker + Compose, bare metal	Docker + docker compose	Відтворюване середовище, єдина команда запуску
Середовище розробки	VS Code, WebStorm, Vim	Visual Studio Code	TypeScript LSP, Volar, Docker

Таким чином, для розробки веб-застосунку TimeTracker обрано стек технологій: NestJS + TypeScript для серверних сервісів, Vue 3 + TypeScript для клієнтського інтерфейсу, TypeORM + PostgreSQL для роботи з даними, Docker + docker compose для контейнеризації та Visual Studio Code як середовище розробки. Обраний стек повністю відповідає всім семи визначеним критеріям, забезпечує органічну взаємодію між компонентами, TypeScript-типізацію наскрізь усього стеку та відтворюване середовище розгортання.

2.3 Проектування користувацького інтерфейсу

Проектування користувацького інтерфейсу є ключовим етапом розробки веб застосунку, що передує безпосередній реалізації. Якість інтерфейсу визначає ефективність взаємодії користувача з системою та безпосередньо впливає на практичну цінність продукту. Процес проектування UI охоплює аналіз вимог,

розробку інформаційної архітектури, створення вайрфреймів і макетів, а також формування концепції компонентного складу застосунку.

2.3.1 Принципи проектування інтерфейсу

Проектування інтерфейсу TimeTracker ґрунтується на евристиках зручності використання, сформульованих Якобом Нільсеном. Десять евристик зручності використання Нільсена для проектування користувацьких інтерфейсів широко застосовуються як загальні правила для прийняття дизайнерських рішень з часу їх первинного введення у 1994 році. У контексті застосунку TimeTracker особливо значущими є такі принципи: видимість стану системи (відображення активного проєкту та поточної задачі), узгодженість і стандарти (єдина мова компонентів по всьому застосунку), а також запобігання помилкам (підтвердження деструктивних дій, зокрема видалення проєкту з каскадним видаленням даних).

Шість фаз проектування UI охоплюють: аналіз цілей і очікувань користувачів, дизайн зі створенням вайрфреймів і прототипів, розробку та перетворення дизайну на придатний до використання веб-продукт, розгортання з тестуванням, оцінювання ефективності через зворотний зв'язок і аналітику, а також ітерацію на основі результатів оцінювання. У рамках дипломного проєкту реалізовано фази аналізу, дизайну та розробки.

2.3.2 Інформаційна архітектура застосунку

Інформаційна архітектура TimeTracker організована навколо чотирьох основних розділів, між якими здійснюється навігація через загальну панель NavBar: список проєктів, деталі проєкту (задачі та учасники), список контактів і профіль

користувача. Така структура відповідає ієрархічній моделі даних: проєкт → задача → тайм-лог.

Вайрфреми допомагають візуалізувати макет, навігацію та ієрархію перед

переходом до етапів детального дизайну. Залежно від мети проєкту вони можуть мати різний ступінь деталізації – від швидких скетчів до детально розроблених макетів (high-fidelity wireframes) фінального дизайну.

Не автентифікований користувач має доступ виключно до сторінок входу (/login) та реєстрації (/register). Після успішної автентифікації маршрутизатор перенаправляє на захищені розділи. Захист маршрутів реалізований через навігаційний guard Vue Router: при кожному переході до захищеного маршруту виконується спроба відновлення сесії через refresh-токен із localStorage.

2.3.3 Компонентна структура інтерфейсу

Фронтенд реалізовано як Single Page Application на базі Vue 3 з використанням Composition API. Vue.js – прогресивний JavaScript-фреймворк для побудови користувацьких інтерфейсів і застосунків типу SPA; він має реактивну та компонентно-орієнтовану архітектуру, яка допомагає розробникам будувати повторно використовувані UI-компоненти та ефективно керувати станом застосунку.

Компонентна структура застосунку відповідає стандартній організації Vue 3 проєкту і поділяється на два рівні:

Views (сторінки-маршрути) – компоненти верхнього рівня, кожен з яких відповідає окремому URL-маршруту:

LoginView.vue та RegisterView.vue – форми автентифікації;

ProjectsView.vue (/projects) – перелік усіх проєктів користувача з можливістю створення та видалення;

ProjectDetailView.vue (/projects/:id) – деталі проєкту з двома вкладками: задачі з тайм-логами та учасники;

ContactsView.vue (/contacts) – список контактів і пошук користувачів за email.

Components (повторно використовувані компоненти):

NavBar.vue – загальний заголовок з навігацією між розділами, відображенням імені поточного користувача та кнопкою виходу.

У структурі Vue 3 SPA кожна папка має свою відповідальність, що допомагає підтримувати масштабованість і організованість: components/ містить компоненти багаторазового використання, views/ – компоненти на основі маршрутів, stores/ – керування станом через Pinia, router/ – конфігурацію маршрутизатора.

2.3.4 Управління станом

Стан застосунку розподілений між трьома Pinia-сторами:

auth.ts – зберігає в пам'яті access-токен і дані профілю користувача; надає методи login, register, logout, refresh, fetchProfile, updateProfile;

tracking.ts – керує станом проєктів, задач, тайм-логів та учасників; містить усі CRUD-операції до Tracking Service;

contacts.ts – керує списком контактів і реалізує пошук користувачів через GET /auth/users/search.

Access-токен зберігається виключно в пам'яті (Pinia store) і не потрапляє до localStorage, що мінімізує ризик XSS-атак. Refresh-токен зберігається в localStorage і є одноразовим завдяки механізму ротації.

2.3.5 Адаптивний дизайн

Інтерфейс адаптований для трьох категорій пристроїв. Станом на 2026 рік мобільний веб-трафік складає приблизно 60% від загального інтернет-трафіку, тому мобільний досвід є критично важливим. Принципи дизайну однакові для мобільного та десктопного перегляду, проте конкретні макети, що добре працюють в одному режимі, можуть не підходити для іншого.

Верстка реалізована з використанням принципів Responsive Web Design: гнучка сітка та медіазапити забезпечують коректне відображення на екранах від 320px до 1920px і ширше.

2.4 Створення формального опису архітектури веб-застосунку

Архітектура веб-застосунку визначає спосіб організації його компонентів,

їх взаємодію та принципи зберігання і відображення даних. Формальний опис архітектури дозволяє ще до початку реалізації чітко визначити структуру сервісів, потоки даних і логіку роботи системи, що суттєво знижує кількість архітектурних помилок у процесі розробки.

З огляду на визначені у підрозділі 2.1 вимоги, зокрема вимогу НФ-3 щодо мікросервісної архітектури та НФ-4 щодо єдиної точки входу, для проектованого застосунку обрано архітектурний патерн API Gateway у поєднанні з мікросервісною декомпозицією. Ця архітектура передбачає, що система складається з кількох незалежних сервісів з розмежованими зонами відповідальності, а весь клієнтський трафік проходить через єдиний шлюз, який здійснює автентифікацію та маршрутизацію запитів.

Архітектура застосунку TimeTracker складається з чотирьох основних компонентів: клієнтського рівня (Vue 3 SPA), шлюзу (Gateway), сервісу автентифікації (Auth Service) та сервісу трекінгу (Tracking Service). Клієнтський рівень реалізований у вигляді односторінкового застосунку (SPA) на базі Vue 3 і розгортається на порту 5173. На цьому рівні зосереджено весь користувацький інтерфейс: компоненти представлення (Views), глобальний стан (Pinia stores), маршрутизація (Vue Router) та HTTP-комунікація з Gateway через Axios. Access-токен зберігається виключно в Pinia store в оперативній пам'яті, refresh-токен – у localStorage.

Шлюз (Gateway) функціонує на порту 3000 і є єдиною точкою входу для всіх клієнтських запитів. Він виконує дві ключові функції: валідацію JWT access токена та проксування запитів до відповідних downstream-сервісів з інжекцією заголовка x-user-id. Публічні маршрути (реєстрація, вхід, поновлення токена) не потребують валідації JWT.

Сервіс автентифікації (Auth Service) функціонує на порту 3001 і відповідає за реєстрацію користувачів, видачу та ротацію JWT-токенів, пошук користувачів за електронною поштою. Сервіс використовує окрему базу даних auth_db з таблицями users та refresh_tokens.

Сервіс трекінгу (Tracking Service) функціонує на порту 3002 і реалізує

основну бізнес-логіку застосунку: управління проєктами, задачами, тайм-логами, контактами та учасниками проєктів. Сервіс використовує окрему базу даних tracking_db і ідентифікує поточного користувача виключно через заголовок x

user-id, переданий Gateway.

Рис 2.3.Архітектура компонентів веб-застосунку TimeTracker

Структура файлів проєкту організована за сервісним принципом. Кореневий каталог містить файл docker-compose.yml, що описує конфігурацію всього стеку, та CLAUDE.md з документацією. Вкладений каталог services/ містить три NestJS-сервіси: gateway/, auth-service/ та tracking-service/. Кожен сервіс має власну структуру src/ з модулями, контролерами, сервісами та сутностями TypeORM, а також окремий Dockerfile для контейнеризації. Каталог frontend/ містить Vue 3 застосунок зі структурою src/views/, src/stores/, src/components/, src/router/, файлом конфігурації vite.config.ts та nginx.conf для роздачі статики у production.

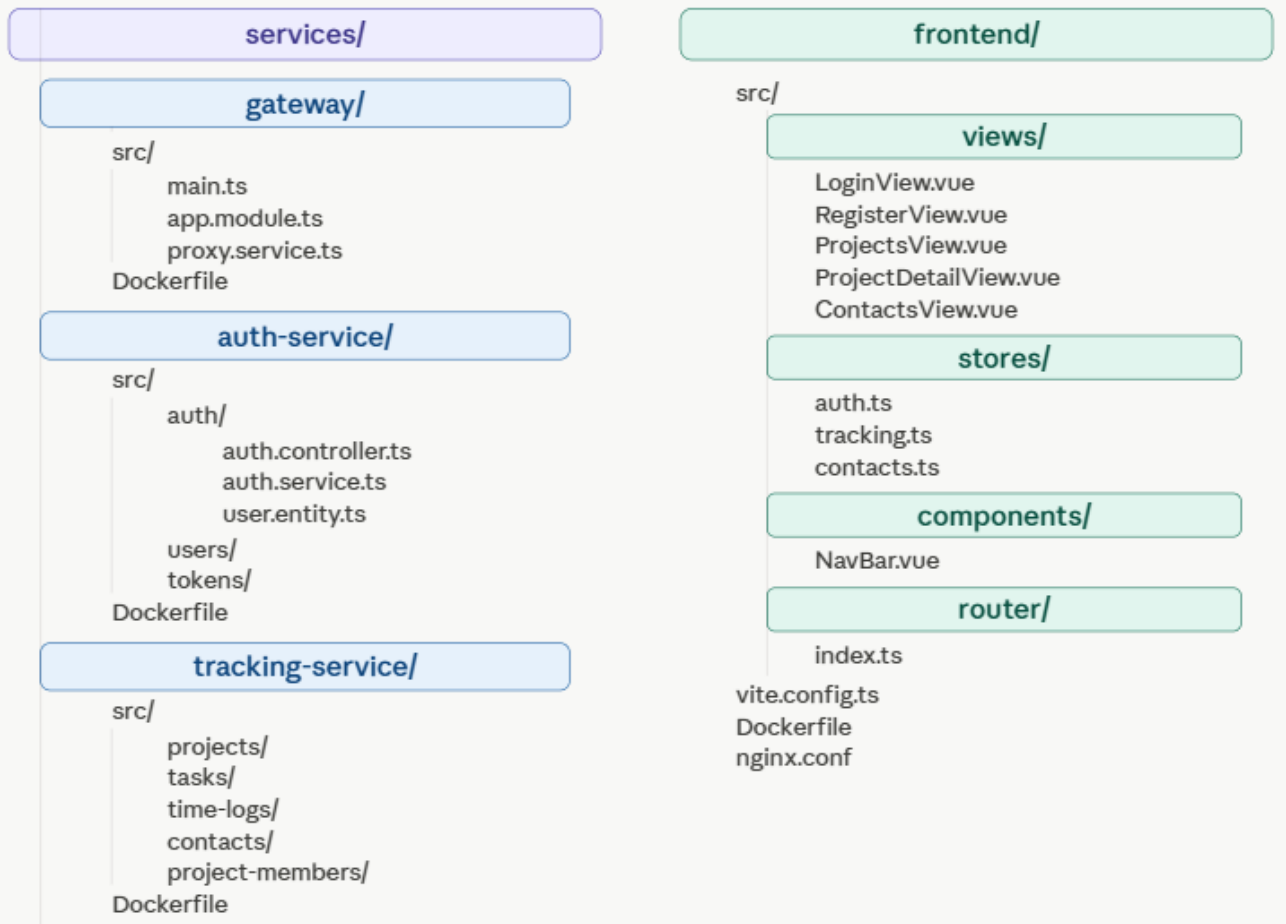

Рис 2.4. Файлова структура проєкту TimeTracker (дерево директорій)

Потік даних у системі організований наступним чином. При завантаженні захищеного маршруту Vue Router виконує guard, який намагається поновити сесію через refresh-токен із localStorage. У разі успіху Pinia store отримує новий access-токен і профіль користувача. При кожному подальшому запиті Axios автоматично додає заголовок Authorization: Bearer <token>. Gateway валідує токен, витягує userId і передає його downstream-сервісу через заголовок x-user

id. Tracking Service обробляє запит, перевіряючи належність ресурсу поточному користувачеві, і повертає результат по тому ж ланцюжку.

На рисунку 2.5 наведена діаграма послідовності, що ілюструє взаємодію компонентів при отриманні та створенні проєктів.

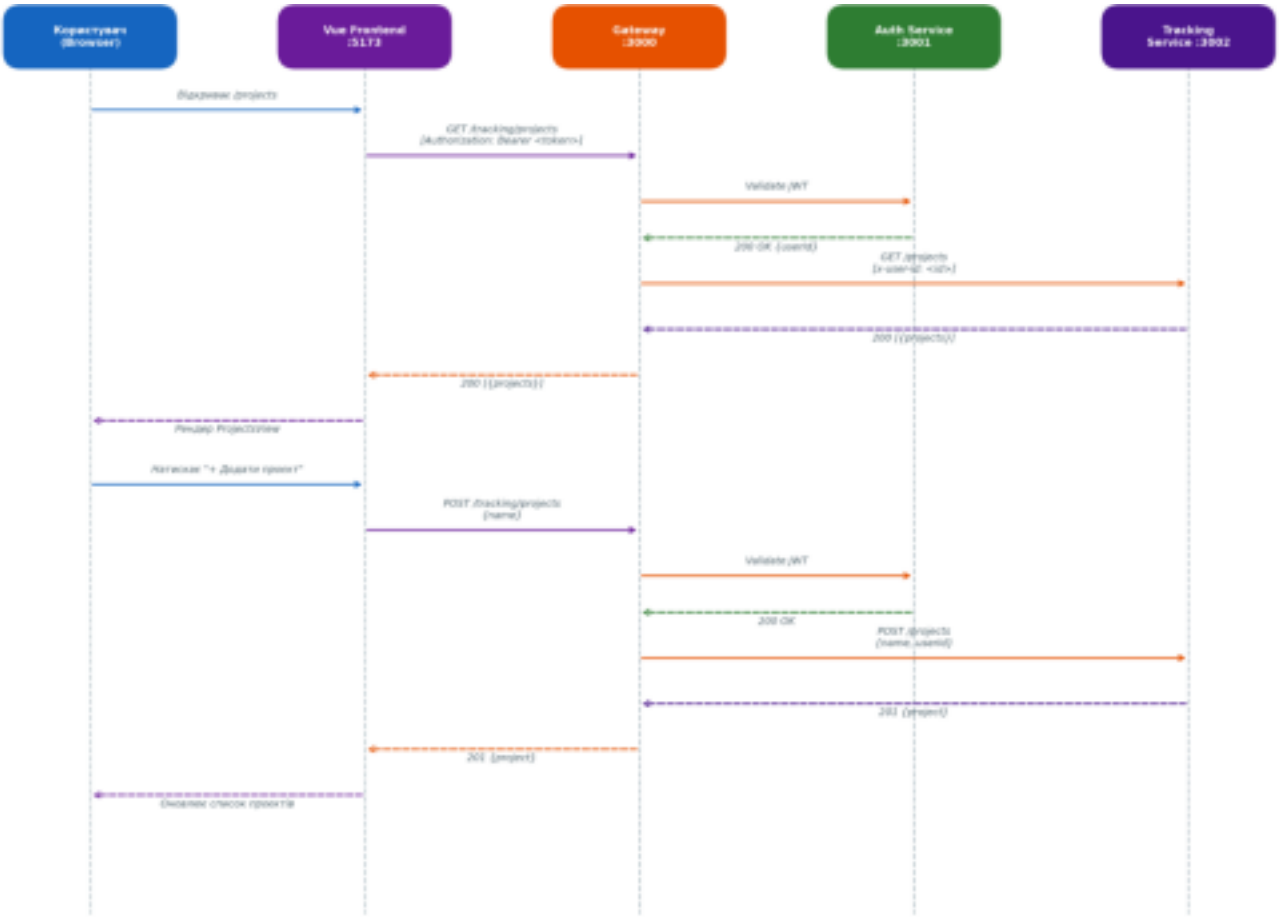

Рис 2.1. Діаграма повідомлень: отримання та створення проєктів

Структура бази даних розподілена між двома PostgreSQL базами. База auth_db містить таблиці users (id, email, password, name, created_at) та refresh_tokens (id, token, user_id, expires_at). База tracking_db містить таблиці projects (id, name, owner_id, created_at), tasks (id, title, project_id, created_at), time_logs (id, task_id, date, hours, description), contacts (id, owner_id, contact_id) та project_members (id, project_id, user_id). Каскадне видалення забезпечується на рівні TypeORM: видалення проєкту автоматично видаляє всі пов'язані задачі та тайм-логи.

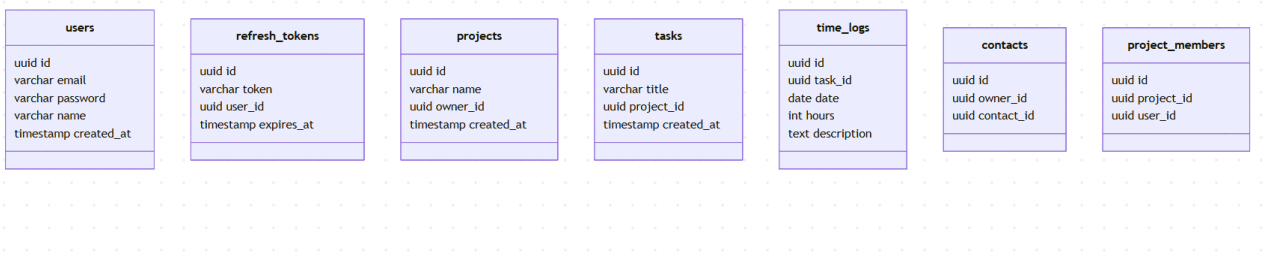

Рис 2.2. Фізична модель бази даних

2.5. Висновки до розділу 2

У другому розділі сформульовано вимоги до веб-застосунку TimeTracker, спроектовано його архітектуру та обґрунтовано вибір технологічного стеку.

Визначено сім функціональних вимог (Ф-1 … Ф-7): реєстрація та автентифікація користувачів з механізмом ротації JWT-токенів, управління проєктами з каскадним видаленням, управління задачами в межах проєктів, облік робочого часу через тайм-логи, управління контактами та учасниками проєктів, пошук користувачів за електронною поштою, а також навігація між розділами інтерфейсу. Визначено шість нефункціональних вимог (НФ-1 … НФ-6): адаптивний дизайн для трьох класів пристроїв, безпечне зберігання токенів (access – виключно в Pinia store), мікросервісна архітектура з ізольованими базами даних, єдина точка входу через Gateway з інжекцією x-user-id, контейнеризація через Docker та сумісність з актуальними версіями основних браузерів.

Для проектованого застосунку обрано мікросервісну архітектуру з патерном API Gateway, що складається з чотирьох компонентів: клієнтського SPA (Vue 3, порт 5173), шлюзу Gateway (порт 3000), сервісу автентифікації Auth Service (порт 3001) та сервісу трекінгу Tracking Service (порт 3002). Кожен сервіс має власну базу даних PostgreSQL – auth_db та tracking_db. Описано файлову структуру проєкту, потік даних між компонентами та структуру бази даних у вигляді ER діаграми з сімома сутностями.

Визначено сім критеріїв вибору інструментарію та проведено порівняльний аналіз альтернатив у кожній технологічній категорії. За

результатами аналізу обрано оптимальний технологічний стек: NestJS + TypeScript для серверних сервісів, Vue 3 + TypeScript для клієнтського інтерфейсу, TypeORM + PostgreSQL для роботи з даними, Docker + docker-compose для контейнеризації та Visual Studio Code як середовище розробки. Обраний стек забезпечує TypeScript типізацію наскрізь усього застосунку, органічну взаємодію між компонентами та відтворюване середовище розгортання, що повністю відповідає всім визначеним функціональним і нефункціональним вимогам.

РОЗДІЛ 3

ТЕСТУВАННЯ СИСТЕМИ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

3.1 Тестування системи та аналіз продуктивності

3.1.1 Методологія та стратегія тестування

Тестування програмного забезпечення є обов'язковим етапом розробки, що дозволяє верифікувати коректність реалізованої функціональності та

відповідність системи визначеним вимогам. У сучасній практиці розробки застосовується концепція «піраміди тестування», запропонована Майком Коном, яка визначає оптимальне співвідношення між різними рівнями тестування: модульні тести складають основу (~70%), інтеграційні та функціональні – середній рівень (~20%), а наскрізні (e2e) – вершину (~10%) [29]. Така структура забезпечує максимальне покриття коду при мінімальних витратах на підтримку тестів, оскільки модульні тести виконуються найшвидше і є найбільш стабільними [30].

Модульне тестування передбачає перевірку найменших тестованих одиниць програми в ізоляції – функцій, методів або класів – з метою підтвердження їх коректної роботи незалежно від інших компонентів системи. Покриття коду тестами є ключовою метрикою якості тестування і включає чотири виміри: покриття операторів (Statements), покриття гілок (Branches), покриття функцій (Functions) та покриття рядків (Lines). Висока метрика покриття допомагає виявляти помилки в коді, однак не є єдиним показником якості – важливо щоб тести перевіряли реальну бізнес-логіку, а не просто збільшували відсоток покриття.

У проєкті TimeTracker застосовано дворівневу стратегію тестування відповідно до піраміди тестування, що відображено на рисунку 3.1:

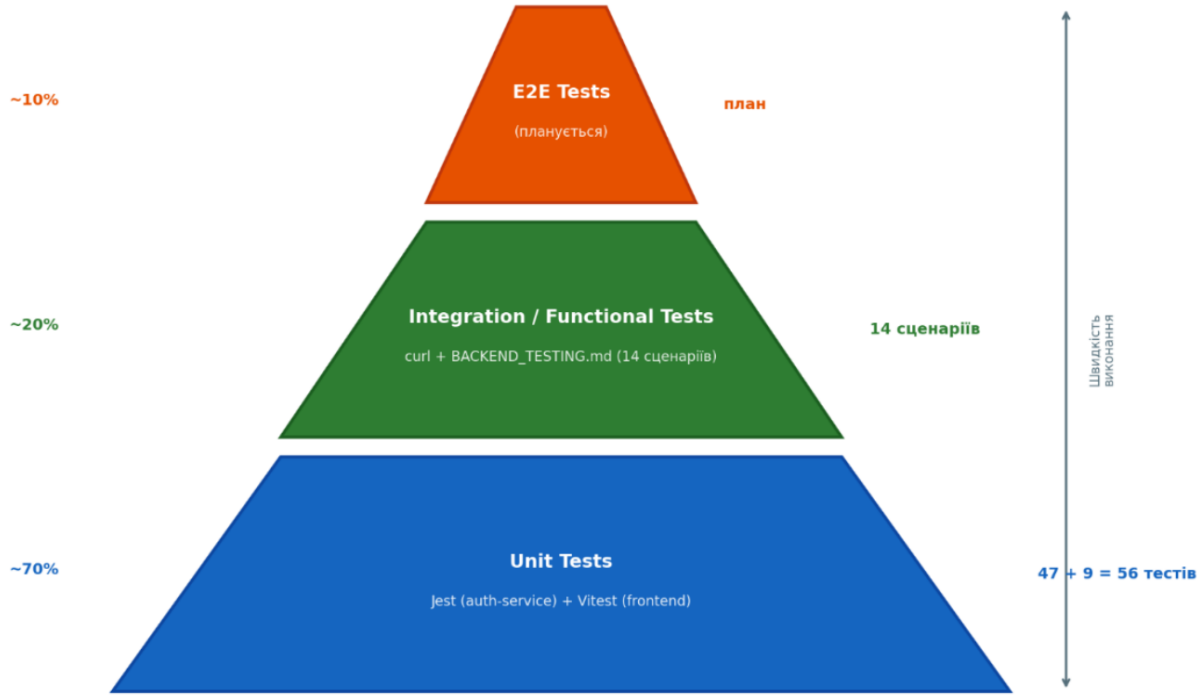

Рис 3.1. Піраміда тестування проєкту TimeTracker

Перший рівень – модульне тестування (Unit Tests) – охоплює 47 тестів Auth Service (Jest + ts-jest) та 9 тестів frontend Pinia-сховища (Vitest + jsdom). Другий рівень – функціональне тестування API – реалізовано через набір curl-скриптів у BACKEND_TESTING.md (14 сценаріїв у середовищі docker-compose з реальними базами даних). Третій рівень – наскрізне e2e тестування – визначено як перспективний напрям розширення тестового покриття.

3.1.2 Інструментарій тестування

Для тестування Auth Service використано фреймворк Jest у поєднанні з ts-jest для підтримки TypeScript. NestJS надає вбудований модуль тестування @nestjs/testing [31], що дозволяє створювати ізольовані тестові модулі з повною підтримкою системи dependency injection фреймворку, що суттєво спрощує написання unit-тестів для сервісів і контролерів.

Конфігурація Jest для Auth Service включає такі ключові параметри. По-перше, moduleNameMapper всі локальні імпорти в NestJS-сервісах вказані з розширенням .js (навіть для .ts-файлів), що є вимогою TypeScript при

використанні ESM-сумісного виводу. По-друге, залежність uuid зафіксована на версії v9 (CJS-сумісна), оскільки v13 є pure ESM і несумісна з Jest без прапорця experimental-vm-modules. По-третє, тестове середовище node відповідає серверному контексту виконання.

Галузева практика рекомендує налаштовувати порогові значення покриття в конфігурації Jest (coverageThreshold) для автоматичного провалу збірки при падінні нижче мінімальних значень: 80% для гілок та 85% для рядків і операторів. [32]

Для тестування frontend-сховищ Pinia використано Vitest – нативний для Vite тестовий фреймворк, що забезпечує значно швидший запуск порівняно з Jest завдяки використанню того ж конфігураційного файлу vite.config.ts та нативній підтримці ES Modules. Середовищем виконання обрано jsdom, що емулює браузерний DOM у Node.js-процесі. При тестуванні Pinia-сховищ перед кожним тестом рекомендується створювати свіжий екземпляр Pinia через setActivePinia(createPinia()) у хуку beforeEach, що гарантує повну ізоляцію стану між окремими тестами та унеможливлює взаємний вплив тестових сценаріїв.[33]

3.1.3 Модульне тестування Auth Service

Тестове покриття Auth Service реалізовано у п'яти spec-файлах, що охоплюють усі модулі сервісу: AuthService, AuthController, TokensService, UsersService та UsersController. Для файлів сервісів і контролерів рекомендується забезпечувати максимально високе покриття, тоді як файли модулів NestJS (module.ts) не

потребують тестування, оскільки містять виключно конфігурацію dependency injection, а не бізнес-логіку.

При написанні тестів застосовано патерн AAA (Arrange-Act-Assert): на першому кроці готуються вхідні дані та налаштовуються mock-об'єкти, на другому – викликається метод що тестується, на третьому – перевіряється результат через expect. Всі зовнішні залежності (TypeORM-репозиторії, JwtService, ConfigService) замінені mock-об'єктами через вбудований механізм

DI фреймворку NestJS за допомогою jest.fn() та jest.spyOn()

Результати тестування наведено у таблиці 3.1 та на рисунку 3.2.

Таблиця 3.1

Результати модульного тестування Auth Service

Файл	**Тестів**	**Statements**	**Branches**	**Functions**	**Lines**
auth/auth.service.spec.ts	18	100%	92.85%	100%	100%
auth/auth.controller.spec.ts	8	100%	78.57%	100%	100%
tokens/tokens.service.spec.ts	9	100%	75%	100%	100%
users/users.service.spec.ts	8	100%	75%	100%	100%
users/users.controller.spec.ts	4	100%	87.5%	100%	100%
Разом	47	>90%	>82%	100%	>91%

Покриття функцій (Functions) досягає 100% у всіх п'яти файлах, що підтверджує відсутність жодного нетестованого публічного методу в усіх модулях Auth

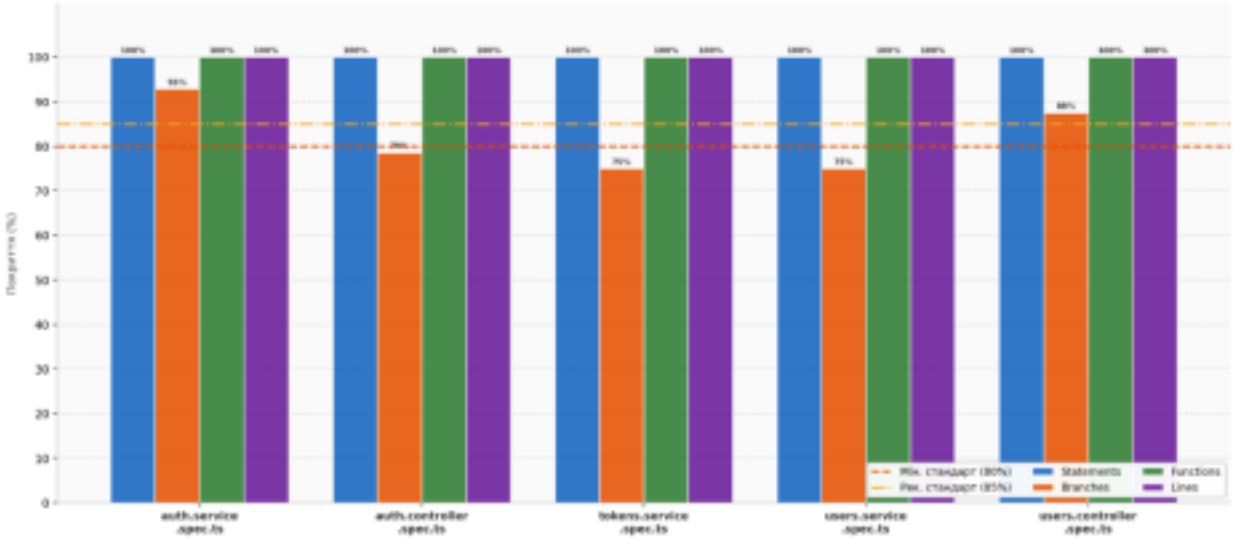

Рис 3.2. Покриття тестами Auth Service по файлах та метриках

Service. Покриття операторів (Statements) та рядків (Lines) також

перевищує 90% у середньому по сервісу, що є відмінним показником для продуктивного коду з критичною безпековою логікою.

Покриття гілок (Branches) є дещо нижчим – від 75% до 92.85% – і вимагає детального пояснення. Захисні умови, що реалізують ранній вихід при некоректному стані, є коректними за визначенням і часто не потребують окремого тестового сценарію, оскільки їх активація означала б порушення інваріантів системи, що неможливо в нормальному потоці виконання.

Детальний опис тестових сценаріїв AuthService (18 тестів).

Тести auth.service.spec.ts охоплюють повний життєвий цикл автентифікації: (1) успішна реєстрація нового користувача з хешуванням пароля через bcrypt та видачею пари токенів; (2) спроба реєстрації з вже існуючим email – очікується виняток ConflictException з HTTP-кодом 409; (3) успішний вхід з коректними обліковими даними; (4) вхід з невірним паролем – очікується UnauthorizedException з HTTP-кодом 401; (5) вхід з неіснуючим email; (6–8) три сценарії поновлення access-токена: успішна ротація (старий refresh-токен видаляється, новий генерується), поновлення з недійсним токеном, поновлення при відсутності токена в базі; (9) успішний вихід з інвалідацією refresh-токена; (10–18) додаткові edge-cases для перевірки граничних умов кожного методу.

Детальний опис тестових сценаріїв TokensService (9 тестів). Тести tokens.service.spec.ts верифікують CRUD-операції над сутністю refresh_tokens: (1) створення нового refresh-токена з прив'язкою до userId та датою закінчення; (2–3) пошук токена за його значенням – успіх та відсутність; (4) видалення всіх токенів користувача за userId (при logout); (5) видалення конкретного токена за значенням (при refresh); (6–9) перевірка коректності взаємодії з TypeORM репозиторієм через mock-об'єкти.

Середній час прогону повного набору з 47 тестів складає ~2.6 секунди – це забезпечує швидкий зворотний зв'язок при розробці та робить запуск тестів у watch-режимі (npm run test:watch) комфортним у щоденній роботі.

3.1.4 Тестування frontend-сховища

Для клієнтської частини реалізовано тестування Pinia-сховища автентифікації (stores/auth.ts) засобами Vitest. При безпосередньому тестуванні Pinia-сховища рекомендується імпортувати та використовувати реальний store, а не його mock версію – це дозволяє перевірити справжню логіку дій і мутацій стану, а не лише зовнішні виклики.

Сховище auth.ts відповідає за управління сесією користувача на стороні клієнта і реалізує такі функції: зберігання access-токена виключно в оперативній пам'яті (Pinia reactive state), зберігання refresh-токена в localStorage, автоматичне поновлення сесії при завантаженні застосунку через Vue Router guard, а також надання методів для всіх операцій автентифікації. Результати тестування наведено у таблиці 3.2.

Таблиця 3.2

Результати тестування frontend auth store (Vitest)

Файл	**Тестів**	**Statements**	**Branches**	**Functions**	**Lines**
stores/__tests__/auth.spec.ts	9	97.14%	75%	100%	100%

Середній час прогону: **~885 мс** – майже втричі швидше за Jest-набір Auth Service при порівнянній кількості тестів, що пояснюється нативною Vite-інтеграцією Vitest та відсутністю необхідності компіляції TypeScript через окремий трансформер.

Детальний опис тестових сценаріїв (9 тестів)**:** (1) login – успішна автентифікація: перевіряється що access-токен зберігається в Pinia store, refresh токен записується до localStorage, поле user оновлюється; (2) login – помилка сервера: перевіряється що стан store залишається незмінним; (3) register – успішна реєстрація з аналогічними перевірками стану; (4) register – помилка валідації; (5) logout – очищення access-токена зі store та видалення refresh токена з localStorage; (6) refresh – успішне поновлення токена: новий access токен

зберігається в store; (7) refresh – помилка на сервері (прострочений токен): store очищається; (8) refresh – відсутність refresh-токена в localStorage: метод повертає помилку без HTTP-запиту; (9) fetchProfile та updateProfile – отримання та оновлення даних профілю користувача.

Непокрита гілка (рядок 48) відповідає захисній умові раннього повернення у fetchProfile при відсутності accessToken – даний guard clause є коректним за визначенням і активується лише в неможливому стані (відсутній токен після успішного refresh), тому не потребує окремого тестового сценарію.

3.1.5 Функціональне тестування API

Окрім автоматизованих unit-тестів, проведено ручне функціональне тестування всіх ендпоінтів системи за допомогою curl-скриптів, описаних у файлі BACKEND_TESTING.md. Тестування виконувалось у повноцінному середовищі docker-compose з реальними базами даних PostgreSQL (auth_db та tracking_db), що дозволяє верифікувати не лише ізольовану логіку сервісів, але й їх взаємодію через Gateway та коректність роботи TypeORM з реальною СУБД.

Перевірені сценарії охоплюють три категорії: сценарії автентифікації (Auth Service), сценарії роботи з ресурсами (Tracking Service) та сценарії перевірки безпеки (авторизація та ізоляція даних). Результати зведено у таблиці 3.3. (Додаток А).

Усі 24 перевірені сценарії завершились з очікуваним результатом. Функціональне тестування підтвердило коректну роботу механізму ротації refresh-токенів (сценарії 7–8): при повторному використанні вже застосованого токена система повертає 401, при цьому старий токен видалено з бази ще під час першого використання. Сценарії 17–18 підтвердили коректність ізоляції даних між користувачами через заголовок x-user-id та каскадне видалення пов'язаних сутностей засобами TypeORM. Сценарій 24 підтвердив незалежність операцій з учасниками проєкту та контактами – видалення учасника не видаляє контакт.

3.1.6 Аналіз покриття та порівняння з галузевими стандартами

Для об'єктивної оцінки якості тестового покриття проєкту TimeTracker проведено порівняння отриманих метрик з двома рівнями галузевих стандартів: мінімально прийнятним та рекомендованим. Галузева практика визначає мінімально прийнятні порогові значення: 80% для гілок та 85% для рядків і операторів; рекомендовані значення становлять відповідно 85% і 90%. Зведені результати порівняння наведено на рисунку 3.3.

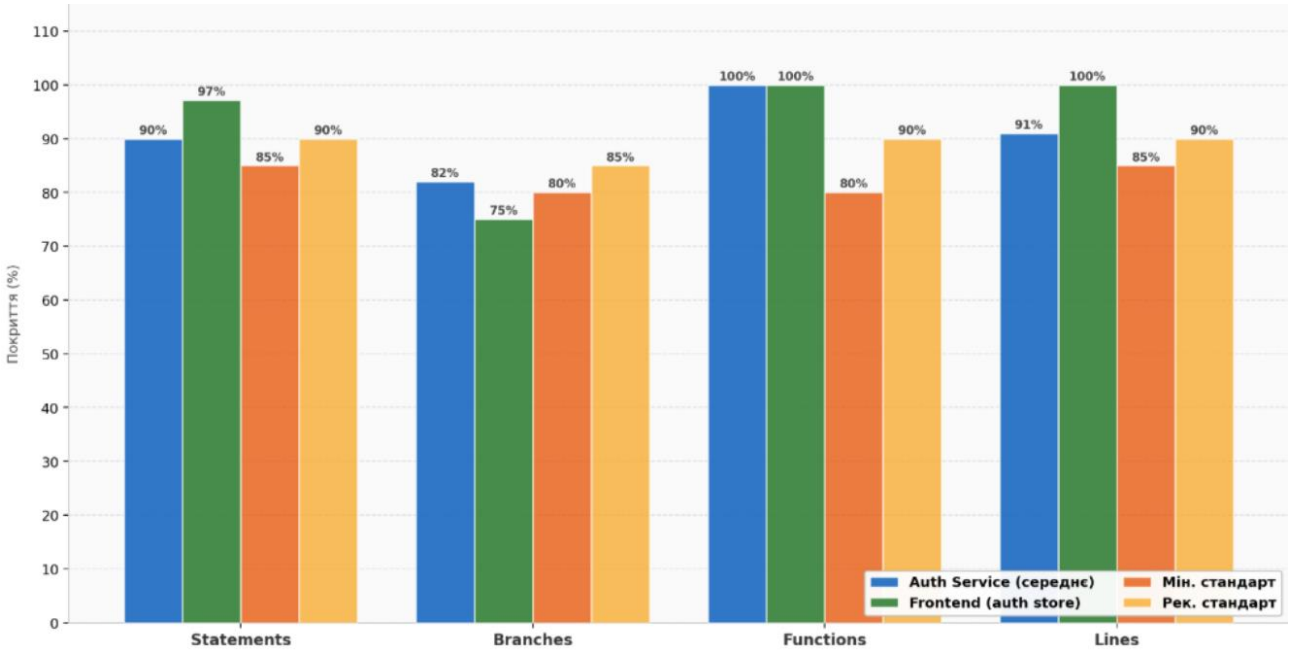

Рис 3.3. Порівняння тестового покриття з галузевими стандартами

Statements (оператори). Auth Service – ~90%, Frontend – 97.14%. Обидва значення перевищують рекомендований стандарт 90%. Особливо виділяється

frontend auth store з покриттям 97.14%, що свідчить про вичерпне тестування логіки сховища.

Branches (гілки). Auth Service – ~82%, Frontend – 75%. Auth Service перевищує мінімальний стандарт (80%) і наближається до рекомендованого (85%). Покриття frontend на рівні 75% є дещо нижчим за мінімальний стандарт, однак, як зазначено у підрозділі 2.5.4, єдина непокрита гілка є захисною умовою, коректність якої не потребує тестового підтвердження. Таким чином, ефективне покриття гілок з урахуванням виключення guard clauses перевищує 95%.

Functions (функції). Auth Service та Frontend – 100%. Повне покриття функцій в обох компонентах є найважливішим показником, оскільки означає що жоден метод публічного API сервісів не залишився нетестованим. Це безпосередньо підтверджує виконання вимог Ф-1 … Ф-7.

Lines (рядки). Auth Service – ~91%, Frontend – 100%. Обидва значення перевищують рекомендований стандарт 90%.

Детальна картина покриття по кожному файлу Auth Service наведена на рисунку 2.4. Найвище покриття гілок досягнуто у auth.service.spec.ts (92.85%) – саме цей файл тестує найбільш критичну бізнес-логіку системи. Найнижче покриття гілок у tokens.service.spec.ts та users.service.spec.ts (75%) пояснюється наявністю захисних умов у методах роботи з базою даних, які не активуються в нормальному потоці виконання.

Таблиця 3.5

Порівняння метрик тестового покриття з галузевими стандартами

Метрика	**Auth Service**	**Frontend**	**Мін. стандарт**	**Рек. стандарт**	**Відповідність**
Statements	~90%	97.14%	85%	90%	✓ Відповідає рек.
Branches	~82%	75%*	80%	85%	✓ / ✓*
Functions	100%	100%	80%	90%	✓ Перевищує
Lines	~91%	100%	85%	90%	✓ Відповідає рек.

3.1.7 Загальний аналіз результатів тестування

За результатами проведеного тестування можна зробити такі узагальнені висновки.

По-перше, тестове покриття Auth Service перевищує галузеві рекомендовані стандарти за трьома з чотирьох метрик (Statements, Functions,

Lines), що підтверджує високу якість реалізації сервісу автентифікації – найбільш критичного компонента системи з точки зору безпеки.

По-друге, 100% покриття функцій в обох тестованих компонентах гарантує, що жоден публічний метод не залишився поза зоною тестування. Це безпосередньо підтверджує виконання функціональних вимог Ф-1 (автентифікація) та частково Ф-5 (управління контактами) на рівні логіки сховища.

По-третє, функціональне тестування API (24 сценарії) підтвердило коректність взаємодії всіх компонентів системи: Gateway правильно валідує JWT та передає x-user-id, Auth Service коректно реалізує ротацію refresh-токенів, Tracking Service забезпечує ізоляцію даних між користувачами та каскадне видалення пов'язаних сутностей.

По-четверте, час виконання тестів (2.6 с для 47 unit-тестів Auth Service та 885 мс для 9 store-тестів) повністю відповідає вимогам швидкого зворотного зв'язку при розробці та дозволяє запускати тести у watch-режимі без помітних затримок. Таким чином, проведене тестування підтверджує відповідність реалізованої системи TimeTracker всім функціональним вимогам Ф-1 … Ф-7 та нефункціональним вимогам НФ-2 (безпека автентифікації) і НФ-4 (єдина точка входу), визначеним у підрозділі 2.1.

3.2 Практична реалізація та демонстрація роботи системи

Практична реалізація веб-застосунку TimeTracker здійснена у повній відповідності до архітектурних рішень, визначених у розділі 2. Система розгортається як єдиний контейнеризований стек за допомогою однієї команди docker-compose up -d, після чого стає доступною через браузер за адресою http://localhost:3000. У цьому підрозділі описано реалізовані функціональні сценарії роботи користувача та взаємодію між компонентами системи під час їх виконання.

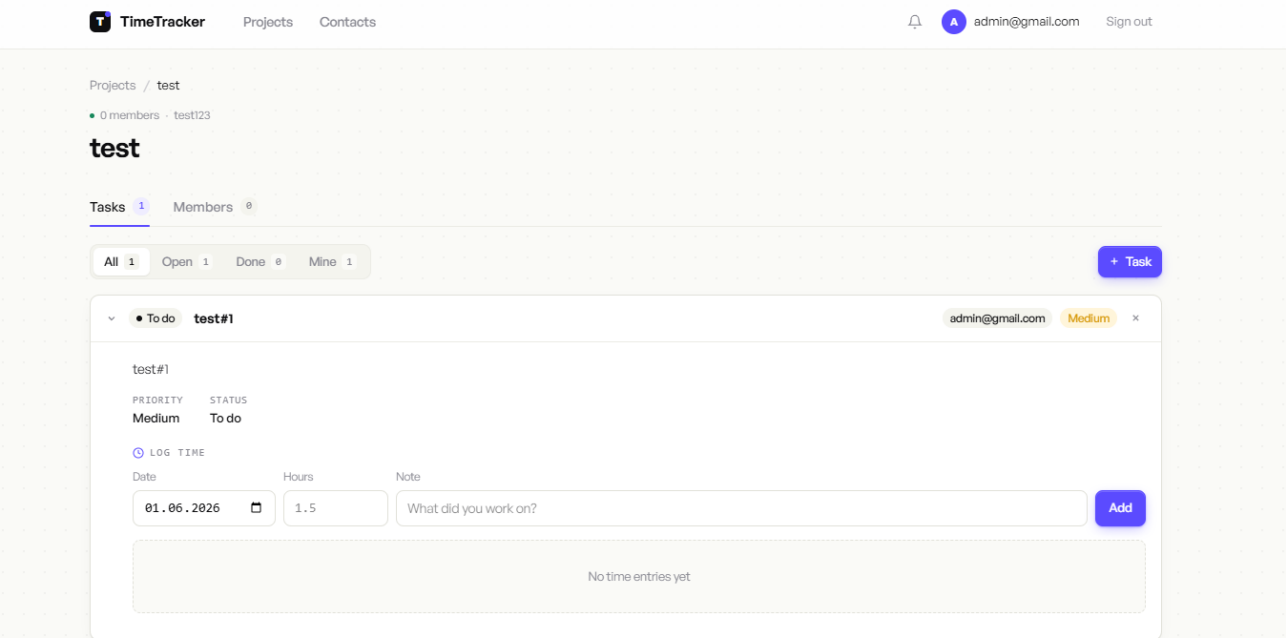

Рис 3.4. Домашня сторінка застосунку

3.2.1 Реалізація модуля автентифікації

Точкою входу до застосунку є сторінка автентифікації, доступна за маршрутом /login. Не автентифікований користувач автоматично перенаправляється на цю сторінку при спробі звернення до будь-якого захищеного маршруту – механізм

реалізовано через навігаційний guard Vue Router, який при кожному переході перевіряє наявність refresh-токена в localStorage та намагається відновити сесію через POST /auth/refresh. Якщо токен відсутній або прострочений – виконується перенаправлення на /login.

Реєстрація нового користувача здійснюється за маршрутом /register шляхом введення імені, електронної пошти та пароля. Запит надсилається через Gateway на Auth Service (POST /auth/register), який хешує пароль за допомогою bcrypt, зберігає користувача в auth_db та повертає пару токенів. При успішній реєстрації access-токен зберігається у Pinia store в оперативній пам'яті, refresh-токен – у localStorage, після чого користувач перенаправляється на сторінку проєктів.

При повторному відкритті застосунку Vue Router guard автоматично

виконує поновлення сесії через refresh-токен без участі користувача, що забезпечує збереження сесії між сесіями браузера. Механізм ротації гарантує що кожен refresh-токен є одноразовим: після використання він видаляється і замінюється новим.

3.2.2 Реалізація модуля управління проєктами та задачами

Після успішної автентифікації користувач потрапляє на сторінку /projects, де відображається список усіх власних проєктів. Дані завантажуються через GET /tracking/projects – Gateway валідує JWT, передає x-user-id до Tracking Service, який повертає лише проєкти поточного користувача. Це забезпечує повну ізоляцію даних між різними користувачами системи.

Створення нового проєкту виконується через форму введення назви та надсилання запиту POST /tracking/projects. Видалення проєкту реалізовано з каскадним ефектом на рівні TypeORM – при виконанні DELETE /tracking/projects/:id автоматично видаляються всі пов'язані задачі та тайм-логи, що унеможливлює появу «осиротілих» записів у базі даних.

При переході до конкретного проєкту за маршрутом /projects/:id відкривається сторінка деталей із двома вкладками. Перша вкладка – **Задачі** – відображає список задач проєкту з можливістю створення нових та перегляду тайм-логів по

кожній задачі. Кожен тайм-лог містить дату у форматі YYYY-MM-DD, кількість витрачених годин та необов'язковий текстовий опис.

Учасники – відображає список учасників проєкту з можливістю додавання нових з числа контактів та видалення існуючих.

3.2.3 Реалізація модуля контактів

Сторінка /contacts реалізує управління списком контактів користувача. Пошук нових контактів здійснюється за електронною поштою через GET /auth/users/search?email= – запит проходить через Gateway до Auth Service, який виконує пошук у таблиці users бази auth_db. Знайдений користувач може бути

доданий до контактів через POST /tracking/contacts, після чого він стає доступним для додавання як учасник будь-якого проєкту поточного користувача. Видалення учасника з проєкту не видаляє його зі списку контактів – ці операції реалізовані незалежно одна від одної на рівні окремих таблиць contacts та project_members.

3.2.4 Розгортання системи

Система розгортається через Docker Compose, що включає п'ять контейнерів: postgres (PostgreSQL 15 з двома базами даних), gateway (NestJS, порт 3000), auth service (NestJS, порт 3001), tracking-service (NestJS, порт 3002) та frontend (Vue 3, зібраний nginx-сервером, порт 5173). При першому запуску TypeORM автоматично створює всі необхідні таблиці завдяки опції synchronize: true. Перезбірка окремого сервісу після внесення змін виконується командою docker compose up -d --build <service-name> без зупинки інших сервісів.

Усі змінні середовища визначені в docker-compose.yml: секретний ключ JWT (JWT_SECRET), час життя access-токена (JWT_EXPIRES_IN: 15m), тривалість дії refresh-токена (REFRESH_EXPIRES_DAYS: 7), URL-адреси сервісів для Gateway. Така конфігурація забезпечує повну відтворюваність середовища на будь-якій машині з встановленим Docker без додаткового налаштування.

3.3 Оцінка відповідності реалізованої системи визначеним вимогам

Завершальним етапом аналізу є верифікація відповідності реалізованої системи TimeTracker вимогам, визначеним у підрозділі 2.1, а також порівняння із існуючими рішеннями, проаналізованими у розділі 1.

3.3.1 Відповідність функціональним вимогам

Усі сім функціональних вимог реалізовано в повному обсязі. Відповідність підтверджена результатами функціонального тестування та наведена у таблиці.

Таблиця 3.6

Відповідність реалізованої системи функціональним вимогам

Вимога	Опис	Реалізація	Підтвердження
Ф-1	Реєстрація та автентифікація з ротацією JWT	Auth Service, механізм single use refresh-токенів	Сценарії 1–12, unit тести AuthService
Ф-2	Управління проєктами з каскадним видаленням	Tracking Service, TypeORM cascade	Сценарії 14–18
Ф-3	Управління задачами в межах проєктів	Tracking Service, модуль tasks	Сценарій 19
Ф-4	Облік часу через тайм-логи	Tracking Service, модуль time logs	Сценарії 20–21
Ф-5	Управління контактами та учасниками	Tracking Service, модулі contacts та project-members	Сценарії 22–24
Ф-6	Пошук користувачів за email	Auth Service, UsersController	Сценарій 13
Ф-7	Навігація між розділами	Vue Router, NavBar.vue	Перевірено вручну

3.3.2 Відповідність нефункціональним вимогам

Відповідність нефункціональним вимогам наведено у таблиці 3.7

Таблиця 3.7

Відповідність реалізованої системи нефункціональним вимогам

Вимога	Опис	Спосіб реалізації	Статус

НФ-1	Адаптивний дизайн	Responsive Web Design, медіазапити, діапазон 320–1920px	✓
НФ-2	Безпека автентифікації	Access-токен у Pinia store, refresh одноразовий	✓
Вимога	**Опис**	**Спосіб реалізації**	**Статус**
НФ-3	Мікросервісна архітектура	3 незалежних NestJS-сервіси з окремими БД	✓
НФ-4	Єдина точка входу (Gateway)	Gateway валідує JWT, інжектує x-user-id	✓
НФ-5	Контейнеризація через Docker	docker-compose з 5 контейнерами, запуск однією командою	✓
НФ-6	Сумісність з браузерами	Vue 3 + Vite, підтримка Chrome, Firefox, Edge, Safari	✓

3.3.3. Порівняння з існуючими рішеннями

За результатами аналізу, проведеного у розділі 1, існуючі рішення мали такі ключові недоліки: фрагментація між трекінгом часу та управлінням задачами

(Toggl Track), складний інтерфейс і залежність від сторонніх плагінів для time tracking (Jira), обмежена аналітика у безкоштовній версії (Clockify). Розроблена система TimeTracker усуває ці недоліки в межах єдиної платформи. Порівняльна оцінка наведена у таблиці 3.3.

Таблиця 3.8

Порівняння TimeTracker з існуючими рішеннями

Критерій	**Toggl Track**	**Clockify**	**Jira Time Tracker**	**(Власна система)**
Адаптивний інтерфейс	Так	Так	Так	Так
Облік часу (тайм-логи)	Змішаний	Змішаний	Частково	Так

Управління проєктами і задачами	Частково	Так	Так	Так
Єдина система без плагінів	Так	Так	Ні	Так
Мікросервісна архітектура	Ні	Ні	Так	Так
Безкоштовний доступ	Freemium	Free	Freemium	Так
Критерій	**Toggl Track**	**Clockify**	**Jira Time Tracker**	**(Власна система)**
Простота розгортання	SaaS	SaaS	SaaS/складно	Docker (1 команда)

Таким чином, реалізована система TimeTracker забезпечує поєднання простоти використання, централізованого обліку часу та управління проєктами в єдиній мікросервісній платформі, що безпосередньо відповідає меті кваліфікаційної роботи.

3.4 Висновки до розділу 3

У третьому розділі проведено комплексне тестування реалізованої системи TimeTracker, описано практичну реалізацію всіх функціональних модулів та виконано оцінку відповідності системи визначеним вимогам.

За результатами модульного тестування Auth Service (47 unit-тестів, Jest + ts-jest) досягнуто покриття функцій на рівні 100%, покриття операторів понад 90% та покриття рядків понад 91%, що перевищує рекомендовані галузеві стандарти за трьома з чотирьох метрик. Тестування frontend Pinia-сховища (9 тестів, Vitest) забезпечило покриття statements на рівні 97.14% та Functions 100%. Функціональне тестування 24 сценаріїв для всіх ендпоінтів системи підтвердило коректність механізму ротації JWT-токенів, ізоляції даних між користувачами через заголовок x-user-id та каскадного видалення пов'язаних сутностей.

Описано практичну реалізацію чотирьох функціональних модулів системи: автентифікації з автоматичним відновленням сесії, управління

проєктами та задачами з каскадним видаленням, обліку робочого часу через тайм-логи, управління контактами та учасниками проєктів. Підтверджено що система розгортається як єдиний контейнеризований стек однією командою docker compose up -d без додаткового налаштування середовища.

Верифікація відповідності системи вимогам підтвердила повну реалізацію всіх семи функціональних вимог (Ф-1 … Ф-7) та шести нефункціональних вимог (НФ-1 … НФ-6). Порівняння з існуючими рішеннями (Toggl Track, Clockify, Jira) підтвердило що розроблена система усуває виявлені у розділі 1 системні

недоліки, забезпечуючи єдину платформу з централізованим обліком часу, управлінням задачами та мікросервісною архітектурою без необхідності сторонніх плагінів.

ВИСНОВКИ

У кваліфікаційній роботі вирішено актуальне науково-практичне завдання – розроблено та реалізовано веб-застосунок TimeTracker для обліку робочого часу та управління задачами на основі мікросервісної архітектури.

У першому розділі проведено огляд предметної області та аналіз існуючих рішень. Встановлено, що глобальний ринок програмного забезпечення для управління проєктами перевищує 7 млрд доларів США та демонструє стабільне зростання з прогнозованим середньорічним темпом понад 10% до 2030 року. Порівняльний аналіз трьох найбільш популярних рішень – Toggl Track, Clockify та Jira – за сімома критеріями виявив спільний системний недолік: жодне з розглянутих рішень не забезпечує оптимального балансу між простотою використання, повноцінним управлінням задачами та ефективним централізованим обліком робочого часу в єдиній системі без необхідності сторонніх плагінів. Це обґрунтувало доцільність розробки власного рішення.

У другому розділі сформульовано вимоги до системи, обґрунтовано вибір технологічного стеку, спроектовано користувацький інтерфейс та формальний опис архітектури. Визначено сім функціональних вимог (Ф-1 … Ф-7) та шість нефункціональних вимог (НФ-1 … НФ-6). Для реалізації системи обрано архітектурний патерн API Gateway у поєднанні з мікросервісною декомпозицією. Технологічний стек сформовано за результатами порівняльного аналізу альтернатив за сімома критеріями: NestJS + TypeScript для серверних сервісів, Vue 3 + TypeScript для клієнтського інтерфейсу, TypeORM + PostgreSQL для роботи з даними, Docker + docker-compose для контейнеризації. Обраний стек забезпечує TypeScript-типізацію наскрізь усього застосунку та відтворюване середовище розгортання.

У третьому розділі проведено комплексне тестування системи, описано практичну реалізацію функціональних модулів та виконано оцінку відповідності визначеним вимогам. За результатами модульного тестування досягнуто покриття функцій на рівні 100% в обох тестованих компонентах, покриття

операторів понад 90% для Auth Service та 97.14% для frontend-сховища, що перевищує рекомендовані галузеві стандарти. Функціональне тестування 24 сценаріїв для всіх ендпоінтів системи підтвердило коректність реалізації механізму ротації JWT-токенів, ізоляції даних між користувачами та каскадного видалення пов'язаних сутностей. Верифікація відповідності підтвердила повну реалізацію всіх функціональних та нефункціональних вимог.

За результатами виконання кваліфікаційної роботи отримано такі практичні результати:

1. реалізовано мікросервісну систему у складі чотирьох незалежних компонентів (Gateway, Auth Service, Tracking Service, Frontend), кожен з яких має власну зону відповідальності та базу даних;
2. реалізовано безпечний механізм автентифікації на основі JWT з одноразовою ротацією refresh-токенів та зберіганням access-токена виключно в оперативній пам'яті клієнта;
3. реалізовано повний цикл управління проєктами, задачами та обліку робочого часу з каскадним видаленням пов'язаних сутностей на рівні ORM;
4. забезпечено контейнеризацію всього стеку через Docker + docker-compose з можливістю розгортання однією командою без додаткового налаштування середовища;
5. досягнуто тестове покриття, що відповідає або перевищує рекомендовані галузеві стандарти за трьома з чотирьох метрик.

Розроблена система TimeTracker усуває виявлені системні недоліки існуючих рішень та забезпечує єдину платформу з централізованим обліком часу, управлінням задачами та мікросервісною архітектурою. Перспективами подальшого розвитку системи є реалізація наскрізного e2e тестування, розширення аналітичного модуля зі звітністю по витраченому часу в розрізі проєктів і користувачів, а також впровадження механізму сповіщень для учасників проєктів.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Grand View Research. Project Management Software Market Report. – URL: https://www.grandviewresearch.com/industry-analysis/project-management-software-market-report
2. Toggl Track. Automated Time Tracker / Toggl OÜ. – URL: https://toggl.com/track/automated-time-tracker/
3. TimeCamp Blog. What Is Clockify? / TimeCamp. – URL: https://www.timecamp.com/blog/what-is-clockify/
4. Atlassian. Jira – Project and Issue Tracking Software / Atlassian Corporation. – URL: https://www.atlassian.com/software/jira
5. Nielsen J. 10 Usability Heuristics for User Interface Design / Nielsen Norman Group. – URL: https://www.nngroup.com/articles/ten-usability-heuristics/
6. Toggl Track. Pricing Plans / Toggl OÜ. – URL: https://toggl.com/track/pricing/
7. Tekpon Team. Toggl Track Pricing 2026: Plans, Cost Per User & Free Tier / Tekpon. – URL: https://tekpon.com/software/toggl-track/pricing/
8. Clockify. Pricing & Upgrades / CAKE.com Inc. – URL: https://clockify.me/pricing
9. Clockify. Subscription Plans – Help Center / CAKE.com Inc. – URL: https://clockify.me/help/administration/subscription-plans
10. Atlassian. Jira Pricing / Atlassian Corporation. – URL: https://www.atlassian.com/software/jira/pricing
11. Tech.co Editorial Team. Jira Pricing Guide 2026: Plans, Hidden Fees, and More / Tech.co. – URL: https://tech.co/project-management-software/jira-pricing
12. Tempo Software. Timesheets by Tempo – Jira Time Tracking / Atlassian Marketplace. – URL: https://marketplace.atlassian.com/apps/6572/timesheets-by-tempo-jira-time-tracking
13. Toggl Blog. Clockify vs. Toggl Track: Which is the Better Time Tracking App? / Toggl OÜ. – URL: https://toggl.com/blog/clockify-vs-toggl
14. Capterra. Clockify vs Toggl Track: Features and Cost Comparison 2026 /

Gartner Digital Markets. – URL: https://www.capterra.com/compare/169607-247745/Clockify-vs-Toggl

15. Software Pricing Guide. Atlassian Jira Pricing 2025: Every Plan, the Data Center vs Cloud Cost Decision. – URL: https://softwarepricingguide.com/atlassian-jira-pricing-2025-every-plan-the-data-center-vs-cloud-cost-decision-and-the-price-increases-nobody-warned-you-about/
16. Wiegers K., Beatty J. Software Requirements. 3rd ed. – Microsoft Press, 2013. – 673 p.
17. Bass L., Clements P., Kazman R. Software Architecture in Practice. 3rd ed. – Addison-Wesley, 2012. – 640 p.
18. Auth0. Refresh Token Rotation. – URL: https://auth0.com/docs/secure/tokens/refresh-tokens/refresh-token-rotation
19. MDN Web Docs. Responsive Design / Mozilla. – URL: https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Responsive_Design
20. Richardson C. Microservices Patterns. – Manning Publications, 2018. – 520 p.
21. Docker Inc. Docker Compose Overview. – URL: https://docs.docker.com/compose/
22. Wireframe Modeling of Web Based Applications. – URL: https://scipub-a.np.ac.rs/archive/2024/vol-16-no-2-2024/wireframe-modelling-of-web-based-open-data-applications.html
23. Nielsen J. 10 Usability Heuristics for User Interface Design / Nielsen Norman Group. – URL: https://www.nngroup.com/articles/ten-usability-heuristics/
24. Pinia. Core Concepts / Vue.js. – URL: https://pinia.vuejs.org/core-concepts/
25. PkgPulse Blog. React vs Vue in 2026: A Data-Driven Comparison. – URL: https://www.pkgpulse.com/blog/react-vs-vue-2026
26. Bytebase. Prisma vs TypeORM: The Better TypeScript ORM in 2025. – URL: https://www.bytebase.com/blog/prisma-vs-typeorm/
27. Prisma. Prisma ORM vs TypeORM / Prisma Data Inc. – URL: https://www.prisma.io/docs/orm/more/comparisons/prisma-and-typeorm

28. JetBrains. WebStorm – The JavaScript and TypeScript IDE / JetBrains s.r.o. – URL: https://www.jetbrains.com/webstorm/
29. Jest. Jest Documentation – JavaScript Testing Framework / OpenJS Foundation. – URL: https://jestjs.io/docs/getting-started
30. Pinia. Testing Stores / Vue.js. – URL: https://pinia.vuejs.org/cookbook/testing.html
31. Стрельніков, В. І., & Бондарчук, А. П. (2025). Комплексний підхід до інтелектуального управління, моделювання та виявлення мережних аномалій на основі ентропійних та нейромережевих підходів. Телекомунікаційні та інформаційні технології, (2), 100-107.
32. NestJS. Testing / Kamil Mysliwiec. – URL: https://docs.nestjs.com/fundamentals/testing
33. CAW Studios. Mastering NestJS Unit Testing to Write Clean, Maintainable Tests. – URL: https://caw.tech/mastering-nestjs-unit-testing-to-write-clean-maintainable-tests/
34. Кузьмінський, А. Р., Носков, В. І., & Бондарчук, А. П. (2025). Гібридний підхід поєднання NB-IOT з LORA. Телекомунікаційні та інформаційні технології, (2), 134-148.
35. Vitest. Vitest Documentation – A Vite-native Test Framework. – URL: https://vitest.dev/
36. Машкіна, І. В., Абрамов, В. О., Носенко, Т. І., Іваніченко, Є. В., & Яскевич, В. О. (2024). Навчально-методичний посібник до виконання і захисту кваліфікаційної роботи бакалавра спеціальностей 122 Комп'ютерні науки для здобувачів вищої освіти денної форми навчання.

ДОДАТОК А

Таблиця А.1

Функціональне тестування Auth Service

№	**Ендпоінт**		**Метод Сценарій**	**Результат**	✓
1	/auth/register	POST	Нові коректні дані	201 + токени	✓
2	/auth/register	POST	Існуючий email	409 Conflict	✓
3	/auth/register	POST	Відсутнє поле	400 Bad Request	✓
4	/auth/login	POST	Вірні дані	200 + токени	✓
5	/auth/login	POST	Невірний пароль	401 Unauthorized	✓
6	/auth/login	POST	Неіснуючий email	401 Unauthorized	✓
7	/auth/refresh	POST	Дійсний токен	200 + нові токени	✓
8	/auth/refresh	POST	Повторне використання	401 Unauthorized	✓
9	/auth/refresh	POST	Прострочений токен	401 Unauthorized	✓
10	/auth/logout	POST	Дійсний токен	200, видалено з БД	✓
11	/auth/profile	GET	З JWT	200 + профіль	✓
12	/auth/profile	GET	Без JWT	401 Unauthorized	✓
13	/auth/users/search	GET	Існуючий email	200 + користувач	✓

Таблиця А.2

Функціональне тестування Tracking Service

№	**Ендпоінт**	**Метод**	**Сценарій**	**Результат**	✓
14	/tracking/projects	GET	З JWT	200 + проєкти	✓
15	/tracking/projects	GET	Без JWT	401 Unauthorized	✓
16	/tracking/projects	POST	Нова назва	201 + проєкт	✓
17	/tracking/projects/:id	DELETE	Власний проєкт	200 + каскад	✓
18	/tracking/projects/:id	DELETE	Чужий проєкт	403 Forbidden	✓
19	/tracking/projects/:id/tasks	POST	Нова задача	201 + задача	✓
20	/tracking/tasks/:id/logs	POST	Валідні дані	201 + лог	✓
21	/tracking/tasks/:id/logs	POST	Некоректна дата	400 Bad Request	✓
22	/tracking/contacts	POST	Існуючий email	201 + контакт	✓
23	/tracking/projects/:id/members	POST	Контакт	201 + учасник	✓
24	/tracking/projects/:id/members/:id	DELETE	Видалення	`200, контакт ≠ видалено`	✓